Simple and Fault-Tolerant Key Agreement for Dynamic Collaborative Groups

Anonymous submission

Abstract

Secure group communication is an increasingly popular research area having received much attention in the last several years. The fundamental challenge revolves around secure and efficient group key management. While centralized methods are often appropriate for key distribution in large groups, many collaborative group settings require distributed key agreement techniques. This work investigates a novel approach to group key agreement by blending binary key trees with Diffie-Hellman key exchange. The resultant protocol suite is very simple, fault-tolerant and secure. In particular, the efficiency of our protocols significantly surpasses that of prior art.

1 Introduction

Fault-tolerant, scalable, and reliable communication services have become critical in modern computing. A major current trend is in converting traditional centralized services(such as file sharing, authentication, web, and mail) into distributed services spread across multiple systems and networks. These distributed applications and other inherently collaborative applications (such as conferencing, white-boards, shared instruments and command-and-control systems) are very difficult to implement. One common and successful approach to developing these types of applications is to use a reliable group communication platform as a base messaging service.

At the same time, the increasing popularity and diversity of distributed and collaborative applications prompts the need for security platforms geared specifically towards such applications. However, experience shows that security mechanisms for collaborative (or peer) groups tend to be both expensive and complex. This is in contrast to non-collaborative, one-to-many disseminatory groups such as those encountered in Internet-wide multicast.

Security requirements of collaborative groups present some interesting research challenges. In particular, key management, as the cornerstone of all other security services, represents the first and the most difficult obstacle. In the last decade, a number of key management solutions for peer groups have been proposed(See Section 8). Some are not truly applicable as they involve a key distribution center of some sort while others are insecure and most of the rest are simply too costly or too fragile.

In this work, we unify two important trends in group key management: 1) the use of so-called *key trees* to efficiently compute and update group keys and 2) the use of Diffie-Hellman key exchange hybrids to achieve

provably secure and fully distributed protocols. This yields a surprisingly simple, highly secure and very efficient key management solution actually composed of a single protocol. Moreover, the protocol is inherently robust by virtue of being able to cope with cascaded (nested) key management operations which can stem from tightly spaced group membership changes. We believe this to be an issue of independent interest.

The rest of this paper is organized as follows. Section 2 introduces our notation and terminology, section 3 explains our assumptions and requirements. We describe our TGDH (Tree-based Group Diffie-Hellman) protocols in section 5, and show refinements in section 6. Section 7 treats the security, complexity, and implementation issues. Finally, we describe the previous work in section 8.

2 Notation and Definitions

We use the following notations:

N	number of protocol parties (group members)
M_i	i-th group member; $i \in [1, N]$
r_i	M_i's session random
h	height of a tree
$\langle l, v \rangle$	v-th node at level l in a tree
T_i	member M_i's tree
$\widehat{T_i}$	member M_i's updated tree after membership operation
BK_i^*	set of member M_i's blinded keys, i.e. $\{BK_{\langle 0,0 \rangle}, BK_{\langle 1,0 \rangle}, \dots\}$.
p, q	prime integers
α	exponentiation base

Key tree was firstly introduced at [22]. Though key tree has been used for centralized key distribution systems, we used it for distributed key agreement protocols. Figure 1 is an example of our key tree. Note that the root is at level 0 and the lowest leaves are at level h. Since our tree is binary, every node has either two children or it is a leaf node. The nodes are denoted as $\langle l, v \rangle$, where $0 \leq v \leq 2^l - 1$ since each level l hosts at most 2^l nodes[1]. Every leaf node $\langle h, v \rangle$ is associated with a member M_i participating in the group communication. Each node $\langle l, v \rangle$ has key $K_{\langle l,v \rangle}$ and blinded key $BK_{\langle l,v \rangle} = \alpha^{K_{\langle l,v \rangle}}$. Every member M_i at node $\langle h, v \rangle$ knows every key along the path from $\langle h, v \rangle$ to $\langle 0, 0 \rangle$. Let's call that path as the user's *key-path*, and denoted as KEY_i^*. In figure 1, if member M_2 has the tree T_2, then M_2 knows every key $\{K_{\langle 3,1 \rangle}, K_{\langle 2,0 \rangle}, K_{\langle 1,0 \rangle}, K_{\langle 0,0 \rangle}\}$ in $KEY_2^* = \{\langle 3, 1 \rangle, \langle 2, 0 \rangle, \langle 1, 0 \rangle, \langle 0, 0 \rangle\}$ and $BK_2^* = \{BK_{\langle 0,0 \rangle}, BK_{\langle 1,0 \rangle}, \dots, BK_{\langle 3,7 \rangle}\}$. Every key $K_{\langle l,v \rangle}$ is of the form $\alpha^{K_{\langle l+1,2v \rangle} K_{\langle l+1,2v+1 \rangle}}$, and $K_{\langle 0,0 \rangle}$ is the group secret shared by all members. For example, the group key $K_{\langle 0,0 \rangle}$ in figure 1 is

$$K_{\langle 0,0 \rangle} = \alpha^{(\alpha^{r_3 \alpha^{r_1 r_2}})(\alpha^{r_4 \alpha^{r_5 r_6}})}.$$

[1] Even though tree is not balanced, we will use same numbering for convenience. In other words, if a node $\langle l, v \rangle$ has left and right child, then their indexes are $\langle l+1, 2v \rangle$ and $\langle l+1, 2v+1 \rangle$, respectively.

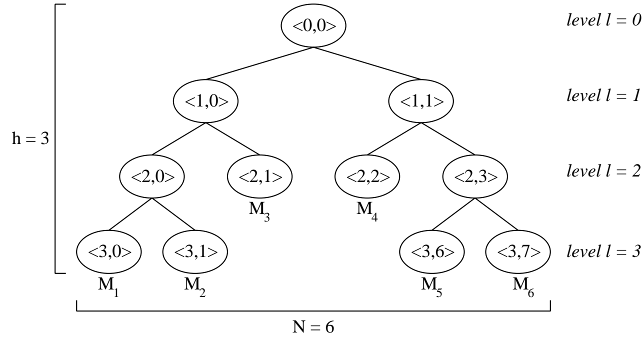

Figure 1: Notations for tree

As an example, M_2 can compute $K_{\langle 2,0\rangle}, K_{\langle 1,0\rangle}$ and $K_{\langle 0,0\rangle}$ using $BK_{\langle 3,0\rangle}, BK_{\langle 2,1\rangle}$ and $BK_{\langle 1,1\rangle}$. To simplify our subsequent protocol description, we introduce the term *co-path*, denoted as CO_i^*, which is the set of siblings of each node in the key-path on tree T_i of member M_i. In other words, every member M_i at leaf node $\langle l, v\rangle$ can derive the group secret $K_{\langle 0,0\rangle}$ from all blinded keys on the co-path CO_i^* and its session random $K_{\langle l,v\rangle}$.

In our protocol, a group member might take on a special role, which can involve to compute a key and to broadcast the blinded key to the group, for example. Any member in the group can take on this responsibility, we call this member *sponsor*.[2] The sponsor who handles the membership change is determined differently for each protocol. For example, for the join protocol, the member at the shallowest leaf node becomes a sponsor to handle the join.

3 Group Communication and Group Key Agreement

We noted in the introduction that many modern collaborative and distributed applications require a reliable group communication platform. The latter, in turn, needs specialized security mechanisms to perform – among other things – group key management. This dependency (or need) is mutual since practical group key agreement protocols themselves rely on the underlying group communication semantics for protocol message transport and strong membership semantics. Implementing multi-party and multi-round cryptographic protocols without such support is foolhardy as, in the end, one winds up reinventing reliable group communication tools.

In this section we begin with a brief discussion of reliable group communication. Next, we summarize the relationship between group membership events and group key management protocols and conclude with the summary of desired cryptographic properties.

[2] The terms group controller or group leader would be a misnomer because they are too strong in this context.

3.1 Group Communication Semantics and Support

There are two commonly used strong group communication semantics: Extended Virtual Synchrony (EVS) [14, 1] and View Synchrony (VS) [11]. Both guarantee that: 1) group members see the same set of messages between two sequential group membership events, and, 2) the sender's requested message order (e.g., FIFO, Causal, or Total) is preserved. VS provides a stricter service whereas EVS implementations are generally more efficient.

The main difference between EVS and VS is that EVS guarantees that messages are delivered to all receivers in the same membership as existed when the message was originally sent on the network. VS, in contrast, offers a stricter guarantee that messages are delivered to all recipients in the same membership as viewed by the sender application when it originally sent the message.

Providing the latter property requires an extra round of acknowledgment messages from all members before installing a new membership. This need for acknowledgments dictates that the groups be closed, only allowing members of the group to send messages to it. However, the knowledge that a message is received in the membership the sender believed it was sent in makes implementing secure group communication easier because every message is encrypted with the same key as the receiver believes is current when the message is delivered to them.

The implementation of any distributed fault-tolerant key agreement protocol requires VS. This is because, to implement a group key agreement protocol on top of EVS would require the key agreement protocol to incorporate and implement semantics identical to those of VS in order to correctly keep state of which messages were sent using in which key *epoch*. (Intuitively, this is because membership events are unpredictable and each triggers an instance of a key agreement protocol. Thus, multiple key agreement protocols can overlap in time and cause instability unless significant amount of state is kept within the key agreement protocol implementation.) For this reason, there is no particular benefit to building key agreement on top of EVS semantics.

The issues surrounding implementation of key agreement in dynamic peer groups are addressed in detail in [2]. Suffice it to say that, in the context of this paper we require for the underlying group communication to provide View Synchrony (VS). However, we also note that VS is needed for the sake of fault-tolerance and robustness; the security of our protocols is in no way affected by the lack of VS.

3.2 Group Membership Events

A comprehensive group key agreement solution must handle adjustments to group secrets subsequent to all membership change operations in the underlying group communication system.

We distinguish among single and multiple member operations. Single member changes include member addition or deletion. The former occurs when a prospective member wants to join a group and the latter occurs when a member wants to leave (or is forced to leave) a group. While there might be different reasons for member deletion – such as voluntary leave, involuntary disconnect or forced expulsion – we believe that group

key agreement must only provide the tools to adjust the group secrets and leave the rest up to the higher-layer (application-dependent) security mechanisms.

Multiple member changes also include addition and deletion. We refer to the multiple addition operation as *group merge*, in which case two or more groups merge to form a single group. We refer to the multiple leave operation as *group partition*, whereby a group is split into smaller groups. A group partition can take place for several reasons of two of which are fairly common:

1. Network failure – this occurs when a network event causes disconnectivity within the group. Consequently, a group is split into fragments some of which are singletons while others (those that maintain mutual connectivity) are sub-groups.

2. Explicit (application-driven) partition – this occurs when the application decides to split the group into multiple components or simply exclude multiple members at once.

Similarly, a group merge be either voluntary or involuntary:

1. Network fault heal – this occurs when a network event causes previously disconnected network partitions to reconnect. Consequently, groups on all sides (and there might be more than two sides) of an erstwhile partition are merged into a single group.

2. Explicit (application-driven) merge – this occurs when the application decides to merge multiple pre-existing groups into a single group. (The case of simultaneous multiple-member addition is not covered.)

At the first glance, events such as network partitions and fault heals might appear infrequent and dealing with them might seem to be a purely academic exercise. In practice, however, such events are common owing to network misconfigurations and router failures. In addition, in the environment of *ad hoc* wireless communication, network partitions are both common and expected. In [14], Moser et al. offer some compelling arguments in support of these claims. Hence, dealing with group partitions and merges is a crucial component of group key agreement.

In addition to the aforementioned membership operations, periodic refreshes of group secrets are advisable so as to limit the amount of ciphertext generated with the same key and to recover from potential compromises of members' contributions or prior session keys. This is discussed in the next section.

4 Cryptographic Properties

There are four important security properties encountered in group key agreement. (Assume that a group key is changed m times and the sequence of successive group keys is $\mathcal{K} = \{K_0, ..., K_m\}$).

1. Group Key Secrecy – this is the most basic property. It guarantees that it is computationally infeasible for a passive adversary to discover any group key.

2. Forward Secrecy – (not to be confused with Perfect Forward Secrecy or PFS) guarantees that a passive adversary who knows a contiguous subset of old group keys cannot discover subsequent group keys.

3. Backward Secrecy – guarantees that a passive adversary who knows a contiguous subset group keys cannot discover preceding group keys.

4. Key Independence – the strongest property. It guarantees that a passive adversary who knows a proper subset of group keys $\hat{K} \subset \mathcal{K}$ cannot discover any other group key $\bar{K} \in (\mathcal{K} - \hat{K})$.

The relationship among the properties is intuitive. Either of Backward or Forward Secrecy subsumes Group Key Secrecy and Key Independence subsumes the rest. Also, the combination of Backward and Forward Secrecy yields Key Independence.

Our definitions of Backward and Forward Secrecy are stronger than those typically found in the literature. The two are often defined (respectively) as [21, 17]:

- Previously used group keys must not be discovered by new group members.
- New keys must remain out of reach of former group members.

The difference is that the adversary here is assumed to be a current or a former group member. Our definition additionally includes the cases of inadvertently leaked or otherwise compromised group keys. We refer to the above as Weak Forward Secrecy and Weak Backward Secrecy, respectively.

In this paper we do not consider implicit key authentication as part of the group key management protocols. All communication channels are public but authentic. The latter means that (as mentioned later in the paper) that all messages are digitally signed by the sender using some sufficiently strong public key signature method such as DSA or RSA. All receivers are required to verify signatures on all received messages. Since no other long-term secrets or keys are used, we are not concerned with Perfect Forward Secrecy (PFS) as it is achieved trivially.

5 TGDH Protocols

In this section, we introduce the four basic protocols that form the TGDH protocol suite: join, leave, merge, and partition. These protocols all share a common framework with the following notable features:

- Each group member contributes its (equal) share to the group key, which is computed as a function of all shares of current group members.
- This share is secret (private to each group member) and is never revealed.
- As the group grows, new members' shares are factored into the group key but old members' shares remain unchanged.

- As the group shrinks, departing members' shares are removed from the new key and at least one remaining member changes its share[3].
- Current group members' shares are modeled as leaf nodes in a binary tree
- Each link (edge) in the tree is labeled $f(k)$ where k is the value of the node below the link
- Each non-leaf node is labeled $f(k_l * k_r)$ where k_l and k_r are the labels of the left and right child node, respectively
- The particular function $f()$ used in our protocols is modular exponentiation in prime-order groups, i.e., $f(k) = \alpha^k \pmod p$
- Computing the labeled value of a non-leaf node requires the knowledge of the value of one of the two child nodes and the value of the other incident link (i.e., link value emanating from the other child node).
- All protocol messages are signed by the sender. (We use RSA for this purpose).

Put another way, the nodes in the tree represent keys whereas the links represent blinded keys. Consequently, the root node is the group key. The actual key computation for each non-leaf node is simply an instance of Diffie-Hellman key exchange where the two incident links can be viewed as the messages exchanged in the process of the exchange. It follows that all link values (blinded keys) are public and all node values are secret. In effect, a key (node value) is known only to group members corresponding to the leaf nodes in the subtree formed by that node.

Upon each membership change, all members in the resulting group independently update the tree structure according to the strategy described in section 5.6. Since we assume that the underlying communication system provides *view synchrony* (see section 3), all members who correctly execute the protocol, recompute the identical key tree after a membership change.

In the event of an additive change (join or merge), all group members identify a particular group member. This special member, referred to as *sponsor* , is responsible for broadcasting all link values of the current tree to the incoming member(s). The common criteria for sponsor selection is determined by the tree maintenance strategy described in Section 5.6 below. We emphasize from the outset that sponsor is not a privileged entity; its only task is the broadcasting of current tree information to the joining members. In fact, barring efficiency considerations, any and all current members could perform this task.

In the event of a subtractive change (leave or partition), all members update the tree in the same manner. Since the case of partition is the most general– it subsumes the case of a single leave – we discuss it in more detail. In general, a partition yields a smaller tree as some leaf nodes disappear. As a result, some subtrees

[3]This prevents the group from reusing old keys. For example, if a member joins and immediately leaves, the group key would be the same before the join and after the leave. Although, in practice, this is not always a problem and might even be a desirable feature, we choose to err on the side of caution and change the key. In more concrete terms, changing the key upon all membership changes preserves key independence [21, 4].

acquire new siblings. Consequently, new keys must be computed through a Diffie-Hellman exchange between the new siblings. The computation proceeds in a bottom-up fashion with each member computing keys and blinded keys until it 1) cannot go further, i.e., a dependency exists on a heretofore un-computed sibling blinded key, or 2) it actually computes the new root (group) key. In (1), a node broadcasts all newly computed blinded keys to the rest of the group and waits for someone to broadcast a needed sibling blinded key. In (2), a node similarly broadcasts new blinded keys and proceeds to actually use the group key. This is repeated at most h times where h is the height of the tree, i.e., until all remaining members compute the new group key.

Fact 1. *Any member can compute the group key from its secret share and all the blinded keys on the co-path.*

Since each member knows, at the very least, its own secret share (and perhaps other keys on the key path to the root), it can compute the intermediate keys on their key path, and, eventually, the group (root) key. Similarly to other tree-based schemes [24, 23], each member knows all the keys on the path from its leaf to the root. Minimally, as expressed in fact 1, each member knows all the blinded keys on the co-path. In our protocol, however, each member knows all the blinded keys in the key tree, which makes the subsequent protocols we present more efficient.

Despite the separate descriptions which follow, there is actually only one protocol that handles all key adjustments (see section 6).

5.1 TGDH Membership Events

As discussed in section 3, a group key agreement scheme needs to provide key adjustment protocols stemming from membership changes. TGDH includes protocols in support of the following operations:

- Join: a new member is added to the group
- Leave: a member is removed from the group
- Merge: a subgroup is added to the group
- Partition: a subgroup is split from the group
- Key refresh: the group key is updated

The following sections (5.2 to 5.5), present the four protocols. In each section, we assume that every member can uniquely determine the sponsors and the insertion location in the tree (in case of an additive event). We present the details on how to identify sponsors and tree insertion points in section 5.6. Note that key refresh operation is, in fact, a special case of leave protocol, without leaving any members.

5.2 Join Protocol

Figure 2 depicts the join protocol. Assume that the group is made up of n users: $[M_1..M_n]$ and the current sponsor M_s resides at node $\langle l, v\rangle$.

The new member M_{n+1} initiates the protocol by sending a join request message that includes its own blinded key $BK_{\langle 0,0 \rangle}$.[4]

When current group members receive this message, they generate a new intermediate node and a new member node, and promote the new intermediate node to the parent of its node and the new member node. After updating the tree, only the sponsor can compute the group key, since it is the sibling of the joining node. After computing the group key, the sponsor broadcasts the new tree which contains all blinded keys. All other members update their tree using this message, and compute the new group key (see fact 1).[5]

Figure 3 shows an example of member M_4 joining to a group, where the sponsor M_3 performs the following actions:

1. Renames node $\langle 1,1 \rangle$ to $\langle 2,2 \rangle$,
2. generates a new intermediate node $\langle 1,1 \rangle$ and a new member node $\langle 2,3 \rangle$,
3. and promotes $\langle 1,1 \rangle$ as the parent node of $\langle 2,2 \rangle$ and $\langle 2,3 \rangle$.

Since all members know $BK_{\langle 2,3 \rangle}$ and $BK_{\langle 1,0 \rangle}$, M_3 can compute the new group key $K_{\langle 0,0 \rangle}$. Every other member performs step 1 and 2, but cannot compute the group key in the first round. Upon receiving the broadcast message, every member can compute the group key, since the broadcast message contains all blinded keys.

5.3 Leave Protocol

Assume that we have n members and a member M_d leaves the group. In this case, the sponsor is the sibling node of M_d. If the sibling is not a leaf node, the sponsor is the right-most leaf node of the subtree which has the sibling node as root of the subtree. In the leave protocol (figure 4) every member updates its key tree by deleting the node of M_d and its parent node. The sponsor picks a new secret share, computes all keys on its key path up to the root, and broadcasts the new blinded keys of its key path to the group.This information allows all members to recompute the group key.

Assuming the setting of figure 5, if member M_3 leaves the group, every member deletes node $\langle 1,1 \rangle$ and $\langle 2,2 \rangle$. After updating the tree, the sponsor M_5 picks a new key $K_{\langle 2,3 \rangle}$, recomputes $K_{\langle 1,1 \rangle}, K_{\langle 0,0 \rangle}, BK_{\langle 2,3 \rangle}$ and $BK_{\langle 1,1 \rangle}$, and broadcasts the updated tree $\widehat{T}_5$ with BK_5^*. Upon receiving the broadcast message, all members compute the group key, since the broadcast message contains every blinded key (see fact 1). Note that M_3 cannot compute the group key, though it knows all the blinded keys, because its share is no longer in the group key.

[4] We note that this message is separate from any JOIN messages generated by the group communication system although, in practice, the two might be combined for efficiency's sake.

[5] It might appear superfluous to broadcast the entire blinded key tree to all group members, since they know most of the blinded keys already. Our reason is that since the sponsor needs to send a broadcast message to the group, it might as well include more information which will be useful for the new member, therefore saving one unicast message to the new member which contains the blinded key tree.

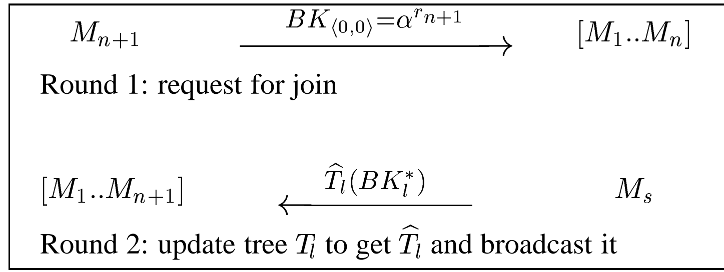

Figure 2: Join Protocol

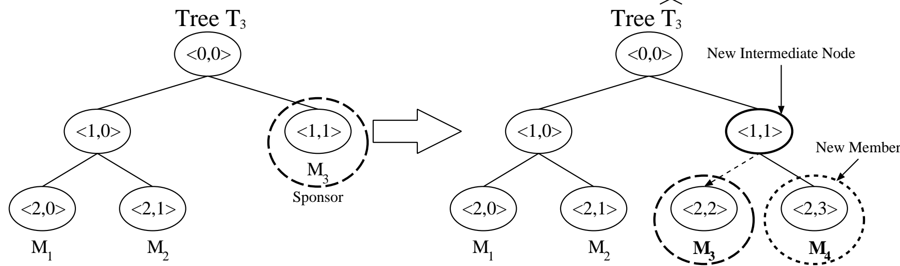

Figure 3: Tree updating in join operation

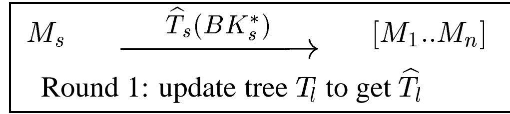

Figure 4: Leave Protocol

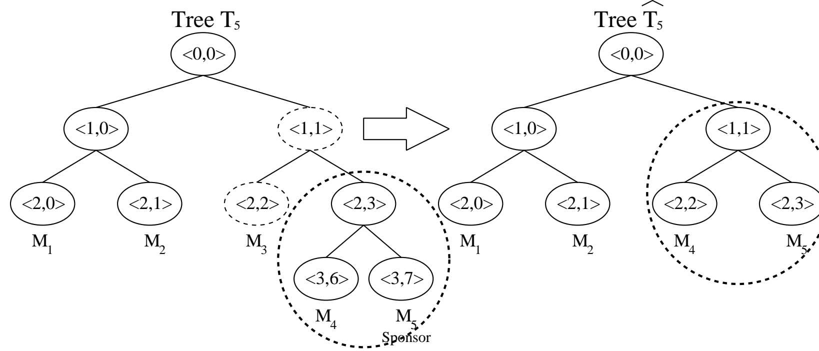

Figure 5: Tree updating in leave operation

5.4 Partition Protocol

Assume that a network fault occurs while n members $[M_1..M_n]$ have same key. From the viewpoint of each member, this event looks as a concurrent leave of multiple members. Our partition protocol is a multi-round protocol and it runs until every member knows the new group key.

In the first round, every remaining member updates its tree by deleting each leaving member and its parent node. Each member then computes the keys and blinded keys on its key-path as far up the tree as possible. When a member recomputes a new key $K_{\langle i,j \rangle}$, and it is at the same time the right-most member of the subtree rooted at node $\langle i,j \rangle$, then it broadcasts the new blinded key $BK_{\langle i,j \rangle}$ to the other group members.Upon receiving

this message, each member checks whether the message contains a new and necessary blinded key (the blinded key is necessary if it is on the co-path). This procedure iterates until all members know the group key.

To prevent reusing the old group key, one of the remaining members needs to change its key share. In the first round of the partition protocol the shallowest rightmost sponsor changes its share.

Figure 7 demonstrates an example, where all remaining members delete all nodes of leaving members and compute the keys and the blinded keys in the first round. In the figure on the right, M_5 and M_6 cannot compute the new group key, since they lack the blinded key $BK_{\langle 1,0\rangle}$. However, M_3 broadcasts $BK_{\langle 1,0\rangle}$ in the first round. Hence, every member knows all blinded keys and can compute the group key. As explained above, before computing $K_{\langle 1,1\rangle}$, M_6 changes its share $K_{\langle 2,3\rangle}$.

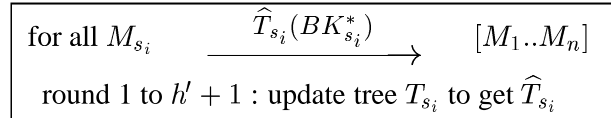

Figure 6: Partition Protocol

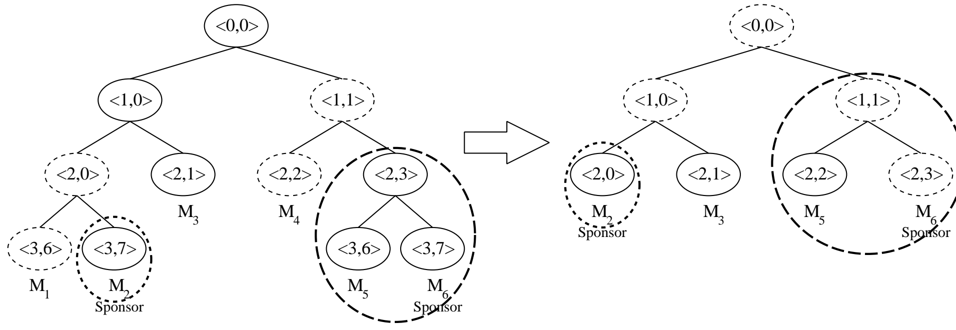

Figure 7: Tree updating in partition operation

If a member computes the group key at round h', then every other member can compute the group key at the latest in round $h'+1$, since the broadcast message contains every group key on the key tree. Hence, every member can detect the completion of the partition protocol independently.

5.5 Merge Protocol

As we discuss in section 3, network faults can partition a group into several subgroups. After the network faults heals, two or more subgroups may need to merge. In the first round of merge protocol, each sponsor broadcasts its tree information with all of its blinded keys to the other group. Upon receiving this message, all members can uniquely and independently determine the merge position of the two trees (for details see section 5.6). The rightmost member of the subtree rooted at the joining location becomes the sponsor of this operation. The sponsor computes every key on the key-path and the corresponding blinded keys. It, then, broadcasts the tree with the blinded keys to the other members.

Figure 9 shows an example, where the sponsors M_2 and M_7 broadcast their trees (T_2 and T_7) containing all the blinded keys, along with BK_2^* and BK_7^*. Upon receiving these broadcast messages, every member checks

whether it is the sponsor in the second round. Every member in both groups merges two trees, and then M_2, the sponsor in this example updates the key tree and computes and broadcasts blinded keys.

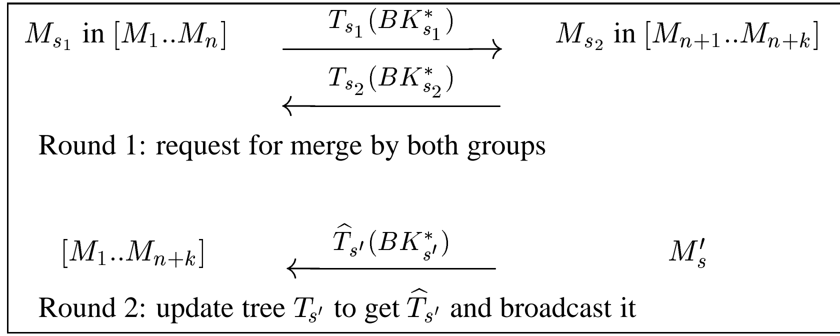

Figure 8: Merge Protocol

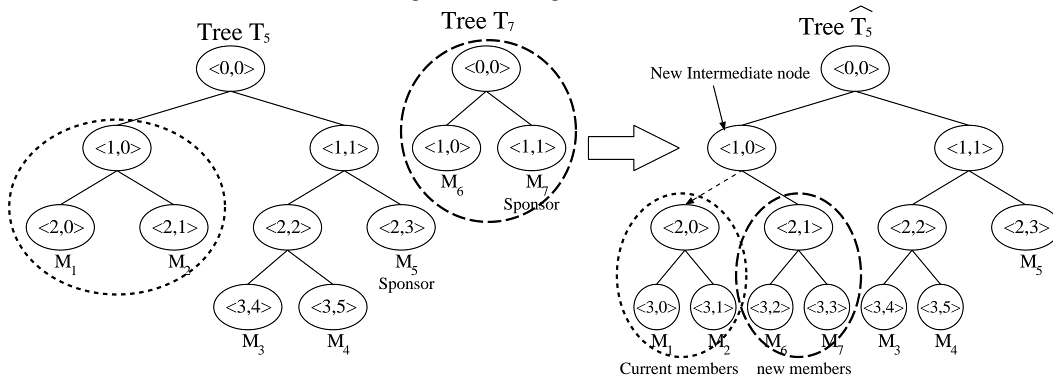

Figure 9: Tree updating in merge operation

5.6 Tree Management

Modular exponentiation is an expensive basic operation in TGDH. The number of exponentiations can vary for membership events, depending on the tree structure. For example, if a single member or a subtree merges to the root node of the current tree, then exactly two modular exponentiations are required. If a key tree is fully balanced, and a member joins to a leaf node, then the number of exponentiations is $\log_2 n$ where n is the current number of users. Hence, it is easy to see that joining to the root always requires the minimal number of exponentiations for additive membership operations. If n members join to the root, however, the resulting tree becomes unbalanced(similar to a linked list). If a member in the deepest node leaves the group, $n - 1$ exponentiations are required to update the group key. However, if a key tree is fully balanced, the number of exponentiations is $\log_2 n$. These examples show that a well-balanced key tree reduces the cost of leaves.

Considering these, our criteria for additive events are 1) join to the shallowest node (not necessarily a leaf node), if the join will not increase the height of the tree, and 2) join to root if join to any node will increase the height of the key tree. Since every member decides the joining points independently, we need just need pick one unambiguously. For example, if there are multiple joining points(same depth from the root), we simply choose the rightmost such node. If two tree of same height are joining, we compare hash value of first user's

name in each group. Note that we do not have criteria for subtractive events since we cannot determine who will leave (or will be partitioned out).

Currently we do not have any tree rebalancing scheme. However, it seems that the technique described in [15] can be used to rebalance our key tree. Further details about rebalancing will be described in the final version of this paper.

The main duty of the sponsor is 1) to broadcast newly computed blinded keys to other members, and 2) to broadcast key tree along with all blinded keys to the new members in case of additive events. The sponsor in the first round of the additive event is defined as the right-most leaf node of each tree, which performs task 2). Using this tree information, every member(including the new members) can uniquely determine the joining node according to the policy described above. The sponsor in the second round of join or merge protocol is the right-most child of the joining node. If the joining node is a leaf node, sponsor is the leaf node. In case of leave protocol, sponsor is the rightmost leaf node of the sibling of the leaving member. Only partition protocol has multiple sponsors. Any member who computes new blinded keys can be the sponsor. However, there can be also multiple such nodes who compute same new information. To prevent superfluous messages, we define sponsor as the rightmost node who computes the same new blinded keys.

6 Self-Stabilization and Fault Tolerance

In this section we address perhaps the most interesting (and important) feature of the proposed protocol suite, namely, self-stabilization.

6.1 Protocol Unification

Although described separately in the preceding sections, the four TGDH protocols: join, leave, merge and partition, actually represent different strands of a single protocol. We justify this claim with an informal argument below.

Obviously, join and leave are special cases of merge and partition, respectively. It is less clear that merge and partition can be collapsed into a single protocol. To see why this is so we observe that, in either case, the key tree changes and remaining group members lack some number (sometimes none) of blinded keys which prevents them from computing the new root key unilaterally. When a partition occurs, the remaining members (in any surviving fragment) reconstruct the tree where some blinded keys are missing. In case of a merge, let us suppose that a taller (deeper) tree $\mathcal{A}$ is merged with a shorter (shallower) tree $\mathcal{B}$. Similar to a partition, all members formerly in $\mathcal{A}$ construct the new tree where some blinded keys – those in $\mathcal{B}$ – are missing. (This view is symmetric since the members in $\mathcal{B}$ see the same tree but with missing blinded keys in the subtree $\mathcal{A}$.)

We established that both partition and merge initially result in a new key tree with a number of missing blinded keys. In case of merge, the missing blinded keys can be distributed in two rounds. This is because a sponsor in both of $\mathcal{A}$ and $\mathcal{B}$ broadcasts its own subtree including all blinded keys. Any member in a given subtree can compute the new root key after receiving both broadcasts. The case of partition is very similar

except that the missing blinded keys are not concentrated in a new subtree (as in merge) but are, in the most general case, spread all around. As we discuss in section 5.4, every member reconstructs the key tree after a partition in at most h rounds, where h is the tree height. The merge scenario can be viewed as a special case of partition that always completes in two rounds.

```
receive msg (msg type = membership event)
construct new tree
while there are missing blinded keys
  if (I can compute any missing keys) /* sponsor? */
    compute missing blinded keys      /* as many as possible */
    broadcast new blinded keys
  endif
  receive msg (msg type = broadcast)  /* including own broadcast */
  update current tree
endwhile
```

Figure 10: Unified protocol pseudocode

This apparent similarity between partition and merge allows us to lump the protocols stemming from all membership events into a single, unified protocol. (See figure 10 for the pseudocode.) The incentive for this is to simplify the implementation and minimize its size. Furthermore, the overall security and correctness are easier to demonstrate with a single protocol. Most importantly, however, we can now claim that (with a slight modification) the TGDH protocol is self-stabilizing and fault-tolerant as discussed below.

6.2 Cascaded Events

Since network disruptions are random and unpredictable it is natural to consider the possibility of so-called *cascaded membership events*. (In fact, this is typically done in group communication literature, but, alas, not often enough in the security literature.) A cascaded event occurs when a join, leave, merge or partition takes place while a prior event is being handled.

We claim that the TGDH partition protocol is self-stabilizing, i.e., robust against cascaded network events. This is quite rare as most multi-round cryptographic protocols are not geared towards handling of such events. In general, self-stabilization is a very desirable feature since lack thereof requires extensive and complicated protocol ”coating” to either 1) shield the protocol from cascaded events, or 2) harden it sufficiently to make the protocol robust with respect to cascaded events (essentially, by making it re-entrant).

The high-level pseudocode for the self-stabilizing protocol is shown in figure 11. The changes from figure 10 are minimal.

Instead of providing a formal proof of self-stabilization (which we omit due to submission page limitations) we demonstrate it with an example. Figure 12 shows an example of a cascaded partition event. The first part of the figure depicts a partition of M_1, M_4, and M_7 from the prior group of ten members $[M_1..M_{10}]$. This partition normally requires two rounds to complete the key agreement. As described in section 5.4, every

```
receive msg (msg type = membership event)
construct new tree
while there are missing blinded keys
  if (I can compute any missing keys) /* sponsor? */
    compute missing blinded keys      /* as many as possible */
    broadcast new blinded keys
  endif
  receive msg                         /* including own broadcast */
  if (msg type = broadcast)
    update current tree
  else (msg type = membership event)
    construct new tree
endwhile
```

Figure 11: Self-stabilizing protocol pseudocode

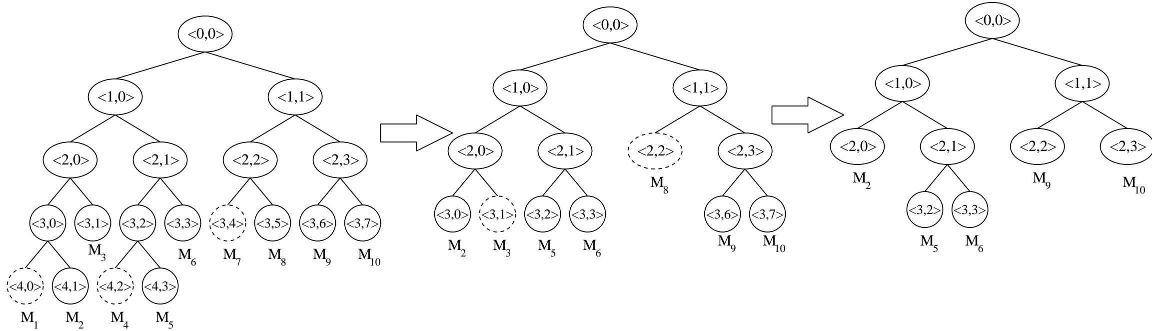

Figure 12: An Example of Cascaded Partition

member constructs the same tree after completing the initial round. The middle part shows the resulting tree. In it, all non-leaf nodes except $K_{\langle 2,3 \rangle}$ must be recomputed as follows:

1. First, M_2 and M_3 both compute $K_{\langle 2,0 \rangle}$, M_5 and M_6 compute $K_{\langle 2,1 \rangle}$ while M_8, M_9 and M_{10} compute $K_{\langle 1,1 \rangle}$. All blinded keys are broadcasted by each sponsor M_2, M_5 and M_8.

2. Then, as all broadcasts are received, M_2, M_3, M_5 and M_6 compute $K_{\langle 1,0 \rangle}$ and $K_{\langle 0,0 \rangle}$. The blinded keys are broadcasted by the sponsor M_6.

3. Finally, all broadcasts are received and M_8, M_9 and M_{10} compute $K_{\langle 0,0 \rangle}$.

Suppose that, in the midst of handling the first partition, another partition (of M_3 and M_8) takes place. Note that, regardless of which round (1,2,3) of the first partition is in progress, the departure of M_3 and M_8 does not affect the keys (and blinded keys) in the subtrees formed by M_9 and M_{10} as well as M_5 and M_6. All remaining members update the tree as shown in the rightmost part of figure 12. The blinded key of $K_{\langle 1,0 \rangle}$ is the only one missing in all members' view of the tree. It is computed by M_2, M_5 and M_6 and broadcasted. When the broadcast is received, all nodes compute the root key.

The only remaining issue is whether a broadcast from the first partition can be received after the notification of the second (cascaded) partition. Here we rely on the underlying group communication system to guarantee that **all membership events are to be delivered in sequence after all outstanding messages are delivered**. In other words, if a message is sent in one membership view and membership changes while the message is not yet delivered, the membership change must be postponed until the message is delivered to the (surviving) subset of the original membership. This is essentially a restatement of View Synchrony (as discussed in section 3).

7 Discussion

7.1 Security

We define our desired security properties in section 4. We now show that TGDH satisfies group key secrecy, and weak forward and backward secrecy.

First we show that TGDH satisfies group key secrecy by proving that even if an attacker knows all the blinded keys, it cannot derive the group key. We proved the group key secrecy in the random-oracle model [6], but due to size constraints, we cannot add it to this submission. For the purpose of this submission, we refer to the proof of Becker and Wille [5]. Their group key is very similar to our TGDH key, hence their proof is also applicable for our scheme. They show that the group key secrecy reduces to DDH [13].

We now give an informal argument that TGDH satisfies weak forward and backward secrecy. We first consider weak backward secrecy, which states that a new member which knows the current group key cannot derive any previous group key.

From the group key secrecy property we know that the group key cannot be derived from the blinded keys alone. At least one secret key K is needed to compute all secret keys from K up to the root key. Hence, we need to show that the joining member M cannot find any keys of the previous key tree. First, M picks its secret share r, blinds it and publishes α^r with its join request. Once M receives all the blinded keys on its co-path, it can compute all the secret keys on its key path. Clearly, all these keys will contain M's contribution (r) and they are hence independent of the previous secret keys on that path. Hence, M cannot derive any previous keys.

Similarly, we show that TGDH provides weak forward secrecy. When a member M leaves the group, the rightmost member of the subtree rooted at the sibling node will change its secret share, M's leaf node is deleted and its parent node is replaced with its sibling node. This operation causes that all of M's contribution is removed from every key on M's key path. Hence, M only knows all blinded keys, and the group key secrecy property prevents M from deriving the new group key.

7.2 Complexity Analysis

We discuss the communication and computation overhead for join and leave operations only, because these operations are more frequent than merge or partition operations.[6] We compare the protocols to the state-of-the-art authenticated group key agreement scheme A-GDH.2 described in [4]. Our worst-case computation complexity is based on the assumption that the maximum number of group members is $N = 2^d$, in which case the maximum height of the key tree is d. The cost for join is computed from the case when we have $N - 1$ members and a member is joining to the current group. The leave cost is when we have N members and one member leaves the group. We do not include the number of exponentiations for the long-term key computation (A-GDH) and signature/verification (TGDH). Table 1 shows the comparison. We would like to emphasize that these numbers represent worst-case numbers for TGDH. In practice, we can optimize the communication overhead of TGDH to $O(\log N)$, because only the keys on the key path of the joining or leaving member change, hence only these blinded keys need to be broadcasted, and not the entire key tree. For the computation overhead, TGDH offers a substantial saving for consecutive (serial or non-parallelizable) exponentiations, because every member only needs to perform at most $\log N$ exponentiations, which is much faster than the A-GDH.2 protocol. Note that since the joining point in the join and merge operation is located at a shallower node than the deepest node, the average join and leave cost is lower than the worst case. The savings for the total number of exponentiations, however, is smaller, so TGDH offers less of a benefit if all members run on the same workstation, which is a rare case.

Operations	Join		Leave	
Protocol	A-GDH	TGDH	A-GDH	TGDH
Communication cost				
Rounds	2	2	1	1
Broadcasts	1	2	1	1
Total messages	2	2	1	1
Maximum bandwidth	N	$2N$	$N-1$	$2N-2$
Computation cost				
Serial exponentiation	$2N+1$	$2(d-1)$	$N-1$	$2(d-1)$
Total exponentiation	$3N+2$	$2(N-2)$	$2N-3$	$2(N-d-2)$

Table 1: Communication Cost

7.3 Implementation

We developed a prototype implementation of our protocols, which includes the basic protocols (join, leave, partition, and merge). Our central goal is to develop a general-purpose toolkit for key agreement and related

[6] We defer the discussion of the merge and partition complexity to our final submission.

security services for dynamic collaborative groups. Applications include replicated Web-servers, collaborative simulations and voice conferencing. At first ties of this process, we developed a group key management API called TREE_API. Currently, all the basic protocols are implemented, and cascaded partition is handled by partition protocols. We use the OpenSSL 0.9.4 [16] cryptographic library.

In a second step, we implement the one-function approach described in section 6. The integration with a reliable group communication system is also planned.

We note that $f(x) = (\alpha^x \pmod p) \pmod q$ where we use $x \in \mathbb{Z}_q$ to enhance the efficiency of our protocol.

8 Previous Work

Secure group communication protocols come in two different flavors, namely contributory key agreement protocols for small groups and server-based key distribution protocols for large groups. The focus of this work is on group key agreement protocols, hence we emphasize on that work in this review.

Steer et al. propose a group key agreement protocol, referred to as STR [19], which is based on an extension of the Diffie-Hellman (DH) key agreement protocol [10]. Their protocol is of particular interest to us, since their group key has a similar structure as the protocols we propose:

$$K_n = \alpha^{N_n \alpha^{N_{n-1} \cdots N_3 \alpha^{N_1 N_2}}}.$$

STR is well-suited for adding new group members, which requires only two rounds and two modular exponentiations. Member exclusion, however, is difficult in STR (for example, consider excluding N_1 from the group key).

One interesting result was proposed by Burmester and Desmedt [7]. They construct an efficient protocol which requires only three rounds and two modular exponentiations per member to generate a group key. This efficiency allows all members to re-compute the group key for every membership change by performing this protocol.However, according to a recent result [21], most of the members need to change their session random on every membership event. Their group key is different from others:

$$K_n = \alpha^{N_1 N_2 + N_2 N_3 + \ldots + N_n N_1}.$$

Becker and Wille[5] analyze the minimal communication complexity of a contributory group key agreement protocol. Their group key has the same structure as the key we generate in our protocol. For example, for 8 users their group key is:

$$K_n = \alpha^{(\alpha^{\alpha^{r_1 r_2} \alpha^{r_3 r_4}})(\alpha^{\alpha^{r_5 r_6} \alpha^{r_7 r_8}})}.$$

Their protocol handles join and merge operations efficiently, but the member leave operation is inefficient. We further discuss this paper in section 7.1.

Tsudik et al. address dynamic membership issues [20, 4, 21]. Their key agreement protocol is based on a ring variant of the Diffie-Hellman key agreement, and provides contributory authenticated key agreement, key

independence, key integrity, resistance to known key attacks, and perfect forward secrecy. Their protocol suite is efficient in leave and partition operation, but the merge protocol requires as many rounds as the number of new members, to complete the key agreement.

Perrig extends the work of one-way function trees (OFT, introduced by McGrew and Sherman [12]) to design a tree-based key agreement scheme for peer groups [17]. This work, however, did not contain details on how to deal with real-world issues such as group partition or merge.

Many researchers focus on server-based group key distribution schemes for large groups [12, 9, 8]. These schemes address a different setting than what we propose, since the key agreement is not contributory, and group settings are different (i.e. not for dynamic peer group, but for hierarchical group settings). Their schemes are of interest in this context, because they are using key trees to address the group key distribution problem These protocols aim to keep a balanced key tree to achieve a low communication overhead for key distribution. Moyer, Rao, and Rohatgi [15] address the problem of how to balance the key tree. Their scheme can be applied to our scheme with slight modification of our protocols. Further details will be provided in the final version of this paper.

Rodeh et al. [18] propose a distributed group key distribution protocol. A particular group member chooses the group key and distributes it to all other members, hence the protocol does not offer contributory key agreement. Another drawback of their protocol is that it requires N secure two-party channels between group members, on which the new key is distributed with unicast. Maintaining these channels in dynamic groups can be expensive ($O(N)$ new channels need to be set up if the group leader leaves) since setting up each channel involves a two-party key agreement scheme. They also use a tree to distribute a group key, and propose how to balance the key tree.

Asokan et al. look at the problem of small-group key agreement, where the members do not have previously set up security associations [3]. Their motivating example is a meeting where the participants want to bootstrap a secure communication group. They adapt password authenticated DH key exchange to the group setting. Their setting, however, is different from ours, since they assume that all members share a secret password, whereas we assume a PKI where each member can verify any other members authenticity and authorization to join the group.

Waldvogel et al. consider a similar setting [9]. They propose efficient protocols for small-group key agreement and large-group key distribution. Unfortunately, their scheme for autonomous small group key agreement is not collusion resistant.

References

[1] Y. Amir. *Replication using Group Communication over a Partitioned Network.* PhD thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1995.

[2] Y. Amir, G. Ateniese, D. Hasse, C. Nita-Rotaru Y. Kim, T. Schlossnagle, J. Schultz, J. Stanton, and G. Tsudik. Secure group communication in asynchronous networks with failures: Integration and experiments. In *20th IEEE International Conference on Distributed Computing Systems (ICDCS)*, April 2000.

[3] N. Asokan and P. Ginzboorg. Key-agreement in ad-hoc networks. In *Nordsec'99*, 1999.

[4] G. Ateniese, M. Steiner, and G. Tsudik. Authenticated Group Key Agreement and Friends. In *5th ACM Conference on Computer and Communications Security*, pages 17–26. ACM, November 1998.

[5] C. Becker and U. Wille. Communication complexity of group key distribution. In *ACM conference on Computer and Communication Security*, November 1998.

[6] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proc. 1st ACM Conference on Computer and Communications Security*, 1993.

[7] M. Burmester and Y. Desmedt. A Secure and Efficient Conference Key Distribution System. In Alfredo De Santis, editor, *EUROCRYPT94*, pages 275–286. LNCS 950, 1994.

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

[9] G. Caronni, M. Waldvogel, D. Sun, N. Weiler, and B. Plattner. The VersaKey framework: Versatile group key management. *IEEE Journal on Selected Areas in Communications*, 17(9), September 1999.

[10] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Trans. Inform. Theory*, IT-22:644–654, November 1976.

[11] A. Fekete, N. Lynch, and A. Shvartsman. Specifying and using a partionable group communication service. In *Proceedings of the 16th annual ACM Symposium on Principles of Distributed Computing*, pages 53–62, Santa Barbara, CA, August 1997.

[12] D. McGrew and A. Sherman. Key Establishment in Large Dynamic Groups Using One-Way Function Trees, May 1998. `http://www.cs.umbc.edu/~sherman/Papers/itse.ps`.

[13] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[14] L. Moser, Y. Amir, P. Melliar-Smith, and D. Agarwal. Extended virtual synchrony. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, pages 56–65, Los Alamitos, CA, USA, June 1994. IEEE Computer Society Press.

[15] M. J. Moyer, J. R. Rao, and P. Rohatgi. *Maintaining Balanced Key Tree for Secure Multicast*, June 1999. draft-irtf-sumg-key-tree-balance-00.txt.

[16] OpenSSL Project team. Openssl, May 2000. http://www.openssl.org/.

[17] A. Perrig. Efficient collaborative key management protocols for secure autonomous group communication. In *International Workshop on Cryptographic Techniques and E-Commerce (CrypTEC '99)*, pages 192–202, 1999.

[18] O. Rodeh, K. Birman, and D. Dolev. Optimized rekey for group communication systems. In *NDSS2000*, pages 37–48, 2000.

[19] D. Steer, L. Strawczynski, W. Diffie, and M. Wiener. A secure audio teleconference system. In S. Goldwasser, editor, *Advances in Cryptology – CRYPTO '88*, number 403 in Lecture Notes in Computer Science, pages 520–528, Santa Barbara, CA, USA, August 1988. Springer-Verlag, Berlin Germany.

[20] M. Steiner, G. Tsudik, and M. Waidner. Diffie-hellman key distribution extended to groups. In *3rd ACM Conference on Computer and Communications Security*. ACM, November 1996.

[21] M. Steiner, G. Tsudik, and M. Waidner. Cliques: A new approach to group key agreement. *Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 380–387, 1998. Extended version available from `http://www.isi.edu/~gts/paps/stw99.ps.gz`.

[22] D. Wallner, E. Harder, and R. Agee. Key management for multicast: Issues and architecture. Internet-Draft draft-wallner-key-arch-00.txt, June 1997.

[23] D. Wallner, E. Harder, and R. Agee. Key Management for Multicast: Issues and Architectures. Technical report, IETF, September 1998. draft-wallner-key-arch-01.txt.

[24] C. Wong, M. Gouda, and S. Lam. Secure group communications using key graphs. Technical Report TR-97-23, University of Texas at Austin, Department of Computer Sciences, August 1997.