Probabilistic Optimization Techniques for Multicast Key Management ⋆

Ali Aydın Selçuk [1]

Department of Computer Sciences
Purdue University
West Lafayette, IN 47906, USA
selcuk@cs.purdue.edu

Deepinder Sidhu

Maryland Center for Telecommunications Research
Department of Computer Science and Electrical Engineering
University of Maryland Baltimore County
Baltimore, MD 21250, USA
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 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. 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 amount of variation in the rekey characteristics of the group members.

Key words: Network security, multicast security, group key management.

[1] This work was done while the author was at the Maryland Center for Telecommunications Research, University of Maryland Baltimore County.

⋆ 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

1 Introduction

Multicast is a technology for efficient one-to-many and many-to-many communications over a wide area network. It enables a single packet sent out by a source to be delivered to multiple destinations by replication at multicast routers over a routing tree. Multicast is an attractive technology for sending data to large groups since it reduces the sender bandwidth requirements, the network load, and the latency for the receivers. Some applications that benefit from multicast include audio and video distribution over the Internet, group collaboration applications such as video conferencing, distributed simulations, and multiplayer games, and distribution of large files to multiple destinations such as the Usenet News distribution. Issues in multicasting include routing, reliability, and security. The requirements are similar to those in unicasting, with the additional requirement that the protocols must be highly efficient and scalable to support large groups.

Multicast security is concerned with guaranteeing the privacy, authentication, and integrity of the group communication. These functionalities in multicast security are provided by encrypting and authenticating the group messages with a group key that is shared by all members of the group. Typically, a *group key manager* is in charge of maintaining and distributing the group key. The issue of efficient key management is a significant problem in large multicast groups. Especially if the group is highly dynamic in terms of membership and if the security policy requires that the group key be changed after every leaving member so that nobody outside the group has the key, then key management can be a very significant problem.

Currently, the most efficient methods for multicast key management are based on the Logical Key Hierarchy (LKH) scheme, which was independently discovered by Wallner et al. [23] and Wong et al. [24]. In LKH, group members are organized as leaves of a tree with logical internal nodes. When a member is compromised, possibly due to its leaving the group, the cost of recovering the group from this compromise is proportional to the depth of the compromised member in the LKH tree. The original LKH scheme proposes maintaining a balanced 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 the Crypto'99 conference [17]; but no solutions have been proposed to date. We propose two algorithms which combine ideas from data compression with the special re-

expressed or implied, of the Department of Defense or the U.S. Government.

quirements of multicast key management. Simulation results show that these algorithms can reduce the cost of multicast key management significantly, depending on the amount of variation in the rekey characteristics of the group members.

The remainder of this paper is organized as follows: In Section 2, we review the LKH scheme for multicast key management. In Section 3, we introduce and analyze the problem of probabilistic LKH optimization. In Sections 4 and 5, we propose two algorithms for this problem and discuss their design rationale. In Section 6, we prove an upper bound for the possible inoptimality of these algorithms. In Section 7, we propose a weight assignment technique needed for practical use of the algorithms. In Section 8, we describe the simulation experiments and present their results. In Section 9, we conclude with a discussion of the issues regarding an effective utilization of the techniques discussed in this paper.

1.1 Related Work

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 [5,21,18]. 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 small 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. [10], the group key is passed to each member by a unicast communication with the group key manager. This protocol has the disadvantage that when a member is compromised, possibly due to a departure, the group key manager has to re-initialize the group by distributing the new key to every individual member one by one. 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. To distribute a new group key, the key manager passes the key to the neighbor routers in the multicast tree over a secure unicast channel, and the routers pass the key to the group members and to the other routers in the tree. This protocol is scalable to large groups, but has the disadvantage of requiring trusted routers and being specific to the CBT routing protocol. The

Iolus protocol [16] 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. [9], where the group members are divided into "leaf regions", and the managers of the 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. [23] and Wong et al. [24]. 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 [15,6]. Certain similarities between LKH trees and some information-theoretic concepts have been pointed out in [17].

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 [14] 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: ith 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 the summations $\sum_i$ ranges from 1 to n, unless otherwise is stated.

2 The LKH Scheme

The main motivation behind the LKH scheme is to be able to efficiently rekey the group (i.e. to update and distribute a new group key) in case of a member compromise, possibly due to a departure.

In a simple, naive group key distribution protocol, the key manager contacts each member individually to pass the new group key, using a pairwise shared secret key to encrypt it. When a member is compromised in such a group, there is no way of recovery but the key manager contacting all members one by one to give a new group key. The cost of such a recovery operation can be prohibitively high for the key manager if the multicast group is large consisting of thousands or millions of members.

The LKH scheme aims to reduce the cost of a compromise recovery operation by adding extra encryption keys into the system. The members of the group are organized as leaves of a "logical" key tree which is maintained by the key manager. The internal (non-leaf) nodes in this tree are "logical" entities which do not correspond to any real-life entities of the multicast group, but are used for key distribution purposes only. 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.

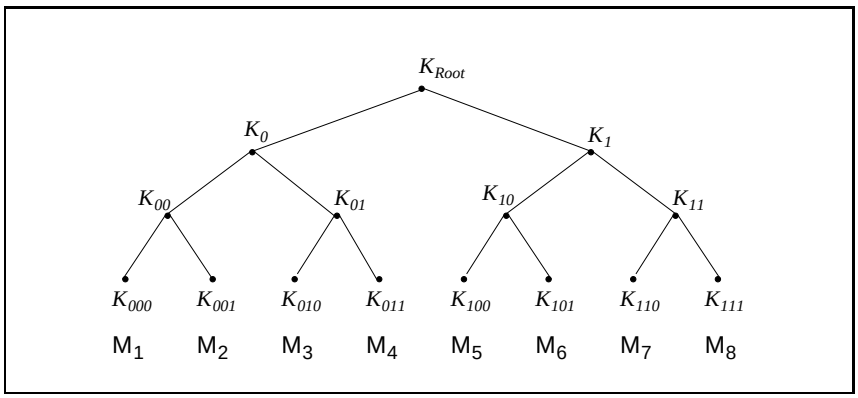


Fig. 1. An example LKH tree with eight members. Each member holds the keys on the path from its leaf node to the root. K_{Root} is the key shared by all group members.

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 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 [23,15,6]. 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 and its remaining child is linked directly to the node above it.)

3 The Problem of 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 pushing more stable members deeper down the tree.

The rekey operations caused by a periodic key update or a joining member can be achieved relatively easily with a single fixed-size multicast message by using a one-way function to update the keys [15]. The more costly rekey operations are those which 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 $ad_i + b$, $a > 0$, where the exact values of a and b depend on the specifics of the LKH implementation. The objective in this study is to minimize the cost of this more costly kind of rekey operations that are caused by a member compromise or eviction event.

The optimal solution to this problem is the tree organization that will minimize

the average cost of all future rekey operations. However, finding this optimal solution is not possible in practice since that would require the knowledge of the rekey probability distributions of 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 of this intractable problem, 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 $\sum_i p_i d_i$ 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.

3.1 LKH Optimization and Data Compression

The problem mentioned above has many similarities to the data compression problem with code trees. In the latter problem, messages to be encoded are organized as leaves of a binary tree. The codeword used to encode a message is obtained by traversing the path from the root of the tree to the leaf for that message, assigning a 0 for every left branch taken and a 1 for every right branch. The length of a codeword obtained by this method is equal to d_i bits, where d_i is the depth of the message in the tree. The average code length per message is equal to $\sum_i p_i d_i$, where p_i is the probability of message m_i to be the next to appear. The quantity $\sum_i p_i d_i$ is known as the *average external path length* of the tree. Shannon's Noiseless Source Coding Theorem [20] shows that the minimum average code length that can be attained by an encoding scheme is $-\sum_i p_i \log p_i$, known as the *entropy* of the message source. The difference between the entropy and the actual average code length is known as the *redundancy* of the code.

The optimal tree that minimizes the average external path length for a given set of messages $\{m_1, m_2, \ldots, m_n\}$ with probabilities $\{p_1, p_2, \ldots, p_n\}$ is obtained by the Huffman algorithm [11]. The Huffman algorithm builds a tree from the set of messages recursively, combining each time the two nodes with the smallest probabilities until all the nodes are combined as a tree. Another solution that gives very good compression in practice but which is slightly sub-optimal is the Shannon-Fano trees [4]. The Shannon-Fano algorithm builds a tree from a given set of messages by repeatedly dividing the message set into two parts of (roughly) equal probability until every set includes a single element. The redundancy of the Huffman trees is proven to be less than $p_{max} + 0.086$, where p_{max} is the probability of the most likely message [8]. The redundancy of the Shannon-Fano trees is less than 1 [4]. Although finding the best partition in the Shannon-Fano coding is NP-hard, there are many partition heuristics that maintain the redundancy bound of 1.

Although the data compression problem and the LKH optimization problem share many similarities, they have significant differences as well. For dynamic data compression problems where message probabilities can change and new messages can be added into the system during the process, there are dynamic Huffman algorithms (e.g. [12,22]) which dynamically maintain a Huffman tree preserving the the optimal average external path length. These algorithms maintain the Huffman structure by changing the location of the messages in the tree according to the changing probabilities. Such changes do not incur any overhead on the objective function (i.e., to minimize the average code length). On the other hand, locational changes in a dynamic LKH tree would mean rekeying entire paths of the tree where members are moved, just contrary to the objective (i.e., to minimize the average number of rekeys). Therefore, for the purposes of efficient key management, an LKH scheme with sub-optimal $\sum_i p_i d_i$ can have a better overall performance than the one which tries to keep $\sum_i p_i d_i$ minimal all the time.

Another difference of the LKH trees from the 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, 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. We will focus on the insertion operations of this kind, which preserve the relative location of the present members. These operations will work by first choosing an insertion point in the existing tree for a given new node, and then inserting the new node at the

chosen location by the Put procedure, illustrated in Figure 2.

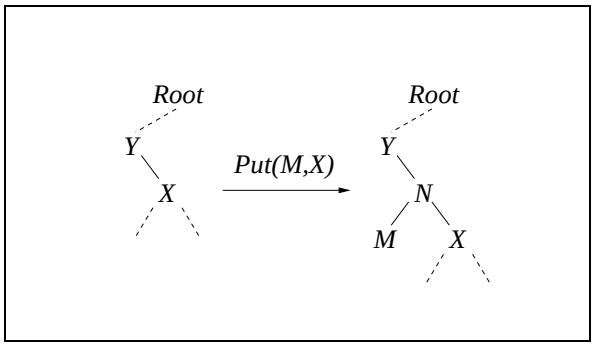


Fig. 2. The Put procedure, inserting the new member M at the chosen location X. Relative location of existing nodes is kept the same to avoid extra rekey operations.

That is, to insert a new member M into the group, a new internal node N is created at the chosen 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 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 on the insertion operation in an n-member tree.

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. 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 path length close to the optimality 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 X is a leaf node
    Put(M, X);
else 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 $d\,M.p + X.p$. The algorithm $Insert_2$ searches the whole tree to find the location that minimizes this quantity. In Figure 4, $d(X)$ denotes the depth of node X in tree T.

```
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.

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 searched. More sophisticated shortcuts which improve the performance further are also possible. But in the worst case, $Cost[X]$ will need to 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 and the bandwidth is scarce, $Insert_2$ can be the method of choice which obtains improved reduction in number of rekeys at the expense of computational cost.

6 A Redundancy Bound for $Insert_1$ Trees

In this section, we prove a bound on the maximum redundancy of a tree that is created with $Insert_1$ (henceforth an $Insert_1$ *tree*). We first prove that a node at depth d in an $Insert_1$ tree must have a probability less than or equal to $1/\mathcal{F}_d$, where $\mathcal{F}_d$ is the dth Fibonacci number, defined by the recurrence $\mathcal{F}_0 = 1$, $\mathcal{F}_1 = 1$, $\mathcal{F}_i = \mathcal{F}_{i-1} + \mathcal{F}_{i-2}$ for $i \geq 2$. Then we show that the redundancy of the tree is bounded by approximately 0.3 times the average external path length of the tree.

In the following discussion, $d(X)$ denotes the depth of a node X in the tree, $P(X)$ denotes the probability of X, $T(X)$ denotes the subtree rooted at X, and $s[X]$, $f[X]$, $u[X]$, and $g[X]$ denote the nodes that are sibling, parent, uncle (i.e. the sibling of the parent), and grandparent to X in the tree respectively.

Definition 1 *An $Insert_1$ tree is a binary tree that is created by a sequence of $Insert_1$ operations, with no deletion operations in between.*

Lemma 2 *For any node X with depth $d(X) \geq 2$ in an $Insert_1$ tree,*

$$P(u[X]) \geq P(X).$$

PROOF. We prove the result by induction on the insertions into the tree. Let U, F, and S denote the nodes $u[X]$, $f[X]$, and $s[X]$ respectively.

The base case is the insertion operation where U becomes X's uncle for the first time. This event can happen as a result of the insertion of either U, X, or S.

- If U is the node just inserted, then for its being inserted as a sibling to F it is necessary that $P(U) \geq P(X)$ as well as $P(U) \geq P(S)$.
- If X is the node just inserted, and it is inserted into the subtree adjacent to U, then $P(X) < P(U)$ as well as $P(U) \geq P(S)$.
- If S is the node just inserted, and it is inserted into the subtree adjacent to U, then $P(S) < P(U)$ as well as $P(U) \geq P(X)$.

In all three cases, $P(U) \geq P(X)$ must be true for U to become the uncle of X.

In future insertion operations, the relative proportion of $P(U)$ and $P(X)$ is changed if and only if a new node is put either into $T(U)$ or into $T(X)$.

Assume a new node is inserted into $T(U)$ at a time when $P(U) \geq P(X)$ is already true. Then $P(U)$ gets even larger by this insertion, and $P(U) \geq P(X)$ is still true.

Assume a new node N is inserted into $T(X)$, hence into $T(F)$. Then the following must have been true before the insertion:

$$P(U) \geq P(F), P(N)$$
$$P(S) \geq P(X), P(N)$$

Let P' denote the probabilities after the insertion, whereas P is used for the probabilities before the insertion.

$$\begin{aligned} P'(X) &= P(X) + P(N) \\ &\leq P(X) + P(S) \\ &= P(F) \\ &\leq P(U) \\ &= P'(U) \end{aligned}$$

Hence, $P(U) \geq P(X)$ remains true whether a new node is inserted into $T(U)$ or into $T(X)$. $\square$

Corollary 3 *For any node X with depth $d(X) \geq 2$ in an $Insert_1$ tree,*

$$P(g[X]) \geq 2P(X).$$

Theorem 4 *Let $a_i[X]$, $i \geq 1$, denote the i^{th} generation ancestor of node X in an $Insert_1$ tree. Then,*

$$\frac{P(X)}{P(a_i[X])} \leq \frac{1}{\mathcal{F}_i}$$

where $\mathcal{F}_i$ is the i^{th} Fibonacci number, defined by the recurrence $\mathcal{F}_0 = 1$, $\mathcal{F}_1 = 1$, $\mathcal{F}_i = \mathcal{F}_{i-1} + \mathcal{F}_{i-2}$ for $i \geq 2$.

PROOF. For $i = 1$, the result is trivial: For any ancestor A of any node X, it is necessarily true that $P(A) \geq P(X)$, and so is it for $a_1[X]$. Therefore,

$$\frac{P(X)}{P(a_1[X])} \leq 1 = \frac{1}{\mathcal{F}_1} .$$

For $i = 2$, the result follows from Corollary 3:

$$\frac{P(X)}{P(a_2[X])} \leq \frac{1}{2} = \frac{1}{\mathcal{F}_2} .$$

For $i > 2$, the result is proven by induction:

$$\begin{aligned} P(a_i[x]) &= P(a_{i-1}[x]) + P(u[a_{i-2}[x]]) \\ &\geq P(a_{i-1}[x]) + P(a_{i-2}[x]) \\ &\geq \mathcal{F}_{i-1}\, P(X) + \mathcal{F}_{i-2}\, P(X) \\ &\geq \mathcal{F}_i\, P(X). \qquad \Box \end{aligned}$$

Theorem 5 *For any node X in an $Insert_1$ tree,*

$$P(X) \leq \frac{1}{\mathcal{F}_{d(X)}}\ . \tag{1}$$

PROOF. For a node X at depth 1 or deeper in the tree, the statement is a corollary of Theorem 4 since the root node is the $d(X)$th degree ancestor of X with probability 1. The only exception of this is when X is the root itself, where $d(X) = 0$, $P(X) = 1$. Equation (1) is satisfied in this case as well, since $P(root) = 1 = 1/\mathcal{F}_0$. $\Box$

The closed formula for the Fibonacci numbers is

$$\mathcal{F}_i = \frac{1}{\sqrt{5}} \left(\alpha^{i+1} - \beta^{i+1} \right) \tag{2}$$

where $\alpha = (1 + \sqrt{5})/2 \approx 1.618$, $\beta = (1 - \sqrt{5})/2 \approx -0.618$. One consequence of (2) is that, since $\beta^{i+1}/\sqrt{5} < 0.5$ for $i \geq 0$, $\mathcal{F}_i$ is equal to $\alpha^{i+1}/\sqrt{5}$ rounded to the nearest integer. For relatively large i, the difference between $\mathcal{F}_i$ and $\alpha^{i+1}/\sqrt{5}$ is extremely small, and in logarithmic scale the difference is negligible for $i \geq 2$.

Before we proceed with the proof of the redundancy bound, let us define the *redundancy of a node X* to be the difference $d(X) - (-\log P(X))$. We write $R(X)$ to denote this difference.

Lemma 6 *The redundancy of a node X in an $Insert_1$ tree is bounded by* $(1 - \log \alpha)d(X) + \log(\sqrt{5}/\alpha) \approx 0.306 d(X) + 0.467$.

PROOF.

$$\begin{aligned} R(X) &= d(X) - (-\log P(X)) \\ &\leq d(X) + \log \frac{1}{\mathcal{F}_{d(X)}} \end{aligned}$$

$$
\begin{aligned}
&\approx d(X) + \log\left(\frac{1}{\alpha^{d(X)+1}/\sqrt{5}}\right) \\
&= d(X)(1 - \log\alpha) + \log(\sqrt{5}/\alpha). \\
&\approx 0.306 d(X) + 0.467. \qquad \square
\end{aligned}
$$

Theorem 7 *The upper bound for the redundancy of an $Insert_1$ tree is approximately*

$$0.306 \cdot AEPL + 0.467\,,$$

where $AEPL$ denotes the average external path length of the tree (i.e. the summation $\sum_i p_i d_i$ over the leaf nodes).

PROOF. The redundancy of the tree is

$$
\begin{aligned}
\sum_i p_i d_i - \sum_i p_i(-\log p_i) &= \sum_i p_i(d_i + \log p_i) \\
&\le \sum_i p_i \left(d_i(1 - \log\alpha) + \log(\sqrt{5}/\alpha)\right) \\
&\approx 0.306 \left(\sum_i p_i d_i\right) + 0.467\,. \qquad \square
\end{aligned}
$$

Like the similar bounds on Huffman and Shannon-Fano trees [8,4], the bound given in Theorem 7 is not tight in the sense that most $Insert_1$ trees have a much less redundancy than that is suggested by this bound. For now, we leave finding a tighter redundancy bound as an open problem.

A more important limitation for the use of the bound in Theorem 7 as a practical performance measure is that, as stated in Definition 1, this bound is derived for a tree with no deletion operation in its construction. When deletion operations are possible, there is no bound on the redundancy of any tree that does not allow relocation of members, including the $Insert_1$ and $Insert_2$ trees. For such cases, where there is no bound on the worst-case behavior, an average-case analysis is more applicable rather than a worst-case analysis. For the average performance of the LKH trees, obtaining an analytical result is difficult since the average performance is a complicated function of the join, leave, and compromise time distributions of all potential members in the system. We use computer simulations to evaluate the practical performance of the algorithms. The simulation experiments and their results are discussed in Section 8.

7 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 of 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.

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.,

$$w_i = 1/\mu_i \tag{3}$$

where μ_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 μ_i values can be done quite efficiently from the average of past rekey times of members, which should be maintained anyway if a probabilistic LKH organization will be implemented. More sophisticated estimates of μ_i can be obtained by further analysis of the members' behavior.

8 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 according to the membership dynamics as *dynamic*, *static*, or *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 static groups, members join the group at the beginning of the session and remain—or, at least, keep their access rights—till the end. No rekeying is needed for leaving members in these groups, and all rekeys are due to compromised members. Semi-static groups are similar to static groups in that all members keep their access rights till the end of the session. But they are different in member arrivals; new members keep arriving throughout the session according to a Poisson distribution. Both static and semi-static groups are typically terminating groups.

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 μ_i for member M_i, are distributed according to a *source probability distribution function*, D_S, with a mean value of μ_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 μ_i for member M_i.

So, when a new member M_i joins a group in the simulations, first it is assigned a μ_i value from D_S, and then it generates the times of future rekey events according to μ_i and D_R. Regarding the variance of the distributions D_S and D_R, a parameter c_σ, 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

λ_A: arrival rate of new members

D_S: source probability distribution function for μ_i values

μ_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

μ_i: mean inter-rekey time for member M_i

D_R: probability distribution function for the inter-rekey time of individual members

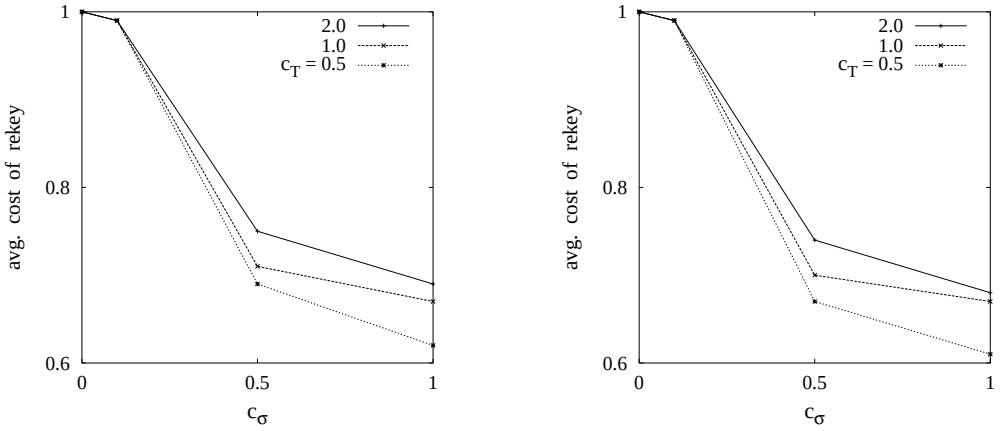
In the simulations, we used many different distribution functions and many different variance factors for D_S and D_R. The tests showed that the form of the distribution functions (i.e. their being normal, uniform or exponential) has little effect on the performance results. Similarly, the variance factor of D_R also turned out to have very little effect on the results. The single most important factor affecting the performance results turned out to be the variance factor of D_S; i.e. the variance in the group members' mean inter-rekey times. Unless otherwise is stated, the simulations presented below 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 figures, c_σ is used exclusively to denote the variance factor of D_S.

8.1 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 following results show the number of key updates in the trees of $Insert_1$ and $Insert_2$ as a ratio of the key updates in the basic balanced LKH tree. The presented results are the averages calculated over 100 randomly generated simulation runs for every data point. Due to space limitations, the results given here are only for a set of selected parameters. For a more complete account of the simulation results, please see Appendix A.

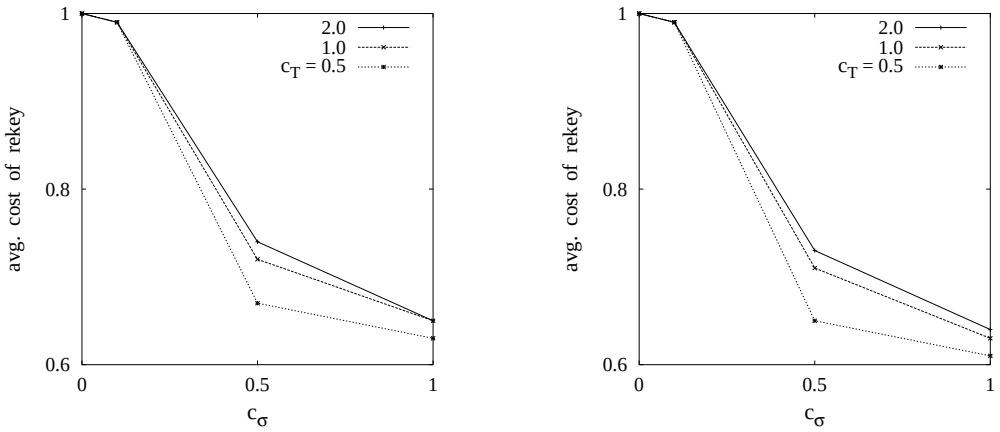
8.1.1 Scenario 1

The first scenario we consider is a terminating, static group. All members join the group at the beginning of the session and stay till the end of the session. The important parameters in this scenario are the size of the group, n, the lifetime of the session, T, and the average inter-rekey time of the members, μ_S. In fact, nominal values of T and μ_S do not matter and the important parameter is their ratio T/μ_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 simulation results for this scenario are summarized in Figure 5.



(a) $Insert_1$ (b) $Insert_2$

Fig. 5. Simulation results for Scenario 1 with $n = 10,000$. Rekey costs of $Insert_1$ and $Insert_2$ are shown as a ratio of the rekey costs of the balanced LKH tree. The improvement figures depend heavily on c_σ, the variation in the member rekey rates. Rekey costs can be reduced significantly by the insertion algorithms when c_σ is relatively large. The session lifetime factor c_T also affects the performance, but its impact is relatively less significant.



(a) $Insert_1$ (b) $Insert_2$

Fig. 6. Simulation results for Scenario 2 with $T\lambda_A = 10,000$. As in Scenario 1, the algorithms provide significant improvements when there is a significant variation in the rekey rates of the members (i.e. large values of c_σ).

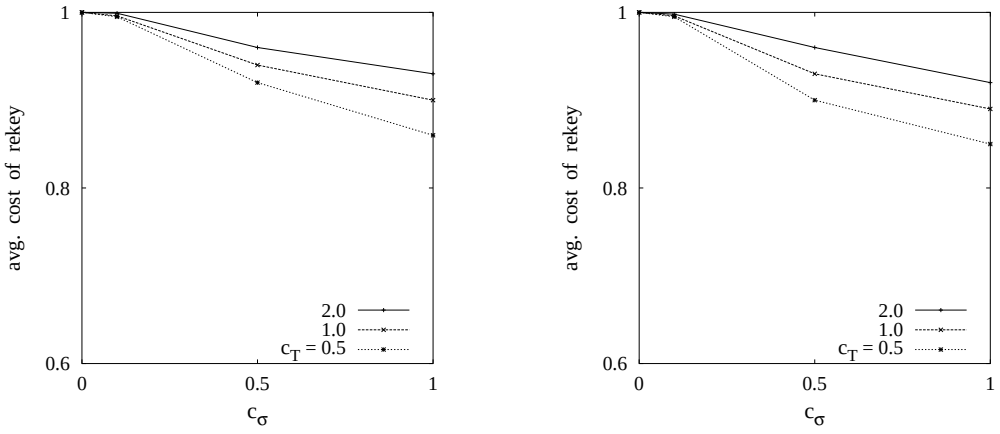
8.1.2 Scenario 2

In the second scenario, we consider a terminating, semi-static group. Members join the group at a constant rate according to a Poisson distribution, and

joining members remain in the group till the end of the session. Important parameters for this scenario include the lifetime of the session in terms of the mean inter-arrival time, $T/(1/\lambda_A) = T\lambda_A$ [2], and in terms of the average inter-rekey time of the members, T/μ_S, denoted by c_T. The results are summarized in Figure 6.

8.1.3 Scenario 3

The third scenario we consider is a terminating, dynamic group. Members join and leave the group at a certain rate till 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 μ_S and μ_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 Figure 7.



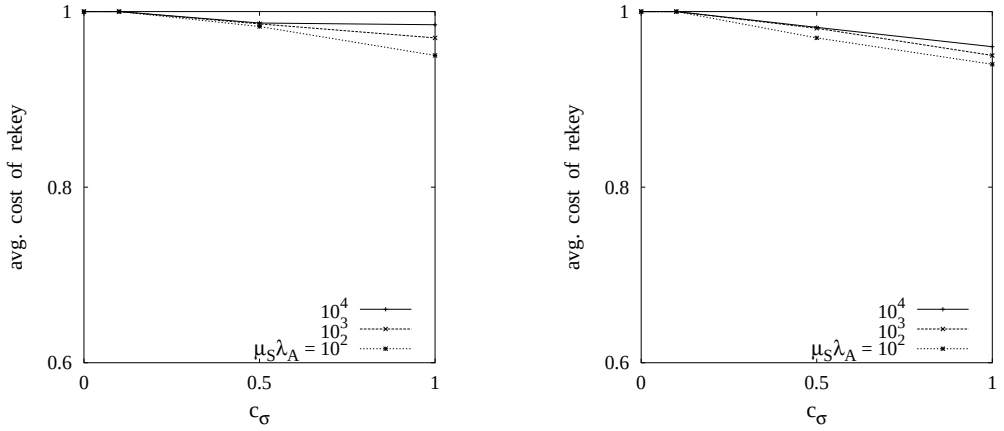
(a) $Insert_1$ (b) $Insert_2$

Fig. 7. Simulation results for Scenario 3 with $T\lambda_A = 10,000$. Similar to the previous scenarios, high improvement rates are obtained for larger values of c_σ. However, the improvement rates are lower than those in Scenario 1 and 2. This is mainly due to the fact that all rekeys here are caused by member departures; hence, each member causes at most one rekey. When the session is significantly longer than the average member stay time (i.e. larger values of c_T), most members cause exactly one rekey event and the variation in the mean stay time becomes less significant.

[2] Intuitively, $T\lambda_A$ shows the mean number of members to join the group till the end of the session.

8.1.4 Scenario 4

The fourth scenario we consider is 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, $\mu_S/(1/\lambda_A) = \mu_S\lambda_A$. In the steady state [13], 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. The results are summarized in Figure 8.



(a) $Insert_1$ (b) $Insert_2$

Fig. 8. Simulation results for Scenario 4. The improvement rates are quite modest compared to those for Scenarios 1, 2 and 3.

8.2 Comments on Results

The test results show that the algorithms for probabilistic LKH organization make the biggest difference when there is a significant variation in the rekey rates of the group members (i.e. large c_σ 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 also contribute positively to the reduction rates.

When the main source of rekeys is leaving members (i.e. Scenarios 3 and 4), the algorithms provide significant gains if average member inter-rekey time μ_S is close to the session time or longer (i.e. the smaller values of c_T in Scenario 3); because in this case most rekey events come from short-living members allocated closer to the root of the LKH tree. If the session time is significantly

longer than μ_S while the main source of rekeys is member evictions (Scenario 4, larger values of c_T in the Scenario 3), then 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.

We would like to note that in all these simulations it is assumed that the overall space of potential members is so large that the distribution of the joining members is unaffected by the leaving members of the group, which is practically equivalent to the case where leaving members never return. For example, in Scenario 3 and 4, the improvement figures are relatively low, mainly because even the most dynamic member causes a single rekey at most. 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 significant gains even when all rekey operations are due to leaving members as in Scenario 3 and 4.

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

9 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. [23] but are also applicable to the more sophisticated LKH-based techniques such as the OFT scheme of McGrew and Sherman [15] 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 8.

The requirement of using actual probabilities for tree organization can be a major limitation for probabilistic LKH organization techniques. Instead of using the actual probabilities, we suggest using a heuristic weight assignment technique, which is described in Section 7. Simulations summarized in Section 8, 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, 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] R. G. Gallager. Variations on a theme by Huffman. *IEEE Trans. Information Theory*, 24:668–674, 1978.

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

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

[11] D. Huffman. A method for the construction of minimum redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

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

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

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

[15] 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.

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

[17] 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.

[18] 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.

[19] Ali Aydın Selçuk and Deepinder Sidhu. Probabilistic methods in multicast key management. In *Information Security Workshop 2000*, pages 179–193. Springer-Verlag, Wollongong, Australia, December 2000.

[20] C. E. Shannon. A mathematical theory of communication. *Bell Systems Technical Journal*, 27:329–423, 1948.

[21] 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.

[22] Jeffrey S. Vitter. Design and analysis of dynamic Huffman codes. *Journal of the ACM*, 34:825–845, 1987.

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

[24] 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.

A Simulation Results in Detail

The following tables summarize the simulation results for different values of the test parameters. The presented results are the averages calculated over 100 randomly generated simulation runs for every data point. I_1 and I_2 denote $Insert_1$ and $Insert_2$ respectively.

Scenario 1:

(a) $n = 10^2$

	$c_T = 0.5$		$c_T = 1$		$c_T = 2$		$c_T = 4$	
c_σ	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

(b) $n = 10^3$

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

(c) $n = 10^4$

0.1	.99	.99	.99	.99	.99	.99	.99	.99
0.5	.69	.67	.71	.70	.75	.74	.73	.73
1.0	.62	.61	.67	.67	.69	.68	.69	.69

Scenario 2:

(a) $T\lambda_A = 10^2$

	$c_T = 0.5$		$c_T = 1$		$c_T = 2$		$c_T = 4$	
c_σ	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

(b) $T\lambda_A = 10^3$

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

(c) $T\lambda_A = 10^4$

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	.63	.65	.64	.64	.63

Scenario 3:

(a) $T\lambda_A = 10^2$

	$c_T = 0.5$		$c_T = 1$		$c_T = 2$		$c_T = 4$	
c_σ	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

(b) $T\lambda_A = 10^3$

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

(c) $T\lambda_A = 10^4$

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

Scenario 4:

	$\mu_S\lambda_A = 10^2$		$\mu_S\lambda_A = 10^3$		$\mu_S\lambda_A = 10^4$	
c_σ	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