Available online at www.sciencedirect.com
ScienceDirect
Computer Standards & Interfaces 31 (2009) 192–208
COMPUTER STANDARDS & INTERFACES
www.elsevier.com/locate/csi

Secure and efficient group key management with shared key derivation

Jen-Chiun Lin [a,*], Kuo-Hsuan Huang [a], Feipei Lai [a,b,c], Hung-Chang Lee [d]

[a] Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan, ROC
[b] Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan, ROC
[c] Grad. Institute of Biomedical Electronics and Bioinformatics, National Taiwan University, Taipei, Taiwan, ROC
[d] Department of Information Management, Tamkang University, Taipei, Taiwan, ROC

Received 24 April 2007; received in revised form 13 November 2007; accepted 18 November 2007
Available online 3 December 2007

Abstract

In many network applications, including distant learning, audio webcasting, video streaming, and online gaming, often a source has to send data to many receivers. IP multicasts and application-layer multicasts provide efficient and scalable one-to-many or many-to-many communications. A common secret key, the group key, shared by multiple users can be used to secure the information transmitted in the multicast communication channel. In this paper, a new group key management protocol is proposed to reduce the communication and computation overhead of group key rekeying caused by membership changes. With shared key derivation, new keys derivable by members themselves do not have to be encrypted or delivered by the server, and the performance of synchronous and asynchronous rekeying operations, including single join, single leave, and batch update, is thus improved. The proposed protocol is shown to be secure and immune to collusion attacks, and it outperforms the other comparable protocols from our analysis and simulation. The protocol is particularly efficient with binary key trees and asynchronous rekeying, and it can be tuned to meet different rekeying delay or key size requirements.

Keywords: Secure group communication; Group key management; Key tree; Shared key derivation

1. Introduction

More and more network applications are based on group communications model [27,39,40], where a message originated from a member will be delivered to the whole group. Typical group applications include distant learning, audio webcasting, video streaming, collaborative work, online gaming, and so on. Group communications can take advantages of multicast services, which provide more efficient, best-effort message delivery from a sender to multiple receivers. In the Internet, Internet Protocol (IP) multicast protocols, such as DVMRP [17] and PIM [18,19], have been widely used. Recently, application-layer multicasts [14] have emerged as an alternative to IP multicast protocols. Application-layer multicast protocols do not reply on IP-multicast-enabled routers, but on computers running the applications, and are suitable for peer-to-peer or wireless *ad hoc* networks. Security issues become more important as these applications and protocols become more mature. In this paper, we propose a new protocol to efficiently manage a shared group key among group members. The key can be used to encrypt transmitted data, or other session keys for group communication confidentiality.

The proposed key management protocol relies on a centralized key server that coordinates protocol runs to distribute the group key to group members securely. The server is responsible to update the group key when members join or leave the group [30]. A secure group key management protocol has to ensure that not only any user not belonging to the group cannot get the group keys, but also, for any member, it is computationally infeasible to derive group keys used before its participation, which means backward key secrecy, and after its departure, which means forward key secrecy. If group key secrecy is provided, group communication secrecy can be provided, since all transmitted data are protected by the group key.

Key trees [44], or hierarchical key structures, are widely used to realize efficient group key management. These configurations

* Corresponding author. Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan, ROC.
E-mail address: jenchiunlin@ntu.edu.tw (J.-C. Lin).

0920-5489/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.csi.2007.11.005

provide a scalable way to manage a group key for a large amount of users. While the group membership changes, usually due to users participation or members departure, the key server is able to update the group key with computation and communication costs logarithmically proportional to the group size. It has a huge advantage over the simple key distribution center solution. For a group of N members, the key server needs extra $O(N)$ storage for all auxiliary keys of the key tree, and each member only has to store additional $O(\log N)$ auxiliary keys. The storage overhead is acceptable in most circumstances.

Yang and Lam [47] have shown that, when there is only one key server, the amortized cost of a rekey operation of a group key management protocol basing solely on encryption cannot be done in less than $\Omega(\log N)$ encryptions for a group of N members. The lower bound of the total number of encryptions implies that key tree is an ideal hierarchical structure for centralized group key management, since it can achieve near optimal performance for group key rekeying. To further reduce the communication and computation costs of a rekey operation without resorting to a new kind of key hierarchy, besides encryption and random key generation, new cryptographic techniques should be used by the protocol to improve the rekeying performance.

In this paper, a group key management protocol based on shared key derivation, SKD, is proposed to reduce the communication and computation overhead of centralized secure group communication systems with key trees. With shared key derivation, the server does not have to encrypt and transmit new keys to members who have enough information to derive the keys on their own. The protocol can handle synchronous and asynchronous rekeying operations, and a new k-node insertion algorithm is designed to further optimize the key tree in batch update operations. The rest of this section give a brief overview of related researches on group key management protocols. Section 2 explains key trees and rekeying operations. Section 3 gives a detailed description of protocol SKD. In Sections 4 and 5, the performance of the protocol and several comparable ones is analyzed and simulated. Practical issues and tradeoffs are discussed in Section 6.

1.1. Related work

Centralized protocols are widely used to manage the group key of a group with a server, which handles requests from all members, and is responsible for all rekeying. The server can choose a specialized cryptographic algorithm, such as [46], or it can use hierarchical structures with symmetric ciphers to share the group key securely. Wong et al. [44] proposed key graphs where keys are arranged into a hierarchy, and the key server maintains all the keys.

The key tree is particularly attractive since each member only holds the keys along the path, and the joining and leaving operations only affect the keys along the path. A similar hierarchical arrangement is also used by LKH [12]. The centralized tree key management model of VersaKey framework by Waldvogel et al. [41] also uses binary key trees to manage group keys. Keystone [45] uses forward error correction (FEC) to provide reliable group key delivery.

Li et al. [26] gave a detailed discussion about batch rekeying for LKH. To optimize the performance of rekeying multicast, Yang and Li designed a R-BFS algorithm to take advantage of the cluster property of keys in multicast packets, and improve the performance of FEC multicast transmission [48]. In [50], a protocol supporting efficient packet encoding scheme was proposed, and parity packets are added to provide proactive error correction.

Key tree balancing can be used to improve the performance of tree-based group key management protocol. Moyer et al. [31] proposed techniques to keep a centralized binary key tree balanced. They can either use a modified leaf deletion operation or a periodic rebalancing approach. Snoeyink et al. [37] showed that an optimal key distribution tree for a centralized key server with simple private key encryption is a special case of 2–3 tree. Goshi and Ladner [11] also proposed algorithms to height-balance or weight-balance the 2–3 key trees. Ng et al. [32] proposed merging algorithms to balance key trees.

Sherman and McGrew [2,36] proposed one-way function trees for efficient key generation. Their work is subject to a particular kind of collusion attack [13], and an improvement is done by Ku in [22]. ELK [34] uses key derivations based on pseudo-random functions, and the authors also implement recovery mechanisms in their algorithms.

The distributed schemes are often based on the two-party Diffie-Hellman (2DH) key agreement protocol or its extensions. Kim et al. [21] propose a tree based group Diffie-Hellman protocol (TGDH). Distributed group key management protocol proposed in [28] also uses an extended form of Diffie-Hellman algorithm. Application class awareness is introduced in [20] for contributory group key management schemes. Yu, Sun, and Liu [49] proposed a method that utilizes empty positions in key trees to optimize the performance of contributory key trees.

Iolus [30] divides the group to subgroups to avoid 1-affect-n and 1-does-not-equal-n problems occurring in the single group key schemes, such as LKH. The dual encryption protocol proposed by Dondeti et al. uses hierarchical subgroups to achieve scalability [7]. Mykil [15] divides a multicast group to areas, and there is a controller in every area. A controller runs LKH to manage the area key for all members in its area. SAKM [3] also makes the multicast group a hierarchy of subgroups. Subgroup merge and split mechanisms are provided, so that subgroups can form cluster dynamically to get the best performance.

2. Key tree and group key rekeying

We overview the basic concepts of group key management, key tree [44], and rekeying operations of centralized secure group communication systems.

2.1. Key tree

A key tree is a hierarchical arrangement of a set of keys. Nodes of a key tree are called k-nodes. Auxiliary keys are assigned to internal k-nodes, and individual keys to external k-nodes. Each external k-node is associated with a unique u-node,

which represents a particular member in the group. An individual key is only shared by the server and the corresponding member, and an auxiliary key is shared by members that belong to the subtree rooted at the k-node of the auxiliary key. An individual key is established at the time a member joins the group, and it remains valid until the member leaves. On the other hand, auxiliary keys are frequently updated and sent to members to keep group communication secrecy. The root k-node stores the group key, which is a special auxiliary key shared by all members.

The requests of join and leave operations have to be authenticated by the server to ensure that they are coming from the user herself. To authenticate leave requests is simpler, since the leaving member can use the individual key to produce a message authentication code, MAC [35], of its leave request. The server receives the leave requests with correct MACs can be sure it is sent by the owners of the individual keys. To authenticate join requests, the server and each user must have pre-established knowledge about each other. One widely used approach is public-key authentication [35], and the server can verify the signatures of users whose public-key certificates are trusted. ID-based authentication systems [5,16] can also be employed. The advantage of ID-based systems is that registered users use their identifiers as their public keys, and hence no public-key certificates have to be maintained and distributed. Password authentications [4,23] are suitable for systems that require users to enter passwords to login. The server can also put the individual keys of registered users in temper-resistant devices, such as Smart Card [25,43], which can help users perform the authentication operations. In the rest of the paper, we assume that all requests are authenticated, and put emphasis on how the keys can be efficiently and securely shared by members.

Fig. 1 shows typical key trees and the effects of join operations and leave operations. In the key tree, auxiliary keys are stored in internal k-nodes, and individual keys are stored in external k-nodes. In Sections 2 and 3, the key stored in k-node $x_{i,j}$ is usually denoted by $k_{i,j}$, where i is the level of the k-node in the key tree, and j is the index of the k-node relative to the leftmost k-node in the same level. The level number i and index number j are used to better explain protocols, algorithms, and examples in the paper. In practice, servers do not label the k-nodes this way. For example, binary numbers can be used to encode the positions of k-nodes in a binary key tree. When k-nodes are external, they store individual keys of members. An individual key stored in $x_{i,j}$ can also be denoted by k_m, if the k-node is associated with member u_m. The group key k_g can also be represented as $k_{1,1}$. In the figure, it is clear that to securely encrypt the new group key and send it to all members, the server only has to update the additional auxiliary keys on the path of the join or leave member. Therefore, key tree provides a scalable way to manage the group key for large dynamic groups.

2.2. Synchronous rekeying and asynchronous rekeying

Ideally, the server has to generate and send new keys to all members in response to every join or leave request immediately. The scheme is called synchronous rekeying, since the server updates the group key for every request without any delay. With asynchronous rekeying, the server only has to update the group key in a period of time, the rekeying interval, and it can handle multiple join and leave requests in the interval. Asynchronous rekeying can take advantage of the possible overlap of new keys for multiple requests, and thus has less communication and computation overhead.

In [26], Li et al. suggested that running asynchronous rekeying can alleviate the out-of-sync problem the synchronous rekeying scheme suffers. The out-of-sync problem between keys and data arises when a member receives a data encrypted by an old group key, or receives a data encrypted by a new group key that has not been received yet. With asynchronous rekeying, the server can make the rekeying interval long enough, so that all members will receive the rekeying message before the server performs the next rekeying operation. A member then only has to store at most one additional old group key. The drawback of asynchronous rekeying is that it will enlarge the vulnerability window, which is the period of time from the key server receiving a join or leave request to all members receiving the corresponding rekeying message.

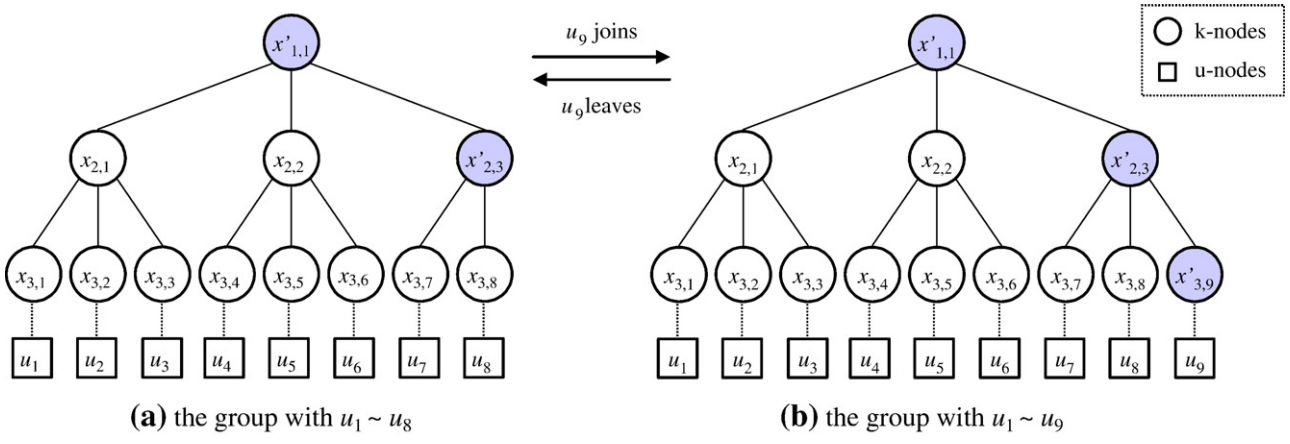

Fig. 1. Key tress (a) before u_9 joins (after u_9 leaves) (b) after u_9 joins (before u_9 leaves).

3. Shared key derivation (SKD) protocol

In this section, we show how existing cryptographic functions can be used as key derivation functions. Rekeying operations of the proposed shared key derivation protocol, SKD, are explained in detail.

3.1. Key derivation function

Key derivation function $f(\cdot)$ is used by the server and group members to generate new key values based on old keys, which are called the derivation keys. The basic idea behind shared key derivation is that, if a new key can be computed by users themselves, it does not have to be encrypted and sent by the server. Therefore, shared key derivation can effectively reduce the communication and computation overhead of rekeying. A good key derivation function should have the following two criteria.

Criterion 1. [One-way]

Given derivation key k, it is easy to compute f(k), but given f (k), it is computationally infeasible to compute k.

Criterion 2. [Pseudo-randomness]

Given keys k_1, k_2,..., k_m, if it is computationally infeasible to compute derivation key k, then it is computationally infeasible to compute f(k).

The derivation function should be one-way, so that the values of the derivation keys will not be computed. It should also act like a pseudo-random number generator, so that the new key values are unpredictable without knowing the derivation keys. To avoid producing repetitive key values with the same derivation key k, a non-zero salt value K can be used to compute

$$f(k \oplus K),$$

where K should be chosen so that it is computationally infeasible to deduce k from it. The security analysis of SKD is shown in the Appendix, and we proceed to show that hash functions, pseudo-random number generators, and one-way trapdoor functions can be used by key derivation. All these functions produce pseudo-random results which are indistinguishable and unpredictable from their inputs, so we focus on the discussion of selecting appropriate functions to satisfy Criterion 1.

3.1.1. Hash function

SHA-1 [8] and SHA-256 [9] are commonly-used secure hash functions which output 160-bit and 256-bit digests, respectively. For instance, we can use either one of

$$f(x) \equiv SHA\text{-}1(x) \bmod 2^{128},$$

$$f(x) \equiv SHA\text{-}256(x) \bmod 2^{128}$$

to derive 128-bit auxiliary keys. Recent research [42] has shown that collisions of SHA-1 can be found in reasonable time. However, we believe that SHA-1 is secure enough for SKD for two reasons. First, up to now, there is still no literature of how to find the pre-image x given SHA-1(x) under some specific constraints. In other words, it is still an open problem whether we can find an 128-bit x' given SHA-1(x), it is also not known how many such x' can be found. Second, the last 32 bits of the output of SHA-1(x) is not known if 128-bit keys are used, which will increase the uncertainty of finding x. To be conservative, we can choose the more computational intensive SHA-256, whose collisions are still computationally infeasible to be found.

3.1.2. Pseudo-random number generator

The ANSI X9.17 [35] random number generator is a widely adopted secure pseudo-random number generating function. The kernel of the function can use any kind of symmetric block cipher, such as AES [10]. Given pseudo-random number generator PRNG($\cdot$), we can construct the key derivation function with AES

$$f(x) \equiv \mathrm{PRNG}[\mathrm{AES}](x),$$

where x is used as the seed feeding the generator. This function is one-way because, given $f(x)$, finding x means the symmetric encryption algorithm, AES, has to be broken.

3.1.3. One-way trapdoor function

One-way trapdoor functions, which are used extensively in public-key cryptosystems, basing on hard mathematical problems, can also be used to construct the key derivation function. The discrete logarithm problem [35] is one good candidate. Given a large prime p and a generator g over GF[p], the 128-bit key derivation function can be

$$f(x) \equiv (g^x \bmod p) \bmod 2^{128}.$$

When we choose a 512-bit prime, it is computationally infeasible to compute the exponent given $f(x)$. One drawback of these trapdoor functions is that they are considerably slower than hash functions or symmetric encryption functions.

3.2. Synchronous rekeying operations

The synchronous rekeying scheme supports single join operation and single leave operation. The server updates keys according to the join or leave request of a member, and does not process another request before it finishes the rekeying operation. In the rest of the section, a key tree (or subtree) whose root k-node is $x_{i,j}$ is denoted by $y_{i,j}$, and key $k_{i,j}$ of $x_{i,j}$ is also called the key of $y_{i,j}$. The notation $[k']_k$ denotes the cipher text generated by encrypting key k' with key k, and it is used throughout the paper.

3.2.1. Single join operation

Each time a new member wishes to join the group, she sends a join request to the server. After receiving the request, the server verifies it, authenticates the identity of the new user, randomly generates a new individual key for the member, and sends it via a secure channel. The server then assigns a new u-node to represent the new member, and creates a new k-node to hold the new

individual key. The k-node is inserted in the end of one of the shortest paths of the key tree.

After the new k-node is inserted into the key tree, all auxiliary keys of the internal k-nodes on the path from the k-node of the new member to the root must be updated. Fig. 2 shows how these new keys are computed when a new user u joins. Assume that x_{h,p_h}, the root of subtree y_{h,p_h}, is the last internal k-node on the join path. When x_{h,p_h} is not full, the new external k-node, $x'_{h+1,p_{h+1}}$, which represents u, will be placed under it (Fig. 2(a)). If the last k-node is full, then a new internal k-node, $x'_{h+1,p_{h+1}}$, which replaces the position of old k-node $x'_{h+2,s_{h+2}}$, is created to be the parent of $x'_{h+2,s_{h+2}}$ and the new external k-node, $x'_{h+2,p_{h+2}}$, which represents u (Fig. 2(b)), where $x'_{h+2,s_{h+2}}=x_{h+1,s_{h+1}}$ is a child external k-node of x_{h,p_h}. In both scenarios, the new auxiliary key of k-nodes x'_{i,p_i}, $1\le i\le h$, can be computed by

$$k'_{i,p_i}=f\left(k_{i,p_i}\right),$$

where k_{i,p_i} is the derivation key. In scenario (b), the auxiliary key of the newly created internal k-node $x'_{h+1,p_{h+1}}$ can be computed by

$$k'_{h+1,p_{h+1}}=f\left(k_{h+2,s_{h+2}}\oplus k_g\right),$$

where $k_{h+2,s_{h+2}}=k_{h+1,s_{h+1}}$ is the derivation key. The salt value $k_g=k_{1,1}$ is to ensure that the derived key is different even the same derivation key is used, since k_g will be different each time. Since old members who own old derivation keys are able to derive corresponding new auxiliary keys, the server only has to notify them the information of the join path, and does not have to send the new keys to them. Since u does not know any auxiliary key on the join path, the server encrypts these new keys and sends them to u via multicast or unicast channels.

We use Fig. 1 to explain the join operation, where the server handles the join request of new member u_9. In the example, a new k-node $x'_{3,9}$ is created to hold u_9's individual key k_9. Auxiliary keys of k-nodes $x'_{1,1}$ and $x'_{2,3}$ on the path are to be updated. According to Fig. 2(a), new key values are $k'_{1,1}=f(k_{1,1})$ and $k'_{2,3}=f(k_{2,3})$. After the new keys are computed, the server continues to send the rekeying data to group members. The server only has to send the information about the position of the new k-node of the new member to notify all old members about the updated path. The server only encrypts the new keys for the new member who cannot derive the keys:

$$s\rightarrow u_9:\left[k'_{1,1}\right]_{k'_{2,3}}||\left[k'_{2,3}\right]_{k_9}.$$

3.2.2. Single leave operation

When a member wishes to leave the group, she sends a leave request to the server and then leaves. After receiving the request, the server has to verify it to ensure that it really comes from the member. If the verification passes, the server can continue to finish the leave operation. A member is also possible to be expelled from the group by the server because of network failure, member's misbehavior, or other reasons. The server then removes the u-node of the member and the k-node associated with the member from the key tree.

After the k-node of the leave member is removed from the key tree, all auxiliary keys on the path from the position of the removed k-node to the root have to be updated. Fig. 3 shows the shared key derivation method used in this operation when user u leaves. Assume that x_{h,p_h}, the root of subtree y_{h,p_h}, is the last internal k-node on the leave path. After the removal of the external k-node, $x_{h+1,p_{h+1}}$, of u, if x_{h,p_h} has more than two children, the old external node x_{h,p_h} will become x'_{h,p_h} (Fig. 3(a)); otherwise, x_{h,p_h} will be replaced by the root k-node of the remaining child subtree $y_{h+1,s_{h+1}}$ and becomes x'_{h,p_h} (Fig. 3(b)). In scenario (b), no key derivation is needed and $k'_{h,p_h}=k'_{h+1,s_{h+1}}$. The new auxiliary key of x_{i,p_i}, either $1\le i\le h-1$ in scenario (a) or $1\le i\le h$ in scenario (b), can be computed by

$$k'_{i,p_i}=f\left(k_{i+1,s_{i+1}}\oplus k_{i,p_i}\right),$$

where $k_{i+1,s_{i+1}}$ is the derivation key. The old auxiliary key k_{i,p_i} is used as a salt value, and this will not cause security problem,

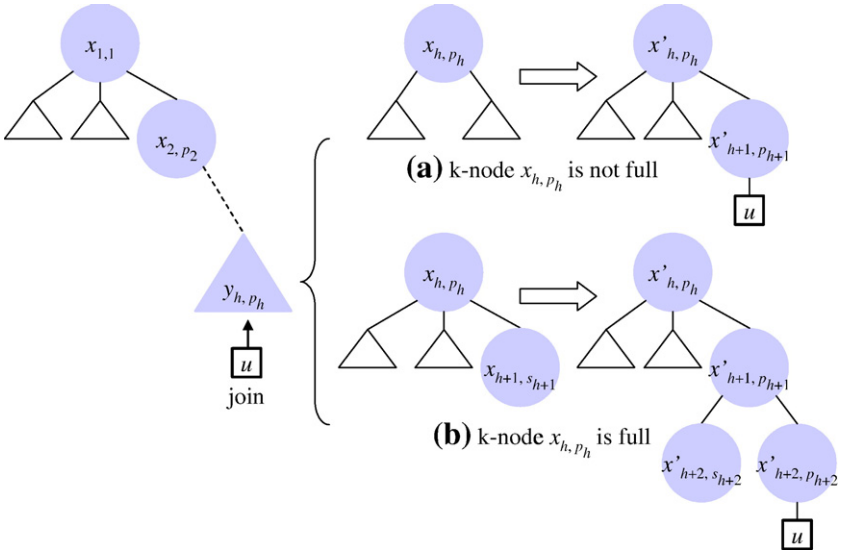

Fig. 2. Shared key derivation for the single join operation.

because only members knowing $k_{i+1,s_{i+1}}$ have enough information to compute the corresponding new auxiliary key. This derivation will only benefit members in the subtree $y_{i+1,s_{i+1}}$, and how to select the subtrees depends on the policy of the server. After all new keys are computed, they are encrypted and sent to all members through multicast channels.

We use Fig. 1 to demonstrate how the server handles the leave request of member u_9. After u_9 left the group, the associated k-node $x_{3,9}$ is no longer needed and is removed. The auxiliary keys of k-nodes $x'_{1,1}$ and $x'_{2,3}$ have to be updated. Suppose the keys of k-nodes $x_{2,1}$ and $x_{2,3}$ are chosen by the server to derive $k'_{1,1}$ and $k'_{2,3}$, respectively, it follows that $k'_{1,1} = f(k_{2,1} \oplus k_{1,1})$ and $k'_{2,3} = f(k_7 \oplus k_{2,3})$, where $k_{1,1}$ and $k_{2,3}$ are salt values. Unlike the join operation, not all old members can derive the new keys, and the server has to encrypt and send these keys to members who cannot compute them:

$$\begin{aligned} s \rightarrow u_4 - u_6 &: \left[k'_{1,1}\right]_{k_{2,2}} \\ s \rightarrow u_7 &: \left[k'_{1,1}\right]_{k'_{2,3}} \\ s \rightarrow u_8 &: \left[k'_{1,1}\right]_{k'_{2,3}} || \left[k'_{2,3}\right]_{k_8} \end{aligned} .$$

3.3. Asynchronous rekeying operations

Three operations, the join operation, the leave operation, and the batch update operation are supported. Among them only the batch update operation actually updates the group key. During the rekeying interval, the server queues all join and leave requests. In the end of the interval, the server updates the key tree with respect to all the requests, and performs a batch update operation to send new keys to all group members.

3.3.1. Join operation

This operation is similar to the single join operation, except that there is no key updates. The server verifies the join request is authentic, generates a new individual key, and securely sends it to the new member. The member is then queued to wait for the rekeying message by the next batch update operation. Since the member will not get the latest group key until the next batch update operation, the data transmitted during the waiting time cannot be decrypted. This waiting time can be shortened with techniques discussed in Section 6.

3.3.2. Leave operation

The conditions to trigger a leave operation are similar to those of a single leave operation. If the leave member is not a new member just joining the group in the same interval, the request is queued, and will be processed in the next batch update operation. Otherwise, the server ignores both the join and the leave requests of the member in the interval. This buffering effect of asynchronous rekeying will improve the performance of the system, since requests from members staying only short-term will be filtered out.

3.3.3. Batch update operation

In this operation, the server processes all queued join and leave requests in the interval, generates new keys, and sends them to group members. The server first removes all the u-nodes of leave members and the k-nodes containing their individual keys. Meanwhile, all k-nodes on the leave paths are marked LEAVE. If there are new members, their k-nodes are inserted in the end of one of the shortest paths chosen by the algorithm in Fig. 4. During the insertion process, all the old k-nodes on the join path are marked JOIN, and the newly-created internal k-nodes marked NEW. It should be noted that these flags are not mutually exclusive. For instance, if a k-node is marked JOIN and LEAVE, its new auxiliary key will be used by old and new members.

To compute the new auxiliary keys of the marked internal k-nodes, the server traverses the key tree bottom-up, and applies the key derivation algorithm shown in Fig. 5. The key derivation

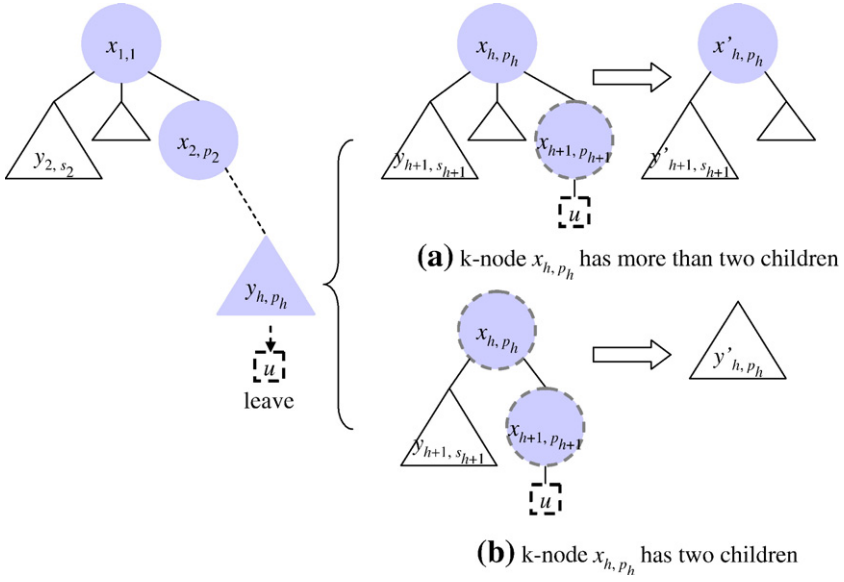

Fig. 3. Shared key derivation for the single leave operation.

```
x ← the root k-node
while x is internal
    mark x JOIN
    x_s ← the left most child k-node of x
    foreach child k-node x_c of x
      let h_c denote the shortest path length of subtree x_c
        and h_s the shortest path length of subtree x_s
      if (h_c < h_s) or ((h_c = h_s)
            and (x_c is marked LEAVE)
            and (x_s is not marked LEAVE))
        x_s ← x_c
    if x_s is external
      return x_s
    x ← x_s
```

Fig. 4. Path selection algorithm.

algorithm of the batch update operation is a generalization of the derivation methods of the single join operation and the single leave operation. The derivation method of Fig. 5(a) is analogous to that of Fig. 3(a), where x_{i,p_i} is on the paths of leaving members, the one of Fig. 5(b) to that of Fig. 2(a), where x_{i,p_i} is on the selected path of joining members, and the one of Fig. 5(c) to that of Fig. 2(b), where x_{i,p_i} is a newly created internal k-node on the join path. Note that according to the path selection algorithm, at most one JOIN path will be selected in any rekeying interval. If there are n new members, and the last internal k-node, x_{h,p_h}, on the join path is not full, all new members will be placed in a subtree, which will be a child of x_{h,p_h}. If x_{h,p_h} is full, all new members and a selected external k-node, $x_{h+1,s_{h+1}}$, of an old member will be put in a newly-created subtree, which will be inserted in the original position of $x_{h+1,s_{h+1}}$. The insertion procedure is a direct extension of that of the single join operation.

In scenario (a), k-node x_{i,p_i} is marked LEAVE, the JOIN mark is not important and is thus ignored. The server then selects subtree $y_{i+1,s_{i+1}}$, which has the most external k-nodes and is not marked JOIN, among the child subtrees of x_{i,p_i}, and computes the new auxiliary key

$$k'_{i,p_i} = f\left(k_{i+1,s_{i+1}} \oplus k_{i,p_i}\right).$$

The server will encrypt k'_{i,p_i} with the keys of the sibling subtrees of $y_{i+1,s_{i+1}}$, and send them by multicasts. In scenario (b), k-node x_{i,p_i} is marked JOIN but not NEW, so it is an old internal k-node. Its old auxiliary key value is used as the derivation key, and the server computes

$$k'_{i,p_i} = f\left(k_{i,p_i}\right).$$

The server will encrypt k'_{i,p_i} with the key of the child subtree in which the new members are placed, and send it by unicasts or multicasts. In scenario (c), k-node x_{i,p_i} is newly created. The key value of the root k-node of subtree $y_{i+1,s_{i+1}}$ is selected for key derivation with the following rules. If all child k-nodes of x_{i,p_i} are newly created, the subtree with the most external k-nodes is selected. The new auxiliary key can be computed by

$$k'_{i,p_i} = f\left(k_{i+1,s_{i+1}} \oplus K_1\right),$$

where $k_{i+1,s_{i+1}}$ is the derivation key and a non-zero constant, K_1, is the salt value. If the leftmost child subtree is an external k-node of an old member, the new auxiliary key is computed by

$$k'_{i,p_i} = f\left(k_{i+1,s_{i+1}} \oplus k_g\right),$$

where $k_{i+1,s_{i+1}}$, the individual key of the old member, is the derivation key and the old group key, k_g, is the salt value. The server will encrypt k'_{i,p_i} with the keys of the sibling subtrees of $y_{i+1,s_{i+1}}$, and send them by unicasts or multicasts.

We can safely use a non-zero constant K_1, because the derivation method of Fig. 5(c) will be applied at most once to any internal k-node when it is newly created. Since it is possible that the derivation key of scenarios (a) or (c) is itself a newly derived key, the server has to use a bottom-up key derivation order. It guarantees that the server will not reuse an old auxiliary key value known by a leave member, or use an undefined key value of a new k-node as a derivation key.

The algorithm can be demonstrated by the example in Fig. 6. Assume that, in the rekeying interval, member u_4 leaves, and members u_8 and u_9 join. K-node $x_{3,4}$ and u-node u_4 are removed, and by the path selection algorithm, old k-node $x_{3,7}=x_{2,3}$ and new ones $x'_{3,8}$, $x'_{3,9}$ become children of a new internal k-node $x'_{2,3}$, which is inserted in the original position of $x_{2,3}$. The new keys are $k'_{2,2} = f\left(k_{3,5} \oplus k_{2,2}\right)$ (scenario (b)), $k'_{2,3} = f\left(k_{3,7} \oplus k_g\right)$ (scenario (c)), and $k'_{1,1} = f\left(k_{2,1} \oplus k_{1,1}\right)$ (scenario (c)), where $k_{3,5}=k_5$, $k_{3,7}=k_7$, $k_{2,1}$ are chosen as derivation keys, and the old group key $k_g=k_{1,1}$ is used as the salt value to compute both $k'_{2,3}$ and $k'_{1,1}$. These new encrypted keys are sent to all members:

$$s \rightarrow u_5 : \left[k'_{1,1}\right]_{k'_{2,2}}$$

$$s \rightarrow u_6 : \left[k'_{1,1}\right]_{k'_{2,2}} || \left[k'_{2,2}\right]_{k_6}$$

$$s \rightarrow u_7 : \left[k'_{1,1}\right]_{k'_{2,3}}$$

$$s \rightarrow u_8 : \left[k'_{1,1}\right]_{k'_{2,3}} || \left[k'_{2,3}\right]_{k_8}$$

$$s \rightarrow u_9 : \left[k'_{1,1}\right]_{k'_{2,3}} || \left[k'_{2,3}\right]_{k_9}.$$

Comparing to three separated join operations and leave operations, the overlapped keys of k-nodes $x'_{1,1}$ and $x'_{2,3}$ are saved. Though the derivation method of Fig. 5(a) does not produce as much saving as that of Fig. 5(b), it is used more frequently in batch update operations, since there will usually be leave members in a rekeying interval, and a LEAVE mark has higher priority than a JOIN mark.

4. Communication, computation, and storage costs

Table 1 shows the communication, computation, and storage costs of SKDC (key star in [44]), LKH [12], OFT [36], ELK [34], and our protocol SKD. There are five rekeying operations: single join (SJ), single leave (SL), multiple join (MJ), multiple leave (ML), and batch update (BU). All the costs are measured in terms of the number of new keys generated or transmitted,

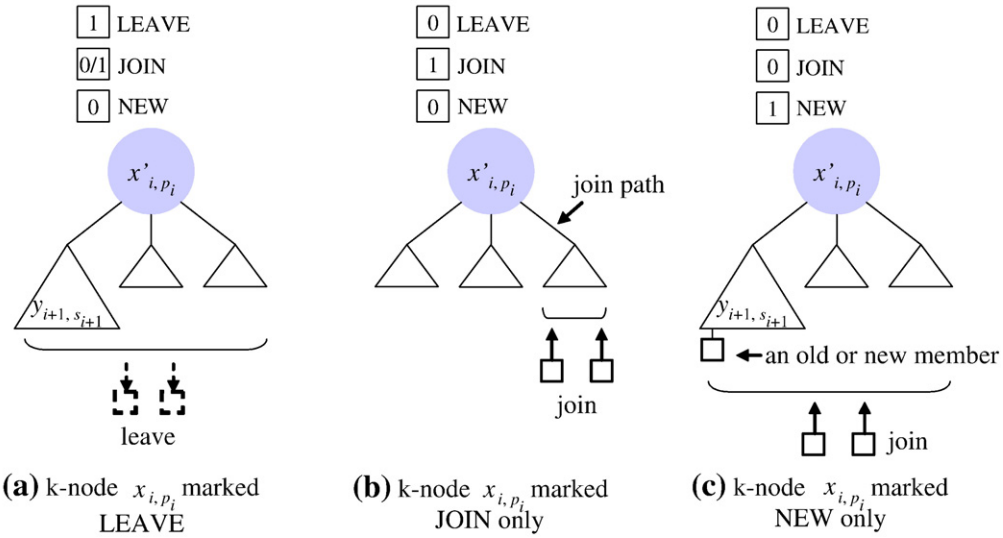

Fig. 5. Key derivation algorithm for the batch update operation.

which consume most of the computation power and the rekeying bandwidth. The one-way function of OFT and the pseudo-random function of ELK are considered equivalent to the key derivation function of SKD. For ELK, the hint size is set to zero, and the left and right child keys are of the same size. Multicast bandwidth and unicast bandwidth are compared separately. For asynchronous rekeying operations MJ, ML, and BU, the server can choose to send keys to new members via unicast or multicast channel. When the server uses multicast channels to send the rekeying messages to all new and old members, the communication costs are shown in *multicast all* rows.

In the table, n denotes the group size, j the number of join members, l the number of leave members. The length of a path of the key tree of LKH, OFT, ELK, SKD is h, in particular, h_2 for binary key trees, and h_d for d-ary key trees, and $h_d=[\log_d n]$ for balanced key trees. Values s_L, s_J, and s_N represent the number of k-nodes marked LEAVE, JOIN only, and NEW only, respectively. Constant K is the key size, and E, D, F, R denote the computation costs of encryption, decryption, key derivation, and random key generation, respectively.

If binary key trees are used, the communication overhead of SKD is the lowest, with the exception of ELK in join operations, since ELK does not have to send any information to members, while SKD still has to multicast join notifications with the information of join paths, but no keys. However, in join operations, the server running ELK will compute all auxiliary keys of the key tree, which consumes lots of computation power. If s_J and j are set to zero, the costs of BU and MJ can be compared. Likewise, if s_L and l are set to zero, the costs of BU and ML can be compared. SKD is still the best choice regarding to multicast communication costs (*multicast* and *multicast all*). LKH with 4-ary key trees requires the least bandwidth according to [26,44], but the bandwidth

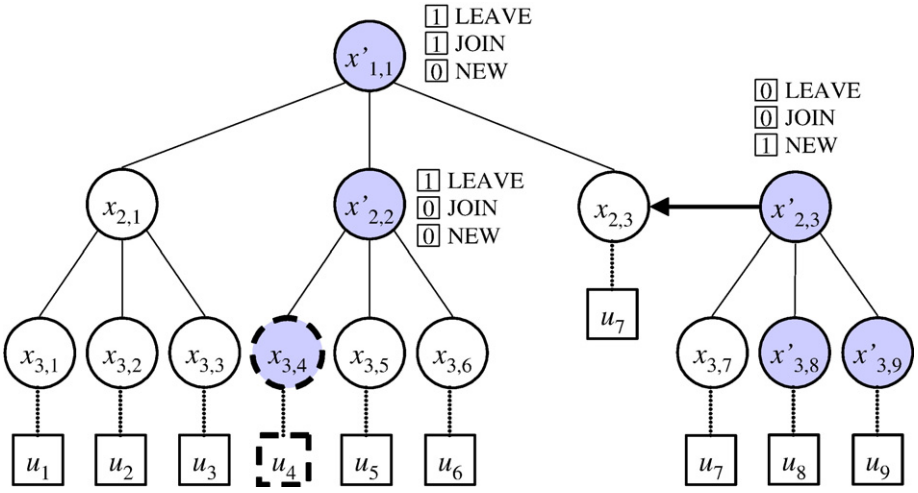

Fig. 6. A batch update example.

Table 1
Communication, computation, and storage costs of SKDC, LKH, OFT, ELK, and SKD

Protocol		SKDC	LKH	OFT	ELK	SKD
Communication costs						
SJ	Multicast	K	$2h_dK$	h_2K	0	h_2
	Unicast	K	h_dK	h_2K	h_2K	h_dK
SL	Multicast	$(n-1)K$	dh_dK	h_2K	$2h_2K$	$(d-1)h_dK$
MJ	Multicast	K	n.a.	$(h_2+1)K$	0	n.a.
	Unicast	jK		jh_2K	jh_2K	
	Multicast all	jK		$(2s_J+s_N+j+1)K$	$(s_J+s_N+j-1)K$	
ML	Multicast	$(n-l)K$	n.a.	$(s_L+l)K$	$2s_LK$	n.a.
BU	Multicast	$(n-l)K$	$(ds_L+2s_J)K$	n.a.	n.a.	$(d-1)s_LK$
	Unicast	jK	jh_dK			jh_dK
	Multicast all	$(n-l+j)K$	$(ds_L+2s_J+s_N+j)K$			$((d-1)s_L+s_J+j)K$
Computation costs						
SJ	Server	$2E+R$	$h_d(2E+R)$	$h_2(2E+2F)+2R$	$2(n+1)F+h_2E+R$	$h_d(E+F)+R$
	Old member	D	h_dD	$h_2(D+F)$	$2h_2F$	h_dF
	New member	D	h_dD	h_dD	h_2D	h_dD
SL	Server	$(n-1)E+R$	$h_d(dE+R)$	$h_2(E+2F)$	$h_2(2E+7F)$	$h_d((d-1)E+F)$
	Member	D	h_dD	$h_2(D+F)$	$h_2(D+4F)$	$h_d((d-1)D+F)/d$
MJ	Server	$(j+1)E+R$	n.a.	$(s_J+j)(2E+2F)+E+(j+1)R$	$2(n+j)F+(j-1)R+(s_J+2j-1)E$	n.a.
	Old member	D		$h_2(D+F)$	h_2F	
	New member	D		h_2D	h_2D	
ML	Server	$(n-l)E+R$	n.a.	$s_L(E+2F)+l(E+R)$	$s_L(2E+7F)$	n.a.
	Member	D		$h_2(D+F)$	$h_2(D+4F)$	
BU	Server	$(n+j-l)E+R$	$(ds_L+2s_J+s_N+j)E+(s_L+s_J+s_N)R$	n.a.	n.a.	$((d-1)s_L+s_J+j)E+(s_L+s_J+s_N)F$
	Old member	D	h_dD			$h_d((d-1)D+F)/d$
	New member	D	h_dD			h_dD
Storage costs						
	Server	$(n+1)K$	$(dn-1)K/(d-1)$	$(2n-1)K$	$(2n-1)K$	$(dn-1)K/(d-1)$
	Member	$2K$	h_dK	h_2K	h_2K	h_dK

reduction by shared key derivation makes binary or ternary key trees more attractive. In our simulation, we determine the best tree structure for SKD.

Considering the computation costs of LKH, OFT, ELK, and SKD, SKD uses about the same number of encryptions as OFT, about half the number of encryptions as ELK. SKD will likely have the lowest computation overhead for the server, since a key derivation replaces a random key generation and a key encryption. The space requirement of SKDC is the lowest, and those of LKH, OFT, ELK, and SKD are comparable.

5. Simulation

Protocols LKH, OFT, ELK, and SKD, are implemented in our simulation. Key trees of degree 2 to 5 are simulated for LKH and SKD, and a number is attached to the protocol name to indicate its degree. For example, SKD2 means that the server runs protocol SKD with binary key tree. OFT and ELK are designed specifically for binary key trees, so only binary key trees are used. Single join and single leave operations are used to evaluate the performance of synchronous rekeying. For asynchronous rekeying, batch update operations are used by LKH and SKD. For OFT or ELK, the server queues all join and leave requests in the interval, and then performs a multiple leave operation, if there are leave members, and a multiple join operation, if there are join members, in the end of the interval.

Session data collected by *Mlisten* in the MBone [1], which served as a testbed for the development of multicast protocols and group conference tools, are used in the simulation. The data came from the real audio sessions from November 18 to December 9, 1996. Three longest sessions are chosen, and the rekeying interval is fixed at 300 s.

5.1. Communication cost

Fig. 7 shows the simulation results of LKH, OFT, ELK, and SKD. The communication cost of a protocol is measured by the total number of keys transmitted in the session, and it is normalized to the number of keys unicast to new members by the server performing single join operations of LKH2. For all synchronous and asynchronous rekeying operations, the *join unicast*, *join multicast*, and *leave multicast* costs represent the unicast bandwidth overhead due to join requests, multicast bandwidth overhead due to join requests, and multicast bandwidth overhead due to leave requests, respectively. For asynchronous rekeying operations, we also evaluated the communication costs for multicast only scheme, where the server uses multicasts to deliver new keys to all new and old members. The *join multicast all*, and *leave multicast all* costs

represent the multicast bandwidth overhead due to join requests, and multicast bandwidth overhead due to leave requests, respectively.

We use an example to show the communication costs of the asynchronous rekeying operation of Fig. 6. When both unicasts and multicasts are used, the transmitted keys of *join unicast* are $[k'_{1,1}]_{k'_{2,3}}$, $[k'_{2,3}]_{k_8}$ (u_8), and $[k'_{1,1}]_{k'_{2,3}}$, $[k'_{2,3}]_{k_9}$ (u_9); those of *leave multicast* are $[k'_{1,1}]_{k'_{2,2}}$, $[k'_{1,1}]_{k'_{2,3}}$, and $[k'_{2,2}]_{k_6}$ (u_5, ..., u_9). When only multicasts are used to send keys to all members, the transmitted keys of *join multicast all* are $[k'_{2,3}]_{k_8}$ and $[k'_{2,3}]_{k_9}$; those of *leave multicast all* are $[k'_{1,1}]_{k'_{2,2}}$, $[k'_{1,1}]_{k'_{2,3}}$, and $[k'_{2,2}]_{k_6}$. In both situations, the number of new keys of *join multicast* is always zero; the number of new keys on the paths of leaving members will not be affected by how the server transmits the keys to new members, so the costs of *leave multicast* and *leave multicast all* are the same.

Though all sessions have different average membership duration and average inter-arrival time, the simulated results are quite consistent for most protocols. The unicast communication bandwidth is used by single join operations. The cost is mainly determined by the lengths of join paths of new members, and is thus affected by the degree of the key tree. The results show that key trees of higher degrees will have lower unicast costs, because, with the same number of external k-nodes, they will have a shorter average path length in general.

First of all, we compare the communication costs of LKH and SKD with key trees of degrees 2 to 5. The following results can be obtained by inspecting Fig. 7:

a) Comparing the costs of *join unicast*, the number of keys sent to new members decreases from 100% to 46% of that of LKH2 with synchronous rekeying, and from 92% to 46% of that of LKH2 with asynchronous rekeying. The cost of *join unicast* decreases, because the number of keys sent to new members is dominated by the lengths of the join paths, which are shorter when the degrees of the key trees are higher.
b) Comparing the costs of *leave multicast*, which is equivalent to *leave multicast all*, the number of keys sent to old members increases from 50% to 97% of that of LKH2 with synchronous rekeying, and from 50% to 88% of that of LKH2 with synchronous LKH2 with asynchronous rekeying. The cost of *leave multicast* increases with the degree, because, for any newly derived auxiliary key k of k-node x, the server has to encrypt k with individual keys or auxiliary keys of all child subtrees of x except the one whose key is the derivation key.
c) With asynchronous rekeying, the server can send the encrypted keys which are planned to be sent to new members by unicasts (*join unicast*), and those planned to be sent to old members by multicasts (*leave multicast*) together by multicasts (*join multicast all*+*leave multicast all*). The cost of *join multicast all* decreases from 51% to 26% of that of LKH2 with asynchronous rekeying. However, the decrease cannot balance the increasing overhead of the cost of updating the auxiliary keys due to leaving members (*leave multicast all*).

From results (a) and (b), comparing to higher degree key trees, binary key trees result in lower multicast communication overhead (*leave multicast*) and higher unicast communication overhead (*join unicast*). Binary key trees are the better choice, since often we will sacrifice unicast performance for better multicast performance due to the much higher overhead of multicasts. From results (b) and (c), it is very clear that, if the server uses multicasts to send encrypted keys to all members (*join multicast all*+*leave multicast all*), binary key trees are better than the higher degree ones. No matter which degree we choose, SKD performs better than its LKH counterpart. Therefore, SKD2 will be the best choice with the lowest communication overhead, which conforms with the expected communication costs listed in Table 1.

Since the best configuration of SKD and LKH is SKD2, we further compare its communication costs with those of OFT and ELK, which both use binary key trees, and the communication cost of LKH2 is used as the referenced benchmark. The following results can be obtained by inspecting Fig. 7:

d) Comparing the costs of *join unicast*, the numbers of keys sent to new members are the same for all protocols with synchronous rekeying, and SKD2 achieves the most reduction, approximately 7% of the number of LKH2 with asynchronous rekeying. With asynchronous rekeying, SKD2 can use the derivation method of 4(c) to further reduce the number of keys when there are multiple new members in a rekeying interval, but the performance improvement is minute.
e) Comparing the cost of *leave multicast*, which is equivalent to *leave multicast all*, SKD2 and OFT both reduce 50% of the communication cost of LKH2 with synchronous rekeying, while ELK performs the same as LKH2. SKD2, OFT, and ELK lower the communication costs to 50%, 55%, 99% of that of LKH2 with asynchronous rekeying. OFT has to use about 5% more keys due to leaving members, since it has to randomly generate new individual keys for old members who are the siblings of the leaving ones. Additionally, new auxiliary keys generated due to adding new members also have to be multicast to old members (*join multicast*), which dramatically increases the multicast communication overhead.
f) With asynchronous rekeying, when the encrypted keys needed by new members (*join multicast all*) are sent with those needed by old members (*leave multicast all*) by multicasts, SKD2 and ELK reduce the costs of *join multicast all* to 50% and 85%, respectively, of that of LKH2, but OFT sends 86% more keys than LKH2 does. Comparing the costs of sending keys with multicasts only (*join multicast all*+*leave multicast all*), SKD2, ELK and OFT lower the numbers of transmitted keys to 50%, 93%, and 93%, respectively, of that of LKH2.

From results (d), (e), and (f), ELK has good performance in updating keys generated due to new members, but does not do well in updating keys generated due to leaving members. The results of OFT are the exact opposite of those of ELK. Therefore, SKD2 is obviously the best choice.

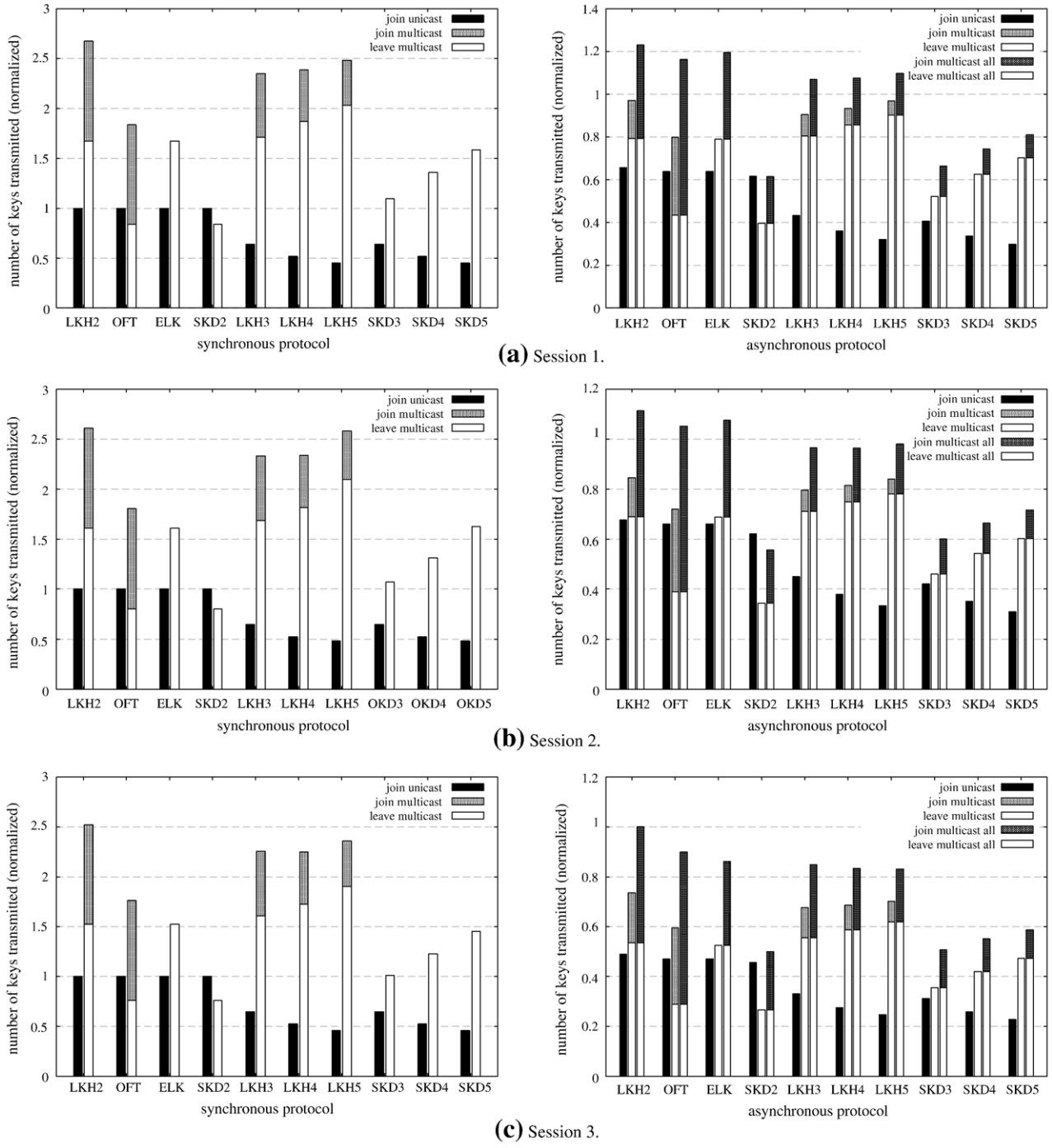

Fig. 7. Communication costs of LKH, OFT, ELK, and SKD.

Depending on different sessions and protocols, the bandwidth reduction of batch rekeying can be more than 70%. It is found that, for all three sessions, the buffering of asynchronous rekeying filters out about 41% to 58% of the requests. More multicast bandwidth is saved by overlapping join paths with leave paths. The actual distribution of join events and leave events will determine how these two factors affect the total bandwidth reduction. Therefore, among the four protocols, SKD2 is better than LKH2, OFT, and ELK, and the performance gain is greater with asynchronous rekeying.

Comparing the bandwidth reduction of asynchronous rekeying over synchronous rekeying, it can be observed that LKH and SKD provide greater reduction on join multicast costs than OFT and ELK, since LKH and SKD both directly support batch update operations, which provide more reduction than multiple join and multiple leave operations. The *join multicast* and *join multicast all* costs show that batch update operations

Table 2
Computation cost of LKH, OFT, ELK, SKD

		LKH2	OFT	ELK	SKD2	LKH3	LKH4	LKH5	SKD3	SKD4	SKD5
Session 1											
Synchronous	Random key generation	1.000	0.173	0.086	0.000	0.683	0.560	0.489	0.000	0.000	0.000
	Key derivation	0.000	2.000	29.555	1.000	0.000	0.000	0.000	0.683	0.560	0.489
	Key encryption	2.000	1.544	1.456	1.000	1.626	1.582	1.597	0.943	1.022	1.109
Asynchronous	Random key generation	0.335	0.074	0.000	0.000	0.221	0.180	0.156	0.000	0.000	0.000
	Key derivation	0.000	0.836	9.967	0.335	0.000	0.000	0.000	0.221	0.180	0.156
	Key encryption	0.670	0.633	0.650	0.335	0.583	0.585	0.597	0.361	0.405	0.441
Session 2											
Synchronous	Random key generation	1.000	0.207	0.103	0.000	0.699	0.572	0.530	0.000	0.000	0.000
	Key derivation	0.000	2.000	19.860	1.000	0.000	0.000	0.000	0.699	0.572	0.530
	Key encryption	2.000	1.554	1.446	1.000	1.651	1.587	1.698	0.952	1.016	1.168
Asynchronous	Random key generation	0.309	0.078	0.000	0.000	0.202	0.166	0.145	0.000	0.000	0.000
	Key derivation	0.000	0.758	5.991	0.309	0.000	0.000	0.000	0.202	0.166	0.145
	Key encryption	0.617	0.583	0.596	0.309	0.535	0.534	0.543	0.333	0.368	0.397
Session 3											
Synchronous	Random key generation	1.000	0.263	0.132	0.000	0.708	0.582	0.516	0.000	0.000	0.000
	Key derivation	0.000	2.000	13.899	1.000	0.000	0.000	0.000	0.708	0.582	0.516
	Key encryption	2.000	1.568	1.432	1.000	1.650	1.577	1.601	0.943	0.994	1.086
Asynchronous	Random key generation	0.284	0.085	0.000	0.000	0.194	0.161	0.140	0.000	0.000	0.000
	Key derivation	0.000	0.652	3.980	0.284	0.000	0.000	0.000	0.194	0.161	0.140
	Key encryption	0.568	0.510	0.490	0.284	0.482	0.474	0.473	0.289	0.313	0.333

generate less new keys than single join operations, since the join paths overlap the leave paths, and the new keys on the overlapped paths are already counted as the *leave multicast* costs.

5.2. Computation cost

The numbers of random key generations, key derivations, and key encryptions are used to measure the computation costs of LKH, OFT, ELK, and SKD, and the results are listed in Table 2. All these numbers are normalized to the number of random keys generated by the server running LKH2 with synchronous rekeying.

From the simulation results, it can be verified that random key generations of LKH are totally replaced by key derivations of SKD. The decryption counts of SKD are less than those of LKH, and the differences are exactly their respective key derivation counts of SKD. SKD and LKH have the lowest computation overhead when binary key trees and 4-ary key trees are used respectively.

The server running OFT has to encrypt new keys for joins and leaves, but it can save about half of the number of encryptions needed by a leave operation of LKH2. The server running ELK needs fewer number of encryptions because in join operations, the server only has to encrypt new keys for new members. Comparing to OFT, SKD uses fewer encryptions in join operations, and to ELK, SKD uses fewer encryptions in leave operations. Therefore, regarding the encryption costs, OFT and ELK are better than LKH, but worse than SKD. Since the server running ELK has to recompute all keys of the key tree for join requests, it will do an excessive amount of computation, especially when the group becomes larger. Whether the extra computation of ELK is justifiable depends on the applications, but we believe that the use of asynchronous rekeying operations makes ELK's approach less attractive.

We use the software implementation of random number generators, hash functions, and symmetric ciphers of cryptographic library Crypto+ [6] to evaluate the computation time of these functions. AES [10] is chosen as the encryption and decryption algorithms, as it supports both 128-bit and 256-bit keys. We use the ANSI X9.17 [35] random number generator with block cipher AES. SHA-1 [8] and SHA-256 [9] are used as the key derivation functions for 128-bit and 256-bit keys, respectively. Table 3 lists the execution time of each function run on a 2.4 GHz Pentium 4 PC.

The total computation time of a protocol is estimated by $n_R R + n_F F + n_E E$, where n_R, n_F, and n_E are the numbers of random key generations, key derivations, and key encryptions performed by the server, respectively. Table 4 shows the average computation time, which is normalized to that of LKH2 with 128-bit keys, of three sessions of each protocol. As expected, with 256-bit keys, the server approximately needs twice as much computation power as the one with 128-bit keys, and the asynchronous rekeying protocols are all better than their synchronous versions. Though key trees of higher degrees

Table 3
Execution time (Seconds/1,000,000 computations)

Key size (bit)	128	256
R: random key generation (X9.17 with AES)	1.656	3.218
F: key derivation (SHA-1/SHA-256)	1.140	1.907
E: key encryption (AES)	0.516	0.953
D: key decryption (AES)	0.797	1.437

Table 4
Estimated server computation time

Protocol		LKH2	OFT	ELK	SKD2	LKH3	LKH4	LKH5	SKD3	SKD4	SKD5
128-bit key	Synchronous	1.00	1.28	9.29	0.62	0.74	0.66	0.63	0.48	0.44	0.43
	Asynchronous	0.31	0.48	2.93	0.19	0.23	0.21	0.19	0.15	0.14	0.14
256-bit key	Synchronous	1.91	2.23	15.6	1.06	1.42	1.24	1.19	0.83	0.76	0.76
	Asynchronous	0.60	0.83	4.92	0.33	0.43	0.39	0.37	0.26	0.25	0.24

will need less computation power, they will require more rekeying bandwidth. Therefore, SKD2 should be chosen. SKD2 has the lowest communication cost, and it is faster than LKH2, OFT, and ELK that use binary key trees.

6. Practical issues and tradeoffs

Practical issues, such as reliable rekeying, and tradeoffs, including rekeying delay and key size, are discussed in this section.

6.1. Reliable multicast rekeying

Reliable multicast with forward error correction (FEC) has been studied extensively [33,24]. Though more bandwidth is consumed to transmit additional FEC packets, receivers are more likely to be able to correct errors, reconstruct the original message by receiving enough packets, and avoid asking the source to retransmit. This approach works well for a large amount of receivers. Recently, reliable multicast is used to provide reliable rekeying service in [48] and [50]. SKD can take advantage of the reliable multicast rekeying techniques to further improve the performance of group key rekeying.

6.2. Rekeying delay

One problem of asynchronous rekeying is the delayed rekeying of join requests. A new member may wait for the entire rekeying interval before the new group key is received. The delay is particularly unacceptable when the rekeying interval is too long. The improved scheme of LKH, the LKH+ [12], uses a suitable function to compute the new group key whenever a new member joins. The server then sends the new key to new member, and multicasts a notification message to other members so that they can compute the new group key themselves. OFT+ also uses a similar scheme.

The method shown in Fig. 8 is designed for SKD to shorten the rekeying delay. Suppose the rekeying interval is divided into n sub-intervals, and the latest group key is k_g, which is available to all members. Group key k_g is not used directly, but it is used to derive the temporary group key

$$k_i = f^i(k_g)$$

of each sub-interval $i=1,\ldots, n$. The server and all members are able to compute these keys. If a new member joins the group in some sub-interval, the temporary group key is unicast to the new member. The new member can compute the temporary group keys of later sub-intervals by itself. If there are members leaving the group, the derivation algorithm in Fig. 5 will be used. Otherwise, the next batch update operation will compute a new key

$$k_g' = f(k_g \oplus K_2),$$

where K_2 is a predefined non-zero constant. The minor modification ensures that new members without k_g cannot compute the new key k_g' from any $f^i(k_g)$. Even if a new member leaves before the end of the rekeying interval, k_g' is not derivable and forward group key secrecy is preserved.

6.3. Key size

The protocol can use keys shorter than the default key size of the chosen encryption algorithm. Shortened keys are stored, encrypted, or transmitted, and the corresponding encryption keys are generated on-the-fly. Extension function $g(\cdot)$ is used to transform a shortened key k to an encryption key

$$k' = g(k),$$

where $|k|<|k'|$. This approach basically trades off computation power for storage and bandwidth. However, the system becomes less secure due to the smaller effective key size.

7. Conclusion

In this paper, a group key management protocol based on novel shared key derivation methods is proposed to reduce the

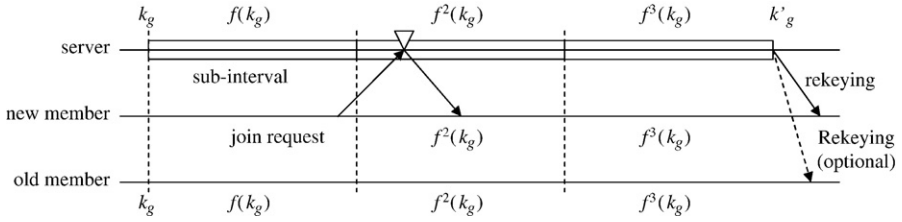

Fig. 8. Sub-intervals and group key derivations for new and old members.

communication and computation overhead of centralized secure group communication systems with key trees. With shared key derivation, the server does not have to encrypt and transmit new keys to members who have enough information to derive the keys by themselves. With strong encryption function and key derivation function, protocol SKD is provably secure. It is shown in our analysis and simulation that, comparing to LKH, OFT, and ELK, SKD requires the least communication bandwidth and computation power, and the protocol is most efficient with binary key trees and asynchronous rekeying. The protocol supports synchronous and asynchronous rekeying operations, and, with minor modification, the rekeying delay and the key size of the protocol can be tuned to meet different system needs.

In many network applications, often a source has to send data to many receivers. IP multicasts and application-layer multicasts provide efficient and scalable one-to-many or many-to-many communications. The proposed group key management protocol is designed for multiple users to share a common secret key securely and efficiently, and can be easily adopted by large-scaled network applications requiring secure data distribution, such as distant learning, audio broadcasting, video streaming, and online gaming. To support massive-scaled communications, to increase fault-tolerance, to bridge the gaps between networks separated by firewalls, or to promote fairness in peer-to-peer environments, users might have to form interconnected or hierarchical subgroups. Every subgroup can assign its own key server, which can be a dedicated server, or a peer, to manage the shared subgroup key with protocol SKD, and then collaborates with other subgroups to provide secure group communications.

Acknowledgment

The authors would like to thank Professor K. C. Almeroth for providing the audio session data collected in the MBone.

Appendix

Security analysis

In the analysis, we will prove that the proposed protocol is immune to collusion attacks, and provides backward and forward group key secrecy. There are two ways for a member to get a new auxiliary key. They can either decrypt the encrypted key in a rekeying message sent by the server, or compute the new key using key derivation functions.

If the server sends an encrypted key $[k']_k$, a decryption dependency $k \prec_0 k'$ is created, since members knowing key k will be able to decrypt key k'. However, it is assumed that adversaries can get all encrypted keys, which are transmitted by the server over insecure unicast or multicast channels. Therefore, only the decryption key k is crucial in our analysis.

If key k' is derived with derivation key k, it creates a derivation dependency $k \prec_0 k'$. Though salt values are used in key derivations to avoid producing repetitive key values, they do not introduce additional dependencies, since, according to the protocol, a member knowing a derivation key must also know the salt values, which are old keys or constants.

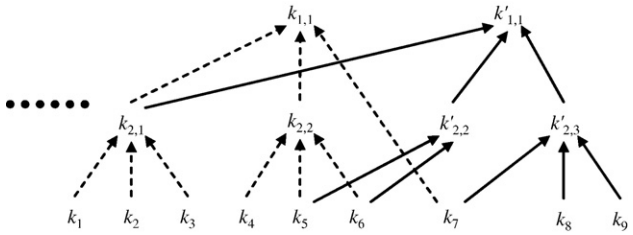

Fig. 9. A key dependency graph example.

Key dependency graphs extend the concept of encryption graphs of [47], such that all individual keys and auxiliary keys are modeled as nodes, and all key dependencies are modeled as directed edges. In the following discussion, we implicitly assume that all keys and dependencies are generated by the server or users running protocol SKD. In these graphs, we do not distinguish derivation dependencies and decryption dependencies, since both kinds of dependencies are treated equally by the lemmas and theorems.

Definition 3. (Rekeying dependency graph)

A rekeying dependency graph G_n, $n \geq 0$, is a key dependency graph with vertices $V(G_n)$ containing all the new individual keys, the derivation keys, the decryption keys, and the newly derived keys in the nth rekeying operation (single join, single leave, or batch update), and edges $E(G_n) = \{(k,k') | k \prec_0 k' \wedge k,k' \in V(G_n)\}$.

In the rest of the analysis, it is assumed that the server does not have pre-initialized individual keys and auxiliary keys, so $G_0 = \phi$.

Definition 4. (Rekeying dependency history graph)

A rekeying dependency history graph H_n, $n \geq 0$, is defined by $H_n = \cup_{i=0}^{n} G_i$.

Definition 5. (Key owner set)

Key owner set S of key k is the set of users who knows k by the protocol.

Definition 6. (Ascendant key)

Key k is said to be an ascendant key of k', denoted by $k \prec k'$, if and only if there are m keys, $m \geq 0$, such that

$$k \prec_0 k_1 \prec_0 \cdots \prec_0 k_m \prec_0 k'.$$

When $m = 0$, k is an immediate ascendant key of k'.

Fig. 9 gives a key dependency graph example according to the batch update example in Fig. 6. Assuming that the batch update operation is the *n*th rekeying operation. The dashed arrows represent a subset of the dependencies of H_{n-1}. The solid arrows represent all dependencies created by the batch update operation. Then, $V(G_n) = \{k_5, k_6, k_7, k_8, k_9, k_{2,1}, k'_{2,2}, k'_{2,3}, k'_{1,1}\}$, $E(G_n) = \{(k_5,k'_{2,2}), (k_6,k'_{2,2}), (k_7,k'_{2,3}), (k_8,k'_{2,3}), (k_9,k'_{2,3}), (k_{2,1},k'_{1,1}), (k'_{2,2},k'_{1,1}), (k'_{2,3},k'_{1,1})\}$. In the example, $k_5 \prec k'_{1,1}$, and key owner sets S_5 of k_5 and $S'_{2,2}$ of $k'_{2,2}$ are $\{u_5\}$ and $\{u_5,u_6\}$, respectively.

Since the single join operation and the single leave operation can be viewed as special cases of the batch update operation, we only have to discuss the three derivation methods in Fig. 5. Both

derivation methods of Fig. 5(a) and (c) produce one derivation dependency, $k_{i+1,s_{i+1}} \prec_0 k'_{i,p_i}$, and $m-1$ decryption dependencies, $k_{i+1,t_j} \prec_0 k'_{i,p_i}$, where m is the number of children of x'_{i,p_i} and t_j, $0<j<m$, is the index of sibling key tree of $y_{i+1,s_{i+1}}$. The derivation method of Fig. 5(b) produces one derivation dependency, $k_{i,p_i} \prec_0 k'_{i,p_i}$, and one decryption dependency, $k_{i+1,j} \prec_0 k'_{i,p_i}$, where j is the index of the child subtree of x'_{i,p_i} containing the new members. Generally speaking, given all immediate ascendant keys, $k_{a_1},\ldots, k_{a_m}$, of auxiliary key k', and their respective key owner sets $S_{a_1},\ldots, S_{a_m}$, S', we have

$$S' = \bigcup_{i=1}^{m} S_{a_i}, \tag{1}$$

and $S_{a_i} \cap S_{a_j} = \phi$ for all $0<i,j\le m$, $i\neq j$. Since every new auxiliary key has at least two immediate ascendant keys ($m\ge 2$) whose key owner sets are not empty, it follows that

$$S_{a_i} \subset S', \tag{2}$$

for all $0<i\le m$. For instance, in Fig. 9, since $S'_{2,2} = S_5 \cup S_6$ and $S'_{1,1} = S_{2,1} \cup S'_{2,2} \cup S'_{2,3}$, we find that $S_5 \subset S'_{2,2}$ and $S'_{2,2} \subset S'_{1,1}$.

Criterion 7. (Invariance of key owner set)

Given two rekeying dependency history graphs H_m, H_n, $m<n$, of the server running protocol SKD, and keys $k\in V(H_m)$, $k'\in V(H_n)$, and their respective key owner sets S, S', if $k=k'$, then $S=S'$.

Criterion 7 states that, for every key k, the users who know it are entirely determined by the time the key is created, and the key owner set S is final with respect to future rekeying operations. The key owner set of any key will not be changed after the key is created, which implies that Eq. (2) will always be true for all individual keys and auxiliary keys. The rekeying operations in Section 3 can be verified that (a) no members will derive an auxiliary key which has already been used by the others, and (b) the server will not encrypt an auxiliary key which has already been used by some members, and send it to them.

Lemma 8. *Given rekeying dependency history graph H_n, $n>0$, two keys k, $k'\in V(H_n)$, and their respective key owner sets S, S', if $k \prec k'$, then $S\subset S'$.*

Proof. If $k \prec_0 k'$, the lemma is true from Eq. (2). Otherwise, there exist keys $k_1,\ldots, k_m$ in $V(H_n)$, $m>0$, such that

$$k \prec_0 k_1 \prec_0 \cdots \prec_0 k_m \prec_0 k'.$$

From Eq. (2) and Criterion 7, it can be concluded that

$$S \subset S_1 \subset \cdots \subset S_m \subset S',$$

where S_i is the key owner set of k_i, for all $1\le i\le m$. □

Theorem 9. *Protocol SKD is immune to collusion attacks.*

Proof. To prove that our protocol is immune to collusion attacks, it only has to be shown that several members cannot use their combined knowledge to compute a key that they do not already know [29,38]. We prove the theorem by contradiction.

Assume that after n rekey operations, a set of malicious users, M, can successfully collude to find key $k'\in V(H_n)$ that they should not know according to the protocol. Since k' should not be known by the protocol, the key owner set S' should not include any user in M, that is,

$$M \cap S' = \phi. \tag{3}$$

To be able to compute k' without breaking the key derivation algorithm or the key encryption algorithm, at least one user $u\in M$ has to know some key $k\in V(H_n)$, such that $k \prec k'$. From Lemma 8,

$$\{u\} \in S \subset S', \tag{4}$$

where S is the key owner set of k. It follows that Eq. (4) contradicts Eq. (3), which completes the proof. □

A group key management protocol is said to provide backward group key secrecy, when a new member who joins the group in the nth rekeying operation cannot collude with users who do not know k_{g_i} to compute the past group keys generated in the ith, $i<n$, rekeying operation. The protocol is said to provide forward group key secrecy, when a member who leaves the group in the nth rekeying operation cannot collude with users who do not know k_{g_j} to compute the future group keys generated in the jth, $j\ge n$, rekeying operation. A protocol providing backward and forward group key secrecy ensures that users know the group keys only when they are members of the group.

Theorem 10. *Protocol SKD provides backward and forward group key secrecy.*

Proof. Let k_{g_i} be the group key generated in the ith rekeying operation, and u be a member joining the group in the nth rekeying operation, where $i<n$. Assume that, according to the protocol, user u and several other users who do not know k_{g_i} try to compute k_{g_i}. By Theorem 9, the collusion attack will fail. Therefore, protocol SKD provides backward group key secrecy.

Let k_{g_j} be the group key generated in the jth rekeying operation, and u be a member leaving the group in the nth rekeying operation, where $j\ge n$. Assume that, according to the protocol, user u and several other users who do not know k_{g_j} try to compute k_{g_j}. By Theorem 9, the collusion attack will fail. Therefore, protocol SKD provides forward group key secrecy. □

References

[1] K.C. Almeroth, M.H. Ammar, Collecting and modeling the join/leave behavior of multicast group members in the MBone, Proceedings of the Symposium on High Performance Distributed Computing, IEEE, 1996.

[2] D. Balenson, D. McGrew, A. Sherman, Internet-draft: key management for large dynamic groups: one-way function trees and amortized initialization, Internat Draft, IRTF, August 2000, http://www.securemulticast.org/smug3-balenson.pdf.

[3] Y. Challal, H. Bettahar, A. Bouabdallah, SAKM: a scalable and adaptive key management approach for multicast communications, SIGCOMM Computer Communications Review 34 (2) (2004) 55–70.

[4] T.-H. Chen, W.-B. Lee, G. Horng, Secure sas-like password authentication schemes, Computer Standards & Interfaces, Elsevier Science 27 (2004) 25–31.

[5] Y.F. Chung, K.H. Huang, F. Lai, T.S. Chen, Id-based digital signature scheme on the elliptic curve cryptosystem, Computer Standards & Interfaces, Elsevier Science 29 (2007) 601–604.

[6] W. Dai, Crypto++ library, URL: http://www.cryptopp.com/.

[7] L. Dondeti, A. Samai, S. Mukherjee, A dual encyrpion protocol for scalable secure multicasting, The Fourth IEEE Symposium on Computers and Communications, Red Sea, Egypt, 1999.

[8] FIPS 180-1, Secure Hash Standard (SHS) (April 1995).
[9] FIPS 180-2, Secure Hash Standard (SHS) (August 2002).
[10] FIPS 197, Advanced Encryption Standard (AES) (May 2002).
[11] J. Goshi, R.E. Ladner, Algorithms for dynamic multicast key distribution trees, Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing, ACM Press, Boston, Massachusetts, 2003.
[12] H. Harney, E. Harder, Logical key hierarchy protocol, Internat Draft, IETF, Expired in August 1999, April 1999, http://tools.ietf.org/html/draft-harney-sparta-lkhp-sec-00.
[13] G. Horng, Cryptanalysis of a key management scheme for secure multicast communications, IEICE Transcations on Communication E85-B (5) (2002) 1050–1051.
[14] M. Hosseini, D.T. Ahmed, S. Shirmohammadi, N.D. Georganas, A survey of application-layer multicast protocols, IEEE Communications Surveys & Tutorials 9 (3) (2007) 58–74.
[15] J.-H. Huang, S. Mishra, Mykil: a highly scalable key distribution protocol for large group multicast, IEEE 2003 Global Communications Conference (GLOBALCOM 2003), San Francisco, CA, 2003.
[16] M.-S. Hwang, J.-W. Lo, S.-C. Lin, An efficient user identification scheme based on id-based cryptosystem, Computer Standards & Interfaces, Elsevier Science 26 (2004) 565–569.
[17] IETF, RFC1075: Distance Vector Multicast Routing Protocol (November 1988).
[18] IETF, RFC2362: Protocol Independent Multicast-Sparse Mode (PIM-SM): Protocol Specification (June 1998).
[19] IETF, RFC3973: Protocol Independent Multicast — Dense Mode (PIM-DM): Protocol Specification (January 2005).
[20] R. Ingle, G. Sivakumar, Tunable group key agreement, Proceedings of the 32nd IEEE Conference on Local Computer Networks, IEEE Computer Society, 2007.
[21] Y. Kim, A. Perrig, G. Tsudik, Simple and fault-tolerant key agreement for dynamic collaborative group, Proceedings of ACM CCS (CCS-7), ACM (Association of Computing Machinery), 2000.
[22] W.-C. Ku, S.-M. Chen, An improved key management scheme for large dynamic groups using one-way function trees, Proceedings of the IEEE International Conference on Parallel Processing Workshops, 2003.
[23] J.O. Kwon, I.R. Jeong, K. Sakurai, D.H. Lee, Efficient verifier-based password-authenticated key exchange in the three-party setting, Computer Standards & Interfaces, Elsevier Science 29 (2007) 513–520.
[24] M.S. Lacher, J. Nonnenmacher, E.W. Biersack, Performance comparison of centralized versus distributed error recovery for reliable multicast, IEEE/ACM Transactions on Networking 8 (2) (2000) 224–238.
[25] N.-Y. Lee, Y.-C. Chiu, Improved remote authentication scheme with smart card, Computer Standards & Interfaces, Elsevier Science 27 (2005) 177–180.
[26] X.S. Li, Y.R. Yang, M.G. Gouda, S.S. Lam, Batch rekeying for secure group communications, Proceedings of the 10th International Conference on World WideWeb, ACM Press, 2001.
[27] T. Liao, Webcanal: a multicast web application, Proceedings of the Sixth Intenational WWW Conference, Santa Clara, California, 1997.
[28] J.C. Lin, C.Y. Chou, F. Lai, K.P. Wu, A distributed key management protocol for dynamic groups, Proceedings of the 27th Annual IEEE Conference on Local Computer Networks, IEEE Computer Society, 2002.
[29] S.H. Low, N.F. Maxemchuk, S. Paul, Anonymous credit cards and their collusion analysis, IEEE/ACM Transactions on Networking 4 (6) (1996) 809–816.
[30] S. Mittra, Iolus: a framework for scalable secure multicasting, Proceedings of the ACM SIGCOMM'97, ACM, 1997, pp. 277–288.
[31] M. Moyer, J. Rao, P. Rohatgi, Maintaining balanced key trees for secure multicast, IETF, draft-irtf-smug-key-tree-balance-00.txt, June 1999, http://tools.ietf.org/html/draft-irtf-smug-key-tree-balance-00.
[32] W.H.D. Ng, M. Howarth, Z. Sun, H. Cruickshank, Dynamic balanced key tree management for secure multicast communications, IEEE Trans. on Computers 56 (5) (2007) 590–605.
[33] J. Nonnenmacher, E.W. Biersack, D. Towsley, Parity-based loss recovery for reliable multicast transmission, IEEE/ACM Transactions on Networking 6 (4) (1998) 349–361.
[34] A. Perrig, D. Song, J.D. Tygar, ELK: a new protocol for efficient large-group key distribution, Proceedings of the IEEE Security and Privacy Symposium, 2001.
[35] B. Schneier, Applied cryptography, 2nd edition, John Wiley & Sons, Inc., 1996.
[36] A.T. Sherman, D.A. McGrew, Key establishment in large dynamic groups using one-way function trees, IEEE Transactions on Software Engineering 29 (5) (2003) 444–458.
[37] J. Snoeyink, S. Suri, G. Varghese, A lower bound for multicast key distribution, Proceedings of IEEE INFOCOM, vol. 1, IEEE (Institute of Electrical and Electronics Engineers, Inc.), 2001.
[38] N.F.M. Steven, H. Low, An algorithm to compute collusion paths, INFOCOM 1997, Kobe, Japan, 1997.
[39] T. Tung, Mediaboard: a shared whiteboard application for the mbone, Master's thesis, U.C. Berkeley (1998).
[40] T. Turletti, C. Huitema, Video-conferencing on the internet, ACM/IEEE Trans. Networking 4 (3) (1996) 340–351.
[41] M. Waldvogel, G. Caronni, D. Sun, N. Weiler, B. Plattner, The VersaKey framework: versatile group key management, IEEE JSAC 17 (9).
[42] X. Wang, Y.L. Yin, H. Yu, Finding collisions in the full SHA-1, Advances in Cryptology — CRYPTO'05, LNCS, Springer-Verlag, 2005, pp. 17–36.
[43] X.-M. Wang, W.-F. Zhang, J.-S. Zhang, M.K. Khan, Cryptanalysis and improvement on two efficient remote user authentication scheme using smart cards, Computer Standards & Interfaces, Elsevier Science 29 (2007) 507–512.
[44] C. Wong, M. Gouda, S. Lam, Secure group communications using key graphs, Proceedings of the ACM SIGCOMM'98, ACM, 1998, pp. 68–79.
[45] C.K. Wong, S.S. Lam, Keystone: a group key management service, International Conference on Telecommunications, ICT 2000, IEEE, 2000.
[46] K.P. Wu, S.J. Ruan, F. Lai, C.K. Tseng, On key distribution in secure multicasting, Proceedings of the 25th Annual IEEE Conference on Local Computer Networks, IEEE Computer Society, 2000.
[47] Y.R. Yang, S.S. Lam, A secure group key management protocol communication lower bound, Tech. rep. TR2000-24, Dept. of Computer Sciences, University of Texas at Austin, July 2000.
[48] Y.R. Yang, X.S. Li, X.B. Zhang, S.S. Lam, Reliable group rekeying: design and performance analysis, Proceedings of ACM SIGCOMM '01, San Diego, CA, 2001.
[49] W. Yu, Y. Sun, K.R. Liu, Optimizing the rekeying cost for contributory group key agreement schemes, IEEE Trans. On Dependable and Secure Computing 4 (3) (2007) 228–242.
[50] X.B. Zhang, S.S. Lam, D.-Y. Lee, Y.R. Yang, Protocol design for scalable and reliable group rekeying, IEEE/ACM Transactions on Networking 11 (6) (2003) 908–922.

Jen-Chiun Lin received a B.S. degree and a M.S. degree in Electrical Engineering from National Taiwan University in 1995 and 1997, respectively. He is currently a Ph.D. candidate in Computer Science Division of the Electrical Engineering Department in National Taiwan University. His research interests include network security, cryptography, distributed systems, and wireless mobile ad hoc networks.

Kuo-Hsuan Huang received the B.S. and the M.S. degrees from Dayeh University in 2001 and 2003 respectively, both in Computer Science and Information Engineering, Taiwan. He is currently a Ph.D. candidate in Computer Science of the Electrical Engineering Department in National Taiwan University, and doing research, i.e., information security, cryptography, and medical security.

Feipei Lai received a B.S.E.E. degree from National Taiwan University in 1980, and M.S. and Ph.D. degrees in Computer Science from the University of Illinois at Urbana–Champaign in 1984 and 1987, respectively. He is a Professor in the Graduate Institute of Biomedical Electronics and Bioinformatics, the Department of Computer Science and Information Engineering and the Department of Electrical Engineering at National Taiwan University. He is a Vice Superintendent of National Taiwan University Hospital. He is the Chairman of Taiwan Network Information Center. He was a Visiting Professor in the Department of Computer Science and Engineering at the University of Minnesota, Minneapolis, USA. He was also a Guest Professor at the University of Dortmund, Germany and a Visiting Senior Computer System Engineer in the Center for Supercomputing Research and Development at the University of Illinois at Urbana–Champaign. Dr. Lai holds 6 Taiwan patents and 3 USA patents currently. His current research interests are SOC low power computing and medical information system. Prof. Lai is one of the founders of the Institute of Information and Computing Machinery. He is also a member of Phi Kappa Phi, Phi Tau Phi, ACM, and the Chinese Institute of Engineers. Dr. Lai is the chairman of Taiwan Internet Content Rating Foundation. He received the Taiwan Fuji Xerox Research award in 1991. Dr. Lai is a senior member of IEEE and included in "Who's Who in Science and Engineering" and "Who's Who in the World".

Hung-Chang Lee, born in 1961, Taiwan. He got his Ph.D. degree from National Taiwan University. Currently, he is an Associate Professor in the Department of Information Management of Tamkang University.