Optimizing Rekeying Cost for Contributory Group Key Agreement Schemes

Wei Yu[*], Yan Sun[†] and K. J. Ray Liu[*]

[*]Department of Electrical and Computer Engineering,

University of Maryland, College Park, MD 20742

Email: weiyu, kjrliu@umd.edu

[†]Department of Electrical and Computer Engineering,

University of Rhode Island, 4 East Alumni Ave, Kingston, RI 02881

Email: yansun@ele.uri.edu

Abstract

While 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 cost. In this paper 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 two-party DH upon any single user join event and $O(\log n)$ rounds of two-party DH upon any single user leave event. Both theoretical bound analysis and simulation show that the proposed scheme achieves lower rekeying cost than the existing tree-based contributory group key agreement schemes.

Index Terms

Security, Key Management, Tree Structure.

This work was supported in part by the Army Research Office under URI Award No. DAAD19-01-1-0494

I. Introduction

One fundamental challenge in securing group 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 group key agreement schemes [1]–[20]. Since many practical group applications do not prefer utilizing centralized key servers, contributory solutions of key agreement have drawn extensive attention [1]–[11], especially for applications where centralized administration and pairwise secure channels are not applicable.

In contributory group key agreement schemes, all group members contribute their shares and compute the group key collaboratively, and the group key is generated as a (usually one-way) function of individual contributions from all group members [1]–[11]. Upon membership changes, the group key needs to be updated to incorporate the share from the joining user or to eliminate the share of the leaving user to maintain *backward secrecy* and *forward secrecy* [9]. 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.

The early design of contributory group key agreement schemes mostly focuses the efficiency of initial group key establishment, such as in [1]–[3]. These schemes, however, encounter high rekeying cost upon group membership changes. Later, Steiner et al. proposed a family of Group Diffie-Hellman (GDH) protocols by extending the two-party Diffie-Hellman (DH) protocols [21] to the group scenarios [4], [6], [7]. The GDH protocols achieve efficient key update upon user join, but still require high cost for member leave. Recently, logical key tree structures are used to improve the scalability of contributory key agreements [9], [10]. Kim et al. proposed a tree-based contributory group key agreement protocol called TGDH, where binary balanced tree is adopted to maintain the keying material [9]. In TGDH, the group key can be updated by performing $\log n$ rounds of two-party DH upon any single user join or leave, where n is the group size. Mao et al. proposed another tree-based contributory key agreement scheme called DST [10].

By using a special join-tree/exit-tree topology and exploiting cost amortization, DST can reduce the average time cost to $\Theta(\log\log n)$ rounds of two-party DH for single user join or leave. However, 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 $\Theta(\log n + \log\log n)$, which is higher than that of TGDH.

What is the lowest possible cost of contributory group key agreement schemes? The theoretical analysis in [22] indicates that for any tree-based contributory group key management scheme, the lower bound of the worst case cost is $\Theta(\log n)$ rounds 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 $\Theta(\log n)$. In addition, it is obvious that at least one round of two-party DH needs to be performed for adding or deleting a user in any circumstance. Therefore, lowest possible cost for contributory key agreement is $\Theta(\log n)$ for user join and $O(1)$ for user leave; or $\Theta(\log n)$ for user leave and $O(1)$ for user join. Both TGDH and DST do not achieve these lower bounds. In addition, from [22] we can also derive that the total rounds of two-party DH for any sequence of n single user join events and n user leave events is bounded by $\Theta(n \log n)$. This bound is in fact looser than the previous two bounds.

To achieve the lower bound of the rekeying cost, in this paper we propose a novel and efficient logical key tree structure, called PFMH tree, as well as a cost-minimizing PFMH tree-based contributory group key agreement protocol suite (PACK) that handles dynamic group membership events. The optimality of the proposed PACK protocol suite lies in that it only needs $O(1)$ rounds of two-party DH upon any single user join event and $O(\log n)$ rounds of two-party DH upon any single user leave event, which achieves the lower bound. Both theoretical analysis and simulation studies show that PACK has much lower rekeying cost than the existing tree-based contributory group key agreements.

The rest of this paper is organized as follows. Section II briefly introduces security requirements and performance metrics for contributory group key agreement schemes. Section III presents the proposed PFMH tree structure as well as two basic procedures to manage the PFMH key trees. Section IV describes the proposed PFMH tree-based contributory group key agreement protocol suite to optimize rekeying cost upon single user join and leave events. Section V analyzes the performance of the proposed scheme and demonstrates its efficiency by comparing it with existing tree-based contributory group key agreement schemes through both bound analysis and simulation studies. Section VI discusses the detection of untruthful users who do not perform key agreement protocol honestly. Finally, conclusion is drawn in Section VII.

II. Security Requirement and Performance Metric

In this section, we briefly introduce the security requirements of 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* [9]. Single user addition (deletion) means that one user joins (leaves) the group. Group merge (partition) involves multiple users who join (leave) the group simultaneously. The security requirements with dynamic membership include *group key secrecy*, *forward secrecy*, *backward secrecy*, and *key independence* [9]. Group key secrecy, which is the most basic property, requires that it should be computationally infeasible for a passive adversary to discover any group key. Forward secrecy requires that a passive adversary who knows a contiguous subset of old group keys cannot discover subsequent group keys, while backward secrecy requires that a passive adversary who knows a contiguous subset group keys cannot discover preceding group keys. Key independence, which is the strongest property, requires that a passive adversary who knows a proper subset of group keys cannot discover any other group key. According to [9], key independence can be achieved when both forward secrecy and backward secrecy are achieved.

The overhead of group key agreement involves *computation cost*, *communication cost* and *time cost*. Since most of the existing contributory key agreement schemes use two-party DH protocol [21] as a basic building module, the computation cost comes mainly from the cryptographic primitives that are needed to perform 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 possible parallelism when performing group key establishing and updating, the time cost can be significantly reduced.

Among existing contributory group key agreement schemes [1]–[11], tree-based schemes are most promising because of their scalability. Next we introduce the implementation of two-group DH (two-party DH among two groups), which is the basic building module for most tree-based contributory group key agreement schemes. Let $\mathbf{A}$ and $\mathbf{B}$ denote two subgroups, where the users in $\mathbf{A}$ share a common group key K_A, and the users in $\mathbf{B}$ share a common group key K_B. Let $f(K)$ (which we refer to as the blinded key of key K) denote the modular exponentiation operation, that is

$$f(K) = g^K \ mod\ p, \tag{1}$$

where g is the exponential base and p is the modular base. The two-group DH can be implemented as follows. Each subgroup elects one member as its delegate, which will compute and send its blinded subgroup key to all members of the other subgroup. Suppose that member A_1 is the delegate elected by the subgroup $\mathbf{A}$, and member B_1 is the delegate elected by the subgroup $\mathbf{B}$. To perform 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 members of subgroup $\mathbf{B}$, and B_1 sends the blinded key $f(K_B)$ to all members of subgroup $\mathbf{A}$. Now each member in $\mathbf{A}$ or $\mathbf{B}$ then calculates the new group key K_{AB} as follows:

$$K_{AB} = (f(K_B))^{K_A} \text{ mod } p = (f(K_A))^{K_B} \text{ mod } p. \tag{2}$$

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 paper, we use $C_{cast}(n, \ell)$ to denote the communication cost needed to send a message with length ℓ to n nodes, and use C_{me} to denote the computation cost of a modular exponentiation operation. Thus, for each round of 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 $C_{cast}(n_1, \ell) + C_{cast}(n_2, \ell)$, and the computation cost is no more than $(n_1 + n_2 + 2)C_{me}$.

It is worth noting that sending a message to n nodes can be implemented in many ways. It can either be implemented through multicast communications, 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 an 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 an 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, while in wired networks without link-level multicast support, the gap is usually not that obvious.

In this paper, 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 cost of multicast-n_1 and multicast-n_2 with $n_1 \neq n_2$ are usually different, to simplify our illustration, in this paper we will not distinguish them. Let $C_{multicast}(\ell)$ denote the communication cost of an multicast-n operation, and let $C_{unicast}(\ell)$ denote the communication cost of a unicast-1 operation, where ℓ is the length of the message to be sent. Further, when performing two-group DH between two subgroups, 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 of the group members when it wants to join or leave a group. In this paper we use $C_{broadcast}(\ell)$ to denote the communication cost incurred by broadcasting a message with length ℓ to all group members.

III. PFMH Key Tree Structure and Basic Procedures

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, leaf nodes represent 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 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 [9]. Since two-group DH is used, the key tree is binary. For each node in the key tree, *key-path* denotes the path from this node to the root, and *co-path* denotes the sequence of siblings of each node on its key-path. Fig. 1 shows a simple key tree example with 6 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 co-path is the sequence of nodes $\{(3,0),(2,1),(1,1)\}$.

According to [9], in order to compute the group key, a node only needs to know its own key and all the blinded keys on its co-path. In other words, for a node being able to calculate

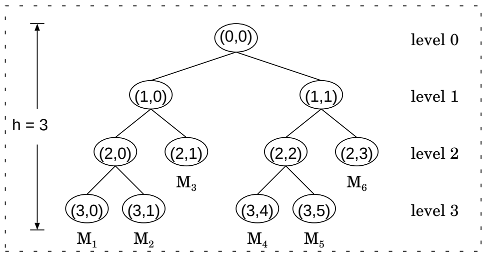


Fig. 1. A simple key tree example

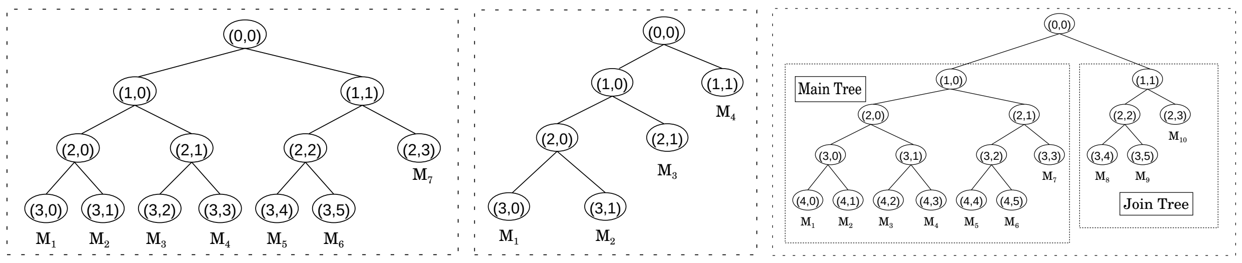
the group key, it only needs to know its own keys and all the blinded keys on its co-path. For example, as shown in Fig. 1, M_2 only needs to know its own key and the blinded keys represented by the nodes $(3,0)$, $(2,1)$ and $(1,1)$ in order to calculate the group key.

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 [1]. When using a balanced key tree structure, as in TGDH [9], the worst-case rekeying time cost for both user leave and user join is $O(\log n)$. In order to further 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 scheme may result in an extremely unbalanced key tree structure and increase the rekeying cost for user leave to $O(n)$.

In order to achieve the lower bound for both user join and user leave simultaneously, in this paper we propose a novel and efficient key tree structure for contributory group key agreement schemes, which we refer to as PFMH tree. PFMH tree is a combination of two special key tree structures: *partially-full* (PF) key tree and *maximum height* (MH) key tree. In this paper, the size of a key tree is defined as the total number of leaf nodes in this tree, the function $\log()$ and $\log_2()$ will be used exchangeably, and when we say a “full (key) tree”, we always mean a fully balanced binary (key) tree with size 2^k, where k is a non-negative integer.

Definition 1 (PF key tree) *Let T be a binary key tree of size n, and let $n' = 2^{\lfloor \log n \rfloor}$. T is a PF key tree if and only if it satisfies one of the following properties: 1) T is a full key tree; 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 $(n - n')$.*

[1]The case that users have varying computation and communication capabilities is beyond the scope of this paper.



(a) PF key tree (b) MH key tree (c) PFMH key tree

Fig. 2. Some examples of PF/MH/PFMH key trees

Definition 2 (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 a MH key tree with size $n - 1$.*

Definition 3 (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 a MH tree.*

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 a MH tree with size n is $n - 1$. In this paper, without introducing ambiguity, we will use $\lceil \log n \rceil$ and $\log n$ exchangeably. Also, given a PFMH key tree T, we will use *main tree* to refer to the PF subtree of T, denoted by T_{main}, and use *join tree* to refer to the MH subtree of T, denoted by T_{join}. It is easy to see that the height of T_{main} is always bounded by $\log n$. Fig. 2 illustrates these special key tree structures. Next we describe two basic procedures to manage and update PFMH key trees: *unite* and *split*.

Let $\mathcal{T} = \{T_1, \ldots, T_L\}$ be a set of full key trees. Each key tree $T_i \in \mathcal{T}$ represents a subgroup, and each leaf node of T_i is a member of this subgroup. If a group member belongs to T_i and $T_i \in \mathcal{T}$, then this group member belongs to $\mathcal{T}$. The procedure *unite($\mathcal{T}$)* is to combine those key trees in $\mathcal{T}$ into a single PF key tree through performing a series of two-group DH among these subgroups as well as the subgroups generated during this procedure. In general, given a set of full key trees $\mathcal{T}$, the result of *unite($\mathcal{T}$)* may not be unique, but all of the obtained PF key trees have similar structure. In this paper we consider a special case where the full key trees in $\mathcal{T}$ are ordered and indexed according to their sizes. And, any group member in $\mathcal{T}$ knows the indices and sizes of any trees in $\mathcal{T}$ as well as these structure of these trees. The structure of a tree refers to the list of group members belonging to this tree and their exact positions in this tree. Then a group member can decide with whom it should perform two-group DH and in what order.

Procedure 1 presents one specific implementation of *unite($\mathcal{T}$)* for this special case. According to Procedure 1, the whole procedure is partitioned into many rounds. At the beginning of each round, there remains a set of full trees (subgroups) indexed according to their sizes and their subtrees' indices in previous rounds. The larger the size of a subgroup, the lower its index. In each round, a remaining subgroup may either keep alone or be paired with another remaining subgroup according to the following rule: two subgroups T_i and T_j ($i < j$) will be paired together if and only if all of the three conditions can be satisfied:

- There is no other remaining subgroup in the round with index lying between i and j;
- The total number of subgroups with size equal to $|T_i|$ and with index lying before T_i is even;
- $|T_i| = |T_j|$ or T_j is the subgroup with largest index.

It is easy to see that in each round, a subgroup will either keep alone or be paired with one and only one other subgroup to build a larger subgroup. Further, in each round all pairs of subgroups can perform two-group DH between them in parallel, which can significantly reduce the time cost. If Procedure 1 is followed by all group members, the obtained PF key tree is unique and each member can know its location in the final PF before starting the procedure, and each member can locally construct the final PF tree without explicitly exchanging key tree updating information.

Given a key tree T, the procedure *split(T)* is to partition T into a set of full key trees with minimum set size. Specifically, after applying the procedure *split(T)*, any obtained key tree is a full key tree, and no any two or more obtained key trees comes from any full subtree of T. Procedure 2 presents a way to locally and virtually *split* a key tree, where "locally" means that no inter-communication is needed among group members and each member only needs to update the key tree structure maintained by itself locally, while "virtually" means that no any two-group DH is needed to perform "split". Meanwhile, the set of obtained full key trees are also indexed according to their size and their positions in the original key tree.

Fig. 3 shows two examples of key tree update when applying *unite* and *split* procedures. The left figure demonstrates how the key tree is updated when 5 full key trees are *united* into a PF key tree. The right figure demonstrates how the key trees are updated when a PFMH key tree

Procedure 1 $unite(\{T_1, \ldots, T_L\})$

```
▷ $\mathcal{T} = \{T_1, \ldots, T_L\}$; $|T_i| \geq |T_j|$ for any $1 \leq i < j \leq L$; each member in $T_i \in \mathcal{T}$ knows the index and size of any tree $T_j \in \mathcal{T}$ as well as the structure $T_j$, including the list of group members in $T_j$ and their exact positions in $T_j$.

$\mathcal{T}' = \mathcal{T}$; $L' = L$;
**while** ($|\mathcal{T}'| > 1$) **do**
  /*Executed in parallel:*/
  **for** (each pair of trees $T_i, T_{i+1} \in \mathcal{T}'$) **do**
    **if** ((the total number of trees in $\mathcal{T}'$ with size equal to $|T_i|$ and with index before $T_i$ is even) AND ($|T_i| = |T_{i+1}|$ or $T_{i+1}$ is the tree in $\mathcal{T}'$ whose index is the largest)) **then**
      Two delegates will be elected by subgroups $T_i$ and $T_{i+1}$ to perform two-group DH between them, and a new group key $K$ will be generated. A new key tree will be generated with its root node representing $K$, with the left child of the root node being $T_i$ and with the right child of the root node being $T_{i+1}$. Remove $T_i$ and $T_{i+1}$ from $\mathcal{T}'$.
    **end if**
  **end for**
  Put all newly generated key trees in this round into $\mathcal{T}'$ and let $L'$ be the total number of key trees now in $\mathcal{T}'$. Re-index all the key trees in $\mathcal{T}'$ with integers ranging from 1 to $L$ in such a way that a tree is assigned index $i$ (that is, this tree's name becomes $T_i$) if and only if: 1) for any tree $T_j \in \mathcal{T}'$ with index $j < i$, all subtrees of $T_j$ that directly come from $\mathcal{T}$ have lower indices than all subtrees of $T_i$ that directly come from $\mathcal{T}$; and 2) for any tree $T_j \in \mathcal{T}'$ with $j > i$, all subtrees of $T_j$ that directly come from $\mathcal{T}$ have higher indices than all subtree of $T_i$ that directly come from $\mathcal{T}$.
**end while**
Return the remaining tree $T_1$ in $\mathcal{T}'$, which is the final PF key tree. Meanwhile, each member in the final PF key tree will construct the final key tree structure locally by following the above key tree generation procedure.
```

is *split* into a set of full key trees.

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 cost are needed. In the *unite* procedure, extra cost will be incurred when performing a sequence of two-group DH to generate the new key tree. Next we analyze the cost associated with the *unite* procedure described in Procedure 1. The results will be used later to analyze the cost of those proposed key agreement protocols.

Procedure 2 $split(T)$

```
if ($T$ is a full tree) then
    Return $\{T\}$;
else if ($T$ is empty) then
    Return $\emptyset$.
else
    Let $T_{left}$ and $T_{right}$ be the left and right subtrees of $T$;
    Return $split(T_{left}) \bigcup split(T_{right})$.
end if
Let $L$ be the number of obtained full key trees. Index these key trees with the integers ranging from 1 to $L$ in such a way that a tree is indexed as $T_i$ if and only if: 1) for any tree $T_j \in \mathcal{T}'$ with $j < i$, $|T_j| > |T_i|$ or $T_j$ lies in the left side of $T_i$ in $T$; and 2) for any tree $T_j \in \mathcal{T}'$ with $j > i$, $|T_j| < |T_i|$ or $T_j$ lies in the right side of $T_i$ in $T$.
```

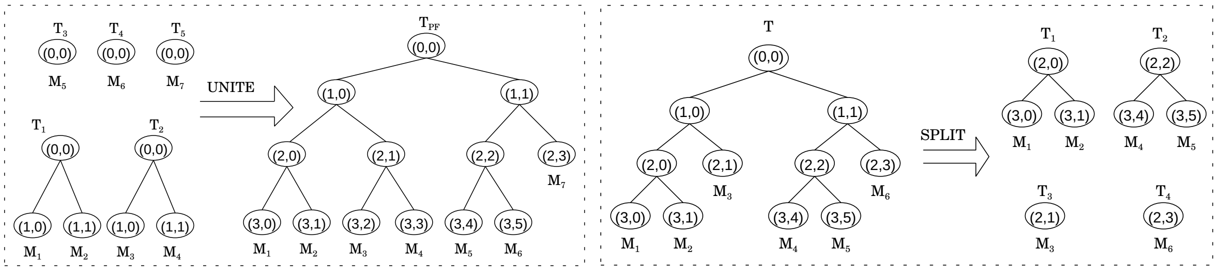


Fig. 3. Examples of key tree update after applying *unite* and *split* procedures

Theorem 1: Let $\mathcal{T} = \{T_1, \ldots, T_L\}$ be a set of full key trees with $\sum_{i=1}^{L} |T_i| = n$ and $|T_1| \geq |T_2| \geq \cdots \geq |T_L|$, and with the subscript ℓ being the index of the full tree T_ℓ. Assume that any group member in any full tree $T_i \in \mathcal{T}$ knows the index and size of any tree $T_j \in \mathcal{T}$ as well as the structure of T_j. Then the costs associated with the $unite(\mathcal{T})$ using Procedure 1 can be bounded as follows:

1) The time cost, which is the number of parallel rounds that needs to executed, is upper-bounded by $\log n$ in all situations.
2) The total communication cost is upper-bounded by $2(L-1)C_{multicast}$ in all situations provided that the exchange of keying materials between two subgroups during performing two-group DH is implemented using multicast.
3) Consider the special situation that $|T_i| = 1$ for all $1 \leq i \leq L$, the total computation cost

is upper-bounded by $n(\log n + 2)C_{me}$, and the total communication cost is upper-bounded by $(n \log n)C_{unicast}$ provided that the exchange of keying materials between two subgroups during performing two-group DH is implemented using unicast.

4) Consider the special situation that $|T_i| \neq |T_j|$ for any $1 \leq i \neq j \leq L$, the total computation cost is upper-bounded by $2(n + \log n)C_{me}$, and the total communication cost is upper-bounded by $2nC_{unicast}$ provided that the exchange of keying materials between two subgroups during performing two-group DH is implemented using unicast.
5) Consider the special situation that $|T_1| \geq n/2$ and for any tree $T_i \in \mathcal{T}$ there exists no more than one other tree in $\mathcal{T}$ with the same size as T_i, the total computation cost is upper-bounded by $(2.5n + 2L)C_{me}$, and the total communication cost is upper-bounded by $2.5nC_{unicast}$ provided that the exchange of keying materials between two subgroups during performing two-group DH is implemented using unicast.
6) Consider the special situation that $|T_1| < n/2$ and for each tree $T_i \in \mathcal{T}$ there exists no more than one other tree in $\mathcal{T}$ with the same size as T_i, the total computation cost is upper-bounded by $(3n + 2L)C_{me}$, and the total communication cost is upper-bounded by $3nC_{unicast}$ provided that the exchange of keying materials between two subgroups during performing two-group DH is implemented using unicast.

Proof: See Appendix.

IV. PACK: AN PFMH TREE-BASED CONTRIBUTORY GROUP KEY AGREEMENT

In this section we describe the proposed PFMH tree-based contributory group key agreement protocol suite, referred to as PACK. As a contributory scheme, in PACK each group member equally contributes its share to the group key and this share is never revealed to the others. To satisfy the security requirements, 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 minimum rekeying time cost upon membership change events in the sense that for any single user join event, the rekeying time cost is of order $O(1)$, and for any single user leave event, the rekeying time cost is of order $O(\log n)$. Meanwhile, the rekeying computation and communication cost can also be significantly reduced compared with the existing tree-based contributory group key agreement schemes. This is achieved through adopting the proposed PFMH tree as the underlying key tree structure and introducing *phantom* nodes in the key tree to handle member leave.

In PACK, each member will maintain and update the global key tree locally. 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. As to be shown next, upon group membership change event, a group members only needs to update the global key tree maintained by itself, which can greatly reduce the communication overhead. 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, according to the leaving member's location in the key tree as well as whether this member has phantom location in the key tree, different procedures will be applied, and the basic idea is to update the group key in $O(\log n)$ rounds and simultaneously reduce the communication and computation cost.

A. Single User Join Protocol

When a prospective user M wants to join the group $\mathcal{G}$, 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 group keys' backward secrecy. Procedure 3 describes the single user join protocol in PACK.

In PACK, the rekeying upon single user join needs to perform at most 2 rounds of two-group DH. If the join tree is not empty, a new join tree is generated by performing 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 two-group DH between the new join tree and the main tree. Since all the current members know the group key tree structure and knows the location that the new member should be put in, they can update the key tree themselves.

Procedure 3 $join(\mathcal{G}, M)$

```
▷ $T$ is the PFMH key tree of group $\mathcal{G}$, $T_{main}$ 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 $\mathcal{G}$ to perform 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 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 will send an updated copy of the key tree to the new joining member $M$.
```

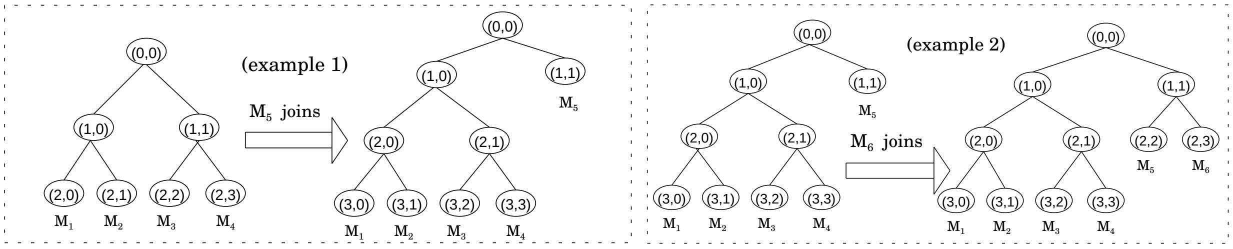


Fig. 4. Examples of key tree update upon single user join event

Fig. 4 shows two examples of key tree update upon single user join events. In the first example, the join tree is empty, and the main tree consists of 4 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 which represents M_5. In the second example, when M_6 joins the group, at the first round, two-group DH is first performed between M_5 and M_6 to generate a new join tree, at the second round, two-group DH is performed between the new join tree and the main tree to generate a new group key.

TABLE I

REKEYING COST UPON SINGLE USER JOIN EVENT

	time cost	communication cost in term of multicast	communication cost in term of unicast	computation cost
case 1	1	$2C_{multicast}$	$nC_{unicast}$	$(n+2)C_{me}$
case 2	2	$4C_{multicast}$	$(n+\|T_{join}\|+1)C_{unicast}$	$(n+\|T_{join}\|+3)C_{me}$

Table I lists the rekeying cost upon single user join event in PACK where n denotes the total number of leaf nodes in the new group and $|T_{join}|$ 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 two-group DH. Case 2 considers the situation that the join tree is not empty, and the protocol needs to perform two rounds of two-group DH. For case 2, the term $|T_{join}|+2$ in the computation cost comes from performing two-group DH between the new member and the old join tree. Since in general $|T_{join}| \ll n$, 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 “round” as unit. In other words, we do not strictly require the two messages exchange to be synchronized. Instead, how this can be implemented is really depend on the specific implementation of the two-ground DH.

B. 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 single user leave event, we introduce the concept of *phantom* node which allows an existing member to simultaneously occupy more than one leaf node in the key tree. In particular, when member M leaves the group, another group member M' will move to the position occupied by M in the key tree, generate a new secret key, and all the keys on M’s key-path will be recursively updated. It is worth noting that here “moving” only means that each member adjusts the location of M' and M in the key tree.

After moving M' to M's position, the node that M' previously occupied will not be deleted immediately. As a result, now M' occupies two leaf nodes in the key tree. We refer to the node associated to M''s previous position as the *phantom* node, which is known by all group members. In order to maintain group keys' forward secrecy, a phantom node should be deleted no later than the associated group member leaving the group. Procedure 4 describes the single user leave protocol in PACK.

Procedure 4 $Leave(\mathcal{G}, M)$

```
▷ $T$ is the PFMH key tree of $\mathcal{G}$, and $n$ is the size of $T$, $T_{main}$ and $T_{join}$ is the main tree and join tree of $T$.

if (($M \in T_{join}$) AND ($1 < |T_{join}| \leq \log n$)) then
  SCENARIO I: 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 two-group DH.
else if (($M \in T_{join}$ AND ($|T_{join}| = 1$ OR $|T_{join}| > \log n$))
     OR ($M \in T_{main}$ AND $|T_{join}| > 1$)
     OR ($M \in T_{main}$ AND $M$ is the rightmost non-phantom leaf node)
     OR ($M \in T_{main}$ AND $M$ has a phantom node in $T$)) then
  SCENARIO II: First, remove all phantom nodes and $M$ from $T$. Second, apply the split procedure, and let $\mathcal{T} = \{T_1, \ldots, T_L\} = split(T)$. Third, change $T_L$'s rightmost leaf node's secret share, and recursively update all the subgroup keys on this left node's key-path in $T_L$. Fourth, apply the unite procedure $unite(\mathcal{T})$.
else
  SCENARIO III: Find the rightmost non-phantom leaf node $M'$ in $T$. Let $P_{new}$ denote the node occupied by $M$, and $P_{old}$ denote the node occupied by $M'$. $M'$ 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 $M'$'s phantom node. Recursively update all the keys on $P_{new}$'s key path by applying multiple rounds of two-group DH.
end if
All members update the key tree maintained by them locally according to the above key tree update procedure.
```

SCENARIO I: 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 (which member's share should be changed is described in Protocol 4), and recursively updating all the keys on M's key-path.

Meanwhile, all members update the key tree maintained by themselves.

Let h be M's depth in T. Since at 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 $i^{th}(1 \le i < h-1)$ round at most $|T_{join}| - h + i + 1$ members are involved. Then the total computation cost is upper-bounded by $(n + h - 1 + \sum_{k=|T_{join}|-h+2}^{|T_{join}|-1} k)C_{me}$, where n comes from the last round, $h-1$ comes from the number of blinded keys that need to be calculated, and $|T_{join}| - h + 1 + i$ comes from the i^{th} round. Since $|T_{join}| \le \log n$, a loose upper-bound is $(n + \sum_{k=1}^{\log n} k)C_{me}$, or $(n + 0.5(\log n)^2)C_{me}$. Similarly, it is easy to check that the total communication cost in term of multicast is upper-bounded by $2(h-1)C_{multicast}$, and the total communication cost in term of unicast is upper-bounded by $(n + 0.5(\log n)^2)C_{unicast}$.

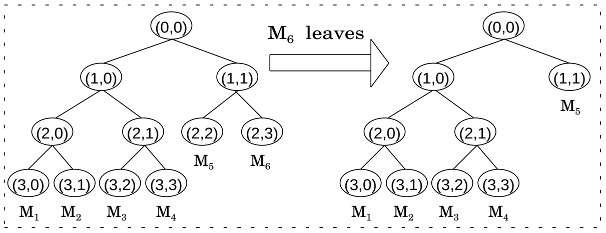


Fig. 5. Example of key tree update upon single user leave under the first scenario

Fig. 5 shows one example of 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 main tree and node $(1,1)$ is the root of join tree. Since the size of join tree is 2, according to Procedure 4, the node representing M_6 will be directly removed from the key tree, M_5 changes its secret share, and a new group key will be generated by applying two-group DH between M_5 and the subgroup in the main tree.

SCENARIO II: This scenario considers the case that any of the following situations happens:

1) The leaving member M is in the join tree, and the size of join tree is either larger than $\log n$ or equal to 1;
2) M is in the main tree, and the size of 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, then apply the *split* procedure to partition the remaining key tree into many small full key trees which 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 members will update the key tree structure maintained by themselves according 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 4 scenario III, only those leaf nodes on the rightmost of the tree can be phantom nodes. In other words, all phantom nodes lie in the right-most part of the tree. It is easy to check that for any PF-tree, after removing any number of right-most leaf nodes and those corresponding non-leaf nodes, the remaining part is still a PF-tree.

Since all the remaining members (leaf nodes) know the exact structure of key tree, after applying the *split* procedure the set of obtained full key trees will be indexed in the same way by all group members. Since the total number of remaining members is less than n, according to Theorem 1 clause 1, 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 $\log(n)+|T_{join}|$. In this case, the total communication cost in term of multicast is upper-bounded by $2(\log(n)+|T_{join}|)C_{multicast}$. If situation 4 happens, the total communication cost in term of multicast is upper-bounded by $2(2\log n+|T_{join}|)C_{multicast}$, where the extra $2\log nC_{multicast}$ is due to the fact that the main tree can be split into at 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 1 full key tree with this size when situation 1, 2, or 3 happens, and there exists no more than 2 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 $T_1 \geq n/2$ where T_1 is the largest full key tree obtained after applying the *split* procedure. According to Theorem 1 clause 3 and clause 5, in this case the computation cost is upper-bounded by $C_{me}(2.5n+2(\log n+|T_{join}|)+|T_{join}|(\log(|T_{join}|)+1))$, where the term $|T_{join}|(\log(|T_{join}|+1))$ 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, according to Theorem 1 clause 3 and clause 6, the total computation cost is upper-bounded by $(3n+2(\log n+|T_{join}|)+|T_{join}|(\log(|T_{join}|)+1))C_{me}$. Similarly, the total communication cost in term of unicast is upper-bounded by $(2.5n+|T_{join}|\log(|T_{join}|))C_{unicast}$ if $T_1 \geq n/2$ and is upper-bounded by $(3n+|T_{join}|\log(|T_{join}|))C_{unicast}$ if $T_1 < n/2$.

If condition 4 is satisfied, which is a very rare event, at most $(n+\log n)C_{me}$ 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 most $nC_{unicast}$ extra communication cost in term of unicast is needed.

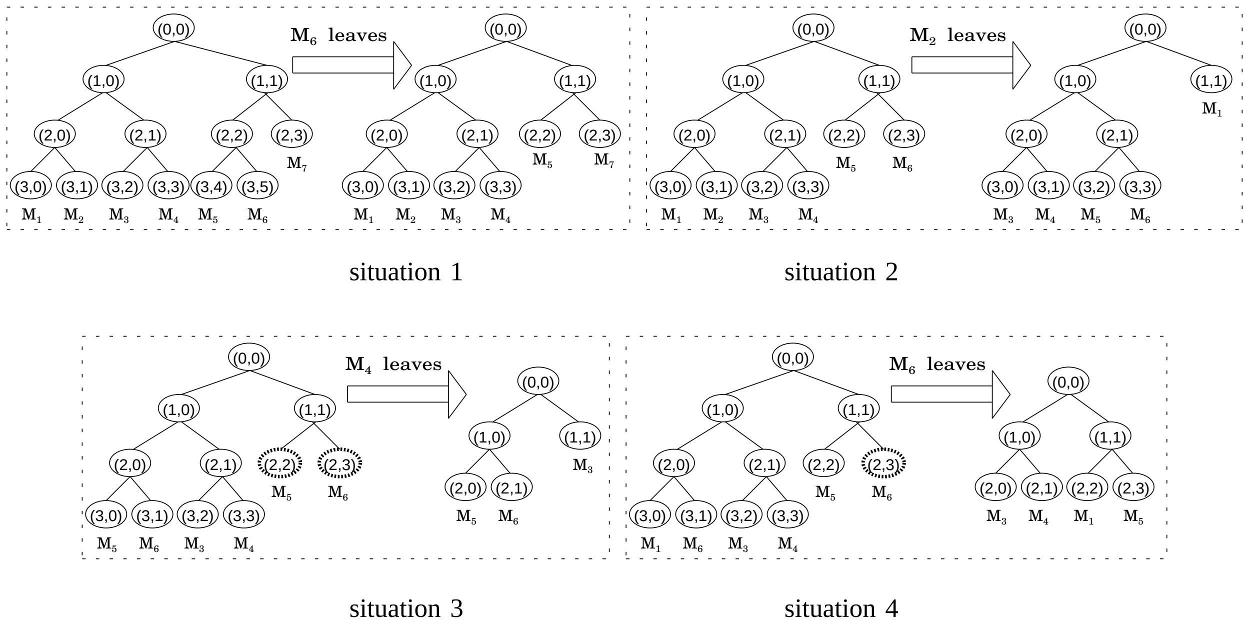


Fig. 6. Examples of key tree update upon single user leave under the second scenario

Fig. 6 shows four examples of key tree update upon single user leave under this scenario.

- The first example corresponds to situation 1: the leaving member M_6 is in the join tree, and the size of join tree with root $(1,1)$ is larger than $\log n$. In this example, after removing M_6 and applying the *split* procedure, 3 full key trees (subgroups) are obtained: $\{M_1, M_2, M_3, M_4\}$, $\{M_5\}$, $\{M_7\}$. The result of *unite* procedure has also been demonstrated.

- The second example corresponds to situation 2: the leaving member M_2 is in the main tree with root $(1,0)$, and the size of join tree with root $(1,1)$ is larger than 1. In this case after removing M_2 and applying *split*, three full key trees are obtained: $\{M_3, M_4\}$, $\{M_5, M_6\}$, $\{M_1\}$. The result of *unite* has also been illustrated in the right side of the figure.
- The third 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 *split*, 2 full key trees are obtained: $\{M_5, M_6\}$ and $\{M_3\}$. The result of *unite* has also been illustrated in the right side of the figure.
- The fourth 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 *split*, 3 full key trees are obtained: $\{M_3, M_4\}$, $\{M_1\}$ and $\{M_5\}$. The result of *unite* has also been illustrated in the right side of the figure.

SCENARIO III: 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; 2) the join tree is empty, and M is in the main tree and is not the right-most non-phantom node and does not have phantom node in the key tree. Under scenario III, the leaving member M is removed from the key tree, and M', which is the member who occupies the right-most 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, M' occupies two positions, and the original position is called M''s phantom position. It is easy to check that the time cost is bounded by $\log n$, the communication cost in term of multicast is bounded by $2(\log n)C_{multicast}$, the computation cost is upper-bounded by $(n + 2|T_{left}| + \log n)C_{me}$, where T_{left} is T_{main}'s left subtree, and the total communication cost in term of unicast is upper-bounded by $(n + 2|T_{left}|)C_{unicast}$.

Fig. 7 shows one example of key tree update upon single user leave under this scenario. In this example the join tree is empty and the root of 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 is $\{(2,0),(1,0),(0,0)\}$.

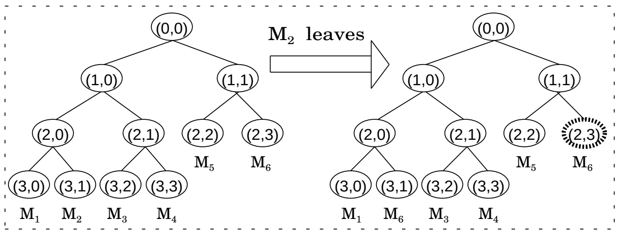


Fig. 7. Example of key tree update upon single user leave under the third scenario

TABLE II

REKEYING COST BOUNDS UPON SINGLE USER LEAVE EVENT

	time cost (rounds)	communication cost ($C_{multicast}$)	communication cost ($C_{unicast}$)	computation cost (C_{me})
Scenario 1	$O(h)$	$O(2h-2)$	$O(n+0.5\|T_{join}\|^2)$	$O(n+0.5\|T_{join}\|^2)$
Scenario 2	$O(\log n)$	$O(2\log n+2\|T_{join}\|)$	$O(2.5n+\|T_{join}\|\log(\|T_{join}\|))$	$O(2.5n+\|T_{join}\|\log(\|T_{join}\|))$
Scenario 3	$O(\log n)$	$O(2\log n)$	$O(n+2\|T_{left}\|)$	$O(n+2\|T_{left}\|)$

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

C. Group Merge and Group Partition Protocol

PACK also has group merge and group partition protocol to handle simultaneously 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 5 describes the group merge protocol, which combines two or more groups into a single group, and returns a PF key tree. Procedure 6 describes the group partition protocol, which removes multiple group member simultaneously from the current group and construct a new PF key tree for the rest of the group members.

In the group merge protocols, after removing all phantom nodes from those key trees corresponding to different subgroups, each key tree is split into many full key trees. The final result is obtained by unite these full key trees into a PF tree following Procedure 1. Similar for the group

Procedure 5 $merge(\{\mathcal{G}_1, \ldots, \mathcal{G}_K\})$

```
    ▷ $T_1, \ldots, T_K$ are the key trees of $\mathcal{G}_1, \ldots, \mathcal{G}_K$;
    Remove all *phantom* nodes from $T_1, \ldots, T_K$;
    $T = unite(split(T_1) \bigcup \cdots \bigcup split(T_K))$;
    Return $T$.
```

Procedure 6 $Partition(\mathcal{G}, \mathcal{G}_1)$

```
    ▷ $T$ is the key tree of $\mathcal{G}$;
    Remove all *phantom* members and members belonging to group $\mathcal{G}_1$ from $T$;
    $T = unite(split(\mathcal{T}))$;
    Return $T$.
```

partition protocol, after removing all phantom nodes and leaving nodes, the original key tree is split into many 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 group merge/partition is bounded by $O(\log n)$. Obviously, the group merge and partition protocols have lower cost than the sequential implementations.

V. Performance Evaluation and Comparison

A. Forward and Backward Security

The group key secrecy means that attackers cannot obtain the group key even if they know all blind keys, which has been proved in the random-oracle model [23]. To show that PACK satisfies forward and backward secrecy, similar arguments as in [9] can be used, which have provided detailed proof for TGDH. PACK and TGDH use similar group key update procedures. The major difference between them are the underlying key tree structures which do not affect the security of the scheme. Therefore in this paper we will not provide 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 two-group DH, M gets all blinded keys on its co-path, and it can compute all secret keys on its key-path using its own secret share and the blinded keys on its co-path. Clearly, all these keys contain M's secret share; hence they are independent of previous secret keys on that path. Therefore, M cannot derive any previous keys. The forward secrecy can be shown in a similar

TABLE III

REKEYING COST COMPARISON AMONG DIFFERENT SCHEMES

	time cost	communication cost	computation cost
Upon Single User Join Event			
PACK	$1 \sim 2$	$2 \sim 4C_{multicast}$	nC_{me}
TGDH	$\log n$	$2(\log n)C_{multicast}$	$2nC_{me}$
DST	$1 + \log\log n$	$(1 + \log\log n)C_{multicast}$	$(n + \log n)C_{me}$
Upon Single User Leave Event			
PACK	$\log n$	$2(\log n)C_{multicast}$	$(1 \sim 2.5)nC_{me}$
TGDH	$\log n$	$2(\log n)C_{multicast}$	$2nC_{me}$
DST	$1 + \log n + \log\log n$	$2(1 + \log n + \log\log n)C_{multicast}$	$3nC_{me}$

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 most all 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.

B. Cost Comparison

This section compares the rekeying cost in PACK upon single user join and leave events with two existing tree-based contributory group key agreement schemes: TGDH [9] and DST [10]. All three types of cost are considered: time, computation, and communication in term of multicast. Since in general members' leaving time is not known in advance, in DST, only join-tree is used. Table III lists the approximate bounds of different cost for the three schemes.

From the above comparison, we can see that PACK has the lowest cost in terms of time, computation, and communication. For example, for user join, only 1 or 2 rounds are needed in time cost, while DST needs $1 + \log\log n$ rounds and TGDH needs $\log n$ rounds. Similar results can also be seen in communication cost for user join. The total computation cost is computed as the average of user join cost and leave cost, DST has similar cost as TGDH, which is an order of $2n$, while for PACK, the order is from n to $1.75n$, with the saving ranging from 15% to 50% compared with DST and TGDH.

C. Simulation Results

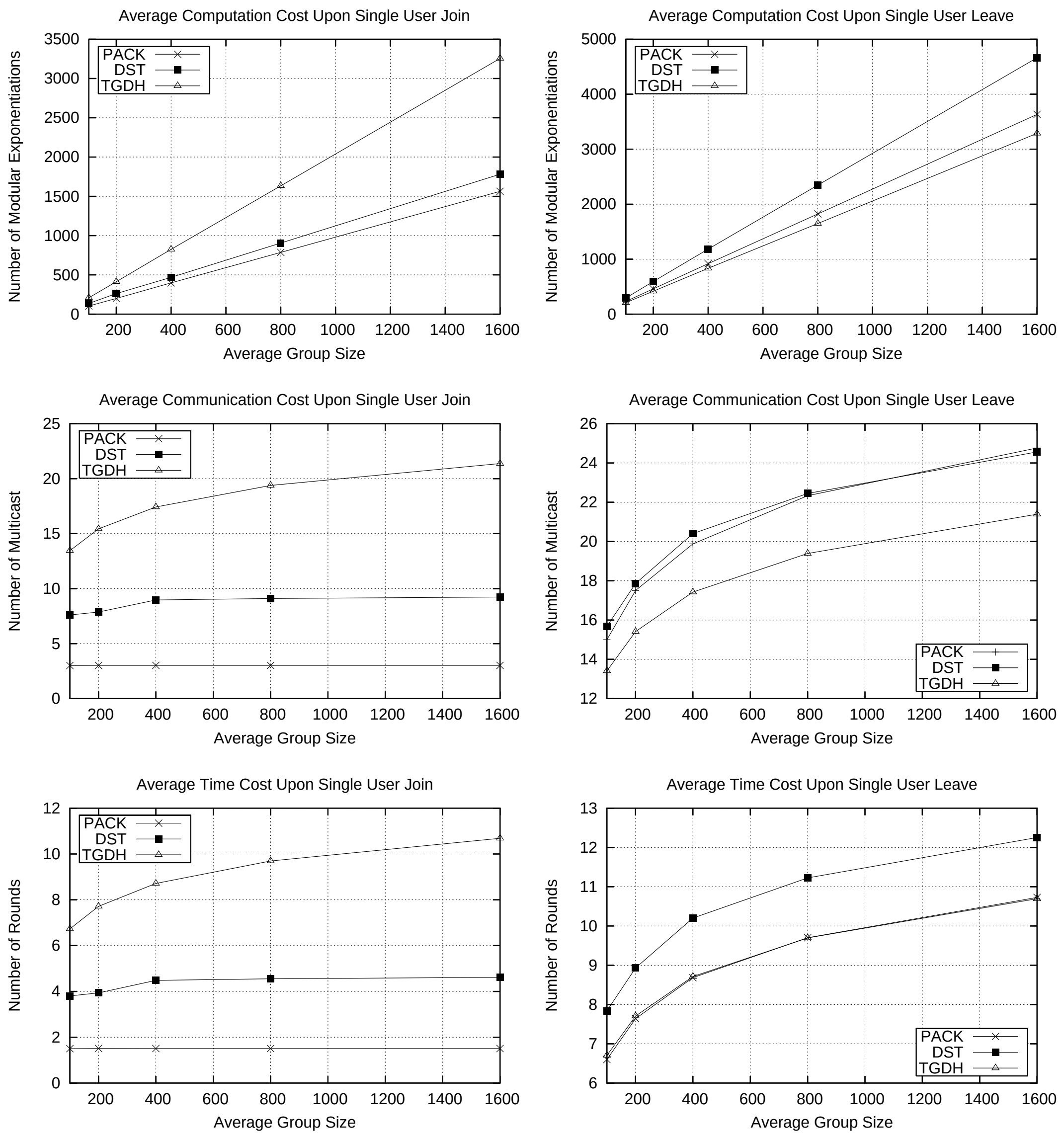


Fig. 8. Comparison of rekeying cost among PACK, TGDH and DST

In our simulations we generate the user activities according to the following probabilistic models: users join the group according to a Poisson process with average arrival rate λ, and users' staying time in the group follows an exponential distribution with mean μ (such a model is motivated by the user statistics in study of Mbone [24], [25]). 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 λ, and vary μ to get different average group size configuration. For each configuration (different average group size), a sequence of $100\lambda\mu$ users join the group according to the Poisson process with rate λ, and each user's staying time is drawn independently from an exponential distribution with mean μ. In the simulations, we have compared the rekeying cost of the following three schemes: PACK, TGDH [8] and DST [10], in all three aspects: computation, communication and time.

The simulation results are presented in Fig. 8. From these results we can see that upon single user join event, PACK has the lowest cost among all three schemes. Compared with DST, PACK has more than 10% reduction in computation cost, and 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 cost. Upon single user leave event, compared with DST, PACK has about 25% reduction in computation cost, about 15% reduction in time cost, and has similar communication cost. Although PACK has slightly higher computation and communication cost than TGDH upon single user leave event, when averaged over both join and leave events, the reduction is still significant, with 20% reduction in computation cost, 35% reduction in communication cost, and 40% reduction in time cost.

VI. Contributory Group Key Agreement with Key Validation

In practice, there may exist malicious or compromised group members who do not perform 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. Therefore, besides the four security requirements discussed in Section II, the group key management should also have the key validity property. That is, without being detected by other users, malicious users cannot prevent valid group key from being generated by providing false information. In this section, we discuss the possible damage that untruthful users can cause and the mechanisms to check the key validity.

When implementing the two group DH using the method described in Section II, an untruthful member can cause key generation failure only if it has been elected as a delegate. In this case, an untruthful member, e.g., a in subgroup $\mathbf{A}$, can send false blinded key $f(K'_A)$ to selected members in subgroup $\mathbf{B}$. As a consequence, those members in $\mathbf{B}$ who have received false blinded keys from a cannot obtain the valid group key K_{AB}, that is, these members have been implicitly revoked from the new group.

We introduce two methods to check the validity of 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 and broadcast the blinded key. Then each group member checks whether these m copies of blinded keys are same. Since all the keying messages have been signed by the senders, the member who has sent false information can be easily detected by other group member. In the detective scheme, after the 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 false blinded key or false encrypted messages can be detected.

Although colluders can compromise both preventive and detective schemes, the probability of successful collusion attack is very low because those m delegates or m users who broadcast the encrypted message are randomly selected. In addition, the detective method are more resistant to collusion attacks than the preventive methods. In the preventive method, the m delegates are selected within one subgroup, while in the preventive method, the m users are selected from both subgroups.

Key validation requires extra cost. In each round of two-group DH, the preventive scheme require $2m$ broadcast and the reactive scheme require m broadcast, m encryption and n decryption, where n is the size of the new subgroup after the DH round. It is noted that the extra cost due to checking is proportional to the cost of the key management schemes without the checking schemes. Thus, in previous analysis and comparisons, we did not count the extra cost associated with key validation.

VII. Conclusion

In this paper, we designed PACK, a highly efficient contributory key agreement scheme that has much lower communication, computation and time overhead then existing schemes and achieves performance lower bound derived in [22]. PACK reduces the overhead associated with key updating in two ways. First, it uses the novel PFMH tree structure that consist of a main tree, which is optimal for user leave, and a join tree, which is optimal for user join. Second, the concept of phantom user location in the PFMH allows the cost amortization when handling user leave. Upon single user join, PACK has the time cost as 1 or 2 rounds of two-group DH, the communication cost as 2 or 4 multicast, and the average computation cost as 1 modular exponentiation per user. Upon single user leave event, PACK takes at most $\log n$ rounds of 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, where n is the current group size. The performance of the PACK is compared with TGDH and DST. Both theoretical bound analysis and simulation results have shown that PACK has much lower rekeying cost in terms of communication, computation and cost than existing schemes.

APPENDIX

Proof of Theorem 1:

1) Consider the worst-case scenario: $L = n$, that is, $|T_l| = 1$ for all l. Then Procedure 1 works as follows: in the first round, the set of group members are partitioned into $\lceil n/2 \rceil$ subgroups, with each subgroup consisting of 1 or 2 members. For any subgroup of size 2, two-group DH is performed between the two members in this subgroup to generate a new key tree of size 2. In the i^{th} round, the set of existing key trees are partitioned into $\lceil n/2^i \rceil$ subgroups, with each group consisting of 1 or 2 existing key trees. If there is a subgroup consisting of only one existing key tree, then this key tree must have the minimum size (largest index) among all the existing trees. For any subgroup with two existing key trees, two-group DH is performed between these two key trees to generate a new key tree with its right child be the key tree which has smaller size (larger index). Repeat this procedure until only one tree is left, which is the final PF key tree. Since there are only n group members, at most $\log n$ rounds are needed, so the time cost is upper-bounded by $\log n$. For other scenarios where there exists $|T_i| \neq 1$, the time cost is always no more than $\log n$, since in these cases T_i can be viewed as the result of merging all the leaf nodes in T_i without introducing any time cost.

2) Since we need and only need to perform $L - 1$ times of two-group DH protocols to unite L full key trees into one PF tree, and since each two-group DH protocol needs 2 multicast in communication cost provided that the exchange of keying material between two subgroups during performing two-group DH is implemented using multicast, the total communication cost

is always upper-bounded by $2(L-1)C_{multicast}$.

3) According to Procedure 1 we know that at most $\log n$ rounds of two-group DH need to be performed in this situation. At the first round each member calculates its blinded key and a new subgroup key. At i^{th} round ($i > 1$), at most $\lceil n/2^{i-1} \rceil$ users (which are selected as delegates) need to calculate blinded keys and at most n users need to calculate their subgroup keys. Following this analysis we can see that the total computation cost is upper-bounded by $n(\log n + 2)C_{me}$. Further, if the exchange of keying material between two subgroups during performing two-group DH is implemented using unicast, it is easy to see that a blinded key needs to send to a certain member if and only if this member needs to calculate a key for a newly generated subgroup that it belongs to, which is equivalent to say that the total communication cost is upper-bounded by $(n \log n)C_{unicast}$ in this situation.

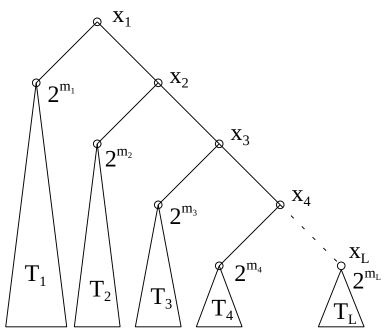


Fig. 9. Obtained PF key tree after applying *unite* procedure

4) In this special situation, according to the definition of PF tree, it is easy to check that the key tree illustrated in Fig. 9 is the obtained PF key tree after applying Procedure 1. Assume that the size of each full subtree T_i is 2^{m_i}, and let x_i denote both the PF subtree and its size. According to Procedure 1, we know that the PF subtrees $x_L, \ldots, x_1$ are generated sequentially with x_L first (directly from $\mathcal{T}$) and x_1 last. Also, when x_i is generated, at most $x_i + 2$ modular exponentiation operations are needed, so the total computation cost is upper-bounded by

$$C_{me} \sum_{i=1}^{L} (x_i + 2) = C_{me}(2 \log n + \sum_{i=1}^{L} x_i). \tag{3}$$

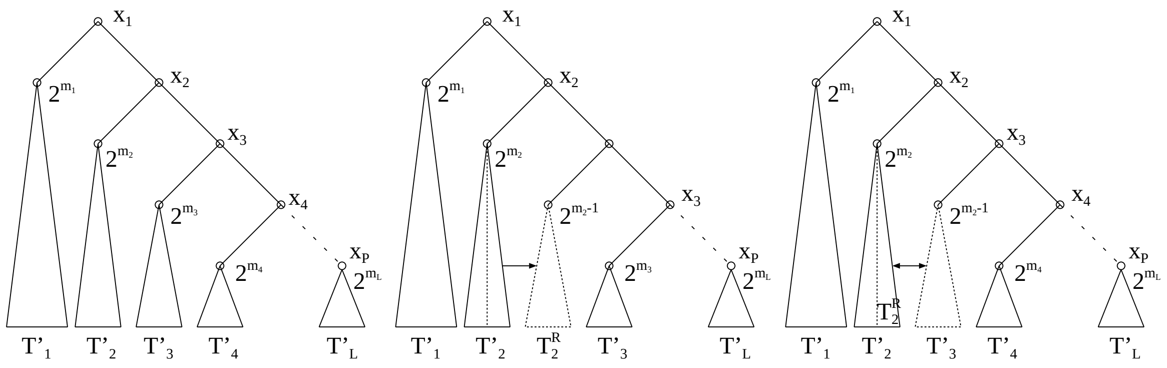
Since we have

$$x_i = 2^{m_i} + x_{i+1}, \tag{4}$$

$$x_i \geq 2x_{i+1}, \tag{5}$$

$$\textstyle\sum_{i=2}^{L} x_i < x_1 = n, \tag{6}$$

the above bound is further upper-bounded by $2(n + \log n)C_{me}$. Meanwhile, it is easy to check that the total communication cost is upper-bounded by $2nC_{unicast}$ in this situation provided that the exchange of keying material between two subgroups during performing two-group DH is implemented using unicast.



(a) Obtained PF key tree (b) Create a virtual subtree for T_2^R (c) Exchange T_2^R with T_3'

Fig. 10. Analysis of computation cost

5) In this special situation, let the key tree illustrated in Fig. 10 (a) be the PF tree obtained after applying the *unite* procedure. Now consider the computation cost incurred by the full subtree T_i':

- Case 1: If T_i' comes directly from the original set $\mathcal{T}$, the cost introduced by T_i' has been totally included in (3).
- Case 2: If T_i' is the merging result of two full trees directly from the original set $\mathcal{T}$ and each with size 2^{m_i-1}, compared with the first case, extra computation cost is needed to first merge the two trees into a single full tree. Since the total number of leaf nodes in T_i' is 2^{m_i}, and each leaf node needs 1 modular exponentiation to calculate the new subgroup key associated to T_i', the extra computation cost introduced by T_i' is $2^{m_i}C_{me} + 2$.
- Case 3: If T_i' is the merging result of more than two full key trees of the original set $\mathcal{T}$, since we have assumed that for each size the number of trees with this size in $\mathcal{T}$ is no more

than 2, then at least one child of T_i' with size 2^{m_i-1} comes directly from $\mathcal{T}$. Let T_i^L and T_i^R be the left and right child of T_I', and assume that T_i^L comes directly from $\mathcal{T}$. In this case, either there exists no key tree with size 2^{m_i-1} in the right side of T_i', or if there exists $|T_{i+1}'| = 2^{m_i-1}$, then T_{i+1}' must come directly from $\mathcal{T}$ in order not to violate the assumption that no more than two key trees in $\mathcal{T}$ have the same size, and T_{i+1}' will not introduce extra cost except those included in (3). If there exists no subtree with size 2^{m_i-1} in the right side of T_i', we add a virtual subtree T_i^R to the generated PF tree as in Fig. 10 (b) and move all the cost introduced by merging smaller full trees into this subtree. If $|T_{i+1}'| = 2^{m_i-1}$, we can simply exchange the subtree T_{i+1}' with the right subtree T_i^R of T_i' that is not directly from the original set $\mathcal{T}$, as in Fig. 10 (c). Now the total cost is kept to be the same but the extra cost introduced by T_i' is the same as in case 2.

Following the above analysis and the condition that $|T_1| \geq n/2$ (that is, T_1 comes directly from $\mathcal{T}$), the total extra computation cost that are not included in (3) is upper-bounded by $\sum_{i=1}^{m_2} 2^i C_{me}$. Now the total computation cost is upper-bounded by

$$2LC_{me} + C_{me}(\sum_{i=1}^{m_2} 2^i + \sum_{i=1}^{L} x_i) \tag{7}$$

By applying (4), (5), (6) and $|T_1| = 2^{m_1} \geq n$, we have

$$\sum_{i=1}^{m_2} 2^i + \sum_{i=1}^{L} x_i \leq 2^{m_2+1} + 2x_1 \leq x_1/2 + 2x_1 = 2.5x_1 = 2.5n \tag{8}$$

That is, the total computation cost is upper-bounded by $(2.5n + 2L)C_{me}$. Meanwhile, we can conclude that in this situation when the exchange of keying materials is implemented using unicast, the total communication cost is upper-bounded by $2.5nC_{unicast}$.

6) For the special situation that $|T_1| < n/2$ and for each tree $T_i \in \mathcal{T}$ there exists no more than 1 other tree in $\mathcal{T}$ with the same size as T_i, by following the same analysis as in (5), we can show that the total computation cost is upper-bounded by

$$2LC_{me} + C_{me}(\sum_{i=1}^{m_1} 2^i + \sum_{i=1}^{L} x_i), \tag{9}$$

where the only change from (7) to (9) is that m_2 is changed to m_1 due to the reason that T_1' does not come directly from $\mathcal{T}$.

By applying (4) we have that

$$\sum_{i=1}^{m_1} 2^i + \sum_{i=1}^{L} x_i = x_1 + (x_2 + 2^{m_1}) + \sum_{i=1}^{m_2} 2^i + \sum_{i=3}^{L} x_i \leq 2x_1 + 2^{m_2+1} + x_2 \leq 3x_1 = 3n \qquad (10)$$

That is, the total computation cost is upper-bounded by $(3n + 2L)C_{me}$. Meanwhile, we can conclude that in this situation when the exchange of keying materials is implemented using unicast, the total communication cost is upper-bounded by $3nC_{unicast}$.

End of proof.

References

[1] I. Ingemarsson, D. T. Tang, and C. K. Wong, “A conference key distribution system,” *IEEE Transactions on Information Theory*, vol. IT-28, no. 5, pp. 714–720, Sep. 1982.

[2] D. G. Steer, L. Strawczynski, W. Diffie, and M. Wiener, “A secure audio teleconference system,” in *Proceedings on Advances in cryptology*, 1990, pp. 520–528.

[3] M. Burmester and Y. Desmedt, “A secure and efficient conference key distribution scheme,” *Advances in Cryptology-Eurocrypt*, pp. 275–286, 1994.

[4] M. Steiner, G. Tsudik, and M. Waidner, “Diffie-Hellman key distribution extended to group communication,” in *ACM Conference on Computer and Communication Security*, 1996, pp. 31–37.

[5] K. Becker and U. Wille, “Communication complexity of group key distribution,” in *ACM Conference on Computer and Communication Security*, 1998, pp. 1–6.

[6] G. Ateniese, M. Steiner, and G. Tsudik, “Authenticated group key agreement and friends,” in *ACM Conference on Computer and Communication Security*, 1998, pp. 17–26.

[7] M. Steiner, G. Tsudik, , and M. Waidner, “Key agreement in dynamic peer groups,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, no. 8, pp. 769–780, Aug 2000.

[8] Y. Kim, A. Perrig, and G. Tsukid, “Simple and fault-tolerant key agreement for dynamic collaborative groups,” in *ACM Conference on Computer and Communication Security*, Athens, Greece, May 2000.

[9] Y. Kim, A. Perrig, and G. Tsudik, “Tree-based group key agreement,” *ACM Transactions on Information and System Security*, vol. 7, no. 1, pp. 60–96, Feb. 2004.

[10] Y. Mao, Y. Sun, M. Wu, and K. J. R. Liu, “Dynamic join-exit amortization and scheduling for time-efficient group key agreement,” in *IEEE INFOCOM*, 2004.

[11] Y. Amir, Y. Kim, C. Nita-Rotaru, J. L. Schultz, J. Stanton, and G. Tsudik, “Secure group communication using robust contributory key agreement,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 5, pp. 468–480, May 2004.

[12] G. H. Chiou and W. T. Chen, “Secure broadcasting using the secure lock,” *IEEE Transactions on Software Engineering*, vol. 15, pp. 929–934, Aug 1989.

[13] S. Mittra, “Iolus: A framework for scalable secure multicasting,” in *ACM SIGCOMM*, 1997, pp. 277–288.

[14] C. K. Wong, M. Gouda, and S. S. Lam, “Secure group communications using key graphs,” in *SIGCOMM*, Sep. 1998.

[15] D. M. Wallner, E. J. Harder, and R. C. Agee, “Key management for multicast: issues and architectures,” Internet Draft Report, Sept. 1998, Filename: draft-wallner-key-arch-01.txt.

[16] M. J. Moyer, J. R. Rao, and P. Rohatgi, “A survey of security issues in multicast communications,” *IEEE Network*, vol. 13, no. 6, pp. 12–23, Nov.-Dec. 1999.

[17] 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*, vol. 17, no. 9, pp. 1614–1631, Sep. 1999.

[18] L. R. Dondeti, S. Mukherjee, and A. Samal, “DISEC: a distributed framework for scalable secure many-to-many communication,” in *Proceedings of Fifth IEEE Symposium on Computers and Communications*, 2000, pp. 693–698.

[19] A. Perrig, D. Song, and D. Tygar, “ELK, a new protocol for efficient large-group key distribution,” in *Proc. IEEE Symposium on Security and Privacy*, 2001, pp. 247–262.

[20] Y. Sun, W. Trappe, and K. J. R. Liu, “A scalable multicast key management scheme for heterogeneous wireless networks,” *IEEE/ACM Trans. on Networking*, vol. 12, no. 4, pp. 653–666, Aug. 2004.

[21] W. Diffie and M. Hellman, “New directions in cryptography,” *IEEE Transactions on Information Theory*, vol. IT-22, no. 6, pp. 644–654, Nov. 1976.

[22] Jack Snoeyink, Subhash Suri, and George Varghese, “A lower bound for multicast key distribution,” in *IEEE INFOCOM*, 2001.

[23] M. Bellare and P. Rogaway, “Random oracles are practical: A paradigm for designing efficient protocols,” in *ACM Conference on Computer and Communication Security*, 1993.

[24] K. C. Almeroth and M. H. Ammar, “Multicast group behavior in the internets multicast backbone (mbone),” *IEEE Communications Magazine*, pp. 124 – 129, June 1977.

[25] K.C. Almeroth, “A long-term analysis of growth and usage patterns in the multicast backbone (mbone),” in *Proceedings of the IEEE INFOCOM00*, March 2000, vol. 2, pp. 824 – 833.