Preventing Collusion Attacks on the One-Way Function Tree (OFT) Scheme

Xuxin Xu

A Thesis
in
The Department
of
Concordia Institute for Information Systems Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Applied Science (Information Systems Security) at
Concordia University
Montreal, Quebec, Canada

October 2007

Your file Votre référence
ISBN: 978-0-494-40904-6
Our file Notre référence
ISBN: 978-0-494-40904-6

Abstract

Preventing Collusion Attacks on the One-Way Function Tree (OFT) Scheme

Xuxin Xu

The group key management problem is an important research area in multicast communication security field. The one-way function tree (OFT) scheme proposed by Balenson *et al.* is widely regarded as an efficient key management solution for multicast communication in large dynamic groups. Following Horng's claim that the original OFT scheme was vulnerable to a collusion attack, Ku *et al.* proposed a solution to prevent the attack. The solution, however, requires to broadcast about $h^2 + h$ (h is the height of the key tree) keys for every eviction operation, whereas the original OFT scheme only requires about h keys. This modified OFT scheme thus loses a key advantage that the original OFT has over the logical key hierarchy (LKH) scheme, that is a halving in broadcast size.

In this thesis, we revisit collusion attacks on the OFT scheme. We generalize the examples of attacks given by Horng and Ku *et al.* to a generic collusion attack on OFT, and derive necessary and sufficient conditions for such an attack to exist. We then show a solution for preventing collusion attacks while minimizing the average broadcast size. Our simulation results show that the proposed solution outperforms LKH in many cases and it has an improved performance over Ku and Chen's scheme. This performance gain is due to the fact that our method requires additional key updates only when attacks become possible.

We also extend our analysis for the case where only the root (group) key needs to be protected. Using this relaxed security assumption, a more efficient key updating scheme is proposed and analyzed. Our simulation results confirm that if only the group key needs to be protected, the proposed OFT-based scheme outperforms LKH in all cases.

Acknowledgements

First, I would like to express my deepest gratitude to advisors, Dr. Amr Youssef and Dr. Lingyu Wang. They gave me their encouragement and advice without any reserve. This thesis could not be completed without their help. Their scientific attitude and enthusiasm to academic research work will always remain as my model.

I am also indebted to my friends, Xiaochun Yang and Zhao (Eric) Yin, who helped in various ways in carrying the research discussed in this thesis.

Last but not least, my thanks go to my parents, my wife and my brothers. They provided whole-hearted support, not only during my graduate study, but also throughout my entire life. Their generous help allowed me to fully dedicate to work and made my study a comfortable process.

Thank you all.

Table of Contents

List of Figures

List of Tables

List of Notations

ACL	Access control lists
AKD	Area key distributors
AMESP	Application layer MESP
CCNT	Cryptographic context negotiation template
ESP	Encapsulating security payload
GKC	Group key controller
GSA	Group security association
GSI	Group security intermediary
GSPT	Group security policy token
IETF	Internet engineering task force
LKH	Logical key hierarchy
MESP	Multicast ESP
NTP	Network time protocol
OFT	One-way function tree

Chapter 1

Introduction

1.1 Motivation

The past few years have witnessed the development of many applications which require one-to-many or many-to-many group communications. Examples for these applications include stock quote streaming, multimedia conferencing, video distribution, web caching, and software distribution. Using multicast, the sender needs only to send one copy of data to the intended multiple receivers instead of sending multiple almost identical copies as it does in unicast communications. Thus, the multicast dramatically reduces the computational cost at the sender as well as the volume of the network traffic [16].

Despite the progress made in multicast research over the past few years, multicast deployment is not as widely spread as one might have expected. One of the critical obstacles for widespread deployment of multicast applications is the associated security concerns in the multicast communication environment.

1.2 Multicast security Problems

Multicast security has three main problem areas: secure multicast data handing, management of keying material, and multicast security policy [16]. In what follows, we briefly review each one of the above problems.

Secure multicast data handing area covers the secure data transmission from the sender to the receiver. Source authentication and data integrity are required to make sure that the data comes from the claimed sender and it was not changed during the transit. The simplest solution for source authentication is to digitally sign each packet but its performance is unacceptable because it is computationally expensive and involves extra communication overhead for each packet. The Internet engineering task force (IETF) has proposed a few variants of Encapsulating Security Payload (ESP) called multicast ESP (MESP) [5] and application layer MESP (AMESP) [6] to accommodate source authentication schemes for multicasting. MESP works in the network layer, whereas AMESP operates in the application layer. Group authentication can assure that the sender is a group member but cannot guarantee that the data had never been modified since it left the data sender [20].

Multicast security policy focuses on the following problems: policy creation, high-level policy translation, secure group policy components, and policy representation. Multicast security policies provide the rules of operation for the above two multicast security problem areas: management of keying material and multicast data handling. In small interactive groups, multicast security policies may be negotiated [12] but negotiation does not converge in large groups. Alternatively, the group controller (owner) may distribute and enforce policies. To create and translate unambiguous specification of group policies, policy languages such as Ismene [30], cryptographic context negotiation template (CCNT) [12], and group security policy token (GSPT) [18] have been proposed. Secure group policy components include access control enforcement mechanisms such as access control lists (ACL), encryption or authentication algorithms to use for rekeying as well as secure data communications, selected group rekeying algorithms, and the expected member behavior when a member does not have the latest group security association (GSA) [9], [37].

The group key management problem can be classified to four parts [16]: architecture [2], [13], [17], protocols, algorithm, and policies. Group key architecture focuses on the relationship and placement of group keys, the effect of the change in the spread and density of membership, dynamicity of memberships, and topology of the structure. Group key protocols refer to the set of procedures, message exchanges, and message payloads that govern the behavior of the entities involved in supporting a group. Group key management algorithm refers to the method used to arrange and update the supporting keys which manage the group key [4]. Group key management algorithm gives the detailed description of when and how to update the group key. Group key management policy gives some rules to be followed during the group initialization, the group key distribution, membership changes, emergency situations, and so on [15], [42].

1.3 The One-way Function Tree (OFT) Scheme

Cryptographic key management schemes are required to ensure the confidentiality of a multicast communication. More specifically, backward security requires that a joining member cannot learn previous messages, and forward security requires that an evicted member cannot learn future messages. The adjective *perfect* can be added to the two properties if they can be satisfied against an arbitrary number of colluding members [41].

To satisfy perfect forward and backward security, the group key must be changed whenever a member is added to or evicted from a group. The new key needs to be conveyed to all members at the minimum communication cost since the group is usually large and dynamically changing. The OFT scheme, originally proposed by Balenson *et al.,* is one of the most popular schemes for this purpose [6], [19], [20], [21]. A key advantage of OFT over another popular method, the Logical Key Hierarchy (LKH) [40], is that OFT halves the number of bits broadcasted upon adding or evicting a member. Specifically, if a key has k bits and the key tree used by OFT and LKH has a height h, then the broadcast size of OFT is $hk + h$ bits, whereas that of LKH is $2hk + h$ bits. OFT achieves such a halving in broadcast size by deriving its key tree in a bottom-up manner, in contrast to LKH's top-down approach. Consequently, unlike the independently chosen keys in LKH, the keys in an OFT key tree are functionally dependent, and this functional dependency allows OFT to save half of the broadcasted bits.

Unfortunately, the same functional dependency among keys that brings OFT the reduced communication cost also subjects it to collusion attacks. Although OFT was claimed to achieve perfect forward and backward security [41], only the collusion among evicted members was considered. A collusion that includes current members was claimed to be uninteresting, because a (current) member knows the group key. However, the claim implicitly assumes the colluding members are trying to learn the current group key, which is not necessarily true. An evicted member may collude with a current member to learn group keys that were used after the former was evicted but before the latter joins the group. In this case, OFT will fail on both forward security and backward security.

In 2002, Horng first showed an example of collusion attacks on OFT [22]. In 2003, Ku and Chen provided new attack examples to show that the two assumptions required by Horng's attack were actually not necessary conditions [14]. Ku and Chen also proposed a modified OFT scheme that was immune to the collusion attack. The solution, however, needs to broadcast $(h^2+h)k$ bits on every member eviction and hk bits on each member addition. Ku and Chen's scheme thus loses a key advantage which OFT has over LKH, that is a halving in broadcast size. Because their scheme requires a broadcast of quadratic size on evicting any member, it is only suitable for applications where member eviction is rare.

In this thesis, we revisit collusion attacks on the OFT scheme. To better understand collusion attacks on OFT, we first generalize the examples of attacks given by Horng and Ku *et al.* to a generic attack. Instead of these examples of two or three members, we study the collusion among arbitrary number of evicted and joining members with arbitrary number of other, non-colluding members leaving or joining in between. Based on this understanding of the general attack, we derive necessary and sufficient conditions for a collusion attack on OFT to exist. These conditions reveal that the solution proposed by Ku *et al.* is unnecessarily conservative. Their solution prevents potential collusion attacks by invalidating any knowledge that is brought out of the group by evicted members. However, our results show that such knowledge is not always useful to a joining member in colluding. In particular, we study a different approach where such leaked knowledge is not immediately invalidated but is recorded by a key manager who is responsible for managing the group. When a member joins the group, the key manager then checks whether it is possible for this new member to collude with previously evicted members. If a potential collusion exists, the key manager will update keys as part of the joining operation such that the collusion becomes impossible. Because additional re-keying is performed only when a collusion is possible, this solution has the advantage of minimizing broadcast size. Following the discussion of a straightforward stateful method that has an unacceptable storage requirement, we present a modified version of the method whose storage requirement is proportional to the size of the key tree. These methods pose no additional communication cost on evicting a member but may require more broadcasted bits when a member joins. We study the average performance of the scheme and the simulation results show that our scheme is more efficient than LKH in many cases.

1.4 Thesis Outline

The rest of the thesis is organized as follows. Some important group key algorithms are reviewed in Chapter 2. In chapter 3, we revisit the examples of collusion attacks on OFT given by Horng and Ku *et al.* and generalize it to a generic attack on OFT. We also derive necessary and sufficient conditions for such an attack to exist. An algorithm that minimizes broadcast size while preventing collusion attack is proposed after the generic attack analysis. In chapter 4, we extend our analysis for the case where only the root (group) key needs to be protected. Using this relaxed, yet practical in many applications, security assumption, a more efficient key updating scheme is proposed and analyzed. Chapter 5 concludes the thesis and gives future directions.

Chapter 2

Literature Review

Throughout this section, we briefly review few, albeit representative and important, group key management algorithms. While some of these key management algorithms are used in centralized group key management architectures, the others are used in decentralized architectures.

To ensure the group privacy and control the access to the group, it is necessary to encrypt the group data and distribute the group key to appropriate group members. Group keys need to be changed in the following two cases. First, when the group key has been used for a certain amount of time or used to encrypt a certain amount of data. In order to keep the group key fresh, the group key has to be updated. Second, when the status of the group member has been changed. Some new group members join the group or some group members leave the group. In order to guarantee the backward security, i.e., disallowing joining members from decrypting past data, and the forward security, i.e., disallowing departing members from decrypting future data, the algorithm needs to update the group key [16], [43].

In the rest of this chapter, we first discuss the naïve rekeying scheme whose communication and computational complexity is linear in group size, which is inefficient and only suitable for small groups. Then, the hierarchical subgrouping efficient rekeying scheme, Iolus, is introduced. Next, the batch and periodic rekeying algorithms, Kronos, is reviewed. After that, MARKS algorithm is discussed as a representative example of an efficient key distribution mechanism for a group in which GKC updates the group key at fixed instances and the GKC knows each group member's leaving time when it joins the group. We then discuss LKH, LKH+ and OFT.

2.1 Naïve Rekeying

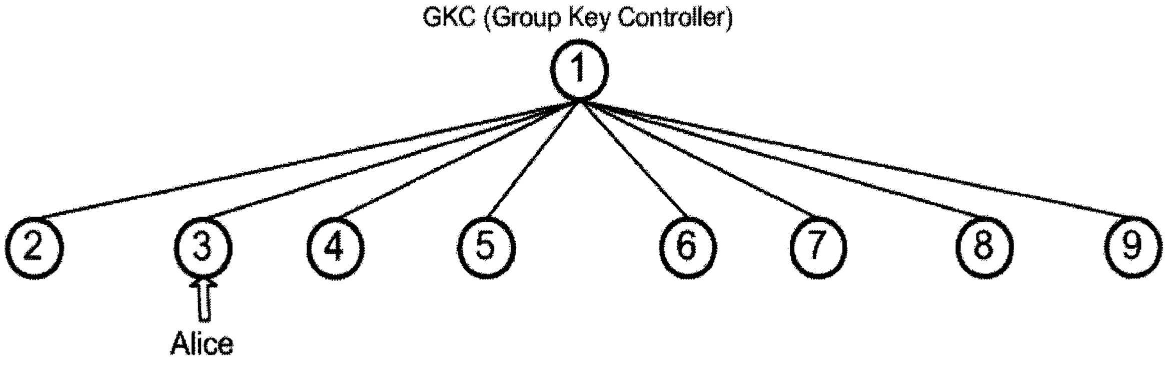

Figure 1 : An Example for a Naïve Group Rekeying Scheme

As showing in Figure 1, the naïve rekeying scheme is a straightforward way to implement the group key management which guarantees both the forward and backward security. The structure of the naïve rekeying scheme works like a star structure. The Group Key Controller (GKC) communicates with each group member directly. Like other centralized group key management schemes, the naïve rekeying scheme has only one GKC which makes it vulnerable to single point of failure. In the naïve rekeying scheme, the GKC stores $n+1$ keys: one pairwise key with each group member, plus a shared group key. Each group member stores a constant number of keys. In general, each group member stores two keys, a shared key with the GKC and the group key [35], [37].

As mentioned above, this naïve solution provides both forward and backward security. On each join or leave, the GKC changes the group key by securely unicasting the refresh key to each group member. To keep the backward security, the GKC updates the group key whenever a new member joins the group by sending the fresh key through multicast package encrypted with the old group key. However, to keep the forward security when group members leave the group, the GKC needs to send the fresh group key to each remaining group member. In another words, the naïve scheme requires a linear-cost key update when group members leave the group.

To do the key update, the GKC needs to generate a new group key first, then encrypt and sign n messages containing the new key. With the naïve scheme, the collusion between group members is impossible. No group member can collude with other group member to get some knowledge of group key for certain time period which they are not supposed to know. If a group member missed a key update message, it is easy to detect and correct it because the naïve scheme uses unicast to distribute keys to all group members.

The naïve rekeying scheme is simple and easy to implement and if the GKC runs on a powerful computer, the key storage requirements are reasonable. It can provide the forward and backward security. It is also resistant to arbitrary collusion among group members. The naïve rekeying scheme does not need a specialized underlying infrastructure like a reliable multicast. On the other hand, the naïve rekeying scheme scales poorly in terms of both group size and group dynamics. The network load communication cost and GKC's computation cost for updating the group key when group member join or leave the group are both linearly proportional to the group size which renders the naïve rekeying scheme suitable only for small groups.

2.2 Iolus

Iolus, proposed by Mittra [34] in 1997, is a framework with hierarchy agents which separate the large group into small subgroup. As shown in Figure 2, the whole group shares a Group Security Controller (GSC) and each subgroup has a Group Security Intermediary (GSI). GSC and GSI are also called Group Security Agents (GSAs) which are trusted by group members. Their main functions are handling the packages routing and managing security (subgroup's keys) for the group [35], [38]. There is no common group key in Iolus, and each subgroup has its own independent key. This means that membership changes in a subgroup only affects the subgroup key, i.e., other subgroups' key don't need to be changed [37].

Whenever a new group member joins a subgroup, the GSI, i.e., the key distributor of that subgroup, sends the new group key to the new member via secure unicast and multicast the new group key, which is encrypted by the old group key, to all existing group members [23]. When a group member leave a subgroup, the GSI generates a new group key and sends it to all remaining group member via secure unicast. Thus the communication cost for leaving operation is linear with the subgroup's size.

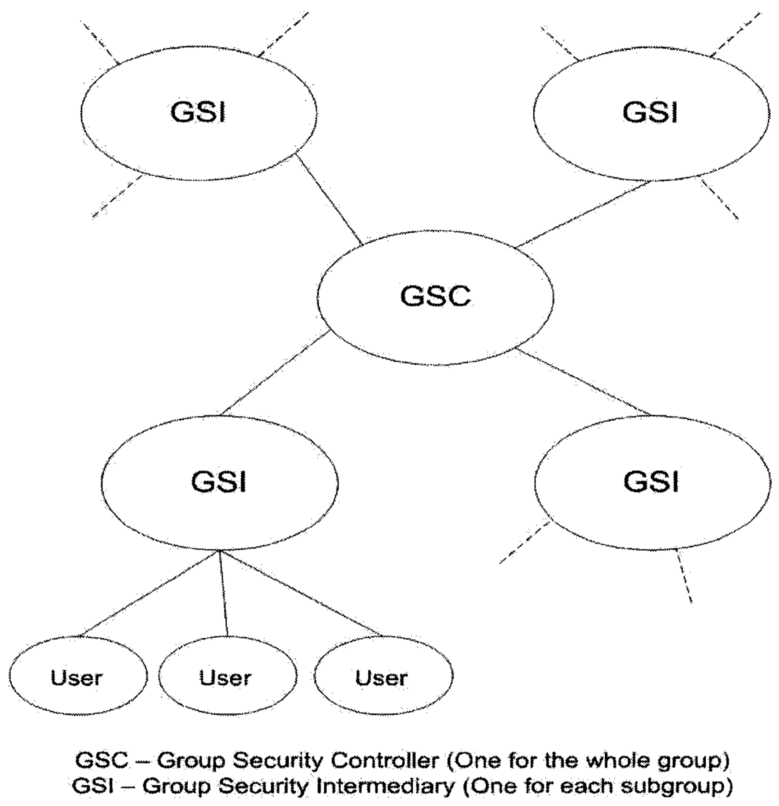
GSI
GSI
GSC
GSI
GSI
User
User
User
GSC – Group Security Controller (One for the whole group)
GSI – Group Security Intermediary (One for each subgroup)

Figure 2 : The Iolus Secure Distribution Tree

Group security agents take the responsibility of the data transmission between different subgroups. If a package needs to be sent from subgroup A to subgroup B, the package is encrypted by the key of subgroup A first. After the GSI of subgroup A gets this package, it decrypts the package and re-encrypts it with the key of the top level subgroup in which the Group Security Controller (GSC) is the key distributor. Once the GSI of the subgroup B gets the package via the top level subgroup communication, it decrypts it and then re-encrypts it with the key of subgroup B. Then this package can be read by the members of the subgroup B.

By separating the whole big group into many small subgroups, Iolus limits the key updating caused by the membership change inside a small subgroup. Iolus has many GSIs, so if one of them failed, only the corresponding subgroup goes down and all other subgroups survive the failure. On the other hand, the inherent complexity of the Iolus hierarchy brings substantial management overhead to the multicast group. In addition, the whole group only has one GSI and it is vulnerable to single point of failure [35], [38].

2.3 Batch and Periodic Rekeying

To ensure the security of the group key, it needs to be updated whenever certain amount of data is encrypted using this key or after certain time interval independent of the amount of data encrypted using this group key. To ensure forward and backward security, the group key may also need to be updated whenever a change occurs in the group membership.

Depend on the application, the requirement for the forward and backward security might be very strict as in military communication in which the group key has to be updated immediately after each group membership change. The strict forward and backward security requirement brings significant rekey communication cost and computation cost to the group especially for the large and highly dynamic group. This drawback makes the strict forward and backward security not very practical for many commercial applications whose security requirement is not that high. To update the key efficiently, the batch rekeying and periodic rekeying have been proposed to relax the forward and backward security requirements. Some applications use batch or periodic rekeying for usual key update and use immediate rekeying only for deleting misbehaving group members [36], [50]. Figure 3 shows the immediate rekeying, batch rekeying and periodic rekeying.

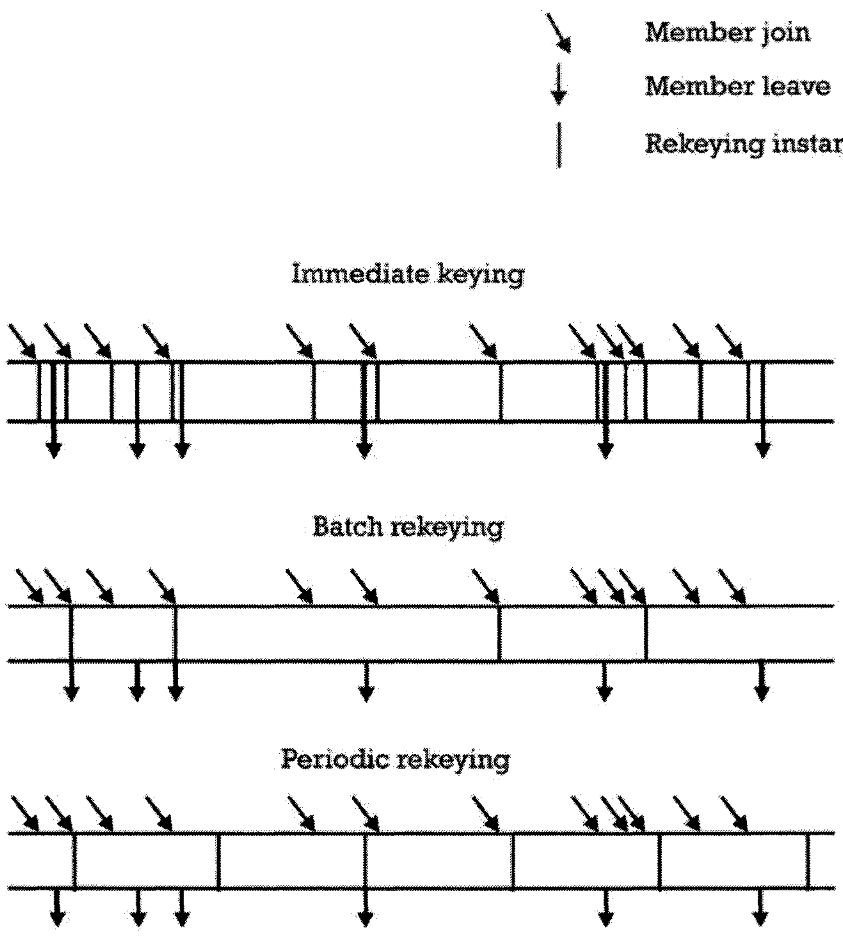

Figure 3 : Immediate, Periodic, and Batch Rekeying [16]

Using batch or periodic rekeying degrades the security for the time period between two key updates. During this time period, the leaving group member can still decrypt the group communication. Also, the new joining group member will not be able to decrypt the group communication until the next rekeying instance [16].

2.3.1 Kronos

Kronos [38], [39] is an example of group key management protocol with periodic key updates. In this protocol, the change of the group membership doesn't lead to an immediate rekeying process. In fact, that is exactly the motivation of this approach is. The authors of Kronos believe that with the increases of the group size and the rate at which members join or leave the group, updating the group key on each membership change is not practical because the frequent rekeying process will occupy most of the communication and computation resource [37].

Kronos does not have a key manager for the whole group. On the other hand, the subgroup group key managers (called area key distributors (AKD) in [39] and [13], have the ability to generate the new keys and distribute them at the end of the predetermined period independently. Kronos is categorized as decentralized architecture and hence avoids the single point of failure problem.

Because the group key updated at certain time, all key managers must synchronize their time and agree on some time period. Using the Network Time Protocol (NTP) [33] to synchronize the time is a reasonable implementation in the real world. Besides the same time, to generate new keys, the subgroup key managers also need to agree on two shared secret factors, K, the master key and R_0, the initial value which is obtained from the group key manager. The subgroup key use the following formula to generate the new key $R_{i+1} = E_k(R_i), i \geq 0$ where E is an encryption algorithm. From this formula, it is clear that Kronos uses the previous key to compute the new one. Thus, if one key gets compromised and the attacker knows the master key, the attacker will be able to compute all the future keys.

2.3.2 MARKS

MARKS [3], [38], [52] divides the whole group's life time into many same length time periods. Each time period has its own group key. Each group number can join the group at any time during the group's life time but each group number's leaving time is assumed to be known at the time of join. The group key controller distributes the key to the new join group member according to its life time period. MARKS uses a binary hash tree to generate keys for all different time periods. Each leaf in the binary hash tree is the group encryption key for a certain time. The root nodes and internal nodes of the binary hash tree are called seeds. Seeds use a blinding function, such as MD5 [32], to generate the whole tree. The height of the tree, h, is predefined according the total number of time periods, N. N different time periods need $N = 2^h$ different keys. Each key corresponds to one leaf in the binary tree. MARKS randomly chooses $S_{0,0}$ to be the root seed. It applies the one-way function G_L to $S_{0,0}$ and get the left children of the root, key $S_{1,0} = G_L(S_{0,0}')$. $S_{0,0}$ is shifted one bit to left before applying the blinding function G_L. MARKS applies the one-way function G_R to the $S_{0,0}$ and get the right children of the root, key $S_{1,1}$. $S_{1,1} = G_R(S_{0,0}')$. $S_{0,0}$ is shifted one bit to right before applying the blinding function G_R. MARKS applies the one-way function G_L and G_R to the following levels until the predefined height h is reached. Figure 4 shows an example for a MARKS' binary hash tree.

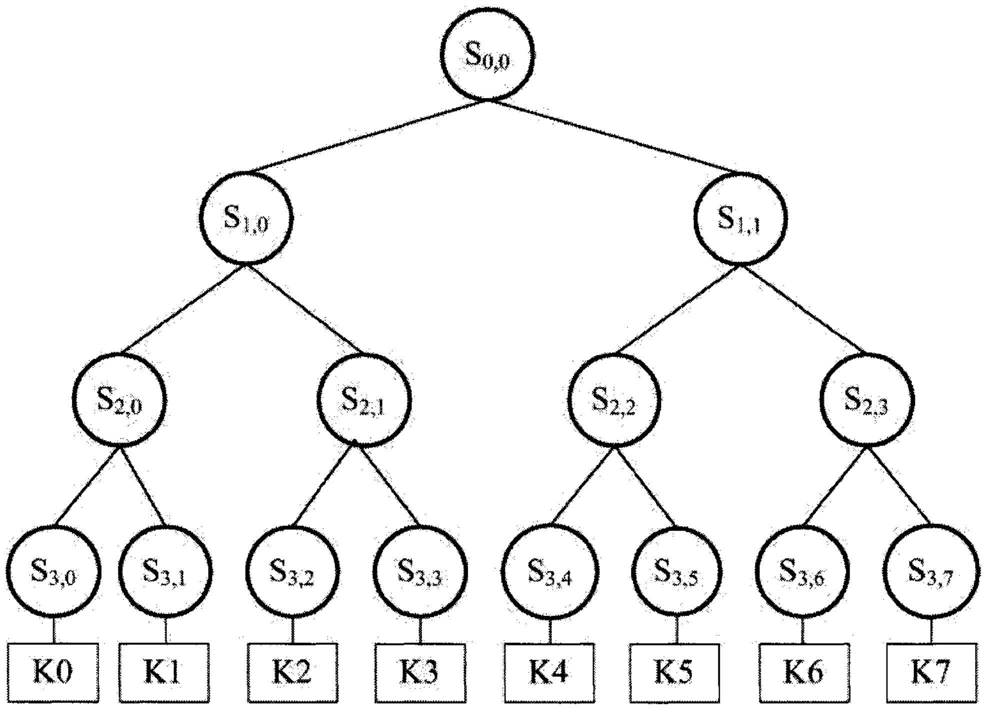

Figure 4 : MARKS Binary Hash Tree

In MARKS, the key distribution process is done when a new member joins the group. With the knowledge of a seed corresponding to a node in the hierarchy, the group member can compute seeds of all descendant nodes. Only necessary and sufficient keys are transported to the new joined member over unicast which is easy to ensure reliable delivery. For example, if a new join member is authorized to know keys from time *2* to *7*, instead of transfer six keys (K_2, K_3, K_4, K_5, K_6 and K_7), the group key controller only sends two seeds, $S_{2,1}$ and $S_{1,1}$, to the new member. Then the new group member computes K_2 and K_3 from seed $S_{2,1}$ and computes K_4, K_5, K_6 and K_7 from seed $S_{1,1}$. If the group key controller sends the root seed to the new group member, the new group member gets authorization to know all keys for the whole group's life time.

To eliminate a group member from the group, the entire group members need to update their keys. In large groups, this character turns MARKS to be a non practical solution for frequent leaving situations.

2.4 Logical Key Hierarchy (LKH)

LKH [8], [16], [19], [25], [29], [38], [40], [46], [51] uses a logical key tree to update the keys of large group efficiently. Each group member is represented by a leaf node in the key tree. The root node is associated with the key for the whole group. Each internal node is associated with the key for corresponding subgroup. Each leaf node knows all keys for the nodes in the path from itself to the root (Figure 5). The number of keys known by each leaf node is *log_2 n*, the height of the tree. If a group member leaves the group, to assure the forward security, all keys known by the leaving node need to be updated.

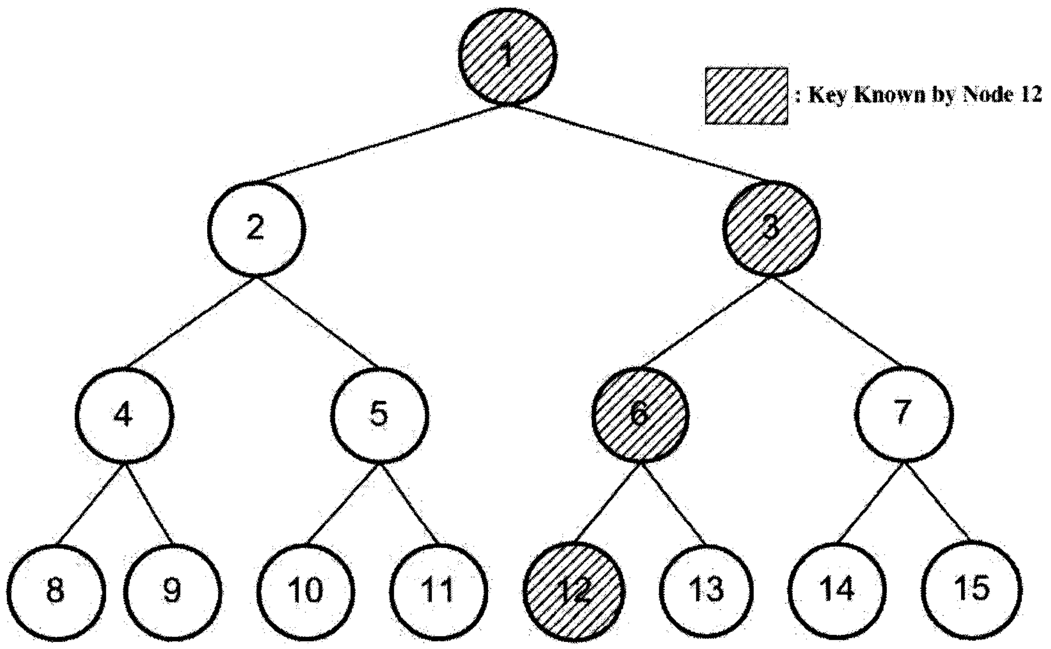

Figure 5 : LKH Binary Tree

2.4.1 Initializing LKH

The Group Key Controller (GKC) generates a binary tree that has as many leaf nodes as there are members. Then each leaf node shares a unique key with the GKC. Next, GKC generates keys for all internal nodes, encrypts them with their two different children nodes' keys and sends them to the group. The alternative way of initializing a key tree is to send each leaf node all the keys in its path to the root encrypted by the key shared between the GKC and the leaf node.

2.4.2 Join rekeying in LKH

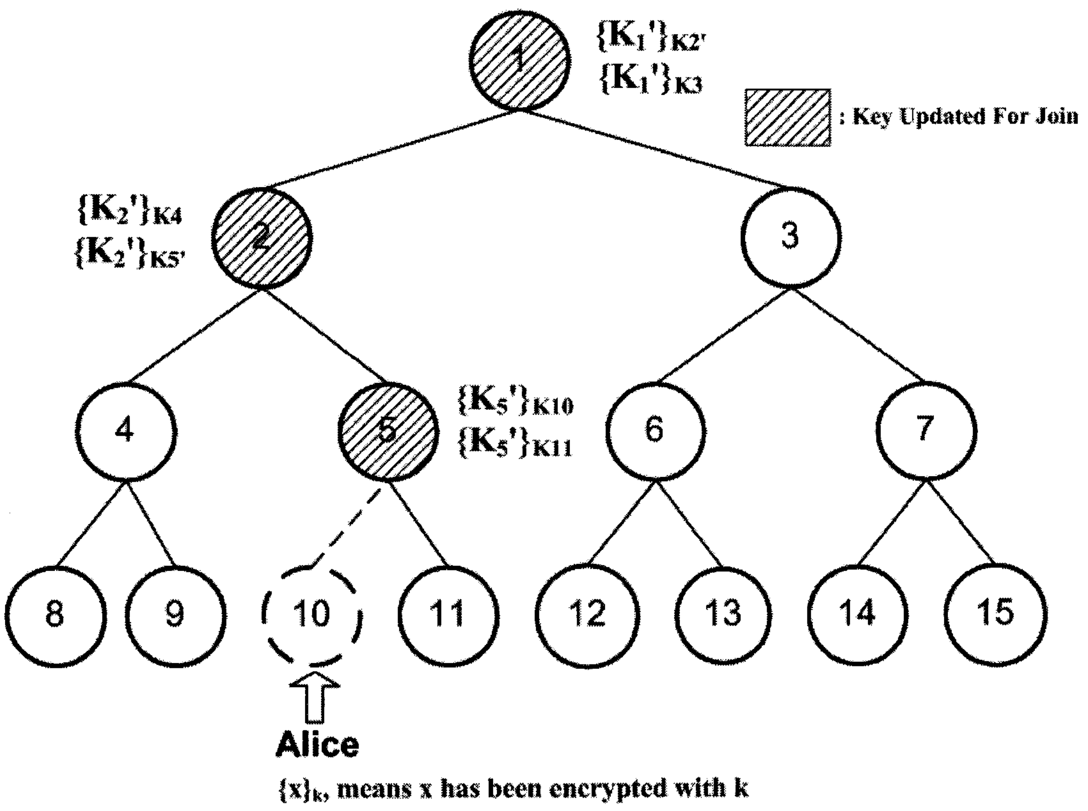

Figure 6 : Join Rekeying in LKH

As shown in Figure 6, after getting a join request from a new group member, the GKC generates a new leaf node in the binary tree and assigns it to the new group member. GKC also generates a key shared only between GKC and the new group member. To assure the backward security, all keys along the path from the joining point to the root node need to be changed. Each of these changed keys will be encrypted twice using its two children nodes' keys respectively and sent to the group via multicast. A new group member will know $log_2 n$ keys, so the GKC needs to send out $2log_2 n$ messages to update the keys for whenever a new member joins the group.

2.4.3 Eviction rekeying in LKH

After getting a leave request from a current group member, GKC deletes the corresponding leaf node from the binary tree. The parent node of the corresponding deleted leaf node is called the leaving point. To assure the forward security, all keys known by the leaving member need to be updated. The GKC generates all these new keys and send them to the remaining members securely, without being known by the leaving member. To illustrate the above process, consider the example shown in Figure 7 where a group member, Alice, leaves the group. Alice knows the keys for nodes *5*, *2* and *1*. In order to keep the forward security, the GKC will update those three keys securely without being known by Alice. As shown in Figure 7, new key for node *5*, K_5', is encrypted by K_{11}, the key for node *11*, and sent to the group. The new key for node *2*, K_2', is encrypted by K_4 and K_5', the key for node *4* and the new key for node *5* respectively, and sent to the group. The new key for node *1*, K_1', is encrypted by K_3 and K_2', the key for node *3* and the new key for node *2* respectively, and sent to the group. After the above updating steps, all group members except the leaving node (Alice) will get the new keys.

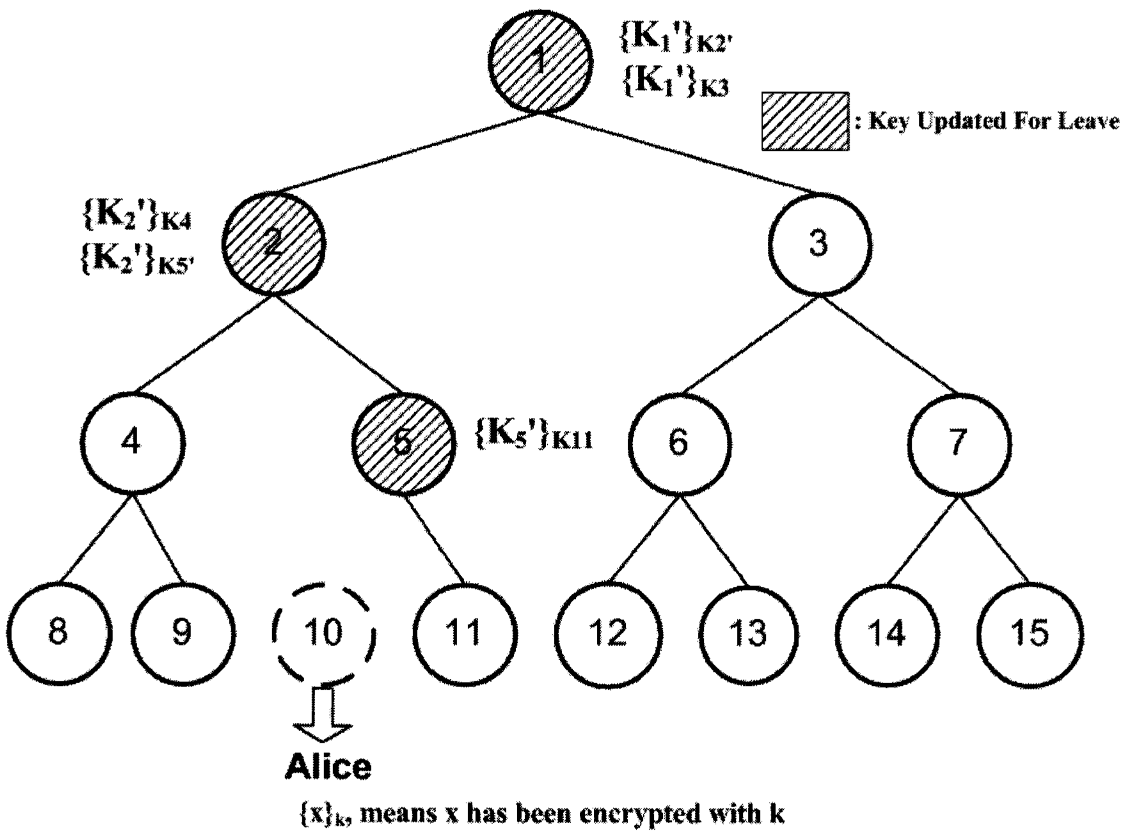

Figure 7 : Eviction Rekeying in LKH

2.4.4 LKH+

Several group key management algorithms, such as LKH+ [7], have been proposed in order to improve the LKH. In LKH+, a one-way function is used to compute the new key. It is different from LKH when a new group member joins the group. GKC doesn't generate and distribute the new keys to certain group members. In contrast, all keys that need to be changed will be updated locally through applying the one-way function. If the group member knows the old key, it can calculate the new one. Because of this, this method cannot apply to the node leaving case. Otherwise, it will break the forward security. The leaving node knows the old keys and it can compute the new key by using the one-way function. Thus when a member leaves the group, LKH+ uses the same way as LKH to handle the key updating process [10], [11].

2.5 OFT

The one-way function tree (OFT) scheme proposed by Balenson *et al.* [1], [31], [52] is widely regarded as an efficient key management solution for multicast communication in large dynamic groups. Each group is associated with a one-way function tree, which is also a binary tree, and is maintained by the group controller. The keys are computed up the tree, from the leaves to the root. This approach reduces rekeying broadcasts to only about $log_2\, n$ keys, where n is the number of group members.

The group manager maintains a binary tree, each node x of which is associated with two cryptographic keys, a node key k_x and a blinded node key $k'_x = g(k_x)$. The blinded node key is computed from the node key using a one-way function g; it is blinded in the sense that a computationally limited adversary can know k'_x and yet cannot find k_x. The group controller uses a symmetric encryption function E to communicate securely with subsets of group members [1], [26], [27], [28], [29], [31], [41], [53].

Figure 8 shows that the member at the leaf labeled *11* knows only the keys of the nodes *11*, *5*, *2* and *1* (the root key, which is used as the group key) and the blinded keys of the nodes *10*, *4* and *3*.

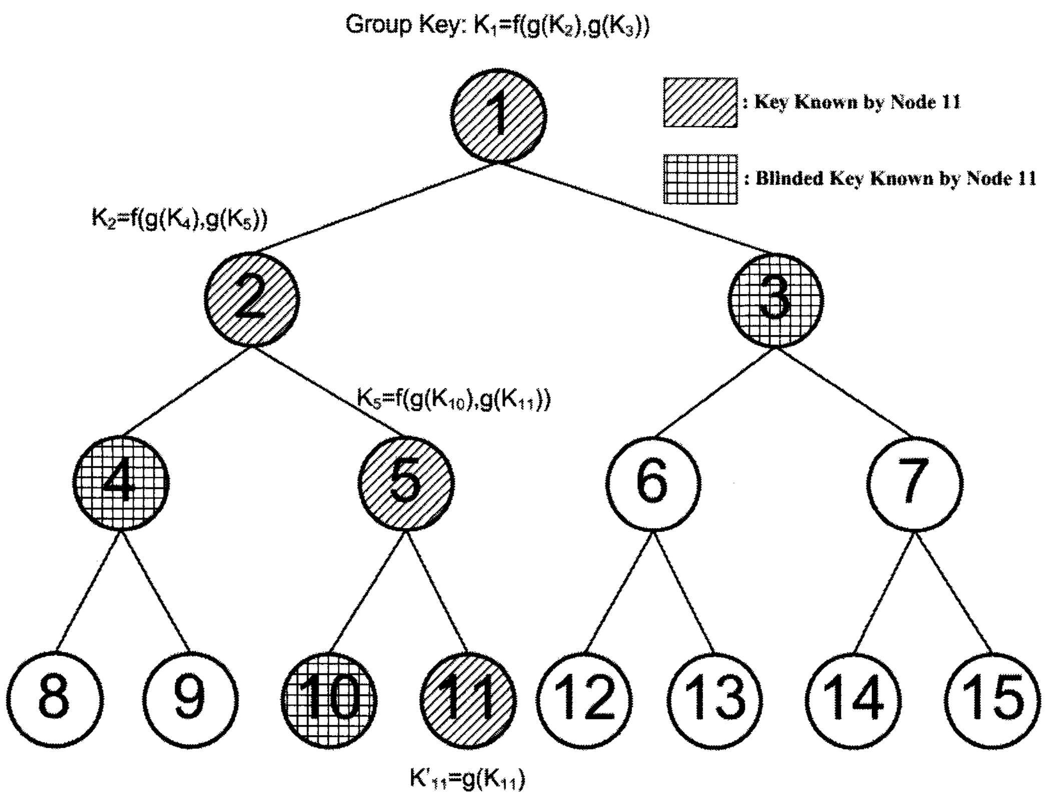

Figure 8 : An Example for OFT Key Tree

2.5.1 Structure of the Original OFT scheme

Every internal nodes of the tree has exactly two leaves. Every leaf of the tree is associated with a group member. Each internal node key can be used as a communications subgroup key for the subgroup of all descendent members. The group controller assigns a randomly chosen key to each member, securely communicates this key to the member using an external secure channel, and sets the node key of the member's leaf to the member's key. The interior node keys are defined by the rule

$$k_x = f(g(k_{left(x)}),\ g(k_{right(x)}))$$

where *left(x)* and *right(x)* denote the left and right children of the node x. The function g is one-way, and the function f is a "mixing" function. For example, the key tree in the left hand side of Figure 8 can be constructed as $x_4 = f(g(x_8), g(x_9))$, $x_2 = f(g(x_4), g(x_5))$, $x_7 = f(g(x_{14}), g(x_15))$, $x_3 = f(g(x_6), g(x_7))$, and $x_1 = f(g(x_2), g(x_3))$.

Each group member knows the unblinded node keys on the path from its node to the root, including the root (computed from the blinded node key it knows), and the blinded node keys that are siblings to its path to the root, and no other blinded or unblinded keys. If one of the blinded node keys changes and it is told the new value, then it can recompute the keys on the path and find the new group key [31], [41], [1]. For example, in the left side of Figure 8, a member, Alice, who is associated with the node *8* will be given the blinded keys $g(x_9)$, $g(x_5)$, and $g(x_3)$. Alice can then compute the group key as: $x_4 = f(g(x_8), y_9)$, $x_2 = f(g(x_4), y_5)$, and $x_1 = f(g(x_2), y_3)$.

2.5.2 Adding a Member

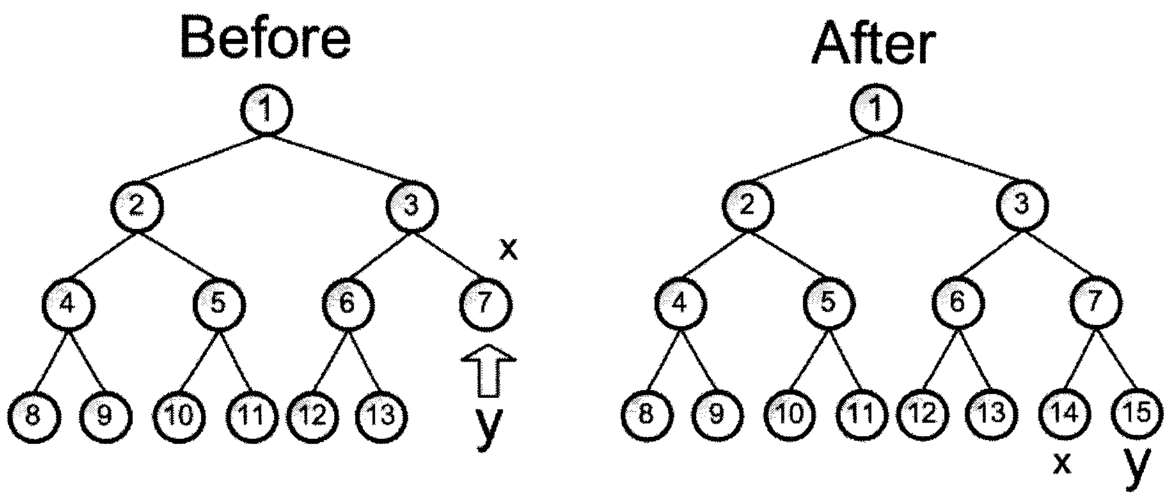

Figure 9 : Adding a Member in OFT

In Figure 9, both members (x and y) are given new keys. The new values of the blinded node keys that have changed are broadcast securely to the appropriate subgroups. The number of blinded keys that must be broadcast to the group is equal to the distance from x to the root plus two. In addition, the new member is given her set of blinded node keys, in a unicast transmission.

When a new member joins the group, an existing leaf node x is split, the member associated with x is now associated with *left(x)*, and the new member is associated with *right(x)*. Both members are given new keys. The whole path from the interior node to the root will be updated due to the two new keys, and the updated blinded keys must be conveyed to those members who need them.

The new values of the blinded node keys that have changed are broadcasted securely to the appropriate subgroups. For example, in Figure 9, the joining member y causes the existing node *7* to be split into two nodes, with each assigned a new node key. The node keys of node *7*, node *3*, and node *1* then need to be updated and their blinded version will be broadcasted to the members who need them (for example node *12* and *13* will need the updated $g(x_7)$). A new member always joins at a leaf node closest to maintain the balance of the key tree. The number of blinded keys that must be broadcast to the group is equal to the distance from x to the root plus two. In addition, the new member is given her set of blinded node keys, in a unicast transmission, using the external secure channel. In order to keep the height h of the tree as low as possible, the leaf closest to the root is split when a new member is added.

Each blinded node key must only be communicated to the appropriate subset of members to maintain security. If the blinded key k'_x changes, then its new value must be communicated to all of the members who store it. These members are all associated with the descendants of the sibling s of x, and they all know the unblinded node key k_s. The manager encrypts k'_x with k_s before broadcasting it to the group, providing the new value of the blinded key to the appropriate set of members, while keeping it from other members.

2.5.3 Evicting a Member

When the member associated with the node y is evicted from the group, the member assigned to the sibling of y is reassigned to the parent p of y and given a new leaf key value. If the sibling s of y is the root of a subtree, then p becomes s, moving the subtree closer to the root, and one of the leaves of this subtree is given a new key (so that the evictee no longer knows the blinded key associated with the root of the subtree). The new values of the blinded node keys that have changed are broadcast securely to the appropriate subgroups, as described above. The number of keys that must be broadcast is equal to the distance from y to the root.

The eviction of a member is similar to the addition with following differences. The sibling of the node associated with the leaving member replaces its parent, and is assigned a new node key. Keys on the path leading that node to the root are then updated and their blinded versions are broadcasted, as in the case of addition. However, if the sibling of the leaving member is an interior node, then we cannot directly change its node key due to the functional dependency among keys. Instead, we need to change the node key of a leaf node in the subtree whose root is that interior node. For example, in Figure 10, the evicted member associated with node *15* causes node *14* to replace node *7*. The node keys of nodes *7*, node *3*, and node *1* will then be updated, and their blinded version will be broadcasted to those who need them [48], [49].

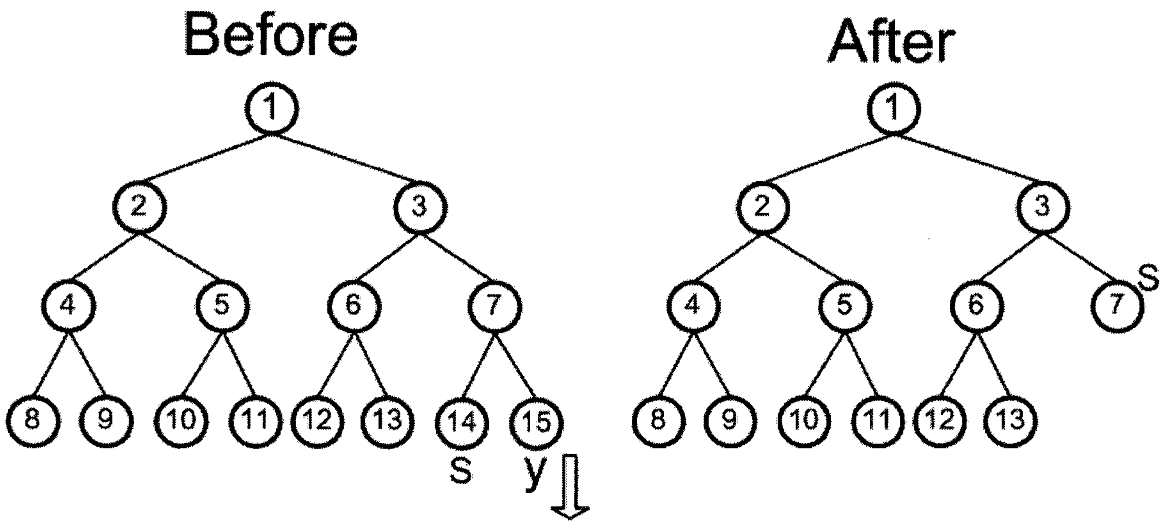

Figure 10 : Evicting a Member in OFT

2.5.4 Broadcast Cost

Let the height of a balanced key tree be *h*. Then approximately *h* new blinded keys must be broadcasted on each member addition or eviction. On the other hand, a unicast is used to send the joining member its blinded keys. In addition, *h* bits are broadcasted to notify members about the position of the joining or eviction. In contrast, the broadcast size of LKH is *2h* multiplied by the key size (plus the same *h* bits for the position of the addition or eviction). The reason that OFT achieves a halving in broadcast size is that keys in an OFT key tree are functionally dependent, but keys in a LKH key tree are all independent.

In OFT, an updated node key is propagated through the sibling of the node, whereas in LKH the key is propagated through the children of the node. In other words, LKH propagates through two children nodes but OFT propagates through only one sibling. That is why the broadcast sizes of LKH and OFT are different [44], [45].

2.6 Summary

Group key management is one of the important components of a multicast security solution. Different key management algorithms have been reviewed in this section. Despite being a straightforward solution, the naïve rekeying algorithm is inefficient for most cases. Iolus uses hierarchy agents to separate a large group into small subgroups; therefore the rekeying process is limited in a corresponding subgroup. However, it brings substantial management overheads to the multicast group. Kronos and Marks are examples of periodic rekeying algorithm. In Kronos, the new keys are generated and distributed by the subgroup group key managers in the end of the predetermined period independently. MARKS algorithm is used in groups where member departure times are known a priori. When a new group member joining the group, a seed is distributed to this new member using a reliable one-to-one secure channel and is used to compute group keys, thus no rekeying and reliable multicast channels are needed. Both LKH and OFT use a hierarchy of keys to achieve scalability. In LKH, the Group Key Controller is responsible for key generation and distribution; thus it has large communication overhead. LKH+ uses key computation techniques to reduce communication overhead due to rekeying. OFT uses group member's keys to compute the group key. This reduces communication overhead compared to LKH, but as we will mention in the next chapter, it brings collusion attack to the algorithm. All these hierarchical keys-based algorithms including LKH, LKH+, and OFT rely on reliable transmission of rekey messages.

Chapter 3

Preventing collusions in OFT

3.1 Examples of Collusion Attack on OFT

Horng [22] observed that the functional dependency among keys in an OFT key tree subjects the OFT scheme to a special collusion attack and gave two conditions for such an attack to exist. Referring to Figure 11, the attack example given by Horng can be described as follows. Suppose Alice, associated with the node *8*, is evicted at time t_1, and later Candy joins the group at time t_2 (ignore Bob's joining for the time being). By the OFT scheme, the node key of node *3* is not affected by the eviction of Alice, so Alice knows the blinded version of this key between t_1 and t_2. Moreover, the node key of node *2* is updated when Alice is evicted, and then remains the same even after Candy joins. Candy can thus see the blinded version of this key between t_1 and t_2. Knowing the blinded node key of both node *3* and node *2* between t_1 and t_2, Alice and Candy can collude to compute the group key during that time interval. The OFT scheme thus fails to provide forward security (Alice knows future group key) and backward security (Candy knows previous group key).

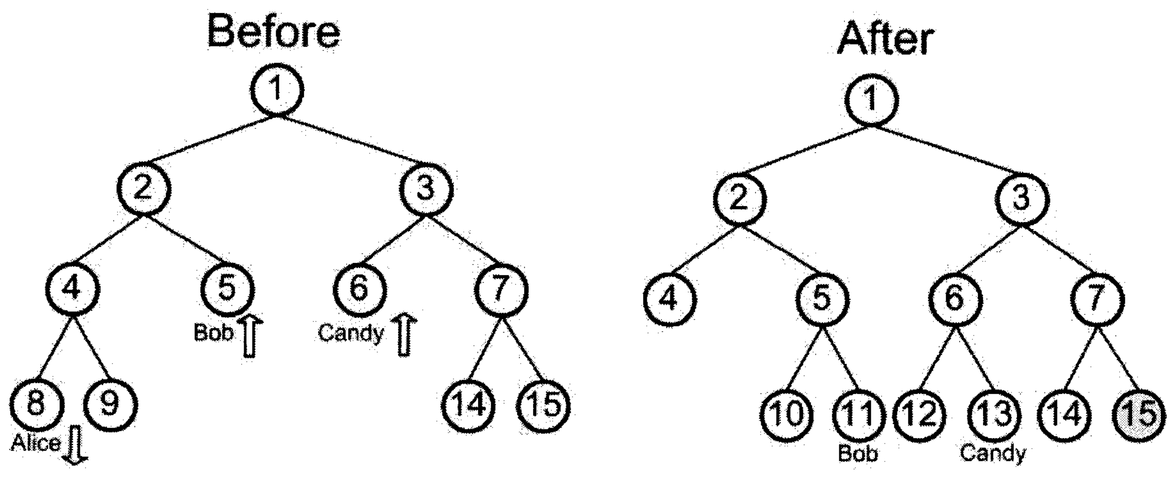

Figure 11 : An Example for a Collusion Attack on OFT

Intuitively, the above example is a result of the unchanging keys of the root's children. Horng thus stated two necessary conditions for such an attack to exist, that is the two colluding nodes evicted and joining at different side of the root and no key update happening between time t_1 and t_3 [22]. Later, Ku and Chen showed, through two more attack examples, that Horng's conditions are actually not necessary [24]. First, referring to Figure 11, if Alice is evicted at time t_1 and Bob joins later at time t_2, then they can collude to compute the node key of node *2* between t_1 and t_2 due to a similar reason. In addition, both Alice and Bob know the blinded node key of node *3* between t_1 and t_2, so they can compute the group key between the same time interval. Second, assume Alice is evicted at time t_1, and Bob and Candy join at time t_2 and t_3, respectively, with $t_1 < t_2 < t_3$.

By similar arguments, Alice knows the blinded node key of node *3* between t_1 and t_3, and

Candy knows the blinded node key of node *2* between t_2 and t_3. Thus, they can collude to compute the group key between t_2 and t_3. The two examples show that Horng's two conditions are actually not necessary.

Ku and Chen [24] also provided a solution to prevent the collusion attack on OFT. Intuitively, an evicted member brings out knowledge about some keys that will remain the same for a certain time interval after the eviction. Ku and Chen modified the OFT scheme to change all the keys known by an evicted member upon the eviction. For example, when Alice is evicted in Figure 11, the node 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 OFT scheme). With this solution, no evicted member can bring out any knowledge about future keys, so collusion with future joining members is prevented. However, the solution updates the node key of all the h siblings on the path of an evicted node (node *5* and node *3* in above example). Each such update requires the broadcast of h keys (for ex-ample, to update the node key of node *3*, we must update one of the leaf nodes in the subtree rooted as node *3*). The broadcast size is thus h^2 multiplied by the key size plus h bits. Because such a broadcast is required for every eviction, the modified OFT is less efficient than LKH (which broadcast $2h$ keys on an eviction) in most cases, unless member eviction is rare.

3.2 Generic Collusion Attack on OFT

The examples given by Horng, Ku and Chen are not sufficient for deriving the necessary and sufficient conditions of collusion attacks on the OFT scheme. In this section, we study such conditions by generalizing their examples into a generic collusion attack on OFT. Section 3.4.1 first studies a special case where an evicted node colludes with another node that joins later on. This turns out to be the only interesting case. Section 3.4.2 then discusses the general case where multiple evicted nodes and joining nodes may collude.

3.2.1 Collusion Between An Evicted Node and A Joining Node

We first consider the collusion attack between a node A evicted at time t_A and a node C joining the group at time t_C $(t_A < t_C)$. Without loss of generality, we assume A is the leftmost node in the key tree, as shown in Figure 12 (notice that this figure actually combines two different key trees at t_A and t_C, which will be justified later in this section). We also need following notations. For any node v, we use $xv[t_1,t_2]$ and $yv[t_1,t_2]$ for its node key and blinded node key between time t_1 and t_2, respectively. We shall also interchangeably refer to a node and the member who is associated with that node. I is the node where the path of A to the root and that of C merges. Let L, R, I', I'' be the left child, right child, parent of I, and parent of I', and let R' and R'' be the right child of I' and I'', respectively. Let B, D, E, and F denote the subtree with the root L, R, $right(I')$, and $right(I'')$, respectively. Let t_{DMIN}, t_{EMIN}, and t_{FMIN} be the time of the first key update after t_A that happens in D, E, and F, respectively. Let t_{BMAX}, t_{EMAX}, t_{FMAX} be the time of the last key update before t_C that happens in B, E, and F, respectively. We then have the following result.

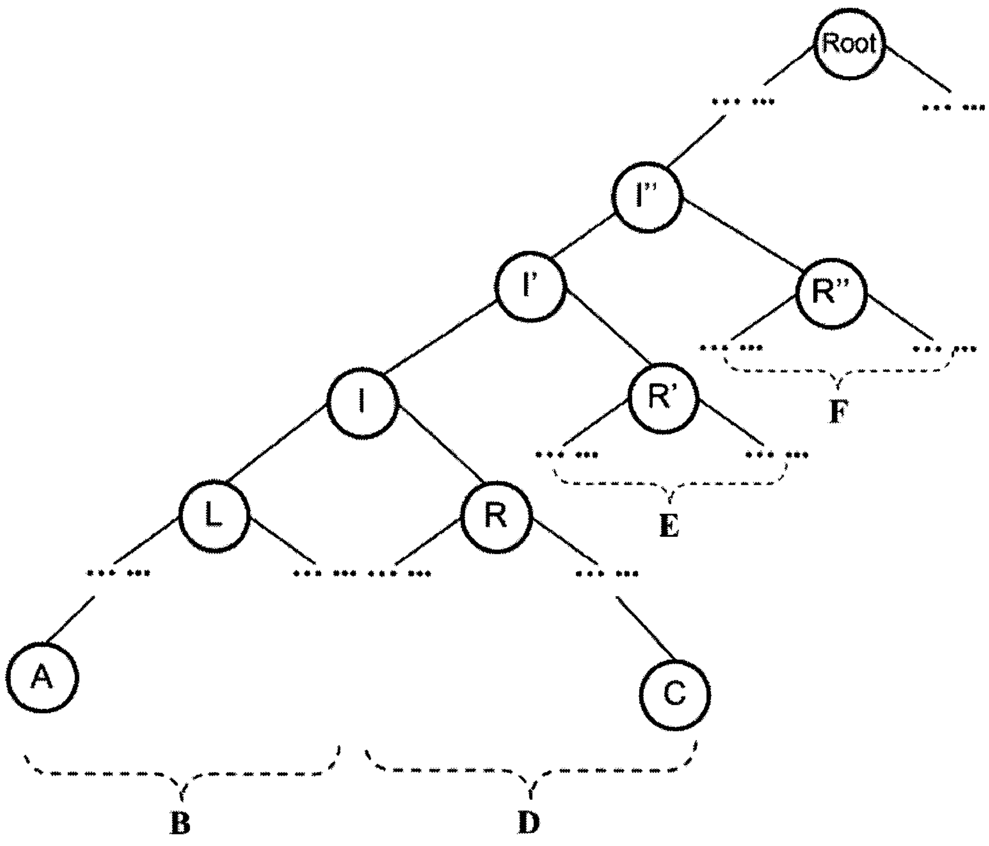

Figure 12 : A Generic Collusion Attack on OFT

Proposition 1: *Referring to Figure 12, the only node keys 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. Notice that these intervals may be empty.

Proof: When the node A is evicted, it knows the blinded node key of each sibling on its path to the root before the time t_A. This includes $y_{R[-,t_A]}$ and $y_{R'[-,t_A]}$ (recall that the dash means the time when each key is last updated before t_A). By the OFT scheme, the node key of R will not change until a new node joins a node in D (that is, the subtree with the root R) or a node in D leaves, and similarly the node key of R' will not change until a key is updated in E. That is, $y_{R[-,t_A]} = y_{R[-,t_{DMIN}]}$ and $y_{R'[-,t_A]} = y_{R'[-,t_{EMIN}]}$. Node A thus knows these values even after it is evicted. On the other hand, when node C joins, it is given the blinded node key of the siblings on its path to the root. Node C then knows the values $y_{L[t_{C},-]}$ and $y_{R'[t_{C},-]}$ (recall that the dash here means the time of the next update of these keys after t_C). By the OFT scheme, the node key of L and R' will not be updated when C joins so they have remained the same since the last key update in B and E, respectively. Then we have $y_{L[t_{C},-]} = y_{L[t_{BMAX},-]}$ and $y_{R'[t_{C},-]} = y_{R'[t_{EMAX},-]}$, which are both known by C.

When A and C colludes, what can be computed depends on the relationship between the timestamps. As shown in Figure 13, A and C can first compute the subgroup key $x_{I[t_{BMAX},,t_{DMIN}]} = f(y_{R[-,t_{DMIN}]}, y_{L[t_{BMAX},-]})$. We notice that this statement assumes $t_{BMAX} < t_{DMIN}$. Under this assumption, nodes A and C can compute $y_{I[t_{BMAX},t_{DMIN}]} = g(x_{I[t_{BMAX},t_{DMIN}]})$. This will enable them to further compute another subgroup key I' in two different time intervals. Let $t_{DEMIN} = MIN(t_{DMIN}, t_{EMIN})$ and $t_{BEMAX} = MAX(t_{BMAX}, t_{EMAX})$. Then $x_{I'[t_{BMAX},,t_{DEMIN})]}$ can be computed by A and C as $f(y_{I[t_{BMAX},,t_{DMIN}]}, y_{R'[-,t_{EMIN}]})$ and $x_{I'[t_{BEMAX},t_{DMIN}]}$ can be computed as $f(y_{I[t_{BMAX},t_{DMIN}]}, y_{R'[t_{EMAX},-]})$. In another word, they can compute the node key of *I'* in $[t_{BMAX}, t_{DMIN}] \cap ([t_A, t_{EMIN}] \cup [t_{EMAX}, t_C])$. Clearly, this result can be easily extended to the parent of *I'* and so on, up to the root.

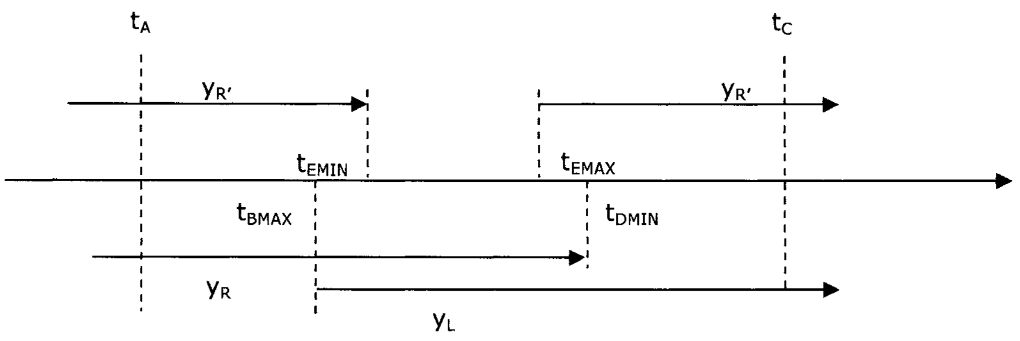

Figure 13 : The Timeline of Collusion Attacks

On the other hand, the above result also depicts all that *A* and *C* can compute by colluding. By the OFT scheme, when A is evicted all the node keys along its path to the root are updated, so *A* no longer knows them. Similarly, *C* cannot learn any node key on its path to the root prior to its joining. Besides the blinded keys of nodes *R, R',* and *R''* (and all the sibling nodes on the path from *I* to the root), A may also know the blinded node key of sibling nodes in the subtree *B* for a time interval after t_A, and similarly *C* may know about nodes in the subtree *D* for a time interval before t_C. However, such knowledge does not help them in computing any keys. By the OFT scheme, a node key can only be computed from the blinded key of its two children, but we can never pick a node from the set $B - \{L\}$ and another from $D - \{R\}$ such that they are the children of the same node.

One subtlety lies in the dynamics of the key tree. The key tree from which *A* is evicted is different from the one that *C* joins. Although we show *A* and *C* in the same key tree in

Figure 12 for simplicity purpose, the tree structure may have been changed after *A* leaves and before *C* joins. However, the key facts that our results depend on will not be affected by such changes. First, *A* knows $y_{R[-,t_{DMIN}]}$ and $y_{R'[-,t_{EMIN}]}$ regardless of any changes that may happen to the subtree with root L, and the definition of t_{DMIN} and t_{EMIN} excludes any change in the subtree with root *R* and *R'* to happen before t_{DMIN} and t_{EMIN}, respectively. It is worth noting that the whole subtree with root *L* may disappear due to evictions, and consequently the node *R* will replace its parent *I* (and the node *R* will be replaced by *right(R)*) by the OFT scheme. In this case, it seems that *A* will no longer know y_R even when no key update happens in the set *D*, invalidating the result that *A* knows $y_{R[-,t_{DMIN}]}$. However, this is not true. When the node *R* replaces *I*, the OFT scheme also requires it to be assigned a new node key, which means at least one of the leaf nodes in the set *D* must change its node key. That is, a key update does happen in *D* in this operation, and our result still holds. Similarly, *C* knows the value $y_{L[t_{BMAX},-]}$ regardless of any change in the key tree after the last key update in the set *B*.

3.2.2 The General Case

We first consider other cases of collusion between pairs of evicted and joining nodes and show that the above eviction-joining scenario turns out to be the only interesting case, as explained by Proposition 2. We then discuss the collusion among more than two nodes, and we show that it is sufficient to only consider collusion between pairs of nodes, which is stated in Proposition 3.

Proposition 2: *A pair of colluding nodes 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.

Proof: First, we consider the joining-eviction case. In Figure 12, suppose *C* first joins the group and later *A* is evicted. If *A* and *C* collude, then they trivially know all node keys in the intersection of their paths to the root (for example, node *I* and *I'*) and the siblings (for example, node *R'*) before *C* joins and after *A* is evicted, because *A* is in the group before *C* joins and *C* stays in the group after *A* is evicted. In addition, although *A* knows the blinded node key of some siblings in the subtree *B* and *C* knows the blinded node key of some siblings in the subtree *D*, these keys cannot be combined to compute any node key since no two nodes share a parent. In summary, two nodes colluding in the joining-eviction case cannot compute any node key besides what they already know.

Next consider the eviction-eviction case. Suppose in Figure 12 *A* is first evicted at time t_A and later *C* is evicted at time t_C. Because *C* stays in the group longer than *A* does, their knowledge about the shared keys in the intersection of their paths (such as *I* and *I'*) and the siblings (such as *R'*) is the same as *C*'s knowledge. That is, colluding with *A* does not help *C* with respect to these keys. Similar to the above cases, *A*'s knowledge about nodes in the subtree *B* cannot be combined with *C*'s knowledge about nodes in *D* to compute any node key. The only exception is their knowledge about *L* and *R*, which can potentially be combined to compute I (and consequently *I'* and so on). However, the OFT scheme updates the node key of *R* when *C* is evicted, so *A* can at best know $y_{R[-,t_A]} = y_{R[t_A,t_C]}$ (if no other key update happens between t_A and t_C), which is useless to *C*. In summary, two evicted nodes colluding cannot compute any node key in addition to what is already known by the later-evicted node. The joining-joining case is similar to the eviction-eviction case and is omitted.

Proposition 3: *An arbitrary collection of evicted nodes and joining nodes can collude to compute some node key not already known, if and only if the same node key can be computed by a pair of nodes in the collection.*

Proof: The if part is trivial, and the only if part can be justified as follows. To compute $xv[t_1,t_2]$, the colluding nodes must know both $y_{left(v)}$ and $y_{right(v)}$ for some time intervals that are supersets of $[t_1, t_2]$. *Suppose* $y_{left(v)}$ is known by m nodes in time period $[t_{ai}, t_{bi}](1 \leq i \leq m)$, and $y_{right(v)}$ is known in $[t_{cj}, t_{dj}](1 \leq j \leq n)$.

Because $(\bigcup_{i=1}^{m} [t_{ai}, t_{bi}]) \cap (\bigcup_{j=1}^{n} [t_{cj}, t_{dj}])$ is a superset of the non-empty time interval $[t_1, t_2]$, it cannot be empty, either. Consequently, there must exist a pair of *i* and *j* such that $[t_{ai}, t_{bi}] \cap [t_{cj}, t_{dj}] \neq \varphi$. The pair of nodes that has such knowledge (no single node can possess this knowledge because we assume $x_{v[t_1,t_2]}$ is not already known by the colluding nodes) can thus collude to compute x_v during the time interval $[t_{ai}, t_{bi}] \cap [t_{cj}, t_{dj}]$.

We now show that the attack examples given by Ku et al., as described in Section 3.1.2, are special cases of our generic attack. Referring to Figure 11, the first example says that

Alice evicted at t_1 colludes with Bob joining at t_2, and Candy joins at t_3 $(t_1 < t_2 < t_3)$. This corresponds to the case where $A = 8$, $C = 5$, $I = 2$, $I' = 1$ (referring to Figure 11), and Candy joins at t_3 in the set E. We thus have $t_{BMAX} = t_1$, $t_{DMIN} = t_2$, and $t_{EMIN} = t_{EMAX} = t_3$. It then follows that Alice and Bob can collude to compute $x_2[t_1, t_2]$ and $x_1[t_1, t_2]$ (notice that $[t_1, t_2] \cap ([t_1, t_3] \cup [t_3, t_2]) = [t_1, t_2]$). The second example says that Alice evicted at *t1* colludes with Candy joining at t_3, with Bob joining in between at t_2. This corresponds to the case where $A = 8$, $C = 6$, $I = 1$ (I' does not exist), and Bob joins in the set B. We thus have $t_{BMAX} = t_2$ and $t_{DMIN} = t_3$, and consequently Alice colluding with Candy can learn $x_1[t_2, t_3]$.

3.3 A Solution for Preventing Collusion Attacks

The previous section shows that a joining node may collude with previous evicted nodes to compute node keys in certain time intervals, which none of them is supposed to know. However, these results also show that such a collusion is not always possible, and whether it is possible depends on the temporal relationship among joining and evicted nodes. As discussed in Section 3.1.2, Ku and Chen's solution prevents any evicted node from bringing out knowledge about future node keys. Although it suffices to prevent any collusion attack, this conservative approach has a quadratic broadcast size (in the height of the key tree) on every member eviction and thus is less efficient than the LKH scheme in most cases.

One apparent way to reduce the broadcast size is to update additional keys only when a collusion attack is indeed possible. Unfortunately, this cannot be achieved with Ku and

Chen's approach of updating the siblings along the path of an evicted node, because at the time a node is evicted, we do not yet know with whom it may collude in the future. On the other hand, our results in Section 3 make it possible to check whether a joining node can collude with any previously evicted node. If a collusion is possible, we can update a minimum number of additional keys to prevent the joining node from combining its knowledge with the evicted node for that specific collusion. This approach minimizes the communication cost for each joining operation (the eviction operation has no additional communication cost) because a key is updated only when necessary.

We first describe a stateful method that explicitly records all the knowledge of evicted nodes. This straightforward method simply applies the results in the previous section to check for possible collusions. However, because the method needs to keep information about all evicted nodes, the storage requirement is proportional to the number of all evicted nodes, which is not acceptable in most applications. Later in this section, we modify this method such that its storage requirement becomes proportional to the size of the key tree. Both methods will eliminate collusion attacks while minimizing the broadcast size.

A Stateful Method:

For the stateful method, the key manager tracks all evicted nodes and checks whether a joining node can collude with any previously evicted node. If a collusion is possible, additional key updates are performed to remove the joining node's knowledge about past node keys such that the collusion becomes impossible. The key manager needs to record two kinds of knowledge. First, the knowledge about *future* node keys that each evicted node brings out of the group. Second, the knowledge about *past* node keys that a joining member is given when it joins. For this purpose, the key manager stores a modified key tree as follows. Each node in the OFT key tree is now associated with a pair $< t_u, L >$, where *tu* is a timestamp and L is a collection of timestamp pairs $< t_{x1}, t_{y1} >,< t_{x2}, t_{y2} >, \ldots,< t_{xn}, t_{yn} >$.

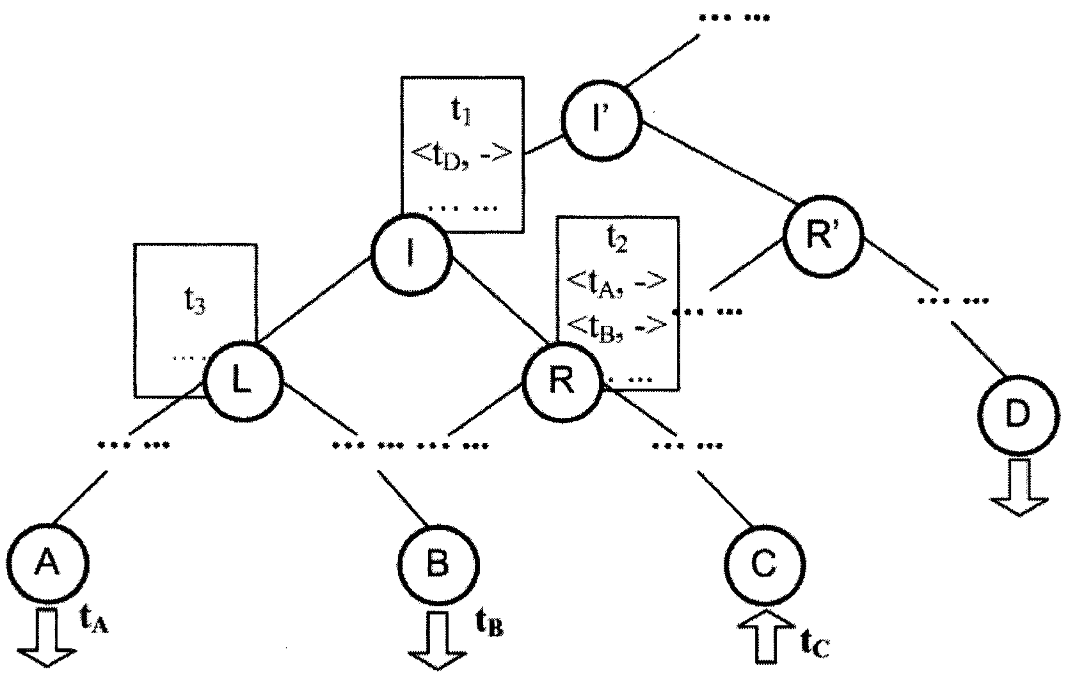

Figure 14 : A Stateful Method for Preventing Collusion Attack

The OFT scheme will be modified such that the timestamp t_u records the time that the current node was last updated, and each pair $< t_{xi}, t_{yi} >$ records the time interval in which some evicted node knows the blinded node key of the current node. For example, Figure 14 shows such a modified OFT tree. Due to space limitation, only the three nodes *I, L,* and *R* have part of their timestamps shown in the figure. In the example, nodes *A, B,* and *D* were evicted at time t_A, t_B, and t_D, respectively. Another node *C* joined at time t_C. Node

R was only updated once between t_A and t_B, and the update happened at time t_2. Node I was last updated at time t_1, which is before t_D (t_1 is equal to either t_2 or t_3). In the table attached to R, the timestamp t_2 records the time of its last update. The first pair $< t_A, t_2 >$ records the fact that node A knows the value $y_{R[t_A, t_2]}$. The second pair $< t_B, - >$ records that B knows the value $y_{R[t_B, -]}$ (that is, the value of y_R from t_B until now). In the table attached to I, t_1 is the last update time of I, and $< t_D, - >$ records that node D knows the value $y_{I[t_D, -]}$. In the table of L, the timestamp $t3$ records the time of its last update.

The OFT scheme is modified as follows to update the timestamps and to stop collusions when they become possible. When a node v is evicted at time t, the key manager will also insert a pair $< t, - >$ into each sibling node along the path of v to the root. For example, in Figure 14 the pair $< t_B, - >$ is inserted to the table attached to node R when node B is evicted at time t_B because R is a sibling of L and L is on the path of B to the root. After a node v joins the group, the key manager will check if v can collude with any previously evicted node to compute any node key along the path of v to the root. In Figure 14, after the node C joins the group, for each node on the path of C to the root (excluding C), the key manager needs to do the following. Taken R as an example, the key manager will check whether the intersection $[t_3, -] \cap ([t_A, t_2] \cup [t_B, -])$ is empty. If the intersection is not empty, then the node key x_L will be updated, such that C can no longer collude with A and B to compute the node key x_I (in applications where only the root's key needs to be secure, the key manager can ignore the collusion of a subgroup key here).

Whenever the key manager updates the node key of a node v, regardless of the reason of this update, it will take following two additional actions. First, it will change the corresponding timestamp t_u associated with v to be the time of the current update. Second, it will scan all pairs of timestamps associated with v and change every dash in these pairs to the current time. The second action records the fact that the key update has invalidated the evicted node's knowledge about this node key. For example, in Figure 14 when the node A leaves, a pair $< t_A, - >$ is inserted into the table attached to R. Later at time t_2 the node key R is updated for some reason, and the dash in $< t_A, - >$ is replaced by the current time t_2, leading to the pair $< t_A, t_2 >$ shown in the figure. This reflects the fact that A no longer knows the new node key of R after time t_2.

An Improved Method With Linear Storage Requirement:

The stateful method keeps all necessary information for checking possible collusions. This requires the key manager to build up an infinitely increasing list of evicted nodes, which is not acceptable in most applications. A closer look at the method reveals that it is not necessary to keep the whole list, if no collusion is to be tolerated. Actually for each node, it suffices to only keep at most one pair of timestamps (plus the timestamp for its last update). The storage requirement is thus linear in the size of the key tree, because for each node at most three timestamps need to be stored. Following two observations jointly lead to this result.

First, in Figure 14, if $t_A < t_B < t_2$, then after B is evicted the list of timestamps associated with R will be $< t_A, - >, < t_B, - >$. However, the pair $< t_B, - >$ is redundant and can be removed because $[t_B, -]$ is a subset of $[t_A, -]$. In another word, after the first pair of timestamps with a dash appears in the list, no other pair of timestamps needs to be stored until the next key up date happens to the current node. Second, suppose in Figure 14 $t_A <$

$t_2 < t_B$ is true, so none of $< t_A, - >$ and $< t_B, - >$ is redundant. We then have that $t_2 < t_B \leq t_3$ ($t_B \leq t_3$ holds, because t_3 is the time when x_L is last updated and the eviction of *B* will update x_L). Now that we know $t_2 < t_3$, the pair $< t_A, t_2 >$ can be safely removed, because the interval $[t_A, t_2]$ will never have a non-empty intersection with $[t_3, -]$.

Based on these two observations, we modify the eviction operation and key update operation of the stateful method as follows. First, when a node *v* is evicted at time *t* and a pair of timestamps $< t, - >$ is to be inserted into each sibling node along the path of *v* to the root, the key manager inserts this pair only if the pair of timestamps already associated with *v* does not contain a dash. Second, whenever the node key of a node *v* is updated, the key manager deletes any pair of timestamps associated with the sibling of *v* that does not contain a dash. For example, in Figure 14 if another node in the subtree with root *R'* is evicted after t_D but before *I* is updated, then nothing will be inserted into the table shown in the figure. If I is updated and the dash in $< t_D, - >$ is replaced, then this new pair will stay until the next key update in the subtree with root *R'*.

3.4 Experimental Results

This section compares the average communication overhead of our solution, the LKH scheme, the original OFT scheme, and Ku and Chen's modified OFT scheme. Among the four schemes, the original OFT scheme is vulnerable to collusion attacks, and it is included as a baseline to show the additional overhead for preventing collusion attacks. Both our scheme and the modified OFT scheme by Ku and Chen can prevent collusion attacks. The keys in an LKH key tree are independently chosen, so LKH is not vulnerable to the collusion attack discussed in previous sections. We expect our scheme to outperform Ku and Chen's scheme in most cases, because the latter has a quadratic broadcast size for every eviction operation. We also expect our scheme to have a smaller average-case broadcast size than the LKH scheme in some cases.

The communication overhead is measured as the total number of keys broadcasted during a random sequence of joining and eviction operations. We do not consider the unicast of keys to a new member. As discussed in previous sections, collusions depend critically on the order of joining and eviction operations (on the other hand, the specific time duration between these operations is not significant). Starting from an initial key tree of G nodes, a sequence of totally N operations are performed using each of the four schemes. The probability that each operation is the eviction of a member is P (and that of a joining operation $1-P$). As required by the OFT scheme, the position for each joining operation is chosen to be a leaf node closest to the root. For each eviction operation, the node to be evicted is randomly chosen among all existing leaf nodes.

Figure 15 shows the total broadcast size (the number of keys to be broadcasted) versus the size of the key tree. Totally *20000* operations are performed (about half of them are evictions). As expected, the communication overhead of our solution is much less than that of Ku and Chen's scheme (their scheme broadcasts about five times more keys). Compared to the original OFT scheme, our scheme only has small additional overhead until the key tree size increases over *20000* nodes. The broadcast size of our scheme is also smaller than LKH when the key tree size is smaller than *40000.* Table 1 shows a more detailed comparison between the two schemes.

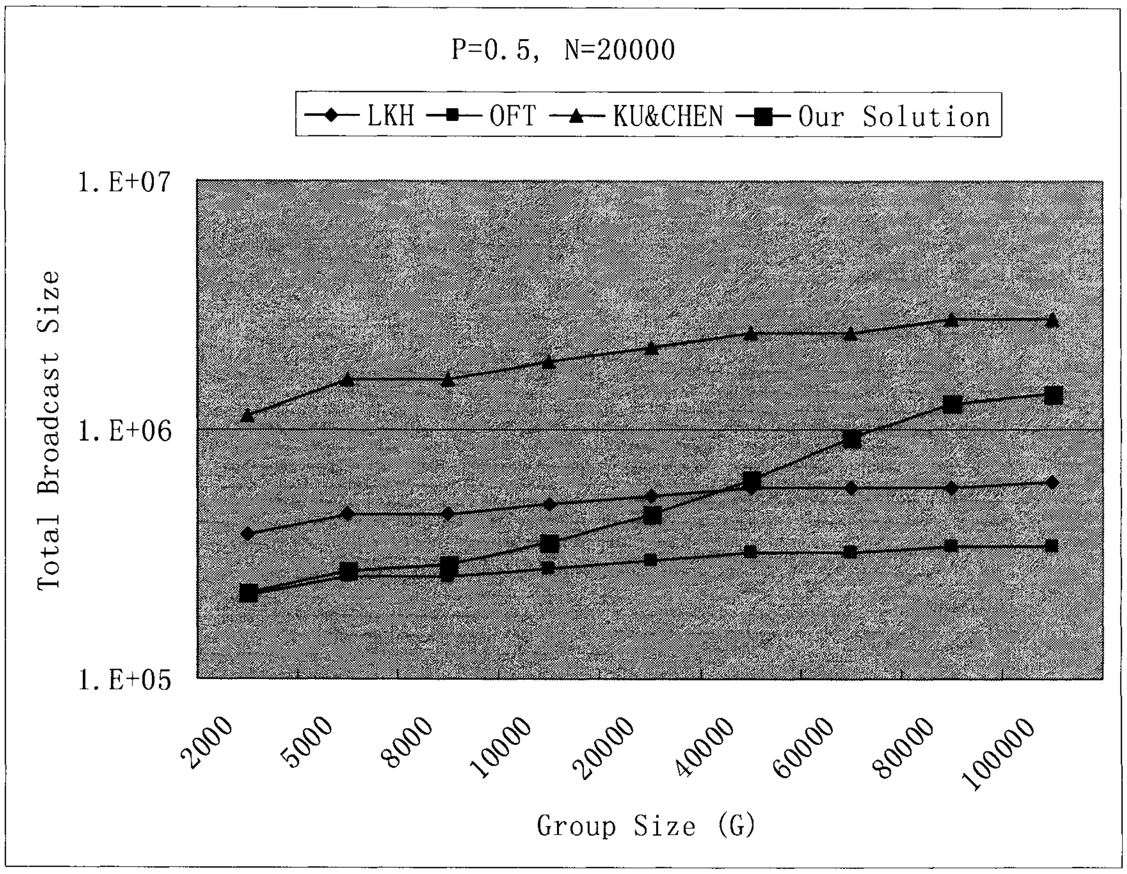

Figure 15 : The Broadcast Size vs. Key Tree Size

Key Tree Size	2000	5000	8000	10000	20000	40000	60000	80000	100000
Our Solution/LKH	0.59	0.60	0.62	0.70	0.84	1.08	1.61	2.19	2.24
No. of Collusion	242	1063	2113	5154	10385	18991	38417	54720	61799
Tree Height	10	12	12	13	14	15	15	16	16

Table 1 : Comparing Broadcast Size of Our Solution to LKH

For larger key trees, our scheme is less efficient than LKH. As shown in the second row of Table 1, the broadcast size of our scheme is about double the size of LKH when the key tree has *80000* or more nodes. This can be explained by the fact that more collusions are possible in a larger tree as shown in Table 1, and the larger height of the tree also increases the number of keys to be broadcasted upon each key update. Ku and Chen's scheme also has a similar trend as ours, which confirms that to prevent collusion attacks, both modified OFT schemes are less scalable than LKH. However, because our scheme only performs additional key updates when necessary, the broadcast size for each operation is already minimal. This indicates an inherent disadvantage of using functionally dependent keys in the face of collusion attacks. For large groups where perfect forward and backward security is important, the LKH scheme will be a better choice.

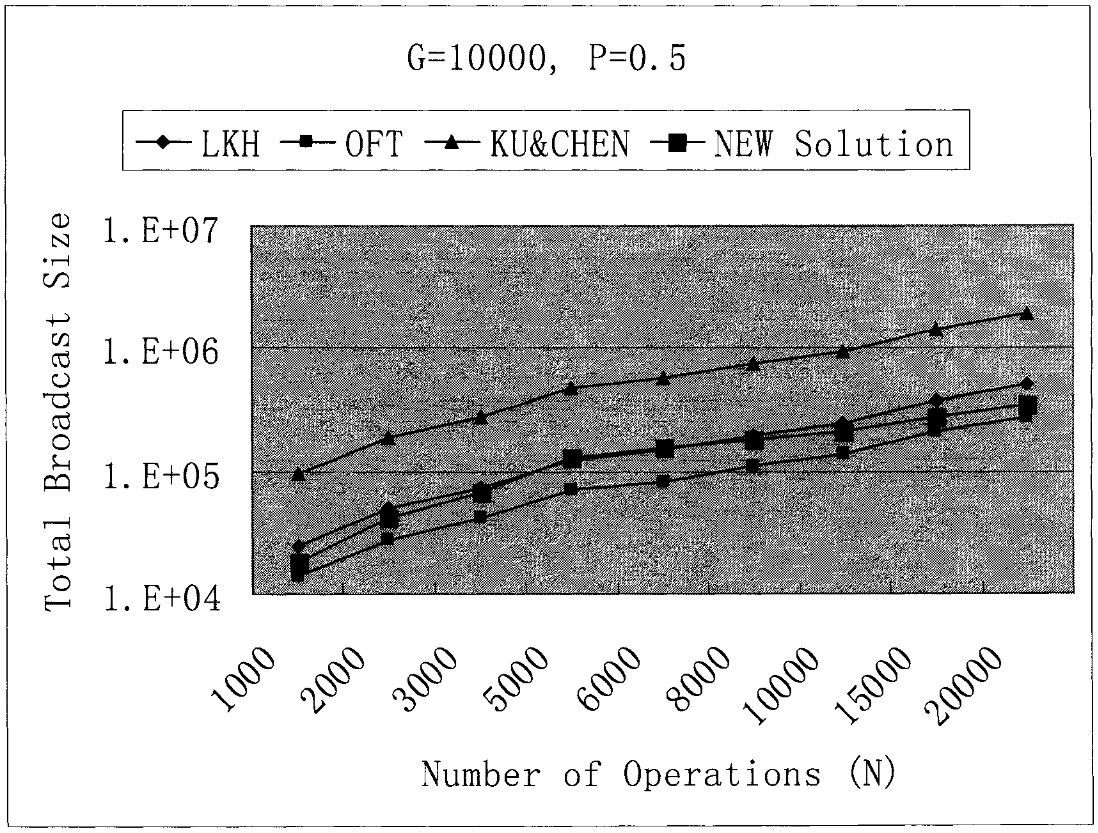

Figure 16 : The Broadcast Size vs. the Number of Operations

Figure 16 shows the total broadcast size versus the number of operations, with about half of the operations being evictions, on a key tree with *10000* keys. Because collusion attacks depend on the order of operations but not on the specific time durations, we can also regard the number of operations as the intensity of operations, thus, Figure 15 and Figure 16 also show the broadcast size versus the degree of group dynamics. The broadcast size of all four schemes increases with the number (intensity) of operations. The original OFT scheme, the LKH scheme, and our modified OFT scheme all scale in roughly the same manner, whereas Ku and Chen's scheme is less scalable. The column chart inside Figure 15 and Figure 16 show the total number of collusions. Interestingly, while the number of collusions remains roughly the same when the number of operations goes over *6000*, Ku and Chen's scheme still shows a significant increase in the broadcast size, because their scheme requires additional key updates for every eviction operation even when such operation do not cause collusion (in contrast, our scheme scales in the same way as the original OFT).

Figure 17 and Figure 18 show the total broadcast size versus the ratio of evictions among all operations. The two experiments differ in the key tree size and in the total number of performed operations. In both experiments, the original OFT scheme and the LKH scheme have a constant broadcast size because in both schemes the joining and eviction require the same amount of keys to be broadcasted. The broadcast size of Ku and Chen's scheme increases linearly in the ratio of eviction, because their scheme requires additional key updates and hence additional broadcasted bits on every eviction operation but not on the joining operation. Our scheme shows an interesting pattern. The broadcast size first increases with the eviction ratio and then decreases after the ratio reaches about

40%. Because a collusion requires both joining nodes and evicted nodes, the total number of collusions reaches a maximal value when about half of the operations are evictions. The maximal broadcast size shifts a little to the left (40% instead of 50%) because our scheme requires additional key updates for joining nodes, but not for evicted nodes. Each joining node thus contributes to the overall broadcast size slightly more than an evicted node does.

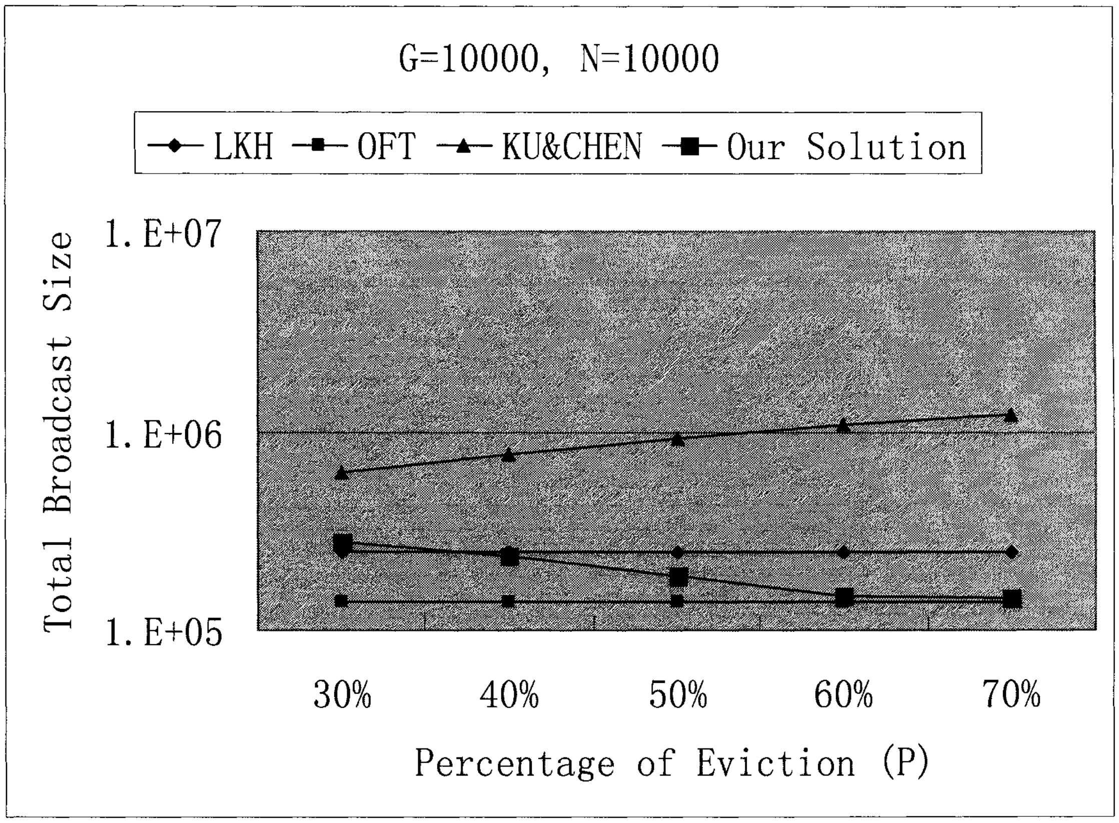

Figure 17 : The Broadcast Size vs. the Ratio of Eviction (Case 1)

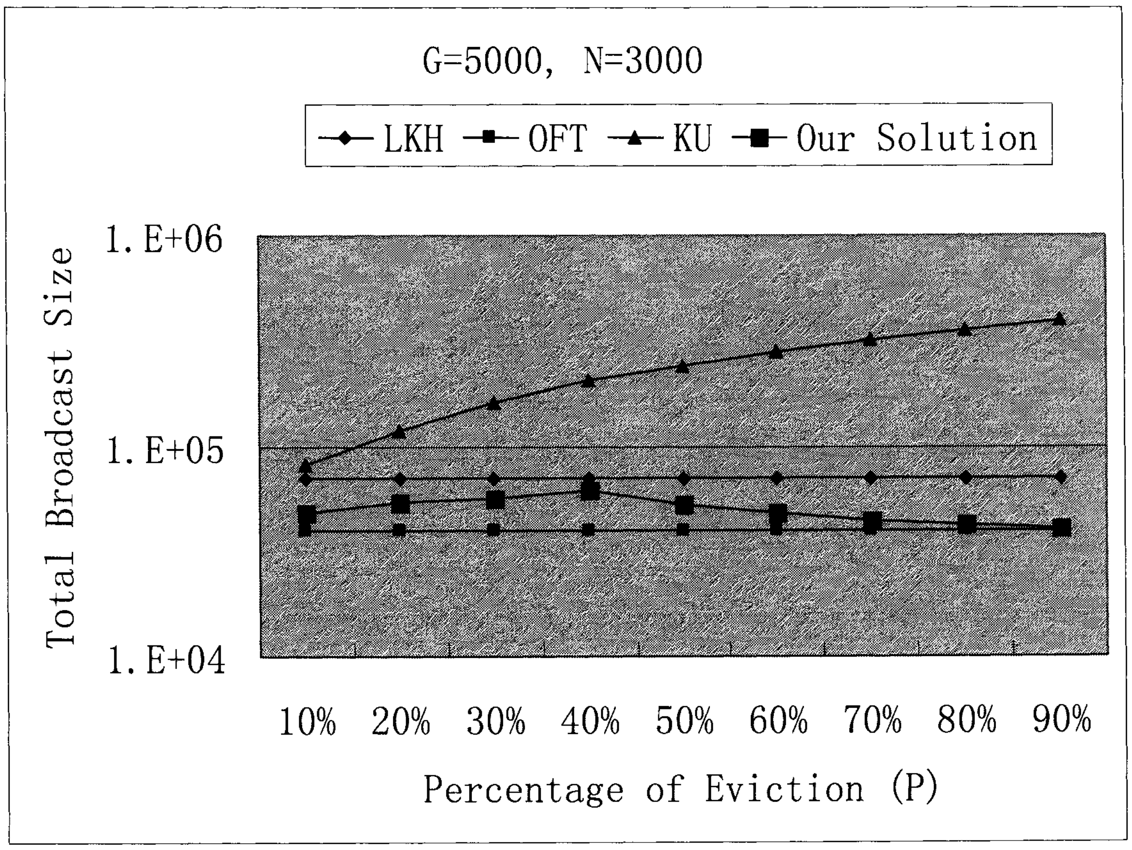

Figure 18 : The Broadcast Size vs. the Ratio of Eviction (Case 2)

Chapter 4

Preventing Collusion Only for Group Key

In this chapter, we consider the case where only the group key needs to be protected. We show that better performance results can be obtained under this relaxed requirement.

4.1 Motivation

Last chapter proposes a method for preventing collusion attacks on the OFT scheme. The method has an improved performance over Ku and Chen's scheme. This performance gain is due to the fact that our method causes additional key updates only when attacks become possible. However, a closer look reveals that the method is still a conservative approach because we prevent the disclosure of any node keys. To recall the method in Chapter 3, when a node joins, we follow its path to the root and check the intersection between the node's knowledge and leaving nodes' knowledge. Upon observing a non-empty intersection, we immediately invalidate the joining node's knowledge through additional key updates. While this approach can provably prevent any collusion from ever becoming a reality, it is unnecessarily restrictive in some applications. As a matter of fact, both the original LKH and OFT schemes focus on protecting the root's key, i.e., the group key. In some applications, some sub-groups may also be used for secure communications and thus need to be protected. Our solution in Chapter 3 certainly can handle these situations. However, for other cases where only the group key is relevant to security, the previous solution becomes inefficient. Notice that in Chapter 3, the performance overhead of our method is mainly due to additional key updates for preventing collusions. If we choose to tolerate collusion attacks on irrelevant sub-group keys (keys of non-root nodes) and only to protect the group key (key of the root node), it is clear that the performance can be significantly improved. From this perspective, the comparison between the efficiency of our solution and that of LKH and the original OFT given in Chapter 3 is not fair since our solution protects all sub-group keys while the other two only protect the group key. This motivates us to develop new methods to protect only the group key. The rest of this chapter presents details of the solution, a sample case analysis, and the experimental results on communication cost comparison.

4.2 The Solution

At first glance, it may seem trivial to keep the group key secure since we can just check the time stamps saved in the root node's two children nodes. If collusion is possible upon the joining of a group member, we apply the same key updates introduced in the last chapter to prevent the collusion. Unfortunately, the above solution is not valid. Figure 19 explains why this straightforward implementation is wrong.

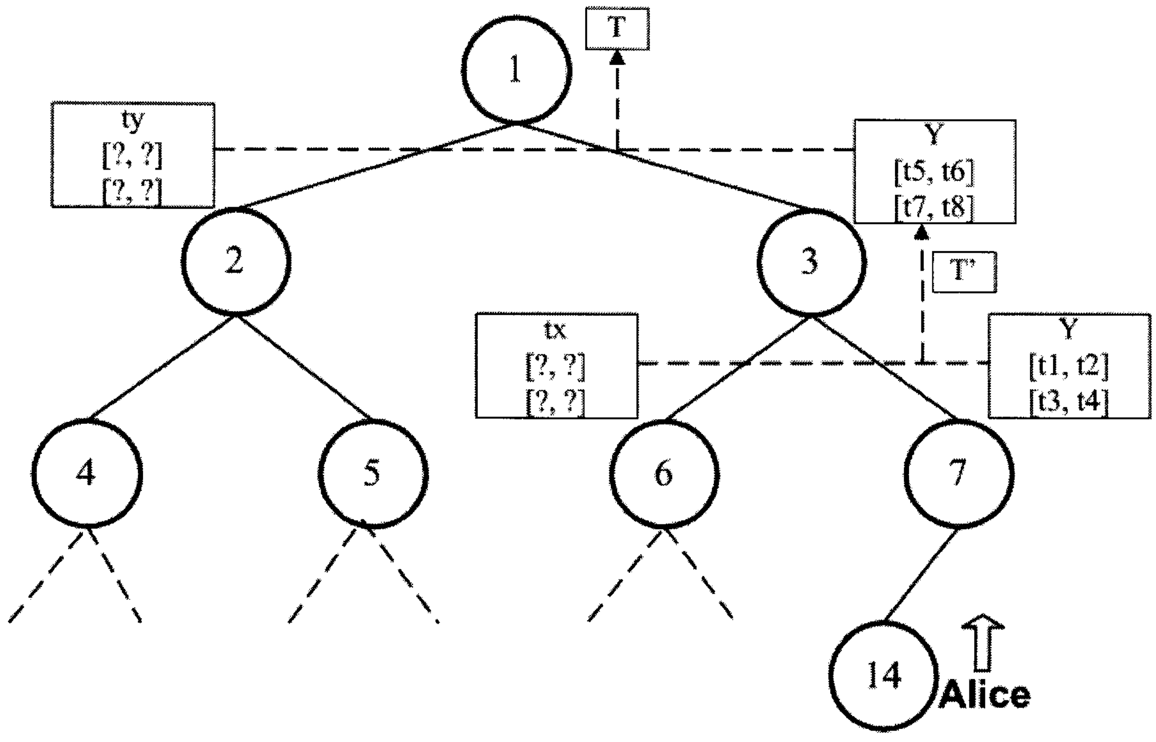

Figure 19 : Propagation of Collusion

In Figure 19, when Alice joins the group at time *Y*, Alice will know the blinded key of node 6 after time t_x, the last update time for node *6*, denoted as $[t_x,-]$; the blinded key of node 2 after time t_y, denoted as $[t_y,-]$; the key of node *1*, *3* and *7* after the joining time *Y*. Assume Alice will collude with other leaving nodes who possess the knowledge about the blinded key of node *7* for some time periods $[t_1, t_2]$ and $[t_3, t_4]$. Then the collusion will enable Alice to learn the key of node *3* for $T' = [t_x,--] \cap ([t_1, t_2] \cup [t_3, t_4])$. Such collusion adds to the knowledge that leaving nodes under nodes *4* and *5* may already possess for, say, time periods $[t_5, t_6]$ and $[t_7, t_8]$. If we had only looked the tables attached to node *2* and *3* for potential collusion, then we may miss the additional intersection between *T'* and $[t_y,-]$ (that is, Alice's knowledge about the blinded key of node *2*), which leads to the compromise of the key of node *1*, that is the group key. This example shows that collusion may potentially propagate upwards and lead to the compromise of the group key.

To make sure that we can identify all possible compromises of the group key and to take actions for preventing them, we adopt the following approach. After a new member joins the group, we will check for collusion for every node along the joining node's path to the root. If we find collusion for a certain time period in a sub-group key, we record this fact as a pair of timestamps in the same way as we record the knowledge brought out by leaving nodes. Indeed, the knowledge obtained through collusion has no difference from the knowledge of leaving nodes from our point of view. If no collusion for a sub-group key is found, we do not stop since collusion may become possible again at a higher level. We repeat this process until reaching the root node. At this point, we will determine whether the group key is compromised, and if so we will update the group key's sibling to prevent its compromises. In Figure 19, after Alice joins the group at time *Y* we first check for collusion with knowledge about node *6* and *7* blinded keys. If we find collusion for a non-empty time interval *T'* we then record *T'* as a pair of timestamps in the table attached to node *3*, which is the parent node of node *6* and *7*. We then move one level up and check for collusion between the knowledge about the blinded keys of node *2* and *3* (including that for *T'*). If T is not empty, we will need to update the key of node *2* to prevent the compromise of the key of node *1*, that is the group key.

As mentioned in Chapter 3, each node in the binary key tree only needs to keep one pair of time stamps to record the information disclosed to leaving group members. However, due to the propagation of collusion, we shall need to record multiple time stamps in the table attached to each node. The storage requirement will be similar to the stateful method introduced in Chapter 3. We implement the tables using an array and resize it by doubling the allocated space once it becomes full.

4.3 Case Study

To demonstrate the method introduced in the previous section, we describe a case study of a group of 51 members with a random sequence of joins and leaves of members. The following shows our program outputs in a few segments. For each segment, we illustrate the outputs through figures and detailed explanations. This case study will clearly show how our method deals with the propagation of collusion along the key tree.

In Figure 20, when node *54* joins to node *27* at time *276*, the time stamps of all nodes along the path from node *54* to the root node *1* will be updated. Specifically, the time stamps of node *27*, *13*, *6*, *3*, and *1* are updated.

In our implementation, we use a $2 \times N$ array to record the timestamps for each node. The last updated time of the node is written to the first location of the array, represented as *info[0][0]*. Every time, if the key of the node needs to be changed because of the group membership chang, we update *info[0][0]* right away. Once a group member is evicted from the group, the current time is written to the timestamps array of all its sibling nodes on a specific location, *info[x][0]*. Once the key of the node is changed and the *info[x][1]* in its timestamps table is empty, the current time is written to *info[x][1]*. For example, at time *276*, Table 2 shows the node *6* has following *26* pairs of timestamps.

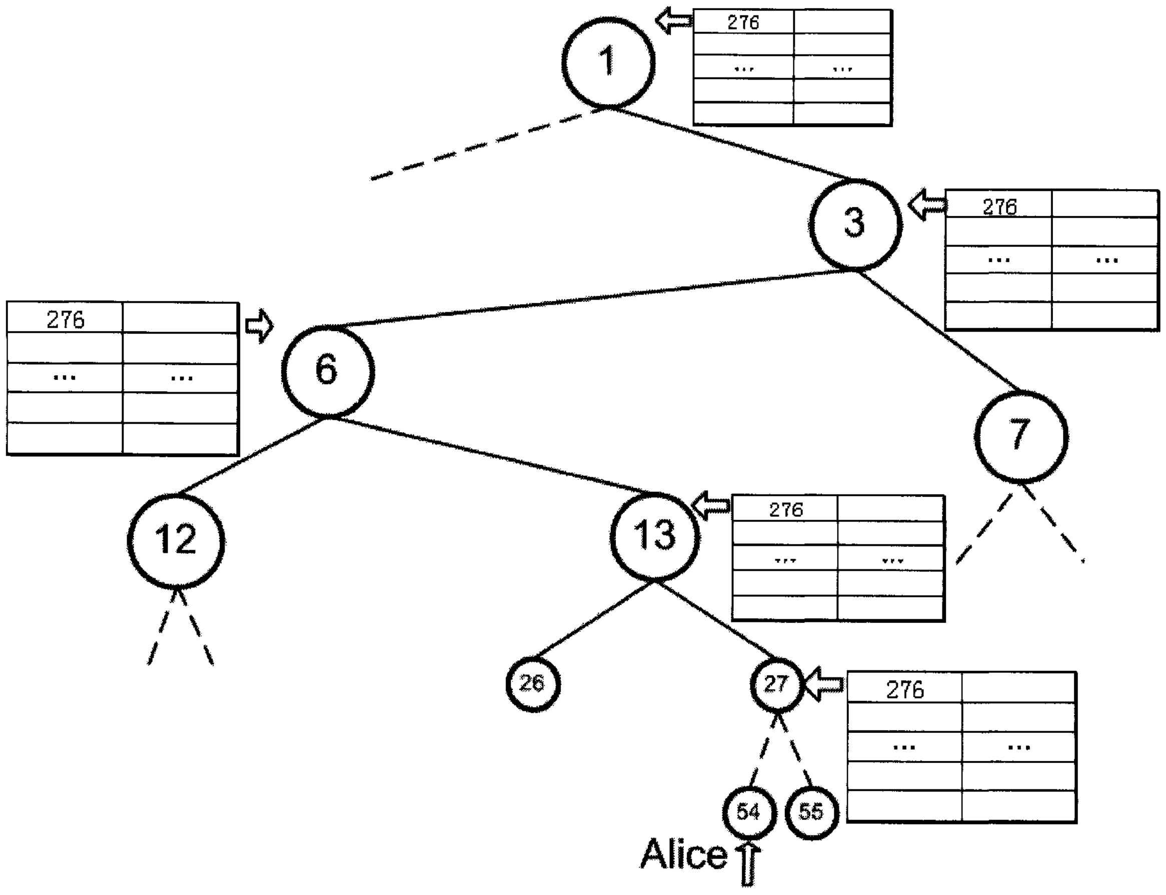

Figure 20 : Time Stamp Update When Node 54 Joins the Group

In Table 2, line *25*, *Node 6--info[24][0]-[24][1] : 260 --> 262*, means that in the timestamps table of node *6*, the *24th* time period record for the evicted node is from time *260* to time *262*. This implies that an evicted node knows the key of node *6* between time *260* and time *262*.

	Time Stamps in Node 6	Time
1	Node 6--info[0][0]-[0][1] :	276 --> 0
2	Node 6--info[1][0]-[1][1] :	102 --> 109
3	Node 6--info[2][0]-[2][1] :	114 --> 115
4	Node 6--info[3][0]-[3][1] :	120 --> 125
5	Node 6--info[4][0]-[4][1] :	128 --> 132
6	Node 6--info[5][0]-[5][1] :	126 --> 132
7	Node 6--info[6][0]-[6][1] :	136 --> 137
8	Node 6--info[7][0]-[7][1] :	140 --> 145
9	Node 6--info[8][0]-[8][1] :	137 --> 145
10	Node 6--info[9][0]-[9][1] :	156 --> 159
11	Node 6--info[10][0]-[10][1] :	160 --> 163
12	Node 6--info[11][0]-[11][1] :	167 --> 168
13	Node 6--info[12][0]-[12][1] :	179 --> 180
14	Node 6--info[13][0]-[13][1] :	181 --> 183
15	Node 6--info[14][0]-[14][1] :	187 --> 189
16	Node 6--info[15][0]-[15][1] :	193 --> 194
17	Node 6--info[16][0]-[16][1] :	196 --> 199
18	Node 6--info[17][0]-[17][1] :	203 --> 205
19	Node 6--info[18][0]-[18][1] :	206 --> 207
20	Node 6--info[19][0]-[19][1] :	212 --> 214
21	Node 6--info[20][0]-[20][1] :	225 --> 226
22	Node 6--info[1][0]-[1][1] :	239 --> 240
23	Node 6--info[22][0]-[22][1] :	245 --> 246
24	Node 6--info[23][0]-[23][1] :	249 --> 254
25	**Node 6--info[24][0]-[24][1] :**	**260 --> 262**
26	Node 6--info[25][0]-[25][1] :	247 --> 254
27	**Node 6--info[26][0]-[26][1] :**	**273 --> 274**

Table 2 : Time Stamps in Node 6 at Time 276

After node *54* joins node *27* at time *276*, the key update algorithm will check for collusion at each level along the path from node *54* to the root node *1*. Figure 21 shows there are more than one collusion time periods when we compare all the timestamps saved in node *6* and node *7*, which represents the knowledge of node *54* after it joins the group. Once the collusion time period has been calculated, it is copied to the timestamps table of the parent node.

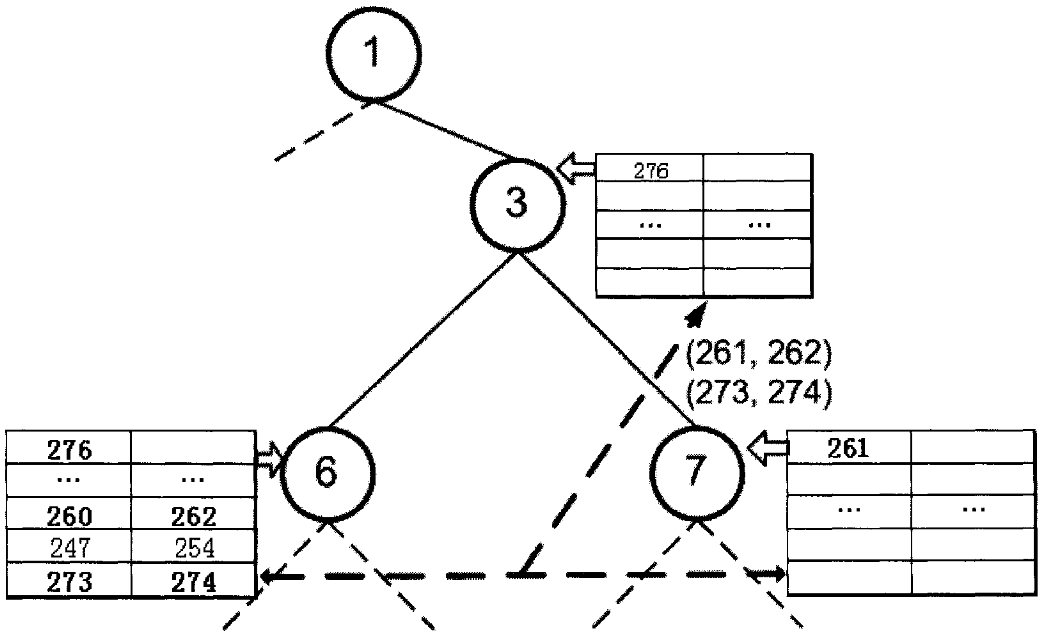

Figure 21 : Adding Collusion Time Periods to Node 3

At time *276*, the time of sibling node of node *54* (the last update time of the sibling node *7*) is *261*. As the original OFT scheme requires, the node *54* will have the knowledge of the blinded key of node *7* because node *7* is one of its sibling nodes. Although node *54* joined at time *276*, it has the key knowledge of node *7* since time *261*, because the key of node *7* has never been updated after time *261*. By comparing all timestamps in node *6* (in Table 2), we find the time period *260* → *262* in line *25* of Table 2 and *273* →*274* in line

27 of Table 2 overlap the time period of the sibling, that is *261* to now. The intersections are *261* → *262* and *273* → *274*. The time period *260* → *262* recorded in node *6* means the key of node *6* between time *260* and *262* is known by leaving group members. If the newly joined node *54* colludes with those leaving group members, it will be able to learn the key of node *6* between time *260* and *262*. Obviously, node *54* can combine his knowledge about the blinded key of node *7* after time *261*, and the blinded key of node *6* between *260* and *262* to get the key of node *3* between time *261* and *262*, which node *54* is not supposed to know. Following the same reasoning, node *54* can collude with leaving nodes to obtain the key of node *3* between time *273* and *274*. Therefore, the two collusion time periods for the key of node *3, 261* → *262* and *273* → *274* should be copied to the time table of node *3*, as shown in Figure 21.

Node *6* is one of the sibling nodes of node *29*. At time *277* (line *1* in Table 3), when node *29* left the group, time period *[277, -]* (line *3* in Table 3) has been inserted to the time table of node *6*, which means that if the key of node *6* doesn't change, node *29* always knows the key of node *6* after time *277*. At time *279* (line *7* in Table 3), node *48*, which is one of the subordinate nodes of node *6*, left the group; the key of node *6* is updated. One of the time period records in node *6*, *[277, -]* (line *3* in Table 3) has been updated to be *[277, 279]* (line *8* in Table 3). That means leaf node *29* can only know the key of node *6* from time *277* to *279* because node *6* changed its key at time *279* (Figure 22).

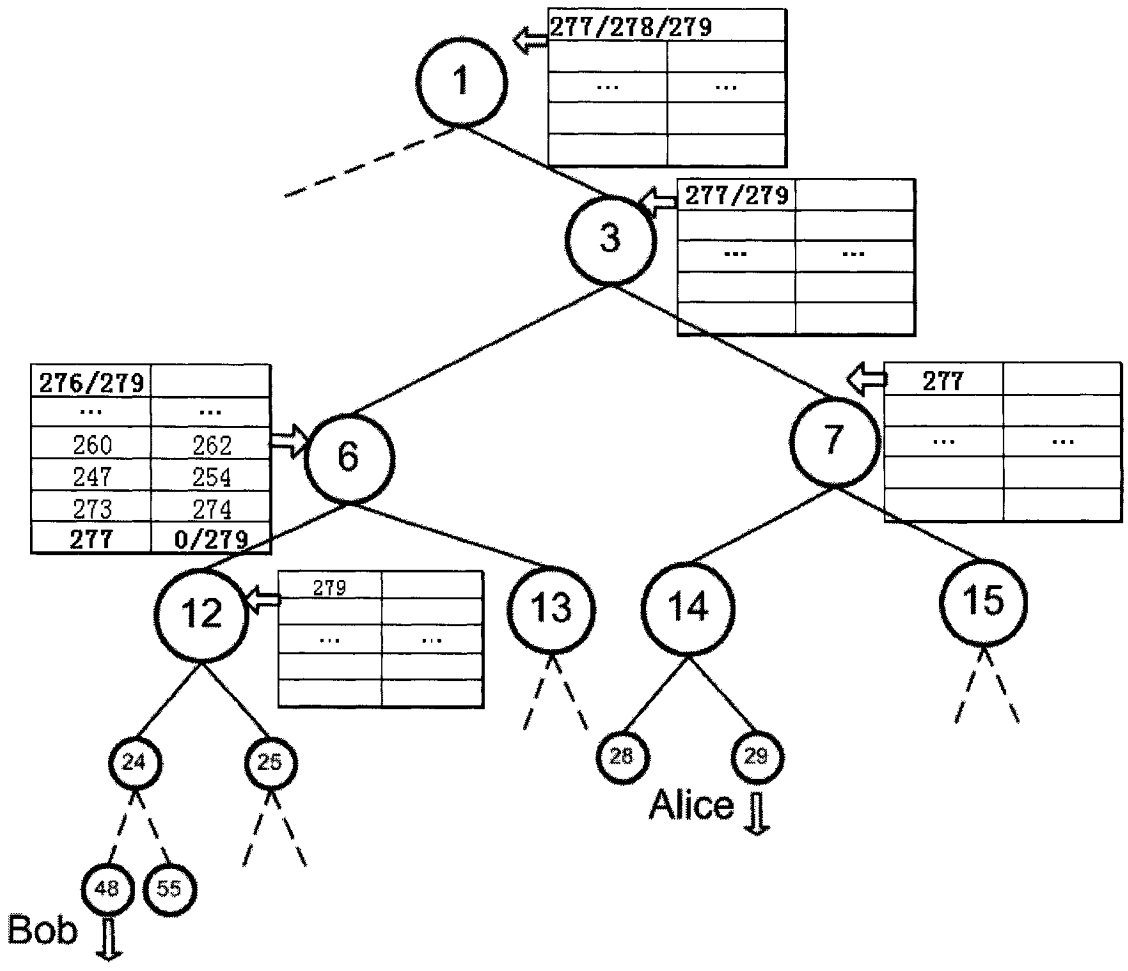

Figure 22 : Time Stamp Update at Node 6 When Node 48 and 29 Are Evicted

	Log of Group Update
1	At time **277**, node 29 left.
2	Node 29 left; update all its path node 7 info[24][1] to: 277
3	When node 29 left, update its sibling **node 6 info[27][0]--[1] to: 277--11**
5	When node 29 left, update its sibling node 2 info[67][0]--[1] to: 277--11
6	At time 278, new node joins node 14.
7	At time **279**, node 48 left.
8	Node 48 left; update all its path **node 6 info[27][1] to: 279**

Table 3 : Log of Group Update

	Time Stamps in Node 12	**Time**
1	At time **285**, new node joins node 24.	
2	Node 12--info[0][0]-[0][1] :	285 --> 0
3	Node 12--info[1][0]-[1][1] :	115 --> 119
4	Node 12--info[2][0]-[2][1] :	132 --> 134
5	Node 12--info[3][0]-[3][1] :	145 --> 150
6	Node 12--info[4][0]-[4][1] :	160 --> 163
7	Node 12--info[5][0]-[5][1] :	155 --> 159
8	Node 12--info[6][0]-[6][1] :	167 --> 171
9	Node 12--info[7][0]-[7][1] :	187 --> 189
10	Node 12--info[8][0]-[8][1] :	193 --> 194
11	Node 12--info[9][0]-[9][1] :	203 --> 205
12	Node 12--info[10][0]-[10][1] :	208 --> 220
13	Node 12--info[11][0]-[11][1] :	222 --> 226
14	Node 12--info[12][0]-[12][1] :	228 --> 233
15	Node 12--info[13][0]-[13][1] :	246 --> 254
16	Node 12--info[14][0]-[14][1] :	237 --> 238
17	Node 12--info[15][0]-[15][1] :	272 --> 274
18	**Node 12--info[16][0]-[16][1] :**	**280 --> 282**
19	**Node 12--info[17][0]-[17][1] :**	**283 --> 284**

Table 4 : Time Stamps in Node 12 at Time 285

Table 4 shows all the timestamps in node *12* at time *285*. Similar to the Table 2, there are two collusion time periods in the time table of node *12*. At time *285* (line *1* in Table 4), a new member joined node *24*, which is the child node of node *12*. When the checking for collusions arrives to the level of node *12*, the sibling node is *13* (line *20* in Table 4). The last updated time of the sibling node is *276* (line *20* in Table 4). By comparing timestamp

[276, -] and all timestamps in node *12*, we obtain two collusions: *[280, 282]* (line *18* in Table 4) and *[283, 284]* (line *19* in Table 4). That means the newly joined group member can collude with other leaving members to obtain the key of node *6* between time *[280, 282]* and *[283, 284]*. We need to add two timestamps to the time table of node *6* and update its sibling node *13* at the same time, as illustrated in Figure 23.

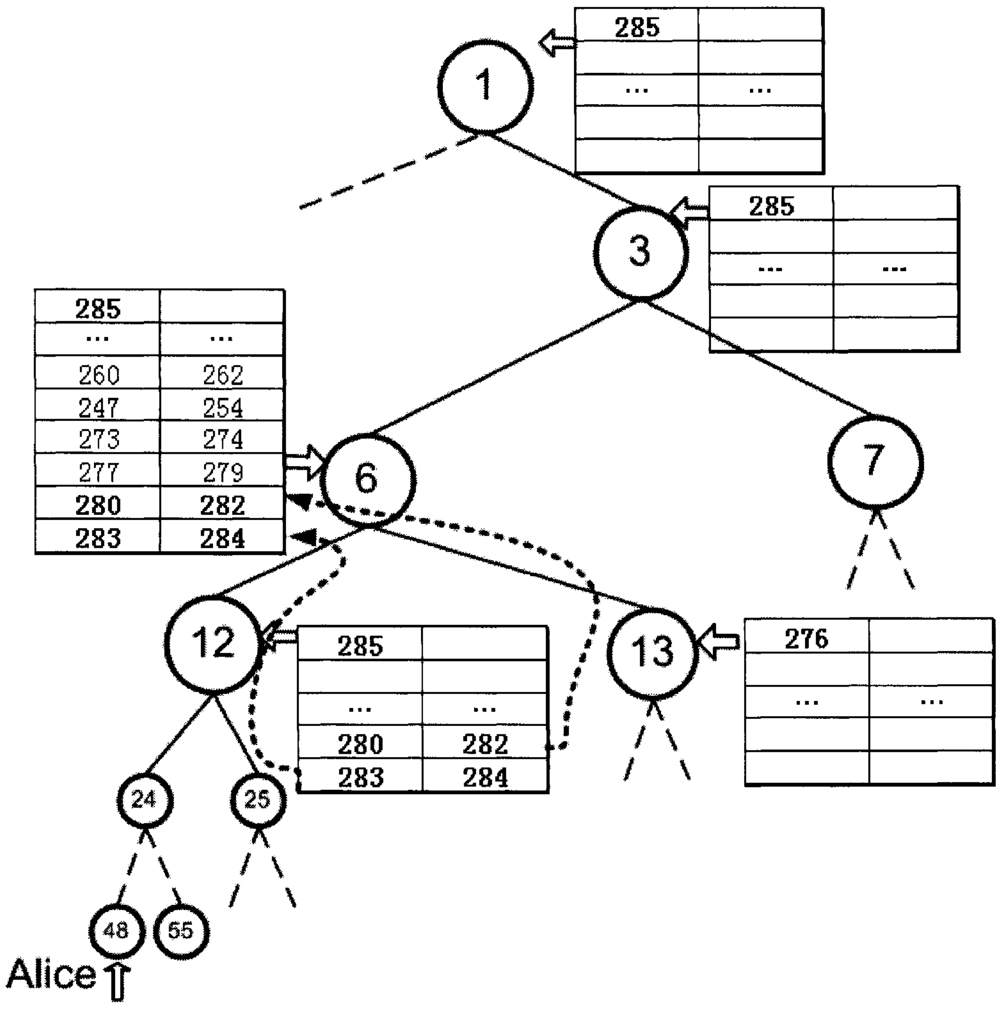

Figure 23 : Adding Collusion Time Period to Node 6

At this point, we can conclude that the collusion in *[280, 282]* (line *18* in Table 4) and *[283, 284]* (line *19* in Table 4) in node *6* have been propagated from node *12*.

	Time Stamps in Node 6	**Time**
1	Node 6--info[0][0]-[0][1] :	285 --> 0
2	Node 6--info[1][0]-[1][1] :	102 --> 109
3	Node 6--info[2][0]-[2][1] :	114 --> 115
4	Node 6--info[3][0]-[3][1] :	120 --> 125
5	Node 6--info[4][0]-[4][1] :	128 --> 132
6	Node 6--info[5][0]-[5][1] :	126 --> 132
7	Node 6--info[6][0]-[6][1] :	136 --> 137
8	Node 6--info[7][0]-[7][1] :	140 --> 145
9	Node 6--info[8][0]-[8][1] :	137 --> 145
10	Node 6--info[9][0]-[9][1] :	156 --> 159
11	Node 6--info[10][0]-[10][1] :	160 --> 163
12	Node 6--info[11][0]-[11][1] :	167 --> 168
13	Node 6--info[12][0]-[12][1] :	179 --> 180
14	Node 6--info[13][0]-[13][1] :	181 --> 183
15	Node 6--info[14][0]-[14][1] :	187 --> 189
16	Node 6--info[15][0]-[15][1] :	193 --> 194
17	Node 6--info[16][0]-[16][1] :	196 --> 199
18	Node 6--info[17][0]-17][1] :	203 --> 205
19	Node 6--info[18][0]-[18][1] :	206 --> 207
20	Node 6--info[19][0]-[19][1] :	212 --> 214
21	Node 6--info[20][0]-[20][1] :	225 --> 226
22	Node 6--info[1][0]-[1][1] :	239 --> 240
23	Node 6--info[22][0]-[22][1] :	245 --> 246
24	Node 6--info[23][0]-[23][1] :	249 --> 254
25	Node 6--info[24][0]-[24][1] :	260 --> 262
26	Node 6--info[25][0]-[25][1] :	247 --> 254
27	Node 6--info[26][0]-[26][1] :	273 --> 274
28	Node 6--info[27][0]-[27][1] :	277 --> 279
29	**Node 6--info[28][0]-[28][1] :**	**280 --> 282**
30	**Node 6--info[29][0]-[29][1] :**	**283 --> 284**

Table 5 : Time Stamps for Node 6 at Time 285

By comparing the Table 5 and Table 2, we find three more collusions in *[277, 279]* (line *28* in Table 5), *[280, 282]* (line *29* in Table 5), and *[283, 284]* (line *30* in Table 5). As we explained above, the *[277, 279]* is due to the fact that node *29* left group at time *277* and node *48* left at time *279*. On the contrary, the time periods *[280, 282]*, and *[283, 284]* are inherited from node *12* when node *48* joins the group at time *285*.

4.4 Experimental Results

In this section, we compare the average communication overhead of our extended solution to the LKH algorithm, the original OFT scheme, and the modified OFT scheme proposed by Ku and Chen. It is clear that our extended solution will be more efficient compared to the modified OFT scheme proposed by Ku and Chen in terms of communication cost. We expect our method to be less efficient than the original OFT scheme since we need to prevent collusion attacks. The solution proposed in Chapter 3 secures all subgroup keys and as a result, it is more efficient than LKH for small groups but is less efficient for large groups. On the other hand, the solution introduced in this chapter only secures the group key. Therefore, we expect this new solution to exhibit better performance. Figure 24 shows the broadcast size versus the key tree size.

Compared to LKH, our solution is always more efficient in terms of communication cost, regardless of the size of the group. Similar to the solution in Chapter 3, the extended solution is slightly less efficient than the original OFT. That is obvious because the extended solution is based on the original OFT algorithm but tries to fix the security problem of original OFT; therefore, it will incur communication overhead.

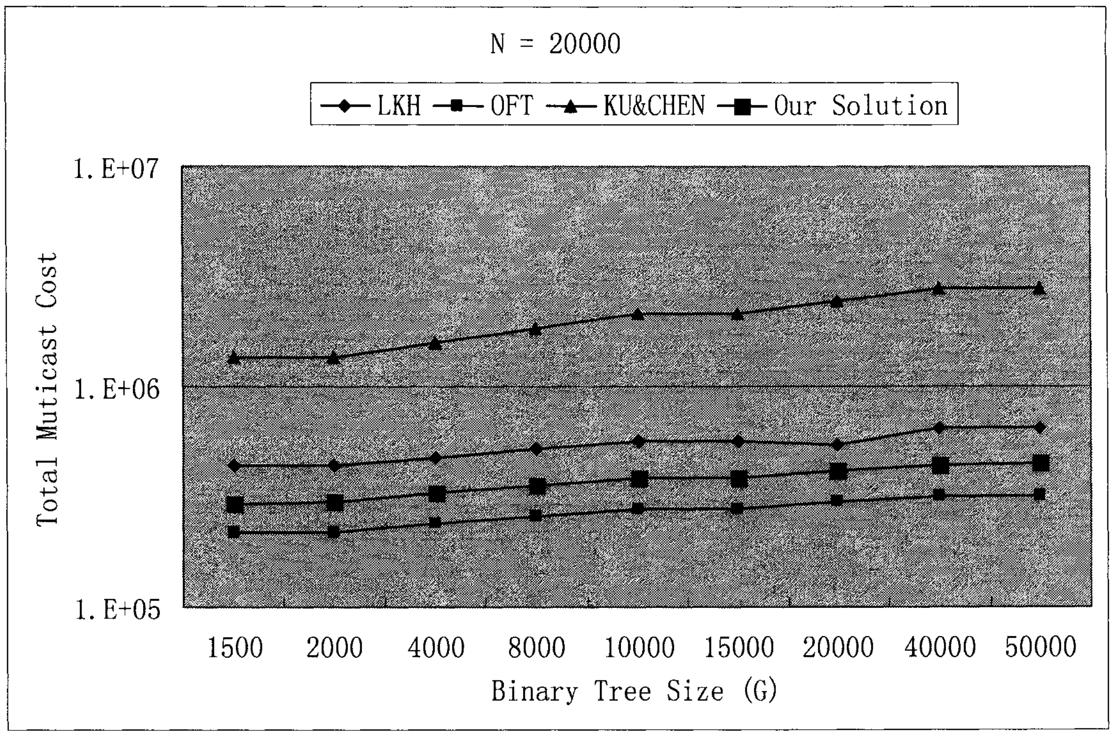

Figure 24 : Broadcast Size vs. Key Tree Size

Table 6 shows a more detailed comparison between the two schemes: LKH and the extended solution.

Key Tree Size	1500	2000	4000	8000	10000	15000	20000	40000	50000
Extended Solution/LKH	0.68	0.68	0.69	0.69	0.69	0.69	0.69	0.69	0.70
Collusion Times	7125	7353	7584	7680	7605	7782	7676	7738	7895
Tree Height	11	11	12	13	14	14	15	16	16

Table 6 : Comparing Broadcast Size of Extended Solution to LKH

Even for large groups where forward and backward security is important, we still can use the extended solution to replace LKH algorithm because it has lower rekeying communication cost than LKH.

Figure 25 shows the total broadcast size versus the number of operations, with about half of the operations being evictions, in a key tree with *10000* keys. The result is similar to the solution in Chapter 3. The only difference is that in this extended solution, the total broadcast size is always smaller than in the LKH. Figure 25 also shows the broadcast size versus the degree of group dynamics. The broadcast size of all four schemes increases along with the number (intensity) of operations. The original OFT scheme, the LKH scheme, and the extended solution all scale in roughly the same manner, whereas the scheme proposed by Ku and Chen is less scalable.

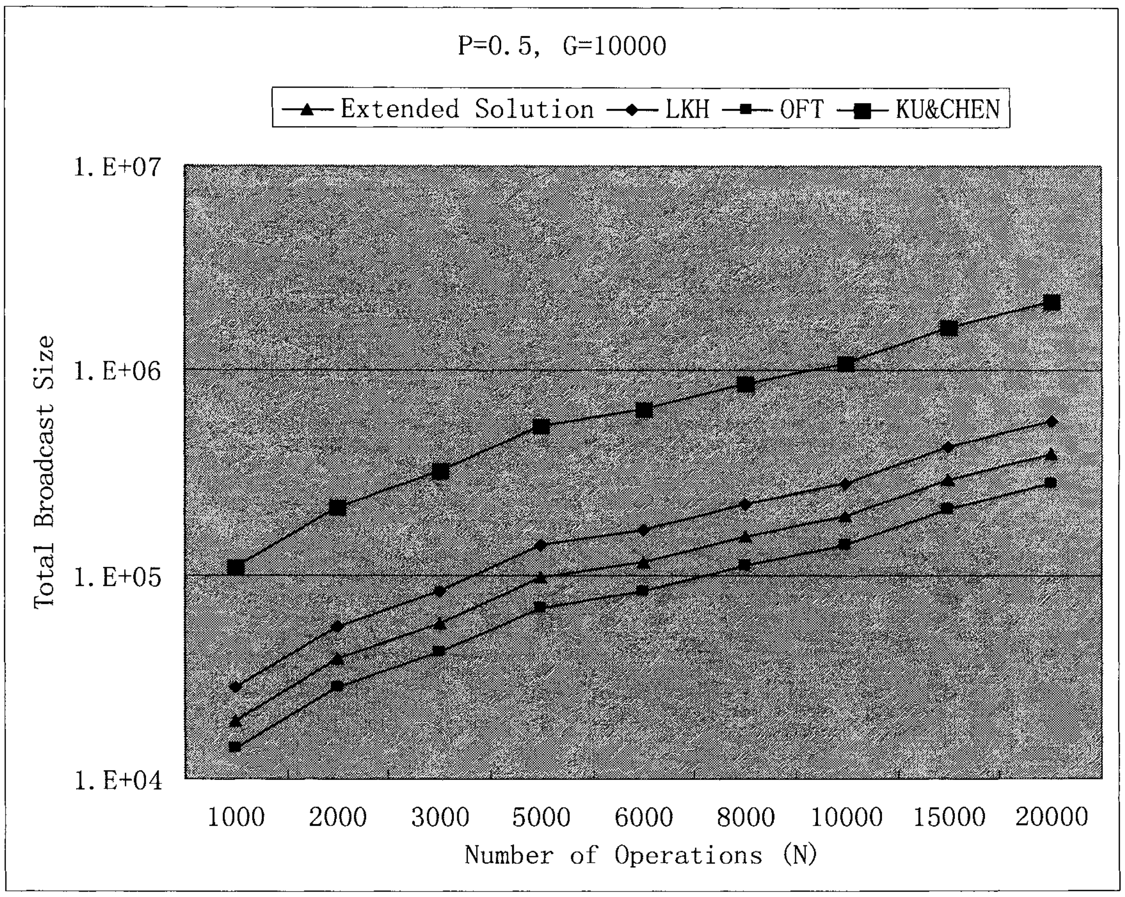

Figure 25 : Broadcast Size vs. Number of Operations

Figure 26 and Figure 27 show the total broadcast size versus the ratio of evictions among all operations. The two experiments differ in the key tree size and the total number of performed operations. In both experiments, the OFT scheme and the LKH scheme have a constant broadcast size because in both schemes, the joining and eviction require the same amount of keys to be broadcasted. The broadcast size of the scheme proposed by Ku and Chen increases linearly in the ratio of eviction because their scheme requires additional key updates, hence additional broadcasted bits on every eviction operation (but not on the joining operation). Regarding the results, the only difference between these two solutions (that is, securing group key or securing subgroup key) is that the extended solution (securing group key) always has less communication cost than LKH, no matter what the eviction ratio is.

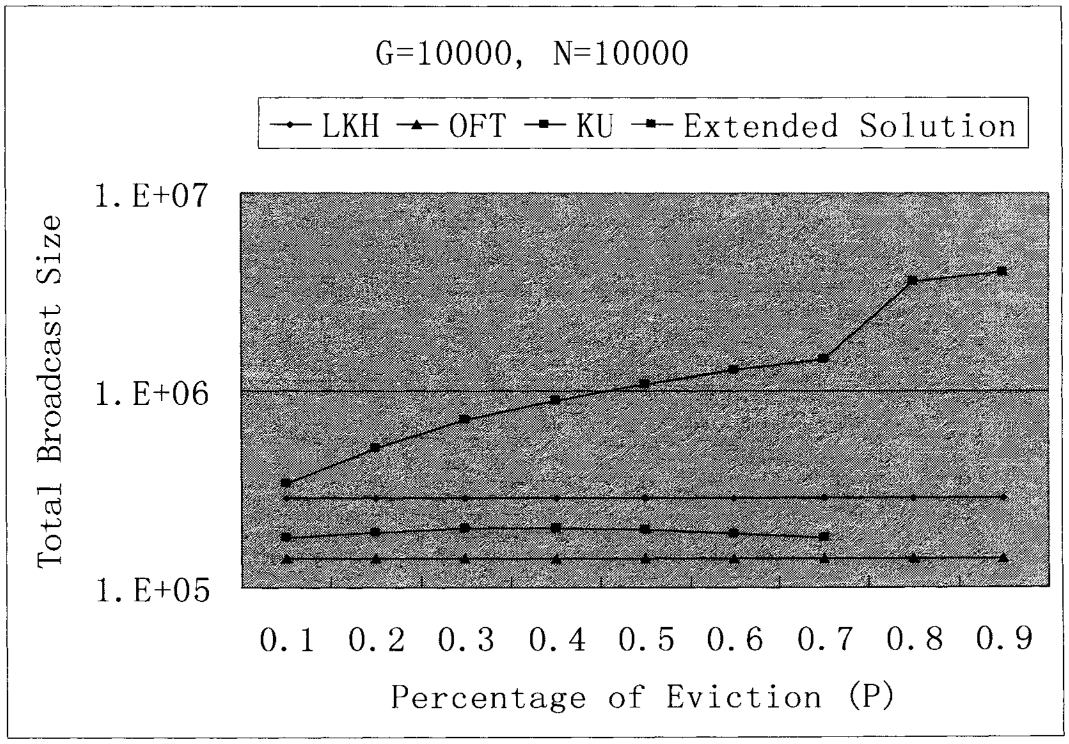

Figure 26 : Broadcast Size vs. Ratio of Eviction (Case 1)

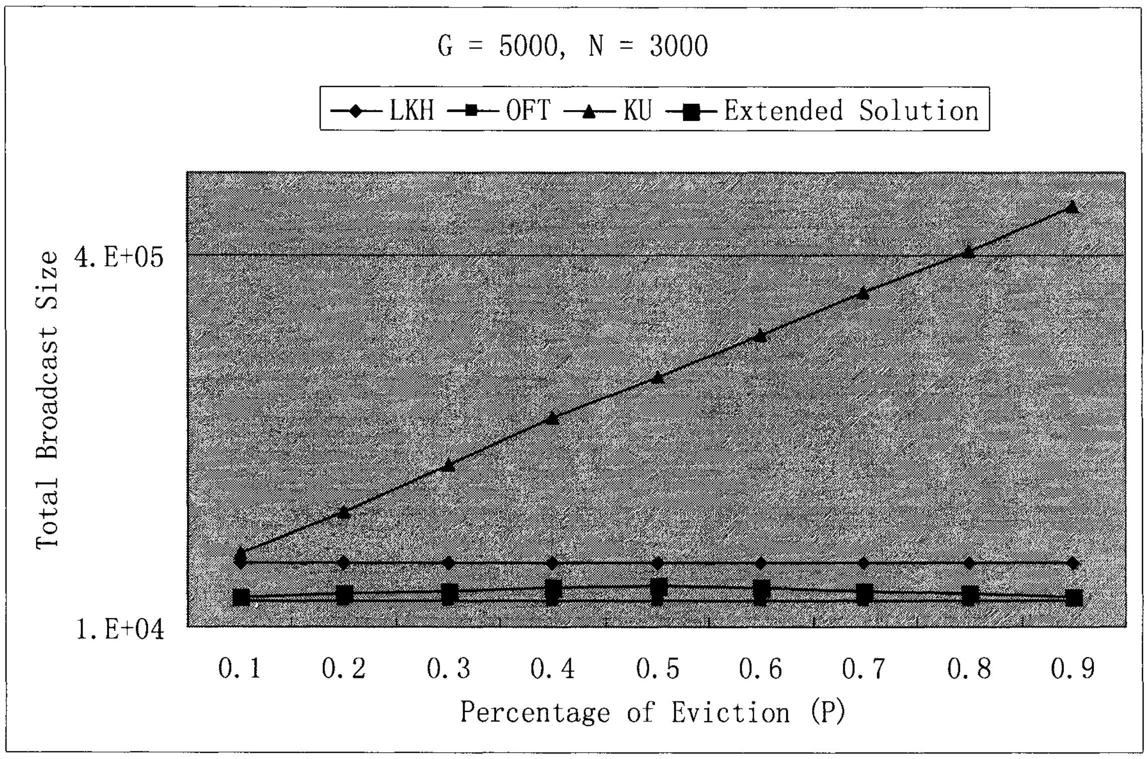

Figure 27 : Broadcast Size vs. Ratio of Eviction (Case 2)

Chapter 5

Conclusion and Future Work

In this thesis, we studied collusion attacks on the one-way function tree (OFT) scheme. The OFT scheme achieves a halving in broadcast size in comparison to the LKH scheme. However, OFT's approach of using functionally dependent keys in the key tree also renders the scheme vulnerable to collusion attacks between evicted members and joining members. Throughout this thesis,

- We have generalized previous observations made by Horng and Ku *et al.* [46] into a generic collusion attack on OFT. This generalization also gave a necessary and sufficient condition for the collusion attack on OFT. The previous work by Horng and Ku *et al.* have only described specific examples of collusion attacks involving two or three colluding nodes but left the general case open [22]. Our results show exactly what can be computed by an arbitrary collection of joining and evicted nodes.
- Based on this condition, we have proposed a modified OFT scheme. The scheme is immune to the collusion among an arbitrary number of joining and evicted members.

It also minimizes the broadcast size for each operation. The scheme secures the group key and all subgroup keys which can achieve perfect forward and backward security. The scheme has a storage requirement proportional to the size of the key tree. Experiments show that our scheme has smaller communication overhead than the LKH scheme for small to medium groups. For large groups, the increasing number of collusions renders the OFT scheme a less efficient choice than LKH.

- By relaxing the security requirement for our algorithm, we presented an extended solution that secures only the group key. This solution is more efficient than LKH algorithm in any case no matter how many members the groups have.

Some of the results presented in chapter 3 have been published in [47].

Based on the research elaborated in this thesis, further studies can be conducted in the following directions:

- When the compromise of some sub-group or group keys happens, one may not only update the compromised key, but also figure out who collude with whom, and during which specific time period the key was compromised or exposed. After getting such detailed information, we can twist our algorithm to achieve higher efficiency.

- We can apply OFT to key graph or N degrees key tree, study the general collusion attack for OFT on key graph or N degrees key tree and propose a secure rekeying algorithm based on OFT on key graph or N degrees key tree.

- The performance of the proposed solutions can be studied under different statistical models of the eviction and joining operations.

- The proposed solution can be studied in the decentralized setup.

References

[1] D. M. Balenson, D. A. McGrew, and A. T. Sherman, "Key Management for Large Dynamic Groups: One-Way Function Trees and Amortized Initialization," InternetDraft (work in progress), Internet Engineering Task Force, draft-irtf-smug-groupkeymgmt-oft-00.txt., August 2000.

[2] M. Baugher, R. Canetti, L. Dondeti, and F. Lindholm, "Multicast Security (MSEC) Group Key Management Architecture," RFC 4046, Network Working Group, IETF, April, 2005.

[3] B. Briscoe, "MARKS: Zero Side Effect Multicast Key Management Using Arbitrarily Revealed Key Sequences," First International Workshop on Networked Group Communication (NGC), Pisa, Italy, Pages 301-320, November 1999.

[4] R. Canetti, J. Garey, G. Itkis, D. Micciancio, M. Naor, and B. Pinkas, "Multicast Security: A Taxonomy and Efficient Constructions," in proceedings of IEEE InfoComm'99, Volume 2, Pages 708-716, March 1999.

[5] R. Canetti, P. Rohatgi, and P. Cheng, "Multicast Data Security Transformations: Requirements, Considerations, and Proposed Design," draft-irtf-smugdata-transforms-00.txt, IRTF, work in progress, June 2000.

[6] R. Canetti and B. Pinkas, "A Taxonomy of Multicast Security Issues," dracanetti-securemulticast-taxonomy-00.txt, IETF Internet Draft (work in progress), 1998.

[7] I. Chang, R. Engel,D. Kandlur, D. Pendarakis, and D. Saha, "Key Management for Secure Internet Multicast Using Boolean Function Minimization Techniques," in proceedings of IEEE INFOCOM, New York, March 1999.

[8] W. Chen, and L. R. Dondeti, "Recommendations In Using Group Key Management Algorithms," In Proceedings of DARPA Information Survivability Conference and Exposition 2003, Volume 2, Pages 222-227, April 2003.

[9] B. DeCleene, L. Dondeti, S. Griffin, T. Hardjono, D. Kiwior, J. Kurose, D. Towsley, S. Vasudevan, and C. Zhang, "Secure Group Communications For Wireless Networks," in proceedings of the MILCOM, June 2001.

[10] W. H. Desmond Ng, M. Howarth, Z. Sun, and H. Cruickshank, "Dynamic Balanced Key Tree Management for Secure Multicast Communications," IEEE Transactions on Computers, Volume 56, No.5, Pages 590-605, May 2007.

[11] W. H. Desmond Ng, H. Cruickshank, and Z. Sun, "Scalable Balanced Batch Rekeying for Secure Group Communication," Computers & Security, 25(4), Pages 265-273, 2006.

[12] P. T. Dinsmore, D. M. Balenson, M. Heyman, P. S. Kruus, C. D. Scace, and A. T. Sherman, "Policy-Based Security Management for Large Dynamic Groups: An Overview of the DCCM Project," in proceedings of the DARPA Information Survivability Conference & Exposition, Vol. I of II (DISCEX), Hilton Head, SC, Pages 64-73, January 2000.

[13] C. Diot, B. N. Levine, B. Lyles, H. Kassem, and D. Balensiefen, "Deployment Issues for the IP Multicast Service and Architecture," IEEE Network, Special Issue on Multicasting, January/February 2000.

[14] J. Fan, P. Judge, and M. Ammar, "HySOR: Group Key Management with Collusion-Scalability Tradeoffs Using a Hybrid Structuring of Receivers," in proceedings of the IEEE International Conference on Computer Communications Networks, Miami, 2002.

[15] T. Hardjono, R. Canetti, M. Baugher, and P. Dinsmore, "Secure IP Multicast: Problem areas, Framework, and Building Blocks," draft-irtf-smug-framework-01.txt, Secure Multicast Group (SMuG) of the IRTF, September, 2000

[16] T. Hardjono and L. R. Dondeti, Multicast and Group Security, Artech House, 2003.

[17] T. Hardjono and B. Weis, "The Multicast Group Security Architecture," RFC 3740, Network Working Group, IETF, March, 2004

[18] H. Harney, A. Colegrove, E. Harder, U. Meth, and R. Fleischer, "Group Secure Association Key Management Protocol," draft-ietf-msec-gsakmp-sec-00.txt, IETF, work in progress, March 2001.

[19] H. Harney and E. Harder, "Logical Key Hierarchy Protocol," Internet Draft (work in progress), draft-harney-sparta-lkhp-sec-00.txt, Internet Engineering Task Force, Mar. 1999.

[20] H. Harney, C. Muckenhirn, and T. Rivers, "Group key management protocol architecture," IETF, RFC2093, 1997.

[21] M. H. Heydari, L. Morales, and I. H. Sudborough, "Efficient Algorithms for Batch Re-Keying Operations in Secure Multicast," in proceedings of the 39th Annual Hawaii International Conference on System Sciences, HICSS 2006, Volume 9, Pages 218b-218b, January 2006.

[22] G. Horng, "Cryptanalysis of a Key Management Scheme for Secure Multicast Communications," IEICE Trans. Comm., Volume E85-B, No. 5, Pages 1050-1051, 2002.

[23] H. Khurana, R. Bonilla, A. Slagell, R. Afandi, H. S. Hahm, and J. Basney, "Scalable Group Key Management with Partially Trusted Controllers," in proceedings of International Conference on Networking, 2005.

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

[25] D. Kwak, S. Lee, J. Kim, and E. Jung, "An Efficient Key Tree Management Algorithm for LKH Group Key Management," The International conference on Information Networking 2006 (ICOIN 2006), Springer-Verlag Lecture Notes in Computer Science, Volume 3961, Pages 703-712, January 2006.

[26] M. Li, R. Poovendran, and D. McGrew, "Minimizing center key storage in hybrid one-way function based group key management with communication constraints," Information Processing Letters, Volume 93 , Issue 4, Pages 191-198, February 2005.

[27] J. C. Lin, F. Lai, and H. C. Lee, "Efficient Group Key Management Protocol with One-Way Key Derivation," in proceedings of The 2005 IEEE Conference on Local Computer Networks, Pages 336-343, 2005.

[28] X. X. Liu, M. Yang, and X. K. Wang, "Key Management for Secure Multicast Communication Using Secret Sharing-Based Revocation Scheme," ISCIT 2005, IEEE International Symposium on Communications and Information Technology, Volume 2, Pages 1309-1313, October 2005.

[29] D. Matthew, J. Moyer, J. R. Rao, and P. Rohatgi, "A Survey of Security Issues in Multicast Communications," IEEE Network Magazine, November/December 1999.

[30] P. McDaniel, and A. Prakash, "Ismene: Provisioning and Policy Reconciliation in Secure Group Communication," Technical Report CSE-TR-438-00, Electrical Engineering and Computer Science, University of Michigan, December 2000.

[31] D. McGrew, A. David, T. Alan, and A. Sherman, "Key establishment in large dynamic groups using one-way function trees," TIS Report no.0755, TIS Labs at Network Associates, Inc., Glenwood, MD, 1998.

[32] A. J. Menezes, P. C. Oorschot, and S. A. Vanstone, Handbook of Applied Cryptography, CRC Press, 1997.

[33] D. L. Mills, "Network Time Protocol (Version 3) Specification, Implementation and Analysis" RFC 1305, Network Working Group, IETF, March 1992.

[34] S. Mittra, ''Iolus: A Framework for Scalable Secure Multicasting,'' in proceedings of ACM SIGCOMM, Cannes, France, Pages 277-288, September 1997.

[35] M. J. Moyer, J. R. Rao, and P. Rohatgi, "A Survey of Security Issues in Multicast Communications," IEEE Network, Pages 12-23, November/December 1999.

[36] G. Noubir, F. Zhu, and A. H. Chan, "Key Management for Simultaneous Join/Leave in Secure Multicast," in proceedings of 2002 IEEE International Symposium on Information Theory, Pages 325-, 2002.

[37] K. Peter, "A Survey of Multicast Security Issues and Architectures," in proceedings of 21st National Information Systems Security Conference, Pages 408-420, Arlington, VA, October 1998.

[38] S. Rafaeli and D. Hutchison, "A survey of key management for secure group communication," ACM Computing Surveys, Volume 35, No. 3, Pages 309-329, September 2003.

[39] S. Setia, S. Koussih, S. Jajodia, and E. Harder, ''Kronos: A Scalable Rekeying Approach for Secure Multicast,'' in proceedings of IEEE Symposium on Security and Privacy, Oakland, CA, May 2000.

[40] 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., 2002.

[41] A. T. Sherman, and D. A. McGrew, "Key establishment in large dynamic groups using one-way function trees," IEEE Transactions on Software Engineering, Volume 29, Issue 5, Pages 444-458, May 2003.

[42] M. Waldvogel, G. Caronni, D. Sun, N. Weiler, and B. Plattner, ''The VersaKey Framework: Versatile Group Key Management,'' IEEE JSAC Special Issue on Service Enabling Platforms For Networked Multimedia Systems, Vol. 17, No. 9, September 1999.

[43] D. Wallner, E. Harder, and R. Agee, ''Key Management for Multicast: Issues and Architectures,'' RFC 2627(informational), IETF, June 1999.

[44] Y. Wang, J. Li, L. Tie, and Q. Li, "An Efficient Key Management for Large Dynamic Groups," CNSR, pp. 131-136, Second Annual Conference on Communication Networks and Services Research (CNSR'04), 2004.

[45] Y. Wang, J. Li, L. Tie, and H. Zhu, "An efficient method of group rekeying for multicast communication," in proceedings of the 6th IEEE Circuits and Systems Symposium, Pages 273-276, June 2004.

[46] C. K. Wong, M. Gouda, and S. S. Lam, ''Secure Group Communications Using Key Graphs,'' IEEE/ACM Trans. on Networking, Volume 8, No. 1, Pages 16–30, February 2000.

[47] X. X. Xu, L.Y. Wang, A. Youssef, B. Zhu, "Preventing collusion attacks on the one-way function tree (OFT) scheme," Proc. 5th International Conference on Applied Cryptography and Network Security (ACNS 2007), Springer-Verlag Lecture Notes in Computer Science, Vol. 4521, pages 177-193, June 5-8, 2007.

[48] S. Xu, Z. Yang, Y. Tan, W. Liu, and S. Sesay, "An Efficient Batch Rekeying Scheme Based On Oneway Function Tree," in proceedings of The IEEE International Symposium on Communications and Information Technology, Pages 490- 493, 2005.

[49] Y. R. Yang, X. S. Li, X. B. Zhang, and S. S. Lam, ''Reliable Group Rekeying: Design and Performance Analysis,'' in proceedings of ACM SIGCOMM, San Diego, CA, August 2001.

[50] X. B. Zhang, S. S. Lam, D. Y. Lee, and Y. R. Yang, ''Protocol Design for Scalable and Reliable Group Rekeying,'' IEEE/ACM Transactions on Networking (TON), Volume 11, Issue 6, Pages 908-922, 2003

[51] J. Zhang, Y. Zhou, F. Ma, D. Gu, and Y. Bai, "An Extension of Secure Group Communication Using Key Graph," Information Sciences, Volume 176, Issue 20, Pages 3060-3078, October 2006.

[52] S. Zhu and S. Jajodia, "Scalable Group Rekeying for Secure Multicast: A Survey," in proceedings of 5th International Workshop on Distributed Computing, LNCS, Volume 2918, Pages 1 – 10, 2004.

[53] S. Zhu, S. Setia, and S. Jajodia, "Performance Optimizations for Group Key Management Schemes," in proceedings of 23rd International Conference on Distributed Computing Systems, Pages 163 – 171, May 2003.