Simple and Fault-Tolerant Key Agreement for Dynamic Collaborative Groups

Yongdae Kim*
Computer Networks Division
USC/ISI
yongdaek@isi.edu

Adrian Perrig†
Computer Science Department
UC Berkeley
perrig@cs.berkeley.edu

Gene Tsudik*
Dept. of Information and Computer Science
UC Irvine
gts@ics.uci.edu

ABSTRACT

Secure group communication is an increasingly popular research area having received much attention in recent 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, secure and fault-tolerant. Moreover, its efficiency surpasses that of prior art.

1. INTRODUCTION

Fault-tolerant, scalable, and reliable communication services have become critical in modern computing. An important trend is to convert traditional centralized services (e.g., file sharing, authentication, web, and mail) into distributed services spread across multiple systems and networks. Many of these newly distributed and other inherently collaborative applications (e.g., conferencing, white-boards, shared instruments, and command-and-control systems) need secure communication. However, experience shows that security mechanisms for collaborative, dynamic peer groups tend to be both expensive and unexpectedly complex. Note that dynamic peer groups are different from non-collaborative, centrally managed, one-to-many broadcast groups such as those encountered in Internet multicast.

Although peer groups tend to be relatively small, group members may be spread throughout the Internet and must be able to deal with arbitrary partitions due to network failures, congestion, and hostile attacks. In essence, a group can be split into a number of disconnected partitions each of which must persist and function as an independent and secure peer group.

Security requirements of collaborative peer groups present interesting research challenges. Key management, as the cornerstone of most other security services, presents the initial and formidable obstacle. Although centralized key management might initially appear attractive, it is inherently unsuitable for dynamic peer groups. The reasons are as follows.

First, centralization violates the peer nature of the group by concentrating all key generation in a single point, hence centralizing trust, and replacing key agreement with key distribution. Second, a centralized key server becomes both a single point of failure and an attractive attack target. Of course, a key server can be sufficiently replicated and fortified to address these issues. However, we claim that it is very costly (if not altogether impossible) to guarantee the availability of a key server in any and all possible partitions of a network, thus, each group member must be prepared to become a key server. This raises a policy issue as far as the criteria for selecting a member to act as a key server. Furthermore, each new key server needs to establish a pairwise secure channel with every other group member in order to distribute keys. This can become prohibitively expensive.

For the above reasons, we focus on contributory key agreement. 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 group Diffie-Hellman key exchange to achieve provably secure and fully distributed protocols. This yields a secure, surprisingly simple and efficient key management solution. Moreover, the resulting protocol suite 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 of the reliable group communication system, while section 4 introduces the cryptographic requirements of our group key agreement protocol. The actual protocols are described in section 5 and refinements are discussed in section 6. Section 7 treats the se-

*Research supported by the Defense Advanced Research Project Agency, Information Technology Office (DARPA-ITO), under contract DABT63-97-C-0031.

†This publication was supported in part by Contract Number 102590-98-C-3513 from the United States Postal Service. The contents of this publication are solely the responsibility of the author and do not necessarily reflect the official views of the United States Postal Service.

CCS '00, Athens, Greece.

curity, complexity, and implementation issues. The paper concludes with the summary of previous and related work in section 8.

2. NOTATION AND DEFINITIONS

We use the following notation:

N	number of protocol parties (group members)
M_i	i-th group member; $i \in \{1, \dots, N\}$
h	height of a tree
$\langle l, v\rangle$	v-th node at level l in a tree
T_i	M_i's view of the key tree
$\widehat{T}_i$	M_i's modified tree after membership operation
p, q	prime integers
α	exponentiation base

Key trees have been suggested in the past for centralized group key distribution systems; the work of Wallner et al. [17] is the earliest such proposal. One of the key features of our work is the adaptation of key trees for use in fully distributed, contributory key agreement. Figure 1 shows an example of a key tree. The root is located at level 0 and the lowest leaves are at level h. Since we use binary trees,[1] every node is either a leaf or a parent of two nodes. The nodes are denoted $\langle l, v\rangle$, where $0 \leq v \leq 2^l - 1$ since each level l hosts at most 2^l nodes.[2] Each node $\langle l, v\rangle$ is associated with the key $K_{\langle l,v\rangle}$ and the blinded key $BK_{\langle l,v\rangle} = f(K_{\langle l,v\rangle})$ where the function $f()$ is modular exponentiation in prime order groups, i.e., $f(k) = \alpha^k \bmod p$ (analogous to the Diffie-Hellman protocol). Assuming a leaf node $\langle l, v\rangle$ hosts the member M_i, then the node $\langle l, v\rangle$ has M_i's session random key $K_{\langle l,v\rangle}$. Furthermore, the member M_i at node $\langle l, v\rangle$ knows every key along the path from $\langle l, v\rangle$ to $\langle 0, 0\rangle$, referred to as the *key-path* and denoted KEY_i^*. In figure 1, if a member M_2 owns 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 every blinded key $BK_2^* = \{BK_{\langle 0,0\rangle}, BK_{\langle 1,0\rangle}, \dots, BK_{\langle 3,7\rangle}\}$ on T_2. Every key $K_{\langle l,v\rangle}$ is computed recursively as follows:

$$
\begin{aligned}
K_{\langle l,v\rangle} &= (BK_{\langle l+1,2v+1\rangle})^{K_{\langle l+1,2v\rangle}} \bmod p \\
&= (BK_{\langle l+1,2v\rangle})^{K_{\langle l+1,2v+1\rangle}} \bmod p \\
&= \alpha^{K_{\langle l+1,2v\rangle} K_{\langle l+1,2v+1\rangle}} \bmod p \\
&= f(K_{\langle l+1,2v\rangle} K_{\langle l+1,2v+1\rangle})
\end{aligned}
$$

In other words, computing a key at $\langle l, v\rangle$ requires the knowledge of the key of one of the two child nodes and the blinded key of the other child node. $K_{\langle 0,0\rangle}$ at the root node is the group secret shared by all members. We note that this value is never used as a cryptographic key for the purposes of encryption, authentication or integrity. Instead, such keys are derived from the group secret, e.g., by setting $K_{\text{group}} = h(K_{\langle 0,0\rangle})$ where h is a cryptographically strong hash function.

For example, in figure 1, M_2 can compute $K_{\langle 2,0\rangle}, K_{\langle 1,0\rangle}$ and $K_{\langle 0,0\rangle}$ using $BK_{\langle 3,0\rangle}, BK_{\langle 2,1\rangle}, BK_{\langle 1,1\rangle}$, and $K_{\langle 3,1\rangle}$. The

[1]Note that the tree needs to be binary, since our protocol uses the two-party Diffie-Hellman key exchange to derive a node key from the contribution of the two children.

[2]Even though the key tree is not balanced, we assume a perfectly balanced tree for node numbering. Thus, a node's $\langle l, v\rangle$ left and right children have indexes $\langle l + 1, 2v\rangle$ and $\langle l + 1, 2v + 1\rangle$, respectively.

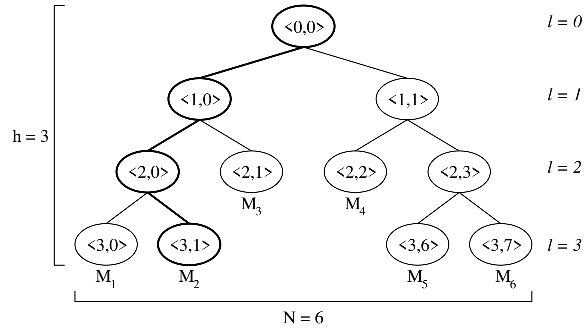

Figure 1: Notation for tree

final group key $K_{\langle 0,0\rangle}$ is:

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

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 of member M_i For example, the co-path CO_2^* of member M_2 in figure 1 is the set of nodes $\{\langle 3, 0\rangle, \langle 2, 1\rangle, \langle 1, 1\rangle\}$. Consequently, 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 $K_{\langle l,v\rangle}$.

3. GROUP COMMUNICATION AND GROUP KEY AGREEMENT

As noted in the introduction, 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.

3.1 Group Communication Semantics and Support

There are two commonly used strong group communication semantics: Extended Virtual Synchrony (EVS) [11, 1] and View Synchrony (VS) [7]. 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.

An implementation of any distributed fault-tolerant group key agreement protocol requires VS. This is because, in order to implement group key agreement 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 stress 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 [11], 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 $\mathcal{K} = \{K_0, \dots, K_m\}$).

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 [16, 13]:

- 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 (as discussed 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].
- All protocol messages are signed by the sender. (We use RSA for this purpose).

Upon each membership change, all members in the resulting group independently update the tree structure. 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 event. The following fact describes the minimal requirement for a group member to compute the group key:

FACT 1. *Any member can compute the group key from its secret share and all the blinded keys on the co-path.*[4]

Since each member knows at least its own secret share (and perhaps other keys on the key path to the root), it can compute the intermediate keys on its key path, and eventually, the group (root) key. Similar to other tree-based schemes [18, 17], 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.

In our protocol, a group member might take on a special role, which can involve to compute keys and to broadcast the blinded keys to the group, for example. Any member in the group can take on this responsibility, we call this member *sponsor*.[5] The sponsor who handles the membership change is determined differently for each membership event.

Despite the separate descriptions that follow, we describe a single protocol that handles all key adjustments in 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 unambiguously determine the sponsors and the insertion location in the tree (in case of an additive event). Note that the key refresh operation is, in fact, a special case of the leave protocol, without any members leaving the group, where the rightmost shallowest member changes it key share.

5.2 Join Protocol

Assume that the group has n users: $\{M_1, \ldots, M_n\}$. The new member M_{n+1} initiates the protocol by sending a join request message that contains its own blinded key $BK_{\langle 0,0\rangle}$. This message is separate from any JOIN messages generated by the underlying group communication system, although, in practice, the two might be combined for efficiency's sake.

When current group members receive this message, they first determine the insertion node in the tree. The insertion node is the shallowest rightmost node, where the join does not increase the height of the key tree. Otherwise, (if the key tree is well balanced), the new member joins to the root node. The sponsor is the rightmost leaf node in the subtree rooted at the insertion node. Next, the sponsor creates a new intermediate node and a new member node, and promotes the new intermediate node to be the parent of both the insertion node and the new member node. After updating the tree, the sponsor computes the new group key,

[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 [16, 3].

[4]We introduce the notion of co-path in section 2.

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

since it knows all the necessary blinded keys. After computing the group key, the sponsor broadcasts the new tree which contains all blinded keys.[6] All other members update their trees accordingly and compute the new group key (see fact 1).

Figure 2 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. 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 blinded keys, every member can compute the group key.

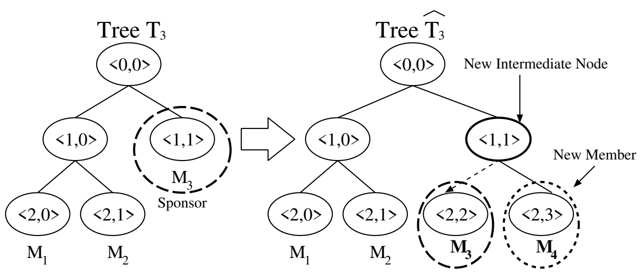

Figure 2: Tree updating in join operation

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 right-most leaf node of the subtree rooted at the leaving member's sibling node. In the leave protocol every member updates its key tree by deleting the leaf node corresponding to M_d. The former sibling of M_d is promoted to replace M_d's parent node. The sponsor picks a new secret share, computes all keys on its key path up to the root, and broadcasts the new set of blinded keys to the group. This information allows all members to recompute the new group key.

Assuming the setting of figure 3, if member M_3 leaves the group, every member deletes nodes $\langle 1,1\rangle$ 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. Note that M_3 cannot compute the group key, though it knows all the blinded keys, because its share is no longer part of the group key.

5.4 Partition Protocol

Assume that a network fault occurs in a group with n members $\{M_1, \ldots, M_n\}$. From the viewpoint of each remaining member, this appears as a concurrent leave of multiple members. Our partition protocol is a multi-round protocol which runs until every member computes the new group key.

[6] Alternatively, we may broadcast only blinded keys which have been changed after the join to reduce the bandwidth. However, we need to send, at least, the whole tree to the new member in this case.

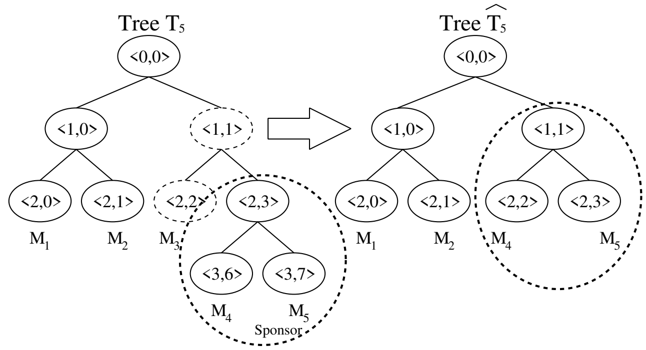

Figure 3: Tree updating in leave operation

In the first round, every remaining member updates its tree by deleting all partitioned members and their respective parent nodes. The procedure for deletion is as follows:

> All leaving nodes are sorted in the order of depth. Starting at the deepest level, each pair of leaving siblings is collapsed into its parent which is then marked as leaving. This node is re-inserted into the leaving nodes list. This is repeated until all leaving nodes are processed.
> The resulting tree has a number of leaving (leaf) nodes but every such node has a remaining sibling node. Now, for each leaving node we identify a sponsor using the same criteria as described in section 5.3.

Each sponsor then computes the keys and blinded keys on its key-path as far up the tree as possible. Then, each sponsor broadcasts the set of new blinded keys. Upon receiving a broadcast, each member checks whether the message contains a new blinded key. This procedure iterates until all members obtain the group key. (A member can compute the group key if it has all the blinded keys on its co-path.)

$$\boxed{\begin{array}{l} \text{for all } M_{s_i} \xrightarrow{\widehat{T}_{s_i}(BK^*_{s_i})} \{M_1, \ldots, M_n\} \\ \text{round 1 to } h'+1 : \text{update tree } T_{s_i} \text{ to get } \widehat{T}_{s_i} \end{array}}$$

Figure 4: Partition Protocol

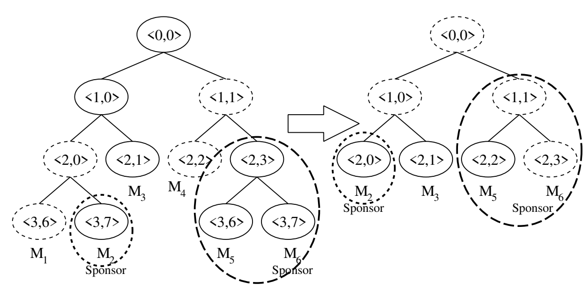

Figure 5: Tree updating in partition operation

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 5 shows an example where all remaining members delete all nodes of leaving members and compute the keys and 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 $K_{\langle 2,3\rangle}$.

If a member M_i computes the group key in round h', then all other members can compute the group key, at the latest, in round $h'+1$, since M_i's broadcast message contains every blinded key in 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 heal, subgroups may re-merge. We describe the merge protocol for two merging groups.

In the first round of the merge protocol, each sponsor (the rightmost member of each group) broadcasts its tree information with all blinded keys to the other group. Upon receiving this message, all members can uniquely and independently determine the merge position of the two trees. If the two trees have the same height, we join one tree to the root node (insertion node) of the other tree.[7] Otherwise, the trees are of different height and we join the shallower tree to the deeper tree. The insertion node can be: 1) the rightmost shallowest node (not necessarily a leaf node), where the join does not increase the height of the tree, and 2) the root node, if join to any other node increases the height of the key tree.[8]

The rightmost member of the subtree rooted at the joining location becomes the sponsor of the key update operation. The sponsor computes every key on the key-path and the corresponding blinded key. It, then, broadcasts the tree with the blinded keys to the other members. All members now have the complete set of blinded keys, which allows them to compute all keys on their key path.

Figure 7 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.

5.6 Tree Management

Modular exponentiation is an expensive operation in TGDH. The number of exponentiations for membership events varies, 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 balanced, and a member joins to a leaf node, then the number of exponentiations is $\lceil \log_2 n \rceil$ 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 $\lceil \log_2 n \rceil$. These examples indicate that a well-balanced key tree reduces the expected cost of leaves. Our heuristic to keep the tree balanced is to choose the insertion node of a join or merge operation as the rightmost shallowest node, which does not increase the height (see also sections 5.2 and 5.5).

[7] To impose an ordering on the two trees, we compare the identifiers of the sponsors.

[8] The rationale for this policy is explained in section 5.6.

M_{s_1} in $\{M_1, \ldots, M_n\}$ $\xrightarrow{T_{s_1}(BK^*_{s_1})}$ M_{s_2} in $\{M_{n+1}, \ldots, M_{n+k}\}$

$\xleftarrow{T_{s_2}(BK^*_{s_2})}$

Round 1: request for merge by both groups

$\{M_1, \ldots, M_{n+k}\}$ $\xleftarrow{\widehat{T}_{s'}(BK^*_{s'})}$ M'_s

Round 2: update tree $T_{s'}$ to get $\widehat{T}_{s'}$ and broadcast it

Figure 6: Merge Protocol

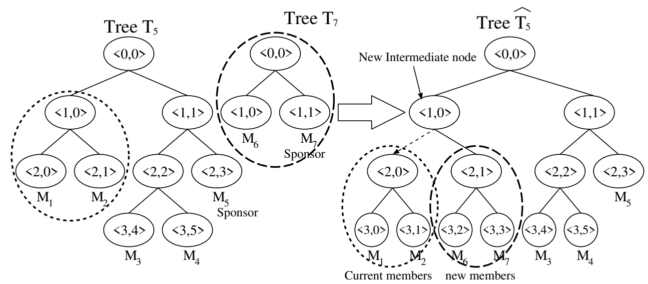

Figure 7: Tree updating in merge operation

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 operations: 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, because in either case, the key tree changes and the remaining group members lack some number (sometimes none) of blinded keys which prevents them from computing the new root key. 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)
    compute missing blinded keys
    broadcast new blinded keys
  endif
  receive msg (msg type = broadcast)
  update current tree
endwhile
```

Figure 8: 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. Figure 8 shows the pseudocode. The incentive for this is threefold. First, unification allows us to simplify the implementation and minimize its size. Second, the overall security and correctness are easier to demonstrate with a single protocol. Third, 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, cascaded events and their impact on group protocols are often considered in group communication literature, but, alas, not often enough in the security literature.) A cascaded event occurs, in its simplest form, when one membership change occurs while another is being handled. Event here means any of: join, leave, partition, merge or a combination thereof. For example, a partition can occur while a prior partition is being dealt with, resulting in a cascade of size two. In principle, cascaded events of arbitrary size can occur if the underlying network is highly volatile.

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 9. The changes from figure 8 are minimal.

```
receive msg (msg type = membership event)
construct new tree
while there are missing blinded keys
  if (I can compute any missing keys)
    compute missing blinded keys
    broadcast new blinded keys
  endif
  receive msg
  if (msg type = broadcast)
    update current tree
  else (msg type = membership event)
    construct new tree
endwhile
```

Figure 9: Self-stabilizing protocol pseudocode

Instead of providing a formal proof of self-stabilization (which we omit due to page limitations) we demonstrate it with an example. Figure 10 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 $\{M_1, \ldots, M_{10}\}$. This partition normally requires two rounds to complete the key agreement. As described in section 5.4, every 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 10. 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 by M_6. When the broadcast is received, all members 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 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

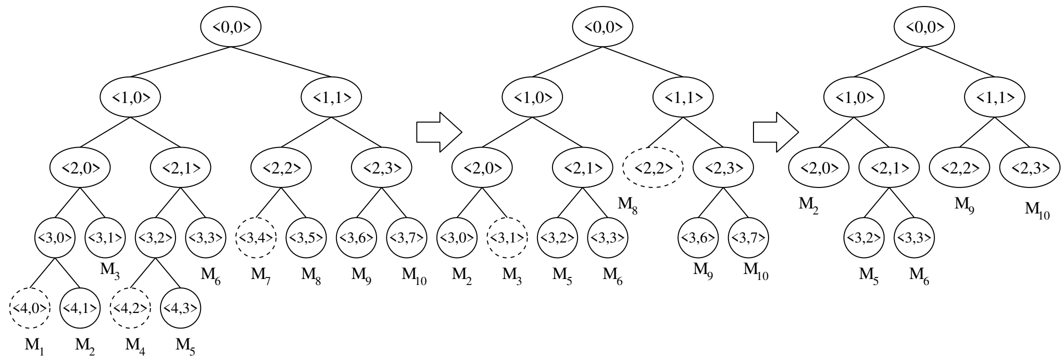

Figure 10: An Example of Cascaded Partition

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

Recall that we defined the desired security properties in section 4. Our goal is to show that TGDH offers group key secrecy as well as weak forward and backward secrecy.

Group key secrecy means that even an attacker who knows all blinded keys cannot derive the group key. This property has been proven in the random-oracle model [5]. The proof itself can be found in a companion technical report [8]. (Due to size constraints we are unable to include it in this paper.) We also refer the reader to the proof of Becker and Wille [4]. Their group key is very similar to our TGDH key and the accompanying proof is applicable to TGDH. In brief, they show that group key secrecy is reducible to the Decision Diffie-Hellman (DDH) problem [10].

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 who knows the current group key cannot derive any previous group key.

The group key secrecy property implies 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 obtain any keys of the previous key tree. First, M picks its secret share r, blinds it and broadcasts α^r as part of its join request. Once M receives all blinded keys on its co-path, it can compute all secret keys on its key path. Clearly, all these keys will contain M's contribution (r); hence, they are independent of previous secret keys on that path. Therefore, 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 changes its secret share, M's leaf node is deleted and its parent node is replaced with its sibling node. This operation causes all of M's contribution to removed from each key on M's former 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 Implementation

TREE_API is a group key agreement API implementing the cryptographic primitives of TGDH. The underlying communication system is assumed to deal with group communication and network events such as merges, partitions, failures and other abnormalities. TREE_API is small and it contains only the following three function calls:

- `tree_new_user` generates a group context for a new group member (including its secret share).
- `tree_merge_req` is called by each sponsor when a merge occurs. The output (new key tree) is then broadcast to the merging group. This function performs no cryptographic operations.
- `tree_cascade` is the core function of TREE_API. Every group member calls this function following a membership event. The function is called repeatedly until the group key is computed.

As mentioned in section 6, `tree_cascade` provides robustness against cascaded network events. Since TREE_API does not provide its own communication facility, the robustness of the API was tested by simulating random events on a single machine running all group members.

We use OpenSSL 0.9.4 [12] as the underlying cryptographic library. Note that we set $f(x) = (\alpha^x \bmod p) \bmod q$ where $x \in \mathbb{Z}_q$ so as to enhance the efficiency of our protocol.

7.3 Complexity Analysis

This section analyzes the communication and computation costs for join, leave, merge and partition protocols of TGDH. Our analysis considers the following costs: number of rounds, number of messages, number of exponentiations, and height of the key tree. Each cost can be further classified into serial and total cost. The former assumes parallelization within each protocol round and represents the greatest cost incurred by any participant in a given round. The total cost is simply the sum of all participants' costs in a given round.

We compare our protocol to the authenticated contributory group key agreement scheme GDH.2 described in [3]. To the best of our knowledge, GDH.2 of the Cliques protocol suite is the only other protocol that provides *contributory* authenticated group key agreement, supports dynamic membership events, and handles both partitions and merges.

Operations	Join		Leave	
Protocol	GDH.2	TGDH	GDH.2	TGDH
Rounds	2	2	1	1
Broadcasts	1	2	1	1
Total messages	2	2	1	1
Max bandwidth	N	$2N$	$N-1$	$2N-2$

Table 1: Communication Cost

However, GDH.2 has very different communication semantics. (In particular, merge is relatively expensive in GDH.2, whereas partition is relatively expensive in TGDH.)

The overhead of TGDH depends on the structure of the key tree. The number of exponentiations depends on the height and density of the key tree, the depth of the joining point (or leaving node), and the number of leaving members in case of partition. For this reason, we analyze the protocol overhead by fixing the maximum number of members.

Figure 11(a) compares the number of exponentiations for a join event in TGDH (with a fully-balanced tree) and GDH.2. As expected, TGDH costs $O(\log n)$ exponentiations. The graph also shows that the number of exponentiations in TGDH depends on the position of the joining points. Figure 11(b) shows the number of serial exponentiations when joining to a random tree of N members. The average nears a constant value, since the joining point in a random tree is near the root. Based on these graphs, the computation overhead of TGDH is lower than that GDH.2. The computation cost of leave is also smaller than GDH.2. A comparison of the communication cost is summarized in table 1. The communication cost for join in TGDH is slightly more expensive than that of GDH.2.

The cost of partition is particularly interesting, since TGDH is a multi-round protocol whereas a partition in GDH.2 is handled in a single round. We measure this cost after generating a uniform random partition. For this experiment, we consider partitions that split the group into two sub groups, since this case reflects the highest cost. To split the group, we first pick a random number x uniformly in the interval $\{1, \ldots, N-1\}$. We then pick x members randomly to form each sub group. Next, we compute the cost of partition for both groups which gives us two sample costs. The average and maximum numbers are computed from these samples.

As figure 11(c) illustrates, the number of serial exponentiations is small, compared to $N-d$ exponentiations of GDH.2, where d is the number of leaving members. The communication cost (total number of messages in figure 11(d) and the number of rounds in figure 11(e)) for partition is noticeably larger than that of GDH.2. Note that TGDH is less efficient for subtractive events, but more efficient for additive events.[9] Figure 11(f) shows the average and maximum height of the key tree after merge operations. The results indicates the logarithmic height of the key tree.

8. RELATED WORK

Group key management protocols come in two different flavors: contributory key agreement protocols for small groups and centralized, server-based key distribution protocols for large groups. Since the focus of this work is on group key agreement protocols, we only consider the latter below.

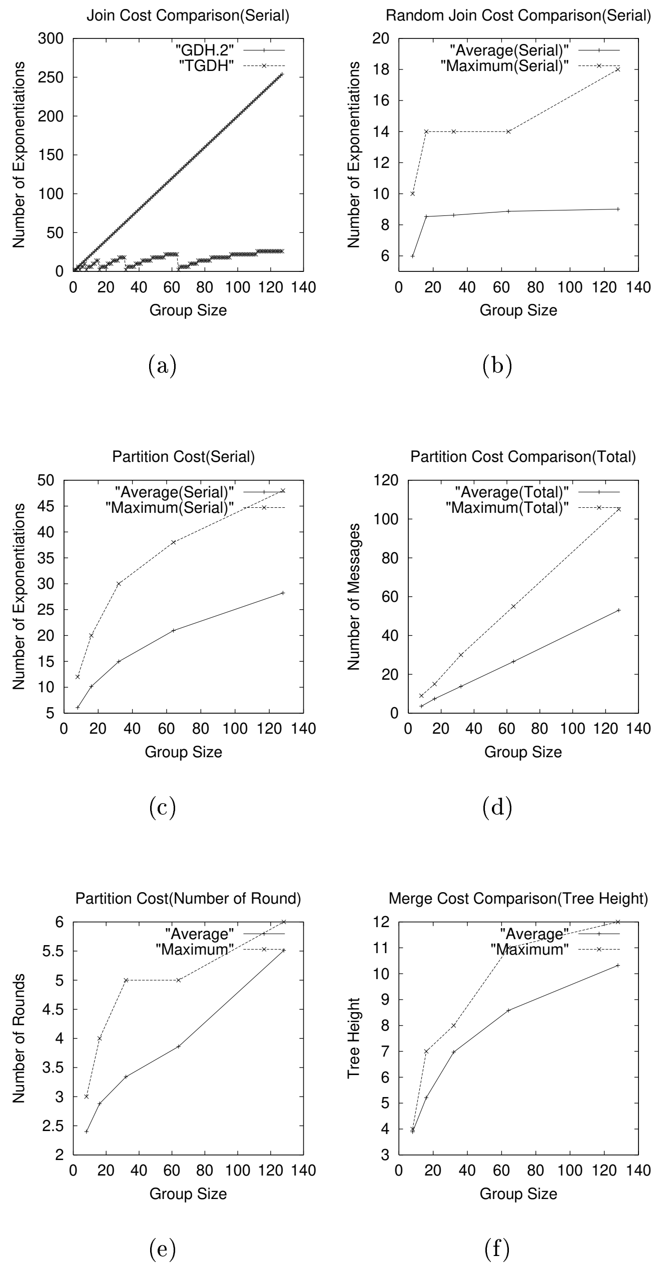

Figure 11: Cost Comparison

In one of the early results, Steer et al. propose a group key agreement protocol – referred to as STR [15] – based on the extension of the two-party Diffie-Hellman (DH) key exchange. This protocol is of particular interest since its group key structure is similar to that in TGDH.

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

[9]In GDH.2, merge requires $k+2$ rounds and $O(kn)$ exponentiations, where k is the number of new members.

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

A more recent result is due to Burmester and Desmedt[6]. They construct an efficient protocol which takes 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 any membership change by performing this protocol. However, according to [16], most (at least half) of the members need to change their session random on every membership event. The group key in this protocol is different from STR and TGDH:

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

Becker and Wille analyze the minimal communication complexity of contributory group key agreement in general [4] and propose two protocols: *octopus* and *hypercube*. Their group key has the same structure as the key in TGDH. For example, for eight users their group key is:

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

The Becker/Wille protocols handles join and merge operations efficiently, but the member leave operation is inefficient. Also, the *hypercube* protocol requires the group to be of size 2^n (for some integer n); otherwise, the efficiency slips.

Steiner et al. address dynamic membership issues [3, 16] in group key agreement and propose a family of Group Diffie Hellman (GDH) protocols based on straight-forward extensions of the two-party Diffie-Hellman. GDH provides contributory authenticated key agreement, key independence, key integrity, resistance to known key attacks, and perfect forward secrecy. Their protocol suite is fairly efficient in leave and partition operation, but the merge protocol requires as many rounds as the number of new members to complete key agreement.

Perrig extends the work of one-way function trees (OFT, originally introduced by McGrew and Sherman [9]) to design a tree-based key agreement scheme for peer groups [13]. However, this work lacked the facilities for handling group partitions and merges.

Rodeh et al. [14] propose a distributed group key distribution protocol. It tolerates network partitions and other network events. In this protocol, a specific group member (leader) chooses the group key and distributes it to all other members, hence the protocol does not offer contributory key agreement. Furthermore, it requires the leader to establish $N-1$ secure two-party channels between itself and other group members in order to securely distribute the new key. Maintaining such 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 separate two-party key agreement.

9. ACKNOWLEDGMENTS

We are very grateful to Dawn Song for her help with the security proof and to Michael Steiner for useful discussions. We would also like to thank the anonymous referees for their helpful comments. We are indebted to Markus Kuhn for his helpful editorial advice.

10. 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, Y. Kim, C. Nita-Rotaru, T. Schlossnagle, J. Schultz, J. Stanton, and G. Tsudik. Secure group communication in asynchronous networks with failures: Integration and experiments. In *ICDCS 2000*, Apr. 2000.

[3] G. Ateniese, M. Steiner, and G. Tsudik. Authenticated Group Key Agreement and Friends. In *ACM CCS '98*, pages 17–26. ACM, November 1998.

[4] C. Becker and U. Wille. Communication complexity of group key distribution. In *ACM CCS '98*, Nov. 1998.

[5] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM CCS '93*, 1993.

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

[7] A. Fekete, N. Lynch, and A. Shvartsman. Specifying and using a partionable group communication service. In *ACM PODC '97*, pages 53–62, Santa Barbara, CA, August 1997.

[8] Y. Kim, A. Perrig, and G. Tsudik. Simple and fault-tolerant key agreement for dynamic collaborative groups. Technical Report 2, USC Technical Report 00-737, August 2000.

[9] D. McGrew and A. Sherman. Key establishment in large dynamic groups using one-way function trees. Manuscript, May 1998.

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

[11] L. Moser, Y. Amir, P. Melliar-Smith, and D. Agarwal. Extended virtual synchrony. In *ICDCS '94*, pages 56–65, June 1994.

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

[13] A. Perrig. Efficient collaborative key management protocols for secure autonomous group communication. In *CrypTEC '99*, pages 192–202, 1999.

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

[15] D. Steer, L. Strawczynski, W. Diffie, and M. Wiener. A secure audio teleconference system. In *CRYPTO '88*, pages 520–528. LNCS 403, 1988.

[16] M. Steiner, G. Tsudik, and M. Waidner. Cliques: A new approach to group key agreement. *IEEE Transactions on Parallel and Distributed Systems*, To appear in 2000.

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

[18] C. Wong, M. Gouda, and S. Lam. Secure group communications using key graphs. *IEEE/ACM Trans. on Networking*, 8(1):16–30, 2000.