CHAPTER 6

OPTIMIZING THE REKEYING COST FOR CONTRIBUTORY GROUP KEY AGREEMENT SCHEMES

6. 1 INTRODUCTION

Although a contributory group key agreement is a promising solution to achieve access control in collaborative and dynamic group applications, the existing schemes have not achieved the performance lower bound in terms of time, communication, and computation costs. In this chapter, we propose a contributory group key agreement that achieves the performance lower bound by utilizing a novel logical key tree structure, called PFMH, and the concept of phantom user position. In particular, the proposed scheme only needs $O(1)$ rounds of the two-party Diffie-Hellman (DH) upon any single-user join event and $O(\log n)$ rounds of the two-party DH upon any single-user leave event. Both the theoretical bound analysis and simulation show that the proposed scheme achieves a lower rekeying cost than the existing tree-based contributory group key agreement schemes.

6.1.1 Need for the Algorithm

One fundamental challenge in securing group multicast applications is to achieve access control such that only authorized group members can access group communications. Group access control is usually achieved by encrypting data using a group key that is shared among all legitimate group members. The issues of establishing and updating group keys are addressed
by various group key agreement schemes as discussed earlier in chapter 2. Since many practical group applications do not prefer utilizing centralized key servers, contributory solutions of key agreement have drawn extensive attention, especially for applications where centralized administration and pair-wise secure channels are not applicable.

6.1.2 Contributory Group Key Agreement

In contributory group key agreement schemes, all the group members contribute their share and compute the group key collaboratively; the group key is generated as a (usually one-way) function of individual contributions from all the group members. Establishing and updating the group key in large dynamic groups often consumes a considerable amount of computation and communication resources. For large-scale dynamic group applications where group members do not have ample communication and computation capability, such as in some mobile ad hoc and sensor networks, the bottleneck of utilizing contributory key agreement schemes for access control will be their cost efficiency.

6.2 Existing Work: TGDH and DST

The early design of contributory group key agreement schemes focussed mostly on the efficiency of initial group key establishment. These schemes, however, encounter a high rekeying cost upon group membership changes. Later, Steiner et al (2000) proposed a family of Group Diffie-Hellman (GDH) protocols by extending the two-party Diffie-Hellman (DH) protocol to the group scenarios. The GDH protocols achieve an efficient key update upon user join but still require a high cost for member leave. Recently, logical key tree structures are used to improve the scalability of contributory key agreements. Kim et al (2004) proposed a tree-based contributory group
key agreement protocol called the Tree-based Group Diffie-Hellman (TGDH), where a binary balanced tree is adopted to maintain the keying material. In TGDH, the group key can be updated by performing \( \log n \) rounds of the two-party DH upon any single-user join or leave, where \( n \) is the group size. Mao et al (2006) proposed another tree-based contributory key agreement scheme called the Dynamic SubTree (DST). By using a special join-tree/exit-tree topology and exploiting cost amortization, the DST can reduce the average time cost to \( O(\log \log n) \) rounds of the two-party DH for single-user join or leave. However, the DST has an unrealistic requirement that members know other members’ leave time in advance. When the members’ leave time is not known, the time cost upon single-user leave is \( O(\log n + \log \log n) \), which is higher than that of TGDH.

For any of the above tree-based contributory group key management schemes, the lower bound of the worst-case cost is \( O(\log n) \) rounds of the two-party DH for either user addition or deletion. That is, either the cost for adding a user or the cost for deleting a user is no less than \( O(\log n) \). In addition, it is obvious that at least one round of the two party DH needs to be performed for adding or deleting a user in any circumstance. Therefore, the lowest possible cost for contributory key agreement is \( O(\log n) \) for user join and \( O(1) \) for user leave or \( O(\log n) \) for user leave and \( O(1) \) for user join. Both the TGDH and DST do not achieve these lower bounds.

To achieve the lower bound of the rekeying cost, in this chapter, we propose a novel and efficient logical key tree structure, called the PFMH tree, as well as a cost minimizing PFMH tree-based contributory group key agreement (PACK) protocol suite that handles dynamic group membership events. Both the theoretical analysis and simulation studies show that PACK has a much lower rekeying cost than the existing tree-based contributory group key agreements.
6.3 PERFORMANCE MEASURES

In this section, we briefly introduce the security requirements of the contributory key agreement, the performance measures, and the implementation cost of the DH protocol between two groups.

Group key management schemes must be able to adjust group secrets subsequent to membership changes, including single-user addition, single-user deletion, group merge, and group partition. Security requirements with dynamic membership include group key secrecy, forward secrecy, backward secrecy, and key independence. The overhead of group key agreement involves the computation cost, communication cost, and time cost. Since most of the existing contributory key agreement schemes use the two-party DH protocol as a basic building module, the computation cost comes mainly from the cryptographic primitives that are needed to perform the two-party DH, such as modular exponentiation, and the communication cost comes from sending and receiving rekeying messages. The time cost is used to describe the latency in group key establishing and updating. In contributory group key agreement, by exploiting the possible parallelism when performing group key establishing and updating, the time cost can be significantly reduced.

6.3.1 Two Party DH

In most tree-based contributory group key agreement schemes the two-party DH is implemented between two groups. The two-group DH can be implemented as follows: Each subgroup elects one member as its delegate, which will compute and send its blind subgroup key to all the members of the other subgroup. Using that blinded key and its own subgroup key they
generate the group key. The blind key is generated by performing the modular exponentiation operation; that is,

\[ f(K) = g^k \mod p \]

where \( g \) is the exponential base and \( p \) is the modular base. Let \( A \) and \( B \) denote two subgroups, where the users in \( A \) share a common group key \( K_A \), and the users in \( B \) share a common group key \( K_B \). To perform the two-group DH between these two subgroups, \( A_1 \) and \( B_1 \) need to exchange the following keying messages: \( A_1 \) sends the blinded key \( f(K_A) \) to all the members of the subgroup \( B \), and \( B_1 \) sends the blinded key \( f(K_B) \) to all the members of the subgroup \( A \). Each member in \( A \) or \( B \) then calculates the new group key \( K_{AB} \) as follows:

\[ K = (f(K_B))^K_A \mod p = (f(K_A))^K_B \mod p \]

In this implementation, each member needs at least one modular exponentiation operation to calculate the new group key. If a delegate does not know its own subgroup’s blinded key, one extra modular exponentiation operation is also needed to calculate the blinded key. For the communication cost, each delegate needs to send a keying message to all the members in the other subgroup. In this chapter, we use \( C_{\text{cast}}(n,l) \) to denote the communication cost needed to send a message with length \( l \) to \( n \) nodes and use \( C_{\text{me}} \) to denote the computation cost of a modular exponentiation operation. Thus, for each round of the two-group DH with the size of subgroups being \( n_1 \) and \( n_2 \) and the keying message length being \( l \), the communication cost is \( C_{\text{cast}}(n_1,l) + C_{\text{cast}}(n_2,l) \), and the computation cost is no more than \( (n_1+n_2+2)C_{\text{me}} \). It is worth noting that sending a message to \( n \) nodes can be implemented in many ways. It can either be implemented
through multicast communication, which we refer to as multicast-n, or be implemented through unicast, which we refer to as unicast-n. In general, the communication cost of a multicast-n operation is not the same as the communication cost of a unicast-n operation. The former usually incurs less communication cost than the latter. Further, the gap between the communication cost of a multicast-n operation and the communication cost of a unicast-n operation may vary according to the underlying network architectures. For example, in wireless networks, the gap is usually very obvious due to the broadcast nature of wireless media, whereas in wired networks without link-level multicast support, the gap is usually not that obvious.

In this chapter, when analyzing the communication cost of sending a message to n nodes, both terms (multicast-n and unicast-n) will be used. Although the communication costs of multicast-n1 and multicast-n2 with n1 ≠ n2 are usually different, to simplify our illustration, in this chapter, we will not distinguish them. Let \( C_{\text{multicast}}(l) \) denote the communication cost of a multicast-n operation, and let \( C_{\text{unicast}}(l) \) denote the communication cost of a unicast-1 operation, where \( l \) is the length of the message to be sent. Further, when performing the two-group DH between two subgroups, the only messages exchanged are their blinded keys. Since, in general, all blinded keys have the same length, without loss of generality; the message length \( l \) will not be explicitly stated. Besides exchanging blinded keys, a user may also need to send messages to all the group members when it wants to join or leave a group. In this chapter, we use \( C_{\text{broadcast}}(l) \) to denote the communication cost incurred by broadcasting a message with length \( l \) to all the group members.
6.4 PFMH KEY TREE STRUCTURE AND BASIC PROCEDURES

6.4.1 General Key Tree Structure

In tree-based contributory group key agreement schemes, keys are organized in a logical tree structure, referred to as the key tree. In a key tree, the root node represents the group key, the leaf nodes represent the members’ private keys, and each intermediate node corresponds to a subgroup key shared by all the members (leaf nodes) under this node. The key of each non-leaf node is generated by performing the two-party DH between the two subgroups represented by its two children, where each child represents the subgroup including all the members (leaf nodes) under this node. Since the two-group DH is used, the key tree is binary. For each node in the key tree, the key path denotes the path from this node to the root, and the copath denotes the sequence of siblings of each node on its key path.

In a key tree, the root node represents the group key, the leaf nodes represent the members’ private keys, and each intermediate node corresponds to a subgroup key shared by all the members (leaf nodes) under this node. The key of each non-leaf node is generated by performing the two-party DH between the two subgroups represented by its two children where each child represents the subgroup including all the members under this node.

Figure 6.1 A Simple Key Tree Example
Figure 6.1 shows a simple key tree example with six members, where \( M_i \) denotes the \( i \)th group member, and \( (l,v) \) denotes the \( v \)th node at level \( l \) of the tree. For example, for member \( M_2 \), its key path is the sequence of nodes \{ (3, 1), (2, 0), (1, 0), (0, 0) \}, and its copath is the sequence of nodes \{ (3, 0), (2, 1), (1, 1) \}.

For a node to be able to calculate the group key, it only needs to know its own keys and all the blinded keys on its copath. For example, as shown in Figure 6.1, \( M_2 \) only needs to know its own key and the blinded keys represented by the nodes \( (3, 0), (2, 1), \) and \( (1, 1) \) in order to calculate the group key. A leaving user can leave from an arbitrary position in the key tree. In fact, for user leave, when group members have similar computation and communication capability, the best tree structure that reduces the worst-case rekeying overhead is a balanced key tree structure. When using a balanced key tree structure, as in TGDH, the worst-case rekeying time cost for both user leave and user join is \( O(\log n) \). In order to reduce the rekeying time cost for user join, one way is to always insert the joining user at the root of the key tree and, consequently, the rekeying time cost for single-user join becomes \( O(1) \). However, such a scheme may result in an extremely unbalanced key tree structure and increase the rekeying cost for user leave to \( O(n) \).

### 6.4.2 PFMH Tree Structure

In order to achieve the lower bound for both user join and leave simultaneously, in this chapter, we propose a novel and efficient key tree structure, which we refer to as the PFMH tree. PFMH tree is a combination of two special key tree structures: partially full (PF) key tree and maximum height (MH) key tree.

- PF key tree: Let \( T \) be a binary key tree of size \( n \), and let \( n' = 2^\lceil \log n \rceil \). \( T \) is a PF key tree if and only if it satisfies one of the following properties: 1) \( T \) is a full key tree, and 2) the left
subtree of $T$ is a full key tree with size $n'$, and the right subtree of $T$ is a PF key tree with size $n-n'$. The full key tree is a fully balanced binary key tree. Figure 6.2 shows an example.

![Figure 6.2 PF Key Tree](image1)

- MH key tree: A key tree $T$ of size $n$ is a MH key tree if and only if it satisfies one of the following properties: 1) $n=1$, and $T$ is a tree with only one leaf node. 2) The right subtree of $T$ is a leaf node, and the left subtree of $T$ is an MH key tree with size $n-1$. Figure 6.3 shows an example of an MH Tree.

![Figure 6.3 MH Tree](image2)
PFMH key tree: A key tree T of size n is a PFMH key tree if and only if it satisfies one of the following properties: 1) T is a PF key tree. 2) The left subtree of T is a PF tree, and the right subtree of T is an MH tree. Figure 6.4 illustrates the special key tree structure PFMH.

![Figure 6.4 PFMH Key Tree](image)

According to the above definitions, we can see that the height of a PF key tree with size n is \(\lceil \log n \rceil\), the height of an MH tree with size n is \(n-1\). In this chapter, given a PFMH key tree T, we shall use the main tree to refer to the PF subtree of T, denoted by \(T_{\text{main}}\), and use the join tree to refer to the MH subtree of T, denoted by \(T_{\text{join}}\). It is easy to see that the height of \(T_{\text{main}}\) is always bounded by \(\log n\).

Next, we shall describe two basic procedures to manage and update PFMH key trees: unite and split.
6.4.3  Unite Procedure

The unite procedure is used to combine the small trees into a large group and also in a user leave event. The steps involved in the unite procedure:

- Index the full key trees according to their size.
- The larger the size of a subgroup, the lower its index.
- Each subgroup is paired with another subgroup under the following rules:
  - in size the two trees are equal
  - there is no other subgroup with the same size and paired with the largest index subgroup.
- According to the unite procedure, first unite the larger indices tree by including intermediate nodes and performing the two-party DH to generate the subgroup key.
- Two delegates are elected to exchange their blind key, and using that, the new group key is generated.

In the unite procedure, an extra cost will be incurred when performing a sequence of two group DHs to generate the new key tree. Figure 6.5 demonstrates how the key tree is updated when five full key trees are united into a PF key tree.
6.4.4 Split Procedure

The split procedure is used to partition the tree into a set of full key tree with minimum set size. The tree U is split into possible full trees until the tree becomes empty. In the split procedure, each group member (leaf node) only needs to truncate the current key tree maintained by itself, so no communication cost and negligible computation and time costs are needed. Figure 6.6 demonstrates how the key trees are updated when a PFMH key tree is split into a set of full key trees.
6.5 PACK PROTOCOL

PACK is a PFMH Tree-Based Contributory Group Key Agreement protocol. PACK includes a set of rekeying protocols to update the group key upon group membership change events. Compared with the existing tree-based contributory group key agreement schemes, PACK can achieve the minimum rekeying time cost upon membership change events, in the sense that for any single-user join event, the rekeying time cost is of the order \(O(1)\), and for any single-user leave event, the rekeying time cost is of the order \(O(\log n)\). In PACK, each group member knows all the subgroup keys on its key path and knows the ID and the exact location of any other current group member in the key tree. In PACK, when a new user joins the group, it will always be attached to the root of the join tree to achieve \(O(1)\) rekeying cost in terms of computation per user, time, and communication. When a user leaves the current group, and he has a phantom location in the key tree, his share is changed and hence the group key is updated in \(O(\log n)\) rounds, and simultaneously, this reduces the communication and computation costs.

6.5.1 Single-User Join Protocol

When a user M wants to join the group, it initiates the single-user join protocol by broadcasting a request message that contains its member ID, a join request, its own blinded key, some necessary authentication information, and its signature for this request message. After receiving this user join request message, the current group members will check whether M has the privilege to join the group based on certain group access control policies. If M has the authorization to join, the key tree will be updated by incorporating M’s share, and a new group key will be generated in order to incorporate a secret share from M and to guarantee the group keys’ backward secrecy.
Procedure Join (G,M)

T is the PFMH key tree of group G, T_{main} is the main tree of T, T_{join} is the join tree of T.

if (T_{join} is empty) then

A delegate will be elected by group G to perform the two-group DH with M, and a new group key K will be generated. A leaf node will be created to represent M, and a new root node will be created to represent K with its right child being the node representing M and its left child being T_{main}. The node representing M becomes the join tree of the updated key tree.

else

Round 1: A delegate will be elected by group T_{join} to perform the two-group DH with M, and a new subgroup key K_{join} will be generated. A leaf node will be created to represent M, and a new intermediate node will be created for K_{join} with its right child being M and its left child being the old T_{join}.

Round 2: Two delegates will be elected separately by T_{main} and the new join tree to perform two-group DH between them, and a new group key K will be generated. A new root node will be created to represent K with its right child being T_{join} and its left child being T_{main}.

end if

Each current member updates the key tree maintained by itself locally according to the above key tree update procedure, and a delegate sends an updated copy of the key tree to the new joining member M.
In PACK, the rekeying upon single-user join needs to perform at most two rounds of the two-group DH. If the join tree is not empty, a new join tree is generated by performing the two-group DH between the new member and the old join tree, with the left subtree being the old join tree and the right subtree being the node representing the new member. If the join tree is empty, the node representing the new member becomes the join tree. The group key is generated by performing the two-group DH between the new join tree and the main tree.

Figure 6.7 Examples of a Key Tree Update upon a Single-User Join Event

Figure 6.7 shows two examples of a key tree update upon single-user join events. In the first example, the join tree is empty, and the main tree
consists of four members. After the new member $M_5$ joins the group, a new node is created to act as the new root, and the node $(1, 1)$ becomes the new join tree that represents $M_5$. In the second example, when $M_6$ joins the group, at the first round, the two-group DH is first performed between $M_5$ and $M_6$ to generate a new join tree; at the second round, the two-group DH is performed between the new join tree and the main tree to generate a new group key.

Table 6.1 Rekeying Cost on a Single-User Join Event

<table>
<thead>
<tr>
<th>Time cost</th>
<th>Communication cost in term of multicast</th>
<th>Communication cost in term of unicast</th>
<th>Computation cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 1 1</td>
<td>$2C_{\text{multicast}}$</td>
<td>$nC_{\text{unicast}}$</td>
<td>$(n+2)C_{\text{me}}$</td>
</tr>
<tr>
<td>Case 1 2</td>
<td>$4C_{\text{multicast}}$</td>
<td>$(n+</td>
<td>T_{\text{join}}</td>
</tr>
</tbody>
</table>

Table 6.1 lists the rekeying cost on a single-user join event in PACK, where $n$ denotes the total number of leaf nodes in the new group, and $|T_{\text{join}}|$ is the old join tree size.

Case 1 considers the situation that the join tree is empty, and the protocol only needs to perform one round of the two-group DH.

Case 2 considers the situation that the join tree is not empty, and the protocol needs to perform two rounds of the two-group DH. For Case 2, the term $|T_{\text{join}}|+2$ in the computation cost comes from performing the two group DH between the new member and the old join tree, since in general $|T_{\text{join}}| << n$, this term usually can be ignored. It is worth pointing out that when we calculate the time complexity, we have not considered the extra time needed for the join user to tell the group that it wants to join. However, this does not affect our results because in our time complexity analysis, we use the
“round” as the unit. In other words, we do not strictly require the two messages’ exchange to be synchronized. Instead, how this can be implemented really depends on the specific implementation of the two-group DH.

6.5.2 Single-User Leave Protocol

When a current group member M wants to leave the group, it broadcasts a leave request message to initiate the single user leave protocol, which contains its ID, a leave request, and a signature for this message. Once M leaves the group, the group key will be updated to remove M’s share, and all the keys on M’s key path will be updated to maintain group keys’ forward secrecy. In PACK, to reduce the rekeying cost upon a single-user leave event, the phantom node is used. This allows an existing member to simultaneously occupy more than one leaf node in the key tree. If any member wants to leave the group the following procedure should be adopted.

Procedure Leave \((G,M)\)

\(T\) is the PFMH key tree of \(G\), and \(n\) is the size of \(T\); \(T_{\text{main}}\) and \(T_{\text{join}}\) are the main tree and join tree of \(T\).

if \((M \in T_{\text{join}}) \text{ AND } (1 < |T_{\text{join}}| \leq \log n))\) then

**SCENARIO 1:** Let \(P\) be M’s sibling, remove M and M’s parent from the key tree. If \(P\) has no children, change \(P\)’s secret share; otherwise, change \(P\)’s right child’s secret share. Recursively update all the keys on \(P\)’s key path by applying multiple rounds of the two-group DH.
else if ((M ∈ T_{join} AND |T_{join}| = 1 OR |T_{join}| > \log n))

    OR (M ∈ T_{main} AND |T_{join}| > 1)

    OR (M ∈ T_{main} AND M is the rightmost non-phantom leaf node)

    OR (M ∈ T_{main} AND M has a phantom node in T))

Then

**SCENARIO 2**: First, remove all the phantom nodes and M from T. Second, apply the split procedure, and let T = \{T_1, \ldots ; T_L\} = \text{split}(T). Third, change TL’s rightmost leaf node’s secret share and recursively update all the subgroup keys on this left node’s key path in TL. Fourth, apply the unite procedure unite(T).

Else

**SCENARIO 3**: Find the rightmost non-phantom leaf node M_0 in T. Let P_{new} denote the node occupied by M and P_{old} denote the node occupied by M_0. M_0 moves to P_{new} and generates a new secret share for this location. If P_{old} lies in the join tree, then remove P_{old} and the root of T; otherwise, let P_{old} be M_0’s phantom node. Recursively update all the keys on P_{new}’s key path by applying multiple rounds of the two-group DH.

end if

All members update the key tree maintained by them locally according to the above key tree update procedure.

6.5.2.1 **Scenario 1**

This scenario considers the case that the leaving member M is in the join tree, and the size of the join tree is no larger than \log n. In this case, since the depth of the join tree is no more than \log n, we can simply remove
M’s share from the group key by removing M from the key tree, changing one current member’s secret share and recursively updating all the keys on M’s key path. Meanwhile, all the members update the key tree maintained by themselves.

Let \( h \) be M’s depth in \( T \). Since, at the most \( h-1 \) rounds of two-group DH protocols need to be performed recursively, the time cost is upper bounded by \( h-1 \). Except the last round, which involves all the existing members, in the \( i \)th \((1 \leq i < h-1) \) round, at the most \( |T_{\text{join}}|-h+1 \) members are involved. Then, the total computation cost is upper bounded by \((n + h-1+ \sum_{k=|T_{\text{join}}|-h+2}^{1} k)C_{\text{me}} \), where \( n \) comes from the last round, \( h-1 \) comes from the number of blinded keys that needs to be calculated, and \(|T_{\text{join}}|-h+1+i \) comes from the \( i \)th round. Since \(|T_{\text{join}}| \leq \log n \), a loose upper bound is \((n+ \sum_{k=1}^{\log n} k) C_{\text{me}} \) or \((n+0.5(\log n)^2)C_{\text{me}} \). Similarly, it is easy to check that the total communication cost in terms of multicast is upper bounded by \( 2(h-1)C_{\text{multicast}} \), and the total communication cost in terms of unicast is upper bounded by \((n+0.5(\log n)^2)C_{\text{unicast}} \).

![Key Tree Update Example](image)

**Figure 6.8 A key tree update upon single-user leave under the first scenario**

Figure 6.8 shows an example of a key tree update upon single-user leave under this scenario. In this example, user \( M_6 \) leaves the group where
node (1, 0) is the root of the main tree and node (1, 1) is the root of the join tree. Since the size of the join tree is 2, according to Procedure 4, the node representing M₆ will be directly removed from the key tree, M₅ changes its secret share, and a new group key will be generated by applying the two-group DH between M₅ and the subgroup in the main tree.

6.5.2.2 Scenario 2

This scenario considers the case where any of the following situations happens:

1. The leaving member M is in the join tree, and the size of the join tree is either larger than \( \log n \) or equal to 1.
2. M is in the main tree, and the size of the join tree is larger than 1.
3. M is in the main tree and is the rightmost non-phantom leaf node.
4. M is in the main tree and occupies a phantom node in the key tree.

In these situations, instead of removing M (as well as its phantom location) from the key tree and recursively updating all the keys on its key path, the whole key tree will be reorganized to generate a new PF tree as the main tree, and the join tree is set to be empty. This will reduce the rekeying cost, as well as maintain a good key tree structure. The basic procedure is to first remove all the phantom nodes in the existing key tree and then apply the split procedure to partition the remaining key tree into many small full key trees that are indexed according to their size and their locations in the original key tree. After changing a certain member’s secret share, the unite procedure will be applied to combine these full key trees into a PF key tree. Finally, all the members will update the key tree structure maintained by themselves according to the above procedure.
It is worth noting that due to the special structure of the PFMH tree, the PFMH tree structure is maintained after removing some phantom nodes. According to Procedure Leave, Scenario 3, only those leaf nodes on the rightmost of the tree can be phantom nodes. In other words, all phantom nodes lie in the rightmost part of the tree. It is easy to check that for any PF tree, after removing any number of rightmost leaf nodes and the corresponding non-leaf nodes, the remaining part is still a PF tree.

Since all the remaining members (leaf nodes) know the exact structure of the key tree, after applying the split procedure, the set of obtained full key trees will be indexed in the same way by all the group members. Since the total number of remaining members is less than \( n \), the total time cost is upper bounded by \( \log n \). If situation 1, 2, or 3 happens, the total number of full key trees after applying the split procedure is upper bounded by \( \log(n) + |T_{\text{join}}| \). In this case, the total communication cost in terms of multicast is upper bounded by \( 2(\log(n) + |T_{\text{join}}|)C_{\text{multicast}} \). If situation 4 happens, the total communication cost in terms of multicast is upper bounded by \( 2(2\log n + |T_{\text{join}}|)C_{\text{multicast}} \), where the extra \( 2\log nC_{\text{multicast}} \) is due to the fact that the main tree can be split into at the most \( 2\log n \) full trees.

Next, we analyze the computation cost under this scenario, which is mainly incurred by the unite procedure. After applying the split procedure, for any size that is greater than 1, there exists no more than one full key tree with this size when situation 1, 2, or 3 happens, and there exists no more than two full key trees with this size when situation 4 happens. The unite procedure can be implemented in two steps. In the first step, all the key trees with only one leaf node will first be combined together into a set of full key trees with different sizes. In the second step, these full key trees will be combined together with the other full key trees obtained by applying the split procedure to get the final PF tree. We first consider the more probable case that \( T_{i} \geq n/2 \),
where $T_1$ is the largest full key tree obtained after applying the split procedure. The computation cost is upper bounded by $C_{\text{me}}(2.5n + 2(\log n + |T_{\text{join}}| + 1) + |T_{\text{join}}| (\log |T_{\text{join}}| + 1))$, where the term $|T_{\text{join}}| (\log |T_{\text{join}}| + 1))$ comes from merging the nodes from the join tree into a set of full key trees with different sizes. If $T_1 < n/2$, which is a less probable case, the total computation cost is upper bounded by $(3n + 2(\log n + |T_{\text{join}}|) + |T_{\text{join}}| (\log |T_{\text{join}}|) + 1))C_{\text{me}}$. Similarly, the total communication cost in terms of unicast is upper bounded by $(2.5n + |T_{\text{join}}|\log |T_{\text{join}}|))C_{\text{unicast}}$ if $T_1 \geq n/2$ and is upper bounded by $(3n + |T_{\text{join}}|\log |T_{\text{join}}|))C_{\text{unicast}}$ if $T_1 < n/2$.

If condition 4 is satisfied, which is a very rare event, at the most $(n + \log n)C_{\text{me}}$ extra computation cost is needed to first combine those full key trees with the same size into a set of larger full key trees, and at the most $nC_{\text{unicast}}$ extra communication cost in terms of unicast is needed.

Figure 6.9 A Key Tree Update Upon Single-User Leave Under Second Scenario Situation 1

Figure 6.9 corresponds to situation 1. The leaving member $M_6$ is in the join tree, and the size of the join tree with root $(1, 1)$ is larger than $\log n$. In this example, after removing $M_6$ and applying the split procedure, three full key trees (subgroups) are obtained: $\{M_1, M_2, M_3, M_4\}$, $\{M_5\}$, and $\{M_7\}$. The result of the unite procedure has also been demonstrated.
Figure 6.10 A Key Tree Update Upon Single-User Leave Under 2nd Scenario Situation 2

The Figure 6.10 example corresponds to situation 2. The leaving member $M_2$ is in the main tree with root $(1, 0)$, and the size of the join tree with root $(1, 1)$ is larger than 1. In this case, after removing $M_2$ and applying the split, three full key trees are obtained: $\{M_3, M_4\}$, $\{M_5, M_6\}$, and $\{M_1\}$. The result of unite has also been illustrated on the right side of the figure.

Figure 6.11 A Key Tree Update Upon Single-User Leave Under 2nd Scenario Situation 3
The Figure 6.11 example corresponds to situation 3. The leaving member $M_4$ is in the main tree with root $(0, 0)$ (the join tree is empty) and is the rightmost non-phantom leaf node, where nodes $(2, 2)$ and $(2, 3)$ are phantom nodes. In this case, after removing the node representing $M_4$ and the phantom nodes and applying the split, two full key trees are obtained: \{$M_5, M_6$\} and \{M_3\}. The result of the unite has also been illustrated on the right side of the figure.

![Figure 6.12 A Key Tree Update Upon Single-User Leave Under 2nd Scenario Situation 4](image)

The Figure 6.12 example corresponds to situation 4. The leaving member $M_6$ is in the main tree with root $(0, 0)$ (the join tree is empty) and has occupied a phantom node $(2, 3)$. In this case after removing the node representing $M_4$ and the phantom node and applying the split, three full key trees are obtained: \{$M_3, M_4$\}, \{M_1\}, and \{M_5\}. The result of the unite has also been illustrated on the right side of the figure.

6.5.2.3 Scenario 3

This scenario covers all the situations that neither of the first two scenarios can cover. Specifically, this scenario considers two situations: 1) M
is in the main tree, and the size of the join tree is 1, and 2) the join tree is empty, and M is in the main tree, is not the rightmost non-phantom node, and does not have a phantom node in the key tree. Under Scenario 3, the leaving member M is removed from the key tree, and M₀, which is the member who occupies the rightmost non-phantom leaf node, moves to M’s previous position, generates a secret share for this node, and recursively updates all the keys on this node’s key path. Now, M₀ occupies two positions, and the original position is called M₀’s phantom position. It is easy to check that the time cost is bounded by \( \log n \), the communication cost in terms of multicast is bounded by \( 2(\log n)C_{\text{multicast}} \), the computation cost is upper bounded by \( (n+2|T_{\text{left}}|+ \log n)C_{\text{me}} \), where \( T_{\text{left}} \) is \( T_{\text{main}} \)’s left subtree, and the total communication cost in terms of unicast is upper bounded by \( (n+2|T_{\text{left}}|)C_{\text{unicast}} \).

Figure 6.13 shows one example of a key tree update upon single-user leave under this scenario. In this example, the join tree is empty, and the root of the main tree is \((0, 0)\). When user \( M_2 \) leaves the group, member \( M_6 \) will move to the location \((3, 1)\) that previously represents \( M_2 \). Meanwhile, \( M_6 \)
will also occupy node (2, 3), which now is a phantom node. $M_6$ will change its secret share and recursively update all the keys on its key path, which are $\{(2,0), (1,0), (0,0)\}$.

**Table 6.2 Rekeying Cost Bounds upon a Single-User Leave Event**

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Time cost (rounds)</th>
<th>Communication cost ($c_{\text{multicast}}$)</th>
<th>Communication cost ($c_{\text{unicast}}$)</th>
<th>Computation cost ($C_{\text{mc}}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scenario 1</td>
<td>$O(h)$</td>
<td>$O(2h-2)$</td>
<td>$O(n+0.5</td>
<td>T_{\text{join}}</td>
</tr>
<tr>
<td>Scenario 2</td>
<td>$O(\log n)$</td>
<td>$O(2\log n + 2</td>
<td>T_{\text{join}}</td>
<td>)$</td>
</tr>
<tr>
<td>Scenario 3</td>
<td>$O(\log n)$</td>
<td>$O(2\log n)$</td>
<td>$O(n+2</td>
<td>T_{\text{left}}</td>
</tr>
</tbody>
</table>

Table 6.2 summarizes the rekeying cost upon single-user leave events under different situations. Usually, we have $|T_{\text{left}}| \geq n/2$, $h \approx \frac{1}{2} \log n$, and $|T_{\text{join}}| << n$, and the average size of $T_{\text{left}}$ is about 0.75n. For the second and third scenarios, in most cases, we can simplify the upper bound of the computation cost as $O(2.5nC_{\text{mc}})$. For the first scenario, we can simplify the bound of the computation cost as $O(nC_{\text{mc}})$.

### 6.5.3 Group Merge Group Partition Protocols

PACK also has group merge and group partition protocols to handle simultaneously the join and leave of multiple users. Although multiple user events can be implemented by applying a sequence of single-user join or leave protocols, such sequential implementations are usually not cost-efficient. Procedure ‘merge’ describes the group merge protocol, which combines two or more groups into a single group, and returns a PF key tree.
Procedure `merge(G_1, \ldots, G_K)`

- \(T_1, \ldots, T_K\) are the key trees of \(G_1, \ldots, G_K\);
- Remove all phantom nodes from \(T_1, \ldots, T_K\);
- \(T = \text{unite}(\text{split}(T_1) \cup \ldots \cup \text{split}(T_K))\);
- Return \(T\).

Procedure ‘Partition’ describes the group partition protocol, which removes multiple group members simultaneously from the current group and constructs a new PF key tree for the rest of the group members.

Procedure `Partition(G, G_1)`

- \(T\) is the key tree of \(G\);
- Remove all phantom members and members belonging to group \(G_1\) from \(T\);
- \(T = \text{unite}(\text{split}(T))\);
- Return \(T\).

In the group merge protocols, after removing all the phantom nodes from those key trees corresponding to different subgroups, each key tree is split into several full key trees. The final result is obtained by uniting these full key trees into a PF tree following the unite Procedure. Similar to the group partition protocol, after removing all the phantom and leaving nodes, the original key tree is split into several full key trees, and the unite procedure is then applied on these full key trees to create a PF key tree. Since the height of the returned tree is \(\log n\), where \(n\) is the group size after merging/partitioning, the time cost of the group merge/partition is bounded by \(O(\log n)\). Obviously, the group merge and partition protocols have a lower cost than the sequential implementations.
6.6 PERFORMANCE EVALUATION AND COMPARISON

6.6.1 Forward and Backward Security

Group key secrecy means that attackers cannot obtain the group key even if they know all the blind keys. To show that PACK satisfies forward and backward secrecy, we can use the detailed proof for TGDH. PACK and TGDH use similar group key update procedures. The major difference between them is in the underlying key tree structures, which do not affect the security of the scheme. Therefore, in this chapter, we will not provide a detailed proof of forward and backward secrecy. Next, we only roughly sketch the proof. We first consider backward secrecy. When a new user M wants to join the group, M picks its secret share r.

After several rounds of the two-group DH, M gets all the blinded keys on its copath, and it can compute all the secret keys on its key path using its own secret share and the blinded keys on its copath. Clearly, all these keys contain M’s secret share; hence, they are independent of the previous secret keys on that path. Therefore, M cannot derive any previous keys. Forward secrecy can be shown in a similar way. When a member M leaves the group, at least one current member changes its share, and all the keys on M’s key path will be updated to remove M’s secret share. Hence, M only knows at the most all the blinded keys, and the group key secrecy property prevents M from deriving any future group keys. By combining backward secrecy and forward secrecy, we can derive the key independence.

6.6.2 Cost Comparison

This section compares the rekeying cost in PACK upon single-user join and leave events with that of two existing tree-based contributory group key agreement schemes: TGDH and DST. All three types of costs are
considered: time, computation, and communication in terms of multicast. Since, in general, a members’ leaving time is not known in advance, in DST, only the join tree is used.

Table 6.3 lists the approximate bounds of the different costs for the three schemes. From that, we can see that PACK has the lowest cost in terms of time, computation, and communication. For example, for user join, only one or two rounds are needed in time cost, whereas DST needs \(1 + \log \log n\) rounds, and TGDH needs \(\log n\) rounds. Similar results can also be seen in the communication cost for user join. The total computation cost is computed as the average of the user join cost and leave cost; DST has a similar cost as TGDH, which is an order of \(2n\), whereas for PACK, the order is from \(n\) to 1.75\(n\), with the savings ranging from 15% to 50% compared with those of DST and TGDH.

**Table 6.3 Rekeying Cost Comparison among Different Schemes**

<table>
<thead>
<tr>
<th></th>
<th>Upon Single User Join Event</th>
<th>Upon Single User Leave Event</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time cost</strong></td>
<td><strong>Communication cost</strong></td>
<td><strong>Computation cost</strong></td>
</tr>
<tr>
<td>PACK</td>
<td>1~2</td>
<td>(2\sim 4C_{\text{multicast}})</td>
</tr>
<tr>
<td>TGDH</td>
<td>(\log n)</td>
<td>(2(\log n)C_{\text{multicast}})</td>
</tr>
<tr>
<td>DST</td>
<td>(1+\log \log n)</td>
<td>((1+\log \log n)C_{\text{multicast}})</td>
</tr>
</tbody>
</table>

|                  | **Time cost**              | **Communication cost**     | **Computation cost** |
| PACK             | \(\log n\)                 | \(2(\log n)C_{\text{multicast}}\) | \(1\sim 2.5)nC_{\text{me}}\) |
| TGDH             | \(\log n\)                 | \(2(\log n)C_{\text{multicast}}\) | \(2nC_{\text{me}}\) |
| DST              | \(1+\log n + \log \log n\) | \(2(1+\log n + \log \log n)C_{\text{multicast}}\) | \(3nC_{\text{me}}\) |
6.6.3 Simulation Results

In our simulations, we generate the user activities according to the following probabilistic models: Users join the group according to a Poisson process with the average arrival rate $\lambda$, and the users’ staying time in the group follows an exponential distribution with mean $\mu$. Then, $\lambda \mu$ is the average number of users in the group, that is, the average group size. For each simulation, we initialize the group size to be 0, fix $\lambda$, and vary $\mu$ to get different average group size configurations. For each configuration (different average group size), a sequence of 100 $\lambda \mu$ users join the group according to the Poisson process with rate $\lambda$, and each user’s staying time is drawn independently from an exponential distribution with mean $\mu$. In the simulations, we have compared the rekeying costs of the three schemes, PACK, TGDH, and DST, in all three aspects: computation, communication, and time.

The simulation results are presented in Figures 6.14 to 6.19. From these results, we can see that upon a single-user join event, PACK has the lowest cost among all the three schemes. Compared with DST, PACK has a more than 10% reduction in computation cost and a more than 65% reduction in communication cost and time cost. Compared with TGDH, the reduction is even more, about 50% in computation cost and about 80% in time and communication costs. Upon a single-user leave event, compared with DST, PACK has about a 25% reduction in computation cost, about a 15% reduction in time cost, and a similar communication cost. Although PACK has slightly higher computation and communication costs than TGDH upon a single user leave event, when averaged over both join and leave events, the reduction is still significant, with a 20% reduction in computation cost, 35% reduction in communication cost, and 40% reduction in time cost.
Figure 6.14  Average Communication Cost Upon Single User Join

Figure 6.15  Average Communication Cost Upon Single User Leave
Figure 6.16 Average Computation Cost Upon Single User Join

Figure 6.17 Average Computation Cost Upon Single User Leave
Figure 6.18  Average Time Cost Upon Single User Join

Figure 6.19  Average Time Cost Upon Single User Leave
6.7 CONTRIBUTORY GROUP KEY AGREEMENT WITH KEY VALIDATION

In practice, there may exist compromised group members who do not perform the key agreement protocol honestly and cause key generation failure. One example of key generation failure is group partition, where some users share one key while the others share another different key. That is, without being detected by other users, malicious users cannot prevent a valid group key from being generated by providing false information. We introduce two methods to check the validity of the key establishment procedure and to detect malicious members. One is preventive, and the other is detective. In the preventive scheme, for each group, \( m \) members are elected as delegates who broadcast the blinded key. Then, each group member checks whether these \( m \) copies of the blinded keys are the same. Since all the keying messages have been signed by the senders, the member who has sent false information can be easily detected by the other group members. In the detective scheme, after each round of DH, \( m \) members are elected to broadcast a common known message encrypted using the newly generated group/subgroup key. Other members check whether they can use their new group/subgroup key to successfully decrypt the message. If a user cannot obtain this commonly known message after decryption, it broadcasts an error message that includes the blinded key and the messages it has received. Again, since keying messages are signed by their senders, those malicious members who have sent a false blinded key or false encrypted message can be detected.

6.8 SUMMARY

In this chapter, we designed PACK, a highly efficient contributory key agreement scheme that has much lower communication, computation, and time overhead than the existing schemes. PACK reduces the overhead
associated with key updating in two ways. First, it uses the novel PFMH tree structure that consists of a main tree, which is optimal for user leave, and a join tree, which is optimal for user join. Second, the concept of a phantom user location in the PFMH allows cost amortization when handling user leave. Upon single-user join, PACK has the time cost as one or two rounds of the two-group DH, the communication cost as two or four multicast, and the average computation cost as one modular exponentiation per user. Upon a single-user leave event, PACK takes at the most $\log n$ rounds of the two-group DH in terms of time cost, $O(\log n)$ multicast in communication cost, and an average of 2 modular exponentiations per user in computation cost. The performance of PACK is compared with that of TGDH and DST. The simulation results have shown that PACK has much lower rekeying costs in terms of communication, computation, and time than the existing schemes.