

# Batch Rekeying for Secure Group Communications\*

Xiaozhou Steve Li, Yang Richard Yang, Mohamed G. Gouda, Simon S. Lam  
Department of Computer Sciences  
University of Texas at Austin  
Austin, TX 78712-1188  
{xli,yangyang,gouda,lam}@cs.utexas.edu

## ABSTRACT

Many emerging web and Internet applications are based on a group communications model. Thus, securing group communications is an important Internet design issue. The *key graph* approach has been proposed for group key management. Key tree and key star are two important types of key graphs. Previous work has been focused on individual rekeying, i.e., rekeying after each join or leave request. In this paper, we first identify two problems with individual rekeying: inefficiency and an out-of-sync problem between keys and data. We then propose the use of periodic *batch rekeying* which can improve efficiency and alleviate the out-of-sync problem. We devise a marking algorithm to process a batch of join and leave requests. We then analyze the key server's processing cost for batch rekeying. Our results show that batch rekeying, compared to individual rekeying, saves server cost substantially. We also show that when the number of requests in a batch is not large, the best key tree degree is four; otherwise, key star (a special key tree with root degree equal to group size) outperforms small-degree key trees.

**Keywords:** Secure group communications, group key management, rekeying.

## 1. INTRODUCTION

Many emerging web applications are based on a group communications model [12, 4, 15]. In the Internet, multicast has been used successfully to provide an efficient, best-effort delivery service from a sender to a large group of receivers [6]. Thus, securing group communications (i.e., providing confidentiality, authenticity, and integrity of messages delivered between group members) will become an important Internet design issue.

One way to achieve secure group communications is to have a symmetric key, called *group key*, shared only by group members (also called users in this paper). The group key is distributed by a key server which provides group key management service. Messages sent by a member to the group are encrypted with the group

key, so that only members of the group can decrypt and read the messages.

Compared to two-party communications, a unique characteristic of group communications is that group membership can change over time: new users can join and current users can leave or be expelled. If a user wants to join the group, it sends a join request to the key server. The user and key server mutually authenticate each other using a protocol such as SSL [7]. If authenticated and accepted into the group, the user shares with the key server a symmetric key, called the user's *individual key*.

For a group of  $N$  users, initially distributing the group key to all users requires  $N$  messages each encrypted with an individual key (the computation and communication costs are proportional to group size  $N$ ). To prevent a new user from reading past communications (called backward access control) and a departed user from reading future communications (called forward access control), the key server may *rekey* (change the group key) whenever group membership changes. For large groups, join and leave requests can happen frequently. Thus, a group key management service should be scalable with respect to frequent key changes.

It is easier to rekey after a join than a 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 members (encrypted with the previous group key). After a leave, however, since the previous group key cannot be used, the new group key may be securely distributed by encrypting it with individual keys. This straightforward approach, however, is not scalable. In particular, rekeying costs 2 encryptions for a join and  $N - 1$  encryptions for a leave, where  $N$  is current group size (see Section 2.2).

The *key graph* approach [16, 17] has been proposed for scalable rekeying. In this approach, besides the group key and its individual key, each user is given several *auxiliary keys*. These auxiliary keys are used to facilitate rekeying. *Key graph* is a data structure that models user-key and key-key relationships. *Key tree* is an important type of key graph where key-key relationships are modeled as a tree. For a single leave request, key tree reduces server processing cost to  $O(\log N)$ . *Key star* is a special case of key tree. We provide more details of the key graph approach in Section 2.

A thorough performance analysis of *individual rekeying* (rekeying after each join or leave) is given in [17]. Individual rekeying, however, has two problems. First, it is relatively inefficient. Second, there is an out-of-sync problem between keys and data (see Section 4).

In this paper, we propose the use of periodic *batch rekeying* that can improve efficiency and alleviate the out-of-sync problem. In batch rekeying, the key server collects join and leave requests in a period of time and rekeys after a batch has been collected. We devise a marking algorithm to process a batch of requests. We then

\*Research sponsored by National Science Foundation grant No. ANI-9977267.

analyze the worst case and average case server processing costs for batch rekeying. Our results show that batch rekeying, compared to individual rekeying, saves server cost substantially. We also show that when the size of a batch is not large (roughly, when the number of joins is less than half of current group size, and the number of leaves is less than a quarter of current group size), four is the best key tree degree; otherwise, key star outperforms small-degree key trees. Our results provide guidelines for a key server to choose an appropriate data structure for group key management.

This paper is organized as follows. Section 2 briefly reviews the key graph approach. Section 3 identifies two problems with individual rekeying. Section 4 presents the batch rekeying idea. Section 5 presents the marking algorithm. Section 6 analyzes the server's processing cost for batch rekeying. Section 7 shows how to minimize server cost. Section 8 discusses related work. Section 9 concludes the paper.

## 2. KEY GRAPH APPROACH

The key graph approach [17] assumes that there is a single trusted and secure key server, and the key server uses a key graph for group key management. Key graph is a directed acyclic graph with two types of nodes: u-nodes, which represent users, and k-nodes, which represent keys. User  $u$  is given key  $k$  if and only if there is a directed path from u-node  $u$  to k-node  $k$  in the key graph. Key tree and key star are two important types of key graph. In a key tree, the k-nodes and u-nodes are organized as a tree. Key star is a special key tree where tree degree equals group size. To avoid confusion, from now on, we use key tree to mean small-degree key tree. We only consider key tree and key star in this paper.

### 2.1 Key Tree

In a key tree, the root is the group key, leaf nodes are individual keys, and the other nodes are auxiliary keys. Consider a group of 9 users  $u_1, \dots, u_9$ . A key tree of degree 3 is shown in Figure 2.1(a). In this figure, boxes represent u-nodes and circles represent k-nodes. User  $u_9$  is given 3 keys on the path from  $u_9$  to  $k_{1-9}$ :  $k_9$ ,  $k_{789}$ , and  $k_{1-9}$ .  $k_{1-9}$  is the group key.  $k_9$  is  $u_9$ 's individual key, which is shared by only  $u_9$  and the key server.  $k_{789}$  is an auxiliary key shared by  $u_7, u_8$ , and  $u_9$ . Suppose  $u_9$  wants to leave the group (from Figure 2.1(a) to 2.1(b)), the key server needs to change the keys that  $u_9$  has, that is, change  $k_{1-9}$  to a new key  $k_{1-8}$ ,  $k_{789}$  to a new key  $k_{78}$ . There are three strategies to distribute the new keys to the remaining users: user-oriented, key-oriented, and group-oriented [17]. For simplicity, we only consider group-oriented rekeying in this paper. Using group-oriented rekeying, the key server constructs the following rekey message and multicasts it to the whole group:

$$s \rightarrow u_1, \dots, u_8 : \begin{cases} \{k_{78}\}_{k_7}, \{k_{78}\}_{k_8}, \{k_{1-8}\}_{k_{123}}, \\ \{k_{1-8}\}_{k_{456}}, \{k_{1-8}\}_{k_{78}} \end{cases}$$

Here  $\{k'\}_k$  means key  $k'$  encrypted with key  $k$ . We call each item in the rekey message an encrypted key. Upon receiving the rekey message, a user extracts the encrypted keys that it needs. For example,  $u_7$  only needs  $\{k_{1-8}\}_{k_{78}}$  and  $\{k_{78}\}_{k_7}$ .

Similarly, suppose  $u_9$  wants to join a group of 8 users (from Figure 2.1(b) to 2.1(a)). The key server finds a joining point ( $k_{78}$  in this example) for the new user, constructs the rekey message, multicasts it to the whole group, and unicasts  $u_9$  the keys needed by  $u_9$ :

$$\begin{aligned} s \rightarrow u_1, \dots, u_9 & : \{k_{1-9}\}_{k_{1-8}}, \{k_{789}\}_{k_{78}} \\ s \rightarrow u_9 & : \{k_{1-9}, k_{789}\}_{k_9} \end{aligned}$$

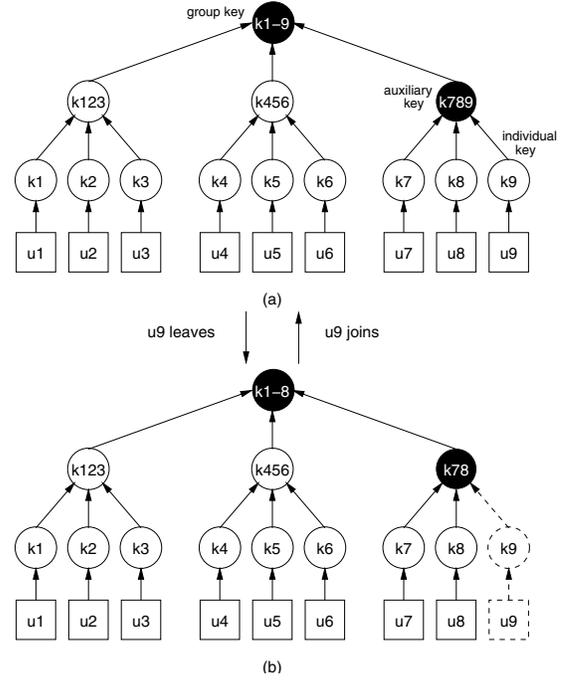


Figure 1: Example of a key tree.

From the above example, we can see that both the server's computation and communication costs are proportional to the number of encryptions to be performed (5 for the first example and 4 for the second example). Thus, we use *server cost* to mean the number of encryptions the key server has to perform. If the key tree degree is  $d$  and there are  $N$  users, assuming the key tree is a completely balanced tree, the server cost is  $2 \log_d N$  for a join and  $d \log_d N - 1$  for a leave. [17] showed that  $d = 4$  is the optimal degree for a leave.

### 2.2 Key Star

Key star is a special case of key tree where tree root degree equals group size. Key star models the straightforward approach. In key star, every user has two keys: its individual key and the group key. There is no auxiliary key. Figure 2(a) shows the key star for 4 users. Suppose  $u_4$  wants to leave the group (from Figure 2(a) to 2(b)), the key server encrypts the new group key  $k_{1-3}$  using every user's individual key, puts the encrypted keys in a message and multicasts it to the whole group:

$$s \rightarrow u_1, u_2, u_3 : \{k_{1-3}\}_{k_1}, \{k_{1-3}\}_{k_2}, \{k_{1-3}\}_{k_3}$$

Suppose  $u_4$  wants to join the group (from Figure 2(b) to 2(a)), the key server sends out the following messages:

$$\begin{aligned} s \rightarrow u_1, u_2, u_3 & : \{k_{1-4}\}_{k_{1-3}} \\ s \rightarrow u_4 & : \{k_{1-4}\}_{k_4} \end{aligned}$$

Clearly, using a key star, the server cost is 2 for a join and  $N - 1$  for a leave.

## 3. PROBLEMS WITH INDIVIDUAL REKEYING

Ideally, a departed user should be expelled from the group, and a new user be accepted to the group, as early as possible. Thus,

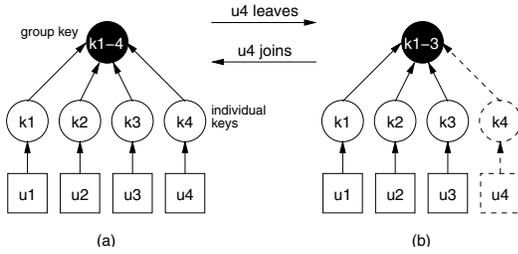


Figure 2: Example of a key star.

the key server should rekey immediately after receiving a join or leave request. We call this *individual rekeying*. Individual rekeying, however, has two problems: inefficiency and an out-of-sync problem between keys and data.

### 3.1 Inefficiency

Individual rekeying is relatively inefficient for two reasons. First, the rekey message has to be signed for authentication purpose, otherwise a compromised group user can send out bogus rekey messages and mess up the whole system. Signing operation is computationally expensive [18]. If, for every single request, the key server has to generate and sign a rekey message, the signing operation alone will place a heavy burden on the key server, especially when requests are frequent.

Second, consider two leaves that happen one after another. The key server generates two sets of new keys (group key and auxiliary keys) for these two leaves. These two leaves, however, might temporally happen so close to each other that the first set of new keys are actually not used and are immediately replaced by the second set of new keys. When requests are frequent, like during the startup or teardown of a multicast session, many new keys may be generated and distributed, while not used at all. This is a waste of server cost.

### 3.2 Out-of-Sync Problem

Individual rekeying also has the following *out-of-sync problem* between keys and data: a user might receive a data message encrypted by an old group key, or it might receive a data message encrypted with a group key that it has not received yet. Figure 3 shows an example of this problem. In this example, at time  $t_1$ ,  $u_2$  receives a data message encrypted with group key GK(2) from  $u_1$ , but  $u_2$  has not received GK(2); at time  $t_2$ ,  $u_1$  receives a data message encrypted with group key GK(0) from  $u_2$ , but  $u_1$ 's current group key is GK(2). Delay of reliable rekey message delivery<sup>1</sup> can be large and variable. Thus, this out-of-sync problem may require a user to keep many old group keys, and/or buffer a large amount of data encrypted with group keys that it has not received.

## 4. BATCH REKEYING

To address the above two problems, we propose the use of periodic *batch rekeying*. In batch rekeying, the key server waits for a period of time, called a *rekey interval*, collects all the join and leave requests during the interval, generates new keys, constructs a rekey message, and multicasts the rekey message.

<sup>1</sup>Reliable delivery of rekey messages is itself a research topic [18]. In this paper, we assume that there exists some mechanism for reliable delivery of rekey messages.

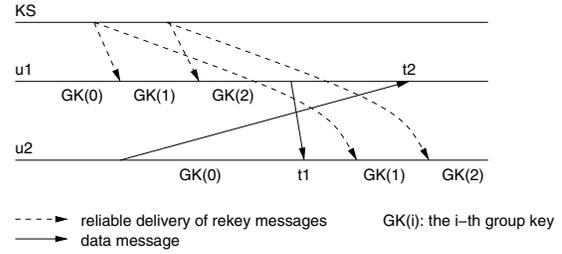


Figure 3: Out-of-sync problem

### 4.1 Efficiency

Batch rekeying improves efficiency because it reduces the number of rekey messages to be signed: one for a batch of requests, instead of one for each. Batch rekeying also takes advantage of the possible overlap of new keys for multiple rekey requests, and thus reduces the possibility of generating new keys that will not be used. In Section 6, we will quantify the performance benefit of batch rekeying.

### 4.2 Out-of-Sync Problem Alleviated

Batch rekeying alleviates the out-of-sync problem. To see this, let  $D_1$  be the maximum delay of reliable delivery of rekey messages,  $D_2$  be the maximum delay of data messages,  $I$  be the length of the rekey interval, and GK( $i$ ) be the  $i$ -th group key. That is, if a user's current group key is GK( $i$ ), when it receives a rekey message, its group key becomes GK( $i + 1$ ). It is not hard to show the following theorem. We skip the proof for simplicity.

**THEOREM 1.** *If  $D_1 + D_2 < I$  and a user's current group key is GK( $i$ ), then the user will only receive data messages encrypted with GK( $i - 1$ ), GK( $i$ ), or GK( $i + 1$ ), but not other group keys.*

This theorem indicates that if we make  $I$  relatively large compared to  $D_1$  and  $D_2$ , then a user only needs to keep GK( $i - 1$ ), GK( $i$ ), and its current auxiliary keys. If the user receives a data message encrypted with GK( $i + 1$ ), it has to keep them in some buffer and wait until the next rekey message arrives. Since the rekey interval is likely to be larger than message delays, the condition  $D_1 + D_2 < I$  is easy to satisfy.

### 4.3 Security Considerations

Batch rekeying is a tradeoff between performance and security. Since the key server does not rekey immediately, a departed user will remain in the group longer, and a new user has to wait longer to be accepted to the group.

We define *vulnerability window* to be the period of time from the key server receives a join or leave request, to all users receive the rekey message. We use the maximum size of vulnerability window, denoted as  $V$ , to measure security, because any group traffic sent within the vulnerability window is susceptible to be read by a departed user. The smaller  $V$ , the more secure the system is. We note that  $V$  also measures how soon a new user can be accepted to the group.

Figure 4(a) shows an example to illustrate the vulnerability window notion for individual rekeying. In this example,  $u_1$  requests to leave the group at  $t_1$ . The key server receives the request at  $t_2$  and rekeys immediately. The rekey message arrives  $u_2$  at  $t_5$ . But between  $t_2$  and  $t_5$ ,  $u_2$  sends out a message encrypted with GK(0) at  $t_3$ . The message arrives  $u_1$  at  $t_4$ . Although  $u_1$  has left the group at  $t_4$ , it is still able to read this message. The period between  $t_2$  and

$t_5$  is the vulnerability window. Figure 4(b) shows the vulnerability window notion for batch rekeying.

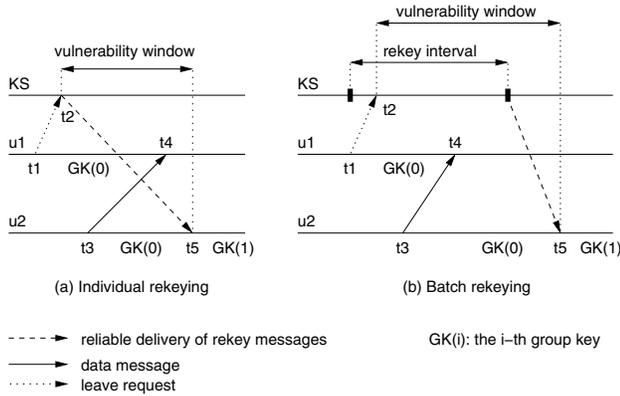


Figure 4: Vulnerability window.

It is not hard to see that  $V = D_1$  for individual rekeying and  $V = I + D_1$  for batch rekeying. So even in individual rekeying, a departed user is able to continue reading group communication for some period of time. Batch rekeying only makes this period longer and adjustable. Many applications can tolerate such a compromise. For example, a pay-per-view application can tolerate a viewer remaining in the group for a few more minutes after its subscription expires. An application can choose an acceptable  $I$  based on its own requirements.

As noted in the Introduction section, it is easier to rekey after a join than a leave. Thus, the access delay of a new user may be reduced by processing each new join immediately without rekeying the entire group, i.e., each new user is given the current group key and its auxiliary keys encrypted with its individual key, but leave requests are processed periodically in a batch. This approach offers new users faster access to the group at the cost of also giving new users a small amount of backward access. For simplicity, we will not discuss this approach in detail in the balance of this paper.

## 5. MARKING ALGORITHM

In this section, we present a *marking algorithm* for the key server to process a batch of requests. Obviously, if the key server uses key star, batch rekeying is a straightforward extension to individual rekeying. Thus, the marking algorithm applies to key tree only. In Section 6, we will analyze the resulting server processing cost for batch rekeying.

We use  $J$  to denote the number of joins in a batch and  $L$  to denote the number of leaves in a batch. We assume that within a batch, a user will not first join then leave, or first leave then join.

### 5.1 Marking Algorithm

Given a batch of requests, the main task for the key server is to identify which keys should be added, deleted, or changed. In individual rekeying, all the keys on the path from the request location to the root of the key tree have to be changed. When there are multiple requests, there are multiple paths. These paths form a subtree, called *rekey subtree*, which includes all the keys to be added or changed. The rekey subtree does not include individual keys.

The key server cannot control which users might leave, but it can control where in the key tree to place the new users. Thus, the key server should carefully place the new users (if there were any) so that the number of encryptions it has to perform is minimized. The

following marking algorithm uses some heuristics to achieve this objective. Figure 5 illustrates the idea. We use the word *key* and *node* interchangeably.

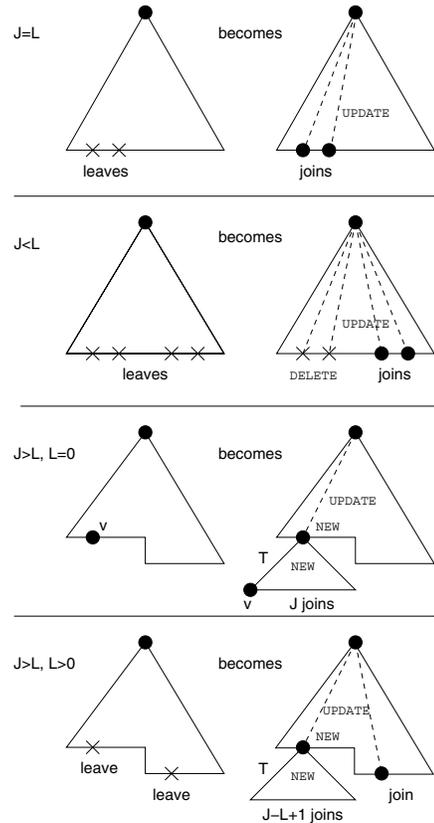


Figure 5: Marking algorithm.

- Case 1:  $J = L$ .
  1. Replace leaves by joins.
  2. Mark all the nodes from the replacement locations to the root UPDATE.
- Case 2:  $J < L$ .
  1. Out of the  $L$  leaves, pick  $J$  shallowest (smallest height) leaves. <sup>2</sup> Replace these  $J$  leaves with the  $J$  joins.
  2. Mark all the nodes from the root to the leave and replacement locations UPDATE or DELETE. Those leaving nodes without joining replacements are marked DELETE. A non-leaf node is marked DELETE if and only if all of its children are marked DELETE.
- Case 3:  $J > L$  and  $L = 0$ .
  1. Find a shallowest leaf node  $v$ . Remove  $v$  from the tree.
  2. Construct  $T$ , a complete but not necessarily balanced tree [11], that has all the new users and  $v$  as leaf nodes. The other nodes of  $T$  are new keys.
  3. Attach  $T$  to the old location of  $v$ .

<sup>2</sup>Keeping height information at the nodes is not difficult. We skip the details in this paper.

4. Mark all  $T$ 's internal nodes NEW and mark all the nodes from the root to the parent of  $v$ 's old location UPDATE.
- Case 4:  $J > L$  and  $L > 0$ .
    1. Replace all leaves by joins.
    2. Find a shallowest leaf node,  $v$ , among the replacement locations. Remove  $v$  from the tree.
    3. Construct a complete tree  $T$  that has the extra joins and  $v$  as leaf nodes. The other nodes of  $T$  are new keys.
    4. Attach  $T$  to the old location of  $v$ .
    5. Mark all  $T$ 's internal nodes NEW and mark all the keys from the replacement locations (except the old location of  $v$ ) to the root UPDATE.

After marking the key tree, the key server removes all nodes that are marked DELETE. The nodes marked UPDATE or NEW form the rekey subtree. The key server then traverses the rekey subtree, generates new keys, encrypts every new key by each of its children, constructs and multicasts the rekey message. It is not hard to see that the running time of the marking algorithm is  $O((J + L) \log_d N + N)$ , which is efficient (simulation results in section 6.3).

## 5.2 Two Examples

Figure 6 shows an example to illustrate the marking algorithm. In this example,  $u_4$  and  $u_9$  leave the group,  $u_{10}$  joins the group. The rekey subtree is marked as in the figure. Using group-oriented rekeying, the key server multicasts the following message to the group:

$$s \rightarrow G : \{k'_{456}\}_{k_{10}}, \{k'_{456}\}_{k_5}, \{k'_{456}\}_{k_6}, \{k'_{789}\}_{k_7}, \{k'_{789}\}_{k_8}, \{k'_{1-9}\}_{k_{123}}, \{k'_{1-9}\}_{k'_{456}}, \{k'_{1-9}\}_{k'_{789}}$$

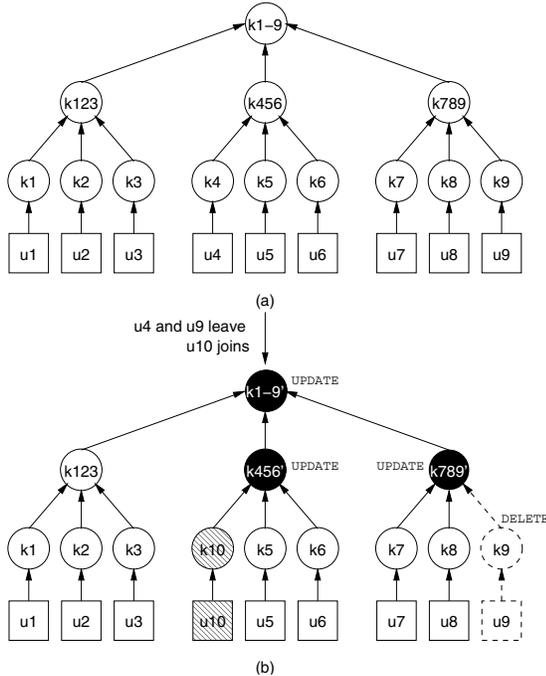


Figure 6: Marking algorithm example 1.

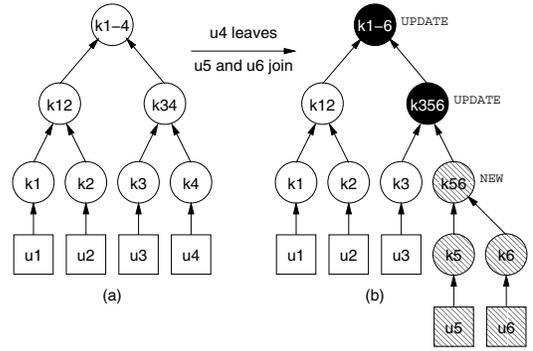


Figure 7: Marking algorithm example 2.

Figure 7 shows another example. In this example,  $u_4$  leaves the group,  $u_5$  and  $u_6$  join the group. The rekey subtree is marked as in the figure. Using group-oriented rekeying, the key server multicasts the following message to the group:

$$s \rightarrow G : \{k_{56}\}_{k_5}, \{k_{56}\}_{k_6}, \{k_{356}\}_{k_{56}}, \{k_{356}\}_{k_3}, \{k_{1-6}\}_{k_{12}}, \{k_{1-6}\}_{k_{356}}$$

## 5.3 Keeping the Key Tree Balanced

To achieve best performance, a key tree should be kept more or less balanced. Our marking algorithm aims to keep the tree balanced across multiple batches, by adding extra joins to the shallowest leaf node of the tree in each batch.

However, depending on the actual locations of the requests, even if the key tree starts complete and balanced, it is possible that the key tree may grow unbalanced after some number of batches. For example, many users located close to each other in the key tree may decide to leave at the same time. It is impossible to keep the key tree balanced all the time, without incurring extra cost. We refer the interested reader to [14] for discussions on this topic.

## 6. ANALYSIS

In this section, we analyze the server processing cost for batch rekeying. We consider the worst case and average case and compare batch rekeying against individual rekeying.

Key star's batch rekeying server cost, denoted as  $R_B(N, J, L)$ , is:

$$R_B(N, J, L) = \begin{cases} J + 1 & \text{if } L = 0 \\ N + J - L & \text{if } L > 0 \end{cases}$$

Thus, our analysis mainly focuses on key trees. For the purpose of analysis, we assume that the key tree is a complete and balanced tree at the beginning of a batch, and that each current user has equal probability of leaving.

Let  $d$  be the key tree degree,  $N$  be the group size at the beginning of a batch,  $h$  be the height of the key tree ( $h = \log_d N$ ),  $W(N, d, J, L)$  be the worst case batch rekeying server cost, and  $E(N, d, J, L)$  be the average case batch rekeying server cost.

### 6.1 Worst Case Analysis

The marking algorithm can control where to place joins, but cannot control where leaves happen. Thus, worst case analysis mainly considers how the locations of leaves affect the server cost. Since the marking algorithm takes different operations for four cases, worst case analysis is also divided into four cases.

- Case 1:  $J = L$ .

For simplicity, we first assume  $L = d^k$  for some integer  $k$ . When  $J = L$ , the worst case happens when the leaves are evenly distributed across the  $N$  leaf nodes in the key tree (see Figure 6.1(a)). The server cost for this case is:

$$W_=(N, d, J, L) = Ld \log_d \frac{N}{L} + \frac{d(L-1)}{d-1}$$

If  $L$  is not some power of  $d$ , suppose  $L = d^k + r$ ,  $0 < r < (d-1)d^k$ , then in the worst case, each of the  $r$  additional leaves adds  $d(h-k-1)$  to the total cost (see Figure 6.1(b)). Thus,

$$W_=(N, d, J, L) = d^{k+1}(h-k) + \frac{d(d^k-1)}{d-1} + rd(h-k-1)$$

In other words, if  $d^k < L < d^{k+1}$ ,  $W_=(N, d, J, L)$  grows linearly between  $W_=(N, d, J, d^k)$  and  $W_=(N, d, J, d^{k+1})$ .

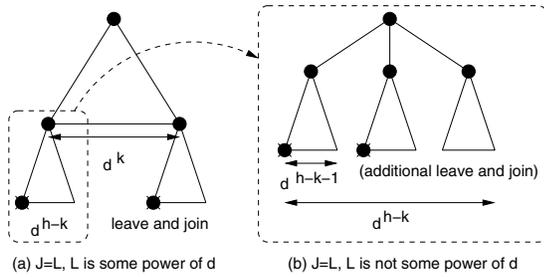


Figure 8: Worst case analysis.

- Case 2:  $J < L$ .

The analysis for this case is similar to the previous case. The only difference is that there are only  $N - (L - J)$  users left in this case. Thus,

$$W_<(N, d, J, L) = W_=(N, d, L, L) - (L - J)$$

- Case 3:  $J > L$  and  $L = 0$ .

In this case, the marking algorithm has full control of the server cost. It is not hard to see that a complete  $d$ -ary tree with  $n$  leaves is of size  $\lceil \frac{dn-1}{d-1} \rceil$ . Thus,  $T$ 's size is  $\lceil \frac{d(J+1)-1}{d-1} \rceil$  (see Figure 5) and there are  $\log_d N + 1$  nodes from the root to  $v$ . Thus:

$$W_>(N, d, J, 0) = \lceil \frac{dJ}{d-1} \rceil + 2 \log_d N$$

- Case 4:  $J > L$  and  $L > 0$ .

When there are more joins than leaves, the analysis is a combination of cases 1 and 3. Thus:

$$W_>(N, d, J, L) = W_=(N, d, L, L) + \lceil \frac{d(J-L)}{d-1} \rceil$$

Figure 9 shows  $W(N, d, J, L)$  for  $(N, d) = (1024, 2)$  and  $(N, d) = (4096, 4)$ , on a wide range of  $J$  and  $L$  values. We can see that, for fixed  $L$ ,  $W(N, d, J, L)$  is an increasing function of  $J$  (because more joins helps the key tree to "grow back"). For fixed  $J$ , as  $L$  becomes larger,  $W(N, d, J, L)$  first increases (because more leaves means more keys to be changed), then decreases (because now some keys are pruned from the tree). Clearly,  $W(N, d, J, L)$  is an increasing function of  $N$ . We will investigate the effect of  $d$  in Section 7.

## 6.2 Average Case Analysis

The server cost depends on the number of nodes belonging to the rekey subtree, and the number of children each node has. Thus, our technique for average case analysis is to consider the probability that an individual node belongs to the rekey subtree, and to consider the node's expected number of children (some children might be pruned). Again, we consider the following four cases.

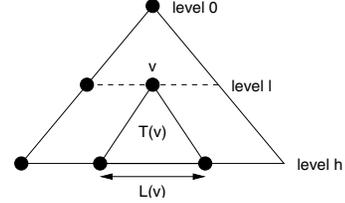


Figure 10: Average case analysis.

- Case 1:  $J = L$ .

This case forms the basis of our analysis for the other cases. Let the root of the key tree be at level 0, and the leaf nodes be at level  $h$ , where  $h = \log_d N$ . Let  $T(x)$  be the subtree rooted at node  $x$  and  $L(x)$  be the leaf nodes of  $T(x)$ . Consider a node  $v$  at level  $l$ ,  $0 \leq l \leq h-1$  (see Figure 10).  $v$  belongs to the rekey subtree if and only if there is at least one leaf in  $L(v)$ . Assuming every current user has equal probability of leaving, there are  $\binom{N}{L}$  ways to pick  $L$  leaving users out of  $N$  users. Among these many ways,  $\binom{N-N/d^l}{L}$  of them have no leaves in  $L(x)$ . Thus, the probability that  $v$  belongs to the rekey subtree is  $1 - \binom{N-N/d^l}{L} / \binom{N}{L}$ . Therefore,

$$E_=(N, d, J, L) = d \sum_{l=0}^{h-1} d^l \left(1 - \frac{\binom{N-N/d^l}{L}}{\binom{N}{L}}\right)$$

- Case 2:  $J < L$ .

When  $J < L$ , we should take into account the probability that some nodes might be marked DELETE and pruned from the tree. A node  $v$  is pruned if and only if all nodes in  $L(v)$  are leaves and none of them are replaced by joins. Using a similar technique as the previous case, we know the probability that a node at level  $l$  is pruned is

$$\frac{\binom{N-N/d^l}{L-J}}{\binom{N}{L}} \cdot \frac{\binom{L-N/d^l}{J}}{\binom{L}{J}} = \frac{\binom{L-J}{N/d^l}}{\binom{N}{N/d^l}}$$

Thus,

$$E_<(N, d, J, L) = E_=(N, d, J, L) - \sum_{l=0}^h d^l \frac{\binom{L-J}{N/d^l}}{\binom{N}{N/d^l}}$$

- Case 3:  $J > L$  and  $L = 0$ .

For this case,

$$E_>(N, d, J, 0) = W_>(N, d, J, 0)$$

- Case 4:  $J > L$  and  $L > 0$ .

The analysis for this case is a combination of cases 1 and 3. Thus:

$$E_>(N, d, J, L) = E_=(N, d, L, L) + \lceil \frac{d(J-L)}{d-1} \rceil$$

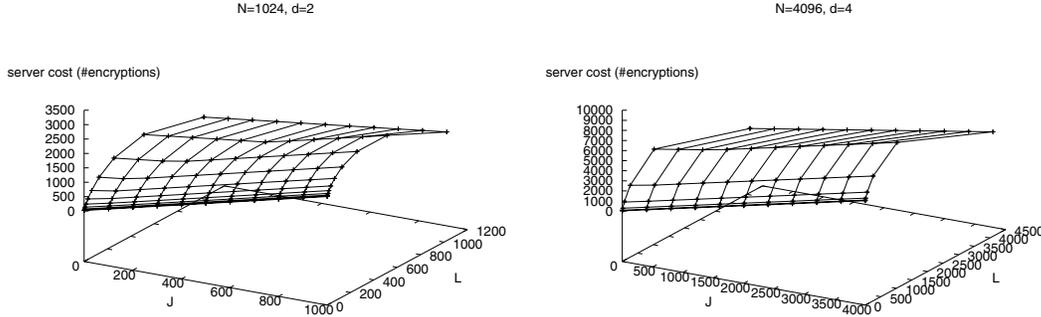


Figure 9: Worst case server cost.

### 6.3 Simulation

We built a simulator for the marking algorithm. The simulator first constructs a key tree and randomly picks  $L$  users to leave, it then runs the marking algorithm and calculates the server cost. We ran the simulation for 100 times and calculated the mean server cost. Simulations were done on a Sun Ultra Sparc I with 167MHz CPU and 128MB memory. The marking algorithm took at most 4.5ms when  $(N, d) = (1024, 2)$ , and at most 10ms when  $(N, d) = (4096, 4)$ , for all  $J < N$  and  $L < N$ .

Figure 11 shows  $E(N, d, J, L)$  for  $(N, d) = (1024, 2)$  and  $(N, d) = (4096, 4)$ , on a wide range of  $J$  and  $L$  values. Our analysis and simulation results match so well that we cannot tell the difference.

Figure 12 compares average case and worst case server costs. These two costs are rather close to each other when  $J$  and  $L$  are relatively small compared to  $N$ . Thus, we can consider worst case a good approximation to average case.

### 6.4 Batch vs. Individual Rekeying

In this section, we show that batch rekeying saves server cost substantially over individual rekeying. The actual save depends on whether the key server uses key star or key tree.

#### 6.4.1 Key Star

Let  $R_I(N, J, L)$  be the cost for processing  $J$  joins and  $L$  leaves individually. Clearly,

$$R_I(N, J, L) = \begin{cases} 2J & \text{if } L = 0 \\ (N-1)L + 2J & \text{if } L > 0 \end{cases}$$

Thus, the difference between batch rekeying and individual rekeying is:

$$R_I(N, J, L) - R_B(N, J, L) = \begin{cases} J-1 & \text{if } L = 0 \\ N(L-1) + J & \text{if } L > 0 \end{cases}$$

The difference is substantial, especially when  $J$  and  $L$  are large.<sup>3</sup>

#### 6.4.2 Key Tree

We have mentioned that using key tree for individual rekeying, the server cost is  $d \log_d N - 1$  for a leave and  $2 \log_d N$  for a join. Let  $S(N, d, J, L)$  be the server cost for rekeying a batch of  $J$  joins

<sup>3</sup>We observe that the exact cost for individually rekeying a batch of requests depends on the actual sequence of the requests. Since our purpose is to estimate the benefit of batch rekeying, instead of finding the exact benefit, we do not go into this detail. We will see that the benefit of batch rekeying is so substantial that the actual sequence variation is unlikely to make a big difference.

and  $L$  leaves individually. Clearly,

$$S(N, d, J, L) = (dL + 2J) \log_d N - L$$

We use this to be individual rekeying's cost<sup>4</sup>, and we use average case cost as batch rekeying's cost.

Figure 13 compares batch rekeying against individual rekeying for  $(N, d) = (1024, 2)$  and  $(N, d) = (4096, 4)$ . On the left figure, we show a narrower range of  $J$  and  $L$  values. On the right figure, we show a wider range. We can see that batch rekeying has a substantial benefit over individual rekeying.

## 7. MINIMIZATION OF SERVER COST

In the previous section, we have shown that batch rekeying saves server cost substantially. But when a key server uses batch rekeying, there are still two questions to answer. Shall the key server use key tree or key star? If it uses key tree, what degree should be the key tree? In this section, we show that the choice of data structure has big influence on server cost, and we provide guidelines for the key server to choose an appropriate one.

### 7.1 Key Tree vs. Key Star

First, we compare key star cost against 2-ary and 4-ary key tree costs. Figure 14 shows the comparison for  $N = 1024$  and  $N = 4096$ . The shadowed area is where key star is better. We observe that key star is better when  $L = 0$  or when  $J$  and  $L$  are large.

Figure 15 compares key star cost against key tree costs for  $N = 1024, d = 2, 4, 32$ , and  $N = 4096, d = 2, 4, 8, 16$ . We temporarily ignore the case where  $L = 0$ . Each curve in the figures corresponds to a  $d$  value. The curves are where key star cost is the same as  $d$ -ary key tree cost. Above a curve is where key star is better, while below a curve is where key tree is better. From this figure we draw a similar conclusion that key star is better than key tree when  $J$  and  $L$  are large. Roughly, when  $J < \frac{N}{2}$  and  $L < \frac{N}{4}$ , key tree is better than key star; otherwise, key star is better than key tree.

### 7.2 What tree degree is the best

[17] showed that  $d = 4$  is optimal for individual rekeying. We show in this section that  $d = 4$  is still optimal for batch rekeying.

Since server cost depends on both  $J$  and  $L$ , we first look at the special case when  $J = L$ . Figure 16 shows key tree costs for various  $d$  values, when  $J = L$ . We can see that  $d = 4$  is better than others when  $J$  and  $L$  are small. Figure 17 shows the server costs for key trees with various  $d$  values when  $J = 0$ . We also observe

<sup>4</sup>See footnote 3.

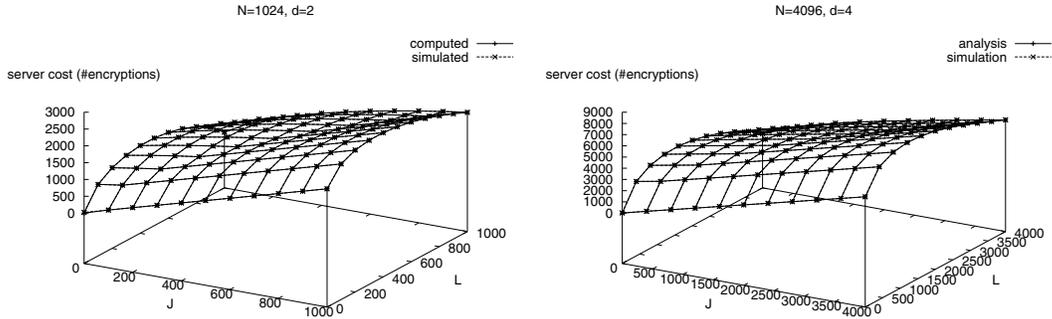


Figure 11: Average case server cost.

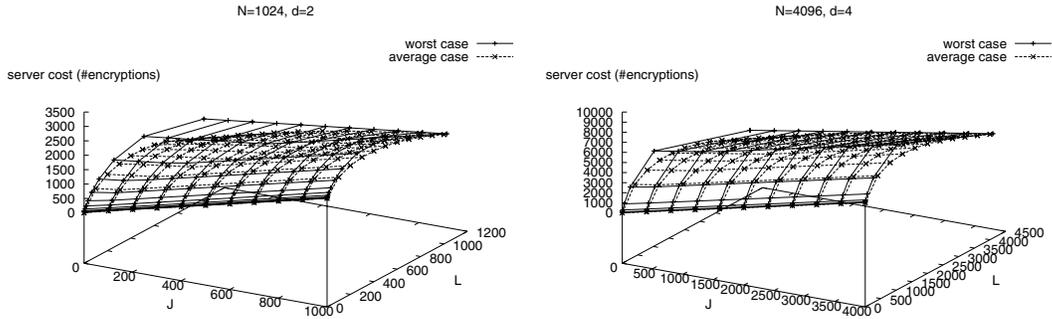


Figure 12: Average case and worst case server costs.

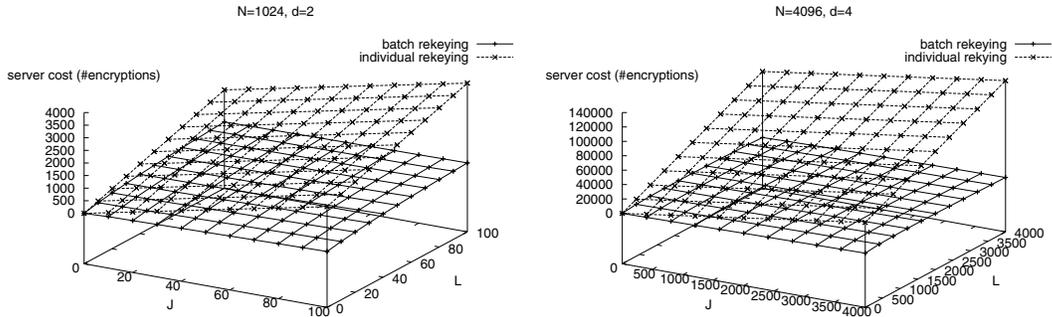


Figure 13: Batch rekeying vs. individual rekeying.

that  $d = 4$  is better than others when  $J$  and  $L$  are small. These two figures also show that  $d$  has a big influence on the resulting server cost. Figure 18 shows the area where  $d = 4$  is better than other  $d$  values. We can see that roughly, in the area where key tree is better than key star,  $d = 4$  is the best.

### 7.3 How to minimize server cost

From the above discussions we have seen that choosing an appropriate data structure has big influence on server cost. An application usually has an estimate of  $J$  and  $L$ , based on the expected number of users, length of the rekey interval, and the application's other characteristics. Thus, the guideline for a key server to choose an appropriate data structure is: if  $J < \frac{N}{2}$  and  $L < \frac{N}{4}$ , use a 4-ary key tree; otherwise, use a key star.

## 8. RELATED WORK

The topic of secure group communications has been investigated in [2, 3, 8, 10, 9, 13]. [13] addressed the need for frequent group key changes and the associated scalability problem. The approach proposed in [13] decomposes a large group of users into many sub-groups and employs a hierarchy of group security agents. This approach, however, requires many trusted entities.

Approaches that assume only a single trusted key server are proposed in [17, 16, 1]. In these approaches, a user is given some auxiliary keys, besides the group key and the individual key, to reduce the server cost for rekeying. These approaches have been mainly focused on reducing the server cost for individual rekeying.

[5] addressed the problem of batch rekeying and proposed using

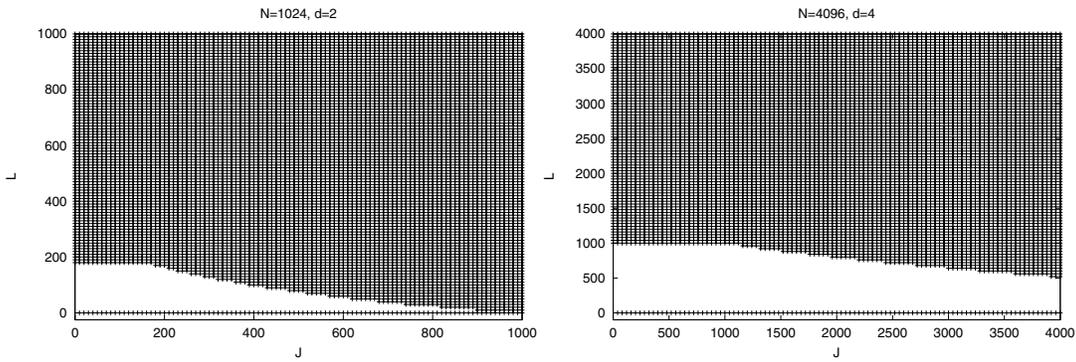


Figure 14: Area where key star is better than key tree.

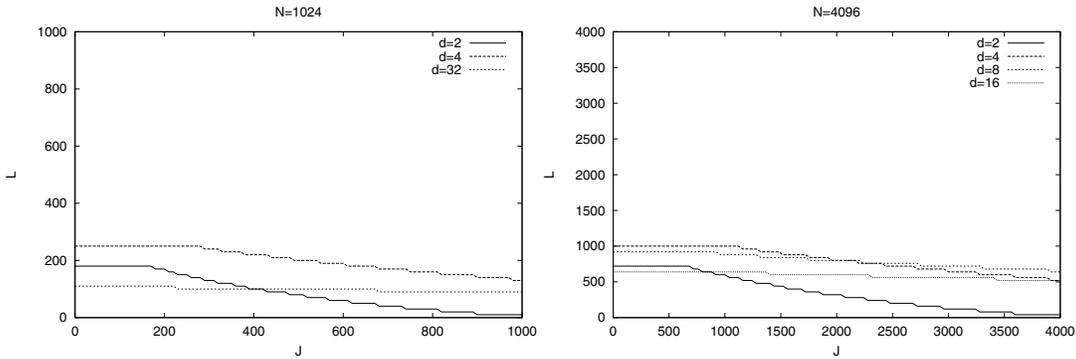


Figure 15: Curves where key tree and key star costs are the same.

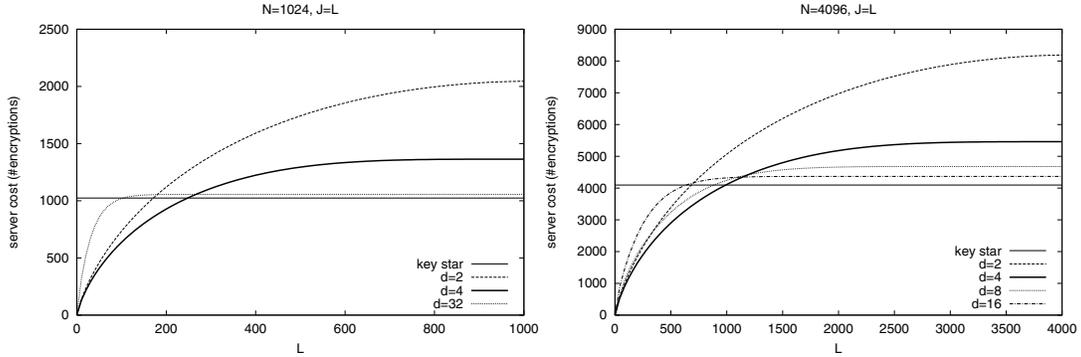


Figure 16: Comparison of different tree degrees, when  $J = L$ .

boolean function minimization technique to facilitate batch rekeying. Their approach, however, has the *collusion problem*, namely, two users can combine their knowledge of keys to continue reading group communications, even after they leave the group.

[14] addressed the problem of keeping the key tree balanced. Their approach essentially is to add joins at the shallowest leaf nodes of the tree, and re-structure the tree periodically. [14] also briefly described an algorithm for batch rekeying, in which joins replace leaves one by one, and if there are still extra joins, they are added to the shallowest leaf nodes of the tree one by one. [14]’s objective is to keep the key tree balanced, while our marking algorithm aims to reduce the server cost for batch rekeying. [14] did not provide any quantitative analysis for the benefit of batch rekeying.

## 9. CONCLUSION

This paper addressed the scalability problem of group key management. We identified two problems with individual rekeying: inefficiency and an out-of-sync problem between keys and data. We proposed the use of periodic batch rekeying to improve the key server’s performance and alleviate the out-of-sync problem. We devised a marking algorithm for the key server to process a batch of join and leave requests, and we analyzed the key server’s processing cost for batch rekeying. Our results show that batch rekeying, compared to individual rekeying, saves server cost substantially. We also show that when the number of requests is not large in a batch, four is the best key tree degree; otherwise, key star outperforms small-degree key trees.

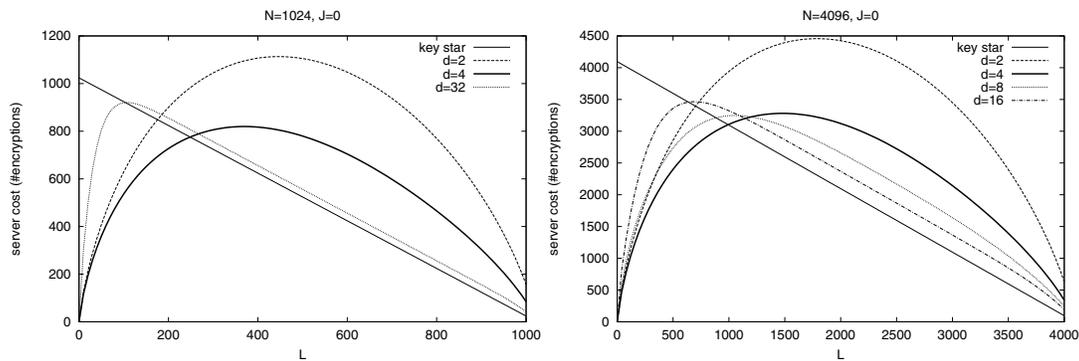


Figure 17: Comparison of different tree degrees, when  $J = 0$ .

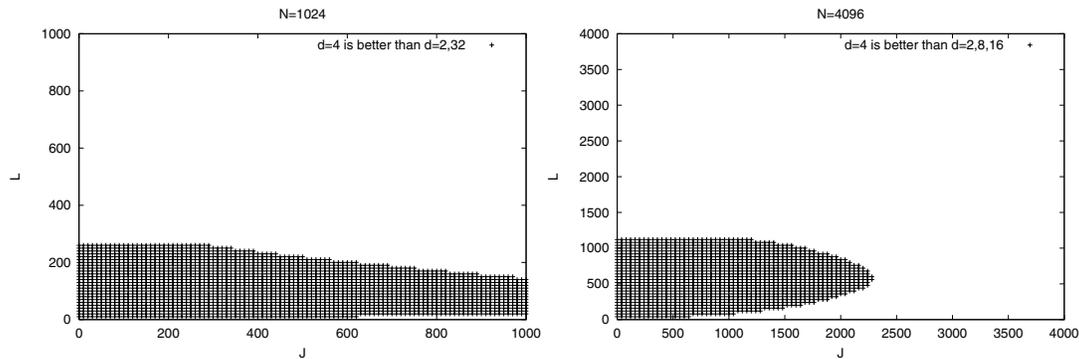


Figure 18: Area where degree 4 is better.

## 10. ACKNOWLEDGEMENT

We thank the anonymous reviewers for helpful comments.

## 11. REFERENCES

- [1] David Balenson, David McGrew, and Alan Sherman. *Key Management for Large Dynamic Groups: One-way Function Trees and Amortized Initialization*, INTERNET-DRAFT, 1999.
- [2] A. Ballardie. *Scalable Multicast Key Distribution*, RFC 1949, May 1996.
- [3] A. Ballardie and J. Crowcroft. Multicast-specific security threats and counter measures. In *Symposium on Network and Distributed System Security*, San Diego, CA, February 1995.
- [4] E. Burns. Webcast: collaborative document sharing via the mbone. Technical report, NCSA, 1995.
- [5] Isabella Chang, Robert Engel, Dilip Kandlur, Dimitrios Pendarakis, and Debanjan Saha. Key management for secure Internet multicast using boolean function minimization techniques. In *Proceedings of IEEE INFOCOM '99*, volume 2, March 1999.
- [6] Stephen E. Deering. Multicast routing in internetworks and extended LANs. In *Proceedings of ACM SIGCOMM '88*, August 1988.
- [7] Alan O. Freier, Philip Karlton, and Paul C. Kocher. The SSL protocol version 3.0. Work in progress, IETF Internet-Draft, March 1996.
- [8] Li Gong. Enclaves: Enabling secure collaboration over the internet. *IEEE Journal of Selected Areas in Communications*, pages 547–575, 1997.
- [9] H. Harney, C. Muckenhirn, and T. Rivers. *Group Key Management Protocol Architecture*, RFC 2094, July 1997.
- [10] H. Harney, C. Muckenhirn, and T. Rivers. *Group Key Management Protocol Specification*, RFC 2093, July 1997.
- [11] Donald Knuth. *The Art of Computer Programming*, volume 1, pages 401–402. Addison Wesley, 3rd edition, 1997.
- [12] T. Liao. Webcanal: a multicast web application. In *6th International WWW Conference, Santa Clara, CA, April 1997*, 1997.
- [13] S. Mitra. Iolus: A framework for scalable secure multicasting. In *Proceedings of ACM SIGCOMM '97*, September 1997.
- [14] M. J. Moyer, J. R. Rao, and P. Rohatgi. *Maintaining Balanced Key Trees for Secure Multicast*, INTERNET-DRAFT, June 1999.
- [15] P. Parnes, M. Mattson, K. Synnes, and D. Schefstrom. The mWeb presentation framework. *Computer Networks and ISDN Systems*, pages 1083–1090, 1997.
- [16] D. Wallner, E. Harder, and Ryan Agee. *Key Management for Multicast: Issues and Architectures*, INTERNET-DRAFT, September 1998.
- [17] Chung Kei Wong, Mohamed Gouda, and Simon S. Lam. Secure group communications using key graphs. In *Proceedings of ACM SIGCOMM '98*, September 1998.
- [18] Chung Kei Wong and Simon S. Lam. Keystone: a group key management system. In *International Conference on Telecommunications*, Acapulco, Mexico, May 2000.