Int. J. Inf. Secur. (2010) 9:275–286
DOI 10.1007/s10207-010-0108-z

REGULAR CONTRIBUTION

# A new probabilistic rekeying method for secure multicast groups

**Alwyn R. Pais · Shankar Joshi**

Published online: 29 May 2010

**Abstract** The Logical Key Hierarchy (LKH) is the most widely used protocol in multicast group rekeying. LKH maintains a balanced tree that provide uniform cost of $O(log\ N)$ for compromise recovery, where $N$ is group size. However, it does not distinguish the behavior of group members even though they may have different probabilities of join or leave. When members have diverse changing probabilities, the gap between LKH and the optimal rekeying algorithm will become bigger. The Probabilistic optimization of LKH (PLKH) scheme, optimized rekey cost by organizing LKH tree with user rekey characteristic. In this paper, we concentrate on further reducing the rekey cost by organizing LKH tree with respect to rekey probabilities of members using new join and leave operations. Simulation results show that our scheme performs 18 to 29% better than PLKH and 32 to 41% better than LKH.

**Keywords** LKH optimization · Key management · Secure group communication · Group Rekey

## 1 Introduction

In many multicast applications, such as pay per view, online auction, conferencing, networked gaming and news dissemination, it is necessary to secure the data from intruders as the data is confidential or it has monetary value. These applications require a message delivery from a service provider (sender) to a large number of authorized receivers who may join or leave frequently. Whenever group membership changes or when the members' keys are compromised, the sender need to update keys used for encryption. It is referred as "group rekeying".

IP Multicast, the multicast service for the Internet does not provide any security mechanisms. Indeed, anyone can join a multicast group to receive data from the data sources or send data to the group. In other words, IP multicast protocol does not support "closed" groups. Therefore, cryptographic techniques have to be employed to achieve data confidentiality.

One of the issues that has to be addressed by key management schemes for secure multicast groups is the need for forward and backward confidentiality [4]. In other words, new members joining a group should not be able to access previously multicast data, and old members should not be able to continue to access data multicast after they have left the group. One solution is to let all members in a group share a key that is used for encrypting data. To provide backward and forward confidentiality, this shared key has to be updated on every membership change and redistributed to all authorized members securely.

A simple approach for rekeying a group is one in which the group key server encrypts and sends the updated group key individually to each member. This approach is not scalable because its cost increases linearly with the group size. Thus, group rekey scalability is a challenging issue for large groups having frequent membership changes.

In recent years, many approaches for scalable group rekeying have been proposed, e.g. LKH [3,4], OFT [10], ELK [16], SDR [11] and SHKD [12]. Among these, LKH and its variants are widely used schemes. Further, many

A. R. Pais (✉)
Department of Computer Engineering, National Institute of Technology Karnataka, Surathkal, Srinivasnagar, Mangalore, 574157 Karnataka, India
e-mail: alwyn.pais@gmail.com

S. Joshi
Department of Information Science & Engineering, B. V. Bhoomaraddi College of Engineering & Technology, Vidyanagar, Hubli, 580031 Karnataka, India
e-mail: shankarjoshi48@gmail.com

optimization techniques are proposed for LKH. The schemes proposed in [5,8] optimize network bandwidth; the schemes in [6,18,20] optimize rekey cost on membership changes; finally the schemes in [1,13,14,17,19] restructure the LKH tree to optimize one or more parameters like rekey cost, bandwidth used, processing time, etc.

In this paper, we present a method for reducing rekey cost in secure multicast groups by organizing LKH tree with respect to member's rekey probabilities. We provide new insert and delete operations on LKH tree, which reduce number of rekey messages generated. The contributions of our paper are as follows:

- We present new insert and delete operations, which organize LKH tree with respect to rekey probabilities of members such that it reduces the cost of rekeying on membership changes or when members key are compromised.
- We present new key identifier assignment algorithm that generates unambiguous key identifiers.
- We provide Modified PUT (MPUT) operation, which reduce rekey cost on member join.
- We show that our method reduces the rekey cost compared to other schemes. We also show that our scheme reduces number of nodes created for a given group size.

**Organization of the paper:** The paper is organized as follows. In Sect. 2, we describe various methods available in literature on secure group communication. In Sect. 3, we describe our scheme. In Sect. 4, we present the simulation results and analysis of the results. Finally, in Sect. 5, we conclude our work.

## 2 Literature survey

There are many approaches proposed in literature for scalable group rekeying. Among them LKH, OFT, ELK, SDR and SHKD are important approaches. The most efficient methods for multicast key management are based on the Logical Key Hierarchy (LKH) scheme [3,4]. In LKH, group members are organized as leaves of a tree with logical internal nodes. The cost of a compromise recovery operation in LKH is proportional to depth of the compromised member in the LKH tree. The original LKH scheme proposes maintaining a balanced tree that gives a uniform cost of $O(log\ N)$ rekeys for compromise recovery in a $N$ member group. More details on LKH is given in Sect. 2.1.

In OFT [10] scheme, one-way function trees are used. Here, each group member maintains the unblinded key of the leaf with which he is associated and a list of blinded node keys for all of the siblings of the nodes along the path from his node to the root. This enables him to compute the unblinded keys along his path to the root, including the root key, which he also stores. If one of the blinded node keys changes and he is told the new value, then he can recompute the keys on the path and find the new group key. This scheme has complexity of $O(log\ N)$ for compromise recovery.

In the efficient large group key distribution (ELK) scheme [16] on membership change to update a key $K$, the node takes help of its left child key $K_L$ and right child key $K_R$. The new key $K^{'}$ is derived from $K$ and contributions from both children. The left child key $K_L$ contributes $n_1$ bits to the new key, which is derived by a pseudo-random function using key $K_L$ and applied to $K$. The left contribution $C_L = PRF^{<n \to n_1>}_{K^{\alpha}_{L}}(K)$ will be of $n_1$ bits long. Similarly, the right contribution $C_R$ is $n_2$ bits long and is derived from the right child key $K_R$ and $K$ as follows: $C_R = PRF^{<n \to n_2>}_{K^{\alpha}_{R}}(K)$. Then concatenate the two contributions to form a new key of length $(n_1 + n_2)$: $C_{LR} = C_L | C_R$. To compute $K'$, ELK uses pseudo-random function with $C_{LR}$ as the key and the previous key $K$ as the data: $K' = PRF_{C_{LR}}(K)$. Instead of broadcasting the key update message that has length of $(n_1 + n_2)$ bits, $K'$ legitimate members who know $K$ and either $K_L$ or $K_R$ can also recover the new key from a hint that is smaller than the key update message, by trading off computation for communication. Consider first the right-hand members who know $K$ and $K_R$. They can derive the right contribution $C_R$ of $n_2$ bits long. If they would also have a checksum, they could brute force the missing $n_1$ bits of $K'$ from the left side contribution. The hint message contains the key verification $V_{K'}$ which is derived from the new key $V_{K'} = PRF_{K'}(0)$ and has a length of $n_3$ bits. The right-hand members compute the following candidate keys. The member verifies the candidate key by checking against the key verification to see if $PRF_{\tilde{K}}(0)$ it equals $V_{K'}$.

Among the rekeying protocols proposed in the literature, the Subset Difference Rekeying (SDR) method [11] is one of the few protocols that have the property of "statelessness". In a stateless rekeying protocol, if a member has missed previous rekey operations, it need not contact the key server to obtain keys that were transmitted in the past to decode the current group key. This property makes SDR very attractive for secure multicast applications where members may go off-line frequently. The idea of subset difference is to specify a subset of valid users as the difference of two subtrees, $v_i - v_j$ where $v_i$ covers the valid users and $v_j$ covers the revoked users. To have all differences necessary to cover all valid users in a tree, a sender has to test all possible combinations of revoked users. A naive exhaustive search for this purpose takes $O(r^3)$ time, where $r$ is number of revoked users. As search time increases with revoked users, SDR method finds less use in dynamic groups.

Another example of a stateless protocol is the Self-Healing Key Delivery (SHKD) protocol proposed in [12]. In

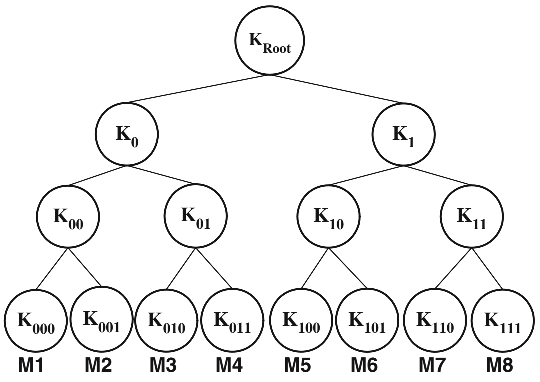

**Fig. 1** An example LKH tree with eight members

addition to statelessness, this protocol has the property called "self-healing" where a group member who has not received a group key (due to network packet loss) can recover the group key on its own without contacting the key server. This property is useful as it reduces network traffic by cutting down retransmission requests. SHKD uses polynomial-based secret sharing techniques to achieve broadcast overhead of $O(t^2m)$ key sizes, where $m$ is the number of sessions over which self-healing is possible and $t$ is the maximum allowed number of revoked nodes in the $m$ sessions. This scheme has several limitations that may discourage its deployment in some applications. First, in this scheme, an application is pre-divided into $m$ sessions, and the key server initiates a group rekeying at the beginning of each session. Thus, this scheme cannot be used for applications that demand immediate user revocation due to security requirement. Second, $t$, the maximum allowed number of revoked users during these $m$ sessions, has to be predetermined and must not be exceeded; otherwise, the security of this scheme is broken. Third, the broadcast size becomes very large even for reasonable values of $t$ and $m$.

## 2.1 LKH scheme

The basis for the LKH approach for scalable group rekeying is a logical key tree which is maintained by the key server. The root of the key tree is the group key used for encrypting data in group communications and it is shared by all users (see Fig. 1). The leaf nodes of the key tree are keys shared only between the individual users and the key server, whereas the middle level keys are auxiliary key encryption keys (KEK's) used to facilitate the distribution of the root key. Of all these keys, each user owns and only owns those on the path from its individual leaf node to the root of the key tree. As a result, when a user join/leave the group, all the keys on its path have to be changed and redistributed to maintain backward / forward data confidentiality.

In Fig. 1, member M1 holds a copy of the keys $K_{000}$, $K_{00}$, $K_0$, and $K_{Root}$; member M2 holds a copy of $K_{001}$, $K_{00}$, $K_0$ and $K_{Root}$ and so on. If keys of a member is compromised, the key server changes compromised keys and multicast new keys to the group encrypted using compromised members KEKs. For example, assume the keys of member M2 are compromised. First, key server changes $K_{001}$ and sends to member M2 over a secure unicast channel. Then, key server changes $K_{00}$ and two copies of the new key encrypted using $K_{000}$ and $K_{001}$ are sent to the group. Then, key server changes $K_0$ and sends to the group, encrypted by $K_{00}$ and $K_{01}$. Finally, $K_{Root}$ is changed by key server and sent to the group, encrypted by $K_0$ and $K_1$. From each encrypted message, the new keys are extracted by the group members who have a valid copy of encryption keys. In case of member join, all the keys in the path from joining member node to root are changed, and the new keys are multicast to the group encrypted using KEKs.

## 2.2 Shortcomings of LKH

Although LKH scheme is a secure rekeying method that provides both backward and forward confidentiality, it has some drawbacks in terms of scalability and reliability.

### *2.2.1 Individual rekeying*

At each arrival or departure of a member, the key server needs to immediately rekey the whole group in order to ensure backward and forward confidentiality, which prevents a member from accessing the data sent before its arrival or after its departure. However, individual rekeying is relatively inefficient for large groups where join/leave requests happen very frequently. For example, if members M1 and M2 (see Fig. 1) leave the group one after the other with a very short delay between the two departures, the key server will need to modify twice, the keys located at same vertices in the tree. On the contrary, if the key server had regrouped these two departures in one rekeying operation, the rekeying cost would be reduced by a half.

### *2.2.2 Key dependency*

At a new rekeying interval, the key server uses the keys of the previous interval to encrypt new keys. Because of this strong dependency between keys, when a member miss some rekeying packets during a rekeying interval, he need to contact the key server to refresh its key set otherwise he will not be able to decrypt multicast data sent after this rekeying interval even though he is still member of the group. Thus, the key server needs a reliable key distribution protocol to ensure the receipt of keys by a maximum number of members before the beginning of the next rekeying interval.

### 2.2.3 *"One Affects All" failure*

In LKH scheme, as well as in most of rekeying solutions, any arrival or departure of a recipient causes the update of the keying material of all members alike. Indeed, any rekeying operation at least requires the update of the data encryption key which is shared with all members of the group. The key server does not minimize the impact of rekeying due to the frequent dynamics of short-lived members on members who remain for longer periods during entire session.

## 2.3 Optimization schemes proposed for LKH

There are many optimization schemes proposed in literature for LKH. Some of them optimize network bandwidth, some reduce rekey cost and some restructure the LKH tree optimize one or more parameters like reduce bandwidth, rekey cost, processing time, etc.

The OFC [5] proposed a variation of LKH by employing a functional relationship among the node keys for binary key trees along the path from the leaf node representing the leaving member to the root. OFC reduces the communication overhead from LKH's $2(log_2\ N)-1$ to $(log_2\ N)$, but it is limited to the binary key tree case.

The Bezawada scheme [8] proposed a key distribution algorithm for distributing keys to only those users who need them. It proposes a compact descendant tracking scheme to track the descendants of the intermediate nodes in the multicast tree. Using this descendant information, a node forwards an encrypted key update only if it believes that there are descendants who know the encryption key. The scheme also proposes identifier assignment algorithm which assigns closer logical identifiers to users who are physically close in the multicast tree.

In the schemes [6,18], the groups are rekeyed periodically instead of on every membership change, which reduce both the processing and communication overhead at the key server. The Kronos scheme [6] is based on periodic rekeying that decouples the frequency of rekeying from the size and membership dynamics of the group. Another feature of Kronos is that it can be used in conjunction with a distributed framework for key management that uses a single group wide session key for encrypting communications between members of the group.

The Reliable Group Rekeying (RGR) scheme [18] also uses periodic batch rekeying to improve scalability and alleviate out of sync problems among rekey messages as well as between rekey and data messages. This scheme discusses a reliable multicast of rekey messages using proactive forward error correction codes (FEC).

In synchro-difference LKH (SDLKH) scheme [20], new keys are generated based on previous ones by employing the distribution of the difference. But this scheme is insecure against attack from collusion of two or more malicious adversaries [9].

The studies of Almeroth and Ammar [7] about the behavior of multicast group members in the MBone show that significant differences may exist among the members of a group. When significant differences exist among the group members, it is practical to maintain data regarding past behavior of the members. The algorithms designed to handle such difference in member's behavior can provide significant reductions in the cost of rekey operations in multicast key management. Based on this study, many LKH optimization schemes are proposed in literature, which restructure the LKH tree to improve rekey cost, bandwidth, processing time, etc.

The scheme [19] proposed two optimizations for logical key tree organizations that utilize information about the characteristics of group members to further reduce the overhead of group rekeying. First, it proposes a partitioned key tree organization that exploits the temporal patterns of group member joins and departures to reduce the overhead of rekeying. The tree is partitioned into S-partition for short duration members and L-partition for long duration members. Second, it proposes an approach under which the key tree is organized based on the loss probabilities of group members.

In the scheme [13], the key server partitions members in different categories based on their membership duration. The key server then uses error correction mechanisms with a degree of reliability that depends on the "loyalty" of each category. This scheme restructure the LKH tree, by separately regrouping members based on their membership duration aiming at preserving members with long duration membership from the impact of rekeying operations caused by arrivals or departures of short-lived members. This scheme uses a hybrid reliability scheme based on a combination of ARQ and FEC that assures a quasi certain delivery of keying material to long-lived members.

In the Refined LKH (RLKH) scheme[17], on member join, his behavior (namely active and non-active) is used to partition the member. On leave by a member, "dirty path" is set in the path from leaving node to root and rekeying is delayed until a join operation in the same path of leaving member. This scheme tries to merge leave operation rekey cost with next join operation in that sub tree. But the performance of algorithm is not adequate in all circumstances.

Most of the LKH-based schemes suggest to keep the key tree balanced so that the rekey cost is fixed to be logarithmic to the height of the key tree. However,the schemes [1] shows that organizing members in a key tree according to their topological locations would also be very beneficial, if the multicast topology is known to the key server.

Probabilistic optimization of LKH [14], called PLKH, show that it could be beneficial to use an unbalanced key tree in some cases. The idea in PLKH is to organize the key tree with respect to the compromise probabilities of members, in a spirit similar to data compression algorithms such as Huffman and Shannon-Fano coding. Basically, the key server places members who are more likely to be revoked closer to the root of the key tree. If the key server can know in advance or can make a good guess of the leaving probability of each member, then PLKH can achieve better performance than that based on a balanced one.

### 2.4 PLKH scheme

PLKH [14] scheme shows that average rekey cost of a LKH-based protocol can be minimized by organizing the LKH tree with respect to rekey likelihoods of members. Instead of maintaining uniform balanced tree, PLKH puts more dynamic members closer to the root and moves more stable members further down the tree. Here, the average rekey cost is reduced by decreasing the cost for more dynamic (i.e. more likely to rekey) members at the expense of increasing that cost for more stable members.

The tree construction requires rekey probability distributions for all current and prospective members of the group, which is not practical. Instead, PLKH uses alternative weight assignment technique. PLKH focuses on minimizing the cost of the next rekey operation. The expected costs of the next rekey operation, due to a leave or compromise event, is equal to

$$\sum_i p_i d_i \tag{1}$$

where $p_i$ is the probability that member $M_i$ will be the next to be evicted/compromised, and $d_i$ is depth of that member in the tree. This quantity $\sum_i p_i d_i$ is known as the average external path length of the tree. This problem has many similarities to the data compression problem (for details see [14]).

PLKH proposes two insert algorithms. The first algorithm, $insert_1$, organizes the LKH tree in a way which imitates the Shannon-Fano data compression trees. The $insert_1$ algorithm tries to keep the subtree probabilities as balanced as possible at every level, so that the resulting tree will have an average external path length close to the optimal bound of $-\sum_i p_i \, log \, p_i$. The second algorithm, $insert_2$, finds the best insertion point for each member by searching all possible insertion points in the tree. The $insert_2$ searches the whole tree to find the location which minimizes average external path length. But $insert_2$ adds extra computational cost of $O(N)$ on $N$-member group.

### 2.5 Shortcomings of PLKH

Although the PLKH scheme reduces the rekeying cost compared to LKH protocol, it has some drawbacks that we discuss below.

#### *2.5.1 Strict binary tree structure*

The $insert_1$ algorithm of PLKH always uses PUT operation to insert new member in the tree. When a member joins the group, PUT operation inserts that member as left child of new internal node in the tree. The PUT operation ensures that all members are inserted as leaf nodes in the tree and maintains strict binary tree structure. This increases the depth of newly inserted member in the tree which in turn increases rekey cost. The $insert_2$ algorithm finds the best insertion point for member $M$ by searching all possible insertion points in the tree. But it reduces number of rekeys at the expense of additional computational cost of $O(N)$.

#### *2.5.2 Probability considered*

In PLKH trees, each node $X$ in the tree has a probability field $X.p$ that shows the cumulative probability of the members in the subtree rooted at $X$, similar to that in Huffman trees. The probability of member $X$ is equal to the rekey probability of the corresponding member if $X$ is a leaf node, and it is equal to $X.left.p + X.right.p$ if $X$ is an internal node.

The insert algorithms of PLKH use this cumulative probability for deciding where to insert new member in tree. The problem with cumulative probability is that it changes on every membership change done in the subtree. The new member is inserted based on cumulative probabilities of internal nodes even though he may have higher rekey probability than some of the members in the tree. This causes new member to be pushed further down the tree, increasing the rekey cost for that member.

#### *2.5.3 Ambiguous key identifiers*

PLKH does not discuss how the key identifiers are assigned. The key identifier generated using original LKH identifier scheme or using special flag indicator for node created in PUT operation [14] will end up in having some nodes with ambiguous key identifier. This problem arises as PLKH does not affect existing nodes on membership change; so when a change occurs, only nodes in the path from member node to root are updated. But this information is not updated in descendants of affected member's node.

## 3 Our method

In this paper, we focus on minimizing the average rekey cost of LKH-based protocols by organizing the LKH tree with respect to the rekey probabilities of members. The average rekey cost can be reduced by decreasing the cost for more dynamic (i.e. more likely to rekey) members at the expense of increasing that cost for more stable members. The communication and computation costs of these rekey operations are proportional to depth of member in tree.

The rekeys happen whenever change occurs in group membership or when keys of group members are compromised. The rekey probability of a member $M$ gives measure of how quick that member will cause rekey due to compromise or eviction. Our key server calculates and stores rekey probabilities of all members of the group.

We organize the LKH tree with respect to members rekey probabilities as opposed to cumulative probability of PLKH. We concentrate on reducing number of rekeys that are caused due to member compromise or eviction. Also, to minimize rekey cost on insert, the keys each member is holding after an *insert* operation should be the same as those it was holding before *insert* operation (or the corresponding new keys, for the keys that are changed), plus possibly some newly added keys to the tree.

The optimal solution to get such probability-based LKH tree is given by Huffman trees [2]. But the Huffman tree requires changes in the locations of the existing members in the tree, which means extra rekey operations. Shannon-Fano trees [2] are another solution which avoid changes in location, but are slightly suboptimal. We use Shannon-Fano trees to preserve the position of members in the tree when group membership changes.

Instead of placing the members as leaf nodes as in PLKH, we provide new *insert* operation which place the members either as leaf node or as internal node in LKH tree based on their probabilities. When a new member $M$ joins the group, we place member $M$ in a position such that all ancestors of $M$ will have higher probability and all descendents of $M$ will have lesser probability. Our *insert* operation imitates Shannon-Fano trees to avoid changes in positions of members and also to balance the probabilities of left and right subtrees of each member. Our *delete* operation removes unwanted nodes in tree based on some conditions.

In the following sections, we present different node types used in our scheme along with new *insert* and *delete* operations. Also, we present our Modified PUT operation and new key identifier assignment scheme, which generates unambiguous key identifiers for nodes in the tree.

## 3.1 Node types

In this section, we define three types of nodes used in our probability-based LKH tree namely physical, logical and replaceable. A physical node represents a group member. A logical node represents an internal node created by our Modified PUT (called MPUT) operation. A replaceable node represents internal node that may be replaced as physical node on future join with a suitable member details.

When a member joins the group, a new physical node is created to represent that member. A logical node is created by MPUT operation to preserve relative locations of present members in the tree. When a member leaves the group, we delete that node only when it is a leaf node. Otherwise, we set it to replaceable node as it may have one or more dependent nodes and deleting that node will cause more rekeys. The replaceable node will be changed to physical node when a suitable member who fits the probability requirement joins the group in future. This reduces rekey cost of a future join operation by suitable member.

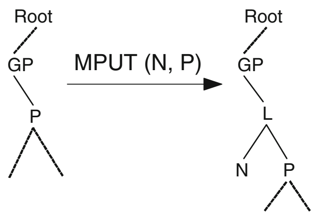

**Fig. 2** Our MPUT operation. It creates new logical node $L$ with $P\&N$ as its children

## 3.2 Modified PUT operation

When a member joins the group, we find suitable location where he can be placed in the tree such that dynamic members will be closer to root and more stable members further down the tree. Our Modified PUT (called MPUT) operation is used to insert new member in a position which preserves relative locations of existing members in the tree. It is similar to PUT operation of PLKH but creates a logical node.

To insert new physical node $N$ into tree using MPUT, new logical node $L$ is created at certain location in tree such that $N$ will be its left child and P will be its right child (see Fig. 2). The $GP$, the previous parent of node $P$, will now point to node $L$.

Every node will have two probability fields, namely initial probability denoted as $initprob$ and aggregate probability denoted as $aggrprob$. The initial probability is same as rekey probability of that member if the node is of type physical or replaceable and for a logical node it is

$$max(leftChild.initprob, rightChild.initprob) + 0.001.$$

This ensures probability of logical node is more than all members in the subtree and the probability tree structure is preserved. The aggregate probability of a node $N$ is the sum of initial probabilities of all nodes in subtree rooted at $N$.

## 3.3 Insert operation

When a new member joins the group, a new physical node is created and is inserted in a position such that its ancestors will have bigger probability and its descendent (if any) will have lesser probability. To insert a new node

```
insert(N, GP , P, direction)

if(P!=NULL){
    if( P.initprob < N.initprob ){
        if( P.type =replaceable)
            replaceNode (P, N)
        else
            MPUT(N, P)
    }
    if( P.left.aggrprob > P.right.aggrprob)
        insert(N, P, P.right, right)
    else
        insert(N, P, P.left, left)
}
else{
    if(direction = left)
        insertNodeToLeft (GP, N)
    else
        insertNodeToRight (GP, N)
}
```

**Fig. 3** Our insert algorithm which considers the node's initial probability

$N$ in suitable position with $GP$ as grandparent and $P$ as parent nodes, our insert algorithm uses one of the following operations $MPUT$, $insertNodeToLeft$, $insertNodeToRight$ and $replaceNode$ (see Fig. 3). Among these, four operations only $MPUT$ creates strict binary tree structure.

A suitable position for new node $N$ is found when we get a parent node $P$ with initial probability less than that of $N$. If $P$ is replaceable node then we replace $P$ as physical node with details of $N$ by calling $replaceNode$. Otherwise, we call $MPUT$ that creates new logical node and puts $N$ as left child of new logical node. If $P$'s initial probability is more than $N$'s, then we use aggregate probability of $P$'s children. We try to balance the probabilities on left and right subtrees by moving $N$ to either left or right subtree which has lesser aggregate probability. The $insertNodeToLeft$ inserts new node as left child of given parent node and $insertNodeToRight$ inserts new node as right child of given parent node. Our insert algorithm takes no additional computational cost and has complexity of $O(log\ d)$, where $d$ is depth of inserted member in the tree.

## 3.4 Delete operation

When a member leaves the group, his corresponding physical node need to be deleted from the tree. The physical node may be a leaf node or an internal node based on how it inserted and whether it has any dependent nodes at present. Our delete operation removes a physical node only if it is a leaf node; otherwise, we set its node type as replaceable and refresh affected keys (see Fig. 4). Our delete operation also removes unwanted logical and replaceable nodes present in affected subtree provided they do not have any dependent physical nodes.

```
deleteNode(N)

P=getParent(N)
direction=getChildDirection(P, N)
if(isLeafNode(N)){
    if(direction=Left)
        deleteNodeFromLeft(P)
    else
        deleteNodeFromRight(P)
}
else{
    if(N.type=physical)
        setNodeAsReplacable(N)
}
if(isLeafNode(P) AND P.type  != physical)
    deleteNode(P)
```

**Fig. 4** Our delete operation which removes unwanted nodes from tree

```
assignKeyID(P,C, direction)

keyid=P.keyid
if(direction=Left)
   strcat keyid, "L"
 else
   strcat keyid, "R"
 if C.type=logical
   strcat keyid, "x"

 id=getFreeSlot(P,C.type)
 strcat keyid, id
 C.keyid=keyid
```

**Fig. 5** Our key identifier assignment algorithm

## 3.5 Key identifier assignment

When a new node $C$ is inserted at some position with $P$ being its parent node, our assignment scheme will generate new key identifier for $C$ as shown in Fig. 5. First, we get parent's key identifier and assign direction of $C$ from $P$ to it. The 'L' and 'R' indicate the child direction with respect to its parent. If $C$ is logical node then we attach 'x' flag to the key identifier. The 'x' flag helps in direction correction to trace descendants. Finally, we attach a child number that we get from first ancestor which is not a logical node starting from $C$. Here, $getFreeSlot$ recursively searches until a non-logical parent is found. It then returns index to a free child slot.

## 3.6 Tracing node using key identifier

An example is given in Fig. 6 to understand working of our scheme. After $MPUT$ operation if we need to trace node $P$ with key identifier $R3R2$, we first move up to new logical node $L$ using direction indicators 'L' & 'R' representing left and right directions. At node $L$, we detect that it is logical. Now, we do direction correction. At any stage when we get a logical node, if the remaining part of node to be traced does not have 'x' flag indicator then we move in right direction irrespective of 'L' or 'R' of remaining part. This is because

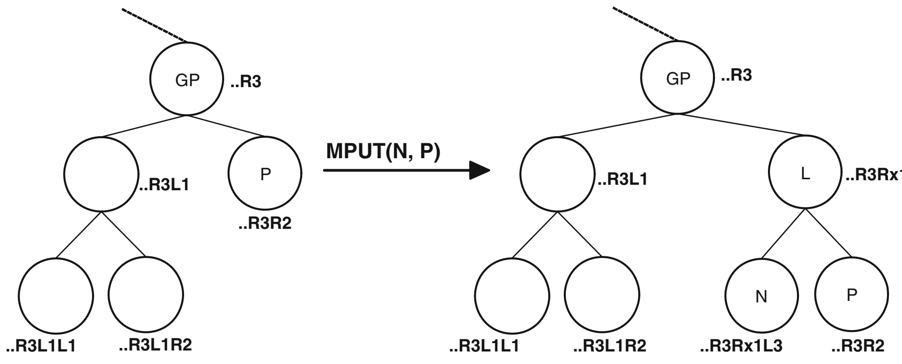

**Fig. 6** Example shows the working of our key identifier assignment algorithm

the $MPUT$ always inserts new node $N$ to left of logical node and parent to right. The left subtree will have nodes with 'x' prefix in their remaining part. Since the $P$ node's remaining part $R2$ does not have 'x' flag indicator, so we move in right direction to find node at next level. Note that on $MPUT(N, P)$ operation, keys of all the nodes from $GP$ to root are refreshed. This scheme takes $d$ steps for locating any member, where $d$ is node's depth.

### 3.7 Choosing probability

To use the insertion algorithm as described earlier, it is crucial to know the rekey probability values of all group members at the time of insertion. The optimal solution to this problem is not practical since that would require the knowledge of rekey probability distributions for all current and prospective members of the group as well as the cost calculations for every possible sequence of future join, leave, and compromise events.

The PLKH [14] has proposed a solution to this problem using weight-based scheme. We are also using the same scheme to calculate rekey probabilities of members. The key server calculates and maintains rekey probability of each member. Whenever a member is evicted or compromised our key server updates his rekey probability.

The rekey probability in weight assignment is calculated using the inverse of the mean inter-rekey time of members as follows:

$$p_i = w_i / W \tag{2}$$

where $p_i$ is rekey probability of member $M_i$, $w_i = 1/\mu_i$ and $W = \sum_i w_i$. The $\mu_i$ is the average time between two rekeys by member $M_i$. The reasons for choice of $1/\mu_i$ as the weight measure among many other candidates are:

1. Its simplicity and convenience.
2. In the special case where the members inter-rekey time distributions are exponential, $p_i = w_i / W$ gives exactly the probability that $M_i$ will be the next member to rekey.

## 4 Simulation results and analysis

We have chosen LKH and PLKH for comparison with our scheme as they address rekey cost reduction on group membership changes and on key compromises by the members, whereas other schemes concentrate on reducing bandwidth and computational cost. We simulated LKH, PLKH and our scheme using *ns2* network simulator [15]. We call our scheme as OPLKH for simplicity. We performed experiments on randomly generated network topologies for groups of 128, 256, 512, 768 and 1,024 members. For each experiment, we selected a random set of members to join and leave the group and recorded number of rekey messages generated. We considered three scenarios namely static, semi dynamic and dynamic group.

### 4.1 Scenario 1-static group

In this scenario, we assign relatively lesser rekey probability to members. Most of the members in this scenario stay in group till session is over. The members are added at random intervals. In this scenario, number of members who leave the group is chosen to be roughly 25% of the group size. The members leave the group at random times (chosen based on their rekey probability) during the entire session of the group. From the results obtained, we observe that for static groups our scheme performs 41% better compared to LKH and 29% better compared to PLKH (see Fig. 7).

### 4.2 Scenario 2-semi dynamic group

In this scenario, we assign relatively higher probability to members compared to static group. The members are added at random intervals. In this scenario, number of members to leave the group is chosen to be roughly 50% of group size. The member's leave time is selected at random based on their probabilities. From the results obtained, we observe that for semi dynamic groups our scheme performs 38% better compared to LKH and 21% better compared to PLKH (see Fig. 8).

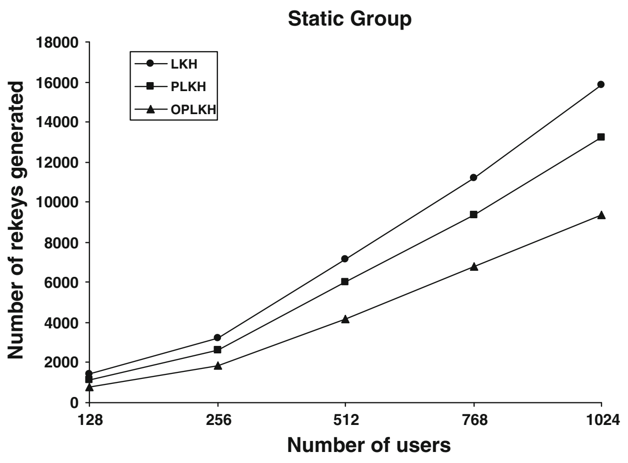

**Fig. 7** Simulation results for static group with various group sizes

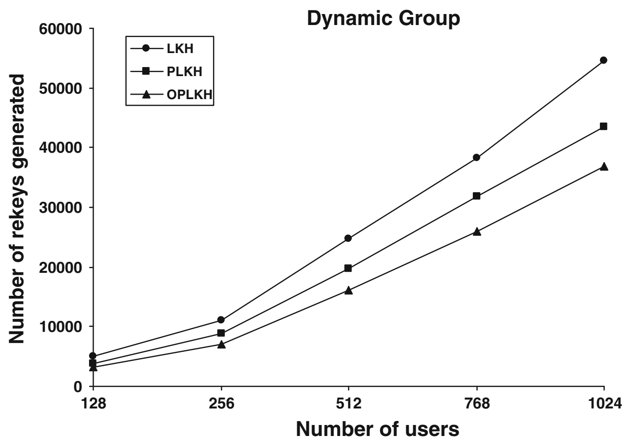

**Fig. 9** Simulation results for dynamic group with various group sizes

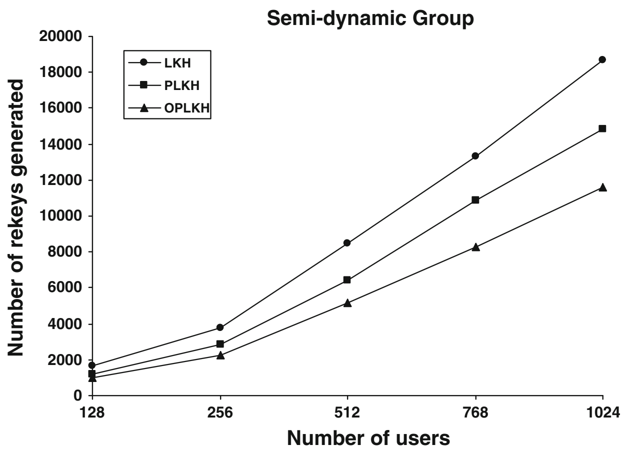

**Fig. 8** Simulation results for semi dynamic group with various group sizes

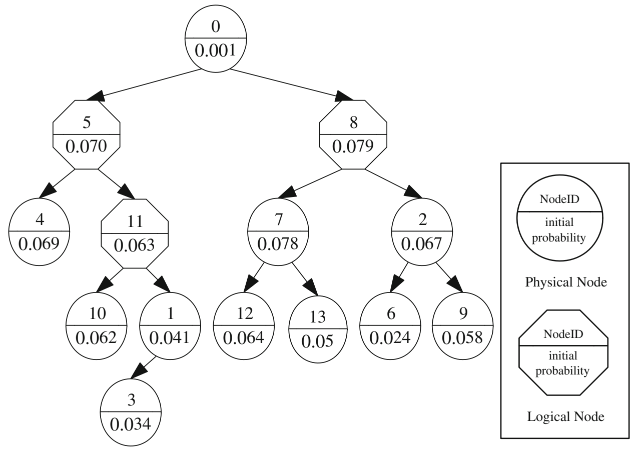

**Fig. 12** OPLKH Tree formed when members arrive in given order

## 4.3 Scenario 3-dynamic group

In this scenario, we assign high probability to members. The members are allowed to join and leave the group at rapid rate. The group experiences lot of join/leave requests in quick time, which increases the rekey messages generated. From the results obtained, we observe that for dynamic groups our scheme performs 32% better compared to LKH and 18% better compared to PLKH (see Fig. 9).

## 4.4 Nodes created

LKH maintains uniform balanced tree that increases number of nodes created in tree. On every join by members, the LKH increases logical node(s) to ensure that members are at leaf. Figure 10 shows the tree structure formed when members with probabilities 0.001, 0.041, 0.067, 0.034, 0.069, 0.024, 0.078, 0.058, 0.062, 0.064, 0.005 are joined in order. The LKH created 23 nodes with 11 physical nodes and 12 logical nodes. Note here that LKH does not consider probabilities of members. The probability shown in figure is for understanding purpose only to indicate where each member node is inserted.

PLKH creates intermediate node on every PUT operation. The PLKH always inserts members using PUT operation. Figure 11 shows the tree structure formed when members with probabilities 0.001, 0.041, 0.067, 0.034, 0.069, 0.024, 0.078, 0.058, 0.062, 0.064, 0.005 are joined in order. PLKH created 19 nodes. Note that PLKH does not distinguish between physical and logical nodes.

OPLKH creates logical node only when MPUT is called. In OPLKH, the replaceable nodes are replaced when suitable member joins in future. It helps in reducing number of nodes created. Figure 12 shows the tree structure formed when members with probabilities 0.001, 0.041, 0.067, 0.034, 0.069, 0.024, 0.078, 0.058, 0.062, 0.064, 0.005 are joined in order. OPLKH created 14 nodes with 3 logical nodes, 11 physical nodes and 0 replaceable nodes.

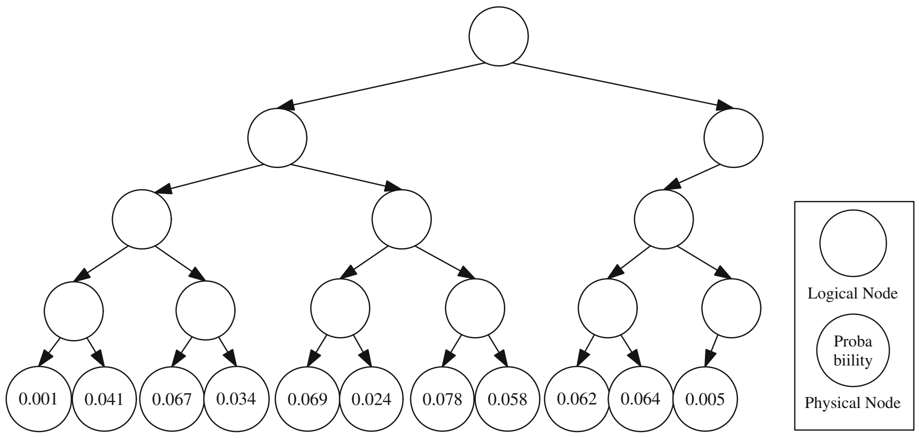

**Fig. 10** LKH Tree formed when members arrive in given order

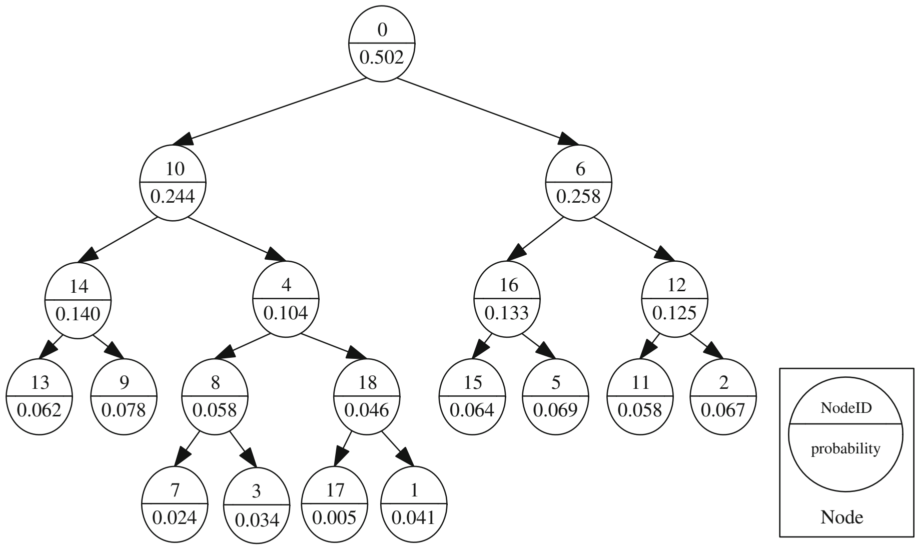

**Fig. 11** PLKH Tree formed when members arrive in given order

## 4.5 Nodes remaining at the end of session

From the results obtained, it is clear that OPLKH has less number of nodes at the end of session. This is because OPLKH removes unwanted nodes (i.e. non-physical nodes which have no dependents) from tree. This reduces total nodes in tree. Whereas PLKH removes internal nodes only if there are no dependent members and LKH removes logical nodes only if they are leaf nodes. Figure 13 shows the comparison of three algorithms with respect to number of nodes remaining at the end of session.

From results obtained, it is clear that OPLKH has 23% to 36% less nodes than LKH (for more details see Table 1). We can see that OPLKH has reduced number of nodes remaining by removing unwanted nodes from tree.

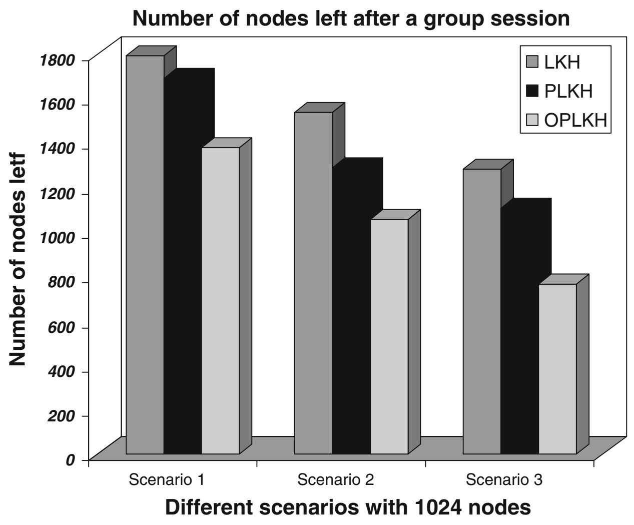

**Fig. 13** Simulation results for nodes left at the end of session

## 4.6 Analysis

The number of rekey messages generated for same member join/leave operations in all schemes show that our scheme, OPLKH, performs better than both LKH and PLKH. OPLKH scheme not only optimizes the rekey cost, it also optimizes number of nodes created in the tree. OPLKH avoids strict binary tree structure, unwanted PUT operations

**Table 1** Reduction in nodes remaining with various scenarios in a 1,024 node group

| | Over LKH (in %) | Over PLKH (in %) |
|---|---|---|
| Scenario 1 | 23 | 17 |
| Scenario 2 | 29 | 20 |
| Scenario 3 | 36 | 25 |

**Table 2** Reduction in rekey cost of OPLKH compared to LKH and PLKH

| | Over LKH (in %) | Over PLKH (in %) |
|---|---|---|
| Scenario 1 | 41 | 29 |
| Scenario 2 | 38 | 21 |
| Scenario 3 | 32 | 18 |

and unnecessary logical nodes. Table 2 shows rekey cost improvement obtained by OPLKH over LKH & PLKH in different scenarios. Table 1 shows improvement obtained on nodes remaining at end of session in the tree by OPLKH over LKH & PLKH in different scenarios.

### 4.7 Limitations of our scheme

Some of the limitations of our scheme are discussed here. First, our key identifier assignment requires more memory to store key identifiers. Typical LKH scheme needs only 1 bit for choosing left or right child. Whereas our scheme needs 6 bits with 1 bit for direction, 1 bit for ‘x’ flag and 4 bits for index to free child slot in parent. Second limitation is that, though total nodes created are less than PLKH & LKH schemes, our scheme treats some nodes harshly in terms of depth assigned. Thirdly, our scheme only ensures that tree structure is binary. It neither tries to maintain strict binary tree as PLKH nor tries to balance all nodes at same level as LKH. Finally, our scheme does not optimize rekey cost when one or more members are compromised in same subtree.

## 5 Conclusion

In this paper, we addressed the issue of reducing rekey messages generated on member leave in secure multicast groups. We presented new method to form the LKH tree with member’s rekey characteristics using our insert and delete operations. Also, we gave generic key identifier assignment scheme which avoids ambiguous key identifiers. The simulation results show that our scheme achieves rekey reduction of 18% compared to PLKH and 32% on original LKH for dynamic group of 1024 members. Our scheme also reduces number of nodes created for given size and number of nodes left after a group session by eliminating unwanted non-physical leaf nodes.

## References

1. Bhattacharjee, B., Banerjee, S.: Scalable secure group communication over ip multicast. International Conference on Network Protocols (ICNP) 2001 (2001)
2. Cleary, J.G., Bell, T.C., Witten, I.H.: Text compression. Prentice-Hall, New Jersey (1990)
3. Gouda, M., Wong, C.K., Lam, S.S.: Secure group communications using key graphs. IEEE/ACM Trans. Networking **8**, 16–30 (2000)
4. Harder E.J., Wallner, D.M., Agee, R.C.: Key management for multicast: Issues and architectures. RFC 2627 (1999)
5. Itkis, G., Micciancio, D., Naor, M., Canetti, R., Garay, J., Pinkas, B.: Multicast security: a taxonomy and some efficient constructions. Proc. IEEE INFOCOM ’99 **2**, 708–716 (1999)
6. Jajodia, S., Setia, S., Koussih, S.: Kronos: A scalable group re-keying approach for secure multicast. IEEE Symposium on Security and Privacy, Oakland, CA (2000)
7. Judge, P., Ammar, M.: Security issues and solutions in multicast content distribution: a survey. IEEE Network **17**, 30–36 (2003)
8. Kulkarni, S.S., Bezawada, B.: Distributing key updates in secure dynamic groups. International Conference on Distributed Computing and Internet Technology, ICDCIT-04 (2004)
9. Lee, Y., Park, Y., Kim, H., Chung, B., Yoon, H.: Weakness of the synchro-difference lkh scheme for secure multicast. IEEE Commun. Lett. **11**(9), 765–767 (2007)
10. McGrew, D.A., Sherman, A.T.: Key establishment in large dynamic groups using one-way function trees. IEEE Trans. Software Eng. **29**(5), 444–458 (2003)
11. Naor, M., Naor, D., Lotspiech, J.: Revocation and tracing schemes for stateless receivers. Advances in Cryptology-CRYPTO 2001, Springer-Verlag Inc. LNCS 2139, pp. 41–62 (2001)
12. Ning, P., Liu, D., Sun, K.: Efficient self-healing group key distribution with revocation capability. In: Proceedings of the 10th ACM conference on Computer and Communications Security (2003)
13. Onen, M., Molva, R.: Group rekeying with a customer perspective. In: Proceedings of Tenth International Conference on Parallel and Distributed Systems (2004)
14. Selçuk A.A., Sidhu, D.P.: Probabilistic methods in multicast key management. In: Proceedings of the Third International Workshop on Information Security (2000)
15. The Network Simulator.: ns-2, vesion 2.32. http://www.isi.edu/nsnam/ns/ (2008)
16. Tygar, D., Perrig, A., Song, D.: Elk, a new protocol for efficient large group key distribution. In: Proceedings of the IEEE Security and Privacy Symposim 2001 (2001)
17. Xu, Y., Sun, Y.: A new group rekeying method in secure multicast. International conference on Computational intelligence and security (2005)
18. Zhang X., Yang, Y., Li, X., Lam, S.: Reliable group rekeying: Design and performance analysis. In: Proceedings of ACM SIGCOMM2001 (2001)
19. Zhu, S.S.S., Jajodia, S.: Performance optimizations for group key management schemes for secure multicast. In: Proceedings of 23rd International Conference on Distributed Computing Systems (2003)
20. Zhu, W.T.: Optimizing the tree structure in secure multicast key management. IEEE Commun. Lett. **9**(5), 477–479 (2005)

## Author Biographies

**Alwyn R. Pais** is Assistant Professor in Department of Computer Engineering, NITK Surathkal. He completed his B.Tech.(CSE) from Mangalore University, India and M.Tech. (CSE) from IIT Bombay, India. His area of interest include Information Security, Image Processing and Computer Vision.

**Shankar Joshi** is Senior Lecturer in Department of Information Science & Engineering, B. V. Bhoomaraddi College of Engg. & Tech., Hubli. He completed B.E.(CSE) from Visvesvaraya Technological University, India and M.Tech.(CSE) from NITK Surathkal, India. His areas of interest include Information Security, Distributed Systems and Computer Networks.