Secure Group Communications Using Key Graphs*

Chung Kei Wong Mohamed Gouda Simon S. Lam
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712-1188
{*ckwong,gouda,lam*}*@cs.utexas.edu*

Abstract

Many emerging applications (e.g., teleconference, real-time information services, pay per view, distributed interactive simulation, and collaborative work) are based upon a group communications model, i.e., they require packet delivery from one or more authorized senders to a very large number of authorized receivers. As a result, securing group communications (i.e., providing confidentiality, integrity, and authenticity of messages delivered between group members) will become a critical networking issue.

In this paper, we present a novel solution to the scalability problem of group/multicast key management. We formalize the notion of a secure group as a triple (U, K, R) where U denotes a set of users, K a set of keys held by the users, and R a user-key relation. We then introduce key graphs to specify secure groups. For a special class of key graphs, we present three strategies for securely distributing rekey messages after a join/leave, and specify protocols for joining and leaving a secure group. The rekeying strategies and join/leave protocols are implemented in a prototype group key server we have built. We present measurement results from experiments and discuss performance comparisons. We show that our group key management service, using any of the three rekeying strategies, is scalable to large groups with frequent joins and leaves. In particular, the average measured processing time per join/leave increases linearly with the logarithm of group size.

1 Introduction

Most network applications are based upon the client-server paradigm and make use of unicast (or point-to-point) packet delivery. Many emerging applications (e.g., teleconference, real-time information services, pay per view, distributed interactive simulation, and collaborative work), on the other hand, are based upon a *group communications* model. That is, they require packet delivery from one or more authorized sender(s) to a large number of authorized receivers. In the Internet, multicast has been used successfully to provide an efficient, best-effort delivery service to large groups [6]. We envision that deployment of network applications requiring group communications will accelerate in coming years.

While the technical issues of securing unicast communications for client-server computing are fairly well understood, the technical issues of securing group communications are not. Yet group communications have a much greater exposure to security breaches than unicast communications. In particular, copies of a group communication packet traverse many more links than those of a unicast packet, thereby creating more opportunity for traffic interception. We believe that securing group communications (i.e., providing confidentiality, integrity, and authenticity of messages delivered between group members) will become a critical issue of networking in the near future.

Conceptually, since every point-to-multipoint communication can be represented as a set of point-to-point communications, the current technology base for securing unicast communications can be extended in a straightforward manner to secure group communications [9, 10]. However, such an extension is not scalable to large groups.

For a more concrete illustration of this point, we outline a typical procedure for securing unicast communications between a client and a server. Initially, the client and server mutually authenticate each other using an authentication protocol or service; subsequently, a symmetric key is created and shared by them to be used for pairwise confidential communications [4, 17, 19, 22]. This procedure can be extended to a group as follows: Let there be a trusted *group server* which is given membership information to exercise group access control. When a client wants to join the group, the client and group server mutually authenticate using an authentication protocol. Having been authenticated and accepted into the group, each member shares with the group server a key,[1] to be called the member's *individual key*. For group communications, the group server distributes to each member a *group key* to be shared by all members of the group.[2]

For a group of n members, distributing the group key securely to all members requires n messages encrypted with individual keys (a computation cost proportional to group size n). Each such message may be sent separately via unicast. Alternatively, the n messages may be sent as a combined message to all group members via multicast. Either way, there is a communication cost proportional to group

*Research sponsored in part by Texas Advanced Research Program grant no. 003658-063 and by NSA INFOSEC University Research Program grant no. MDA 904-94-C-6106. Experiments were performed on equipment procured with National Science Foundation grant no. CDA-9624082.

To appear in *Proceedings of ACM SIGCOMM '98*, September 2-4, 1998, Vancouver, Canada.

[1] In this paper, *key* means a key from a symmetric cryptosystem, such as DES, unless explicitly stated otherwise.

[2] It is easy to see that sharing a group key enables confidential group communications. In addition to confidentiality, authenticity and integrity can be provided in group communications using standard techniques such as digital signature and message digest. We will not elaborate upon these techniques since the focus of this paper is key management.

size n (measured in terms of the number of messages or the size of the combined message).

Observe that for a point-to-point session, the costs of session establishment and key distribution are incurred just once, at the beginning of the session. A group session, on the other hand, may persist for a relatively long time with members joining and leaving the session. Consequently, the group key should be changed frequently. To achieve a high level of security, the group key should be changed after every *join* and *leave* so that a former group member has no access to current communications and a new member has no access to previous communications.

Consider a group server that creates a new group key after every join and leave. After a join, the new group key can be sent via unicast to the new member (encrypted with its individual key) and via multicast to existing group members (encrypted with the previous group key). Thus, changing the group key securely after a join is not too much work. After a leave, however, the previous group key can no longer be used and the new group key must be encrypted for each remaining group member using its individual key. Thus we see that changing the group key securely after a leave incurs computation and communication costs proportional to n, the same as initial group key distribution. That is, large groups whose members join and leave frequently pose a scalability problem.

The topic of secure group communications has been investigated [1, 2, 8, 15]. Also the problem of how to distribute a secret to a group of users has been addressed in the cryptography literature [3, 5, 7, 18]. However, with the exception of [15], no one has addressed the need for frequent key changes and the associated scalability problem for a very large group. The approach proposed in Iolus [15] to improve scalability is to decompose a large group of clients into many subgroups and employ a hierarchy of group security agents.

1.1 Our approach

We present in this paper a different hierarchical approach to improve scalability. Instead of a hierarchy of group security agents, we employ a hierarchy of keys. A detailed comparison of our approach and the Iolus approach [15] is given in Section 6.

We begin by formalizing the notion of a secure group as a triple (U, K, R) where U denotes a set of users, K a set of keys, and $R \subset U \times K$ a *user-key* relation which specifies keys held by each user in U. In particular, each user is given a subset of keys which includes the user's individual key and a group key. We next illustrate how scalability of group key management can be improved by organizing the keys in K into a hierarchy and giving users additional keys.

Let there be a trusted group server responsible for group access control and key management. In particular, the server securely distributes keys to group members and maintains the user-key relation.[3] To illustrate our approach, consider the following simple example of a secure group with nine members partitioned into three subgroups, $\{u_1, u_2, u_3\}$, $\{u_4, u_5, u_6\}$, and $\{u_7, u_8, u_9\}$. Each member is given three keys, its individual key, a key for the entire group, and a key for its subgroup. Suppose that u_1 leaves the group, the remaining eight members form a new secure group and require a new group key; also, u_2 and u_3 form a new subgroup and require a new subgroup key. To send the new subgroup key securely to u_2 (u_3), the server encrypts it with the individual key of u_2 (u_3). Subsequently, the server can send the new group key securely to members of each subgroup by encrypting it with the subgroup key. Thus by giving each user three keys instead of two, the server performs five encryptions instead of eight. As a more general example, suppose the number n of users is a power of d, and the keys in K are organized as the nodes of a full and balanced d-ary tree. When a user leaves the secure group, to distribute new keys, the server needs to perform approximately $d \log_d(n)$ encryptions (rather than $n - 1$ encryptions).[4] For a large group, say 100,000, the savings can be very substantial.

[3] In practice, such a server may be distributed or replicated to enhance reliability and performance.

1.2 Contributions of this paper

With a hierarchy of keys, there are many different ways to construct rekey messages and securely distribute them to users. We investigate three rekeying strategies, *user-oriented*, *key-oriented* and *group-oriented*. We design and specify join/leave protocols based upon these rekeying strategies. For key-oriented and user-oriented rekeying, which use multiple rekey messages per join/leave, we present a technique for signing multiple messages with a single digital signature operation. Compared to using one digital signature per rekey message, the technique provides a tenfold reduction in the average server processing time of a join/leave.

The rekeying strategies and protocols are implemented in a prototype group key server we have built. We performed experiments on two lightly loaded SGI Origin 200 machines, with the server running on one and up to 8,192 clients on the other. From measurement results, we show that our group key management service, using any of the rekeying strategies with a key tree, is scalable; in particular, the average server processing time per join/leave increases linearly with the logarithm of group size. We found that the optimal key tree degree is around four. Group-oriented rekeying provides the best performance of the three strategies on the server side, but is worst of the three on the client side. User-oriented rekeying has the best performance on the client side, but worst on the server side.

The balance of this paper is organized as follows. In Section 2, we introduce key graphs as a method for specifying secure groups. In Section 3, we present protocols for users to join and leave a secure group as well as the three rekeying strategies. In Section 4, we present a technique for signing multiple rekey messages using a single digital signature operation. Experiments and performance results are presented in Section 5. A comparison of our approach and the Iolus approach is given in Section 6. Our conclusions are in Section 7.

2 Secure Groups

A *secure group* is a triple (U, K, R) where

- U is a finite and nonempty set of users,
- K is a finite and nonempty set of keys, and
- R is a binary relation between U and K, that is, $R \subset U \times K$, called the *user-key* relation of the secure group. User u has key k if and only if (u, k) is in R.

[4] A similar observation was independently made in [20] at about the same time as when this paper was first published as a technical report [21].

Each secure group has a trusted *group server* responsible for generating and securely distributing keys in K to users in the group.[5] Specifically, the group server knows the user set U and the key set K, and maintains the user-key relation R. Every user in U has a key in K, called its *individual key*, which is shared only with the group server, and is used for pairwise confidential communication with the group server. There is a *group key* in K, shared by the group server and all users in U. The group key can be used by each user to send messages to the entire group confidentially.

2.1 Key graphs

A key graph is a directed acyclic graph G with two types of nodes, *u-nodes* representing users and *k-nodes* representing keys. Each u-node has one or more outgoing edges but no incoming edge. Each k-node has one or more incoming edges. If a k-node has incoming edges only and no outgoing edge, then this k-node is called a *root*. (A key graph can have multiple roots.)

Given a key graph G, it specifies a secure group (U, K, R) as follows:

i. There is a one-to-one correspondence between U and the set of u-nodes in G.

ii. There is a one-to-one correspondence between K and the set of k-nodes in G.

iii. (u, k) is in R if and only if G has a directed path from the u-node that corresponds to u to the k-node that corresponds to k.

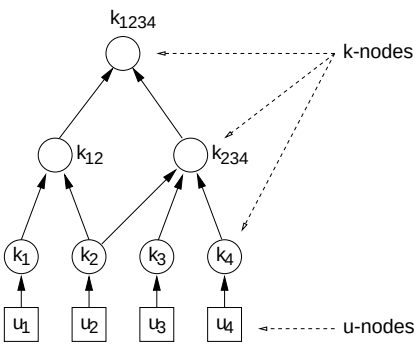


Figure 1: A key graph.

As an example, the key graph in Figure 1 specifies the following secure group:

$$\begin{aligned} U &= \{u_1, u_2, u_3, u_4\} \\ K &= \{k_1, k_2, k_3, k_4, k_{12}, k_{234}, k_{1234}\} \\ R &= \{\ (u_1, k_1),\ (u_1, k_{12}),\ (u_1, k_{1234}), \\ &\quad (u_2, k_2),\ (u_2, k_{12}),\ (u_2, k_{234}),\ (u_2, k_{1234}), \\ &\quad (u_3, k_3),\ (u_3, k_{234}),\ (u_3, k_{1234}), \\ &\quad (u_4, k_4),\ (u_4, k_{234}),\ (u_4, k_{1234})\ \} \end{aligned}$$ □

Associated with each secure group (U, K, R) are two functions, $keyset()$ and $userset()$, defined as follows:

$$keyset(u) = \{\ k \mid (u, k) \in R\ \}$$
$$userset(k) = \{\ u \mid (u, k) \in R\ \}$$

Intuitively, $keyset(u)$ is the set of keys that are held by user u in U, and $userset(k)$ is the set of users that hold key k in K. For examples, referring to the key graph in Figure 1, we have $keyset(u_4) = \{k_4, k_{234}, k_{1234}\}$ and $userset(k_{234}) = \{u_2, u_3, u_4\}$.

We generalize the definition of function $keyset()$ to any subset U' of U, and function $userset()$ to any subset K' of K, in a straighforward manner, i.e., $keyset(U')$ is the set of keys each of which is held by at least one user in U', and $userset(K')$ is the set of users each of which holds at least one key in K'.

When a user u leaves a secure group (U, K, R), every key that has been held by u and shared by other users in U should be changed. Let k be such a key. To replace k, the server randomly generates a new key k_{new} and sends it to every user in $userset(k)$ except u. To do so securely, the server needs to find a subset K' of keys such that $userset(K') = userset(k) - \{u\}$, and use keys in K' to encrypt k_{new}. To minimize the work of rekeying, the server would like to find a minimal size set K'. This suggests the following *key-covering problem*: Given a secure group (U, K, R), and a subset S of U, find a minimum size subset K' of K such that $userset(K') = S$. Unfortunately, the key-covering problem in general is NP-hard [21].

2.2 Special classes of key graphs

We next consider key graphs with special structures for which the key covering problem can be easily solved.

Star: This is the special class of a secure group (U, K, R) where each user in U has two keys: its individual key and a *group key* that is shared by every user in U.[6]

Tree: This is the special class of a secure group (U, K, R) whose key graph G is a single-root tree. A tree key graph (or *key tree*) is specified by two parameters.

- The *height* h of the tree is the length (in number of edges) of the longest directed path in the tree.
- The *degree* d of the tree is the maximum number of incoming edges of a node in the tree.

Note that since the leaf node of each path is a u-node, each user in U has at most h keys. Also the key at the root of the tree is shared by every user in U, and serves as the *group key*. Lastly, it is easy to see that *star* is a special case of *tree*.

Complete: This is the special class of a secure group (U, K, R), where for every nonempty subset S of U, there is a key k in K such that $userset(k) = S$. Let n be the number of users in U. There are $2^n - 1$ keys in K, one for each of the $2^n - 1$ nonempty subsets of U. Moreover, each user u in U has 2^{n-1} keys, one for each of the 2^{n-1} subsets of U that contains u. Since U is a subset of U, there is a key shared by every user in U which serves as the *group key*.

The total number of keys held by the server and the number of keys held by a user are presented in Table 1 where n is the size of U. In particular, in the case of a complete key graph, each user needs to hold 2^{n-1} keys which is practical only for small n. Note that the number of keys in a key tree is $\frac{d^h-1}{d-1} \approx \frac{d}{d-1}n$ when the tree is full and balanced (i.e. $n = d^{h-1}$).

[5] Note that individual keys may have been generated and securely distributed by an authentication service and do not have to be generated by the group server.

[6] This is the base case where no additional keys are used to improve scalability of group key management.

Class of key graph	Star	Tree	Complete
Total number of keys	$n+1$	$\frac{d}{d-1}n$	2^n-1
Number of keys per user	2	h	2^{n-1}

Table 1: Number of keys held by the server and by each user.

3 Rekeying Strategies and Protocols

A user u who wants to join (leave) a secure group sends a join (leave) request to the group server, denoted by s. For a join request from user u, we assume that group access control is performed by server s using an access control list provided by the initiator of the secure group.[7] A join request initiates an authentication exchange between u and s, possibly with the help of an authentication server. If user u is not authorized to join the group, server s sends a join-denied reply to u. If the join request is granted, we assume that the session key distributed as a result of the authentication exchange [17, 22] will be used as the individual key k_u of u. To simplify protocol specifications below, we use the following notation

$$s \Leftrightarrow u \ : \ \text{authenticate } u \text{ and distribute } k_u$$

to represent the authentication exchange between server s and user u, and secure distribution of key k_u to be shared by u and s.

After each join or leave, a new secure group is formed. Server s has to update the group's key graph by replacing the keys of some existing k-nodes, deleting some k-nodes (in the case of a leave), and adding some k-nodes (in the case of a join). It then securely sends *rekey messages* containing new group/subgroup keys to users of the new secure group. (A reliable message delivery system, for both unicast and multicast, is assumed.) In protocol specifications below, we also use the following notation

$$x \rightarrow y \ : \ z$$

to denote

- if y is a single user, the sending of message z from x to y;
- if y is a set of users, the sending of message z from x to every user in y (via multicast or unicast).

In the following subsections, we first present protocols for joining and leaving a secure group specified by a star key graph. These protocols correspond to conventional rekeying procedures informally described in the Introduction [9, 10]. We then consider secure groups specified by tree key graphs. With a hierarchy of group and subgroup keys, rekeying after a join/leave can be carried out in a variety of ways. We present three rekeying strategies, *user-oriented*, *key-oriented*, and *group-oriented*, as well as protocols for joining and leaving a secure group.

3.1 Joining a star key graph

After granting a join request from user u, server s updates the key graph by creating a new u-node for u and a new k-node for k_u, and attaching them to the root node. Server s also generates a new group key $k_{U'}$ for the root node, encrypts it with the individual key k_u of user u, and sends the encrypted new group key to u. To notify other users of the new group key, server s encrypts the new group key $k_{U'}$ with the old group key k_U, and then multicasts the encrypted new group key to every user in the group. (See Figure 2.)

(1) $u \rightarrow s$: join request
(2) $s \Leftrightarrow u$: authenticate u and distribute k_u
(3) s : randomly generate a new group key $k_{U'}$
(4) $s \rightarrow u$: $\{k_{U'}\}_{k_u}$
(5) $s \rightarrow U$: $\{k_{U'}\}_{k_U}$

Figure 2: Join protocol for a star key graph.

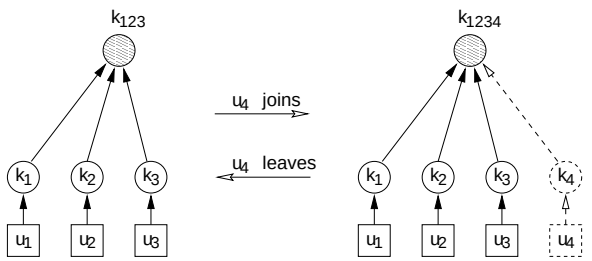


Figure 3: Star key graphs before and after a join (leave).

For example, as shown in Figure 3, suppose user u_4 wants to join the left secure group in the figure, and it is allowed to join. After server s changes the group key from k_{123} to a new key k_{1234}, server s needs to send out the following two rekey messages.

$$s \rightarrow \{u_1, u_2, u_3\} \quad : \quad \{k_{1234}\}_{k_{123}}$$
$$s \rightarrow u_4 \quad : \quad \{k_{1234}\}_{k_4}$$

For clarity of presentation, we have assumed that rekey messages contain new keys only and secure distribution means that the new keys are encrypted just for confidentiality. In our prototype implementation, rekey messages have additional fields, such as, subgroup labels for new keys, server digital signature, message integrity check, timestamp, etc. (See [21] for rekey message format.)

3.2 Leaving a star key graph

(1) $u \rightarrow s$: $\{$ leave-request $\}_{k_u}$
(2) $s \rightarrow u$: $\{$ leave-granted $\}_{k_u}$
(3) s : randomly generate a new group key $k_{U'}$
(4) for each user v in U except user u do
$s \rightarrow v$: $\{k_{U'}\}_{k_v}$

Figure 4: Leave protocol for a star key graph.

After granting a leave request from user u, server s updates the key graph by deleting the u-node for user u and the k-node for its individual key k_u from the key graph. Server s generates a new group key $k_{U'}$ for the new secure group without u, encrypts it with the individual key of each remaining user, and unicasts the encrypted new group key to the user. (See Figure 4.)

[7] The authorization function may be offloaded to an authorization server. In this case, the authorization server provides an authorized user with a ticket to join the secure group [16, 23]. The user submits the ticket together with its join request to server s.

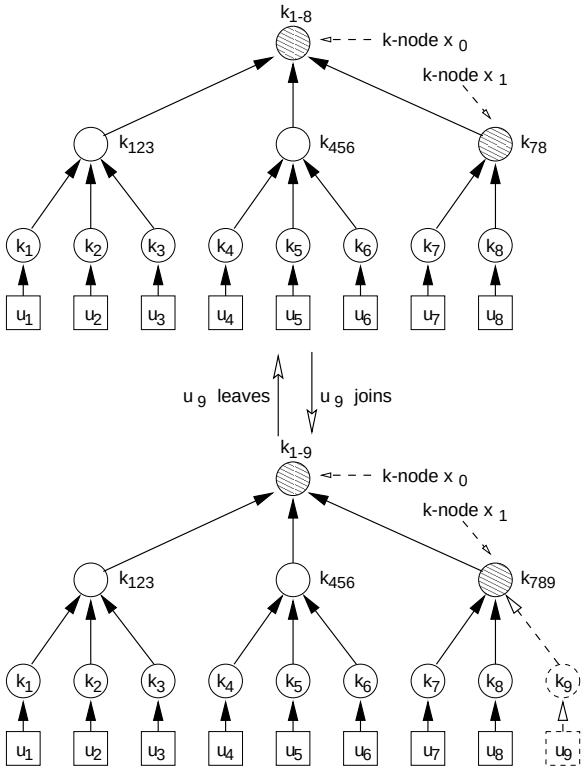


Figure 5: Key trees before and after a join (leave).

3.3 Joining a tree key graph

After granting a join request from u, server s creates a new u-node for user u and a new k-node for its individual key k_u. Server s finds an existing k-node (called the *joining point* for this join request) in the key tree and attaches k-node k_u to the joining point as its child.

To prevent the joining user from accessing past communications, all keys along the path from the joining point to the root node need to be changed. After generating new keys for these nodes, server s needs to securely distribute them to the existing users as well as the joining user. For example, as shown in Figure 5, suppose u_9 is granted to join the upper key graph in the figure. The joining point is k-node k_{78} in the key graph, and the key of this k-node is changed to k_{789} in the new key graph below. Moreover, the group key at the root is changed from k_{1-8} to k_{1-9}. Users $u_1, \ldots, u_6$ only need the new group key k_{1-9}, while users u_7, u_8, and u_9 need the new group key k_{1-9} as well as the new key k_{789} to be shared by them.

To securely distribute the new keys to the users, the server constructs and sends rekey messages to the users. A *rekey message* contains one or more encrypted new key(s), and a user needs to decrypt it with appropriate keys in order to get the new keys. We next present three different approaches to construct and send rekey messages.

User-oriented rekeying. Consider each user and the subset of new keys it needs. The idea of user-oriented rekeying is that for each user, the server constructs a rekey message that contains precisely the new keys needed by the user, and encrypts them using a key held by the user.

For example, as shown in Figure 5, for user u_9 to join the upper secure group in the figure, server s needs to send the following three rekey messages.

$$\begin{array}{lcl} s \rightarrow \{u_1, \ldots, u_6\} & : & \{k_{1-9}\}_{k_{1-8}} \\ s \rightarrow \{u_7, u_8\} & : & \{k_{1-9}, k_{789}\}_{k_{78}} \\ s \rightarrow u_9 & : & \{k_{1-9}, k_{789}\}_{k_9} \end{array}$$

Note that users $u_1, \ldots, u_6$ need to get the new group key k_{1-9}. There is no single key that is shared only by $u_1, \ldots, u_6$. However, key k_{1-8} can be used to encrypt the new key k_{1-9} for $u_1, \ldots, u_6$ without security breach since users u_7 and u_8 will also get this new group key from another rekey message.

User-oriented rekey messages can be constructed as follows. For each k-node x whose key has been changed, say from k to k', the server constructs a rekey message by encrypting the new keys of k-node x and all its ancestors (upto the root) by the old key k. This rekey message is then sent to the subset of users that need precisely these new keys. Either unicast or *subgroup multicast* may be used.[8] Moreover, one rekey message is sent to the joining user which contains all of the new keys encrypted by the individual key of the joining user.

This approach needs h rekey messages. Counting the number of keys encrypted, the encryption cost for the server is given by

$$1 + 2 + \ldots + h - 1 + h - 1 = \frac{h(h+1)}{2} - 1.$$

Key-oriented rekeying. In this approach, each new key is encrypted individually (except keys for the joining user). For each k-node x whose key has been changed, say from k to k', the server constructs two rekey messages. First, the server encrypts the new key k' with the old key k, and sends it to $userset(k)$ which is the set of users that share k. All of the original users that need the new key k' can get it from this rekey message. The other rekey message contains the new key k' encrypted by the individual key of the joining user, and is sent to the joining user.

As described, a user may have to get multiple rekey messages in order to get all the new keys it needs. For example, as shown in Figure 5, for user u_9 to join the upper secure group in the figure, server s needs to send the following four rekey messages. Note that users u_7, u_8, and u_9 need to get two rekey messages each.

$$\begin{array}{lcl} s \rightarrow \{u_1, \ldots, u_8\} & : & \{k_{1-9}\}_{k_{1-8}} \\ s \rightarrow u_9 & : & \{k_{1-9}\}_{k_9} \\ s \rightarrow \{u_7, u_8\} & : & \{k_{789}\}_{k_{78}} \\ s \rightarrow u_9 & : & \{k_{789}\}_{k_9} \end{array}$$

Compared to user-oriented rekeying, the above approach reduces the encryption cost of the server from $\frac{h(h+1)}{2} - 1$ to $2(h-1)$, but it requires $2(h-1)$ rekey messages instead of h.

To reduce the number of rekey messages, all of the rekey messages for a particular user can be combined and sent as one message. Thus, server s can send the following three rekey messages instead of the four rekey messages shown above.

$$\begin{array}{lcl} s \rightarrow \{u_1, \ldots, u_6\} & : & \{k_{1-9}\}_{k_{1-8}} \\ s \rightarrow \{u_7, u_8\} & : & \{k_{1-9}\}_{k_{1-8}}, \{k_{789}\}_{k_{78}} \\ s \rightarrow u_9 & : & \{k_{1-9}, k_{789}\}_{k_9} \end{array}$$

The join protocol based upon this rekeying strategy is presented in Figure 6. Steps (4) and (5) in Figure 6 specify how the combined rekey messages are constructed and distributed by server s.

Using combined rekey messages, the number of rekey messages for key-oriented rekeying is h (same as user-

[8] A rekey message can be sent via multicast to a subgroup if a multicast address has been established for the subgroup in addition to the multicast address for the entire group. Alternatively, the method in [13] may be used in lieu of allocating a large number of multicast addresses for subgroups. See Section 7 for more discussion.

oriented rekeying) while the encryption cost is $2(h-1)$. From this analysis, key-oriented rekeying is clearly better for the server than user-oriented rekeying. (This conclusion is confirmed by measurement results presented in Section 5.)

(1) $u \rightarrow s$: join request
(2) $s \Leftrightarrow u$: authenticate u and distribute k_u
(3) s : find a joining point and attach k_u,
let x_j denote the joining point, x_0 the root,
and x_{i-1} the parent of x_i for $i = 1, \ldots, j$,
let K_{j+1} denote k_u,
and $K_0, \ldots, K_j$ the old keys of $x_0, \ldots, x_j$,
randomly generate new keys $K'_0, \ldots, K'_j$
(4) for $i = 0$ upto j do
let $M = \{K'_0\}_{K_0}, \ldots, \{K'_i\}_{K_i}$
$s \rightarrow (userset(K_i) - userset(K_{i+1})) : M$
(5) $s \rightarrow u : \{K'_0, \ldots, K'_j\}_{k_u}$

Figure 6: Join protocol for a tree key graph (key-oriented rekeying).

Group-oriented rekeying. In key-oriented rekeying, each new key is encrypted individually (except keys for the joining user). The server constructs multiple rekey messages, each tailored to the needs of a subgroup. Specifically, the users of a subgroup receive a rekey message containing precisely the new keys each needs.

An alternative approach, called group-oriented, is for the server to construct a single rekey message containing all new keys. This rekey message is then multicasted to the entire group. Clearly such a rekey message is relatively large and contains information not needed by individual users. However, scalability is not a concern because the message size is $O(\log_d(n))$ for group size n and key tree degree d. The group-oriented approach has several advantages over key-oriented and user-oriented rekeying. First, there is no need for subgroup multicast. Second, with fewer rekey messages, the server's per rekey message overheads are reduced. Third, the total number of bytes transmitted by the server per join/leave request is less than those of key-oriented and user-oriented rekeying which duplicate information in rekey messages. (See Section 5 and Section 7 for a more thorough discussion on performance comparisons.)

For example, as shown in Figure 5, for user u_9 to join the upper secure group in the figure, server s needs to send the following two rekey messages; one is multicasted to the group, and the other is unicasted to the joining user.

$$s \rightarrow \{u_1, \ldots, u_8\} \quad : \quad \{k_{1-9}\}_{k_{1-8}}, \{k_{789}\}_{k_{78}}$$
$$s \rightarrow u_9 \quad : \quad \{k_{1-9}, k_{789}\}_{k_9}$$

The join protocol based upon group-oriented rekeying is presented in Figure 7. This approach reduces the number of rekey messages to one multicast message and one unicast message, while maintaining the encryption cost at $2(h-1)$ (same as key-oriented rekeying).

(1) - (3) (same as Figure 6)
(4) $s \rightarrow userset(K_0) : \{K'_0\}_{K_0}, \ldots, \{K'_j\}_{K_j}$
(5) $s \rightarrow u : \{K'_0, \ldots, K'_j\}_{k_u}$

Figure 7: Join protocol for a tree key graph (group-oriented rekeying).

3.4 Leaving a tree key graph

After granting a leave request from user u, server s updates the key graph by deleting the u-node for user u and the k-node for its individual key from the key graph. The parent of the k-node for its individual key is called the *leaving point*.

To prevent the leaving user from accessing future communications, all keys along the path from the leaving point to the root node need to be changed. After generating new keys for these k-nodes, server s needs to securely distribute them to the remaining users. For example, as shown in Figure 5, suppose u_9 is granted to leave the lower key graph in the figure. The leaving point is the k-node for k_{789} in the key graph, and the key of this k-node is changed to k_{78} in the new key graph above. Moreover, the group key is also changed from k_{1-9} to k_{1-8}. Users $u_1, \ldots, u_6$ only need to know the new group key k_{1-8}. Users u_7 and u_8 need to know the new group key k_{1-8} and the new key k_{78} shared by them.

To securely distribute the new keys to users after a leave, we revisit the three rekeying strategies.

User-oriented rekeying In this approach, each user gets a rekey message in which all the new keys it needs are encrypted using a key it holds. For example, as shown in Figure 5, for user u_9 to leave the lower secure group in the figure, server s needs to send the following four rekey messages.

$$s \rightarrow \{u_1, u_2, u_3\} \quad : \quad \{k_{1-8}\}_{k_{123}}$$
$$s \rightarrow \{u_4, u_5, u_6\} \quad : \quad \{k_{1-8}\}_{k_{456}}$$
$$s \rightarrow u_7 \quad : \quad \{k_{1-8}, k_{78}\}_{k_7}$$
$$s \rightarrow u_8 \quad : \quad \{k_{1-8}, k_{78}\}_{k_8}$$

User-oriented rekey messages for a leave can be constructed as follows. For each k-node x whose key has been changed, say from k to k', and for each unchanged child y of x, the server constructs a rekey message by encrypting the new keys of k-node x and all its ancestors (upto the root) by the key K of k-node y. This rekey message is then multicasted to $userset(K)$.

This approach requires $(d-1)(h-1)$ rekey messages. The encryption cost for the server is given by

$$(d-1)(1+2+\ldots+h-1) = \frac{(d-1)h(h-1)}{2}.$$

Key-oriented rekeying In this approach, each new key is encrypted individually. For example, as shown in Figure 5, for user u_9 to leave the lower secure group in the figure, server s needs to send the following four rekey messages.

$$s \rightarrow \{u_1, u_2, u_3\} \quad : \quad \{k_{1-8}\}_{k_{123}}$$
$$s \rightarrow \{u_4, u_5, u_6\} \quad : \quad \{k_{1-8}\}_{k_{456}}$$
$$s \rightarrow u_7 \quad : \quad \{k_{1-8}\}_{k_{78}}, \{k_{78}\}_{k_7}$$
$$s \rightarrow u_8 \quad : \quad \{k_{1-8}\}_{k_{78}}, \{k_{78}\}_{k_8}$$

The leave protocol based upon key-oriented rekeying is presented in Figure 8. Step (4) in Figure 8 specifies how the rekey messages are constructed and distributed to users.

Note that by storing encrypted new keys for use in different rekey messages, the encryption cost of this approach is $d(h-1)$, which is much less than that of user-oriented rekeying. The number of rekey messages is $(d-1)(h-1)$, same as user-oriented rekeying.

Group-oriented rekeying. A single rekey message is constructed containing all new keys. For example, as shown in Figure 5, for user u_9 to leave the lower secure group in the figure, server s needs to send the following rekey message:

(1) $u \rightarrow s$: { leave-request }$_{k_u}$
(2) $s \rightarrow u$: { leave-granted }$_{k_u}$
(3) s : find the leaving point (the parent of k_u),
remove k_u from the tree,
let x_{j+1} denote the deleted k-node for k_u,
x_j the leaving point, x_0 the root,
and x_{i-1} the parent of x_i for $i = 1, \ldots, j$,
randomly generate keys $K'_0, \ldots, K'_j$
as the new keys of $x_0, \ldots, x_j$
(4) for $i = 0$ upto j do
for each child y of x_i do
let K denote the key at k-node y
if $y \neq x_{i+1}$ then do
let $M = \{K'_i\}_K, \{K'_{i-1}\}_{K'_i}, \ldots, \{K'_0\}_{K'_1}$
$s \rightarrow userset(K) : M$

Figure 8: Leave protocol for a tree key graph (key-oriented rekeying).

let L_0 denote $\{k_{1-8}\}_{k_{123}}, \{k_{1-8}\}_{k_{456}}, \{k_{1-8}\}_{k_{78}}$
let L_1 denote $\{k_{78}\}_{k_7}, \{k_{78}\}_{k_8}$
$s \rightarrow \{u_1, \ldots, u_8\}$: L_0, L_1

Note that for a leave, this single rekey message is about d times bigger than the rekey message for a join, where d is the average degree of a k-node.

The leave protocol based upon group-oriented rekeying is presented in Figure 9. This approach uses only one rekey message which is multicasted to the entire group, and the encryption cost is $d(h-1)$, same as key-oriented rekeying.

(1) - (3) (same as Figure 8)
(4) for $i = 0$ upto j do
let $\{z_1, \ldots, z_r\}$ be the set of the children of x_i
let $J_1, \ldots, J_r$ denote the keys at $z_1, \ldots, z_r$
let L_i denote $\{K'_i\}_{J_1}, \ldots, \{K'_i\}_{J_r}$
$s \rightarrow userset(K'_0) : L_0, \ldots, L_j$

Figure 9: Leave protocol for a tree key graph (group-oriented rekeying).

3.5 Cost of encryptions and decryptions

An approximate measure of the computational costs of the server and users is the number of key encryptions and decryptions required by a join/leave operation. Let n be the number of users in a secure group. For each join/leave operation, the user that requests the operation is called the *requesting user*, and the other users in the group are *non-requesting users*. For a join/leave operation, we tabulate the cost of a requesting user in Table 2(a), the cost of a non-requesting user in Table 2(b), and the cost of the server in Table 2(c). These costs are from the protocols described above for star and tree key graphs, and from [21] for complete key graphs. (Key-oriented or group-oriented rekeying is assumed for tree key graphs.)

For a key tree, recall that d and h denote the degree and height of the tree respectively. In this case, for a non-requesting user u, the average cost of u for a join or a leave is less than $\frac{d}{d-1}$ which is independent of the size of the tree (derivation in [21]).

Assuming that the number of join operations is the same as the number of leave operations, the average costs per operation are tabulated in Table 3 for the server and a user in the group.

(a)	the requesting user		
	Star	Tree	Complete
`join`	1	$h-1$	2^n
`leave`	0	0	0
(b)	a non-requesting user		
	Star	Tree	Complete
`join`	1	$\frac{d}{d-1}$	2^{n-1}
`leave`	1	$\frac{d}{d-1}$	0
(c)	the server		
	Star	Tree	Complete
`join`	2	$2(h-1)$	2^{n+1}
`leave`	$n-1$	$d(h-1)$	0

Table 2: Cost of a join/leave operation.

cost	Star	Tree	Complete
cost of the server	$n/2$	$(d+2)(h-1)/2$	2^n
cost of a user	1	$d/(d-1)$	2^n

Table 3: Average cost per operation.

From Table 3, it is obvious that complete key graphs should not be used. On the other hand, scalable group key management can be achieved by using tree key graphs. Note that for a full and balanced d-ary tree, the average server cost is $(d+2)(h-1)/2 = (d+2)(\log_d(n))/2$. However, each user has to do slightly more work (from 1 to $\frac{d}{d-1}$). For $d = 4$, a user needs to do 1.33 decryptions on the average instead of one. (It can be shown that the server cost is minimized for $d = 4$, i.e., the optimal degree of key trees is four.)

4 Technique for Signing Rekey Messages

In our join/leave protocols, each rekey message contains one or more new keys. Each new key, destined for a set of users, is encrypted by a key known only to these users and the server. It is possible for a user to masquerade as the server and send out rekey messages to other users. Thus if users cannot be trusted, then each rekey message should be digitally signed by the server.

We note that a digital signature operation is around two orders of magnitude slower than a key encryption using DES. For this reason, it is highly desirable to reduce the number of digital signature operations required per join/leave. If each rekey message is signed individually, then group-oriented rekeying, using just one rekey message per join/leave, would be far superior to key-oriented (user-oriented) rekeying, which uses many rekey messages per join/leave.

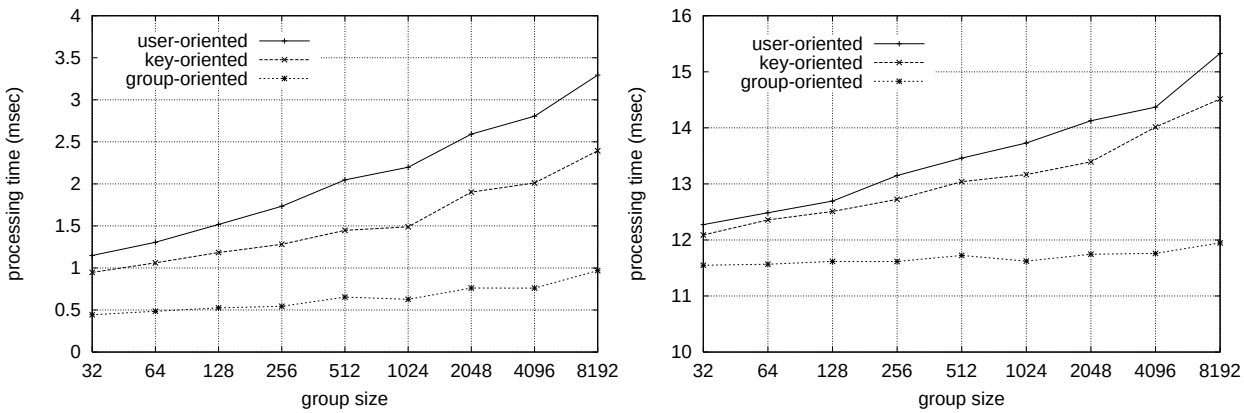
Consider m rekey messages, $M_1, \ldots, M_m$, with message digests, $d_i = h(M_i)$ for $i = 1, \ldots, m$, where $h()$ is a secure message digest function such as MD5. The standard way to provide authenticity is for the server to sign each message digest (with its private key) and send the signed message digest together with the message. This would require m digital signature operations for m messages.

We next describe a technique, implemented in our prototype key server, for signing a set of messages using just a single digital signature operation. The technique is based upon a scheme proposed by Merkle [14].

Suppose there are four messages with message digests d_1, d_2, d_3, and d_4. Construct message D_{12} containing d_1 and d_2, and compute message digest $d_{12} = h(D_{12})$. Similarly, construct message D_{34} containing d_3 and d_4, and compute message digest $d_{34} = h(D_{34})$. Then construct message

key tree degree 4	one signature per rekey msg					one signature for all rekey msgs				
	msg size (byte)		proc time (msec)			msg size (byte)		proc time (msec)		
	join	leave	join	leave	ave	join	leave	join	leave	ave
user	263.1	233.8	76.7	204.6	140.6	312.8	306.9	13.6	17.1	15.3
key	303.0	270.9	76.3	203.8	140.1	352.8	344.0	13.1	15.9	14.5
group	525.5	1005.7	11.9	12.0	11.9	525.5	1005.7	11.9	12.0	11.9

Table 4: Average rekey message size and server processing time (n=8192, DES, MD5, RSA)



(a) with encryption only (b) with encryption and signature

Figure 10: Server processing time per request vs group size (key tree degree 4).

D_{1-4} containing d_{12} and d_{34}, and compute message digest $d_{1-4} = h(D_{1-4})$. The server signs message digest d_{1-4} with its private key. The server then sends the signed message digest, $sign(d_{1-4})$, together with D_{1-4}, D_{34}, and M_4 to a user that needs M_4.

The user verifies, by first decrypting $sign(d_{1-4})$, that $d_{1-4} = h(D_{1-4})$. Subsequently, the user verifies that d_{34} in D_{1-4} is equal to $h(D_{34})$, and also d_4 in D_{34} is equal to $h(M_4)$, which assures that M_4 was indeed sent by the server. The above example can be easily extended to m messages in general.

The benefits of this technique for signing rekey messages are demonstrated in Table 4 for both key-oriented and user-oriented rekeying. (Note that it is not needed by group-oriented rekeying which uses one rekey message per join/leave.) The average rekey message size per join/leave is shown, as well as the server's processing time per join/leave (*ave* denotes the average of average join and leave processing times). The experiments were performed for an initial group size of 8192, with DES-CBC encryption, MD5 message digest, and RSA digital signature (512-bit modulus). Additional details of our experimental setup can be found in Section 5. With the technique for signing rekey messages, the processing time reduction for key-oriented and user-oriented rekeying is about a factor of ten (for example, 14.5 msec versus 140.1 msec in the case of key-oriented rekeying). There is however a small increase (around 50-70 bytes) in the average rekey message size.

5 Experiments and Performance Comparisons

We have designed and constructed a prototype group key server, as well as a client layer, which implement join/leave protocols for all three rekeying strategies in Section 3 and the technique for signing rekey messages in Section 4.

We performed a large number of experiments to evaluate the performance of the rekeying strategies and the technique for signing rekey messages. The experiments were carried out on two lightly loaded SGI Origin 200 machines running IRIX 6.4. The machines were connected by a 100 Mbps Ethernet. The group key server process runs on one SGI machine. The server is initialized from a specification file which determines the inital group size, the rekeying strategy, the key tree degree, the encryption algorithm, the message digest algorithm, the digital signature algorithm, etc. A client-simulator runs on the other SGI simulating a large number of clients. Actual rekey messages, as well as *join*, *join-ack*, *leave*, *leave-ack* messages, are sent between individual clients and the server using UDP over the 100 Mbps Ethernet. Cryptographic routines from the publicly available CrytoLib library are used [11].

For each experiment with an initial group size n, the client-simulator first sent n join requests, and the server built a key tree. Then the client-simulator sent 1000 join/leave requests. The sequence of 1000 join/leave requests was generated randomly according to a given ratio (the ratio was 1:1 in all our experiments to be presented). Each experiment was performed with three different sequences of 1000 join/leave requests. For fair comparisons (between different rekeying strategies, key trees of different degrees, etc.), the same three sequences were used for a given group size. The server employs a heuristic that attempts to build and maintain a key tree that is full and balanced. However, since the sequence of join/leave requests is randomly generated, it is unlikely that the tree is truly full and balanced at any time.

To evaluate the performance of different rekeying strategies as well as the technique for signing rekey messages, we measured rekey message sizes (in bytes) and processing time (in msec) used by the server per join/leave request. Specifically, the processing time per join/leave request consists of the following components. First, the server parses a request, traverses the key graph to determine which keys are to be updated, generates new keys, and updates the key graph.

key tree degree 4	rekey msg size (byte)						no. of rekey msgs					
	per join			per leave			per join			per leave		
	ave	min	max	ave	min	max	ave	min	max	ave	min	max
user	312.8	196	552	306.9	228	412	7.00	6	7	19.02	18	20
key	352.8	212	616	344.0	244	476	7.00	6	7	19.02	18	20
group	525.5	356	564	1005.7	968	1076	1.00	1	1	1.00	1	1
key tree degree 8	rekey msg size (byte)						no. of rekey msgs					
	per join			per leave			per join			per leave		
	ave	min	max	ave	min	max	ave	min	max	ave	min	max
user	287.3	196	496	285.9	228	356	5.00	4	5	29.01	28	30
key	319.3	212	544	314.3	244	404	5.00	4	5	29.01	28	30
group	464.5	284	492	1293.1	1256	1364	1.00	1	1	1.00	1	1
key tree degree 16	rekey msg size (byte)						no. of rekey msgs					
	per join			per leave			per join			per leave		
	ave	min	max	ave	min	max	ave	min	max	ave	min	max
user	274.0	180	452	282.4	244	344	4.00	3	4	46.01	45	47
key	302.0	196	492	306.6	260	384	4.00	3	4	46.01	45	47
group	427.8	248	456	1869.1	1832	1940	1.00	1	1	1.00	1	1

Table 5: Number and size of rekey messages, with encryption and signature, sent by the server (initial group size 8192)

Second, the server performs encryption of new keys and constructs rekey messages. Third, if message digest is specified, the server computes message digests of the rekey messages. Fourth, if digital signature is specified, the server computes message digests and a digital signature as described in Section 4. Lastly, the server sends out rekey messages as UDP packets using socket system calls.[9]

The server processing time per request (averaged over joins and leaves) versus group size (from 32 to 8192) is shown in Figure 10. Note that the horizontal axis is in log scale. The left figure is for rekey messages with DES-CBC encryption only (no message digest and no digital signature). The right figure is for rekey messages with DES-CBC encryption, MD5 message digest, and RSA-512 digital signature. The key tree degree was four in all experiments. We conclude from the experimental results that our group key management service is scalable to very large groups since the processing time per request increases (approximately) linearly with the logarithm of group size for all three rekeying strategies. Other experiments support the same conclusion for key tree degrees of 8 and 16.

The average server processing time versus key tree degree is shown in Figure 11. These experimental results illustrate three observations. First, the optimal degree for key trees is around four. Second, with respect to server processing time, group-oriented rekeying has the best performance, with key-oriented rekeying in second place. Third, signing rekey messages increases the server processing time by an order of magnitude (it would be another order of magnitude more for key-oriented and user-oriented rekeying without a special technique for signing multiple messages). The left hand side of the figure is for rekey messages with DES-CBC encryption only (no message digest and no digital signature). The right hand side of the figure is for rekey messages with DES-CBC encryption, MD5 message digest, and RSA-512 digital signature. The initial group size was 8192 in these experiments.

[9]The processing time is measured using the UNIX system call `getrusage()` which returns processing time (including time of system calls) used by a process. In the results presented herein, the processing time for a join request does not include any time used to authenticate the requesting user (i.e., step (2) in the join protocols of Figure 6 and Figure 7). We feel that any authentication overhead should be accounted for separately.

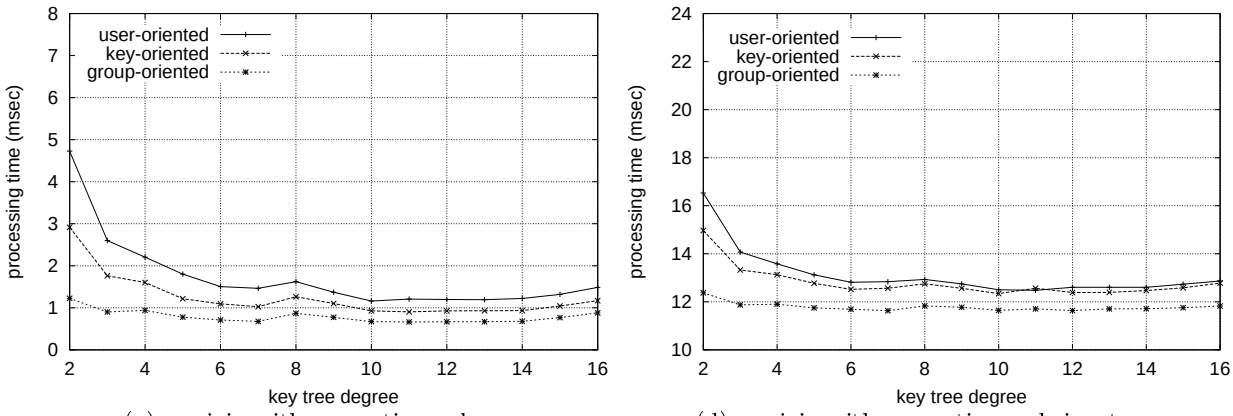
key tree degree 4	rekey msg size (byte) per join ave	rekey msg size (byte) per leave ave	no. of rekey msgs per join/leave
user	209.3	237.4	1
key	227.9	256.0	1
group	525.5	1005.7	1
key tree degree 8	rekey msg size (byte) per join ave	rekey msg size (byte) per leave ave	no. of rekey msgs per join/leave
user	200.0	242.0	1
key	217.2	259.2	1
group	464.5	1293.1	1
key tree degree 16	rekey msg size (byte) per join ave	rekey msg size (byte) per leave ave	no. of rekey msgs per join/leave
user	197.8	246.7	1
key	214.3	263.2	1
group	427.8	1869.1	1

Table 6: Number and size of rekey messages, with encryption and signature, received by a client (initial group size 8192)

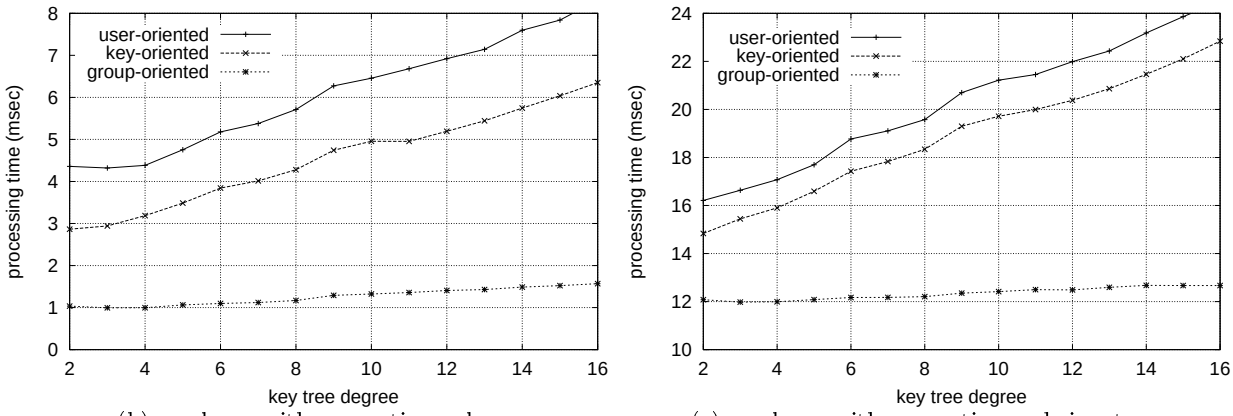
Table 5 presents the size and number of rekey messages sent by the server. Note that group-oriented rekeying uses a single large rekey message per request (sent via group multicast), while key-oriented and user-oriented rekeying use multiple smaller rekey messages per request (sent via subgroup multicast or unicast).[10] Note that the total number of bytes per join/leave transmitted by the server is much higher in key-oriented and user-oriented rekeying than in group-oriented rekeying.

Table 6 presents the size and number of rekey messages received by a client. Only the *average* message sizes are shown, because the minimum and maximum sizes are the same as those in Table 5. Note that each client gets exactly one rekey message for all three rekeying strategies. For key-oriented and user-oriented rekeying, the average message size is smaller than the corresponding average message size in Table 5. The is because the average message size here

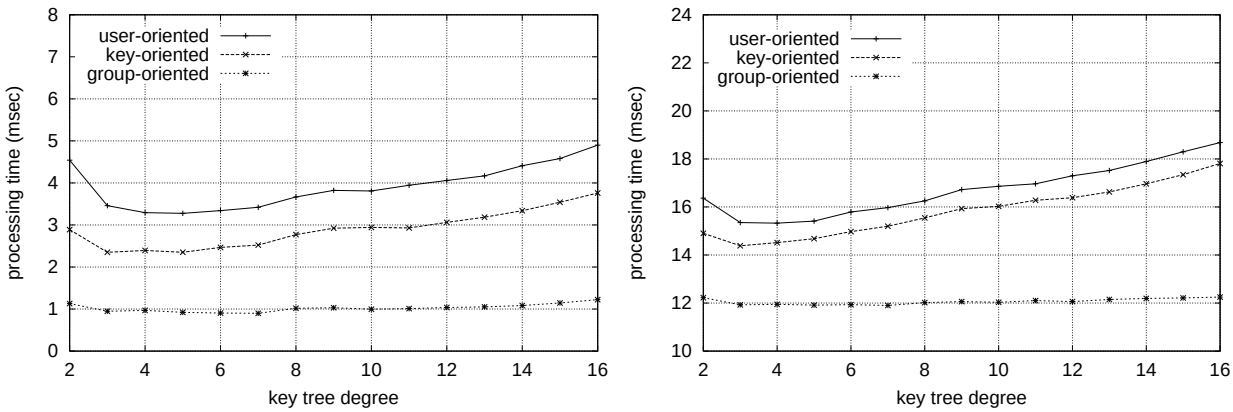
[10]The experiments reported herein were performed with each rekey message sent just once by the server via subgroup multicast.



(a) per join with encryption only



(b) per leave with encryption only



(c) per request with encryption only

(d) per join with encryption and signature

(e) per leave with encryption and signature

(f) per request with encryption and signature

Figure 11: Server processing time vs key tree degree (initial group size 8192).

was calculated over all clients, and many more clients received small rekey messages than clients that received large rekey messages. The results in this table show that group-oriented rekeying, which has the best performance on the server side, requires more work on the client side to process a larger message than key-oriented and user-oriented rekeying. The average rekey message size on the client side is the smallest in user-oriented rekeying.

From the contents of rekey messages, we counted and computed the average number of key changes by a client per join/leave request, which is shown in Figure 12. The top figure shows the average number of key changes versus the key tree degree, and the bottom figure shows the average number of key changes versus the initial group size of each experiment. Note that the average number of key changes by a client is relatively small, and is very close to the analytical result, $d/(d-1)$ shown in Table 3 in Section 3.

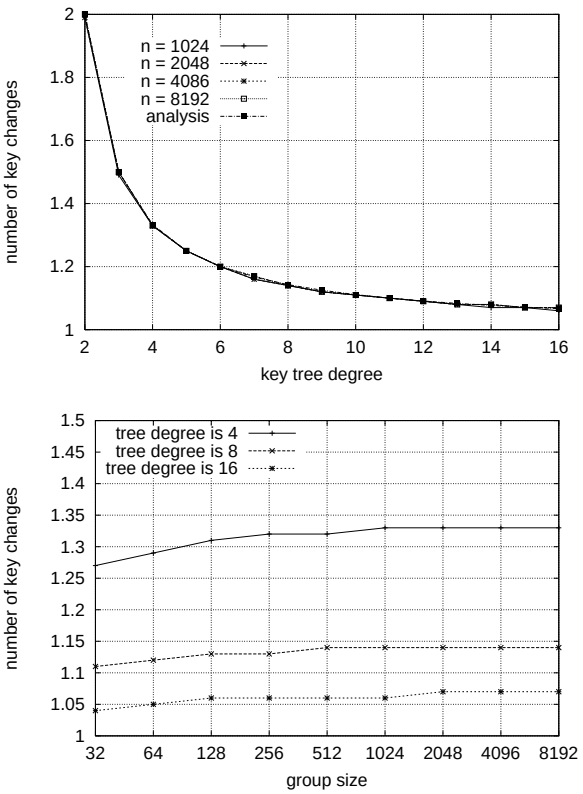


Figure 12: Number of key changes by a client per request.

6 Related Work

The scalability problem of group key management for a large group with frequent joins and leaves was previously addressed by Mittra with his Iolus system [15]. Both Iolus and our approach solve the scalability problem by making use of a hierarchy. The similarity, however, ends here. The system architectures are very different in the two approaches. We next compare them by considering a tree hierarchy with a single root (i.e., a single secure group).

Iolus's tree hierarchy consists of clients at the leaves with multiple levels of group security agents (agents, in short) above. For each tree node, the tree node (an agent) and its children (clients or lower-level agents) form a subgroup and share a subgroup key. There is no globally shared group key. Thus a join and leave in a subgroup does not affect other subgroups; only the local subgroup key needs to be changed.

Our tree hierarchy consists of keys, with individual keys at leaves, the group key at the root, and subgroup keys elsewhere. There is a single key server for all the clients. There are no agents, but each client is given multiple keys (its individual key, the group key, and some subgroup keys).

In comparing the two approaches, there are several issues to consider: performance, trust, and reliability.

Performance. Roughly speaking, since both approaches make use of a hierarchy, both attempt to change a $O(n)$ problem into a $O(\log(n))$ problem where n denotes group size. They differ however in where and when work is performed to achieve secure rekeying when a client joins/leaves the secure group.

Secure rekeying after a leave requires more work than after a join because, unlike a join, the previous group key cannot used and n rekey messages are required (this is referred to in [15] as a **1 does not equal n** type problem). This is precisely the problem solved by using a hierarchy in both approaches.

The main difference between Iolus and our approach is in how the **1 affects n** type problem [15] is addressed. In our approach, every time a client joins/leaves the secure group a rekeying operation is required which affects the entire group. Note that this is not a scalability concern in our approach because the server cost is $O(\log(n))$ and the client cost is $O(1)$.

In Iolus, there is no globally shared group key with the apparent advantage that whenever a client joins/leaves a subgroup only the subgroup needs to be rekeyed. However, for a client to send a message confidentially to the entire group, the client needs to generate a *message key* for encrypting the message and the message key has to be securely distributed to the entire group via agents. Each agent decrypts using one subgroup key to retrieve the message key and reencrypts it with another subgroup key for forwarding [15].

That is, most of the work in handling the **1 affects n** type problem is performed in Iolus when a client sends a message confidentially to the entire group (rather than when a client joins/leaves the group). In our approach, most of the work in handling the **1 affects n** type problem is performed when a client joins/leaves the secure group (rather than when a client sends messages confidentially to the entire group).

Trust. Our architecture requires a single trusted entity, namely, the key server. The key server may be replicated for reliability/performance enhancement, in which case, several trusted entities are needed. Each trusted entity should be protected using strong security measures (e.g. physical security, kernel security, etc.). In Iolus, however, there are many agents and all of the agents are trusted entities. Thus the level of trust required of the system components is much greater in Iolus than in our approach.

Reliability. In Iolus, agents are needed to securely forward message keys. When an agent fails, a backup is needed. It would appear that replicating a single key server (in our approach) to improve reliability is easier than backing up a large number of agents.[11]

7 Conclusions

We present three rekeying strategies, *user-oriented*, *key-oriented* and *group-oriented* and specify join/leave protocols based upon these strategies. For key-oriented and user-oriented rekeying, which use multiple rekey messages per join/leave, we present a technique for signing multiple messages with a single digital signature operation. Compared to using one digital signature per rekey message, the technique provides a tenfold reduction in the average server processing time of a join/leave.

The rekeying strategies and protocols are implemented in a prototype group key server we have built. From measurement results of a large number of experiments, we conclude that our group key management service using any of the three rekeying strategies is scalable to large groups with frequent joins and leaves. In particular, the average server processing time per join/leave increases linearly with the

[11] Craig Partridge observed that agents can be implemented in existing firewalls and derive their reliability and trustworthiness from those of firewalls.

logarithm of group size. We found that the optimal key tree degree is around four.

On the server side, group-oriented rekeying provides the best performance, with key-oriented rekeying in second place, and user-oriented rekeying in third place. On the client side, user-oriented rekeying provides the best performance, with key-oriented rekeying in second place, and group-oriented rekeying in third place. In particular, for a very large group whose clients are connected to the network via low-speed connections (modems), key-oriented or user-oriented rekeying would be more appropriate than group-oriented rekeying.

We next consider the amount of network traffic generated by the three rekeying strategies. With group-oriented rekeying, a single rekey message is sent per join/leave via multicast to the entire group, the network load generated would depend upon the network configuration (local area network, campus network, wide area Internet, etc.) and the group's geographic distribution. With key-oriented and user-oriented rekeying, many smaller rekey messages are sent per join/leave to subgroups. If the rekey messages are sent via unicast (because the network provides no support for subgroup multicast), the network load generated would be much greater than that of group-oriented rekeying.

It is possible to support subgroup multicast by the method in [13] or by allocating a large number of multicast addresses, one for each subgroup that share a key in the key tree being used. A more practical approach, however, is to allocate just a small number of multicast addresses (e.g., one for each child of the key tree's root node) and use a rekeying strategy that is a hybrid of group-oriented and key-oriented rekeying. It is straightforward to design such a hybrid strategy and specify the join/leave protocols. Furthermore a hybrid approach, involving the use of some Iolus agents at certain locations, such as firewalls, may also be appropriate.

Lastly, the reader may wonder why we use key graphs to specify a secure group even though key trees are sufficient for scalable management of a group key. This is because we are constructing a group key management service for applications that require the formation of multiple secure groups over a population of users and a user can join several secure groups. For these applications, the key trees of different group keys are merged to form a key graph [12].

Acknowledgement

We thank Craig Partridge for his constructive comments in shepherding the final revision of this paper.

References

[1] Tony Ballardie. *Scalable Multicast Key Distribution, RFC 1949*, May 1996.

[2] Tony Ballardie and Jon Crowcroft. Multicast-Specific Security Threats and Counter-Measures. In *Proceedings Symposium on Network and Distributed System Security*, 1995.

[3] Shimshon Berkovits. How to Broadcast a Secret. In D.W. Davies, editor, *Advances in cryptology, EUROCRYPT '91*, volume 547 of *Lecture Notes in Computer Science*, pages 535–541. Springer Verlag, 1991.

[4] R. Bird, I. Gopal, A. Herzberg, P. Janson, S. Kutten, R. Molva, and M. Yung. The KryptoKnight family of light-weight protocols for authentication and key distribution. *IEEE/ACM Transactions on Networking*, 3(1), February 1995.

[5] Guang-Huei Chiou and Wen-Tsuen Chen. Secure Broadcasting Using the Secure Lock. *IEEE Transactions on Software Engineering*, 15(8):929–934, August 1989.

[6] Stephen E. Deering. Multicast Routing in Internetworks and Extended LANs. In *Proceedings of ACM SIGCOMM '88*, August 1988.

[7] Amos Fiat and Moni Naor. Broadcast Encryption. In Douglas R. Stinson, editor, *Advances in cryptology, CRYPTO '93*, volume 773 of *Lecture Notes in Computer Science*, pages 480–491. Springer Verlag, 1994.

[8] Li Gong. Enclaves: Enabling Secure Collaboration over the Internet. *IEEE Journal on Selected Areas in Communications*, pages 567–575, April 1997.

[9] H. Harney and C. Muckenhirn. *Group Key Management Protocol (GKMP) Architecture, RFC 2094*, July 1997.

[10] H. Harney and C. Muckenhirn. *Group Key Management Protocol (GKMP) Specification, RFC 2093*, July 1997.

[11] J. B. Lacy, D. P. Mitchell, and W. M. Schell. CryptoLib: cryptography in software. In *Proceedings of USENIX: 4th UNIX Security Symposium*, October 1993.

[12] Simon S. Lam and Chung Kei Wong. Keystone: A Group Key Management Service. Work in progress, Department of Computer Sciences, The University of Texas at Austin.

[13] Brian Neil Levine and J.J. Garcia-Luna-Aceves. Improving Internet Multicast with Routing Labels. In *Proceedings of International Conference on Network Protocols*, 1997.

[14] Ralph C. Merkle. A Certified Digital Signature. In *Advances in Cryptology - CRYPTO '89*, 1989.

[15] Suvo Mittra. Iolus: A Framework for Scalable Secure Multicasting. In *Proceedings of ACM SIGCOMM '97*, 1997.

[16] B. Clifford Neuman. Proxy-Based Authorization and Accounting for Distributed Systems. In *Proceedings of 13th International Conference on Distributed Computing Systems*, pages 283–291, May 1993.

[17] Jennifer G. Steiner, Clifford Neuman, and Jeffrey I. Schiller. Kerberos: An Authentication Service for Open Network Systems. In *USENIX Winter Conference*, pages 191–202, February 1988.

[18] D.R. Stinson. On Some Methods for Unconditionally Secure Key Distribution and Broadcast Encryption. *Designs, Codes and Cryptography*, (12):215–243, 1997.

[19] J.J. Tardo and K. Alagappan. SPX: Global authentication using public key certificates. In *Proceedings of 12th IEEE Symposium on Research in Security and Privacy*, pages 232–244, May 1991.

[20] Debby M. Wallner, Eric J. Harder, and Ryan C. Agee. Key Management for Multicast: Issues and Architectures. Working draft, National Security Agency, July 1997.

[21] Chung Kei Wong, Mohamed Gouda, and Simon S. Lam. Secure Group Communications Using Key Graphs. Technical Report TR 97-23, Department of Computer Sciences, The University of Texas at Austin, July 1997.

[22] Thomas Y.C. Woo, Raghuram Bindignavle, Shaowen Su, and Simon S. Lam. SNP: An interface for secure network programming. In *Proceedings of USENIX'94 Summer Technical Conference*, June 1994.

[23] Thomas Y.C. Woo and Simon S. Lam. Designing a Distributed Authorization Service. In *Proceedings IEEE INFOCOM '98,* San Francisco, March 1998.