Homomorphic One-Way Function Trees and Application in Collusion-Free Multicast Key Distribution

Jing Liu[1] and Bo Yang[2]

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

2. College of Informatics, South China Agricultural University, Guangzhou, People's Republic of China, 510642, byang@scau.edu.cn

Abstract. Efficient multicast key distribution (MKD) is essential for secure multicast communications. Although Sherman et al. claimed that their MKD scheme — OFT (One-way Function Tree) achieves both perfect forward and backward secrecy, several types of collusion attacks on it still have been found. Solutions to prevent these attacks have also been proposed, but at the cost of a higher communication overhead. In this paper, we prove falsity of a recently-proposed necessary and sufficient condition for existence of collusion attack on the OFT scheme by a counterexample and give a new necessary and sufficient condition for nonexistence of any type of collusion attack on it. We extend the notion of OFT to obtain a new type of cryptographic construction — *homomorphic one-way function tree* (HOFT). We propose two graph operations on HOFTs, tree product as well as tree blinding, and prove that both are structure-preserving. We provide algorithms for adding/removing leaf nodes in a HOFT by performing a tree product of the HOFT and a corresponding incremental tree. Employing HOFTs and related algorithms, we provide a collusion-free MKD scheme, which has not only the same leave-rekeying communication efficiency as the original OFT scheme, but also even better join-rekeying communication efficiency.

Key words. Multicast key distribution, One-way function tree, Homomorphism, Collusion

1. INTRODUCTION

Many emerging group-oriented applications, for instance, IPTV, DVB (Digital Video Broadcast), videoconferences, interactive group games, collaborative applications, and so on all require a one-to-many or many-to-many group communication mechanism. Allowing for efficient utilization of network bandwidth, IP multicast [1] is the best way to realize group communication in the Internet setting. One of the most efficient approaches to ensure confidentiality of group communications is employing a symmetric-key encryption scheme. But before the sender encrypts and transmits the data traffic over a group communication channel to a group of privileged users, a shared key called *group key* must be established among them. Compared to secure two-party key establishment, secure *group key establishment* in a dynamic group is a more challenging problem. Like the former, group key establishment can be subdivided into *group key distribution* (GKD) and *group key exchange* (or *group key agreement*). Group key exchange schemes are only suitable for small dynamic peer groups. Two parallel lines of research, commonly referred to as *broadcast encryption* (BE) [2] and *multicast key distribution* (MKD) (or multicast encryption), have been established to study the GKD problem but from different perspectives. This paper only focuses on MKD schemes. In contrast with stateless receivers in BE schemes, each receiver in MKD schemes is *stateful*, which means that they are allowed to maintain a personal state and make use of previously learned keys for decrypting current transmissions. Rather than tackling the general GKD problem as BE schemes, most MKD schemes aim to solve a more specific problem in the multicast encryption setting, called *immediate group rekeying*. Especially, for some security-sensitive multicast applications (e.g. military applications and classified conferences), the group key must be changed for every membership change. To prevent a new member from decoding messages exchanged before it joins a group, a new group 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* [3]. 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* [3]. 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 members. This process is referred to as *immediate group rekeying* in literature. Respectively, the rekeying process due to a joining membership change (resp. a departing membership change) is referred to as *join rekeying* (resp. *leave rekeying*). For large dynamic groups with frequent changes in membership, it is a big challenge to design a scalable MKD scheme. Since the late

1990s, a continuing research effort has been carried out, and today has seen a huge body of literature (See [4] for a good survey and a recent survey is [5]).

Among all generic multicast key distribution schemes in which rekey messages are built using traditional cryptographic primitives (symmetric-key encryption and/or pseudorandom generators), a class of schemes called *tree-based* schemes [6],[7],[8] are the most efficient ones to date in terms of communication overhead. They have a communication complexity of $O(\log_2 n)$ for a group size of n. A recent result by Micciancio et al. [9] has also confirmed that $\log_2 n$ is the optimal lower bound on the communication complexity of generic group key management schemes.

The *Logical Key Hierarchy* (LKH) scheme was independently proposed by Wong et al. [6] and Wallner et al. [7]. In the 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. A key associated with the internal node is shared by all members associated with its descendant leaf nodes. Every member is assigned to 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 current 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 the key server. When a member joins, the key server creates a leaf node whose position is nearest to the root for the joining member, and changes all the keys from this leaf node to the root. Join rekeying using LKH requires encryptions and transmission of $2\log_2 n$ keys by key server.

Another novel tree-based scheme is the *One-way Function Tree* (OFT) proposed by Sherman et al. [8], [10] (see section 2.1 for details). The OFT scheme nearly halves the communication overhead of LKH in case of leaving rekeying. However, Horng [11] showed that OFT is vulnerable to a particular kind of collusion attack (see section 2.2 for details). Soon after, Ku and Chen [12] also found new types of collusion attacks, and they proposed an improved scheme to prevent any collusion attack. But leaving rekeying using their approach requires 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. [13] showed that all the known attacks on OFT can be generalized to a kind of generic collusion attack. They also derived a necessary and sufficient condition for such an attack to exist and further proposed a scheme to prevent collusion attacks while minimizing the average broadcast size of rekeying message.

However their scheme requires a storage linear to the size of the key tree ($O(2n\text{-}1)$) and for large and dynamic groups, it still has a bigger broadcast size than LKH.

In this paper, we prove falsity of Xu et al.'s necessary and sufficient condition for existence of a collusion attack on the OFT scheme by a counterexample and give a new necessary and sufficient condition 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 one-way function and the exclusive-or mixing function in the original one-way function trees. We propose two tree operations — tree product and tree blinding for HOFTs and prove that both are structure-preserving. Tree blinding helps conceal information about each node secret of a key tree without compromising its inner structure. Then, we provide algorithms for adding/removing leaf nodes in a HOFT by performing a tree product of the HOFT and a corresponding incremental tree. Utilizing HOFTs and related algorithms, we design a collusion-free MKD scheme that has not only the same leave-rekeying communication efficiency as the original OFT scheme, but also even better join-rekeying communication efficiency. As already mentioned, two existing solutions [12], [13] to improve OFT have to trade off communication efficiency for collusion resistance.

The remainder of this paper is organized as follows. Section 2.1 gives a closer look at the OFT scheme. Section 2.2 reviews different kinds of collusion attacks on it. Section 2.3 introduces two solutions to prevent collusion attacks on OFT. In section 3, we prove the falsity of Xu et al.'s necessary and sufficient condition for a collusion attack on the OFT scheme to exist by a counterexample. We give a new necessary and sufficient condition for nonexistence of an arbitrary type of collusion attack. In sections 4, we introduce a new cryptographic construction – Homomorphic OFT and related algorithms. Section 5 presents a collusion-free MKD scheme based on HOFTs and related algorithms. Section 6 gives a thorough security analysis of our MKD scheme. Section 7 gives a comparison between our scheme and other related schemes. Section 8 concludes this paper and gives some topics for future research.

2. RELATED RESEARCH

2.1 Introduction to One-way Function Tree

The One-way Function Tree (OFT) scheme was proposed by Sherman, Balenson and McGrew [8],[10]. The idea of using a one-way function (OWF) in a tree structure originated from Merkle. In his report [14], 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 (i.e., the famous Merkle authentication tree).

We adopt the related terminology and formulations from [8]. A *key server* maintains a balanced binary key tree for a group. Each internal node v of the key tree is associated with a node secret x_v, a blinded node secret y_v and a node key K_v. The node secret of the root node is the group key. There exist two different pseudorandom functions f and g. The function f is used to compute each blinded node secret from its corresponding node secret, i.e., $y_v = f(x_v)$; The function g is used to compute each node key from its corresponding node secret, i.e., $K_v = g(x_v)$. The secret associated with an internal node is shared by all members associated with its descendant leaf nodes. The key server shares a secret called *leaf node secret* with every group member via the registration protocol. However, unlike in LKH, the key server does not send each member those node secrets along the path from its associated leaf node to the root. Instead, it supplies each member with the blinded node secrets associated with the siblings of the nodes in its path to the root. Each member uses these blinded node secrets and its leaf node secret to compute the other node secrets in its path to the root according to a functional relationship. A one-way function key tree is computed in a bottom-up manner using the pseudorandom function f and a bitwise exclusive-or operation denoted by '$\oplus$'. The node secret associated with an arbitrary internal node (*internal node secret* for short) is computed by applying the exclusive-or operation to the two blinded node secrets respectively associated with the two child nodes of the internal node (*child blinded node secrets* for short).

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

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

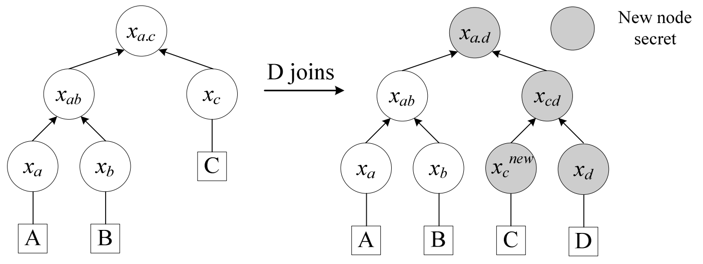

Fig. 1 Join rekeying in OFT

For example, the structure of an OFT is illustrated by Figure 1. *A* shares a leaf node secret x_a with the key server. When *A* joins the group, the key server sends *A* the following blinded node secrets: y_b, y_c. Therefore, *A* is able to compute all the node secrets in its path to the root in a bottom-up manner by sequentially computing $x_{ab} = f(x_a) \oplus y_b$, $x_{a.c}= f(x_{ab}) \oplus y_c$.

Figure 1 also illustrates the join rekeying in OFT. When *D* joins, the key server first chooses a leaf node nearest to the root, the leaf node associated with *C* in this case, and splits it into two nodes so as to create one for *D*. The other one is associated with the old member *C*. Like in LKH, all the node secrets in the path from the parent of *D*'s leaf node to the root need to be changed. To achieve this, the key server generates a new leaf node secret x_c^{new} for *C*, then computes all the node secrets that need to be changed in the same bottom-up manner as described above. Then to control each member's access to these new node secrets in the updated key tree, the key server constructs and transmits a rekeying message as: $\{y_{c.d}\}\_K_{ab}$, $\{x_c^{new}, y_d\}\_K_c$, $\{y_{ab}, y_c^{new}\}\_K_d$. Throughout this paper, we use $\{X\}\_Y$ to denote encryption of *X* with a key *Y* by using a symmetric encryption scheme. That is to say, in the updated key tree, all the rekeyed blinded node secrets are encrypted with their siblings' node keys. It is worth noting that x_c must be changed into x_c^{new}. Otherwise, in the updated key tree, *A* and *B* both know y_c, which violates the key distribution invariant. What is more, since the joining member *D* would be supplied with y_{ab} and y_c, it would be able to obtain the past group key $x_{a.c}$ by computing $x_{a.c} = y_{ab} \oplus y_c$, 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 $2\log_2 n+1$ blinded and unblinded node secrets when a member joins the group. In addition, the key server needs to compute $\log_2 n$ new secret keys in a bottom-up manner and $\log_2 n+1$ new blinded node secrets. That amounts to $2\log_2 n+1$ OWF computations (since the exclusive-or operation is very effective, it is reasonable to omit all of them). The joining member needs to perform $\log_2 n$ decryptions to extract all its $\log_2 n$ blinded node secrets from the rekeying message and then compute all the nodes keys along its path to the root in a bottom-up manner. For other members, if one has l node secrets in need of change, it only needs to perform one decryption to extract its single rekeyed blinded node secret, and then compute the new l node secrets by performing l OWF computations in a bottom-up manner. Whereas in the LKH scheme, this member needs to perform l decryptions to extract the l rekeyed node keys. Surprisingly, this computational advantage of OFT over LKH has been noticed neither by its inventors nor by existing literatures.

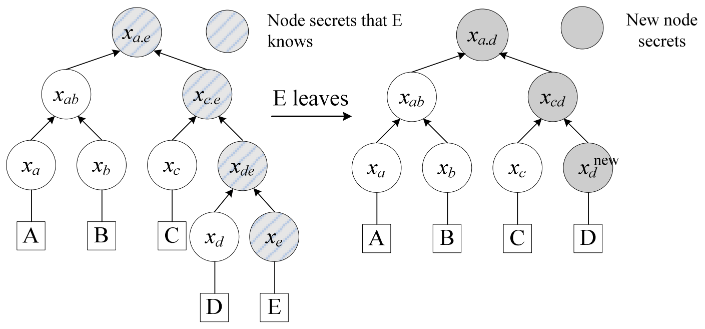

Fig. 2 Leave rekeying in OFT

Leave rekeying is depicted in Figure 2. When E leaves, all the node secrets in the path from the parent of E's leaf node to the root should be changed. If the sibling of the key node associated with E, i.e., x_d, is also a leaf node like in Figure 2, the key server only needs to change x_d into x_d^{new}, and sends x_d^{new} encrypted under K_d. Otherwise, the key server needs to pick one of the descendant leaf nodes of x_d (e.g. the leftmost one) and change its associated leaf node secret to trigger rekeying the subtree rooted at x_d. Then the key server replaces x_e's parent node x_{de} with E's rekeyed sibling node x_d^{new} (or rekeyed sibling subtree rooted at x_d^{new}). This process results in rekeying all the keys in the path from the departing member's parent node to the root in effect. Like in join rekeying, the key server needs to construct and transmit a rekeying message as: $\{y_{cd}\}\_K_{ab}$, $\{y_d^{new}\}\_K_c$, $\{ x_d^{new}\}\_K_d$. It is worth noting that x_d must be changed into x_d^{new}. Otherwise, in the old key tree, C knows y_d, which violates the key distribution invariant. And what's the more important is that since the evicted member E held y_d, y_c, and y_{ab}, E would be able to obtain the new group key $x_{a.d}$ by sequentially computing $x_{cd} = y_c \oplus y_d$, $x_{a.d}= y_{ab} \oplus f(x_{cd})$, 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 $\log_2 n+1$ blinded and unblinded node secrets when a member leaves the group. To compute new node secrets and blinded node secrets, the key server needs to perform $2\log_2 n+1$ OWF computations. If a legitimate member has l node secrets in need of change, it only needs to perform one decryption to extract its single rekeyed blinded node secret, and then compute the l new node secrets by performing l OWF computations in a bottom-up manner. Whereas in the LKH scheme, this member needs to perform l decryptions to extract the l rekeyed node keys.

2.2 Collusion attacks on the OFT scheme

In LKH, all the keys in a 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. In contrast, besides the logical subgroup relationship as LKH, there is also a functional dependency relationship among the node secrets in an one-way function tree. This relationship allows leave rekeying using OFT to save half of communication cost compared to that using LKH. However, the same relationship also renders it vulnerable to collusion attacks.

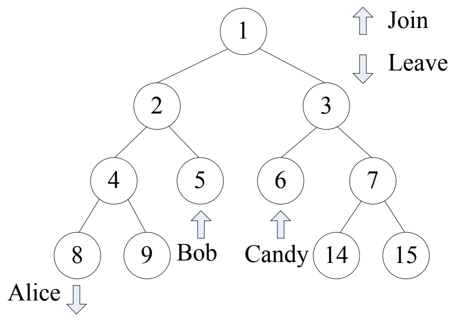

Fig. 3 Scenarios of collusion attacks on OFT

2.2.1 Horng’s attack

The first collusion attack on OFT attributes to Horng [11]. Referring to Figure 3, suppose that Alice, associated with node 8, leaves at time t_A, and later Candy joins the group at time t_C and is associated with node 6. We use $x_{i[t_A,t_C]}$ to denote the node secret associated with node i in the time interval between t_A and t_C. Suppose that there are no changes in group membership between time t_A and t_C. Since x_3 is not changed until Candy joins after the eviction of Alice, Alice holds its blinded version $y_{3[t_A,t_C]}$. Since x_2 is changed when Alice leaves, and then remains unchanged at least until Candy joins, Candy receives its blinded version $y_{2[t_A,t_C]}$ at the time of joining. Collectively knowing $y_{2[t_A,t_C]}$ and $y_{3[t_A,t_C]}$, Alice and Candy can collude to obtain the group key in the time interval [t_A, t_C] by computing $x_{1\,[t_A,t_C]} = y_{2[t_A,t_C]} \oplus y_{3[t_A,t_C]}$. Therefore, the OFT scheme fails to provide not only group forward secrecy against Alice but also group backward secrecy against Candy. Horng proposed two necessary conditions for such a collusion attack to exist: (1) the two colluding members namely A and C must leave and join at different subtree of the root respectively; (2) no group 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

Referring to Figure 3 again, the first kind of collusion attack given by Ku and Chen [12] can be described as follows. Suppose that Alice leaves at time t_A, and later Bob joins the group at time t_B and is associated with node 5. Also suppose that there are no changes in group membership between time t_A and t_B, for the same reason as above, Alice and Bob can collude to compute $x_{2[t_A,t_B]}$. Since x_3 remains unchanged during the time interval [t_A, t_B], Alice and Bob both hold its blinded version $y_{3[t_A,t_B]}$. Therefore, knowing $x_{2[t_A,t_B]}$ and $y_{3[t_A,t_B]}$, both can compute the group key $x_{1[t_A,t_B]}$. This attack does not satisfy the first necessary condition proposed by Horng.

The second kind of collusion attack given by them is described as follows. Suppose that Alice leaves 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 not only between time t_A and t_B, but also between time t_B and t_C. After the eviction of Alice, x_3 is not changed until Candy joins the group. Therefore, Alice holds its blinded version $y_{3[t_A,t_C]}$ even after her eviction. Since x_2 is changed when Bob joins the group, and then remains unchanged at least until Candy joins the group, Candy receives its blinded version $y_{2[t_B,t_C]}$ at the time of joining. Collectively knowing $y_{3[t_A,t_C]}$ and $y_{2[t_B,t_C]}$, Alice and Candy can collude to compute the group key $x_{1[t_B,t_C]}$ (note that $[t_B,t_C]\subset[t_A,t_C]$). This attack does not satisfy the second necessary condition proposed by Horng.

2.3 Improvements on the OFT scheme

In the OFT scheme, when a new member joins, it will be supplied with the blinded node secrets that were once used to compute the past group key. On the other hand, when a member leaves, it still holds the blinded node secrets that may be used to compute the future group key. It is possible for a pair of removed member and joining member to combine their knowledge together to compute a valid group key not already known.

From the above discussion, it is possible to devise a solution to prevent collusion attacks either by preventing a leaving member from taking away blinded node secrets that contain any information about the future group key or by supplying joining member with blinded node secrets that contain no information about the past group key. Each of the following two improvements on the OFT scheme is just aiming at one aspect to achieve collusion-resistance.

2.3.1 Ku and Chen’s improvement

Ku and Chen improve the OFT scheme by changing all the keys known by a leaving member. That is to say, when a member leaves, not only all the node secrets in its path to the root, but also all the blinded node secrets associated with the siblings of those nodes in that path must be changed. The additional updates of node secrets increase the broadcast size by $(\log_2 n)^2$ keys. The key server needs to encrypt and send $(\log_2 n)^2+\log_2 n+1$ keys in total.

Obviously, an opposite solution can be obtained by changing not only all the node secrets in the joining member’s path to the root as required by the original scheme, but also all the blinded node secrets associated with siblings to this path.

2.3.2 Xu et al.’s improvement

Xu et al. [13] observed that collusion between an evicted member and a joining member is not always possible and its success depends on a temporal relationship between them. It is not necessary to always change additional blinded node secrets as above unless a collusion attack is indeed possible. They proposed a stateful approach in which the key server tracks all evicted members and records all the knowledge held by them. Every time a new member joins, the key server checks against that knowledge to decide whether this joining member could have a successful collusion with any previous evicted member. For that purpose, their scheme has a storage requirement linear to the size of the key tree. Since additional blinded node secrets are changed 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 the LKH scheme for small to medium-scale groups, greater broadcast due to increasing number of collusion attacks renders their scheme less efficient than LKH for large dynamic groups.

In their paper [13], Xu et al. make three propositions to support the correctness of their scheme. They first consider a generic collusion attack on the OFT scheme (depicted in Figure 4). Before introducing their propositions, let us get familiar with some notations to be used. Suppose that A leaves at time t_A and C joins at a later time t_C. Let B, D, E, and F respectively denote the subtrees rooted at nodes *L*, *R*, *R'*, and *R"*. Let t_{DMIN}, t_{EMIN}, and t_{FMIN} denote the time of the first group key update after t_A that happens in *D*, *E*, and *F*, respectively. Let t_{BMAX}, t_{EMAX}, and t_{FMAX} denote the time of the last group key update before t_C that happens in *B*, *E*, and *F*, respectively.

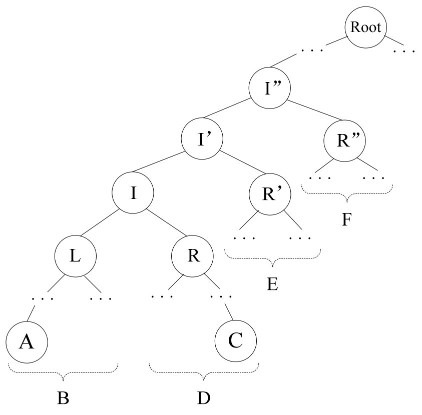

Fig. 4 A generic collusion attack on OFT

Xu's proposition 1**:** *For the OFT scheme, referring to Figure 4, the only node secrets that can be computed by A and C when colluding are:*

- x_I *in the time interval* $[t_{BMAX}, t_{DMIN}]$,
- $x_{I'}$ *in* $[t_{BMAX}, t_{DMIN}] \cap ([t_A, t_{EMIN}] \cup [t_{EMAX}, t_C])$,
- $x_{I''}$ *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.

In fact, it can be easily verified that all kinds of collusion attacks presented in section 2.2 are subsumed by this generic attack.

Xu's proposition 2**:** *A pair of colluding members A and C cannot compute any node secret which they are not supposed to know by the OFT scheme, if one of the following conditions holds*

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

This proposition confirms that the above generic collusion attack is the only pattern of two-party collusion. Based on these two propositions, the authors give the following sufficient and necessary condition for a collusion attack to exist.

Xu's proposition 3: *For the OFT scheme, an arbitrary collection of removed members and joining members can collude to compute some node secret not already known, if and only if the same node secret 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 node secret not already known, the colluding members must already know both child blinded node secrets of it. This claim is wrong, since the colluding members may not know those child blinded node secrets at first, but can collude to compute them.

3. AMENDMENT TO XU'S PROPOSITION 3

In this section, we first present an interesting counterexample that falsifies the necessity of Xu's proposition 3, and then propose a new necessary and sufficient condition for nonexistence of an arbitrary type of collusion attack on the OFT scheme.

3.1 A counterexample

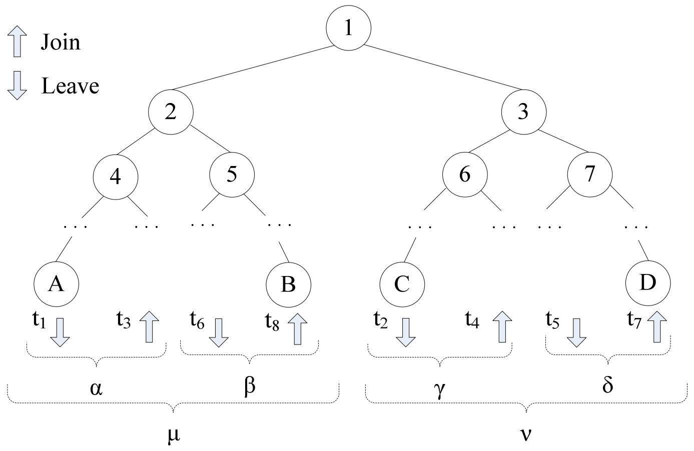

Fig. 5 A counterexample against Xu's proposition 3

We consider a collusion scenario depicted in Figure 5. Suppose that Alice (A) and Colin (C) leave the group at time t_1 and t_2, respectively, and Dean (D) and Bob (B) join the group at time t_7 and t_8, respectively. It is assumed that the chronological order of t_1, t_2, … , and t_8 corresponds with 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 at time t_3, t_4, t_5, and t_6, which happened in α, γ, δ, and β, respectively. Let $t_{\alpha MAX}^{X}$ denote the time of the last group key update before X joins the group that happens in α. Let $t_{\beta MIN}^{Y}$ denote the time of the first group key update after Y leaves the group that happens in β. Recall that x_v denotes the node secret associated with node v and y_v

denotes the blinded version of it. Moreover, $x_{v\,[t1,\,t2]}$ denotes the value of node secret x_v in the time interval $[t_1, t_2]$.

According to Xu's proposition 1, Alice and Bob can collude to compute x_2 in the time interval $[t^B_{\alpha MAX}, t^A_{\beta MIN}]$, i.e., $x_{2[t_3,t_6]}$; Colin and Dean can collude to compute x_3 in the time interval $[t^D_{\gamma MAX}, t^C_{\delta MIN}]$, i.e., $x_{3[t_4,t_5]}$. Thus, collectively knowing $x_{2[t_3,t_6]}$ and $x_{3[t_4,t_5]}$, Alice, Bob, Colin and Dean can collude to compute $x_{1[t_4,t_5]}$. However, we shall show that each pair of evicted member and joining member cannot collude to compute $x_{1[t_4,t_5]}$.

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

- x_2 in the time interval $[t^B_{\alpha MAX}, t^A_{\beta MIN}]$, i.e., $x_{2[t_3,t_6]}$,
- x_1 in the time interval $[t^B_{\alpha MAX}, t^A_{\beta MIN}] \cap ([t_1, t^A_{\nu MIN}] \cup [t^B_{\nu MAX}, t_8])$, but evaluation of this formula results in $[t_3, t_6] \cap ([t_1,t_2] \cup [t_7,t_8])=\varnothing$.

Thus, Alice and Bob cannot collude to compute $x_{1[t_4,t_5]}$. By the same argument, we can prove that for the rest of eviction-joining scenarios, i.e., the collusion between Colin and Dean, that between Alice and Dean, or that between Colin and Bob, $x_{1[t_4,t_5]}$ cannot be computed either. This counterexample thus falsifies the necessity of Xu's proposition 3.

3.2 A new necessary and sufficient condition

Proposition 3.1: *An arbitrary collection of evicted members and joining members cannot collude to compute any node secret not already known, if and only if an arbitrary pair of evicted member and joining member cannot collude to compute any node secret not already known.*

Proof: For an arbitrary node secret x_i in a key tree X, we use x_{2i} and x_{2i+1} to denote its left child and right child respectively. The blinded version of x_i is denoted by y_i. The necessity is trivial. We prove the sufficiency by contradiction. Suppose that there exists a set of evicted members and joining members (let S denote this set) who can collude to compute a new node secret $x_{i[t_1,t_2]}$. Let h denote the height of the subtree rooted at $x_{i[t_1,t_2]}$. Now, we prove that there exists a pair of members of S who can collude to compute a node secret not already known (may or may not be $x_{i[t_1,t_2]}$).

According to the OFT scheme, for the new node secret $x_{i[t_1,t_2]}$, there must exist $x_{2i[a,b]}$ and $x_{2i+1[c,d]}$,wherein $[a,b]\subseteq[t_1,t_2]$ and $[c,d]\subseteq[t_1,t_2]$ such that $x_{i[t_1,t_2]} = f(x_{2i[a,b]}) \oplus f(x_{2i+1[c,d]})$. Moreover, either of the following two conditions must be satisfied:

(1) there exists a pair of members of S who have already known $y_{2i[a,\ b]}$ and $y_{2i+1[c,\ d]}$, respectively. Therefore, they can collude to compute $x_{i[t_1,t_2]}$ by exclusive-oring $y_{2i[a,\ b]}$ and $y_{2i+1[c,\ d]}$.

(2) Otherwise, there exists a subset of members of S who can collude to compute either a new node secret $x_{2i[a,\ b]}$ or a new node secret $x_{2i+1[c,\ d]}$. Suppose it is $x_{2i[a,\ b]}$, then $x_{2i[a,\ b]}$ must be an internal node.

Now we can apply the same argument to $x_{2i[a,\ b]}$ again. In fact, the same argument can be repeated until we found a pair of members who can collude to compute a new node secret as in (1). Otherwise, due to the limited size of the key tree, after repeatedly applying the same argument h-1 times, we must meet a certain new internal node secret in a time interval that is a superset of $[t_1,\ t_2]$. And this new node secret is just the result of exclusive-oring its left leaf blinded node secret with its right one. Then, the two members respectively associated with these two leaf nodes are just the pair of member who can collude to compute a node secret not already known.

Whatever, we always find a pair of members of S who can collude to compute a node secret not already known. According to Xu's Proposition 2, the two colluding members must be a pair of evictee and joining member. That stands in contradiction to our hypothesis, and thus the sufficiency of *Proposition 3.1* follows. □

3.3 Further comments on collusion attacks

Unlike traditional cryptographic protocols (e.g., two-party key establishment protocols), group-oriented cryptographic protocols (group key establishment protocols, e-voting protocols, etc.) have an open number of group members. Malicious users could collude to sabotage any security target of these protocols. Therefore, preventing collusion attack is a paramount requirement when designing such protocols.

Although OFT was claimed to achieve perfect forward and backward secrecy by its inventors, collusion attacks on it still have been found. Because its inventors only consider collusion among removed members (or joining members), but unfortunately ignore the potential collusion between an evicted member and a joining member. Therefore, it is important to give a formal definition of *secure against collusion attacks* in computational security model to ensure it covers all possible patterns of collusion attacks. This work has been done by Panjwani [15].

4. HOMOMORPHIC ONE-WAY FUNCTION TREE

4.1 Definition

Before we give the definition of a homomorphic one-way function tree, let's review some basic mathematical concepts. A group G with its operation "$*$" is denoted by $(G, *)$. Given two groups $(G, *)$ and $(H, \cdot)$, a *group homomorphism* from $(G, *)$ to $(H, \cdot)$ is a function $f: G \rightarrow H$ such that for all u and v in G, it holds that $f(u*v) = f(u) \cdot f(v)$. One can easily deduce that a group homomorphism f maps the identity element e_G of G to the identity element e_H of H, and maps inverses to inverses in the sense of $f(u^{-1}) = f(u)^{-1}$. According to this definition, the *Rabin function* [16] and the *RSA function* [17] are both homomorphic.

Depending on one's viewpoint, homomorphism can be seen as a positive or negative attribute of a cryptosystem. The positive usage of homomorphism in cryptosystem was first proposed by Rivest et al. [18]. Once homomorphism is exploited by certain cryptosystem (encryption, digital signature, MAC, 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.

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

Definition 4.1 Homomorphic OFT — A *homomorphic OFT* (HOFT) over an Abelian group $(G, *)$ is a binary key tree that is computed using a self-homomorphic OWF f and the multiplicative operation "$*$" in a bottom-up manner as follows. For an arbitrary node secret x_i in a HOFT X, suppose that its left child and right child are denoted by x_{2i} and x_{2i+1} respectively, and we have $x_i = f(x_{2i}) * f(x_{2i+1})$.

4.2 Two structure-preserving operations on HOFTs

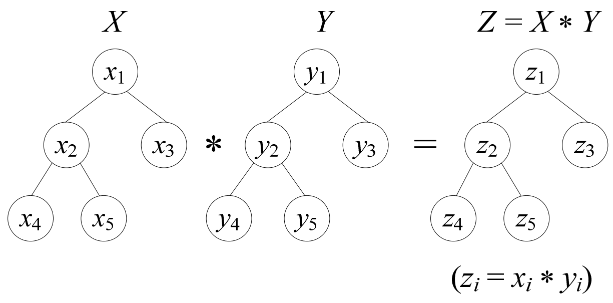

Fig. 6 Tree product

A binary operation (resp. unary operation) is said to be structure-preserving if the operation takes two HOFTs (resp. one HOFT) as inputs (resp. input) and outputs a HOFT. For convenience, we shall interchangeably use the same notation "x_i" or "y_i" to denote either a node itself or its associated node secret in section 4.

Definition 4.2 Tree product — Given two arbitrary HOFTs X and Y, both defined over an Abelian group (G, $*$), and having the same graph structure (i.e., same height and same number of leaf nodes), a tree product of X and Y, denoted by $X * Y$, is computed by multiplying their corresponding node secrets (see Figure 6).

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

Theorem 4.1: *Given two arbitrary HOFTs X and Y, both defined over an Abelian group (G, *), and having the same graph 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 secret $z_i \in Z$, we have

$z_i = x_i * y_i$ (Definition 4.2)

$= (f(x_{2i}) * f(x_{2i+1})) * (f(y_{2i}) * f(y_{2i+1}))$ (Definition 4.1, since X and Y are both HOFT)

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

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

$= f(z_{2i}) * f(z_{2i+1})$ (Definition 4.2).

Thus, Z is a HOFT according to Definition 4.1. □

It follows that tree product is *structure-preserving*.

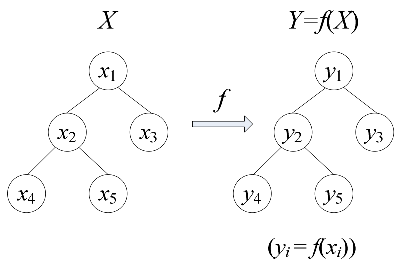

Fig. 7 Tree blinding

Definition 4.3 Tree blinding — For an arbitrary HOFT X defined over an Abelian group $(G, *)$ in conjunction with a homomorphic one-way function (HOWF) f, a *tree blinding* operation based on f maps X to another key tree Y, denoted by $Y = f(X)$. Y is computed by applying f to every node of X (see Figure 7). We call Y a *blinded tree* of X.

Theorem 4.2: *For an arbitrary HOFT X over (G, *), the blinded tree of X,* i.e., *f(X) is also a HOFT.*

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

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

Thus, Y is a HOFT according to Definition 4.1. □

It follows that tree blinding is also a structure-preserving operation. Due to one-wayness of f, tree blinding operation helps conceal information about each node secrets of a key tree without compromising its inner structure.

4.3 Adding/removing leaf nodes in HOFTs

In tree-based MKD schemes, adding or removing members correspond to adding or removing corresponding leaf nodes in a key tree. In this section, we provide algorithms for adding or removing leaf nodes in a HOFT X by performing a tree product of X and an *incremental tree*. First of all, we introduce a concept called *Combined Ancestor Tree* that was proposed by Sherman and McGrew [8]. For a set of evictees or joining members, the subtree consisting of all ancestors of their associated leaf nodes is called a *Combined Ancestor Tree* (CAT). Especially, an *ancestor chain* is an instance of a CAT that has one single leaf node. We only discuss the general case — adding/removing multiple leaf nodes in a HOFT. Algorithms for adding/removing a single leaf node can be easily derived from those given below.

4.3.1 Adding multiple leaf nodes in a HOFT

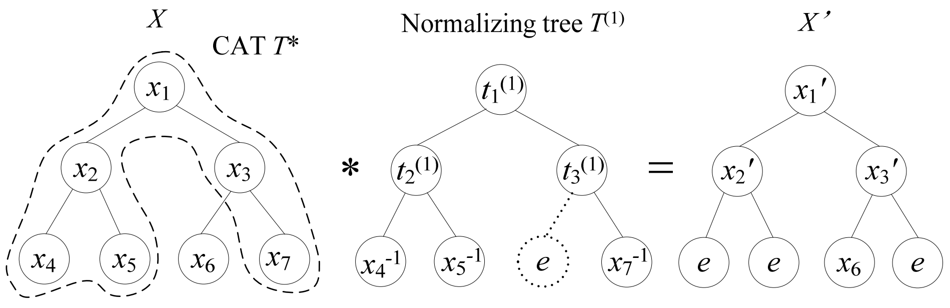

Fig. 8 Normalization for multiple additions

Adding multiple leaf nodes to a HOFT takes two steps. Referring to Figure 8, if we want to add leaf nodes x_9, x_{11}, and x_{15} to X respectively at x_4, x_5, and x_7, then the corresponding CAT is T^*. The first step called *normalization* (depicted in Figure 8) is in fact to perform a tree product of X and a normalizing tree $T^{(1)}$. The purpose is to turn all leaf node secrets of the CAT, i.e., T^* into identity node secrets (whose value is then identity element e of G). The normalizing tree $T^{(1)}$ is constructed from T^* by first replacing each leaf node secret of T^* with its corresponding inverse in G, and then computing all the other internal node secrets in a bottom-up manner as described in Definition 4.1.

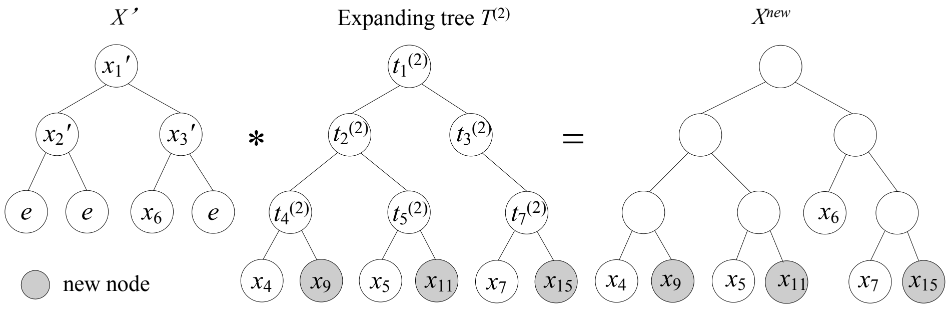

Fig. 9 Expansion for multiple additions

The second step called *expansion* (illustrated in Figure 9) is to perform a tree product of the output of the first step, i.e., X' and an expanding tree $T^{(2)}$. New leaf nodes are actually added onto X in this step. The expanding tree $T^{(2)}$ is also constructed from T^* by first creating two new child nodes for each leaf node x_i of T^* such that the node secret formerly associated with x_i is now associated with the left child of x_i, and a new node secret is associated with the right child of x_i, and then computing all the other internal node secrets in a bottom-up manner. The output of expansion is the final result — an updated key tree X^{new}.

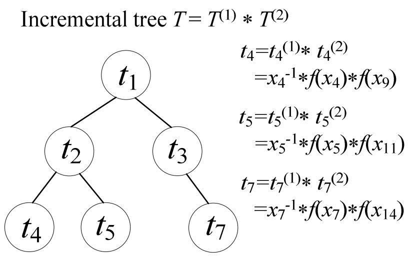
Incremental tree $T= T^{(1)} * T^{(2)}$
$t_4=t_4^{(1)} * t_4^{(2)}$
$=x_4^{-1}*f(x_4)*f(x_9)$
$t_5=t_5^{(1)} * t_5^{(2)}$
$=x_5^{-1}*f(x_5)*f(x_{11})$
$t_7=t_7^{(1)} * t_7^{(2)}$
$=x_7^{-1}*f(x_7)*f(x_{14})$
t_1
t_2
t_3
t_4
t_5
t_7

Fig. 10 An incremental tree for multiple additions

To simplify the two-step process, we introduce a concept called *incremental tree*. For X and X^{new}, the corresponding incremental tree T depicted in Figure 10 is obtained by performing a tree product of the normalizing tree $T^{(1)}$ and its counterpart in the expanding tree $T^{(2)}$.

Now, X^{new} can be obtained from X by firstly performing a tree product of its CAT T^* and the incremental tree T (suppose the output is T^{new}), secondly keeping all node secrets outside T^* unchanged, and thirdly for every leaf node t_i^{new} of T^{new}, creating two new child nodes for it such that the node secret formerly associated with x_i is now associated with the left child of t_i^{new}, and a new node secret is associated with the right child of t_i^{new}.

4.3.2 Removing multiple leaf nodes in a HOTF

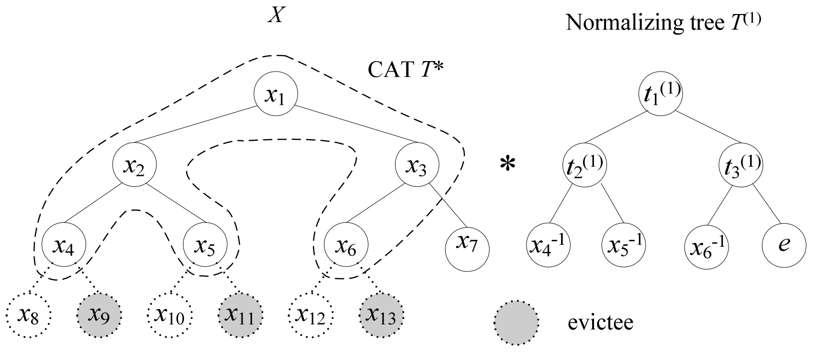

Fig. 11 Normalization for multiple removals

In Figure 11, we use a dotted circle (shaded or not shaded) to denote node that does not directly participate in a tree product computation, but whose position should be remembered. We also use a shaded and dotted node to denote a node to be removed. As illustrated in Figure 11, to remove x_9, x_{11} and x_{13} from a key tree X, the first step is just the same as the normalization step for adding multiple leaf nodes.

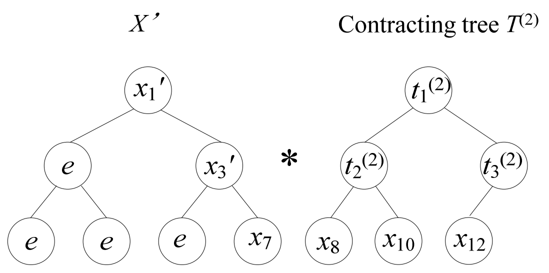

Fig. 12 Contraction for multiple removals

The second step called *contraction* as illustrated in Figure 12, is to perform a tree product of the output of the first step, i.e., X' and a contracting tree $T^{(2)}$. The contracting tree $T^{(2)}$ is constructed from T^* by first replacing each leaf node of T^* with its child in X not to be removed, and then computing all the other internal node secrets in a bottom-up manner.

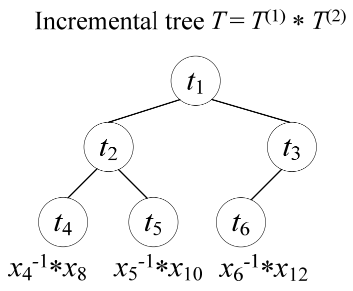

Fig. 13 An incremental tree for multiple removals

The incremental tree T for these multiple removals (see Figure 13) is obtained by performing a tree product of the normalizing tree $T^{(1)}$ and the contracting tree $T^{(2)}$. Now, the updated new key tree X^{new} can be obtained from X by firstly performing a tree product of its CAT T^* and T (suppose the output is T^{new}), secondly keeping all node secrets outside T^* unchanged, and thirdly removing both child nodes of each leaf node of T^* from X.

5 A COLLUSION-FREE MKD SCHEME BASED ON HOFTS

Employing algorithms provided in section 4, we are able to present a collusion-free MKD scheme. When members join or leave, all the node secrets on the corresponding CAT should be changed. The key server use algorithms provided in section 4.3 to construct an incremental tree (or chain), and update the key tree by performing a tree product of it and the incremental tree. After that, the key server needs to communicate all the changes in the key tree to group members by broadcasting the incremental tree (or chain) such that legitimate members can update their rekeyed node secrets and rekeyed blinded node secrets by multiplying those secrets by their corresponding incremental secrets. The essential task of a MKD scheme based on HOFTs is to strictly control access to the incremental tree (or chain) to ensure group forward secrecy and group backward secrecy.

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 simultaneously. The broadcast size and computational effort of multiple additions and removals can be substantially reduced by using a bulk operation that removes and adds multiple members simultaneously rather than repeatedly applying individual adding or removing operations. This reduction results from the fact that a set of individual operations may repeatedly change node secrets along common segments of the key tree. Even though we only present the algorithms for adding/removing multiple members. Algorithms for adding/removing a single member can be easily derived from those given below.

In the following passages, for any leaf node of a key tree, we shall interchangeably refer to it and its associated member for simplicity. Similar to OFT, we also use a pseudorandom OWF g to compute each node key K_v from its corresponding node secret x_v, i.e, $K_v = g(x_v)$.

5.1 Removing multiple members in a bulk operation

Taking Figure 11-13 as an example, to remove members x_9, x_{11} and x_{13} from the current key tree X, the key server uses the same algorithm provided in section 4.3.2 to compute a incremental tree T except that during contraction operation, it needs to replace leaf nodes of T^* (resp. x_4, x_5, x_6) with the rekeyed siblings of those evictees (resp. x_8', x_{10}', x_{12}'). Then the key server sends the blinded version of each incremental secret t_i except the root encrypted under the node key associate with the sibling of x_i^{new} in the updated key tree X^{new}, i.e., $\{f(t_2)\}\_K_3^{new}$, $\{f(t_3)\}\_K_2^{new}$, $\{f(t_4)\}\_K_5^{new}$, $\{f(t_5)\}\_K_4^{new}$ and $\{f(t_6) = t_3\}\_K_7^{new}$ (recall that $K_i^{new} = g(x_i^{new})$). In addition, the key server sends the sibling of each evictee a new node secret encrypted under its old value, i.e., $\{x_8'\}\_K_8$, $\{x_{10}'\}\_K_{10}$, $\{x_{12}'\}\_K_{12}$ (recall that $K_i = g(x_i)$). Note that $K_4^{new} = g(x_8')$, $K_5^{new} = g(x_{10}')$, $K_2^{new} = g(f(x_8')*f(x_{10}'))$, $K_3^{new} = g(f(x_{12}'))$, $K_7^{new} = g(x_7)$.

After every legitimate member receives the rekeying message, it extracts all blinded incremental secrets it is entitled to, and computes all incremental secrets it is entitled to in the incremental tree T in a bottom-up manner, and then updates its own rekeyed node secrets/ blinded node secrets by multiplying their old values by their corresponding incremental secrets. For example, member x_8 is able to extract blinded node secret $f(t_5)$ by sequentially decrypting $\{x_8'\}\_K_8$, $\{f(t_5)\}\_K_4^{new}$. Since it can directly compute $t_4 = x_4^{-1}*x_8'$, member x_8 now can compute $t_2 = f(t_4)*f(t_5)$ and then $x_2^{new}=x_2*t_2$. Now it can decrypt $\{f(t_3)\}\_K_2^{new}$ to obtain $f(t_3)$. In the end, it computes $t_1 = f(t_2)*f(t_3)$ and then the new group key $x_1^{\text{new}} = x_1*t_1$.

Since the incremental tree T has the same structure as the CAT T^*, we can compute the broadcast overhead by the size of CAT T^* denoted by S_L as in paper [8]. To remove l members, the key server needs to encrypt and send S_L+l-1 secrets. In addition, the key server needs to perform $2S_L$ modular multiplication computations and S_L-1 OWF computations.

5.2 Adding multiple members in a bulk operation

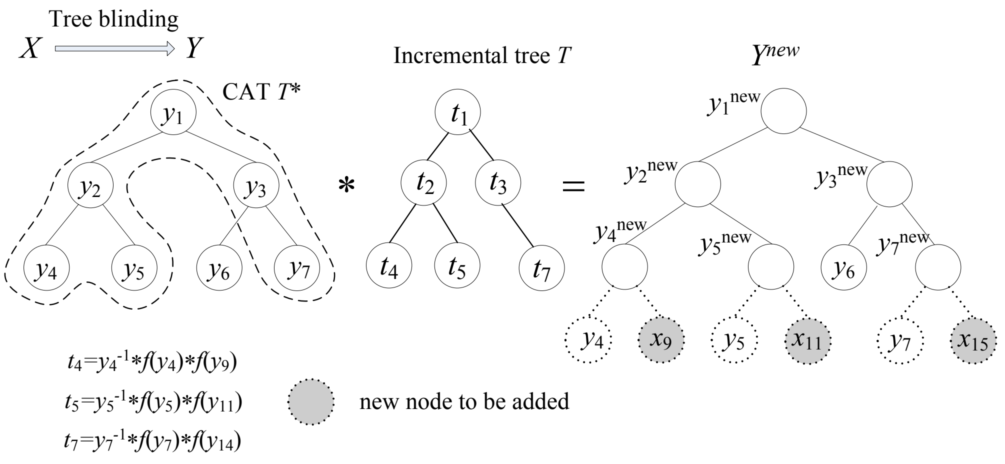

Fig. 14 Adding multiple members

Taking Figure 14 as an example, to add members x_9, x_{11}, and x_{15} to a key tree X, the key server first performs a tree blinding operation f on X to obtain a blinded tree $Y = f(X)$, then uses the algorithm provided in section 4.3.1 on Y to compute the corresponding incremental tree T. The key server sends every leaf node secrets of T encrypted under the $g(x_1)$ (recall that x_1 is the old group key), i.e., $\{t_4, t_5, t_7\}\_g(x_1)$. After decrypting this message, every old member can reconstruct the whole incremental tree T. Therefore, they can accordingly update their own rekeyed node secrets and rekeyed blinded node secrets. In addition, the key server also needs to supply every joining member with blinded node secrets it is entitled to. In a word, to add members x_9, x_{11} and x_{14} to the key tree X, the key server needs to send a rekeying message: $\{t_4, t_5, t_7\}\_g(x_1)$, $\{f(y_4), f(y_5^{new}), f(y_3^{new})\}\_K_9$, $\{f(y_5), f(y_4^{new}), f(y_3^{new})\}\_K_{11}$, $\{f(y_7), f(y_6), f(y_2^{new})\}\_K_{14}$.

Consider a full and balanced OFT with n members (after l members join). When l members join the group, the key server needs to encrypt and send $(l + l*\log_2 n)$ (Keys). In addition, the key server needs to perform $2n-l+S_L-3$ OWF computations and $2S_L+2l$ multiplication computations.

5.5 Comments on choosing OWF candidates for HOFT

Because OWF computations are intensive in our scheme, choosing an efficient homomorphic OWF is crucial for our scheme. The candidates can be homomorphic trapdoor functions like the *Rabin functions* or the *RSA functions* with small encryption exponent. Due to its computational efficiency superior to the *RSA functions*, the Rabin functions are preferred. 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) [16]. Since we only employ 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.

6 SECURITY ANALYSIS

In notation $x_{V[t^-,t]}$, we use t^- to denote the time of the last update of x_V before t; in $x_{V[t,t^+]}$, we use t^+ to denote the time of the first update of x_V after t. We first prove that an arbitrary pair of colluding members cannot compute any useful information about a group key not already known. For that purpose, we use Figure 4 to analyse all possible node keys that can be computed by a pair of colluding members. We need to consider all possible collusion scenarios as follows:

(1) Eviction-eviction scenario

Suppose that in Figure 4, A is 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 without loss of generality. The group forward secrecy is broken only when A and C can collude to compute any future group key after t_C. Because C stays in the group longer than A, their knowledge about the shared node secrets in the intersection of their paths (x_I, $x_{I'}$, and so on) and the siblings ($x_{R'}$, $x_{R''}$, and so on) is no more than C's. Therefore, for these node secrets, colluding with A does not help C. On the other hand, A's knowledge about blinded node secrets in the subtree *B* cannot be combined with *C*'s knowledge about blinded node secrets in the subtree *D* to compute any node secret, since no two nodes share a parent. The only exception is their knowledge about *L* and *R*. However, our MKD scheme updates the node secret x_R when *C* is evicted, *A* can at best know $y_{R[-,t_A]}$ and $y_{R[t_A,t_C]}$ which is useless to *C* (recall that y_v is the blinded version of x_v). In summary, two evicted member colluding in this case cannot compute any node key in addition to what is already known by the later-evicted node.

When performing leave rekeying in our MKD scheme, the key server strictly controls access to the incremental tree (or chain) not only to prevent every evictee from accessing any part of it, but also to restrict every legitimate member to the incremental secrets it is entitled to. If we grant every legitimate member full access to the incremental tree (chain) like in join rekeying, collusion between evictees would be possible. Considering the same scenario as above, suppose that *C* is granted full access to the incremental chain after *A* is evicted. Let $i^{t}_{x_v}$ denote the increment secret corresponding to x_v at time *t*. Then, collectively knowing $i^{t_A}_{x_L}$ and $x_{L[t_A^-,t_A]}$, *C* and *A* can collude to compute $x_{L[t_A,t_A^+]} = x_{L[t_A^-,t_A]} * i^{t_A}_{x_L}$. When *C* is evicted (suppose that after *A* and *C* are evicted, both subtree *B* and subtree *D* still contain at least one legitimate member), $K_{L[t_A,t_A^+]}$ is used to encrypt the blinded version of incremental secret $i^{t_C}_{x_R}$ in the rekeying message (recall that $K_{L[t_A,t_A^+]} = g(x_{L[t_A,t_A^+]})$). Therefore, *C* can extract $f(i^{t_C}_{x_R})$ from the rekeying message and compute $i^{t_C}_{x_{Root}}$ by repeatedly applying OWF *f* to $f(i^{t_C}_{x_R})$. Now, *C* can obtain the group key $x_{Root[t_C,t_C^+]}$ after t_C by computing $x_{Root[t_C,t_C^+]} = x_{Root[t_C^-,t_C]} * i^{t_C}_{x_{Root}}$. Thus, group forward secrecy is violated.

(2) Collusion between a pair of members who are evicted at the same time

Referring to Figure 4, suppose that both *A* and *C* are evicted at time t_{AC}. According to our scheme, they are prevented from accessing any blinded node secret of the incremental tree. Hence, they cannot update any node secret in CAT that is changed after t_{AC}. *A* and *C* cannot collude to compute any new node secret including future group key after t_{AC}.

(3) Joining-eviction scenario

Referring to 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 shared node secrets in the intersection of their paths (x_I, $x_{I'}$, and so on) and the siblings ($x_{R'}$, $x_{R''}$, and so on) 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. In addition, although *A* knows some blinded node secrets in the subtree *B* and *C* knows some blinded node secrets in the subtree *D*, these keys cannot be combined to compute any node key since no two nodes share a parent. In summary, *A* and *C* colluding in the joining-eviction case can never compute any new node secret (including group key) besides what they already know.

(4) Collusion between a pair of members who join at the same time

Suppose that both *A* and *C* join the group at time t_{AC}. The group backward secrecy is violated only when *A* and *C* can collude to compute any past group key before t_{AC}. According to our MKD scheme, before adding their associated leaf nodes onto a key tree, the key server will first apply the tree-blinding operation to the key tree to obtain a blinded tree. *A* and *C* receive their knowledge about the blinded tree at the time of joining. Due to the one-wayness of the OWF *f*, no node secret in the unblinded key tree (including group key) can be computed by *A* and *C* from their knowledge about the blinded tree.

(5) Joining-joining scenario

Suppose that *A* joins the group at time t_A and later *C* joins at time t_C. The group backward secrecy is violated only when *A* and *C* can collude to compute any past group key before t_A. For the similar reason as in case (4), the colluding *A* and *C* cannot compute any node secret in the unblinded key tree before t_A including the past group key.

(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. The group backward secrecy against *C* (resp. group forward secrecy against *A*) is violated by collusion attack only when *A* and *C* can collude to compute the group key $x_{Root[t_A,t_C]}$. After eviction, *A* holds knowledge about some blinded node secrets of $X_{[t_A,t_C]}$. But before adding the leaf node of *C* onto the key tree $X_{[t_A,t_C]}$, the key server will first transform it into a blinded tree $Y_{[t_A,t_C]} = f(X_{[t_A,t_C]})$ by using a tree-blinding operation. *C* receives knowledge about the blinded tree $Y_{[t_A,t_C]}$ at the time of joining. Therefore, knowledge about $X_{[t_A,t_C]}$ held by *A* can never be combined with knowledge about $Y_{[t_A,t_C]}$ held by *C* to compute any node secret in $X_{[t_A,t_C]}$ not already known (including the group key $x_{Root[t_A,t_C]}$) due to the one-wayness of the OWF *f*.

Thus, the above analysis follows that an arbitrary pair of colluding members cannot compute any node secret that may contribute to computing a group key not already known.

Furthermore, it is easy to 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 proposition 3.1.

7. COMPARISON WITH OTHER SCHEMES

We summarize related discussions in section 2 and section 5 to present a comparison between our scheme and related schemes, covering the following measures: collusion attack, broadcast size (in bits), key server's computational overhead and maximum member computational cost. The two solutions to improve OFT respectively proposed by Ku et al. and Xu et al are referred as Ku&Chen scheme and Xu scheme. 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. [19]. 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 or secret in bits. According to [8], the size of the incremental tree S_L satisfies $2l+\log_2(n/l)-2<S_L<2l+l*\log_2(n/l)-1$. C_E, C_h, C_f, and C_M denote the computational cost of one evaluation of the encryption function E, one evaluation of hash function, one evaluation of trapdoor OWF f, and one modular multiplication respectively.

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)	$2log_2n*K$	$2log_2n*K$	$2log_2n*K$	$2log_2n*K$ or $((log_2n)^2+2log_2n)*K$	$(log_2n+1)*K$
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 $((log_2n)^2+2log_2n)*(C_E+C_h)$	$(log_2n+1)*C_E+(2n+log_2n-1)C_f+(log_2n+2)*C_M$
Max. Pre-existng mem. comp.	log_2n*C_E	$C_E+log_2n*C_h$	$C_E+log_2n*C_h$	$C_E+log_2n*C_h$	$C_E+log_2n*(2C_f+C_M)$

(2) Removing a member

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

(3) Adding l members

	LKH	OFT	HOFT
Collusion attack	no	yes	no
Broad. size(bits)	$(2S_L - l)*K$	$(S_L + l*log_2 n)*K$	$(l + l*log_2 n)*K$
Server comp.	$(2S_L - l)*C_E$	$S_L*C_E + (2S_L - l)*C_h$	$(l + l*log_2 n)*C_E + (2n - l + S_L - 3)*C_f + (2S_L + 2l)*C_M$
Max. mem. comp.	$log_2 n*C_E$	$log_2 n*(C_E + C_h)$	$l*C_E + log_2 n*(2C_f + C_M)$

(4) Removing l members

	LKH	OFT	HOFT
Collusion attack	no	yes	no
Broad. size(bits)	$(2S_L - l)*K$	$(S_L + l - 1)*K$	$(S_L + l - 1)*K$
Server comp.	$(2S_L - l)*C_E$	$(S_L + l - 1)*C_E + 2S_L*C_h$	$(S_L + l - 1)*C_E + (S_L - 1)*C_f + 2S_L*C_M$
Max. mem. comp.	$log_2 n*C_E$	$log_2 n*(C_E + C_h)$	$log_2 n*(C_E + C_f) + (2log_2 n + 1)*C_M$

OFT based schemes have better leave-rekeying efficiency than the LKH scheme. Another advantage of OFT-based schemes in processing single membership change over LKH is that members without membership change have less computational overhead. It is worth noting that this merit possessed by OFT has been noticed neither by its inventors nor by existing literatures. Due to using a trapdoor OWF (e.g., Rabin function) instead of a much faster hash function (e.g., SHA1), HOFT has higher computational overhead than the original OFT scheme, especially in conducting join rekeying. But it is worth trading off computational cost for collusion-freeness as well as lower communication overhead. Among all measures used to evaluate a MKD scheme, communication complexity is probably the most important one, as it is the biggest bottleneck in current applications [20]. As a matter of fact, for network-based group communication, the communication efficiency is the main concern rather than computational efficiency within a computer, especially since modern computer has huge storage and very high CPU speeds. 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 tree (or key chain) rather than a whole CAT (which has the same size as the incremental 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 secret update when detecting a possible collusion between an evictee and a joining member, it has lower communication overhead than Ku & Chen scheme.

8. CONCLUSION AND FUTURE RESEARCH

In this paper, we introduce a new cryptographic construction — HOFT. Employing HOFTs and related algorithms, we propose a MKD scheme which not only prevents collusion attack on the OFT scheme without compromising its leave-rekeying communication efficiency, but also improves its join-rekeying communication efficiency. Finding other meaningful application of HOFT is a worthwhile topic.

To obtain a homomorphic authentication tree extended from a Merkle authentication tree, 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. Adding, modifying or removing a leaf node in a homomorphic authentication tree will be more efficient than in original Merkle authentication tree. We leave as an open problem the existence of homomorphic authentication tree.

In our scheme, a HOFT is constructed in a bottom-up manner. We also can construct a top-down homomorphic one-way function tree based on the binary hash tree proposed by Briscoe in [21]. In MARKS, each leaf node in a binary hash tree serves as a group key in a corresponding time slice in the group's lifetime. In a top-down HOFT, updating leaf node secrets can also be performed by tree product too. However, the sequence of leaf node secrets lacks pseudo-randomness and key independency among them due to introduction of homomorphic OWF. Finding meaningful application for top-down HOFT is a future research topic.

So far, a few of group key distribution protocols – OFT, MARKS [21] , the algorithm proposed by Chang et al. [22], LORE [23] have been shown to be vulnerable to collusion attacks. Developing rigorous analysis methodology and formal verification method for these protocols are necessary. For group key exchange protocols, rigorous analysis methodology for their provable security based on DDH (Decisional Deffie-Hellman) or CDH (Computational Deffie-Hellman) assumption has been established [24],[25],[26]. Works on formal verification of group key exchange protocols have been done as well [27],[28]. In contrast, we don't see any research result related to formal verification of group key distribution protocols. To the best of our knowledge, the only result related to provable security of group key distribution protocols is [15]. In this work, Panjwani proves that a corrected version of LKH is provably-secure against adaptive adversaries in computational security model. We can foresee that proving that OFT is secure against adaptive adversaries would be more difficult than LKH due to the functional dependency among secrets in a one-way function tree. Developing formal methods to verify group key distribution protocols as well as rigorous analysis methodology for provable security of OFT and HOFT is the focus of ongoing work.

ACKNOWLEDGEMENT

The authors wish to thank Ronald L. Rivest for valuable discussion.

REFERENCE

[1] S. Deering, “Host Extensions for IP Multicasting,” *RFC 1112*, 1989.

[2] A. Fiat, and M. Naor, "Broadcast encryption," *Advances in Cryptology - Crypto’93*, Lecture Notes in Computer Science D. R. Stinson, ed., USA-Santa Barbara, California: Springer-Verlag, August 1993.

[3] L. Cheung, J. A. Cooley, R. Khazan *et al.*, “Collusion-Resistant Group Key Management Using Attribute-Based Encryption,” in First International Workshop on Group-Oriented Cryptographic Protocols (GOCP), 2007.

[4] S. Rafaeli, and D. Hutchison, “A survey of key management for secure group communication,” *ACM Computing Surveys,* vol. 35, no. 3, pp. 309-329, Sep, 2003.

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

[6] C. K. Wong, M. Gouda, and S. S. Lam, “Secure group communications using key graphs,” *IEEE-ACM Transactions on Networking,* vol. 8, no. 1, pp. 16-30, Feb, 2000.

[7] D. M. Wallner, E. J. Harder, and R. C. Agee, "Key Management for Multicast: Issues and rchitectures," *Internet Draft*, Internet Eng. Task Force, 1998.

[8] 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, pp. 444-458, May, 2003.

[9] D. Micciancio, and S. Panjwani, “Optimal communication complexity of generic multicast key distribution,” *IEEE-ACM Transactions on Networking,* vol. 16, no. 4, pp. 803-813, Aug, 2008.

[10] D. Balenson, D. McGrew, and A. Sherman, “Key management for large dynamic groups: One-way function trees and amortized initialization,” *draft-irtf-smug-groupkeymgmt-oft-00.txt, Internet Research Task Force*, August 2000.

[11] G. Horng, “Cryptanalysis of a Key Management Scheme for Secure Multicast Communications,” *IEICE Transactions on Communications,* vol. E85-B, no. 5, pp. 1050-1051, 2002.

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

[13] X. Xu, L. Wang, A. Youssef *et al.*, “Preventing Collusion Attacks on the One-Way Function Tree (OFT) Scheme,” in Proceedings of the 5th international conference on Applied Cryptography and Network Security, Zhuhai, China, 2007, pp. 177-193.

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

[15] S. Panjwani, “Tackling adaptive corruptions in multicast encryption protocols,” in Proceedings of the 4th conference on Theory of cryptography, Amsterdam, The Netherlands, 2007, pp. 21-40.

[16] M. O. Rabin, *Digitalized signatures and public-key functions as intractable as factorization*, Cambridge: Massachusetts Institute of Technology, Laboratory for Computer Science, 1979.

[17] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” *Commun. ACM,* vol. 21, no. 2, pp. 120-126, 1978.

[18] R. Rivest, L. Adleman, and M. Dertouzos, “On Data Banks and Privacy Homomorphisms,” in Foundations of Secure Computation, 1978, pp. 169-180.

[19] X. S. Li, Y. R. Yang, M. G. Gouda *et al.*, “Batch rekeying for secure group communications,” in Proceedings of the 10th international conference on World Wide Web, Hong Kong, Hong Kong, 2001, pp. 525-534.

[20] R. Canetti, T. Malkin, and K. Nissim, "Efficient communication-storage tradeoffs for multicast encryption," *Advances in Cryptology - Eurocrypt'99*, Lecture Notes in Computer Science J. Stern, ed., pp. 459-474, 1999.

[21] B. Briscoe, “MARKS: Zero side effect multicast key management using arbitrarily revealed key sequences,” in Proceedings of Networked Group Communication 1999, pp. 301-320.

[22] I. Chang, R. Engel, D. Kandlur *et al.*, “Key management for secure lnternet multicast using Boolean function minimization techniques,” in INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE, 1999, pp. 689-698 vol.2.

[23] J. Fan, P. Judge, and M. H. Ammar, “HySOR: group key management with collusion-scalability tradeoffs using a hybrid structuring of receivers,” in Proceedings of Eleventh International Conference on Computer Communications and Networks 2002, pp. 196 - 201.

[24] E. Bresson, M. Manulis, and J. Schwenk, “On security models and compilers for group key exchange protocols (Extended abstract),” *Advances in Information and Computer Security, Proceedings,* vol. 4752, pp. 292-307, 2007.

[25] E. Bresson, and M. Manulis, “Contributory group key exchange in the presence of malicious participants,” *IET Information Security,* vol. 2, no. 3, pp. 85-93, Sep, 2008.

[26] E. Bresson, O. Chevassut, and D. Pointcheval, “Provably secure authenticated group Diffie-Hellman key exchange,” *ACM Transactions on Information and System Security,* vol. 10, no. 3, pp. -, Jul, 2007.

[27] 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, pp. 187-198, 2009.

[28] A. Gawanmeh, S. Tahar, and L. Ayed, “Event-B based invariant checking of secrecy in group key protocols,” in 33rd IEEE Conference on Local Computer Networks, 2008, pp. 950-957.