Homomorphic One-Way Function Trees and Application in Collusion-Free Group Rekeying

JING LIU
School of Information Science and Technology, Sun Yat-Sen University, Guangzhou, People's Republic of China, 510006, liujing3@mail.sysu.edu.cn

Abstract. Efficient group rekeying is an important building-block for secure group communications. Although Sherman et al. claimed that their group rekeying scheme, OFT (One-way Function Tree) achieves perfect forward and backward secrecy, Horng later showed that functional dependency among keys in a one-way function tree subjects OFT scheme to a particular kind of collusion attack. Soon after that, Ku and Chen found several new types of collusion attack. Two solutions to prevent collusion attacks on OFT scheme have been proposed, but at the cost of a broadcast size bigger than original OFT scheme. In this paper, we prove the falsity of recently-proposed necessary and sufficient conditions for existence of a collusion attack on OFT scheme by counterexample and give new necessary and sufficient conditions for nonexistence of an arbitrary type of collusion attack. We introduce a new type of cryptographic construction — *homomorphic one-way function trees* (HOFT) by respectively substituting a homomorphic trapdoor function and a modular multiplication for the pseudorandom one-way function (OWF) and the exclusive-or mixing function in original one-way function trees (OFT). Furthermore, we propose two graph operations — tree product as well as tree blinding for HOFTs and prove that both are structure-preserving. Tree product facilitates processing multiple membership changes in a bulk operation in addition to single membership change. We demonstrate that adding/deleting leaf nodes in a HOFT is equivalent to performing a tree product of the HOFT and an incremental key chain (or tree). Tree blinding helps conceal information about a key tree without compromising its inner structure (functional dependency). Utilizing tree product and tree blinding operations, we are able to design a collusion-free group rekeying scheme based on HOFTs with the same leave-rekeying communication efficiency as original OFT scheme but with even better join-rekeying communication efficiency. This study shows that if we implement it properly, functional dependency in an OFT helps design a collusion-free and communication-efficient group rekeying scheme rather than renders it vulnerable to collusion attack.

Key words. Group rekeying, One-way function tree, Homomorphism, Collusion

1. INTRODUCTION

Many group-oriented applications, for instance, IPTV, DVB (Digital Video Broadcast), videoconferences, interactive group games, collaborative applications, stock quote streaming and web caching all require one-to-many or many-to-many group communication. Allowing for the efficient utilization of network bandwidth, IP Multicast [6] is the best way to realize group communication. Because only one data packet need be transmitted over the network for an arbitrarily sized recipient set and it traverses any link between two network nodes only once. To protect the confidentiality of group communications, a symmetric key called *group key* is used to encrypt the data traffic. For security-sensitive applications (e.g. military applications and highly secretive conferences), the key must be changed for every membership change. To prevent a new member from decoding messages exchanged before it joined a group, a new key must be distributed for the group when a new member joins. Therefore, the joining member is not able to decipher previous messages even if it has recorded earlier messages encrypted with the old key. This security requirement is called *group backward secrecy* [5]. On the other hand, to prevent a departing member from continuing access to the group's communication (if it keeps receiving the messages), the key should be changed as soon as a member leaves. Therefore, the departing member will not be able to decipher future group messages encrypted with the new key. This security requirement is called *group forward secrecy* [5][1]. To provide both *group backward secrecy* and *group forward secrecy*, the group key must be updated upon every membership change and distributed to all the authorized members. This process is referred to as *group rekeying* (or *group key management*) in literature. Respectively, the rekeying process due to a joining membership change (a departing membership change) is referred to as *join rekeying* (*leave rekeying*). For large dynamic groups with frequent changes in membership, how to design a scalable group rekeying scheme is a big challenge. Since the late 1990s, a continuing research effort has been carried out, and today has seen a huge body of literature. See [18] for an excellent survey and a recent survey is [4]).

Among the large number of generic group rekeying schemes (by "generic" we mean making no additional assumption such as making use of trusted platform technology), tree-based schemes [23, 22, 1, 21] are the most efficient ones to date. They have a communication complexity of $O(\log_2 n)$ for a group size of n [18]. Recent research [16] has also confirmed that $\log_2 n$ is the lower bound of the communication complexity for generic secure group key management protocols.

The seminal scheme among all tree-based ones is the named *Logical Key Hierarchies* (LKH) that is independently proposed by Wong et al. [23] and Wallner et al. [22]. In LKH scheme, each internal node in the key tree represents a key encryption key (KEK), each leaf node of the key tree is associated with a group member and the root node represents the group key. Key associated with the internal node is shared by all members associated with its descendant leaf nodes. Every member is assigned the keys along the path from its leaf to the root. When a member leaves the group, all the keys that the member knows should be changed. If n represents the total number of members in a group and we consider a full and balanced binary tree, leave rekeying using LKH requires at least $2\log_2 n$ key encryptions and transmission by key server. When a member joins, the key server chooses a position nearest to the root for it and changes all the keys from the parent of the joining member to the root. Join rekeying using logical key hierarchy requires encryptions and transmission of $2\log_2 n$ keys by key server. Another novel tree-based scheme is *One-way Function Tree* (OFT) proposed by Sherman et al. [1, 21] (see section 2.1 for details). In comparison with LKH, OFT scheme nearly halves the communication overhead in leaving rekeying. However, Horng [10] showed that OFT is vulnerable to a particular kind of collusion attack (see section 2.2 for details). Soon after, Ku and Chen [12] found new types of collusion attacks and also proposed an improved scheme to prevent any collusion attack. But leaving rekeying using their approach has a communication complexity of $O((\log_2 n)^2+\log_2 n)$, and hence their approach loses the advantage of original OFT over LKH. Recently, Xu et al. [24] showed that all the known attacks on OFT can be generalized to a generic collusion attack. They also derived necessary and sufficient conditions for this attack to exist and proposed a scheme to prevent the collusion attack while minimizing the average broadcast size of rekeying message. But the scheme requires a storage linear to the size of the key tree ($O(2n-1)$) and it still has bigger broadcast size than LKH.

In this paper, we prove the falsity of Xu et al.'s necessary and sufficient conditions for existence of a collusion attack on OFT scheme by counterexample and give new necessary and sufficient conditions for nonexistence of an arbitrary type of collusion attack. We introduce a new cryptographic construction — *homomorphic one-way function trees* (HOFT) by respectively substituting a homomorphic trapdoor function and a modular multiplication for the pseudorandom one-way function and the exclusive-or mixing function in original one-way function trees (OFT). We propose two tree operations — tree product and tree blinding for HOFTs and prove that both are structure-preserving. We also demonstrate that adding/deleting leaf nodes in a HOFT is equivalent to performing a tree product of the HOFT and an incremental key chain (or tree). Tree blinding helps conceal information about a key tree without compromising its inner structure. It has been shown that the functional dependency among keys in a one-way functions tree (OFT) subjects OFT scheme to collusion attacks. Two solutions [12, 24] that trade off communication efficiency for collusion resistance have been proposed. Utilizing tree product and tree blinding operations, we are able to design a collusion-free group rekeying scheme based on HOFTs with the same leave-rekeying communication efficiency as original OFT scheme but with even better join-rekeying communication efficiency. Although Sherman et al. discussed multiple addition and eviction in their paper [21], they failed to give the concrete group rekeying algorithms as they dealt with single membership change. To the authors' knowledge, our scheme is the first OFT-based batch rekeying one so far. Tree product facilitates processing multiple membership changes in a bulk operation in addition to single membership change. We present various algorithms to cope with all possible occasions of multiple membership changes.

The remainder of this paper is organized as follows. Section 2.1 gives a closer look at OFT scheme. Section 2.2 reviews different kinds of collusion attacks on it. Section 2.3 introduces two improvements on OFT to prevent collusion attacks. In section 2.4, we prove the falsity of Xu et al.'s necessary and sufficient conditions for a collusion attack on OFT scheme to exist by counterexample and give new necessary and sufficient conditions for nonexistence of an arbitrary type of collusion attack. Section 2.5 gives further comments on collusion attacks. Sections 3.1 present the concept of homomorphic one-way function tree (HOFT). In section 3.2, we propose two graph operations on HOFTs: tree product and tree blinding and prove both are structure-preserving. Section 4 describes a collusion-free group rekeying scheme based on HOFTs which can handle single membership change and multiple membership changes in the same framework at length. Section 5 gives a heuristic security analysis of our group rekeying scheme. Section 6 gives a comparison between our scheme and other related schemes. Section 7 concludes this paper and gives some topics for future research.

2. RELATED RESEARCH

2.1 Introduction to One-way Function Tree

Group rekeying using One-way Function Tree (OFT) was proposed by Sherman, Balenson and McGrew [1, 21]. The idea of using one-way function (OWF) in a tree structure originated from Merkle. In his report [15], Merkle provided a method to authenticate a large number of public validation parameters for a one-time signature scheme by using a tree structure in conjunction with a one-way and collision-resistant hash function (the famous Merkle authentication tree).

For the sake of simplicity, we adopt the terminology and formulations from paper [1] instead of the more recent ones presented in paper [21]. A *key server* maintains a balanced binary key tree for a group. When a member joins/leaves the group, it must update the key tree and distribute a rekeying message to the group to maintain group forward secrecy and group backward secrecy. A one-way function key tree is computed in a bottom-up manner using an OWF (e.g., MD5, SHA-1) and a mixing function (e.g. XOR) as follows. Except the root node, each internal node is associated with two keys: a secret key (also called unblinded key or node key in specific context) and a blinded key. The blinded key is computed by applying an OWF (also called blinding function) g to the secret key. Every internal secret key is computed by applying a mixing function f to the two blinded keys respectively associated with its two child nodes (*child blinded keys* for short). Like LKH, the key server shares a unique secret key (leaf node key) with every group member via a secure channel established during the registration protocol. However, unlike LKH, the key server does not send the members those secrets keys along the path from their leaf node to the root. Instead, it supplies each member the blinded keys of the siblings of the nodes in the path from its associated leaf node to the root of the tree. Each member uses those blinded keys to compute all the secret keys in its path from its parent to the root. Like LKH, secret key associated with the internal node is shared by all members associated with its descendant leaf nodes and the root secret key is the group key.

The structure of an OFT is illustrated by Figure 1. For example, member *A* shares a unique secret key K_a with key server. The key server sends *A* the blinded keys $K_b' = g(K_b)$, $K_c' = g(K_c)$. *A* can compute all the secret keys along the path from its parent to the root by computing $K_{ab} = g(K_a) \oplus K_b'$, $K_{a.c} = g(K_{ab}) \oplus K_c'$. ('$\oplus$' denotes an XOR function)

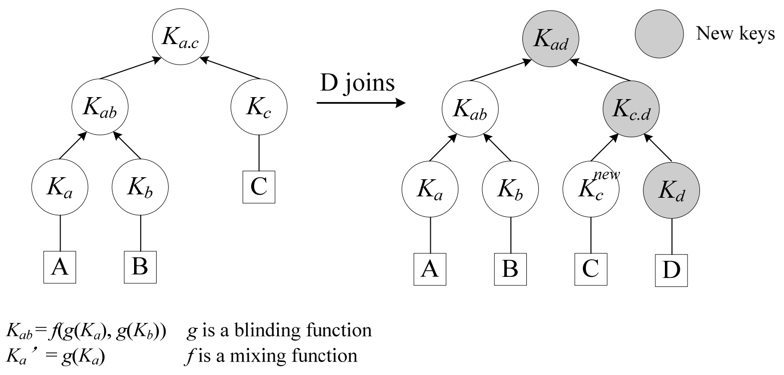


Fig. 1 Join rekeying in OFT

Whatever group rekeying is performed, the following invariant should be maintained.

Key distribution Invariant — Each legitimate member knows the node keys on the path from its associated leaf node to the root (and therefore the corresponding blinded keys along this path), and the blinded keys that are siblings to this path, and no other node keys nor blinded keys.

Figure 1 illustrates the join rekeying in OFT. When *D* joins, the key server chooses a leaf node nearest to the root node (e.g., the leaf node associated with *C*) and splits it into two nodes to create an empty leaf node for *D*. All the secret keys associated with the nodes in the path from the parent of *D*'s leaf node to the root should be changed, and hence those corresponding blinded keys are changed. Throughout this paper, we use $\{X\}\_Y$ to denote encryption of *X* with a key *Y* by using symmetric encryption algorithm. When *D* joins, the key server needs to construct and multicast a rekeying message as: $\{K_{c.d}'\}\_K_{ab}$, $\{K_c^{new}$,

K_d'}_K_c, {K_{ab}', K_c^{new}'}_K_d. That is to say, all the rekeyed blinded keys are encrypted with their siblings' secret keys. Note that K_c must be changed into K_c^{new}. Otherwise, *A* and *B* both know K_c', which violates the key distribution invariant for the updated key tree. What is more important is that since the joining member *D* will be supplied with K_{ab}' and K_c', it will be able to obtain the supposed past group key $K_{a.c}$ by computing $K_{a.c} = K_{ab}' \oplus K_c'$, which violates group backward secrecy.

Consider a full and balanced OFT with *n* members (after the join). The key server needs to encrypt and send $\log_2 n$ blinded keys to the new member. The key server also needs to encrypt and send $\log_2 n$ new blinded keys to the other pre-existing members. It also needs to encrypt and send a new unblinded key K_c^{new} to the rekeyed member *C*. In total, the key server needs to encrypt and send $2\log_2 n+1$ blinded keys when a member joins the group. In addition, the key server needs to compute $\log_2 n$ new secret keys and $\log_2 n+1$ new blinded keys. This amounts to $2\log_2 n+1$ OWF computations. The joining member needs to perform $\log_2 n$ decryptions to extract all its $\log_2 n$ blinded keys from the rekeying message and then compute all its nodes keys including the current group key in a bottom-up manner as illustrated in Figure 1. If a pre-existing member has *l* node keys in need of change due to the addition of a new member to the key tree, it only needs to perform one decryption to extract its single rekeyed blinded key, and then compute the new *l* node keys by performing *l* hash computations and *l* exclusive-or computations in a bottom-up manner as illustrated in Figure 1 (Note that the member already holds *l* unchanged blinded keys corresponding to *l* rekeyed node keys). Whereas for LKH scheme, the pre-existing member needs to perform *l* decryptions to extract the *l* rekeyed node keys. This merit of OFT superior to LKH has not been noticed by existing literatures and even by OFT's inventors.

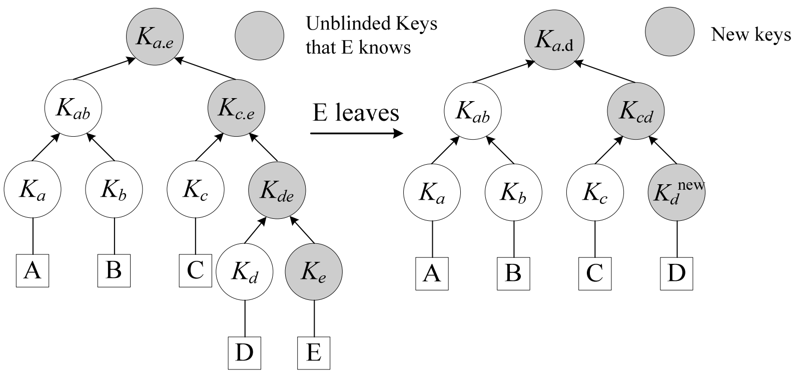


Fig. 2 Leave rekeying in OFT

Leave rekeying is depicted in Figure 2. When *E* leaves, all the secret keys in the path from the parent of *E*'s leaf node to the root should be changed. If *E*'s sibling is a leaf node, the key server changes K_d to K_d^{new}, and sends K_d^{new} encrypted with K_d. If *E*'s sibling is an internal node, the key server needs to pick one of the internal node's descendant leaf nodes (e.g. the leftmost one) and change its unblinded key to trigger rekeying the subtree rooted at K_d. Then the key server replaces *E*'s parent node K_{de} with *E*'s rekeyed sibling node K_d^{new} (or rekeyed sibling subtree rooted at K_d^{new}). This action taken by the key server results in rekeying of all the keys in the path from the departing member's parent node to the root in effect. When *E* leaves, the key server needs to construct and multicast a rekeying message as: $\{K_{cd}'\}\_K_{ab}$, $\{K_d^{new}{}'\}\_K_c$, $\{K_d^{new}\}\_K_d$. K_d must be changed into K_d^{new}. Otherwise, *C* knows K_d', which violates the key distribution invariant for the old key tree. What's more important is that the evicted *E* will be able to obtain the supposed future group key $K_{a.d}$ by computing $K_{cd} = K_c' \oplus K_d'$ and $K_{a.d} = K_{ab}' \oplus K_{cd}'$, which violates group forward secrecy.

Consider a full and balanced OFT with *n* members (after the leave). The key server needs to encrypt and send $\log_2 n$ new blinded keys to the group. In addition, it needs to encrypt and send the new unblined key K_d^{new} to the rekeyed member *D*. In all, the key server needs to encrypt and send $\log_2 n+1$ blinded keys when a member joins the group. The key server also needs to compute new secret keys and new blinded keys, and hence perform $2\log_2 n+1$ OWF computations. If a legitimate member has l node keys in need of change, it only needs to perform one decryption to extract its single rekeyed blinded key, and then compute the new l node keys by performing l hash computations and l exclusive-or computations in a bottom-up manner. Whereas for LKH scheme, the member needs to perform l decryptions to extract the l rekeyed node keys.

2.2 Collusion attacks on OFT scheme

In LKH, all the keys in the key tree are randomly chosen and thus independent with each other. The hierarchical structure of keys only represents the logical subgroup relationship among the members, that is, key associated with the internal node is shared by all members associated with its descendant leaf nodes. While in OFT, besides the logical subgroup relationship, there is a functional dependency relationship among the secret keys. Functional dependency among keys allows leave rekeying in OFT to save half of communication cost compared to LKH. However, the same relationship also renders it vulnerable to collusion attacks. Different kinds of collusion attacks on OFT are found sequentially by Horng [10] as well as Ku and Chen [12]. We depict all the collusion scenarios in Figure 3.

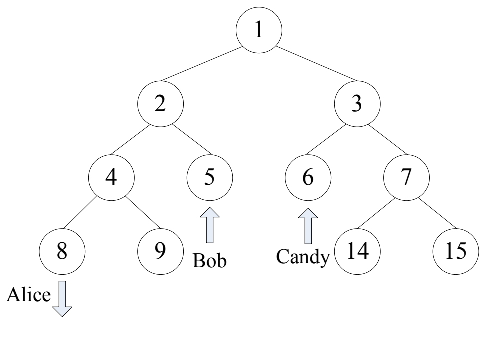


Fig. 3 Scenarios of Collusion Attacks on OFT

2.2.1 Horng's attack

The first collusion attack on OFT attributed to Horng is as follows. Referring to Figure 3, suppose that Alice, associated with node 8, is evicted at time t_A, and later Candy joins the group at time t_C and it is associated with node 6. We denote the secret key of node i in the time interval between t_A and t_C as $\boldsymbol{K}_{i[t_A,t_C]}$. The same notation will be used in the rest of this paper. If there are no changes in group membership between time t_A and t_C, according to the OFT scheme, $\boldsymbol{K}_{3[t_A,t_C]}$ is not affected by the eviction of Alice. The evicted Alice brings out the blinded key $\boldsymbol{K}_3'{}_{[t_A,t_C]}$ with her. Moreover, the secret key of node 2 is updated when Alice is evicted, and then remains unchanged even after Candy joins. Candy obtains the blinded key $\boldsymbol{K}_2'{}_{[t_A,t_C]}$ at the time of joining. Collectively knowing $\boldsymbol{K}_2'{}_{[t_A,t_C]}$ and $\boldsymbol{K}_3'{}_{[t_A,t_C]}$, Alice and Candy can collude to obtain the group key in the time interval [t_A, t_C] by computing $\boldsymbol{K}_{1\,[t_A,t_C]} = \boldsymbol{K}_2'{}_{[t_A,t_C]} \oplus \boldsymbol{K}_3'{}_{[t_A,t_C]}$. Therefore, the OFT scheme fails to provide group forward secrecy against Alice and group backward secrecy against Candy at the same time. The above attack is due to unchanging keys of the root's children. Horng thus proposed two necessary conditions for a collusion attack to exist: (1) the two colluding nodes must be evicted and join at different subtree of the root; (2) no key update happens between time t_A and t_C. Later, Ku and Chen showed that neither of these two conditions is necessary by proposing two new kinds of collusion attacks.

2.2.2 Ku and Chen's attacks

The first kind of collusion attack proposed by Ku and Chen is exemplified by the following scenario. Referring to Figure 3, suppose that Alice is evicted at time t_A, and later Bob joins the group at time t_B and it is associated with node 5. If there are no changes in group membership between time t_A and t_B, for the same reason as Horng's attack, Alice and Bob can collude to compute $\boldsymbol{K}_{2[t_A,t_B]}$. While during the time interval [t_A, t_B], the secret key of node 3 remains unchanged. Alice and Bob both know the blinded key $\boldsymbol{K}_3'{}_{[t_A,t_B]}$. Therefore, knowing $\boldsymbol{K}_{2[t_A,t_B]}$ and $\boldsymbol{K}_3{}'_{[t_A,t_B]}$, they can compute the group key $\boldsymbol{K}_{1[t_A,t_B]}$. This attack does not satisfy the first necessary condition proposed by Horng.

The second kind of collusion attack is illustrated by the following scenario. Suppose that Alice is evicted at time t_A, later Bob joins the group at time t_B, and lastly Candy joins the group at time t_C. We also assume that there are no changes in group membership either between time t_A and t_B or between time t_B and t_C. The secret key of node 3 remains unchanged after the eviction of Alice until Candy joins the group. Alice brings out the blinded key $\boldsymbol{K}_3{}'_{[t_A,t_C]}$ with her after the eviction. The secret key of node 2 is updated when Bob joins the group and then remains unchanged even after Candy joins the group. Candy obtains the blinded key $\boldsymbol{K}_2{}'_{[t_B,t_C]}$ at the time of joining. Collectively knowing $\boldsymbol{K}_3{}'_{[t_A,t_C]}$ and $\boldsymbol{K}_2{}'_{[t_B,t_C]}$, Alice and Candy can collude to compute the group key $\boldsymbol{K}_{1[t_B,t_C]}$(note that $[\boldsymbol{t}_B,\boldsymbol{t}_C]\subset[\boldsymbol{t}_A,\boldsymbol{t}_C]$). This attack denies the second necessary condition proposed by Horng.

2.3 Improvements on OFT scheme

Here, we discuss the essential reasons why OFT scheme is vulnerable to collusion attacks. When a new member joins, it will be supplied with the blinded keys that were once used to compute the past group key before the new member joins. That is to say, the joining member receives partial information about the past group key, which is partially against the requirement of group backward secrecy. It is possible for the joining member to combine its knowledge with other member's to compute a valid past group key. On the other hand, when a member leaves, it brings out the knowledge about the blinded keys that remain the same for a certain time interval after the departing member leaves. These blinded keys will be used to compute the future group key. That is to say, the evicted member brings out partial information about the future group key, which is partially against the requirement of group forward secrecy. It is possible for the evicted member to combine its knowledge with other member's to compute the future group key.

From the above discussion, it is possible to devise a solution for preventing collusion attacks either by preventing evicted member from bringing out any knowledge about future group key or by supplying joining member with no knowledge about past group key. Each of the following two improvements on the OFT scheme is just aiming at one aspect to prevent collusion attack.

2.3.1 Ku and Chen's improvement

Ku and Chen improve the OFT scheme by changing all the keys known by an evicted member upon the eviction. That is to say, not only all the secret keys associated with the nodes in the path from the parent of evictee's leaf node to the root, but also all the blinded keys associated with the siblings of those nodes in that path must be changed. For example, in Figure 3, when Alice is evicted, the secret key of node 5 and node 3 will be updated in addition to that of node 4, node 2 and node 1 as required by the original scheme. The additional updates of secret keys increase the broadcast size by $(\log_2 n)^2$ keys. Therefore, the key server needs to encrypt and send $(\log_2 n)^2+\log_2 n+1$ keys in all.

An opposite solution aiming at the joining member can be obtained by not only changing all the secret keys on the joining member's path to the root as required by the original scheme, but also changing all the blinded keys associated with siblings of those nodes in that path before supplying the joining member with those blinded keys. An interesting option enabled here is that we can decide to use either the above solution or Ku and Chen's solution according to the frequency of evictions or joins. When member evictions are rare, we can choose to use Ku and Chen's scheme. Otherwise, the above solution will be used.

2.3.2 Xu et al.'s improvement

Xu et al. observed that collusion between an evicted member and a joining member is not always possible and its success depends on the temporal relationship among them. One apparent way to reduce the broadcast size is to change additional keys only when a collusion attack is indeed possible. For that purpose, Xu et al. [24] proposed a stateful method in which the key server tracks all evicted members and records all the knowledge of them. Every time a new member joins, the key server checks against that knowledge to decide whether this joining member could collude with any previously evicted node. To track all evicted members and record all the knowledge of them, Xu et al.'s scheme has a storage requirement linear to the size of the key tree. Because Xu et al.'s scheme only performs additional key update when necessary, it has lower communication overhead than Ku and Chen's scheme. Although Xu et al. shows that their scheme has lower communication overhead than LKH scheme for small to medium groups, the increasing number of collusion attacks renders their scheme less efficient that LKH for large groups.

Xu et al. propose three propositions to support the correctness of their method. They first consider a generic collusion attack (depicted in Figure 4) on OFT scheme. Suppose that a member A evicted at time t_A and a member C joining the group at a later time t_C. Let B, D, E, and F respectively denote the subtrees rooted at L, R, R', and R''. Let t_{DMIN}, t_{EMIN}, and t_{FMIN} correspondingly denote the time of the first key update after t_A that happens in D, E, and F. Let t_{BMAX}, t_{EMAX}, and t_{FMAX} denote the time of the last key update before t_C that happens in B, E, and F, respectively.

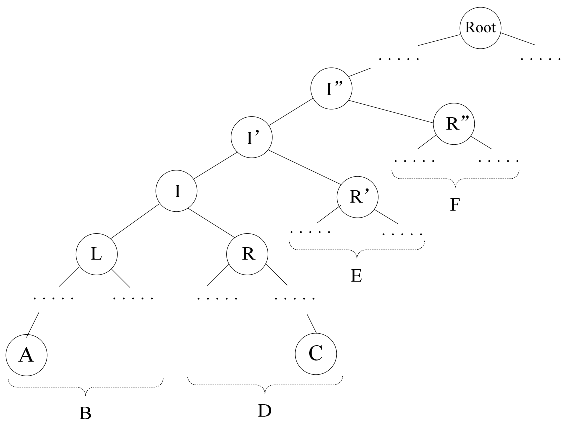


Fig. 4 A Generic Collusion Attack on OFT

Xu’s proposition 1. *For OFT scheme, referring to Figure 4, the only secret keys that can be computed by A and C when colluding are:*

- *K_I in the time interval* $[t_{BMAX}, t_{DMIN}]$,
- *$K_{I'}$ in* $[t_{BMAX}, t_{DMIN}] \cap ([t_A, t_{EMIN}] \cup [t_{EMAX}, t_C])$,
- *$K_{I''}$ in* $[t_{BMAX}, t_{DMIN}] \cap ([t_A, t_{EMIN}] \cup [t_{EMAX}, t_C]) \cap ([t_A, t_{FMIN}] \cup [t_{FMAX}, t_C])$,

and so on, up to the root. Notice that these intervals may be empty and the node I’s position is where the path of A to the root and that of C merges.

In fact, it can be easily verified that all the collusion attacks presented in section 2.2 are special instances of this generic attack.

Xu’s proposition 2. *A pair of colluding members A and C cannot compute any node key which they are not supposed to know by the OFT scheme, if*

- *A is evicted after C joins.*
- *A and C both join*
- *A and C are both evicted.*

This proposition confirms that the above generic collusion attack is the only collusion pattern of interest. Based on the above two propositions, the authors give the following sufficient and necessary conditions for an arbitrary type of collusion attack to exist.

Xu's proposition 3. *For OFT scheme, an arbitrary collection of evicted members and joining members can collude to compute some secret key not already known, if and only if the same secret key can be computed by a pair of members in the collection.*

Unfortunately, in their proof of this proposition, the authors claim that to compute a secret key not already known, the colluding members must know both child blinded keys of the secret key by themselves. However, the colluding members may get to know those child blinded keys by collusion too, but not by themselves. We present an interesting counterexample in the following section that denies the necessity of proposition 3, and then propose new necessary and sufficient conditions for nonexistence of an arbitrary type of collusion attack.

2.4 A counterexample and new necessary and sufficient conditions

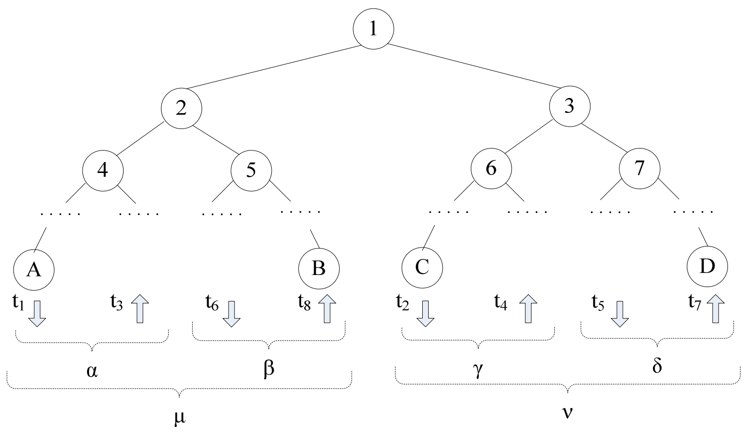


Fig. 5 A Counterexample against Xu's Proposition 3

We consider a collusion scenario depicted in Figure 5. Suppose that Dean (*D*) and Bob (*B*) join the group sequentially at time t_7 and t_8, while Alice (*A*) and Colin (*C*) leave the group sequentially at time t_1 and t_2. The chronological order of t_1, t_2, … , and t_8 is just determined by the numerical order of their subscripts. Let α, β, γ, δ, μ, and v denote the subtrees rooted at node 4, 5, 6, 7, 2, and 3, respectively. In addition to the above changes in group membership, there are changes sequentially at time t_3, t_4, t_5, and t_6, which respectively happened in α, γ, δ, and *β*. Let $\boldsymbol{t}^{\boldsymbol{X}}_{\alpha MAX}$ denotes the time of the last key update before *X* joins the group that happens in *α*. Let $\boldsymbol{t}^{\boldsymbol{Y}}_{\beta MIN}$ denotes the time of the first key update after *Y* leaves the group that happens in *β*. Recall that K_X denotes the secret key associated with node *X* and K_X' denotes the corresponding blinded key. And $K_{X[t1,\, t2]}$ denotes the secret key in the time interval [t_1, t_2].

According to Xu's proposition 1, Alice and Bob can collude to compute K_2 in the time interval [$\boldsymbol{t}^{\boldsymbol{B}}_{\alpha MAX}$,$\boldsymbol{t}^{\boldsymbol{A}}_{\beta MIN}$], i.e.,$\boldsymbol{K}_{2[t_3,t_6]}$; Colin and Dean can collude to compute K_3 in the time interval [$\boldsymbol{t}^{\boldsymbol{D}}_{\gamma MAX}$, $\boldsymbol{t}^{\boldsymbol{C}}_{\delta MIN}$], i.e., $\boldsymbol{K}_{3[t_4,t_5]}$. Thus, collectively knowing $\boldsymbol{K}_{2[t_3,t_6]}$ and $\boldsymbol{K}_{3[t_4,t_5]}$, Alice, Bob, Colin and Dean can collude to compute $\boldsymbol{K}_{1[t_4,t_5]}$. However, we shall show that any possible pair of evicted and joining members cannot collude to compute $\boldsymbol{K}_{1[t_4,t_5]}$.

According to Xu's proposition 1, all the secret keys that can be computed by Alice and Bob when colluding are:

- K_2 in the time interval [$\boldsymbol{t}^{\boldsymbol{B}}_{\alpha MAX}$,$\boldsymbol{t}^{\boldsymbol{A}}_{\beta MIN}$], i.e.,$\boldsymbol{K}_{2[t_3,t_6]}$,

- K_1 in the time interval [$\boldsymbol{t}^{\boldsymbol{B}}_{\alpha MAX}$, $\boldsymbol{t}^{\boldsymbol{A}}_{\beta MIN}$] $\cap$ ([t_1, $\boldsymbol{t}^{\boldsymbol{A}}_{\nu MIN}$] $\cup$ [$\boldsymbol{t}^{\boldsymbol{B}}_{\nu MAX}$,t_8]), but evaluation of that formula equals [t_3, t_6] $\cap$([t_1,t_2]$\cup$[t_7,t_8])=$\varnothing$.

So Alice and Bob cannot collude to compute $\boldsymbol{K}_{1[t_4,t_5]}$. By the same argument, we can prove that for the rest of eviction-joining scenarios, i.e., the collusion between Colin and Dean, between Alice and Dean, or between Colin and Bob, $\boldsymbol{K}_{1[t_4,t_5]}$ cannot be computed either.

This counterexample thus denies that the necessity of Xu's proposition 3. Here we present new necessary and sufficient conditions for nonexistence of an arbitrary type of collusion attack.

Proposition 4 *An arbitrary collection of evicted members and joining members cannot collude to compute any secret key not already known, if and only if arbitrary pair of evicted and joining members cannot collude to compute any secret key not already known.*

Proof: The necessity is trivial. We prove the sufficiency by contradiction. Recall that K_i' is the blinded version of K_i. For an arbitrary node key K_i in a HOFT X, its left child and right child is denoted by K_{2i} and K_{2i+1} respectively. Suppose that a collection of evicted members and joining members can collude to compute a new secret key $\boldsymbol{K}_{i[t_1,t_2]}$ which must attribute to nothing but either of the following two causes:

(1) There exist two colluding members who have already known K_{2i}' and K_{2i+1}' respectively for some time intervals that are supersets of $[t_1, t_2]$. Therefore, they can collude to compute $\boldsymbol{K}_{i[t_1,t_2]}$;

(2) At least a subset of colluding members can collude to compute either K_{2i} or K_{2i+1} for some time interval that is a superset of $[t_1, t_2]$.

If it attributes to the former, the two members must be a pair of evicted and joining members according to Proposition 2, which stands in contradiction to our hypothesis, and thus the proposition follows. Otherwise, let's assume that $\boldsymbol{K}_{2i[t_{a_1},t_{b_1}]}$ ($[\boldsymbol{t}_{a_1}, \boldsymbol{t}_{b_1}]$ is a superset of $[t_1, t_2]$) is not already known and can be computed by a subset of colluding members. Obviously, for $\boldsymbol{K}_{2i[t_{a_1},t_{b_1}]}$, we can repeatedly apply the same argument as above. In fact, the same argument can be repeated until either we found a satisfying pair of evicted and joining members who can collude to compute a certain node key not already known from its two child blinded keys (these keys are also internal blinded keys), each of which is separately known by a colluding member, or due to the limited size of the key tree, we stop at a certain node key not already known that has two leaf blinded keys as its children. Each of these blinded keys is separately known by a colluding member. Whatever the result, it stands in contradiction to our hypothesis, and thus the proposition follows. □

Thus, the correctness of Xu et al.'s scheme follows from proposition 1, proposition 2 and our proposition 4.

2.5 Further comments on collusion attacks

Unlike traditional security protocol (e.g., two-party key establishment protocols), group-oriented security protocols (e.g., group key establishment protocols) have an open number of group members. Two or more malicious members can collude to sabotage the security target of these protocols. Therefore, preventing collusion attack is a paramount requirement when designing a group-oriented security protocol.

Although OFT was claimed to achieve perfect forward and backward security by its inventors [21], collusion attacks on it have been found. Because its inventors only consider collusion among evicted members (or joining members) but ignore the potential collusion between evictees and joining members. Strictly speaking, no matter which kind of membership (evicted, joining or even legitimate) the colluding members may hold, a collusion attack already occurs when any two or more of them can collude to compute new knowledge about intermediate secret keys or group key besides what they already know. Therefore, we must strengthen the definition for collusion attack to ensure that it covers all possible patterns of collusion. We prefer to use Fan et al's informal definition [7] as follows. A *collusion attack* is an attack in which two or more malicious members participate in an effort to achieve more information than the aggregation of the information authorized to them.

3. HOMOMORPHIC ONE-WAY FUNCTION TREE

3.1 Definition

For the security of OFT scheme, Sherman [20] give one sufficient condition that OWF *g* should be pseudorandom (e.g. MD4 or SHA-1). However, determining the precise necessary conditions for the security of OFT scheme still remains unsolved since it was proposed as an open problem by the same authors [21]. We believe that instantiating the blinding function *g* as a homomorphic trapdoor function (e.g., Rabin function [17] or RSA function [19]) will not compromise the security of OFT (see section 5 for security analysis). As a matter of fact, the binary key tree used by the well-known tree-based group Diffie-Hellman key agreement (TGDH) [11] can be regarded as an instance of OFT where the blinding function *g* is α^x (mod *p*). That function is homomorphic and not pseudorandom too.

Before we give the definition of homomorphic OFT, let's review related mathematical concepts. A group *G* with its operation "$*$" is denoted by $(G, *)$. In mathematics, given two groups $(G, *)$ and $(H, \cdot)$, a *group homomorphism* from $(G, *)$ to $(H, \cdot)$ is a function $g : G \rightarrow H$ such that for all *u* and *v* in *G*, it holds that $g(u*v) = g(u)\cdot g(v)$. From this property, one can deduce that *g* maps the identity element e_G of *G* to the identity element e_H of *H*, and it also maps inverses to inverses in the sense that $g(u^{-1}) = g(u)^{-1}$. According to this definition, Rabin function and RSA function are both homomorphic.

Depending on one's viewpoint, homomorphism can be seen as a positive or negative attribute of a cryptosystem [19]. Once homomorphism is exploited by certain cryptosystem (e.g. encryption, digital signature, MAC, hash, etc.), it will enable the ability to perform a specific algebraic operation on the original data by performing a (possibly different) algebraic operation on cryptographically transformed data. Introducing homomorphic trapdoor function for OFTs enables the ability to perform a specific "inner" operation on certain set of leaf nodes in a OFT (e.g., adding or deleting leaf nodes, blinding all nodes) by performing an "outer" graph operation on one or two OFTs without compromising the "inner" structure — functional dependency among keys.

Since all nodes in OFT are homogeneous (i.e., cryptographic keys), we choose to use self-homomorphism mapping an Abelian group to itself. If every secret key associated with a node (node key for short) in an OFT X is an element of an Abelian group G (e.g., Z_n^*, n is a composite), we say that X is defined over G (or X over G for short).

Definition 1 Homomorphic OFT — A *homomorphic OFT* (HOFT) over an Abelian group $(G, *)$ is a binary key tree which is computed using a self-homomorphic OWF g and the multiplicative operation "$*$" in a bottom-up manner as follows. For an arbitrary node key x_i in a HOFT X, if its left child and right child are denoted by x_{2i} and x_{2i+1} respectively, we have $x_i = g(x_{2i}) * g(x_{2i+1})$.

3.2 Two structure-preserving operations

We shall show that introducing homomorphic OWF (HOWF for short) for OFTs enables two structure-preserving operations — tree product and tree blinding. A binary operation (unary operation) is said to be structure-preserving if the operation takes two HOFTs (one HOFT) as input and outputs a HOFT. For convenience, in this section, we shall interchangeably use the same notation "x_i" or "y_i" to denote either a node itself or its associated node key.

3.2.1 Tree product

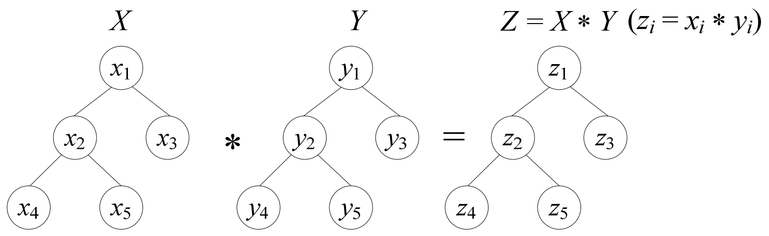


Fig. 6 Tree product

Definition 2 Tree product — Given two arbitrary HOFTs X and Y both over an Abelian group $(G, *)$ with the same structure (i.e., same height and same organization of leaf nodes), a tree product of X and Y, denoted by $X * Y$ is computed by multiplying their corresponding node keys (see Figure 6).

Note that although we use the same notation "$*$" for group operation and tree product, its meaning is context-evident.

Theorem 3.1 Given two arbitrary HOFTs X and Y both over an Abelian group $(G, *)$ with the same structure, the result of a tree product $X * Y$ is also a HOFT.

Proof: Let X and Y are two arbitrary HOFTs defined over an Abelian $(G, *)$ and $Z = X * Y$. We prove Z is also a HOFT. For an arbitrary node key $z_i \in Z$, we have (recall that for an arbitrary node key x_i in a HOFT X, its left child and right child are denoted by x_{2i} and x_{2i+1} respectively)

$z_i = x_i * y_i$ (Definition 2)

$= (g(x_{2i}) * g(x_{2i+1})) * (g(y_{2i}) * g(y_{2i+1}))$ (Definition 1, since X and Y are both HOFT)

$= (g(x_{2i}) * g(y_{2i})) * (g(x_{2i+1}) * g(y_{2i+1}))$ ("$*$" is commutative and associative)

$= g(x_{2i} * y_{2i}) * g(x_{2i+1} * y_{2i+1})$ (g is homomorphic)

$= g(z_{2i}) * g(z_{2i+1})$ (Definition 2)

According to Definition 1, Z is a HOFT. □

In other words, tree product is *structure-preserving*.

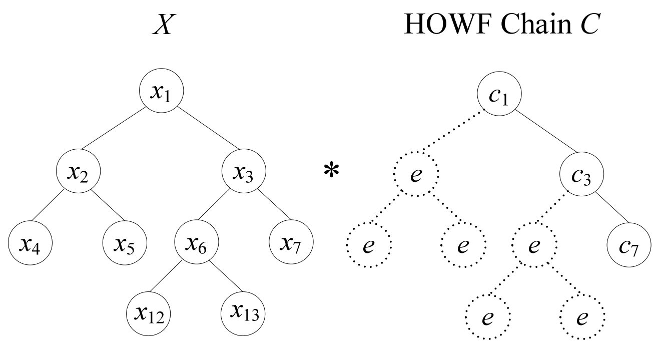


Fig. 7 Tree product of a key tree and a key chain

If a HOWF chain *C* over (*G*, ∗) has the same length as a certain path from a leaf node to the root in a HOFT *X* over (*G*, ∗), we can define a tree product of the HOFT *X* and the HOWF chain *C* based on Definition 2. Recall that the two operands of a tree product must have the same structure. Therefore, we must expand *C* with identity node keys (whose value is the identity element *e* of *G*) as in Figure 7 to make it have the same structure as *X* before performing a tree product operation. Since *g* is a group self-homomorphism, *g*(*e*) equals *e*. It is easy to check that the key tree expanded from *C* is also a HOFT. In this manner, a HOWF chain can always be transformed into a HOFT with arbitrary shape. Since *e* is an identity element, when performing a tree product of the HOFT *X* and the key tree expanded from *C*, those node keys multiplied by identity node key remain unchanged. Therefore, we can directly define the product of a HOFT *X* and a HOWF chain *C* as computed by only multiplying their corresponding node keys. According to Theorem 3.1, the result of the tree product $X * C$ is also a HOFT.

3.2.2 Tree blinding

Definition 3 Tree blinding — For an arbitrary HOFT *X* over an Abelian group (*G*, ∗) in conjunction with its HOWF *g*, a tree blinding operation based on *g* maps *X* to another key tree *Y*, denoted by $Y = g(X)$. *Y* is computed by applying *g* to every node of *X* (see Figure 8). We call *Y* a blinded key tree of *X*.

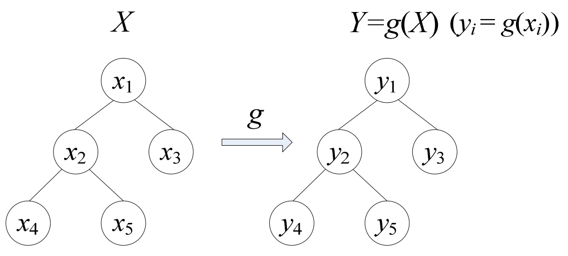


Fig. 8. Tree blinding

Theorem 3.2 For an arbitrary HOFT X over $(G, *)$, the result of a tree blinding operation on X, $g(X)$ is also a HOFT.

Proof: Let X is an arbitrary HOFT and $Y = g(X)$. We prove Y is a HOFT. For an arbitrary node key $y_i \in Y$, we have

$$
\begin{aligned}
y_i &= g(x_i) \\
&= g(g(x_{2i}) * g(x_{2i+1})) && (X \text{ is a HOFT}) \\
&= g(y_{2i} * y_{2i+1}) && (Y = g(X)) \\
&= g(y_{2i}) * g(y_{2i+1}) && (g \text{ is homomorphic})
\end{aligned}
$$

According to Definition 1, Y is a HOFT. □

Theorem 3.2 reveals that tree blinding is also a structure-preserving operation. Obviously, from any node key in a blinded key tree $g(X)$, it is computationally infeasible to gain any information about the node keys in the original key tree X due to the one-wayness of g.

4 A COLLUSION-FREE GROUP REKEYING SCHEME BASED ON HOFTS

In tree-based schemes, adding or evicting members correspond to adding or deleting relevant leaf nodes in a key tree. In this section, we demonstrate that adding or deleting leaf nodes in a HOFT X is equivalent to performing a tree product of X and an *incremental key chain* (or key tree). The incremental key tree (or key chain) includes all the incremental keys that correspond to all those node keys supposed to change when a number of members are added or evicted. Thus, the process of group rekeying is as follows. When members join or leave, the key server first constructs an incremental key tree (or key chain) according to these membership changes, and then updates the key tree by computing a tree product of it and the incremental key tree (or key chain). After that, the key server needs to communicate all the changes in the key tree to legitimate group members by broadcasting the incremental key tree (or key chain) so that those members can update their affected node keys by computing a product of those keys and their corresponding incremental keys. Although Sherman et al. discussed multiple additions and evictions in their paper [21], they failed to give specific group rekeying algorithms as they dealt with single membership change. While in our scheme, tree product facilitates processing either single membership change or multiple membership changes in a same framework.

In the following sections, we present various group rekeying algorithms dealing with all patterns of membership changes. Suppose that after a group initialization, we established a HOFT X over an Abelian group $(G, *)$ (see section 6.4.1 of book [9] for details). We shall interchangeably refer to a node and the member associated with that node. For any node key x_i, we use y_i to denote its blinded version.

4.1 Evicting a member

Due to limited space, we only discuss the case an evictee's sibling is a leaf node. Group rekeying algorithm for the case an evictee's sibling is an internal node can be easily derived from the algorithm given in this section. We first describe how to delete the evictee's leaf node in a HOFT by a tree product, and then discuss the group rekeying algorithm.

(1) Deleting a leaf node by a tree product

Figure 9, 10 and 11 illustrate how to evict a member x_{15} and construct a corresponding incremental key chain C during the same process. In the following figures, we use dotted circles to denote node keys that do not directly participate in a tree product computation (e.g., x_{14}' and x_{15}), but whose positions should be attended to. We also use a shaded and dotted node to denote an evictee.

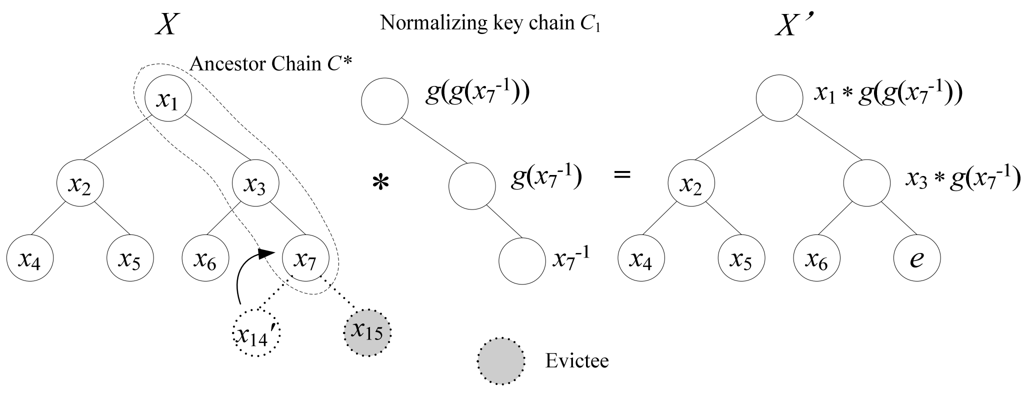


Fig. 9 Normalization

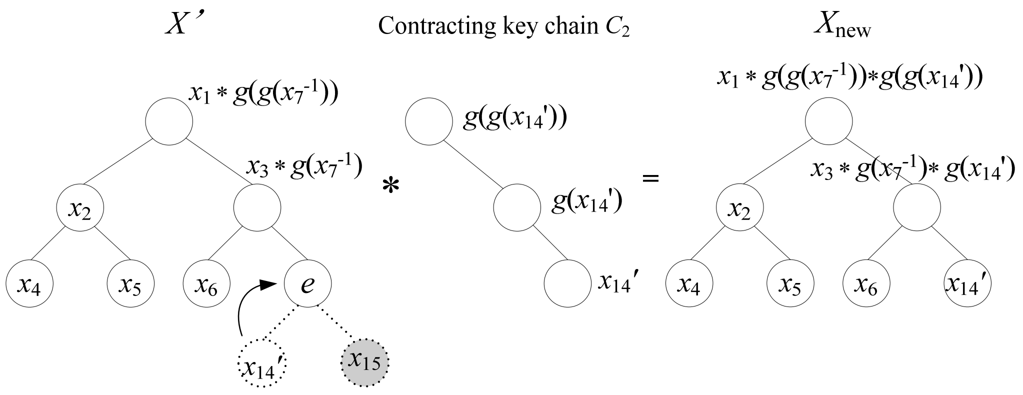


Fig. 10 Contraction

The key chain consisted of all ancestral node keys of an evictee is called an *ancestor chain* (AC) of the evictee. And the leaf node of AC is called an *anchor*. For example, in Figure 9, C^* (surrounded by a dashed line) is an AC for x_{15} and the anchor of C^* is x_7. To delete a leaf node x_{15} from a HOFT X, we take two steps. The first step called *normalization* is in fact a tree product of X and a normalizing key chain C_1 (see Figure 9). The purpose of normalization is to turn AC's anchor into an identity. We construct the normalizing key chain C_1 with the same length as C^* by recursively applying the OWF g to the inverse of AC's anchor (i.e., x_7^{-1}). Before conducting the second-step operation, the sibling node key (x_{14}) of the evictee (x_{15}) should be changed for the reason as discussed in section 2.1 (see Figure 2). We use x_{14}' to denote the new value. The second step called *contraction* (see Figure 10) is in fact a tree product of X' and a contracting key chain C_2. Its purpose is to replace the evictee's parent with the evictee's rekeyed sibling. The contracting key chain C_2 is obtained by replacing x_7 with x_{14}' in C* and recomputing the key chain to root by recursively applying an OWF g to x_{14}'. After the two steps, member x_{15} is successfully evicted.

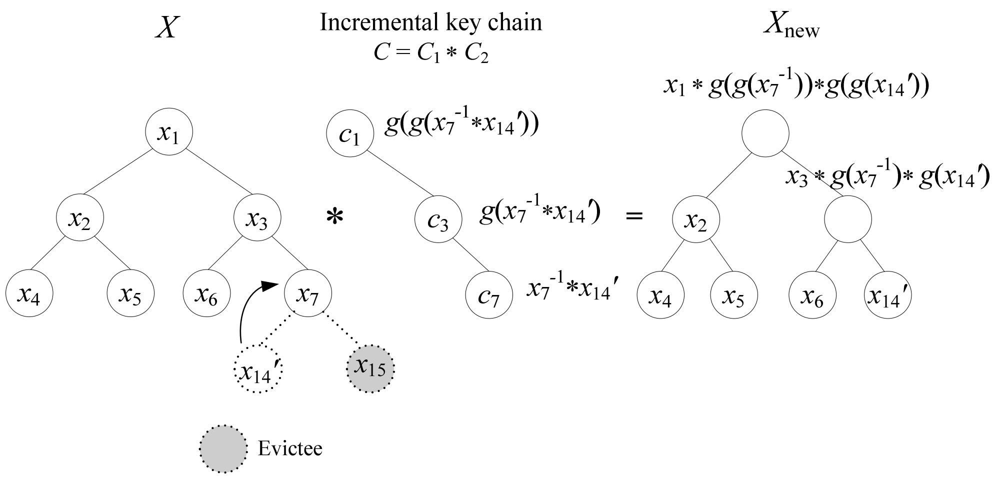


Fig. 11 Evicting a member by performing a tree product

An incremental key chain C is constructed by performing a tree product of the normalizing key chain C_1 and the contracting key chain C_2 (they have the same structure, and therefore a tree product operation can be performed). This incremental key chain has same structure as AC of the evictee and includes all the incremental keys that correspond to all those node keys supposed to change when a member is evicted. Obviously, the resulting C is a HOWF chain too. Thus, the two operations — *normalization* and *contraction* can be combined into a single one-step operation as in Figure 11. In fact, the key server can directly compute the incremental key chain C by recursively applying an OWF g to $x_7^{-1} * x_{14}$'. In a word, deleting a leaf node x_{15} from a HOFT X is equivalent to performing a tree product of X and an incremental key chain C.

(2) Group rekeying algorithm

When a member is evicted, all the node keys on AC of the evictee should be changed. To this end, the key server must construct an incremental key chain as above, update the key tree by performing a tree product of it and the incremental key chain, and then broadcast the incremental key chain to legitimate members. However, it must strictly control access to the incremental key chain not only to prevent every evictee from accessing any part of it, but also to restrict every legitimate member to the incremental keys it is entitled to (see section 5 for the reason). Therefore, the key server sends the blinded version of every incremental key except the root encrypted with the unblinded sibling key of the incremental key's correspondent in the updated key tree, i.e., $\{g(C_3)\}\_x_2$ and $\{g(C_7)\}\_x_6$. In addition, the key server also encrypts the new value of the evcitee's sibling with its old value, i.e., $\{x_{14}'\}\_x_{14}$. In a word, to evict member x_{15}, the key server broadcasts a rekeying message "$\{g(C_3)\}\_x_2$, $\{g(C_7)\}\_x_6$, $\{x_{14}'\}\_x_{14}$".

After receiving the rekeying message, every legitimate member perform one decryption to extract the blinded incremental keys corresponding to its single blinded key in need of rekeying, and then update it by multiplying its old value by corresponding blinded incremental key. After that, it recomputes all its node keys in need of change in a bottom-up manner as discussed in section 2.1.

Consider a full and balanced HOFT with n members (after the eviction). The key server needs to encrypt and send $\log_2 n+1$ blinded incremental keys when a member leaves the group. The key server also needs to compute the incremental key chain, and hence perform one modular multiplication and $\log_2 n$ OWF computations. In addition, to update the HOFT by performing a tree product, it needs to perform $\log_2 n+1$ modular multiplication computations.

If the evictee's sibling is an internal node, the node key associated with the internal node must be changed for the same reason as discussed in Part 2.1 (see Figure 2). Since it is an internal node, we have to trigger rekeying in the subtree rooted at it (we refer to it as the evictee's *sibling subtree*) by rekeying the subtree's shallowest leaf node (see section 4.6 for how to rekey the sibling subtree). Then, the key server replaces the evictee's parent with the evictee's rekeyed sibling subtree.

4.2 Evicting multiple members in a bulk operation

Bursty behaviour (a number of membership changes happen simultaneously), periodic group rekeying or batch group rekeying all require a *bulk operation* that can process multiple membership changes at the same time. The broadcast size and computational effort of multiple additions and evictions can be substantially reduced by using a bulk operation that evicts and/or adds multiple members simultaneously rather than repeatedly applying individual add or evict operations. This reduction results from the fact that a set of individual operations may repeatedly change node keys along common segments of the key tree. Let's first give the concept of a *private subtree*.

Private subtree — For a set *S* of leaf nodes in a tree *T*, a subtree *T'* of *T* is called a *private subtree* for *S*, if any leaf node not contained in *S* does not belong to *T'*. In other words, this subtree is privately and exclusively shared by leaf nodes in *S*.

4.2.1 All the evictees share a private tree

If all the evictees' leaf nodes happen to share a *private subtree* (for example, in Figure 12, members x_{24}, x_{25} and x_{13} share a private subtree *T***'** surrounded by a dotted line), we can evict all those members simultaneously by pruning this private subtree. If we regard the whole private subtree *T***'** as a leaf node x_6, pruning *T***'** is equivalent to using the algorithm in section 4.1 to delete x_6.

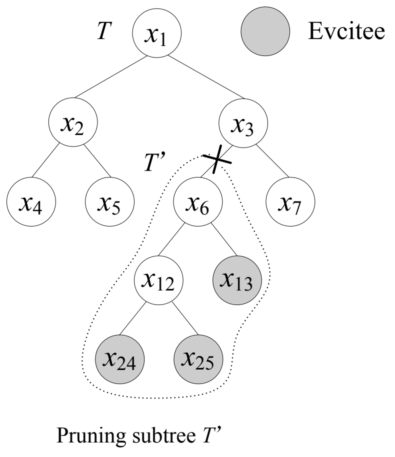


Fig. 12 Pruning

Consider a full and balanced OFT with n members (before the evictions). Based on our discussion in section 4.1-(2), when evicting l members (suppose l is a power of 2), to distribute the incremental key chain securely, the key server needs to encrypt and send $\log_2(n/l)$ blinded incremental keys. If we repeatedly apply algorithm in section 4.1 to individually process every eviction, the key server needs to encrypt and send $l * \log_2(n/l)$ keys at least.

4.2.2 Evictees are sparsely distributed

If any subset of evictees cannot share a private subtree (in other words, all evictees are sparsely distributed), we use the following algorithm to evict them.

(1) Deleting sparsely-distributed leaf nodes by a tree product

Figures 13-15 demonstrate how to delete sparsely-distributed evictees x_9, x_{11}, and x_{13} from a HOFT X by applying a tree product. Based on the discussion in section 4.1, here we only need to focus on how to construct a normalizing key tree, a contracting key tree and an incremental key tree.

① Normalization

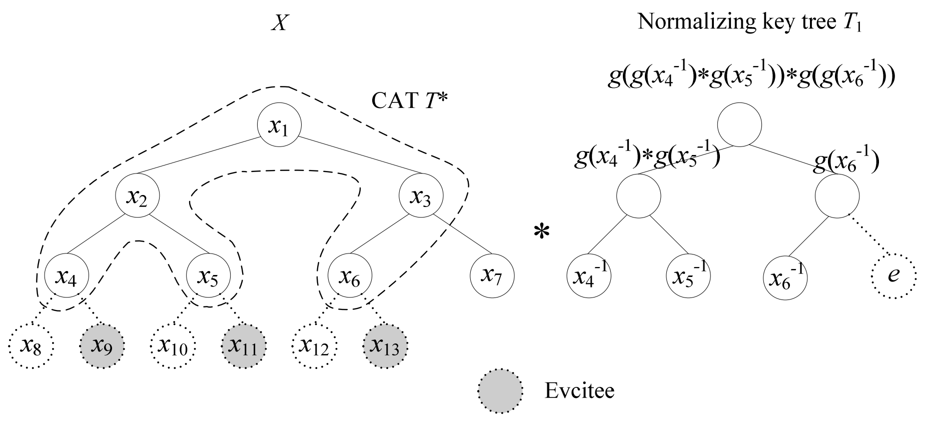


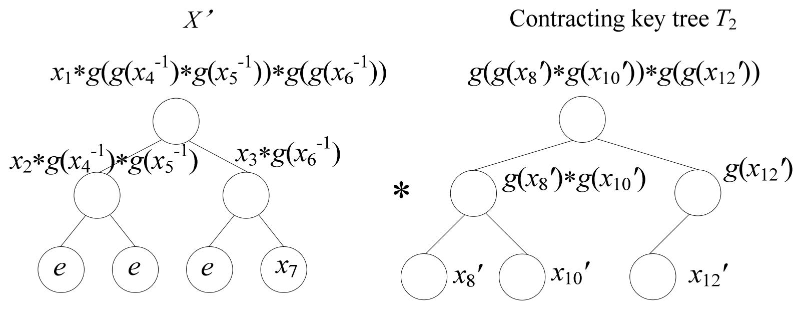
Fig. 13 Normalization for sparsely-distributed evictions

Figure 13 illustrate how to construct the normalizing key tree T_1. First of all, we present a concept *Combined Ancestor Tree* proposed by Sherman and McGrew [21].

Combined Ancestor Tree — For a set of evictees or joining members, the tree consisting of all ancestors of their associated leaf nodes is called a *Combined Ancestor Tree* (CAT).

When a set of members are evicted, all node keys need to be changed are on CAT. For evictees x_9, x_{11}, and x_{13}, their CAT is T^* (surrounded by a dashed line). Normalizing key tree T_1 is obtained by replacing each leaf node key of CAT with its inverse and recomputing the whole OFT to the root.

② Contraction



X'
Contracting key tree T_2
$x_1*g(g(x_4^{-1})*g(x_5^{-1}))*g(g(x_6^{-1}))$
$g(g(x_8')*g(x_{10}'))*g(g(x_{12}'))$
$x_2*g(x_4^{-1})*g(x_5^{-1})$
$x_3*g(x_6^{-1})$
*
$g(x_8')*g(x_{10}')$
$g(x_{12}')$
e
e
e
x_7
x_8'
x_{10}'
x_{12}'

Fig. 14 Contraction for sparsely-distributed evictions

We construct the contracting key tree T_2 by replacing every leaf node of T^* with its adjacent evictee's rekeyed sibling (or rekeyed sibling subtree) and recomputing the resulting key tree to the root (see Figure 14).

Since HOFT T_1 and T_2 have the same structure, the incremental key tree T is obtained by performing a tree product of T_1 and T_2. Obviously, the resulting incremental key tree T has the same structure as the CAT of those evictees and includes all the incremental keys that correspond to all those node keys supposed to change when a number of members are evicted. In fact, the incremental key tree T can be computed while it is traversed in a postorder manner by the key server. However, in original OFT scheme, to update every node keys on CAT, the key server needs to traverse a tree bigger than the incremental key tree here to include not only those nodes on CAT, but also all sibling of them.

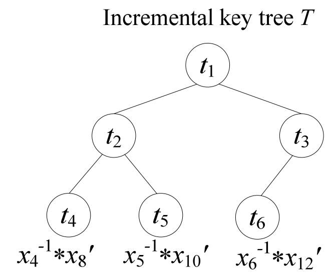


Fig. 15 An incremental key tree for sparsely-distributed evictions

(2) Group rekeying algorithm

After deleting those sparsely-distributed evictees' leaf nodes by performing a tree product, the key server must broadcast all the incremental keys to remaining members. Not only to prevent any evictee from gaining access to incremental key tree, but also to restrict every legitimate member to the incremental keys it is entitled to, the key server sends the blinded version of every incremental key (except the root incremental key) encrypted with the unblinded sibling key of the incremental key's correspondent in the updated key tree, i.e., $\{g(t_2)\}\_x_3$', $\{g(t_3)\}\_x_2$', $\{g(t_4)\}\_x_{10}$', $\{g(t_5)\}\_x_8$' and $\{g(t_6) = t_3\}\_x_7$. In addition, the key server sends the new value of every evictee's sibling encrypted with its old value, i.e., $\{x_8'\}\_x_8$, $\{x_{10}'\}\_x_{10}$, $\{x_{12}'\}\_x_{12}$.

After every legitimate member receives the rekeying message, it will be able to compute all incremental keys it is entitled to in a bottom-up manner, and then update its own rekeyed node keys by multiplying their old values by their corresponding incremental keys. For example, member x_8 is able to obtain blinded key $g(t_5)$ by decrypting $\{g(t_5)\}\_x_8'$ with its new node key x_8'. Since it can directly compute $t_4 = x_4^{-1}*x_8'$, member x_8 can compute $t_2 = g(t_4)*g(t_5)$ and then $x_2'=x_2*t_2$. Now it can decrypt $\{g(t_3)\}\_x_2'$ to obtain $g(t_3)$. In the end, it can compute $t_1 = g(t_2)*g(t_3)$ and then $x_1'=x_1*t_1$.

Since the incremental key tree has the same structure as the CAT, we can compute the broadcast overhead by the size of CAT (denoted by S_L) as in paper [21]. Consider a full and balanced OFT with n members (before l members are evicted). The key server needs to encrypt and broadcast l node keys of l evictees' rekeyed siblings. Since the size of incremental key tree is S_L, it also needs to encrypt and send S_L-1 blinded incremental keys. The key server also needs to compute the incremental key tree, and hence perform S_L modular multiplication and S_L-1 OWF computations. In addition, to update the HOFT by performing a tree product, it needs to perform S_L modular multiplication computations. In all, the key server needs to perform $2S_L$ modular multiplication computations and S_L-1 OWF computations.

Compared with repeatedly applying algorithm in section 4.1 to individually process every eviction, a bulk operation acquires an economy of scale when the size of the incremental tree is less than the number of evictees times the height of the key tree.

4.2.3 Generic case

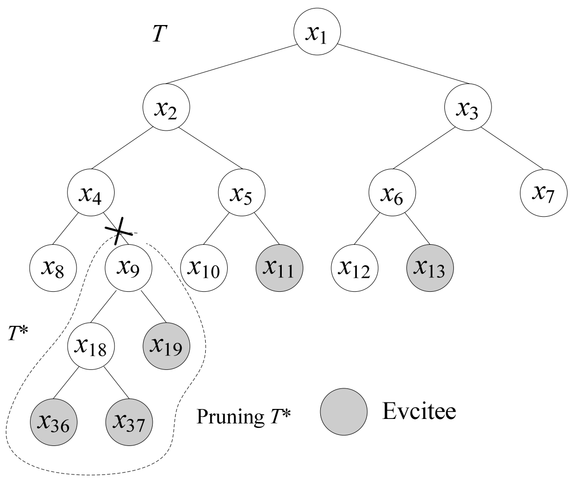


Fig. 16 A generic case for multiple evictions

Based on discussion in sections 4.2.1 and 4.2.2, we are now able to consider a generic case where certain subsets of evictees are able to form private subtrees while the rest are just sparsely distributed. For every subset of evictees capable of forming a private subtree, we regard this private subtree as a single node, i.e., their root. Thus, the generic case will become the case of sparsely-distributed evictees. For example, referring to Figure 16, if we regard the private subtree T^* shared by x_{36}, x_{37}, x_{19} as a single node x_9, T in Figure 16 becomes X in Figure 13. And deleting x_{36}, x_{37}, x_{19}, x_{11}, and x_{13} from T is equivalent to deleting sparsely-distributed evictees x_9, x_{11}, and x_{13} from X in Figure 13 using algorithm in section 4.2.2.

4.3 Adding a member

When members join the group, the key server first performs a tree blinding operation g on the current key tree X to obtain a blinded key tree $Y = g(X)$ (see Figure 8) and then adds those members' leaf nodes onto the blinded key tree Y. Same as in OFT scheme, the key server will supply every joining member with the blinded keys of the siblings of the nodes in the path from its associated leaf node to the root of the updated key tree. These blinded keys are information about the blinded key tree Y and it is computationally infeasible for the joining members (even by collusion) to compute any information about the original key tree X due to one-wayness of *g*. Therefore, a joining member will not be able to combine this knowledge with any evicted member to obtain a valid past group key. In addition, since the incremental key tree or key chain is constructed from a blinded key tree *Y* and it contains no information about the original key tree *X* either, controlling access to the incremental key tree (or key chain) is not necessary any more. The key server just broadcasts the leaf incremental keys encrypted with old group key.

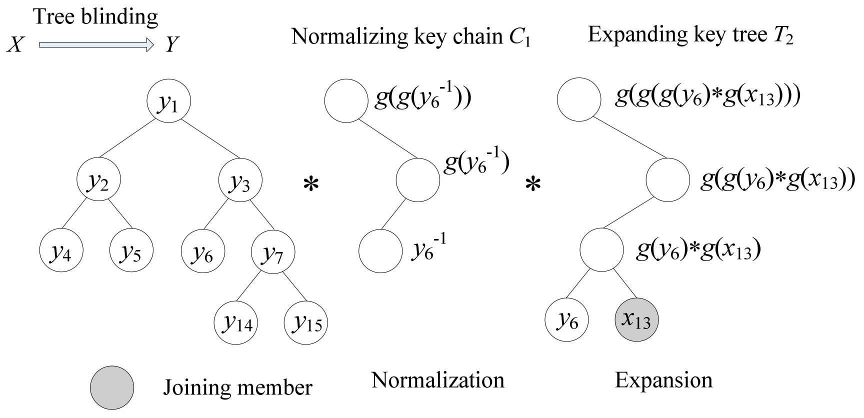


Fig. 17 Adding the joining member’s leaf node

Figure 17 illustrates how to add a leaf node to a HOFT by applying tree product.

(1) **Adding a leaf node onto a blinded key tree by a tree product**

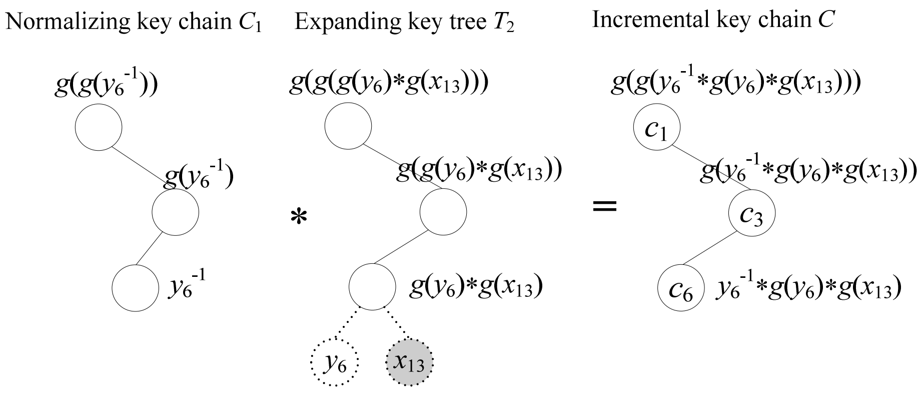


Fig. 18 an incremental key chain for a single join

In the blinded key tree *Y*, we first choose a leaf node whose position is nearest to the root node (e.g., y_6) as the node to be normalized and construct the normalizing key chain C_1

similar to the algorithm in section 4.1. Note that the difference between the normalization operation for a single addition and that for a single eviction is the place of node to be normalized. In the former, the node to be normalized is where we add a new node, whereas, in the latter, the node to be normalized is the parent of the node to be deleted. Now we show how to construct the expanding key tree. Expanding the path from the node to be normalized to the root by creating two new child nodes at its anchor, one for the pre-existing member and the other for the joining member, we get a tree with structure like T_2. Then, we recompute the resulting key tree to root to get the expanding key tree T_2. Note that unlike in leave rekeying, there is no need to change the sibling node key of the joining member's node key. Because even if the joining member is able to compute the root node key (y_1) of the blinded tree Y as discussed in section 2.1, knowing root node key (y_1) of Y reveals no information about the past group key (x_1). We finally construct the incremental key chain C by performing a tree product of the normalizing key chain C_1 and its counterpart in the expanding key tree T_2 like in Figure 18. In fact, the key server can directly compute C by recursively applying OWF g to c_6. In a word, to adding a leaf node x_{13} in a blinded HOFT Y is equivalent to performing a tree product of Y and the incremental key chain C.

(2) Group rekeying algorithm

When a member x_{13} joins the group, the key server first performs a tree blinding operation on the key tree X to obtain a blinded key tree Y, constructs a incremental key chain C based on Y, and then performs a tree product of Y and C to obtain the final updated key tree Y'. Now, the key server needs to broadcast the incremental key chain C to all pre-existing members by sending its leaf node c_6 encrypted with the old group key x_1, i.e., $\{c_6\}\_x_1$. It also need to supply the joining member x_{13} with the blinded keys of the siblings of the nodes in the path from its associated leaf node to the root of Y', after encrypting them with the joining member's unique secret key x_{13}, i.e., $\{g(y_6), g(y_7), g(y_2)\}\_x_{13}$. Each joining member uses those blinded keys to compute node keys in its path to the root including the current group key y_1'. Every pre-existing member can decrypt the broadcast message $\{c_6\}\_x_1$ to obtain the anchor c_6 of the incremental key chain C and restore the whole incremental key chain C by recursively applying OWF g to it. Therefore, they can update all the node keys they are entitled to by applying OWF g to them and then further change those updated node keys whose associated nodes are in the joining member's path to the root by multiplying them by the corresponding incremental keys. In a word, the rekeying message is "$\{c_6\}\_x_1$, $\{g(y_6), g(y_7), g(y_2)\}\_x_{13}$"

Consider a full and balanced OFT with n members (after the join). The key server only needs to encrypt and send 1 incremental key c_6 when a member joins the group. To supply the joining member with blinded keys, it needs to encrypt and send $\log_2 n$ blinded keys. In all, the key server needs to encrypt and send $\log_2 n+1$ keys which nearly half the broadcast size of original OFT scheme.

There is also cost associated with OWF and multiplication computations at the key server. When a member joins the group, the key server needs to compute the blinded key tree Y from original key tree X at first, and hence perform $2n$-2 OWF computations. Then the key server needs to compute the incremental key chain C, and hence perform $\log_2 n+1$ OWF computations and two multiplication computations. At last, the key server needs to compute a tree product of X and the incremental key chain C, and hence perform $\log_2 n$ multiplication computations. In all, the key server needs to perform $2n+\log_2 n-1$ OWF computations and $\log_2 n+2$ multiplication computations.

4.4 Adding multiple members in a bulk operation

As we discussed in section 4.2, there are situations (bursty behaviour, periodic group rekeying or batch group rekeying) where we need to process multiple joins in a bulk operation so that the broadcast size and computational effort can be substantially reduced.

For tree-based schemes (e.g. LKH and OFT), maintaining the balance of a key tree is very important for achieving a good performance. For a large dynamic group, leave changes in membership are usually unpredictable. Random deletions of leaf nodes are most likely to result in an imbalanced key tree. However, when members join, we have the freedom of choosing their positions in a key tree. This is our chance to remedy the imbalance of a key tree. Therefore, when adding nodes in a key tree, we always put that consideration in the first place. We define *imbalance degree* of a key tree as the value of the difference in depths between the shallowest and deepest leaves. We can set a threshold (e.g., 2) for the imbalance degree. If the imbalance degree of a key tree does not exceed the threshold, we add members evenly onto the blinded key tree. Otherwise, we use other methods to add them (see section 4.4.2).

4.4.1 Evenly adding multiple members onto a balanced key tree

(1) Evenly adding multiple leaf nodes by a tree product

After performing a tree blinding operation on current key tree X to obtain a blinded key tree Y, we find all the shallowest leaf nodes in Y, and perform the normalization and expansion operations on Y. Referring to Figure 19, to add members x_9, x_{11} and x_{14} onto the blinded key tree Y, we choose y_4, y_5 and y_7 as the nodes to be normalized. Based on the discussion in section 4.2.2 and 4.3, the normalization and expansion operations will be performed as in Figure 19. We can construct the incremental key tree T (see Figure 20) by performing a tree product of the normalizing key tree T_1 and its counterpart in the expanding key tree T_2. The key server can directly compute the incremental key tree T when traversing it in a postorder manner. To add members x_9, x_{11} and x_{14} onto the blinded key tree Y is equivalent to performing a tree product of Y and T.

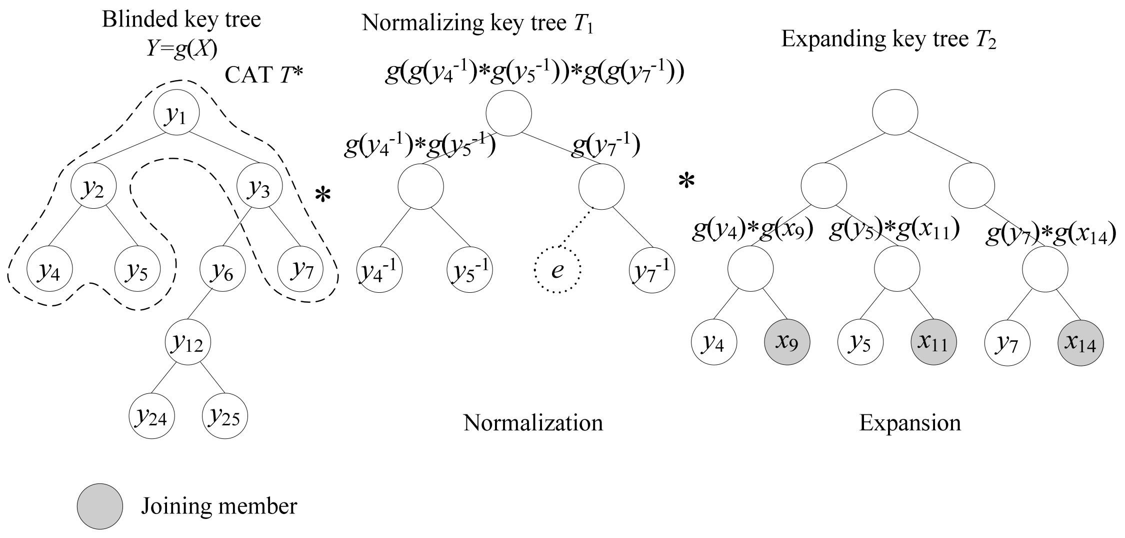


Fig. 19 Evenly adding multiple members

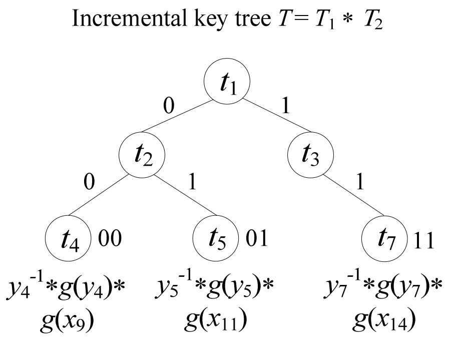


Fig. 20 an incremental key tree for even additions

(2) Group rekeying algorithm

After performing a tree blinding operation on the key tree *X* to obtain a blinded key tree *Y*, the key server constructs an incremental key tree *T*, and performs a tree product of *Y* and *T* to obtain an updated key tree *Y’*. The key server needs to communicate the changes in the key tree by broadcasting the incremental key tree *T*. Since an OFT is constructed in a bottom-up manner, it can be uniquely determined by all its leaf nodes. If the position of every leaf node can be uniquely identified, every member can reconstruct the whole HOFT in a bottom-up manner after receiving all leaf node keys. In an incremental key tree *T*, we identify the position of every leaf node by its *path ID*. Referring to Figure 20, the *path ID* of a leaf node is obtained as follows. Tracking the leaf node's path from the root to it, when visiting a left link (right link) from a node, we append a binary bit "0" (a "1") to an initially-null binary string. Until we visit the leaf node itself, we got a binary string called a *path ID*. For example, the position of t_5 in incremental key tree T is identified by its path ID "01". In fact, for all tree-based group rekeying schemes, legitimate members always need to extract the joining members or evictees' path IDs from a rekeying message to locate the changes in a key tree (see section 6). It follows from the above discussion that instead of broadcasting whole incremental key tree T, the key server only needs to send every leaf node keys of T in conjunction with their path IDs encrypted with the old group key, i.e., $\{t_4,00,t_5,01,t_7,11\}\_x_1$ for T. After decrypting this message, every pre-existing member can reconstruct the whole incremental key tree T. Therefore, they can accordingly update their own rekeyed node keys. In addition, the key server also needs to supply every joining member with blinded keys. In all, to add members x_9, x_{11} and x_{14}, the key server sends a rekeying message as "$\{t_4,00,t_5,01,t_7,11\}\_x_1$, $\{g(y_4), g(y_5'), g(y_3')\}\_x_9$, $\{g(y_5), g(y_4'), g(y_3')\}\_x_{11}$, $\{g(y_7), g(y_6), g(y_2')\}\_x_{14}$".

Consider a full and balanced OFT with n members (after l members join). When l members join the group, to communicate the incremental key tree T to all pre-existing members, the key server needs to encrypt and send l leaf node keys of T as well as l path IDs. In addition, to supply every joining member with their blinded keys, it needs to encrypt and send $l*\log_2 n$ blinded keys. In all, the key server needs to send $(l + l*\log_2 n)$ (Keys) + $l*\log_2 n$ (bits). There is also cost associated with OWF and multiplication computations at the key server. When l members join the group (suppose that l is a power of 2), firstly, the key server needs to compute a blinded key tree Y from original key tree X, and hence perform $2n$-$2l$-2 OWF computations. Recall that the size of an incremental key tree is denoted by S_L. Secondly, the key server needs to compute the incremental key tree, and hence perform S_L+l-1 OWF computations and S_L+2l multiplication computations. Lastly, the key server needs to compute a tree product of X and the incremental key tree T, and hence perform S_L multiplication computations. In all, the key server needs to perform $2n-l+S_L-3$ OWF computations and $2S_L+2l$ multiplication computations.

4.4.2 Adding multiple members by grafting

(1) Adding multiple leaf nodes by grafting

When the imbalance degree of the key tree exceeds a threshold (e.g., 2), we add multiple leaf nodes adjacently instead of evenly. Referring to Figure 21, if the imbalance degree of a key tree is i ($i \geq 3$) and the number of joining members, l is not bigger than 2^i-1 (also suppose that $l+1$ is a power of 2), we choose a shallowest node (e.g., y_2) as the grafting point and construct a new full and balanced homomorphic one-way subtree T^* (surrounded by a dashed line in Figure 21) which takes the shallowest leaf node y_2 and all the joining members x_9, x_{10}, and x_{11} as its leaf nodes. Then we graft T^* onto the main key tree at the grafting point. The normalization and expansion operations will be performed as in Figure 21. Lastly, we construct an incremental key chain C (see Figure 22) by performing a tree product of the normalizing key chain C_1 and its counterpart in the expanding key tree T_2. In a word, to add the joining members x_9, x_{10}, and x_{11} onto an imbalanced key tree is equivalent to firstly constructing a new subtree T^*, and then performing a tree product of Y and an incremental key chain C (i.e., grafting).

If the imbalance degree of a key tree is i ($i \geq$ threshold 3), but the number of joining members, l is bigger than 2^i-1, we just choose 2^i-1 members among them and then perform the grafting operation like above. If the imbalance degree of the updated key tree is still bigger than the threshold, we keep on performing the grafting operation using the remaining joining members until the imbalance degree is not bigger than the threshold or all the joining members have been added. Note that all the above grafting operations can be combined into a single bulk operation. If the imbalance degree of the updated key tree is not bigger than the threshold and there still remain joining members to be added, we use algorithm in section 4.4.1 to add remaining members evenly.

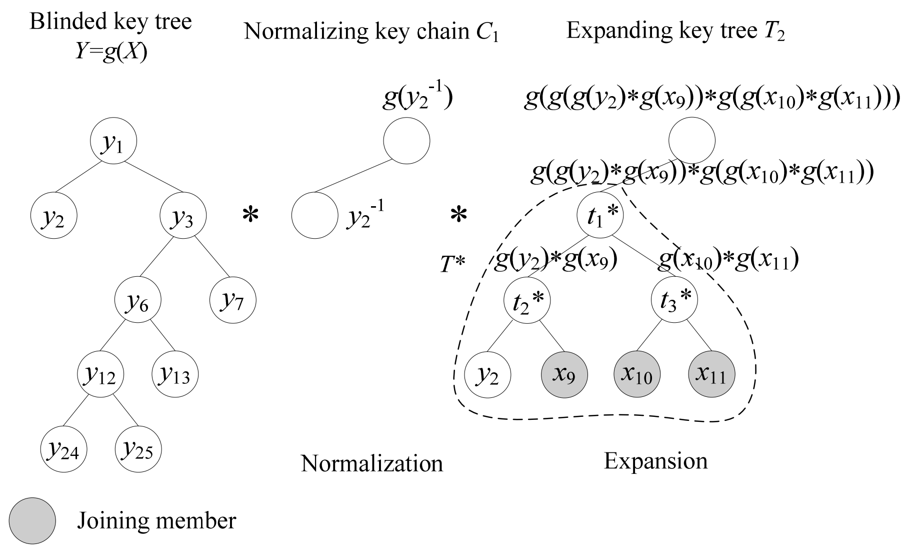


Fig. 21 adding multiple members by grafting

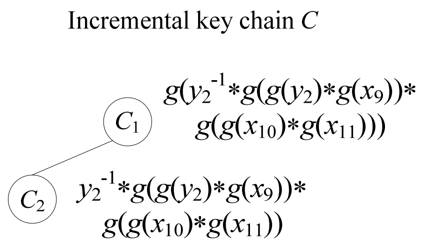

Incremental key chain C
$g(y_2^{-1}*g(g(y_2)*g(x_9))*$
$g(g(x_{10})*g(x_{11})))$
C_1
C_2
$y_2^{-1}*g(g(y_2)*g(x_9))*$
$g(g(x_{10})*g(x_{11}))$

Fig. 22 An incremental key chain for grafting

(2) Group rekeying algorithm

For simplicity, we only consider the scenario where the imbalance degree of a key tree is i ($i\geq3$) and the number of joining members, l is not bigger than 2^i-1. Firstly, the key server performs a tree blinding operation on the key tree X to obtain a blinded key tree Y, and then constructs a new subtree $T*$ taking the node key associated with the grafting point on Y and the joining members' node keys as its leaf nodes. Secondly, it constructs the incremental key chain C, and then performs a tree product of Y and C to obtain an updated key tree Y'. Instead of individually supplying every joining member with the blinded keys of the siblings of the nodes in its path to the root of *Y'*, the key server sends each blinded key in T^* encrypted with its sibling's unblinded key (if it exists) and the blinded keys associated with the siblings of nodes in the path from the grafting point to the root of *Y'* encrypted with their siblings' unblinded keys (these blinded keys are common to all joining members), i.e., $\{g(y_2)\}\_x_9$, $\{g(x_9)\}\_ y_2$, $\{g(x_{10})\}\_ x_{11}$, $\{g(x_{11})\}\_ x_{10}$, $\{g(t_2^*)\}\_ t_3^*$, $\{g(t_3^*)\}\_ t_2^*$, $\{y_2'\}\_t_1^*$. Every joining member can decrypt certain parts of this message to obtain the blinded keys of the siblings of the nodes in its path to the root of *Y'*. To broadcast *C*, it only needs to send *C*'s anchor encrypted with past group key, i.e., $\{c_2\}\_x_1$. Every pre-existing member can decrypt this message to obtain c_2, and thus restore the whole incremental key chain *C* by recursively applying OWF *g* to c_2 (see Figure 22).

Consider a full and balanced OFT with *n* members (after *l* members join) and $l+1$ is a power of 2. When *l* members join, to supply every joining member with their blinded keys, the key server needs to send and encrypt $2l-2+\log_2(n/(l+1))$ keys. That number equals S_L-1. To broadcast the incremental key chain *C* of length $\log_2(n/(l+1))$, the key server only needs to send and encrypt one key. In all, the key server needs to send and encrypt S_L keys.

There is also cost associated with OWF and multiplication computations at the key server. When *l* members join the group, the key server needs to compute the blinded key tree *Y* from original key tree *X* first, and hence perform $2n-2l-2$ OWF computations. To compute the subtree T^*, the key server needs to perform $2l$ OWF computations and *l* multiplications. It also needs to compute the incremental key chain *C* of length $\log_2(n/(l+1))$, and hence perform $\log_2(n/(l+1))$ OWF computations and one multiplication. To compute the updated key tree *Y'*, it needs to compute $\log_2(n/(l+1))$ multiplications. In all, the key server needs to compute $2n+\log_2(n/(l+1))-2$ OWF computations and $l+\log_2(n/(l+1))+1$ multiplications.

4.5 Handling mixed membership changes in a bulk operation

There are situations (bursty behaviour, periodic group rekeying or batch group rekeying) where we need to process mixed membership changes (i.e., joins and evictions mixed together) in one bulk operation. In that case, we firstly deal with all evictees by applying the corresponding leave rekeying algorithm in section 4.1 or 4.2 to current key tree X to obtain an updated key tree X_{new}. Secondly, we deal with all the joining members by applying the corresponding join rekeying algorithm in section 4.3 or 4.4 to key tree X_{new}. Therefore, it is a two-step bulk operation.

4.6 Automatic Rekeying

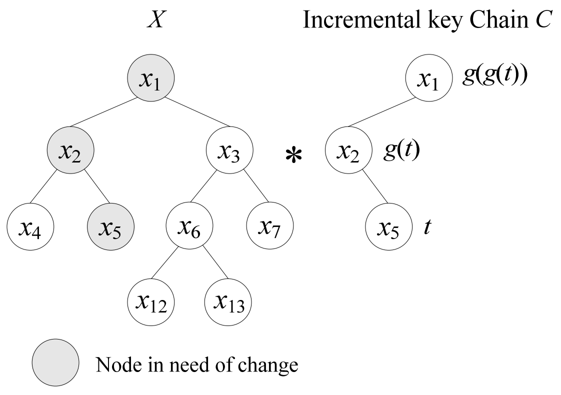


Fig. 23 Automatic Rekeying

There are situations where a special kind of group rekeying called *automatic rekeying* is required. For instance, besides group rekeying due to membership changes, the key server must spontaneously update the traffic encryption key (i.e., the group key here) at regular short intervals to thwart traffic analysis attack, especially when a group has no membership change during a long time interval. Since the type of group rekeying is not triggered by any membership change, we call it *automatic*. Because node keys in LKH are independent, automatic rekeying in LKH scheme is very simple. The key server only needs to generate a new group key and broadcast it encrypted with the old group key. As for OFT, we have to trigger rekeying in the key tree by rekeying the shallowest leaf node because there is a bottom-up functional dependency among keys. We illustrate the automatic rekeying algorithm based on HOFT by Figure 23.

The key server chooses a shallowest leaf node, e.g., x_5. If this node's path to root is of length l, the key server generates a random seed t, and then recursively apply OWF g to t for l times to generate an incremental key chain C of the same length l. It then updates the key tree by performing a tree product of X and C. Finally, it broadcasts t encrypted with old group key x_1, i.e., $\{t\}\_x_1$. Since every current member can decrypt the seed t and thus restore the whole incremental key chain C, it will be able to update its own node keys in need of change. Member x_5 computes its new leaf node key by multiplying x_5 by t. Consider a full and balanced HOFT with n members. The key server needs to encrypt and send 1 incremental key (seed). The key server also needs to compute the incremental key chain, and hence perform $\log_2 n$ OWF computations. In addition, to update the HOFT, it needs to perform $\log_2 n$+1modular multiplications.

In section 4.1, we mentioned that when the evictee's sibling is an internal node, the sibling subtree should be rekeyed before performing the contraction operation. We can use the automatic rekeying algorithm here to rekey the sibling subtree. That's another situation where automatic rekeying is required.

Whereas, for automatic rekeying based on original OFT, the key server needs to generate a new node key x_5' for member x_5 and encrypt all blinded node keys in member x_5's path to root with their corresponding siblings' unblinded keys. That is to say, the key server needs to send the following rekeying message: "$\{x_5'\}\_x_5$, $\{g(x_5)\}\_x_4$, $\{g(x_2)\}\_x_3$". Consider a full and balanced HOFT with n members. The key server needs to encrypt and send $\log_2 n$ blinded keys plus one new node key for the rekeyed member.

4.7 Comments on choosing OWF candidates for HOFT

Since OWF computations are intensive in our scheme, we must choose an efficient homomorphic OWF. Since it is impossible to construct a homomorphic OWF that is as fast as a pseudorandom hash function such as MD5 or SHA1, the only candidates for a HOWF in HOFT are homomorphic trapdoor functions like Rabin functions or RSA functions. Considering the computational performance, Rabin functions are the primary choice and RSA functions with small encryption exponent [19] the secondary choice. For Rabin functions, the public key parameters are generated as follows:

- Choose two large distinct primes p and q with $p \equiv q \equiv 3 \pmod 4$.
- Let $n = p*q$. Such number n is called *Blum number*.

Then the public key is n, and the private key is p and q. The set of all quadratic residues modulo n is denoted by Q_n. Rabin function is a trapdoor OWF mapping Z_n^* to Q_n defined as follows:

- For an integer $x \in Z_n^*$, compute $y = x^2 \bmod n$.

Inverting this function requires computing square roots modulo n. The latter problem is computationally equivalent to factoring n (in the sense of polynomial-time reduction) [14]. Since we only utilize the one-wayness of a trapdoor function, the trapdoor information (i.e., p and q) should be safely destroyed as soon as the public parameters are generated.

5 SECURITY ANALYSIS

Since Figure 4 used by Xu et al. to illustrate their proposition 1 helps to analyse all possible node keys that can be computed by a pair of colluding members, we also use it to analyse collusion attack on our scheme. We first discuss that an arbitrary pair of colluding members cannot compute any useful information about a group key not already known. Therefore, we need to consider all possible collusion scenarios as follows:

(1) Eviction-eviction scenario

In this scenario, we consider the collusion between a pair of members who was individually evicted in different leave rekeying operations. Suppose that in Figure 4, *A* is first evicted at time t_A and later *C* is evicted at time t_C. And we also suppose that there is no other membership changes between t_A and t_C. Since *C* knows the group key $\boldsymbol{K}_{\boldsymbol{Root}[t_A,t_C]}$ after *A* is evicted, the group forward secrecy is violated by collusion attack only when *A* and *C* can collude to compute the group key $\boldsymbol{K}_{\boldsymbol{Root}[t_C,-]}$(the dash here means the time of the next update of this key after t_C). Because *C* stays in the group longer than *A*, their knowledge about the shared node keys in the intersection of their paths (e.g., K_I and $K_{I'}$) and the siblings ($K_{R'}$ and $K_{R''}$) is no more than *C*'s knowledge about those keys. In addition, those shared node keys in the intersection of their paths are changed after *C* is evicted according to our leave rekeying algorithm. Therefore, with respect to these node keys, colluding with *A* does not help *C*. On the other hand, the unique knowledge held by *A* is about node keys in the subtree *B*, but this knowledge cannot be combined with *C*'s knowledge about node keys in subtree *D* to compute any node key. The only exception is *A*'s knowledge about *L* and *R* that can be combined with *C*'s to compute K_I (and consequently $K_{I'}$ and so on). However, according to our leave rekeying algorithms, *A*'s knowledge about *L* and *R* is $\boldsymbol{K}_{L[-,t_A]}$ (the dash here means the time of the last update of this key before t_A) and $\boldsymbol{K}_{R[t_A,t_C]}{}'$ (recall that *K'* is the blinded version of *K*), which is useless for computing new node key $\boldsymbol{K}_{I[t_C,-]}$. Therefore, in this scenario, the colluding members cannot compute any node key not already known.

In each leave rekeying algorithm of our scheme, the key server strictly controls access to the incremental key chain (or key tree) not only to prevent every evictee from accessing any part of it, but also to restrict every legitimate member to the incremental keys it is entitled to. Otherwise, if we grant every legitimate member full access to the incremental key chain (key tree), collusion between evictees is possible. Referring to Figure 4, since *C* has full access to the incremental key chain (or key tree) after *A* is evicted denoted by $\boldsymbol{iKC}_{t_A}$ that contains the incremental key corresponding to $\boldsymbol{K}_{L[-,t_A]}$ denoted by $\boldsymbol{i}_{K_L}{}^{t_A}$, collectively knowing $\boldsymbol{i}_{K_L}{}^{t_A}$ and $\boldsymbol{K}_{L[-,t_A]}$, *A* and *C* can collude to compute $\boldsymbol{K}_{L[t_A,-]} = \boldsymbol{i}_{K_L}{}^{t_A} * \boldsymbol{K}_{L[-,t_A]}$. While after *C* is evicted (suppose that after *C* is evicted, there still remain legitimate members associated with subtrees *B* and *D* respectively), $\boldsymbol{K}_{L[t_A,-]}$ will be used to encrypt the incremental key $\boldsymbol{i}_{K_R}{}^{t_C}$ according to our leave rekeying algorithm. Therefore, *A* and *C* can collude to gain access to $\boldsymbol{iKC}_{t_C}$ even after both of them are evicted. They can further compute the future group key $\boldsymbol{K}_{Root[t_C,-]}$. Thus, group forward secrecy is violated.

(2) Collusion between a pair of members evicted in a same bulk operation

Suppose that *A* and *C* are evicted at time t_{AC} in a same bulk operation. Since both K_L and K_R are changed after t_{AC}, *A* and *C* cannot collude to compute $\boldsymbol{K}_{I[t_{AC},-]}$. What's more, all the shared node keys in the intersection of their paths are changed after t_{AC}. Therefore, although their knowledge about the blinded keys associated with the siblings of those shared node keys is still effective for a certain interval after t_{AC}, they can never collude to compute any node keys including the future group key $\boldsymbol{K}_{Root[t_{AC},-]}$.

(3) Joining-joining scenario

We consider collusion between a pair of joining members individually added in different join rekeying operations. Suppose that *A* is added at time t_A and later *C* is added at time t_C in Figure 4. We also suppose that there is no other membership changes between t_A and t_C. The group backward secrecy is violated by collusion attack only when *A* and *C* can collude to compute the past group key $\boldsymbol{K}_{Root[-,t_A]}$. According to our join rekeying algorithm, before the leaf node of *A* or *B* is added onto a key tree, the key tree will be refreshed as a whole by a tree-blinding operation. If the key tree in an internal [t_A, t_C] is denoted by $\boldsymbol{tree}_{[t_A,t_C]}$, A and B's knowledge respectively relating to totally different key tree $\boldsymbol{tree}_{[t_A,t_C]}$ and $\boldsymbol{tree}_{[t_C,-]}$ (the dash here means that time of the next update of the key tree after t_C) cannot be combined with each other directly. Due to the one-wayness of tree-blinding operation, certain knowledge about blinded key tree $\boldsymbol{tree}_{[t_C,-]}$ can be deduced from knowledge about key tree $\boldsymbol{tree}_{[t_A,t_C]}$, but not vice versa. Therefore, the possible collusion only happens when A transforms its knowledge about $\boldsymbol{tree}_{[t_A,t_C]}$ into that about $\boldsymbol{tree}_{[t_C,-]}$ by performing an OWF computation on each node keys she knows, thus A and C may be able to collude to compute certain knowledge about $\boldsymbol{tree}_{[t_C,-]}$. But that is useless for computing the past group key $\boldsymbol{K}_{\boldsymbol{Root}[-,t_A]}$.

(4) Collusion between a pair of members added in the same operation

In Figure 4, suppose that A and C are jointly added at time t_{AC} in a join rekeying operation. According to our joining rekeying algorithm, before they are added, the key tree will be refreshed as a whole by a tree-blinding operation. That is to say, their knowledge acquired at the time of joining is about key tree $\boldsymbol{tree}_{[t_{AC},-]}$ and is useless for computing the past group key $\boldsymbol{K}_{\boldsymbol{Root}[-,t_{AC}]}$.

(5) Joining-eviction scenario

In Figure 4, suppose that A first joins the group at time t_A and later C is evicted at time t_C. If A and C collude, they trivially know the group key before A joins and after C is evicted, because C is in the group before A joins and A stays in the group after C is evicted. Therefore, colluding A and C can never compute any group key besides what they already know.

(6) Eviction-joining scenario

Suppose that A is evicted at time t_A and later C joins the group at time t_C. We also suppose that there is no other membership changes between t_A and t_C. We need to pinpoint the time when tree-blinding is performed and the time when *C*'s leaf node is added onto the blinded key tree. The former is denoted by t_{Blind} and the latter t_C. The group backward secrecy/group forward secrecy is violated by collusion attack only when *A* and *C* can collude to compute the group key $\boldsymbol{K}_{Root[t_A, t_{Blind}]}$. Because the key tree is refreshed as a whole by a tree-blinding operation before *C* joins, *A*'s knowledge cannot be combined with *C*'s knowledge. However, *A* can transform a part of her knowledge about the unblinded key tree $\boldsymbol{tree}_{[t_A, t_{Blind}]}$ (specifically, the blinded keys associated with the siblings of the node keys in her path to the root, Note that these keys are still effective for a certain interval after *A* is evicted) into that about the blinded key tree $\boldsymbol{tree}_{[t_{Blind}, t_C]}$ by performing an OWF computation on them. On the other hand, *C* acquires some blinded node keys of $\boldsymbol{tree}_{[t_{Blind}, t_C]}$ at the time of joining. Therefore, according to Xu's proposition 1, *A* and *C* can at most collude to compute certain nodes keys of $\boldsymbol{tree}_{[t_{Blind}, t_C]}$ (including the root node key $\boldsymbol{K}_{Root[t_{Blind}, t_C]}$) rather than any node key of the unblinded key tree $\boldsymbol{tree}_{[t_A, t_{Blind}]}$ (including the group key $\boldsymbol{K}_{Root[t_A, t_{Blind}]}$). Moreover, $\boldsymbol{K}_{Root[t_{Blind}, t_C]}$ never acts as a group key in our scheme.

Based on the above results, we can prove that an arbitrary collection of evicted members and joining members cannot collude to compute any group key not already known by using the same argument as in the proof of proposition 4.

6. COMPARISON WITH OTHER SCHEMES

We summarize relevant discussions in section 2.1 and section 4 to present a comprehensive comparison between our scheme and related schemes in Table 1, covering the following measures: collusion attack, broadcast size (in bits), key server's computational overhead and maximum computational cost of members. Ku and Chen's improvement on OFT and Xu et al.'s scheme are referred as Ku&Chen scheme and Xu scheme respectively. Since both schemes did not give the specific algorithms for processing multiple membership changes, we omit them from relevant comparison. Cost analysis for batch group rekeying using LKH is based on scheme proposed by Li et al. [13]. In Table 1, n is the number of members in the group, S_L is the size of the CAT when l changes in membership happen, and K is the size of a cryptographic key in bits. According to [21], the size of the incremental key tree S_L satisfies $2l+\log_2(n/l)-2<S_L<2l+l*\log_2(n/l)-1$. C_E, C_h, C_g, and C_M are respectively the computational cost of one evaluation of the encryption function E, one evaluation of pseudorandom hash function, one evaluation of trapdoor OWF g, and one modular multiplication. Note that for every entry associated with broadcast size, besides the cost for cryptographic keys, the additive log_2n (or $l*log_2n$) bits are used to locate the leaf node (or l leaf nodes) in a key tree that is associated with a changing member (or l changing members).

Table 1 Comparison with related scheme

(1) Adding a member

	LKH	OFT	Ku&Chen	Xu	HOFT
Collusion attack	no	yes	no	no	no
Broad. size(bits)	$2log_2n*K+log_2n$	$2log_2n*K+log_2n$	$2log_2n*K+log_2n$	$2log_2n*K+log_2n$ or $((log_2n)^2+2log_2n)*K+log_2n$	$(log_2n+1)*K+log_2n$
Server comp.	$2log_2n*C_E$	$2log_2n*(C_E+C_h)$	$2log_2n*(C_E+C_h)$	$2log_2n*(C_E+C_h)$ or $((log_2n)^2+2log_2n)*(C_E+C_h)$	$(log_2n+1)*C_E+(2n+log_2n-1)C_g+(log_2n+2)*C_M$
Max. Pre-existng mem. comp.	log_2n*C_E	$C_E+log_2n*C_h$	$C_E+log_2n*C_h$	$C_E+log_2n*C_h$	$C_E+log_2n*(2C_g+C_M)$

(2) Evicting a member

	LKH	OFT	Ku&Chen	Xu	HOFT
Collusion attack	no	yes	no	no	no
Broad. size(bits)	$2log_2n*K+log_2n$	$log_2n*K+log_2n$	$((log_2n)^2+log_2n)*K+log_2n$	$log_2n*K+log_2n$	$(log_2n+1)*K+log_2n$
Server comp.	$2log_2n*C_E$	$log_2n*(C_E+2C_h)$	$((log_2n)^2+2log_2n)*(C_E+C_h)$	$log_2n*(C_E+2C_h)$	$(log_2n+1)*C_E+log_2n*C_g+(log_2n+2)*C_M$
Max. mem. comp.	log_2n*C_E	$C_E+log_2n*C_h$	$log_2n*(C_E+C_h)$	$C_E+log_2n*C_h$	$C_E+log_2n*C_g+(log_2n+1)*C_M$

(3) Adding l members

	LKH	OFT	HOFT	
			Grafting	Evenly Adding
Collusion attack	no	yes	no	
Broad. size(bits)	$(2S_L-l)*K+l*log_2n$	$(S_L+l*log_2n)*K+l*log_2n$	$S_L*K+l*log_2n$	$(l+l*log_2n)*K+l*log_2n$
Server comp.	$(2S_L-l)*C_E$	$S_L*C_E+(2S_L-l)*C_h$	$(2l+1)*C_E+log_2(n/(l+1)*(C_g+C_M)+(2n-2)*C_g+(l+1)*C_M$	$(l+l*log_2n)*C_E+(2n-l+S_L-3)*C_g+(2S_L+2l)*C_M$
Max. mem. comp.	log_2n*C_E	$log_2n*(C_E+C_h)$	$C_E+log_2n/(l+1)*(C_g+C_M)+log_2n*C_g$	$l*C_E+log_2n*(2C_g+C_M)$

(4) Evicting *l* members

	LKH	OFT	HOFT	
			Pruning	Evicting sparsely-distributed evictees
Collusion attack	no	yes	no	
Broad. size(bits)	$(2S_L-l)*K+l*log_2n$	$(S_L+l-1)*K+l*log_2n$	$log_2(n/l)*K+l*log_2n$	$(S_L+l-1)*K+l*log_2n$
Server comp.	$(2S_L-l)*C_E$	$(S_L+l-1)*C_E+2S_L*C_h$	$log_2(n/l)*C_E+(2n+log_2(n/l)-1)*C_g+log_2(n/l)*C_M$	$(S_L+l-1)*C_E+(S_L-1)*C_g+2S_LC_M$
Max. mem. comp.	log_2n*C_E	$log_2n*(C_E+C_h)$	$log_2(n/l)*(C_E+C_g+C_M)$	$log_2n*(C_E+C_g)+(2log_2n+1)*C_M$

(5) Automatic rekeying

	LKH	OFT	HOFT
Broad. size(bits)	$K+log_2n$	$(log_2n+1)*K+log_2n$	$K+log_2n$
Server comp.	C_E	$(log_2n+1)*C_E+2log_2n*C_h$	$C_E+log_2n*C_g+(log_2n+1)*C_M$
Max. mem. comp.	C_E	$C_E+log_2n*C_h$	$C_E+log_2n*(2C_g+C_M)$

OFT based schemes have better leave-rekeying efficiency than LKH scheme. Another advantage of OFT-based schemes in processing single membership change over LKH is that pre-existing members have less computational overhead. This merit possessed by OFT has not been noticed by existing literatures and even by its inventors. Due to using trapdoor OWF Trapdoor OWF (e.g., Rabin function) instead of a much faster pseudorandom hash function (e.g., SHA1), HOFT has higher computational overhead than original OFT scheme, especially in conducting join rekeying. But it is worth trading off computation for collusion-freeness as well as lower communication overhead. What's more, for network based group communication, the communication efficiency is the main concern rather than computational efficiency within a computer, especially considering Moore's law. Among all collusion-free schemes (even including OFT), HOFT has best join-rekeying communication efficiency. Because in join rekeying based on HOFT, the key server only needs to broadcast all the leaf nodes of an incremental key tree (or key chain) rather than a whole CAT (or AC) (which has the same size as the incremental key tree) as required by the original OFT scheme. Ku & Chen scheme prevents collusion attack by changing all the keys known by an evictee on every member eviction, which require a broadcast of quadratic size. Whereas Xu scheme only performs additional key update when detecting a possible collusion between an evictee and a joining member, it has lower communication overhead than Ku & Chen scheme.

7. CONCLUSION AND FUTURE RESEARCH

In this paper, we introduce a new cryptographic construction — HOFT and two structure-preserving operations — tree product and tree blinding. We demonstrate that adding/deleting leaf nodes from a HOFT is equivalent to performing a tree product of the HOFT and an incremental key tree (key chain). Based on tree product and one-wayness of tree blinding operation, we propose a group rekeying scheme which not only prevents collusion attack on OFT scheme without compromising its leave-rekeying communication efficiency, but also improves its join-rekeying communication efficiency.

If we want to construct a homomorphic authentication tree based on Merkle authentication tree [15], the multiplication operation that is substituted for the concatenation operation should be non-commutative and a self-homomorphic OWF with respect to this non-commutative operation (e.g., Cantor pairing function) must be found. If such a homomorphic authentication tree does exist, adding, modifying or deleting a leaf node in a homomorphic authentication tree will become more efficient than in original Merkle authentication tree.

In our scheme, a HOFT is constructed in a bottom-up manner. We can construct a top-down homomorphic one-way function tree based on the binary hash tree proposed in MARKS [3]. In such a key tree, updating leaf node keys can be performed by tree product too. However, pseudo-randomness of the sequence of leaf node keys (each leaf node key serves as the group key for a short time period in MARKS) and key independency among them will be lost due to introduction of homomorphic OWF. Finding meaningful application for top-down HOFT may be of interest for future research.

So far, a lot of group key establishment protocols have been shown to be vulnerable to collusion attacks [10, 18, 3]. This triggered people towards using formal methods to design and verify group key establishment protocols. Rigorous analysis methodology for provable security of group key agreement schemes based on DDH (Decisional Deffie-Hellman) or CDH (Computational Deffie-Hellman) assumption has been established in the framework of modern cryptography [2]. Several works on formal verification of the security of the same kind of schemes have been done as well [8]. However, we don't see any relevant research result related to provable security as well as formal verification of tree-based group key distribution schemes like LKH or OFT scheme. In this paper, we only give a heuristic security analysis of HOFT. Developing formal methods to design and verify tree-based group key distribution protocols as well as rigorous analysis methodology for provable security of these protocols is the focus of ongoing work too.

ACKNOWLEDGEMENT

The authors thank Ronald L. Rivest for illuminating discussions.

FOOTNOTE 1

Most literature simply used terms *backward secrecy* and *forward secrecy* to refer to the above two security requirements. However, *forward secrecy* has its specific meaning in theoretic cryptography community. A protocol is said to have *perfect forward secrecy* (*or forward secrecy*) if compromise of long-term keys does not compromise past session keys [14]. For the sake of accuracy, we suggest the research circle to adopt the terms *group forward secrecy* and *group backward secrecy*).

REFERENCES

[1] D. Balenson, A. T. Sherman, and D. A. McGrew, Key Management for Large Dynamic Groups: One-Way Function Trees and Amortized Initialization. draft-irtf-smug-groupkeymgmt-oft-00.txt, IRTF work in progress, 2000

[2] E. Bresson, O. Chevassut, and D. Pointcheval, Provably-Secure Authenticated Group Diffie-Hellman Key Exchange. ACM Trans. on Information and System Security (TISSEC), vol. 10, no 3 (2007), Article No. 10.

[3] B. Briscoe, MARKS: Zero Side Effect Multicast Key Management Using Arbitrarily Revealed Key Sequences, Proceedings of First International Workshop on Networked Group Communication (NGC), Pisa, Italy, November 1999.

[4] Y. Challal and H. Seba. Group Key Management Protocols: A Novel Taxonomy, International Journal of Information Technology, vol. 2, no 2 (2005), pp. 105-118

[5] L. Cheung, J. A. Cooley, R. Khazan, and C. Newport, Collusion-Resistant Group Key Management Using Attribute-Based Encryption, The First International Workshop on Group-Oriented Cryptographic Protocols (GOCP) 2007.

[6] S. DEERING, Host Extensions for IP Multicasting, RFC 1112, 1989.

[7] J. Fan, P. Judge, AND M. Ammar, HySor: Group Key Management with Collusion-Scalability Tradeoffs Using a Hybrid Structuring of Receivers, Proceedings of the IEEE International Conference on Computer Communications Networks, 2002, pp. 196- 201.

[8] A. Gawanmeh, A. Bouhoula, and S. Tahar, Rank Functions based Inference System for Group Key Management Protocols Verification, International Journal of Network Security, Vol. 8, no 2 (2009), Science Publications, pp. 187-198.

[9] T. Hardjono and L. R. Donteti, Multicast and Group Security, Artech House, 2003.

[10] G. Horng, Cryptanalysis of a Key Management Scheme for Secure Multicast Communications, IEICE Trans. Commun., Vol. E85-B no 5 (2002), pp. 1050-1051.

[11] Y. Kim, A. Perrig, and G. Tsudik, Tree-based group key agreement, ACM Transactions on Information Systems Security, vol. 7 no l (2004), pp. 60-96

[12] W. C. Ku and S. M. Chen, An improved key management scheme for large dynamic groups using one-way function trees, Proceedings of International Conference on Parallel Processing Workshops, 2003, pp. 391-396.

[13] X. S. Li, Y. R. Yang, M. Gouda, and S. S. Lam, Batch Rekeying for Secure Group Communications, Proceedings of the Tenth International World Wide Web Conference, Hong Kong, China, 2001

[14] A. J. Menezes, P.C. van Oorschot, and S. A.Vanstone, Handbook of Applied Cryptography, Boca Raton: CRC Press, 1997.

[15] R. C. Merkle, Secrecy, Authentication, and Public-Key Cryptosystems. Technical Report No. 1979-1, Information Systems Laboratory, Stanford Univ., Palo Alto, Calif.

[16] D. Micciancio and S. Panjwani, Optimal communication complexity of generic multicast key distribution, IEEE/ACM Trans. on Networking, vol. 16, no 4 (2008), pp. 803-813.

[17] M. Rabin, Digitalized Signatures and Public-Key Functions as Intractable as Factorization. Technical Report: TR-212. MIT Laboratory for Computer Science, January 1979

[18] S. Rafaeli and D. Hutchison. A Survey of Key Management for Secure Group Communication, ACM Computing Surveys, vol. 35, no 3 (2003), pp. 309–329.

[19] R. Rivest, A. Shamir, and L. Adleman, A method for obtaining digital signatures and public-key cryptosystems, Communications of the ACM, vol. 21, no 2 (1978), pp. 120–126.

[20] A. T. Sherman, A Proof of Security for the LKH and OFC Centralized Group Keying Algorithms, NAI Labs Technical Report No. 02-043D, NAI Labs at Network Associates, Inc., Rockville, Md., Nov. 2002.

[21] A. T. Sherman and D.A. McGrew, Key establishment in large dynamic groups using one-way function trees, IEEE Transactions on Software Engineering, vol. 29, no 5 (2003), pp. 444 – 458.

[22] D. M. Wallner, E. J. Harder, and R. C. Agee, Key Management for Multicast: Issues and rchitectures, Internet Draft (work in progress), draft-wallner-key-arch-01.txt, Internet Eng. Task Force, Sept. 1998.

[23] C. K. Wong, M.G. Gouda, and S.S. Lam, Secure Group Communications Using Key Graphs, IEEE/ACM Transactions on Networking, Vol. 8, no 1 (Feb. 2000), pp. 16-30.

[24] X. Xu, L.Wang, A. Youssef, and B. Zhu, Preventing Collusion Attacks on the One-Way Function Tree (OFT) Scheme, Proceedings of Applied Cryptography and Network Security, 2007 vol. 4521 of LNCS, pp. 177–193.