# Probabilistic Methods in Multicast Key Management*

Ali Aydın Selçuk and Deepinder Sidhu

Maryland Center for Telecommunications Research
Department of Computer Science and Electrical Engineering
University of Maryland Baltimore County
Baltimore, MD, 21250, USA
{aselcu1,sidhu}@umbc.edu

**Abstract.** The Logical Key Hierarchy (LKH) scheme and its derivatives are among the most efficient protocols for multicast key management. Traditionally, the key distribution tree in an LKH-based protocol is organized as a balanced binary tree, which gives a uniform $O(\log n)$ complexity for compromise recovery for an $n$-member group. In this paper, we study improving the performance of LKH-based key distribution protocols by organizing the LKH tree with respect to the members' rekeying probabilities instead of keeping a uniform balanced tree. We propose two algorithms which combine ideas from data compression with the special requirements of multicast key management. Simulation results show that these algorithms can reduce the cost of multicast key management significantly, depending on the variation of rekey characteristics among group members.

## 1 Introduction

One of the biggest challenges in multicast security is to maintain a group key that is shared by all the group members and nobody else. The group key is used to provide secrecy and integrity protection for the group communication. The challenge of maintaining such a group key becomes greater when the groups are large and highly dynamic in terms of membership.

Currently, the most efficient methods for multicast key management are based on the Logical Key Hierarchy (LKH) scheme of Wallner et al. [18] (also independently discovered by Wong et al. [19]). 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 the depth of the compromised member in the LKH tree. The original LKH scheme proposes maintaining a balanced

* This research was supported in part by the Department of Defense at the Maryland Center for Telecommunications Research, University of Maryland Baltimore County. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Department of Defense or the U.S. Government

tree, which gives a uniform cost of $O(\log n)$ rekeys for compromise recovery in an $n$-member group.

In this paper, we study improving the performance of LKH-based key distribution protocols by organizing the LKH tree with respect to the members' rekeying probabilities instead of keeping a uniform balanced tree. This problem was first pointed out by Poovendran and Baras in Crypto'99 [15]; but no solutions have been proposed to date. We propose two algorithms which combine ideas from data compression with the special requirements of multicast key management. Simulation results show that these algorithms can reduce the cost of multicast key management significantly, depending on the variation of rekey characteristics among group members.

We first start by an analysis of the probabilistic LKH optimization problem (Sections 2, 3). Then we describe two algorithms for this problem and discuss their design rationale (Sections 4, 5, 6). We summarize the simulation results in Section 7. In Section 8, we conclude with a discussion of the issues regarding an effective utilization of these algorithms.

### 1.1 Related Results

Group key establishment protocols can be classified as (contributory) group key agreement protocols and (centralized) group key distribution protocols. Most group key agreement protocols are multi-party generalizations of the two-party Diffie-Hellman key agreement protocol [5, 17, 16]. They have the advantage of doing without an active key management authority; but they also require quite intensive computation power, proportional to the size of the group. Therefore, group key agreement protocols are mostly used for relatively smaller groups (i.e. with 100 members or less).

In Internet multicasting, groups are typically large, and there is an active group manager available. Therefore, most multicast key management protocols are based on centralized key distribution protocols. In the Group Key Management Protocol (GKMP) of Harney et al. [9], each group member obtains the group key by a unicast communication with the group key manager. This protocol has the disadvantage of having to re-initialize the whole group when a member is compromised (possibly due to a departure). A similar but more scalable protocol is the Scalable Multicast Key Distribution (SMKD) protocol proposed by Ballardie [3]. In this protocol, the key manager delegates the key management authority to routers in the Core-Based Tree (CBT) multicast routing. The protocol has the disadvantage of requiring trusted routers and being specific to the CBT routing protocol. The Iolus protocol [14] deals with the scalability problem by dividing the multicast group into subgroups. Each subgroup has its own subgroup key and key manager, and rekeying problems are localized to the subgroups. The multicast group is organized as a tree of these subgroups, and translators between neighbor subgroups help a multicast message propagate through the tree. A similar approach is a group key management framework proposed by Hardjono et al. [8], where the group members are divided into "leaf regions" and the managers of leaf regions are organized in a "trunk region". The

key management problem is localized to the regions, and inter-region communication is maintained by "key translators". This framework provides a scalable solution for key management in large multicast groups.

Currently, the most efficient multicast key distribution protocols which enable all group members to share a common key not known to anyone outside the group are based on the Logical Key Hierarchy (LKH) protocol and its variants. LKH-based protocols, as will be discussed in more detail in Section 2, have the ability to rekey the whole group with $O(\log n)$ multicast messages when a member is compromised. The LKH structure was independently discovered by Wallner et al. [18] and Wong et al. [19]. Modifications to the basic scheme which improve the message complexity by a factor of two with a relatively small computational overhead have been proposed in [13, 6]. Certain similarities between LKH trees and some information-theoretic concepts have been pointed out in [15].

Another different class of group key distribution protocols is the Broadcast Encryption protocols [7]. These protocols guarantee the secrecy of the key against coalitions of up to a specified number of outsiders. Luby and Staddon [12] prove a lower bound for the storage and transmission costs for the protocols in this class, which is prohibitively large for most cases.

### 1.2 Notation

The following notation is used throughout this paper:

| | |
|---|---|
| $n$ | number of members in the group |
| $M_i$ | $i$th member of the group |
| $d_i$ | depth of $M_i$ in the LKH tree |
| $p_i$ | probability of $M_i$ being the next member to cause a rekey (due to a departure, compromise, etc.) |

All logarithms are to the base 2, and $i$ in summations $\sum_i$ ... ranges from 1 to $n$, unless otherwise is stated.

## 2 The LKH Scheme

The LKH scheme organizes the members of a multicast group as leaves of a key distribution tree where the internal (non-leaf) nodes are "logical" entities which do not correspond to any real-life entities in the multicast group and are only used for key distribution purposes. There is a key associated with each node in the tree, and each member holds a copy of every key on the path from its corresponding leaf node to the root of the tree. Hence, the key corresponding to the root node is shared by all members and serves as the group key. An instance of an LKH tree is shown in Figure 1.

In this figure, member $M_1$ holds a copy of the keys $K_{000}$, $K_{00}$, $K_0$, and $K_{Root}$; member $M_2$ holds a copy of $K_{001}$, $K_{00}$, $K_0$, and $K_{Root}$; and so on. In case of a compromise, the compromised keys are changed, and the new keys are multicast to the group encrypted by their children keys. For example, assume the keys of $M_2$ are compromised. First $K_{001}$ is changed and sent to $M_2$ over a secure

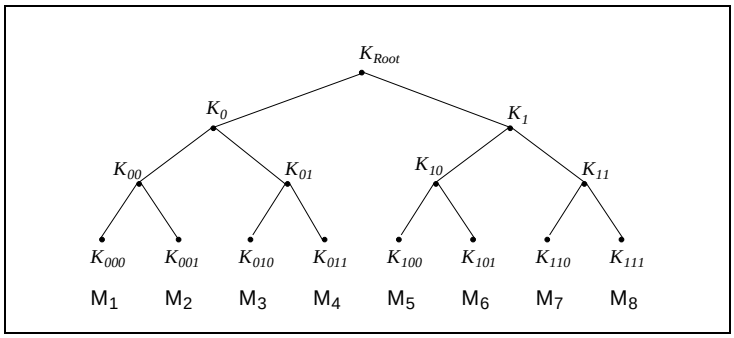


**Fig. 1.** An example LKH tree with eight members. Each member holds the keys on the path from its leaf node to the root. $K_{Root}$ is the key shared by all group members.

unicast channel. Then $K_{00}$ is changed; two copies of the new key are encrypted by $K_{000}$ and $K_{001}$ and sent to the group. Then $K_0$ is changed and sent to the group, encrypted by $K_{00}$ and $K_{01}$; and finally $K_{Root}$ is changed 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 either one of the (child) encryption keys.

If the security policy requires backward and forward secrecy for group communication (i.e. a new member should not be able to decrypt the communication that took place before its joining, and a former member should not be able to decrypt the communication that takes place after its leaving) then the keys on the leaving/joining member's path in the tree should be changed in a way similar to that described above for compromise recovery.

Although an LKH tree can be of an arbitrary degree, most efficient and practical protocols are obtained by binary trees, and studies in the field have mostly concentrated on binary trees [18, 13, 6]. We follow the convention and assume the LKH trees are binary. We also assume that the binary tree is always kept full (i.e. after deletion of a node, any node left with a single child is also removed).

## 3 Probabilistic LKH Optimization

The problem addressed in this paper is how to minimize the average rekey cost of an LKH-based protocol by organizing the LKH tree with respect to the rekey likelihoods of the members. Instead of keeping a uniform balanced tree, 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. This can be achieved by putting the more dynamic members closer to the root and moving more stable members further down the tree.

The rekey operations caused by a periodic key update or a joining member can be realized by a single fixed-size multicast message using a one-way function [13]. In this study, we will concentrate on the more costly rekey operations

that are caused by a member compromise or eviction event. The communication and computation costs of these rekey operations are linearly proportional to the depth $d_i$ of the compromised (or, evicted) member, as $ad_i + b$, $a > 0$. The exact values of $a$ and $b$ depend on the specifics of the LKH implementation.

Finding the optimal solution to this problem that will minimize the average cost of all future rekey operations is not possible in practice 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. Instead, we concentrate on a more tractable optimization problem, that is to minimize the cost of the *next* rekey operation. The expected cost 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 its depth in the tree. This problem has many similarities to the data compression problem with code trees where the average code length per message is $\sum_i p_i d_i$, This quantity $\sum_i p_i d_i$ is known as the *average external path length* of the tree, where $p_i$ is the probability of message $m_i$ to be the next to appear, and $d_i$ is its depth in the code tree. The optimal solution for the problem of minimizing the average external path length is given by Huffman trees [4]. Shannon-Fano trees are another alternative solution which give very good compression in practice but are slightly sub-optimal [4].

### 3.1 Differences from the Data Compression Problem

In an LKH key distribution tree, a change in the tree structure, such as changing the location of an existing member in the tree, causes extra rekey operations, which adversely affects the objective function (i.e. minimizing the average number of rekeys). On the other hand, in a data compression tree, a structural change does not directly induce an overhead on the objective function (i.e. minimizing the average code length). So, changes in the tree structure can be freely utilized in data compression algorithms, such as dynamic Huffman algorithms [10], to maintain the optimal tree structure; whereas they cannot be utilized so freely in dynamic LKH algorithms. Therefore, an LKH scheme with sub-optimal $\sum_i p_i d_i$ can have a better overall performance than one that keeps $\sum_i p_i d_i$ minimal all the time.

Another difference of LKH trees from data compression trees is that, if member evictions are the main reason for rekey operations (i.e. if very few compromise events happen other than member evictions), then each member in the tree will cause a single rekey operation while it is in the tree.

## 4 Design Rationale

As discussed above, finding the optimal solution that minimizes the average number of rekey messages over all future sequences of join, leave, and compromise events is not possible in practice. Therefore, we focus our attention on minimizing the expected cost of the next rekey event, $\sum_i p_i d_i$. The proven optimal solution for minimizing $\sum_i p_i d_i$ is given by a Huffman tree. However, maintaining a Huffman tree requires changes in the locations of the existing members in the tree, which means extra rekey operations. We choose to avoid this kind of extra rekey operations and concentrate on algorithms which do not require changing the location of the existing members.

Given the condition that the locations of existing members will not be changed in the tree, the main structural decision for the tree organization is where to put a new member at insertion time. Also, the insertion operation should observe the current locations of existing members. That is, the keys each member is holding after an insertion operation should be the same as those it was holding before the insertion (or the corresponding new keys, for the keys that are changed), plus possibly some newly added keys to the tree. Therefore, we will focus on insertion operations of the form illustrated in Figure 2, which preserve the relative location of present members.

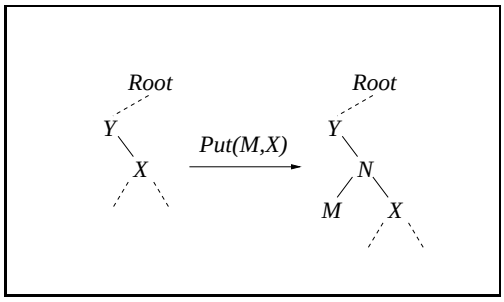


**Fig. 2.** The *Put* procedure. Relative location of existing nodes are kept the same to avoid extra rekey operations.

That is, to insert a new member $M$ into the group, a new internal node $N$ is inserted at a certain location in the tree, and $M$ is linked underneath. To denote this insertion operation at a given location $X$ for a given new member $M$, we will write $Put(M, X)$. Note that the traditional LKH insertion, where every new member is inserted as a sibling to a leaf node, is a specific case of $Put(M, X)$ where $X$ is a leaf node.

In our probabilistic LKH 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 (i.e., $X.p$ is equal to the 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 $Put$ procedure shown above also updates the $p$ field of all nodes affected by the insertion as well as setting up the appropriate links for $M$ and $N$.

## 5 Insertion Algorithms

In this section, we describe two LKH insertion algorithms which seek to minimize the expected number of rekeys for the next member eviction or compromise event. The first algorithm does not induce any additional computational cost over the basic balanced-tree LKH insertion. The second algorithm provides further improvement over the first algorithm in message complexity but induces an $O(n)$ computational overhead for an $n$-member group.

**Algorithm 1:** The first algorithm, $Insert_1$, organizes the LKH tree in a way which imitates the Shannon-Fano data compression trees. In Shannon-Fano coding [4], a tree is constructed from a given set of probabilities by dividing the set into two parts of (roughly) equal probability repeatedly until every set includes a single element. Shannon-Fano coding guarantees a maximum redundancy of 1; i.e. $\sum_i p_i d_i \leq -\sum_i p_i \log p_i + 1$, for $\sum_i p_i = 1$. Even though finding the best partition is NP-hard, there are many partition heuristics that maintain the redundancy bound of 1. The fundamental principle of $Insert_1$ is to insert a new node in a way which obtains the best partitioning 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 algorithm is described in Figure 3. To insert member $M$ in a tree with root node $R$, the procedure is called as $Insert_1(M, R)$.

```
Insert_1(member M, node X):

if (M.p ≥ X.left.p) and (M.p ≥ X.right.p)
    Put(M, X);
else if (X.left.p ≥ X.right.p)
    Insert_1(M, X.right);
else
    Insert_1(M, X.left);
```

**Fig. 3.** Algorithm $Insert_1$. It tries to keep the subtree probabilities as balanced as possible at every level.

**Algorithm 2:** The second algorithm, $Insert_2$, finds the best insertion point for member $M$ by searching all possible insertion points in the tree. The amount of increase in the average external path length that will be caused by $Put(M, X)$ at node $X$ of depth $d$ is equal to $d\,M.p + X.p$. $Insert_2$ searches the whole tree to find the location which minimizes this quantity. In Figure 4, $d(X)$ denotes the depth of node $X$ in tree $T$.

Computational performance of $Insert_2$ can be improved by taking shortcuts in finding $X_{\min}$. For example, when $X.p \leq M.p$ the subtree under $X$ need not be

```
Insert_2(member M, tree T):

Cost_min ← ∞
For each X ∈ T do
    Cost[X] ← d(X)M.p + X.p
    if Cost[X] < Cost_min
        X_min ← X
        Cost_min ← Cost[X]
Put(M, X_min)
```

**Fig. 4.** Algorithm $Insert_2$. It searches the whole tree for the insertion location that would minimize the increase in the average external path length of the tree.

searched. More sophisticated shortcuts which improve the performance further are also possible. But in the worst case, $Cost[X]$ should be computed for all nodes in the tree. Nevertheless, the formula for $Cost[X]$ is quite simple and can be computed quite efficiently. So, when the computation power of the server is plentiful compared to the bandwidth, $Insert_2$ can be the method of choice which obtains improved reduction in number of rekeys at the expense of computational cost.

## 6 Weights other than Probabilities

To use the insertion algorithms as described above, it is crucial to know the $p_i$ values of all members in the tree at insertion time. This requirement is not practical since computing the $p_i$ values would require the knowledge of the rekey time probability functions for all members in the tree. Moreover, even if the rekey time probability functions are known for all members, the $p_i$ values will change continuously as members stay in the group (unless the probability functions are memoryless) which further hinders the usage of actual probability values for insertion.

In this section, we discuss an alternative weight assignment technique to use with the insertion algorithms. First we note that the $Insert_1$ and $Insert_2$ algorithms, as well as Huffman and Shannon-Fano coding, can dispense with the restriction that $\sum_i p_i = 1$ and can work with any non-negative weights $w_i$, as long as the relative proportions are kept the same. Corresponding $p_i$ values that satisfy $\sum_i p_i = 1$ can be calculated as $p_i = w_i/W$, where $W = \sum_i w_i$.

The weight assignment of our choice for the insertion algorithms is the inverse of the mean inter-rekey time of members; i.e.,

$$w_i = 1/\mu_i \tag{2}$$

where $\mu_i$ is the average time between two rekeys by member $M_i$. There are two reasons for our choice of $1/\mu_i$ as the weight measure among many other candidates:

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.

Moreover, the estimation of the $\mu_i$ values can be done quite efficiently from the average of past rekey times of members, which should be maintained if a probabilistic LKH organization will be implemented. In fact, maintaining only the average value of past inter-rekey times along with their count is sufficient to estimate the mean inter-rekey time. More sophisticated estimates of $\mu_i$ can be obtained by further analysis of the members' behavior.

## 7 Simulation Experiments

We tested the performance of $Insert_1$ and $Insert_2$ with a large number of computer simulations. The simulations are run in four different scenarios, classified with respect to a number of group characteristics. In one division, the multicast groups are classified as *terminating* vs. *non-terminating* groups. Terminating groups exist for a specified time period, whereas the lifetime of a non-terminating group is practically infinite. In another division, groups are classified as *dynamic* vs. *semi-static* groups. In dynamic groups, members join and leave the group continually and the main source of rekey operations is the leaving members. In semi-static groups, a joining member stays in the group, or keeps the access rights, till the end of the session. In this case, all rekeys are due to compromised members. Semi-static groups are typically terminating groups. Another factor in the classification of the simulations is the joining time of the group members. Members either join the group at a constant rate, according to an exponential inter-arrival time distribution, or they all join at the beginning of the session.

There are two main sources of randomness in the simulations regarding the rekey times of group members:

1. *Variation among group members.* Mean inter-rekey time, i.e. the average time period between two rekey events by a member, varies among group members. The mean inter-rekey time values, denoted by $\mu_i$ for member $M_i$, are distributed according to a *source probability distribution function*, $D_S$, with a mean value of $\mu_S$.
2. *Randomness within a member.* The time of the next rekey event by each member is a random variable, distributed by a *rekey probability distribution function*, $D_R$, with mean $\mu_i$ for member $M_i$.

So, when a new member $M_i$ joins a group in the simulations, first it is assigned a $\mu_i$ value from $D_S$, and then it generates the times of future rekey events according to $\mu_i$ and $D_R$. Regarding the variance of the distributions $D_S$ and $D_R$, a coefficient $c_\sigma$, called the *variance factor*, is used which denotes the standard deviation of a distribution in terms of its mean, i.e. $\sigma = c_\sigma \mu$.

The following list summarizes the notation used for the group parameters:

| | |
|---|---|
| $T$ | lifetime of the session |
| $\lambda_A$ | arrival rate of new members |
| $D_S$ | source probability distribution function for $\mu_i$ values |
| $\mu_S$ | mean value for $D_S$ |

The following list summarizes the notation used for the rekey time of individual members:

| | |
|---|---|
| $t_i$ | next rekey time for member $M_i$ |
| $\mu_i$ | mean inter-rekey time for member $M_i$ |
| $D_R$ | probability distribution function for the inter-rekey time of individual members |

In the simulations, we used many different distribution functions and many different variance factors for $D_S$ and $D_R$. The tests showed that the form of $D_R$ (i.e. its being normal, uniform or exponential) and its variance have very little effect on the performance results. The tests also showed that the single most important factor for the performance of the probabilistic LKH algorithms is the variance factor of $D_S$ (i.e. variation among group members), quite independent of the form of the function for $D_S$. Unless otherwise is stated, the presented simulations use the normal distribution for $D_S$ and $D_R$, with a fixed variance factor of 0.5 for $D_R$, which is a good representative of the average case. In the following tables, $c_\sigma$ is used exclusively to denote the variance factor of $D_S$.

### 7.1 Simulation Results

During each simulation run, three LKH trees are maintained for the multicast group, one for each insertion algorithm. The performance of each algorithm is calculated as the number of keys updated by member compromise and eviction events. The tables present the number of key updates in the trees of $Insert_1$ and $Insert_2$ as a fraction of the key updates in the basic balanced LKH tree. The presented results are the averages obtained over one hundred randomly generated simulation runs. $I_1$ and $I_2$ denote $Insert_1$ and $Insert_2$ respectively.

**Scenario 1** The first scenario we consider is a terminating, semi-static group, where all members join the group at the beginning of the session. The important parameters for this scenario are the size of the group, $n$, the lifetime of the session, $T$, and the average inter-rekey time of the members, $\mu_S$. In fact, nominal values of $T$ and $\mu_S$ do not matter and the important parameter is their ratio $T/\mu_S$, which we denote by $c_T$. Roughly speaking, $c_T$ denotes the number of rekeys an average member would cause during the lifetime of the session. The results are summarized in Table 1.

**Scenario 2** In the second scenario, we again consider a terminating, semi-static group. But this time new members keep joining at a constant rate till the end of the session. Important parameters for this case include the lifetime of the session in terms of the mean inter-arrival time, $T/(1/\lambda_A)$, and in terms of the average inter-rekey time of the members, $T/\mu_S$. For simplicity, we take the average inter-arrival time $1/\lambda_A$ as the unit time, so $T$ denotes $T/(1/\lambda_A)$. $T/\mu_S$ is denoted by $c_T$. The results are summarized in Table 2.

| | $c_T = 0.5$ | | $c_T = 1$ | | $c_T = 2$ | | $c_T = 4$ | |
|---|---|---|---|---|---|---|---|---|
| $c_\sigma$ | $I_1$ | $I_2$ | $I_1$ | $I_2$ | $I_1$ | $I_2$ | $I_1$ | $I_2$ |
| 0.1 | .99 | .98 | .99 | .98 | .99 | .99 | .99 | .99 |
| 0.5 | .79 | .76 | .79 | .78 | .87 | .85 | .89 | .88 |
| 1.0 | .66 | .63 | .74 | .72 | .76 | .74 | .75 | .74 |

(a) $n = 10^2$

| | $c_T = 0.5$ | | $c_T = 1$ | | $c_T = 2$ | | $c_T = 4$ | |
|---|---|---|---|---|---|---|---|---|
| $c_\sigma$ | $I_1$ | $I_2$ | $I_1$ | $I_2$ | $I_1$ | $I_2$ | $I_1$ | $I_2$ |
| 0.1 | .99 | .99 | .99 | .99 | .99 | .99 | .99 | .99 |
| 0.5 | .73 | .72 | .81 | .80 | .84 | .83 | .85 | .84 |
| 1.0 | .65 | .63 | .69 | .68 | .73 | .72 | .72 | .71 |

(b) $n = 10^3$

| | $c_T = 0.5$ | | $c_T = 1$ | | $c_T = 2$ | | $c_T = 4$ | |
|---|---|---|---|---|---|---|---|---|
| $c_\sigma$ | $I_1$ | $I_2$ | $I_1$ | $I_2$ | $I_1$ | $I_2$ | $I_1$ | $I_2$ |
| 0.1 | .99 | .99 | .99 | .99 | .99 | .99 | .99 | .99 |
| 0.5 | .69 | .67 | .70 | .70 | .75 | .74 | .73 | .73 |
| 1.0 | .62 | .61 | .68 | .68 | .69 | .68 | .69 | .69 |

(c) $n = 10^4$

**Table 1.** Test results for Scenario 1. The results show that the algorithms provide significant reductions in rekey costs when there is a significant variation among the rekey rates of the members (i.e. large values of $c_\sigma$).

**Scenario 3** The third scenario we consider is a terminating, dynamic group. It is similar to Scenario 2 except that members may leave the group before the end of the session. All rekeys are due to leaving members and there are no additional compromise events. Hence, inter-rekey parameters such as $\mu_S$ and $\mu_i$ should be interpreted as parameters for the member lifetime (i.e. time of stay in the group). The test parameters are similar to those in Scenario 2. The results are summarized in Table 3.

**Scenario 4** In the fourth scenario, we consider a long-term dynamic group. The session lifetime $T$ is practically infinite. Members join and leave the session at a certain rate. All rekey operations are due to departing members. The important parameter in this case is the average member lifetime in terms of the average inter-arrival time, $\mu_S/(1/\lambda_A)$. In the steady state [11], the departure rate is equal to the arrival rate, and hence, the group has an average of $n = \mu_S \lambda_A$ members. The measurements are taken over 10,000 consecutive rekey operations in the steady state. Again, $1/\lambda_A$ is taken as the unit time. The results are summarized in Table 4.

| | $c_T = 0.5$ | | $c_T = 1$ | | $c_T = 2$ | | $c_T = 4$ | |
|---|---|---|---|---|---|---|---|---|
| $c_\sigma$ | $I_1$ | $I_2$ | $I_1$ | $I_2$ | $I_1$ | $I_2$ | $I_1$ | $I_2$ |
| 0.1 | .99 | .99 | .99 | .99 | .99 | .99 | 1.00 | 1.00 |
| 0.5 | .80 | .77 | .81 | .78 | .84 | .83 | .88 | .86 |
| 1.0 | .64 | .61 | .70 | .67 | .76 | .73 | .78 | .76 |

(a) $T = 10^2$

| | $c_T = 0.5$ | | $c_T = 1$ | | $c_T = 2$ | | $c_T = 4$ | |
|---|---|---|---|---|---|---|---|---|
| $c_\sigma$ | $I_1$ | $I_2$ | $I_1$ | $I_2$ | $I_1$ | $I_2$ | $I_1$ | $I_2$ |
| 0.1 | .99 | .99 | .99 | .99 | .99 | .99 | 1.00 | 1.00 |
| 0.5 | .72 | .70 | .75 | .74 | .82 | .81 | .81 | .80 |
| 1.0 | .61 | .59 | .65 | .63 | .66 | .64 | .70 | .68 |

(b) $T = 10^3$

| | $c_T = 0.5$ | | $c_T = 1$ | | $c_T = 2$ | | $c_T = 4$ | |
|---|---|---|---|---|---|---|---|---|
| $c_\sigma$ | $I_1$ | $I_2$ | $I_1$ | $I_2$ | $I_1$ | $I_2$ | $I_1$ | $I_2$ |
| 0.1 | .99 | .99 | .99 | .99 | .99 | .99 | 1.00 | 1.00 |
| 0.5 | .67 | .65 | .72 | .71 | .74 | .73 | .78 | .78 |
| 1.0 | .63 | .61 | .65 | .64 | .65 | .64 | .64 | .63 |

(c) $T = 10^4$

**Table 2.** Test results for Scenario 2. The results resemble those in Table 1. The algorithms provide significant improvements when there is a significant variation among the rekey rates of the members (i.e. large values of $c_\sigma$).

### 7.2 Comments on Results

The test results show that the algorithms for probabilistic LKH organization make the biggest difference when there is a significant variance among the rekey rates of the group members (i.e. large $c_\sigma$ in $D_S$) and also when there are compromise events in addition to those caused by leaving members (e.g. Scenarios 1 and 2). In these cases, the algorithms provide up to 40% reduction in the cost of rekey operations. Larger group sizes contribute positively to the reduction rates as well.

When the main source of rekeys is leaving members, the algorithms provide significant gains if average member inter-rekey time $\mu_S$ is close to the session time or longer (i.e. the smaller values of $c_T$ in the third scenario); because in this case most rekey events come from short-living members allocated closer to the root of the LKH tree. When the session time is significantly longer than $\mu_S$ and the main source of rekeys is member evictions (Scenario 4, or larger values of $c_T$ in the Scenario 3), members allocated deeper in the tree also contribute to the rekey events, and the improvement rates obtained by the algorithms drop to 5% or less.

| | $c_T = 0.5$ | | $c_T = 1$ | | $c_T = 2$ | | $c_T = 4$ | |
|---|---|---|---|---|---|---|---|---|
| $c_\sigma$ | $I_1$ | $I_2$ | $I_1$ | $I_2$ | $I_1$ | $I_2$ | $I_1$ | $I_2$ |
| 0.1 | .99 | .98 | .98 | .98 | .99 | .99 | 1.00 | .99 |
| 0.5 | .83 | .80 | .87 | .85 | .92 | .91 | .95 | .93 |
| 1.0 | .73 | .68 | .79 | .76 | .88 | .84 | .90 | .87 |

(a) $T = 10^2$

| | $c_T = 0.5$ | | $c_T = 1$ | | $c_T = 2$ | | $c_T = 4$ | |
|---|---|---|---|---|---|---|---|---|
| $c_\sigma$ | $I_1$ | $I_2$ | $I_1$ | $I_2$ | $I_1$ | $I_2$ | $I_1$ | $I_2$ |
| 0.1 | .99 | .99 | .99 | .99 | .99 | .99 | 1.00 | .99 |
| 0.5 | .88 | .86 | .92 | .90 | .95 | .94 | .97 | .96 |
| 1.0 | .81 | .79 | .87 | .84 | .91 | .89 | .93 | .91 |

(b) $T = 10^3$

| | $c_T = 0.5$ | | $c_T = 1$ | | $c_T = 2$ | | $c_T = 4$ | |
|---|---|---|---|---|---|---|---|---|
| $c_\sigma$ | $I_1$ | $I_2$ | $I_1$ | $I_2$ | $I_1$ | $I_2$ | $I_1$ | $I_2$ |
| 0.1 | .99 | .99 | .99 | .99 | .99 | .99 | 1.00 | 1.00 |
| 0.5 | .92 | .90 | .94 | .93 | .96 | .96 | .97 | .97 |
| 1.0 | .86 | .85 | .90 | .89 | .93 | .92 | .95 | .94 |

(c) $T = 10^4$

**Table 3.** Test results for Scenario 3. The improvement rates are more significant for smaller values of $c_T$. The source of the difference from Scenario 2 is that, in this scenario, all rekeys are due to member departures, so each member causes at most one rekey event. Hence, when the session is significantly longer than the average member stay time (i.e. larger values of $c_T$), differences among expected member stay times become less important.

We would like to note that in all these simulations it is assumed the overall space of potential members is so large that the distribution of the joining members is unaffected by the leaving members (or, the current members) of the group, which is practically equivalent to the case where leaving members never return. So, our simulation scenarios do not represent the cases where a few very dynamic members can affect the rekey dynamics significantly by frequent join and leave operations. In such cases, the probabilistic insertion algorithms can provide very significant gains even if all rekey operations are due to leaving members. Such gains are not reflected in the results of simulation scenarios 3 and 4, where all rekeys are due to leaving members, since the incoming member parameters in the simulations are independent of those who have left the group.

Finally, it is interesting to note that the improvement figures obtained by $Insert_1$, which does not induce any additional computational cost over the basic balanced-tree LKH insertion, are consistently very close to those obtained by $Insert_2$, which searches the whole tree for the best insertion point. This, in our

| | $\mu_s = 10^2$ | | $\mu_s = 10^3$ | | $\mu_s = 10^4$ | |
|---|---|---|---|---|---|---|
| $c_\sigma$ | $I_1$ | $I_2$ | $I_1$ | $I_2$ | $I_1$ | $I_2$ |
| 0.1 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| 0.5 | 0.98 | 0.97 | 0.98 | 0.98 | 0.98 | 0.98 |
| 1.0 | 0.95 | 0.94 | 0.97 | 0.95 | 0.98 | 0.96 |

**Table 4.** Test results for Scenario 4. The improvement rates are quite modest compared to those for Scenarios 1, 2 and 3.

opinion, indicates the strength of the basic idea underlying $Insert_1$, that is to keep the subtree probabilities as balanced as possible.

## 8 Conclusions

In this paper, two algorithms are described which can reduce the cost of multicast key management significantly, depending on certain characteristics of the multicast group. The algorithms described here are not specific to the basic LKH scheme of Wallner et al. [18] but are also applicable to more sophisticated LKH-based techniques such as the OFT scheme of McGrew and Sherman [13] and the OFC scheme of Canetti et al. [6]. The algorithms can work with relatively small computational overhead (and no overhead in case of $Insert_1$) and can provide significant reductions in the message complexity of rekey operations. The improvements can be significant or modest, depending on certain characteristics of the group, as suggested by the simulations on different multicast scenarios in Section 7.

The requirement of using actual probabilities for tree organization can be a major limitation for probabilistic LKH organization techniques. Instead of using actual probabilities, we suggest using a heuristic weight assignment technique for the tree organization, which is described in Section 6. Simulations summarized in Section 7, which were implemented with this weight assignment technique, show that this heuristic method works effectively in practice.

The studies of Almeroth and Ammar [1, 2] 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 and it is practical to maintain data regarding past behavior of the members, we believe the algorithms discussed in this paper can provide significant reductions in the cost of rekey operations in multicast key management.

## Acknowledgments

We would like to thank Eric Harder and Chris McCubbin for many helpful suggestions and informative discussions.

## References


[1] K. Almeroth and M. Ammar. Collection and modeling of the join/leave behavior of multicast group members in the mbone. In *High Performance Distributed Computing Focus Workshop (HPDC'96)*, August 1996.

[2] K. Almeroth and M. Ammar. Multicast group behavior in the internet's multicast backbone (mbone). *IEEE Communications*, 35(6), June 1997.

[3] A. Ballardie. Scalable multicast key distribution, May 1996. Internet RFC 1949.

[4] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression*. Prentice-Hall, 1990.

[5] M. Burmester and Y. Desmedt. A secure and efficient conference key distribution system. In Alfredo De Santis, editor, *Advances in Cryptology—Eurocrypt'94*, pages 275–286. Springer-Verlag, 1994.

[6] R. Canetti, J. Garay, G. Itkis, D. Micciancio, M. Naor, and B. Pinkas. Multicast security: A taxonomy and some efficient constructions. In *Infocomm'99 Conference*, 1999.

[7] A. Fiat and M. Naor. Broadcast encryption. In Douglas R. Stinson, editor, *Advances in Cryptology—CRYPTO '93*, pages 480–491. Springer-Verlag, 1993.

[8] T. Hardjono, B. Cain, and N. Doraswamy. A framework for group key management for multicast security, February 2000. Internet draft (work in progress).

[9] H. Harney, C. Muckenhirn, and T. Rivers. Group key management protocol specification, July 1997. Internet RFC 2093.

[10] D. E. Knuth. Dynamic Huffman coding. *Journal of Algorithms*, 6:163–180, 1985.

[11] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis*. McGraw-Hill, 3rd edition, 2000.

[12] M. Luby and J. Staddon. Combinatorial bounds for broadcast encryption. In *Advances in Cryptology—EUROCRYPT '93*. Springer-Verlag, 1998.

[13] D. A. McGrew and A. T. Sherman. Key establishment in large dynamic groups using one-way function trees. Technical Report 0755, TIS Labs, May 1998. A revised version to appear in the IEEE Transactions on Software Engineering.

[14] S. Mittra. Iolus: A framework for scalable secure multicasting. In *Proceedings of the ACM SIGCOMM'97 Conference*, September 1997.

[15] R. Poovendran and J. S. Baras. An information theoretic analysis of rooted-tree based secure multicast key distribution schemes. In *Advances in Cryptology—Crypto'99*. Springer-Verlag, 1999.

[16] S. Saeednia and R. Safavi-Naini. Efficient identity-based conference key distribution protocols. In *Proceedings of Information Security and Privacy Conference, ACISP'98*. Springer-Verlag, 1998.

[17] M. Steiner, G. Tsudik, and M. Waidner. CLIQUES: A new approach to group key agreement. In *International Conference on Distributed Computing Systems*, pages 380–387. IEEE Computer Society, 1998.

[18] D. Wallner, E. Harder, and R. Agee. Key management for multicast: Issues and architectures, July 1997. Internet draft (work in progress).

[19] C. K. Wong, M. Gouda, and S. S. Lam. Secure group communication using key graphs. In *Proceedings of the ACM SIGCOMM'98 Conference*, September 1998.