EHBT: An efficient protocol for group key management*

Sandro Rafaeli, Laurent Mathy, and David Hutchison

Computing Department, Lancaster University, LA1 4YR, Lancaster, UK

Abstract. Several protocols have been proposed to deal with the group key management problem. The most promising are those based on hierarchical binary trees. A hierarchical binary tree of keys reduces the size of the rekey messages, reducing also the storage and processing requirements. In this paper, we describe a new efficient hierarchical binary tree (EHBT) protocol. Using EHBT, a group manager can use keys already in the tree to derive new keys. Using previously known keys saves information to be transmitted to members when a membership change occurs and new keys have to be created or updated. EHBT can achieve $(I \cdot \log_2 n)$ message size (I is the size of a key index) for join operations and $(K \cdot \log_2 n)$ message size (K is the size of a key) for leave operations. We also show that the EHBT protocol does not increase the storage and processing requirements when compared to other HBT schemes.

1 Introduction

With IP multicast communication, a group message is transmitted to all members of the group. Efficiency is clearly achieved as only one transmission is needed to reach all members. The problems start because any machine can join a multicast group and start receiving the messages sent to the group without the sender's knowledge. This characteristic raises concerns about privacy and security since not every sender wants to allow everyone to have access to its communication.

Cryptographic tools can be used to protect group communication. An encryption algorithm takes input data (e.g. a group message) and performs some transformations on it using a key (where the key is a randomly generated number). This process generates a ciphered message. There is no easy way to recover the original message from the ciphered text other than by knowing the key [9].

When applying such technique, it is possible to run secure multicast sessions. Group messages are protected by encryption using a chosen key (*group key*). Only those who know the group key are able to recover the original message. However, distributing the group key to valid members is a complex problem. Although rekeying a group before the join of a new member is trivial (send the new group key to the old group members encrypted with the old group key),

* The work presented here was done within the context of ShopAware - a research project funded by the European Union in the Framework V IST Programme.

rekeying the group after a member leaves is far more complicated. The old key cannot be used to distribute a new one, because the leaving member knows the old key. A group manager must, therefore, provide other scalable mechanisms to rekey the group.

Several researchers have studied the use of a hierarchical binary tree (HBT) for the group key management problem. Using an HBT, the key distribution centre (KDC) maintains a tree of keys, where the internal nodes of the tree hold key encryption keys (KEKs) and the leaves correspond to group members. Each leaf holds a KEK associated to that one member. Each member receives and maintains a copy of the KEK associated to its leaf and the KEKs correspondent to each ancestor node in the path from its parent node to the root. All group members share key held by the root of the tree. For a balanced tree, each member stores $\log_2 n + 1$ keys, where n is the number of members. This hierarchy is explored to achieve better performance when updating keys.

In this paper, we propose a protocol to efficiently built an HBT, which we call the EHBT protocol. The EHBT protocol achieves $(I \cdot \log_2 n)$ message size for addition operations and $(K \cdot \log_2 n)$ message size for removal operations keeping the storage and processing on both, client and server sides to a minimum. We achieve these bounds using well-known techniques, such as a one–way function and the *xor* operator.

2 Related Work

Wallner et al [13] were the first to propose the use of an HBT. In their approach, every time the group membership changes, internal node keys (affected by the membership change) are updated and every new key is encrypted with each of its children's keys and then multicast. A rekey message conveys $2 \cdot \log_2 n$ keys for including or removing a member.

Caronni et al [12] proposed a very similar protocol to that of Wallner, but they achieve a better performance regarding the size of multicast messages for joining operations. We refer to this protocol as HBT+. Instead of encrypting new key values with their respective children's key, Caronni proposes to pass those keys into a one–way function. Only the indexes of the refreshed keys need to be multicast and an index size is smaller than the key size.

An improvement to the hierarchical binary tree approach is the one–way function tree (OFT) proposed by McGrew and Sherman [5]. The keys of a node's children are *blinded* using a one–way function and then mixed together using the xor operator. The result of this mixing is the KEK held by the node. The improvement is due to the fact that when the key of a node changes, its blinded version is only encrypted with the key of its sibling node. Thus, the rekey message carries just $\log_2 n$ keys.

Canetti et al [3] proposed a slightly different approach that achieves the same communication overhead. Their scheme uses a pseudo–random–generator (PRG) [9] to generate the new KEKs rather than a one–way function and it is applied only on user removal.

Perrig et al proposed the efficient large–group key (ELK) protocol [6]. The ELK protocol is very similar to the OFT, but ELK uses pseudo–random functions (PRFs)[1] to build and manipulate the keys in the tree. ELK employs a timely rekey, hence, at every time interval, the KDC refreshes the root key using the PRF function and then uses it to update the whole key tree. By deriving all keys, ELK does not require any multicast messages during a join operation. All members can refresh their own keys, hence no rekey message is required. When members are deleted, as in OFT, new keys are generated from both its children's keys.

3 Efficient Hierarchical Binary Tree Protocol

In the EHBT protocol, a KDC maintains a tree of keys. The internal nodes of the tree hold KEKs and the leaves correspond to group members. Keys are indexed by randomly chosen numbers. Each leaf holds a secret key that is associated to that member. The root of the tree holds a common key to all members.

Ancestors of a node are those nodes in the path from its parent node to the root. The set of ancestor of a node is called *ancestor set*. Each member knows only its own key (associated to its leaf node) and keys correspondent to each node in its *ancestor set*. For a balanced tree, each member stores $\log_2 n + 1$ keys, where n is the number of members.

In order to guarantee backward and forward secrecy [11], the keys related to joining members or leaving members should be changed every time the group membership changes. The new keys in the *ancestor set* of an affected leaf are generated upwards from the key held by the affected leaf's sibling up to the root. Using keys that are already in the tree can save information to be transmitted to members when a membership occurs and new keys have to be created or updated.

The formula $\mathcal{F}(x, y) = h(x \oplus y)$ is used to generate keys from other keys, where h is a one–way hash function and $\oplus$ is a normal xor operator. The obvious functionality of function h is to *hide* the original value of x and y into value z in a way that if one knows only z he cannot find the original values x and y. The functionality of $\oplus$ is to mix x and y and generate a new value.

We say that a key k_i can be *refreshed* by doing $k'_i = \mathcal{F}(k_i, i)$, where i is the index (or identifier) of key k_i or key k_i can be *updated* by deriving one of its children key by doing $k'_i = \mathcal{F}(k_i^{left|right}, i)$, where $k_i^{left|right}$ is the key of i's either left or right child. Appendix A describes the reason for using index i in function $\mathcal{F}$.

3.1 Rekey Message Format

A member can receive two types of information in a rekey message, one telling him to refresh or update the value of a key, the other telling him the new value

[1] ELK uses the stream cipher RC5 [8] as the PRF.

of a key. In the former case, the member receives an *id* and in the latter case, he receives a key value. After deriving a key, a member will try to derive all other keys by himself (from that key up to the root) unless he receives another information telling him something different. For example, if key K_i is refreshed, the KDC needs to send to K's holders the identification of the key so that they can perform the refresh operation themselves. Or, if a node n has its key updated ($K'_n = \mathcal{F}(k_L, n)$), then it implies sending to member L the index n and to the other child, namely R, the new key value K'_n (because R does not know L's key).

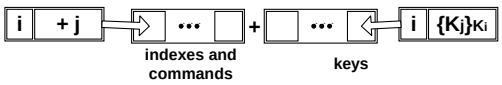

Fig. 1. Example of a rekey message.

The rekey message that relays this information has two parts. The first part carries commands and the second carries keys. Each piece of information is indexed by a key index. Keys are encrypted with the key indicated by the key index (see Figure 1), but commands are not encrypted because they do not carry vital information. Based on commands and keys, members can find out which keys they must refresh or update, or just substitute, because they have received a new key value to a specific key.

Algorithm 1: Reading rekey message algorithm.

```
(1)   receive rekey message
(2)   set last command to "keep key"
(3)   while there is a key to be derived
(4)       get a key index from key–list
(5)       search indexes part of rekey message for key index
(6)       if there is a command
(7)           execute the command on the specific key
(8)           set last command to this command
(9)       else
(10)          search keys part of the rekey message for key index
(11)          if there is a key
(12)              substitute it in the key list
(13)              set last command to "update"
(14)      if there is no command or key
(15)          execute last command in current key
```

The algorithm to handle rekey messages starts with a member holding a list of known keys (key–list). After executing the algorithm, a member will have all his keys freshened up. A simplified version of this algorithm appears in Algorithm 1.

In the remainder of this paper, we use the following notation:

$+i$ or $-i$ or $*i$	are commands to be applied on key i
$\mathcal{R}(k_i)$	refresh k_i applying $\mathcal{F}(k_i, i)$
$\mathcal{U}(k_i)$	update k_j applying $\mathcal{F}(k_i, j)$
$\{x\}_k$	encryption of x with k
$j : command$	command to key j's holder
$[commands, keys]$	message containing commands and keys

4 Basic Operations

In this section, we describe the basic algorithms for join and leave operations for single and multiple cases.

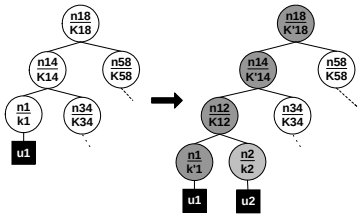

Fig. 2. User u_2 joins the tree.

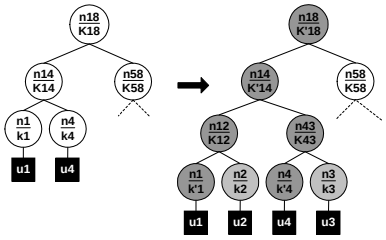

Fig. 3. Users u_2 and u_3 join the tree.

Single Member Join Algorithm. When a member joins the group, it is associated to a leaf node n. The KDC assigns a randomly chosen key k_n to n. Leaf n is then included in the tree at the parent of the shallowest leaf node s (to keep the tree as short as possible). Leaf s is removed from the tree, and in its place a new node p is inserted. Leaves s and n are inserted as p's children. We see an example in Figure 2: Member 2 is placed in leaf n_2, which is inserted at node n_{12}. Node n_{12} becomes the new parent of leaves n_1 and n_2. Leaf n_2 is assigned key k_2.

In order to keep the backward secrecy, keys in n_1's *ancestor set* need to receive new values. Key k_1 is refreshed ($k'_1 = \mathcal{R}(k_1)$), K_{12} receives a value based on k'_1 ($K_{12} = \mathcal{U}(k'_1)$) and keys K_{14} and K_{18} are refreshed ($K'_{14} = \mathcal{R}(K_{14})$ and $K'_{18} = \mathcal{R}(K_{18})$).

Note that during a join operation, keys, which were already in the tree, are just refreshed. Members holding those keys only need to be told those keys' indexes to be able to generate their new values, which means that these keys do not have to be transmitted. In the same way, members that had their keys used for generating new keys just have to be told the index of the new key and they can generate that key by themselves.

The KDC generates unicast messages for member n_2 ($[k_2, K_{12}, K'_{14}, K'_{18}]$) and member n_1 ($[+12]$), and multicast message $[14 : *14, 18 : *18]$.

Member u_2 receives its unicast message and creates its key–list. Member u_1 receives its unicast message and derives key K_{12}, including it in its key–list. Members holding keys K_{14} and K_{18} refresh these keys.

Multiple Members Join Algorithm. Several new members are inserted in the tree as in the single member join algorithm. They are associated to nodes and the nodes are placed at the parent of the shallowest leaves. However, the keys in the tree are modified in a slightly different manner. New nodes' *ancestor sets* converge at some point and all keys that are in more than one *ancestor set* are modified only once.

See Figure 3 for an example. Members u_2 and u_3 joined the group and have been placed at nodes n_{12} and n_{43}, respectively. Following the single member join algorithm, the keys in member u_2's *ancestor set* are changed: first, $k'_1 = \mathcal{R}(k_1)$, and then, $K_{12} = \mathcal{U}(k'_1)$, $K_{14} = \mathcal{R}(K_{14})$, $K'_{18} = \mathcal{R}(K_{18})$. In the same way, keys in member u_3's *ancestor set* are changed: first, $k'_4 = \mathcal{R}(k_4)$, and then, $K_{43} = \mathcal{U}(k'_4)$. Keys K_{12} and K_{18} have already been changed because of member u_2, hence they are not changed again.

The KDC generates unicast messages for member n_2 ($[k_2, K_{12}, K'_{14}, K'_{18}]$), member n_3 ($[k_3, K_{43}, K'_{14}, K'_{18}]$), member u_1 ($[+12]$) and member u_4 ($[+43]$), and multicast message $[14 : *14, 18 : *18]$.

Members u_2 and u_3 receive their unicast messages and create their respective key–lists. Member u_1 receives the unicast message, derives key K_{12}, and includes it in its key–list. Member u_4 does the same with key K_{43}. Members holding keys K_{14} and K_{18} refresh these keys.

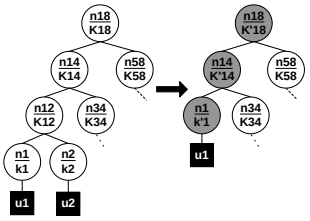

Fig. 4. User u_2 leaves the tree.

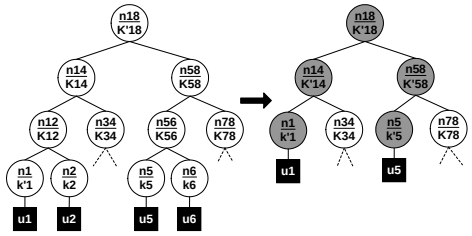

Fig. 5. Users u_2 and u_6 leave the tree.

Single Member Leave Algorithm. When a member u leaves or is removed from the group, its sibling s replaces its parent p. Moreover, all keys known by u should be updated to guarantee forward secrecy. For example, see Figure 4: u_2 leaves (or is removed from) the group and its node is removed from the tree. Node n_{12} is also removed and leaf n_1 is promoted to its place.

In order to keep the forward secrecy, keys in n_1's *ancestor set* need to receive new values. Keys K_{14} and K_{18} have to be updated: $k'_1 = \mathcal{R}(k_1)$, $K'_{14} = \mathcal{U}(k'_1)$ and $K'_{18} = \mathcal{U}(K'_{14})$.

Note that in removal operations, all keys in the removed member's *ancestor set* are updated. Those keys cannot be just refreshed because the removed mem-

ber knows their previous values and could easily calculate the new values. Since the new values are all generated from the removed member's sibling key, which was not known by the removed member, the removed member cannot find the new values.

The KDC generates multicast message $[1:-12, \{K'_{14}\}_{K_{34}}, \{K'_{18}\}_{K_{58}}]$.

Member n_1 refreshes k'_1 and, because it has removed K_{12}, it updates K_{14} and K_{18}. Members holding key K_{34} get new key K'_{14} and then update key K_{18}. Members holding key K_{58} get new key K'_{18}.

Multiple Members Leave Algorithm. This algorithm is handled similarly to the single member leave algorithm. The leaving nodes are removed and the tree shape is adjusted accordingly. As in the multiple join algorithm, there can be several different path from removed nodes to the root, which means that the root key can be updated by several nodes (see Figure 5).

In order to avoid several root key versions for the same operation, the KDC chooses one of the paths and use it to update the root key. For example, in Figure 5, n_2 and n_6 leave the group and nodes n_1 and n_5 are promoted to their respective parents' places (n_{12} and n_{56}). Both are used to derive their new parent keys K'_{14} and K'_{58}, but then they both cannot be used to update key K'_{18}. In this case, the KDC chooses one of them to update key K'_{18} and the other will receive the updated key. For instance, the KDC chooses node n_1 and then the keys are updated as follows: $k'_1 = \mathcal{R}(k_1)$, $K'_{14} = \mathcal{U}(k'_1)$, $K'_{18} = \mathcal{U}(K'_{14})$, $k'_5 = \mathcal{R}(k_5)$ and $K'_{58} = \mathcal{U}(k'_5)$.

The KDC generates multicast message $[1:-12, 5:-56, \{K'_{14}\}_{K_{34}}, \{K'_{58}\}_{K_{78}}, \{K'_{18}\}_{K'_{58}}]$.

Member n_1 refreshes k'_1 and, because it has removed K_{12}, it updates K'_{14} and K'_{18}. Key K_{34}'s holders recover K'_{14} and update K'_{18}. Member n_5 refreshes k'_5 and updates K'_{58}, but since there is a new key encrypted with K'_{58}, n_5 stops updating its keys and just recovers K'_{18}. Key K_{78}'s holders recover K'_{58} and, since there is a key encrypted with it, they just recover K'_{18}.

Rebalancing. The efficiency of the key tree depends crucially on whether the tree remains balanced or not. A tree is said to be balanced if no leaf is much further away from the root than any other leaf. In general, for a balanced binary tree with n leaves, the distance from the root to any leaf is $\log_2 n$, but if the tree is unbalanced, the distance from the root to a leaf can become as high as n. Therefore, it is desirable to keep a key tree as balanced as possible.

The rebalancing works by getting the shallowest and deepest internal nodes and comparing their depths. If the depth gap is larger than two then it means that the tree is unbalanced and needs to be levelled. For balancing the tree, the deepest leaf node is removed, which makes its sibling to go one level up (similarly to the removing algorithm), and inserted at the shallowest node (similarly to the inserting algorithm). This procedure is repeated until the difference between the depths of the shallowest and the deepest nodes is smaller than two.

In a rebalancing operation, the deepest node, which has been moved from one position in the tree to another, requires that its old keys need to be updated (as in a deletion operation) and it needs to have access to the keys in its new

path to the root (as in an insertion operation). Therefore, an insertion and a deletion are performed simultaneously.

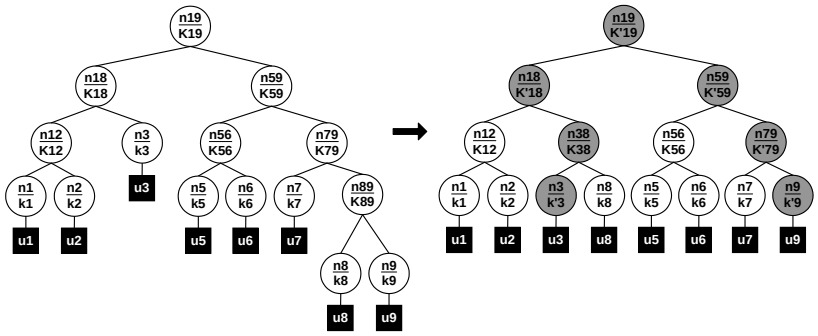

Fig. 6. Rebalancing the tree.

See Figure 6 for an example. The tree needs a rebalancing, so leaf n_8 is deleted from its original position (n_{89}) and inserted into a new position (n_{38}). The deletion starts a removal operation with leaf n_9 updating the new keys. At the same time, leaf n_3 starts refreshing the keys on its path (as an insertion requires). The new keys are calculated as follows: $k'_9 = \mathcal{R}(k_9)$, $K'_{79} = \mathcal{U}(k'_9)$, $K'_{59} = \mathcal{U}(K'_{79})$, $k'_3 = \mathcal{R}(k_3)$, $K_{38} = \mathcal{U}(k'_3)$ and $K'_{18} = \mathcal{R}(K'_{18})$. Key K'_{19} does not need to be changed.

The KDC generates unicast messages for member n_8 ($[K_{38}, K'_{18}]$ and member u_3 ($[+38]$), and multicast message $[9:-89, 18:*18, \{K'_{79}\}_{k_7}, \{K'_{59}\}_{K_{56}}]$.

Member n_8 deletes all its known keys and replaces them by those just received. Member n_9 updates its keys. Members n_7 and key k_{56}'s holders extract their parts and update their keys. Member n_3 derives K'_{38}. Key K_{18}'s holders refresh K'_{18}.

5 Evaluation

In this section, we compare the properties of the EHBT algorithm with the other algorithms introduced in section 2: PRGT[2] (Canetti et al.), HBT+ (Caronni et al), OFT (McGrew and Sherman) and ELK (Perrig). We focus our criteria on KDC computation, joined member computation (for insertions), sibling computation (sibling to the joining/leaving member), size of the multicast message, size of the unicast messages and storage at both KDC and members.

The notations used in this section are:

[2] Canetti does not specify the PRG function to use, hence we assume the same RC5 algorithm used in ELK.

n	number of member in the group
d	height of the tree (for a balanced tree $d = \log_2 n$)
I	size of a key index in bits
K	size of a key in bits
G	key generation
H	hash function execution
X	xor operation
E	encryption operation
D	decryption operation

Table 1 summarizes the computation required from the KDC, joined member and sibling to joined member, and message size of joining member's unicast message, sibling's unicast message and multicast message during single join operations.

Table 1. Single join operation equations.

Scheme/	Computation			Message size		
Resource	KDC	Join member	Sib member	Join unicast	Sib unicast	Multicast
EHBT	$G + (d+1)(X+H+E)$	$(d+1)D$	$(d+1)(X+H)$	$(d+1)K$	I	dI
PRGT	$2G + dH + (d+1)E$	$(d+1)D$	$D + dH$	$(d+1)K$	$I+K$	dI
HBT+	$2G + dH + (d+1)E$	$(d+1)D$	$D + dH$	$(d+1)K$	$I+K$	dI
OFT	$G + (d+1)H + dX + 3dE$	$(d+1)D + d(H+X)$	$2D + d(H+X)$	$(d+1)K$	$I+2K$	$(d+1)K$
ELK	$G + (4n-2)E$ and $(d+3)E$	$(d+1)D$	$2dE$ and $2E$	$(d+1)K$	I	0

Table 2 summarises multiple join operation equations. The parameters analysed are the same parameters used in Table 1. The equations are valid for multiple joins when the original number of members is doubled after the mass join, which means that every old member gets a new sibling (a new member) and all the keys in the tree are affected. This represents the worst case possible for join operations. For the sake of the equations in this table, n is the original number of members in the group previously to the mass join, but d is the new height of the tree after the mass join.

Table 2. Multiple join operation equations.

Scheme/	Computation			Message size		
Resource	KDC	Join member	Sib member	Join unicast	Sib unicast	Multicast
EHBT	$nG + (3n-1)(X+H) + n(d+1)E$	$(d+1)D$	$(d+1)(X+H)$	$n : (d+1)K$	$n : I$	$(n-1)I$
PRGT	$2nG + (n-1)H + n(d+2)E$	$(d+1)D$	$D + dH$	$n : (d+1)K$	$n : I+K$	$(n-1)I$
HBT+	$2nG + (n-1)H + n(d+2)E$	$(d+1)D$	$D + dH$	$n : (d+1)K$	$n : I+K$	$(n-1)I$
OFT	$nG + (4n-2)(H+X) +$ $(nd+5n-1)E$	$(d+1)D +$ $d(H+X)$	$2D +$ $d(H+X)$	$n : (d+1)K$	$n : I+2K$	$(2n-2)K$
ELK	$(8n-2)E$ and $nG + n(d+3)E$	$(d+1)D$	$2dE$ and $2E$	$n : (d+1)K$	$n : I$	0

EHBT requires less computation than the other schemes, but it loses out to ELK when comparing the message sizes. The reason for that is that ELK employs a timed rekey, which means that the tree is completely refreshed at

intervals, despite membership changes, thus only the index of the new parent inserted needs to be sent to the sibling of the joining member. However, this rises two issues: first, at every interval the KDC has to refresh all its 2n-1 keys, which implies unnecessary work for the KDC; second, this scheme does not support rekey on membership changes (regarding join operations). Additionally, ELK imposes some delay on the joining member before he receives the group key.

Table 3. Single leave operation equations.

Scheme/ Resource	Computation		Multicast
	KDC	Sib member	
EHBT	$d(X+H+E)$	$d(X+H)$	$I+dK$
PRGT	$(2d+1)E$	$D+dE$	$I+(d+1)K$
HBT+	$2dE$	dD	$I+2dK$
OFT	$d(H+X+E)$	$D+d(H+X)$	$I+(d+1)K$
ELK	$8dE$	$dD+5dE$	$I+d(n_1+n_2)$

Table 3 summarizes the KDC computation, sibling computation and multicast message size during single leave operations. We also analyse the equations of multiple leave operations, and we show the results in Table 4. For mass leaving, we consider the situation when exactly half of the group members leave the group. The sibling of every leaving member remains in the tree, and hence, all keys in the tree are affected.

Table 4. Multiple leave operation equations.

Scheme/ Resource	Computation		Multicast
	KDC	Sib member	
EHBT	$(2n-1)(X+H)+(n-1)E$	$D+(d+1)(X+H)$	$nI+(n-1)K$
PRGT	$(5n/2-2)E$	$D+dE$	$(3n/2-1)K$
HBT+	$(2n-2)E$	dD	$nI+2(n-1)K$
OFT	$(2n-2)H+(n-1)X+(3n-2)E$	$(d+1)D+d(H+X)$	$nI+(3n-2)K$
ELK	$(7n-3)E$	$dD+5dE$	$nI+(n-1)(n_1+n_2)$

For leaving operations, again EHBT achieves better results than the other schemes regarding the computations involved, but loses out to ELK when comparing the multicast message size. ELK has a slightly smaller multicast message than EHBT, because it sacrifices security. ELK uses only n_1+n_2 bits of a total K possible bits for generating a new key and this procedure weakens that key, Consequently, an expelled member needs to compute only $2^{n_1+n_2}$ possibilities to recover the new key. In EHBT, however, an expelled member needs to compute the full 2^K operations to brute-force the new key.

We have simulated a group with 8192 members. For the calculations of the multiple join operations, we doubled the size of the group to 16384 members, and then we removed all joining members and finished with the 8192 original members. We measured encryption and decryption times for the RC5 algorithm,

MD5 hash function and *xor* operation. We used 16-bit keys for the calculations. We used Java version 1.3 and IAIK [4] cryptographic toolkit on a 850Mhz Mobile Pentium III processor. It takes $1.72 \cdot 10^{-2}$ ms for RC5 to encrypt a 16-bit key with a 16-bit key, and $1.73 \cdot 10^{-2}$ ms to decrypt it. Hashing a 16-bit key takes $4.95 \cdot 10^{-3}$ ms and *xoring* it takes $1.59 \cdot 10^{-3}$ ms. Finally, generating a 16-bit keys takes $7.33 \cdot 10^{-}3$. Applying these numbers into Tables 2 and 4 produces the results in Table 5 that show that EHBT in general is faster to compute than the other protocols.

Table 5. Time in milliseconds for multiple joins and leaves.

Scheme/	Multiple Join			Multiple Leave	
Resource	KDC	Join member	Sib Member	KDC	Sib member
EHBT	2334	0.25	0.09	248.03	0.10
PRGT	2415	0.25	0.08	352.22	0.24
HBT+	2415	0.25	0.08	281.77	0.22
OFT	2951	0.35	0.12	516.78	0.32
ELK	1140 + 2455	0.25	0.48 + 0.03	1105.46	1.34

Finally, EHBT and the other schemes require the KDC to store $2n - 1$ keys and members to store $d + 1$ keys.

6 Security Considerations

The security of the EHBT protocol relies on the cryptographic properties of the h function. One–way hash functions, unfortunately, are not proven secure [2]; nevertheless, for the time being, there has not been any successful attack on either the full MD5 [7] or SHA [1] algorithms [10].

Taking into account the use of hash functions as function h, attacks on the hidden key are limited to brute-force attack. Such an attack can take 2^n hashes to find the original key, with n being the number of bits of the original key used as input.

In order to guarantee backward secrecy and forward secrecy, every time there is a membership change, the keys related to joining members or leaving members are changed.

When a member is added to the tree, all keys held by nodes in its *ancestor set* are changed to avoid giving the new member access to past information. For example, see Figure 2, when member n_2 is inserted in the tree, key K_{12} is created and keys K'_{14} and K'_{18} are refreshed. Node n_2 does not have access to the old values, because it only receives the new key values, which were hidden by the hash function, and assuming the hash function is secure, n_2 has no other way to recover the old key but brute-forcing it. The same rule applies when n_2 leaves; key K_{12} is deleted from the tree and keys K'_{14} and K'_{18} are updated and since n_2 does not have access to their new values it does no longer has access to the group communication.

7 Conclusion

Using one–way hash functions and *xor* operations, we constructed an efficient HBT protocol that achieves better overall performance than other HBT protocols. Our protocol, called EHBT, requires only $(I \cdot \log_2 n)$ message size for join operations and $(K \cdot \log_2 n)$ message size for leaving operations. Additionally, EHBT requires the same key storage as other HBT protocols, and it requires much less computation to rekey the tree after membership changes.

References

1. N. F. P. 180-1. Secure Hash Standard. National Institute of Standards and Technology, U.S. Department of Commerce, DRAFT, May 1994.
2. S. Bakhtiari, R. Safavi-Naini, and J. Pieprzyk. Cryptographic Hash Functions: A Survey. Technical Report 95-09, University of Wollongong, July 1995.
3. R. Canetti, J. Garay, G. Itkis, D. Micciancio, M. Naorr, and B. Pinkas. Multicast Security: A Taxonomy and Some Efficient Constructions. In *Proc. of INFOCOM 99*, volume 2, pages 708–716, New Yok, NY, USA, March 1999.
4. I.-J. Group. IAIK, java–crypto toolkit. Web site at http://jcewww.iaik.tu-graz.ac.at/index.htm.
5. D. A. McGrew and A. T. Sherman. Key Establishment in Large Dynamic Groups Using One-Way Function Trees. Technical Report No. 0755, TIS Labs at Network Associates, Inc., Glenwood, MD, May 1998.
6. A. Perrig, D. Song, and J. D. Tygar. ELK, a New Protocol for Efficient Large-Group Key Distribution. In *2001 IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2001.
7. R. Rivest. The MD5 Message-Digest Algorithm. RFC 1321, April 1992.
8. R. Rivest. The RC5 encryption algorithm. In *Fast Software Encryption, 2^{nd} Int. Workshop*, LNCS 1008, pages 86–96. Springer-Verslag, December 1995.
9. B. Schneier. *Applied Cryptography Second Edition: protocols, algorithms, and source code in C*. John Wiley & Sons, Inc., 1996. ISBN 0-471-11709-9.
10. W. Stallings. *Cryptography and Network Security*. Prentice–Hall, 1998. ISBN 0-138-69017-0.
11. M. Steiner, G. Taudik, and M. Waidner. Cliques: A new approach to group key agreement. Technical Report RZ 2984, IBM Research, December 1997.
12. M. Waldvogel, G. Caronni, D. Sun, N. Weiler, and B. Plattner. The VersaKey Framework: Versatile Group Key Management. *IEEE Journal on Selected Areas in Communications (Special Issue on Middleware)*, 17(9):1614–1631, August 1999.
13. D. Wallner, E. Harder, and R. Agee. Key Management for Multicast: Issues and Architectures. RFC 2627, June 1999.

A Reasoning on Using Index i in Function $\mathcal{F}$

Index i is included in the formula $\mathcal{F}$ to avoid giving the possibility for members to have access to keys that they are not meant to. For example, removing member n_2 in Figure 4, means new keys $k'_1 = \mathcal{U}(k_1)$, $k'_{14} = \mathcal{R}(k'_1)$ and $k'_{18} = \mathcal{R}(k'_{14})$.

If, immediately after member n_2 has left the group, member n_0 joins it and is inserted as a sibling of n_1, then it means new keys $k''_1 = \mathcal{U}(k'_1)$, $k_{10} = \mathcal{R}(k''_1)$ (new node n_{10}), $k''_{14} = \mathcal{U}(k'_{14})$ and $k''_{18} = \mathcal{U}(k'_{18})$.

If we remove i from function $\mathcal{F}$ and instead only apply a simple hash h to update keys then the keys from the removal above become $k'_1 = h(k_1)$, $k'_{14} = h(k'_1)$ (or $h(h(k_1))$) and $k'_{18} = h(k'_{14})$ (or $h(h(h(k_1)))$) and the keys from the join become $k''_1 = h(k'_1)$ (or $h(h(k_1))$), $k_{10} = h(k''_1)$ (or $h(h(h(k_1)))$), $k''_{14} = h(k'_{14})$ and $k''_{18} = h(k'_{18})$. As one can see, key k_{10} and k'_{18} are identical, which means that member n_0 can have access to past messages encrypted with k'_{18} (or k_{10}).