University of Alberta

MULTILEVEL ACCESS CONTROL AND KEY MANAGEMENT IN SCALABLE LIVE STREAMING

by

Xingyu Li

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science
in
Digital Signals and Image Processing

Department of Electrical and Computer Engineering

Spring 2010
Edmonton, Alberta

Examining Committee

Dr. H.Vicky Zhao (co-supervisor), Electrical and Computer Engineering

Dr. Robert J.Davies (co-supervisor), TRLabs

Dr. Herb Yang, Computing Science

Dr. Hai Jiang, Electrical and Computer Engineering

Abstract

To prevent unauthorized access of multimedia content, effective access control and efficient key management schemes are of crucial importance for multimedia live streaming applications. To address the network heterogeneity, this thesis investigates scalable and multilevel access control, where a single bit stream can offer access to different levels of service. It uses the POSET hash-based structure to explore the data dependency between different layers in scalable coding, and proposes a secure and efficient key management scheme that uses public information and hash functions to reduce the rekey overhead. For those applications that require strict forward/backward security, sequential rekey is used to update keys. This thesis also addresses frequent membership update in live streaming, and considers applications that can tolerate a small amount of content leak. Through exploring their unique characteristics in membership dynamics, the proposed schemes reduce the rekey cost significantly.

Acknowledgements

Firstly, I am deeply grateful to my supervisors Dr. H.Vicky Zhao and Dr. Robert J. Davies for providing me this valuable opportunity to work with them and for training me as a M.Sc. student at University of Alberta. Indeed, I would like to thank Dr. H.Vicky Zhao for providing the valuable feedback that has gone into shaping this thesis, as well as my approach to research, and for always encouraging me to seek the best path for me in future.

Next, I would like to thank all members in the multimedia laboratory for their support and friendship. Also, I would like to thank all my friends at University of Alberta for their encourage and advice.

I owe a great deal to my family. I definitely would not be where I am today without their gifts of positivity, understanding, support and everlasting love.

Last but not least, I gratefully acknowledge the financial support of TRLab, the Department of Electrical and Computer Engineering at University of Alberta, and my supervisors.

Contents

List of Tables

List of Figures

Acronyms

Acronyms	Definition
ALX	Non-fixed-degree balanced tree
AVL	Self-balancing binary search tree
DG	Data Group
DH	Diffie-Hellman key exchange protocol
DRM	Digital rights management
ECC	Elliptic curve cryptography
GDH	Group DH key exchange
InS	Independent key management scheme
IPTV	Internet protocol television
JDH	Join-Tree-based Contributory Group Key Management
JET	Join-Exit-Tree
KDC	Key distribution center
KEK	Key encryption key
LKH	Logical key hierarchy tree
LKH+	Improved logical key hierarchy tree
MGS	Multi-group key management scheme
OFCT	One-way function chain tree
OFT	One-way function tree

POSET	Partially ordered set
QoS	Quality of service
RECC	Routing-driven ECC-based key management scheme
SG	Service Group
SH	High layer (enhancement layer) in spatial scalability
SK	Session key
SL	Low layer (base layer) in spatial scalability
SNR	Signal to noise ratio
TH	High layer (enhancement layer) in temporal scalability
TL	Low layer (base layer) in temporal scalability
TOFT	Threshold-based one-way function tree
WFL	Wait-for-leaving

List of Symbols

Symbol	Definition
$a_{\mathbf{I}}$	ALX tree degree for the $\mathbf{I}^{th}$ SG
C_{de}	user's decryption overhead per second
C_{en}	KDC encryption overhead per second
$C(k)$	average number of rekey messages per one user departure
C_{msg}	communication overhead per second
$d^{\mathbf{I}}$	public label of vertex $\mathbf{I}$
$k_{\mathbf{I}}$	group size of SG $\mathbf{I}$
$\mathbf{K}_e$	key encryption key
$\mathbf{K}_g$	group key
$\mathbf{K}_p$	private key
$\mathbf{K}_s$	session key
$n_{\mathbf{I}}$	the number of parent vertex $\mathbf{I}$ has
$P_{\mathbf{I}}^{re}$	probability of a leaving user rejoining the $\mathbf{I}^{th}$ SG within T_l sec
$P_{\mathbf{I}}'$	probability of a leaving user rejoining the $\mathbf{I}^{th}$ SG before next batch moment
$Q_{\mathbf{I}}$	key set associated with vertex $\mathbf{I}$
$\mathbf{R}_h$	parameters for high dynamics update rate
$\mathbf{R}_l$	parameters for low dynamics update rate
T_l	batch interval

$t_{u_i^{\mathbf{I}}}$	user $u_i^{\mathbf{I}}$'s leaving time
$u_i^{\mathbf{I}}$	the i^{th} users in the $\mathbf{I}^{th}$ SG
$v^{\mathbf{I},\mathbf{J}}$	public value of the arrow between vertexes **I** and **J**
$\alpha_{\mathbf{I}}$	probability of reconnection rate per leaving user in the $\mathbf{I}^{th}$ SG
$\lambda_{\mathbf{I}}$	user arrival rate for the $\mathbf{I}^{th}$ SG
$1/\mu_{\mathbf{I}}$	user average staying time in the $\mathbf{I}^{th}$ SG
$1/\nu_{\mathbf{I}}$	average interval between a user's leave and rejoin for the $\mathbf{I}^{th}$ SG
η	ratio of the rekey overhead between two schemes
$\tau_{u_i^{\mathbf{I}}}$	user $u_i^{\mathbf{I}}$'s absent time

Chapter 1

Introduction

Advances in communication, networking and digital media technologies have led to the proliferation of real-time live streaming applications, for example, the internet protocol television (IPTV). Different from the download-and-play applications, a user starts playing the video while he/she is still downloading the data. Therefore, it enables users to enjoy multimedia on the fly. In general, there are two different types of live streaming applications, (audio) video on demand and live broadcast. On-demand data are stored in a content server and transmitted at subscribers' requests. Live broadcast broadcasts multimedia data to a group of users only at certain predetermined time, for example, live broadcast of an NBA game, and a user who subscribes to a live broadcast stream has little control on the streaming session except the ability to join and leave [2].

Since digital data is easy to copy and distribute, it provides adversaries with the capability to modify and illegally redistribute the media content, which violate the intellectual property rights of the content owner. It raises a critical and vital issue to protect multimedia content and to enforce digital right management in live streaming.

1.1 Access Control in Live streaming

Access control ensures that only authorized users can access the transmitted data and thus protects multimedia content confidentiality. It encrypts the transmitted data with a secret key shared by all users in the group, which prevents unauthorized users without the secret key from accessing the content.

Multimedia files have large volumes, and are often compressed before they are stored or transmitted. To ensure the multimedia content security, the compressed bit stream needs to be encrypted before transmission. A straightforward solution to encrypt multimedia is to apply generic encryption to multimedia data before compression. However, encryption often changes the statistical characteristics of multimedia data, and thus results in significant reduction in compression efficiency [3]. Another simple solution is to compress multimedia data first and then encrypt the compressed bit stream. This solution destroys the syntax structures (e.g. headers and markers) in the original bit stream that are important for QoS management at the intermediate nodes, and therefore disables the ability of the stream to adapt to network and device heterogeneity. Though it is possible to allow the intermediate nodes to decrypt the encrypted bit stream, process the data, and then re-encrypt the bit stream to operate transcoding and congestion control when the network condition becomes poor, it enables the intermediate nodes to access the content, which often violates the security requirements. To address this challenge, a promising solution is to jointly consider encryption and compression. To support secure and universal access of the multimedia, format-compliant encryption (or syntax-aware encryption) [4], [5] takes special care to maintain the syntax structure of the multimedia compression standard during encryption. It enables the intermediate nodes to easily identify the structure in the encrypted bit stream and to process the bit stream without decrypting the bit stream. Many format-compliant encryption techniques have been proposed in the literature, for example, the index-mapping

encryption [3], [6] and shuffling-based encryption [7], [8].

In group-oriented applications such as live broadcast applications, users may join/leave the service at any time, and frequent membership update is often observed. Following previous works, it is assumed that the server knows the join/leave time of each user. For such group-oriented applications, forward and backward security is of crucial importance. Forward security prevents leaving users from accessing the future content after they leave, and backward security ensures that the new users can only access the content after they join [9], [10], [11]. When there is a membership update, the encryption keys need to be updated, and another important issue in access control is secure and efficient key update.

This thesis investigates secure and efficient key management schemes in live broadcast applications. This chapter discusses challenges in key management in live streaming, and briefly summarizes the main contribution of this thesis.

1.2 Challenges in Key Management in Live streaming

In multimedia multicast applications, subscribers' bandwidth and their receiving devices may vary drastically. For example, some users with powerful PCs and high-definition displays may have high-speed internet, while users who use cell phones can only access mobile networks with low transmission rates. To address this heterogeneity, a straightforward solution is to encode the same video into multiple streams with different bit rates. Since multiple bit streams of the same content are transmitted over the network simultaneously, this approach leads to a significant increase in the overall bit rate. A more efficient way to address the heterogeneity in network and user devices is to use scalable coding [12], [13]. It decomposes video into layers of different priority in one bit stream, and one bit stream can provide different levels of resolution/quality simultaneously. The lowest layer, called the base layer, conveys basic information of the content, and the enhancement layers pro-

vide gradual quality refinements. With scalable coding, users with low bandwidth receive part of the transmitted bit stream and recover a low resolution copy. For users who have sufficient bandwidth to receive the entire bit stream, they reconstruct the high resolution version. In addition, in scalable coding, one single bit stream may support different types of scalability simultaneously, for example, spatial, temporal, and SNR scalability. Spatial scalability is defined as the representation of the same video in different sizes or resolutions; temporal scalability represents a video in different frame rates; and SNR scalability is defined as the representation of a video with different quality versions [14].

In live broadcast applications that use scalable video coding, to ensure that only authorized users can access the service and to support the business model "what you see is what you pay" [15], multilevel access control and scalable digital rights management (DRM) are often desired. It encrypts different layers of data with different secret keys, called session keys. The session keys are distributed appropriately so that users only have the keys to access what they subscribe to. Note that there exists strong data dependency between different layers in scalable coding, and an enhancement layer is useful only if all lower layers are decoded correctly. Consequently, the secret keys of layers in multilevel access control are not independent, and a user who has a key for a higher layer should also have the keys for all lower layers. Thus, a challenging issue in multilevel access control for scalable live streaming is to address this data dependency.

In addition, recent studies show that live broadcast applications have frequent membership updates with unique characteristics [16], [17], [18], [19]. For example, the total number of users may vary drastically from time to time; some users may temporarily leave the service due to network congestion or poor quality of the received video and then come back later; and others may only stay in the service for a very short time. To achieve forward and backward security, the session keys have to be updated when there is a membership update. These unique membership dynamics in live broadcast applications may cause significant

communication and computation overhead for key update. Thus, another challenge in access control for live streaming is to address this frequent membership update and design secure and efficient key management schemes.

1.3 Thesis Outline and Contributions

This thesis considers live broadcast applications where a single server distributes video to a group of users, and focuses on the scenario where a single bit stream supports multiple types of scalability simultaneously. It addresses the challenging issues in key management, and designs a secure and efficient key update scheme.

Chapter 2 reviews recent works on access control and key management. Chapter 3 introduces the system model, including the POSET structure used to explore the data dependency in scalable coding, and the characteristics of membership dynamics in live streaming. In Chapter 4, a POSET Hash-based key management scheme is proposed. It uses the special structure of the Hasse diagram, the public information and hash functions in POSET to significantly reduce the rekey cost. The thesis considers two types of applications: those that require strict forward and backward security, and those that can tolerate a small amount of content leak. For applications that require strict forward and backward security, sequential rekeying is proposed to be adopted to update the keys. For applications that can tolerate a small amount of content leak and do not require immediate key update after every user leave, the unique user join/leave pattern in live streaming is explored to achieve efficient key management. Chapter 5 provides a thorough performance analysis of the communication overhead, encryption overhead of a central server, called the key distribution center (KDC), and user's computation overhead to update keys, and shows the simulation results. Finally, Chapter 6 draws conclusions and discusses future work.

Chapter 2

Related Works

This chapter reviews recent advances in group access control and key management. Access control and key management guarantee that only authorized users can access the content. During a membership change (e.g. a user join/leave), all secret keys known to the join/leave users have to be updated and informed to all authorized users.

Depending on the key generation and distribution process, key management schemes can be roughly categorized into centralized and contributory approaches. In the centralized approaches, KDC generates and distributes key information to all group members. The contributory key management schemes are designed for scenarios where there is no central server and every participant contributes equally to the generation of the session keys, for example, in distributed wireless sensor networks and mobile ad hoc networks.

2.1 Centralized Key Management Schemes

In centralized key management schemes, there is a central entity, KDC, who generates and distributes keys to a group of users. This section reviews centralized key management schemes for the scenario where all group members subscribe to the same service.

Figure 2.1 shows an example of the early works in centralized key management schemes [20]. In this scheme, $\mathbf{K}_s$ is the session key which is used to encrypt the transmitted data,

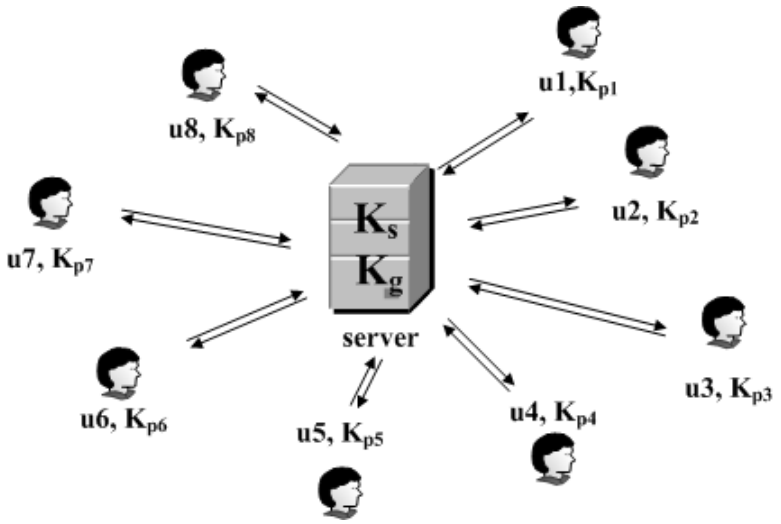

Fig. 2.1. A example of key scheme with session key, group key and numbers of private keys.

and $\mathbf{K}_g$ is the group key used to update the session key. Each user stores his/her own private key $\mathbf{K}_p$, the group key $\mathbf{K}_g$ and the session key $\mathbf{K}_s$. When membership updates, the session key and the group key have to be updated. For example, when user 8 joins, to prevent him/her from accessing the previous communication, $\mathbf{K}_g$ and $\mathbf{K}_s$ need to be updated. To securely transmit the new keys to the existing 7 users, the new keys $\mathbf{K}_{g,new}$ and $\mathbf{K}_{s,new}$ are encrypted using $\mathbf{K}_{g,old}$. These two new keys are also encrypted using $\mathbf{K}_{p8}$ and unicasted to the new member. Since user 8 does not know $\mathbf{K}_{g,old}$ and $\mathbf{K}_{s,old}$, he/she cannot decrypt the multicasted data before his/her join. Consequently, backward security is guaranteed. Compared to a user join, a user leave requires more rekey messages. Since the leaving user knows the old group key $\mathbf{K}_{g,old}$, it cannot be used to update the new group key $\mathbf{K}_{g,new}$ and the new session key $\mathbf{K}_{s,new}$. Thus, KDC has to encrypt these two new keys using each user's private key and multicasts the rekey messages to all users, which requires $O(k)$ rekey messages with k being the number of remaining users in the group.

To achieve efficient key update, the logical key hierarchical tree (LKH) was proposed in [21] and [22] independently. It is now widely used in key management [23], [24]. LKH introduces key encryption keys, and organizes keys as nodes in a balanced binary tree. For example, for the network in Figure 2.1, the hierarchical key tree is shown in Figure 2.2(a). The key above the tree root is the session key $\mathbf{K}_s$. The tree root corresponds to the group key $\mathbf{K}_g$, and the intermediate tree nodes correspond to the key encryption keys $\mathbf{K}_e$, each of which is shared by users who are descendants of the corresponding inner node. A leaf node corresponds to a user's private key $\mathbf{K}_p$. Each user in the group has all keys on the path from the corresponding leaf node to the root node plus the session key. For example, in Figure 2.2(a), user 1 has 5 keys, $\mathbf{K}_s$, $\mathbf{K}_g$, $\mathbf{k}_{e0}$, $\mathbf{k}_{e2}$ and his/her private key $\mathbf{K}_{p1}$.

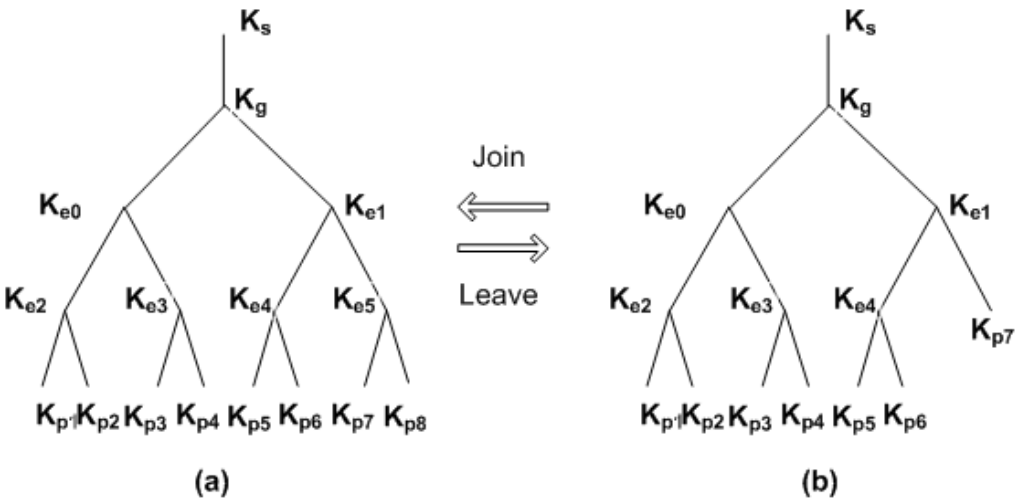

Fig. 2.2. The corresponding hierarchical key tree for Figure 2.1.

When there is a membership update, KDC first updates the key tree, then updates the keys that the leaving/join user knows one by one. Use Figure 2.2(a) as an example. After user 8 leaves, the node denoted by $\mathbf{K}_{e5}$ only has one child, user 7. KDC merges these two

nodes together, replaces the node labeled $\mathbf{K}_{e5}$ with the node representing $\mathbf{K}_{p7}$, and the new tree is shown in Figure 2.2(b). To ensure forward security, all keys known to the leaving user, that is, $\mathbf{K}_s$, $\mathbf{K}_g$ and $\mathbf{K}_{e1}$, should be updated, and the encryption-based method is used to update these keys. To update $\mathbf{K}_{e1}$, KDC generates a new key $\mathbf{K}_{e1,new}$, and multicasts two messages $\{\mathbf{K}_{e1,new}\}_{\mathbf{K}_{p7}}$ and $\{\mathbf{K}_{e1,new}\}_{\mathbf{K}_{e4}}$, where the first message enables user 7 to have the new key $\mathbf{K}_{e1,new}$ and $\{\mathbf{K}_{e1,new}\}_{\mathbf{K}_{e4}}$ enables user 5 and 6 to get $\mathbf{K}_{e1,new}$. Here, $\{\mathbf{K}_1\}_{\mathbf{K}_2}$ is used to denote the key update message $\{\{\mathbf{K}_1, \text{Ind}(\mathbf{K}_1)\}_{\mathbf{K}_2}, \text{Ind}(\mathbf{K}_2)\}$, which means that $\mathbf{K}_2$ is used to encrypt the rekey message to update $\mathbf{K}_1$, and $\text{Ind}(\mathbf{K}_1)$ is $\mathbf{K}_1$'s index. To update the group key $\mathbf{K}_g$, two messages $\{\mathbf{K}_{g,new}\}_{\mathbf{K}_{e0}}$ and $\{\mathbf{K}_{g,new}\}_{\mathbf{K}_{e1,new}}$ are sent to enable user 1 to 7 to have $\mathbf{K}_{g,new}$. Finally, KDC sends one rekey message $\{\mathbf{K}_{s,new}\}_{\mathbf{K}_{g,new}}$ to all users to update the session key. The key update algorithm for a user join is similar and thus omitted here. The LKH approach uses a logical balanced key tree to update keys, and reduces the number of rekey messages to $O(2\log_2(k))$.

The work in [25] proposed an improved LKH key management scheme, called the LKH+ scheme. It uses a secure one-way function $f(\cdot)$ [26] to update keys when a user joins the service. Here, one-way function is a mathematical function that is easy to compute in one direction (the forward direction), while it is hard to compute in the opposite direction. For example, when user 8 joins the service, KDC splits the node denoted by $\mathbf{K}_{p7}$ in Figure 2.2(b), puts user 7 and 8 under it, and generates a new key $\mathbf{K}_{e5}$ for that node, as shown in Figure 2.2(a). To ensure backward security, $\mathbf{K}_{e1}$, $\mathbf{K}_g$ and $\mathbf{K}_s$ need to be updated using the secure one-way functions. For example, KDC generates the new key $\mathbf{K}_{e1,new} = f(\mathbf{K}_{e1})$ and sends the index of $\mathbf{K}_{e1}$, $\text{Ind}(\mathbf{K}_{e1})$, to all users. Since only user 5 to 7 know $\mathbf{K}_{e1}$, they update $\mathbf{K}_{e1}$ using the same one-way function. $\mathbf{K}_g$ and $\mathbf{K}_s$ are updated using the same method. To send the new key $\mathbf{K}_{e5}$ to user 7, KDC encrypts it using $\mathbf{K}_{p7}$. Finally, KDC unicasts $\mathbf{K}_{e5}$, $\mathbf{K}_{e1}$, $\mathbf{K}_g$ and $\mathbf{K}_s$ to user 8. The LKH+ scheme still uses the encryption-based method to update keys for a user leave, and the rekey computation cost is the same as that

of the LKH. Note that the computation cost of one-way functions is much smaller than that of encryption/decryption and can be ignored. Therefore, the LKH+ scheme significantly reduces the rekey computation cost for user join when compared with the LKH scheme.

The one-way function tree (OFT) was proposed in [9] to further reduce the rekey cost. It uses the same balanced binary tree as the LKH scheme to organize keys. Different from the LKH scheme, the keys are not randomly generated by KDC, but using the rule $\mathbf{K}_x = g(f(\mathbf{K}_{x,left}), f(\mathbf{K}_{x,right}))$, where $\mathbf{K}_{x,left}$ and $\mathbf{K}_{x,right}$ are the left and the right children of node $\mathbf{K}_x$. Here, $g(\cdot)$ and $f(\cdot)$ are a mix function (e.g. XOR) and a one-way function, respectively, and $f(\mathbf{K}_x)$ is called the blinded value of $\mathbf{K}_x$ because users cannot derive $\mathbf{K}_x$ from $f(\mathbf{K}_x)$. For example, in Figure 2.2 (a), $\mathbf{K}_{e0} = g(f(\mathbf{K}_{e2}), f(\mathbf{K}_{e3}))$. The session key $\mathbf{K}_s$ is a crypto function of $\mathbf{K}_g$, for example, $\mathbf{K}_s = f(\mathbf{K}_g)$, where $f(\cdot)$ is a one-way function. In this scheme, each user can derive all the keys that he/she needs if he/she knows the blinded values of the keys that are sibling to the keys he/she knows. For example, in Figure 2.2(a), given his/her private key $\mathbf{K}_{p8}$ and blinded value $f(\mathbf{K}_{p7})$, user 8 can calculate $\mathbf{K}_{e5} = g(f(\mathbf{K}_{p7}), f(\mathbf{K}_{p8}))$. Similarly, given $f(\mathbf{K}_{e4})$ and $f(\mathbf{K}_{e0})$, user 8 can calculate $\mathbf{K}_{e1} = g(f(\mathbf{K}_{e4}), f(\mathbf{K}_{e5}))$, $\mathbf{K}_g = g(f(\mathbf{K}_{e0}), f(\mathbf{K}_{e1}))$ and $\mathbf{K}_s = f(\mathbf{K}_g)$ that he/she needs to know.

In the OFT scheme, when there is a membership update, KDC first reorganizes the key tree in the same way as the LKH scheme. Then it updates keys that the join/leaving user knows. For example, when user 8 leaves the key tree in Figure 2.2(a), KDC puts user 7 under the node labeled $\mathbf{K}_{e1}$ in Figure 2.2(b), and generate a new private key $\mathbf{K}_{p7,new}$ for user 7. In this case, $\mathbf{K}_{e1}$, $\mathbf{K}_g$ and $\mathbf{K}_s$ need to be updated. To update $\mathbf{K}_{e1}$ to user 5 to 7, KDC multicasts $\{f(\mathbf{K}_{p7,new})\}_{\mathbf{K}_{e4}}$ which enable user 5 and 6 to derive the new key $\mathbf{K}_{e1} = g(f(\mathbf{K}_{p7,new}), f(\mathbf{K}_{e4}))$. Note that user 7 has $f(\mathbf{K}_{e4})$ and can calculates $\mathbf{K}_{e1} = g(f(\mathbf{K}_{p7,new}), f(\mathbf{K}_{e4}))$ himself/herself. KDC uses the same method to update $\mathbf{K}_g$ and $\mathbf{K}_s$ to user 1 to 7. The key update for a user join follows the same procedure as a user leave, and KDC unicasts to the new user all the blinded values that he/she needs. From this example,

in the OFT scheme, only the blinded values of the updated keys need to be encrypted and multicasted, and only one rekey message is required to update one key. Therefore, compared with the LKH scheme that uses 2 rekey messages to update a key, the number of rekey messages in the OFT scheme is reduce to $O(\log_2(k))$.

The one-way function chain tree (OFCT) was independently proposed in [27]. The key tree design and the key update algorithm are the same as that in the OFT scheme except that the OFCT uses a pseudo-random generator rather than a mix function to generate new key encryption keys (KEK). Therefore, it achieves the same rekey overhead as the OFT scheme. The threshold-based one-way function tree (TOFT) was proposed in [28]. It uses the quad-tree structure (a tree with degree 4) rather than a binary key tree and further reduces the rekey computation cost by half when compared with the OFT scheme.

The efficiency of a fixed-degree key tree approach depends critically on whether the key tree remains balanced as members join and leave. The AVL-based key tree scheme was proposed in [29] to help keep the fixed degree trees balanced over time. In the self-balancing binary search AVL tree, the heights of the two child subtrees of any node differ by at most one [30]. In the AVL-based scheme, the departure time of any member is known at the time he/she joins the group, and users are placed in the tree in the descending order of their departure time. When the key tree is updated, AVL tree rotates itself to maintain the balance requirement, and this tree rotation does not incur any communication overhead [29].

The ALX key tree, a non-fixed-degree balanced tree, was introduced in [31]. In an ALX tree with a total of $l+1$ levels, nodes in the upper l levels (from level 0 to level $l-1$) have a fixed degree a, while there is no constraint on the degree of nodes at level l. Each leaf in the tree corresponds to a user's private key $\mathbf{K}_p$, the root node at level 0 corresponds to the group key $\mathbf{K}_g$, and the nodes at level 1 to level l correspond to the key encryption keys $\mathbf{K}_e$. Figure 2.3 is an example of ALX key tree with $a = 2$ and $l = 3$. The ALX tree scheme

adopts the rekey algorithm proposed in the LKH+ scheme. It was proved in [31] that the optimal ALX key tree, which minimizes the number of rekey messages for a user leave, achieves the lower bound of the rekey communication overhead of fixed-degree trees.

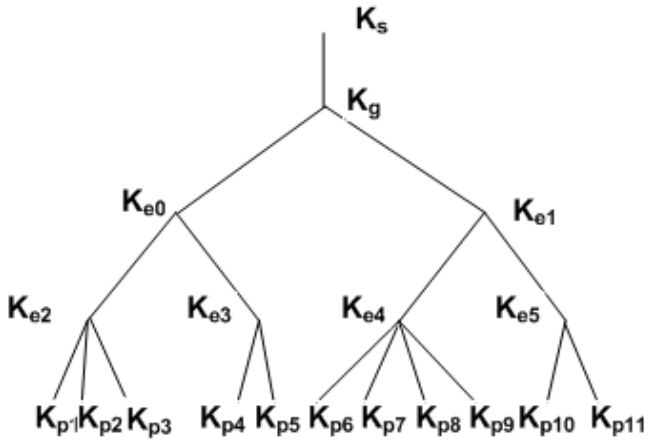

Fig. 2.3. An example of ALX key tree. $a = 2$, $l = 3$

2.2 Contributory Key Management Schemes

In the contributory key management schemes, there is no KDC and each participant contributes equally to the generation of the keys. These schemes are usually used when a central key server cannot be established. In the literature, many contributory key managements have been proposed [32], [33], [34], [35]. In this section, similar to the previous section, contributory key management schemes is reviewed for the scenario where all members subscribe to the same service.

Most contributory key management schemes use Diffie-Hellman key exchange (DH) [36], which is a cryptographic protocol that allows two parties to establish a shared secret key via insecure communications. To establish a key for secure communication between user 1 and user 2, they first agree on a pair of primes p and q. Then user 1 sends $a =$

$(p^{\mathbf{K}_1} mod\ q)$ to user 2, and user 2 replies user 1 with $b = (p^{\mathbf{K}_2} mod\ q)$. Here, $\mathbf{K}_1$ and $\mathbf{K}_2$ are the private keys of user 1 and user 2, respectively. Consequently, both of them can calculate their secret key $\mathbf{K} = (p^{\mathbf{K}_1 \cdot \mathbf{K}_2} mod\ q)$ using $\mathbf{K} = (b^{\mathbf{K}_1} mod\ q)$ and $\mathbf{K} = (a^{\mathbf{K}_2} mod\ q)$.

Group DH key exchange (GDH) [33] is an extension of the two-user DH protocol to a group of users. Consider an example with 4 members. Given the primes p and q, user 1 generates a secret key $\mathbf{K}_1$ and multicasts $(p^{\mathbf{K}_1} mod\ q)$. Then given his/her private key $\mathbf{K}_2$, user 2 multicasts two numbers, $(p^{\mathbf{K}_1 \cdot \mathbf{K}_2} mod\ q)$ and $(p^{\mathbf{K}_2} mod\ q)$. User 3 calculates and multicasts $(p^{\mathbf{K}_1 \cdot \mathbf{K}_3} mod\ q)$, $(p^{\mathbf{K}_2 \cdot \mathbf{K}_3} mod\ q)$ and $d \triangleq (p^{\mathbf{K}_1 \cdot \mathbf{K}_2 \cdot \mathbf{K}_3} mod\ q)$. Finally, user 4 multicasts $a \triangleq (p^{\mathbf{K}_2 \cdot \mathbf{K}_3 \cdot \mathbf{K}_4} mod\ q)$, $b \triangleq (p^{\mathbf{K}_1 \cdot \mathbf{K}_3 \cdot \mathbf{K}_4} mod\ q)$ and $c \triangleq (p^{\mathbf{K}_1 \cdot \mathbf{K}_2 \cdot \mathbf{K}_4} mod\ q)$. Then user 1, 2, 3 and 4 can calculate the group key $\mathbf{K}_g = (a^{\mathbf{K}_1} mod\ q) = (b^{\mathbf{K}_2} mod\ q) = (c^{\mathbf{K}_3} mod\ q) = (d^{\mathbf{K}_4} mod\ q)$, respectively. During membership update, the authorized members update their private keys and generate the new group key in a similar way. In the GDH scheme, because each member has to multicast the results of modular exponential operations to others, users need $O(k)$ rounds to establish the group key. Here, a round is defined as the duration in which each member can send and receive at most one message [37]. Therefore, this scheme is time consuming to establish a group key when the user number is large.

The DH logical key hierarchy (DH-LKH) scheme introduced in [38] uses the DH protocol to generate KEKs in the binary hierarchical tree, and the logical key tree has the same structure as the LKH scheme. Each KEK in the key tree is generated from the keys of its two children using the DH algorithm, for example, $\mathbf{K}_{e5} = (p^{\mathbf{K}_{p7} \cdot \mathbf{K}_{p8}} mod\ q)$ in Figure 2.2(a). In this scheme, since $\mathbf{K}_s$ only has one child and cannot be calculated using the DH protocol, $\mathbf{K}_s$ is defined as a crypto function of $\mathbf{K}_g$. When membership changes, keys known by the arriving/leaving user need to be updated using the DH protocol. For example, when user 8 leaves in Figure 2.2(a), the updated key tree is shown in Figure 2.2(b). User 7 generates a new key $\mathbf{K}_{p7,new}$ himeself/herself, and multicasts $a = (p^{\mathbf{K}_{p7,new}} mod\ q)$, which enables user 5 and 6 to calculate $\mathbf{K}_{e1,new} = (a^{\mathbf{K}_{e4}} mod\ q)$. User 7 knows $(p^{\mathbf{K}_{e4}} mod\ q)$ and can calculate

$\mathbf{K}_{e1} = ((p^{\mathbf{K}_{e4}} mod\ q)^{\mathbf{K}_{p7,new}} mod\ q)$ himself/herself. Users use the same method to update $\mathbf{K}_g$. The DH-LKH scheme takes no more than $O(\log_2 k)$ rounds to update keys for a user join/leave, where k is the total number of users in a group. The logical hierarchical key tree is now adopted in many contributory key management schemes [39], [40].

A dynamic Join-Exit-Tree (JET) was proposed in [1] to efficiently manage the join and leaving users and to reduce the processing time to update keys. As shown in Figure 2.4, a JET tree consists of three parts: the main tree, the join tree and the exit tree, all of which are binary trees built using the DH protocol. When a user joins the group, he/she is put in

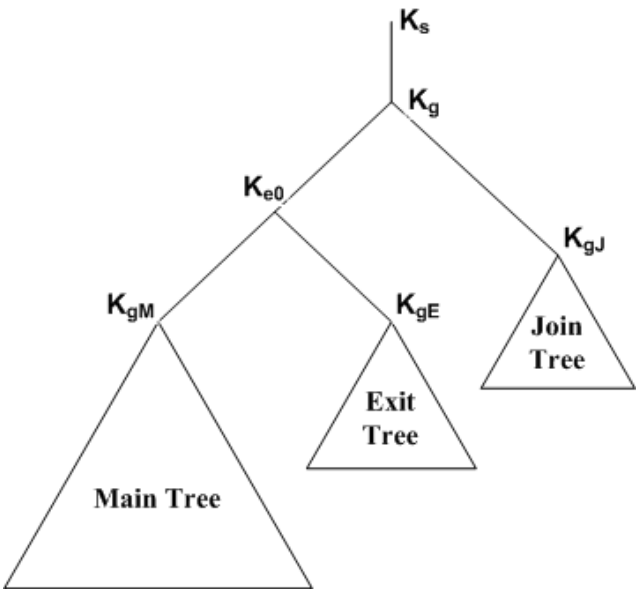

Fig. 2.4. Topology of a JET tree [1].

the join tree first. If the join tree is balanced, the root of join tree is chosen as the insertion node. Otherwise, the node that is closest to the tree root is split and the new user is placed under that node. Then all keys that the new user knows need to be update using the DH protocol, same as the DH-LKH scheme. When the join tree reaches its maximal capacity (which is defined as the maximum number of users in the join tree) or when there is a user leaving from the main tree, all users in the join tree are relocated to the main tree. For a

user leave, the JET scheme assumes that users' exact leaving times are known. It updates the main key tree at predetermined time instances, called batch moments. At each batch moment, users who are going to leave soon are moved from the main tree to the exit tree, and all keys known to the relocated users are updated. In the JET key scheme, since users join/leave the service from the smaller join/exit trees with smaller depths, the number of keys need to be updated for each membership update is small. It was shown in [1] that with parallel key update, the key update time is reduced from $O(\log k)$ to $O(\log(\log k))$, where k is the user number.

A join-tree-based contributory group key management (JDH) was proposed in [41], where new users are always inserted to the root of the join tree and the rekey overhead for a join user is reduced to $O(1)$. However, this rekey algorithm results in an extremely unbalanced binary join tree.

A weighted join-exit tree was proposed in [42] to reduce the rekey overhead when a user leaves. The exit tree is organized so that the sooner the user leaves, the closer he/she is to the root of the exit tree. Thus, the leaving user is always a child of the exit tree root, and the key update time for a user leave is reduced to $O(1)$.

Wireless sensor networks have many applications, for example, in habitat monitoring, and vehicular tracking. Due to the constraints on power, storage apace, and computation capability, traditional key management schemes using exponential operators $\exp(\cdot)$ to generate session keys (such as the DH-based approaches) are not suitable for key management in wireless sensor networks [43], [44] due to the large computation cost. Elliptic curve cryptography (ECC) [45], [46], another approach to generate keys, can achieve the same level of security as the RSA-based method with less storage and power requirements [47], [48].

To reduce the storage overhead, a routing-driven ECC-based key management scheme (RECC) was proposed in [49] for heterogeneous sensor networks. In the RECC scheme, sensors are grouped in clusters. The cluster head corresponds to the information sink, and

each sensor only communicates with neighboring sensors that are on the path from itself to the cluster head rather than all neighboring sensors. Therefore, different from other sensor key management schemes where a sensor needs to store all the keys of its neighboring sensors, a sensor in the RECC scheme needs to store only the keys of those neighboring sensors with whom it communicates, which largely reduces the storage overhead in implementing the security scheme.

2.3 Multilevel Access Control Schemes

There are many group-oriented applications where group members subscribe to different levels of service. For such applications, multilevel access control and key management are often desired to support the business model "what you see is what you pay" [15]. In multilevel access control scheme in live streaming, the key tree consists of the Data Group (DG) subtree and the Service Group (SG) subtrees [50], [51]. The DG subtree is used to manage the keys used to encrypt different layers in scalable video coding, and the SG subtrees are used to manage a group of users who subscribe to the same level of service. The works in this area mainly focus on the DG subtree design and session key update, and many key trees reviewed in the Section 2.1, for example, the LHK scheme, can be used to build the SG subtrees.

Figure 2.5 shows an example where ten users subscribe to a bit stream including three layers with one type of scalability, e.g., temporal scalability. In this example, users are grouped into 3 subgroups depending on their subscriptions. User 1 to 3 subscribe to the low-resolution copy and can access only the base layer; User 4 to 6 subscribe to the medium-resolution copy and receive the base layer and enhancement 1; and user 7 to 10 subscribe to the high-resolution copy and can access all three layers. Given this example, three multilevel access schemes are introduced in the rest of the section.

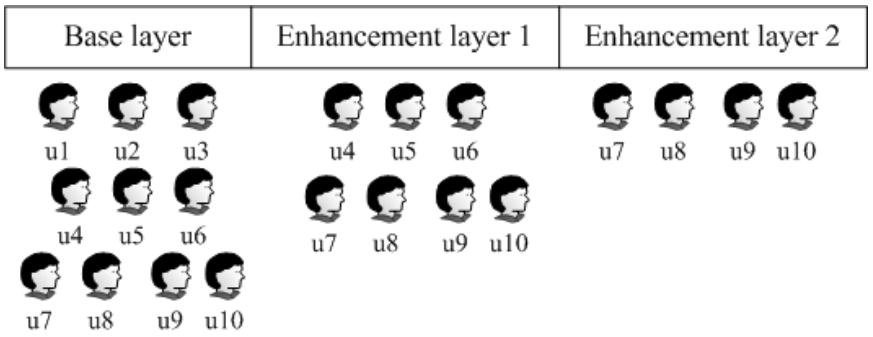

Fig. 2.5. User's subscriptions to a compressed bit streaming.

The independent scheme (InS) proposed in [50] manages different layers independently. It generates independent session keys for different layers in scalable coding and distributes them to the users who have access to them. Given the example in Figure 2.5, the corresponding InS key tree is shown in Figure 2.6, where the DG subtree is made up of three session keys, $\mathbf{K}_s^L$, $\mathbf{K}_s^M$ and $\mathbf{K}_s^H$ that are used to encrypt the base layer, enhancement layer 1 and enhancement layer 2, respectively. $\mathbf{K}_g^L$, $\mathbf{K}_g^M$ and $\mathbf{K}_g^H$ are the subgroup keys of the three service groups. Since user 7 to 10 have access to the enhancement layer 2, they can also access the base layer and enhancement layer 1. Therefore, user 7 to 10 know all three keys. Similarly, user 4 to 6 who have access to enhancement layer 1 should also be able to access the base layer and have $\mathbf{K}_s^M$ and $\mathbf{K}_s^L$. Here, the rekey algorithm proposed in the

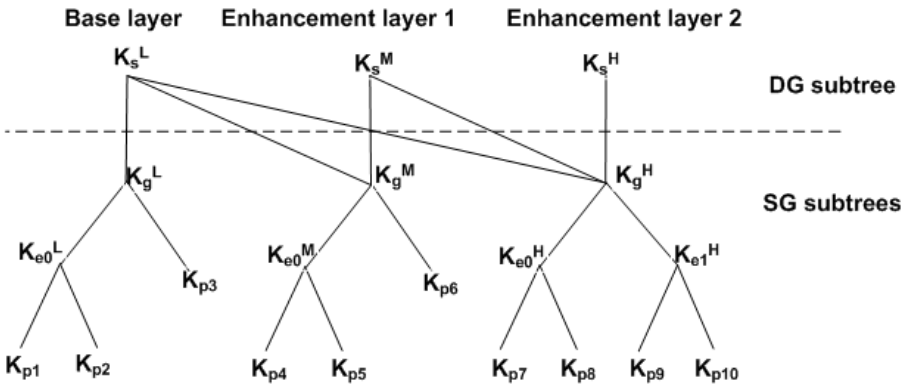

Fig. 2.6. Independent key management scheme for the example shown in Figure 2.5.

LKH+ scheme is used as an example to illustrate the rekey process in the InS scheme. For example, when user 6 joins the subgroup whose members can access only the base layer and the enhancement layer 1, $\mathbf{K}_g^M$, $\mathbf{K}_s^M$ and $\mathbf{K}_s^L$ are updated using one-way functions. When user 6 leaves, these three keys need to be updated using the encryption-based algorithm. To update $\mathbf{K}_g^M$, KDC sends message $\{\mathbf{K}_{g,new}^M\}_{\mathbf{K}_{e0}^M}$. Since only user 4 and 5 have $\mathbf{K}_{e0}^M$, only these two users can decrypt and get the new key $\mathbf{K}_{g,new}^M$. To update $\mathbf{K}_s^M$, KDC multicasts $\{\mathbf{K}_{s,new}^M\}_{\mathbf{K}_{g,new}^M}$ and $\{\mathbf{K}_{s,new}^M\}_{\mathbf{K}_g^H}$. Similarly, three messages $\{\mathbf{K}_{s,new}^L\}_{\mathbf{K}_g^L}$, $\{\mathbf{K}_{s,new}^L\}_{\mathbf{K}_{g,new}^M}$ and $\{\mathbf{K}_{s,new}^L\}_{\mathbf{K}_g^H}$ are multicasted to enable the remaining users to update $\mathbf{K}_s^L$. As seen in the above example, the InS scheme does not explore the data dependency among scalable layers in scalable coding, and several rekey messages are required to update a new session key.

To explore the data dependency between different data groups, a multi-group key management scheme (MGS) was proposed in [51], where the DG subtree employs one more layer of key encryption keys $\mathbf{K}_d$ in the DG subtree, and introduces an integrated scheme to connect the three session keys. Same as the InS scheme, all keys in the MGS are generated and updated independently. Figure 2.7 shows the MGS key tree for the example in Figure 2.5. Users in the scheme need to store more keys when compared to the InS scheme, while this extra layer of keys facilitates efficient session key update. For example, in the MGS scheme, to update the session key $\mathbf{K}_s^L$, different from the InS scheme that requires 3 rekey messages, KDC only needs to broadcast one rekey message $\{\mathbf{K}_{s,new}^L\}_{\mathbf{K}_d^L}$, and all users can decrypt it.

A POSET hash-based access control scheme was proposed in [52] to generate session keys, and the Hasse diagram was used in the DG subtree to address the data dependency between different layers and to reduce the number of session keys each user has to store. In this scheme, each vertex in the DG subtree, representing a layer in scalable coding, is assigned a label d and a session key $\mathbf{K}_s$, and each edge in a poset has a value v calculated by

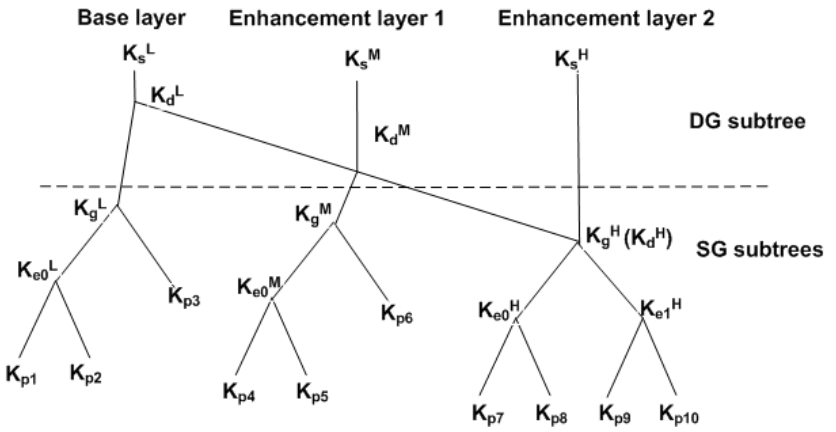

Fig. 2.7. Multi-group key management scheme for the example shown in Figure 2.5.

applying a secure hash-based function to connect session keys. The labels and edge values are public information. Figure 2.8 shows the POSET Hash-based key tree for the example in Figure 2.5. Note that in POSET [53], [54], a user needs to know only one session key

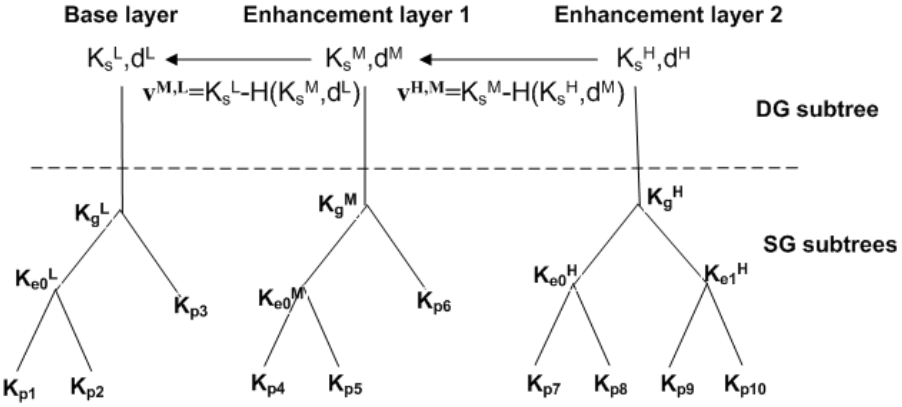

Fig. 2.8. The POSET hash-based access control scheme for the example shown in Figure 2.5.

in the DG subtree, and he/she can derive the rest of the session keys himself/herself using public information. For example, user 6 in the Figure 2.8 who has access to the base layer and the enhancement 1 needs to store only the session key $\mathbf{K}_s^M$ in Figure 2.8 as he/she can derive $\mathbf{K}_s^L$ using $\mathbf{K}_s^L = v^{M,L} + H(\mathbf{K}_s^M, d^L)$ himself/herself, where $H(\cdot)$ is a predefined hash function [26]. However, the work in [52] did not discuss efficient update of the keys during

membership update.

Chapter 3

System Model

This chapter introduces the system model, including the POSET structure that is used to explore the data dependency in scalable coding, and the characteristics of membership dynamics in live streaming.

3.1 Multilevel Access Control in Live Streaming

Scalable coding is often used in live streaming to address the network and device heterogeneity. To achieve multilevel access control in a scalable video coding system, each layer is encrypted using a different session key, and these keys are distributed appropriately to users so that a user has the keys to access only the content that he/she is authorized to. There is strong dependency between different session keys, and a user who subscribes to an enhancement layer should have the session keys for all lower layers.

Following previous works [50], [51], the Data Group (DG) is defined as a particular layer in scalable video coding, and the Service Group (SG) is defined as a group of users who subscribe to the same level of service. Below is an example that explains the concepts of DG, SG and the access privileges in a scenario where a compressed bit stream supports

multiple types of scalability.

Example 1: A compressed bit stream supports temporal and spatial scalability simultaneously, as shown in Figure 3.1. For temporal scalability, it encodes the odd frames in the base layer denoted by TL, and compresses the difference between the even and the odd frames in the enhancement layer TH. In the spatial domain, down-sampled frames are encoded in the base layer SL, and the difference between the original frames and the frames reconstructed from the base layer is encoded in the enhancement layer SH. Assume that 9 users pay for the highest resolutions in both dimensions, 10 users subscribe to the lowest resolution in both temporal and spatial domains, and for the two median resolutions (low resolution in temporal and high resolution in spatial, high resolution in temporal and low resolution in spatial), each has 8 subscriptions. In this example, the data groups, the service groups, and the access privileges of each service group are defined as follows.

- Data Groups (DGs): Temporal and spatial scalability are independent, and each scalability has one base layer and one enhancement layer. Therefore, the compressed bit stream contains four data groups, which are $DG_{(TH,SH)}$, $DG_{(TH,SL)}$, $DG_{(TL,SH)}$ and $DG_{(TL,SL)}$, each encrypted using a different session key. The arrows in Figure 3.1 shows the data dependency between different data groups.

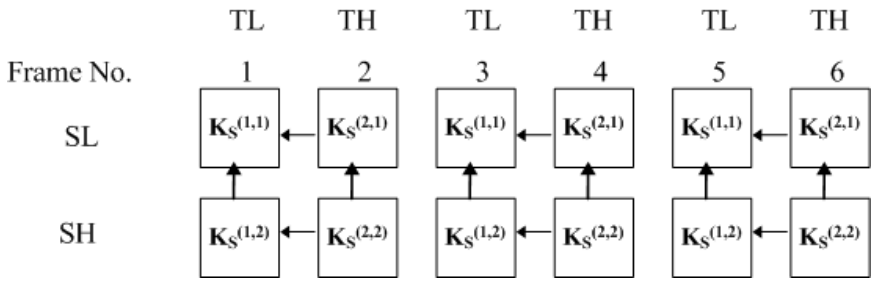

Fig. 3.1. Data of different layers in **Example** 1.

- Service Groups (SGs): Depending on their subscribed service, users can be divided

into four service groups: 9 users in $SG_{(TH,SH)}$; 8 users in $SG_{(TH,SL)}$; 8 users in $SG_{(TL,SH)}$; and 10 users in $SG_{(TL,SL)}$.

- For each service group, its access privilege defines the data groups that its users can access. For the above example, the access privilege of each service group is shown in Table 3.1. For example, users in $SG_{(TH,SH)}$ have all four keys, while users in $SG_{(TL,SL)}$ have only key $K_s^{(1,1)}$ in Figure 3.1.

TABLE 3.1

Access privileges of the SGs in **Example** 1.

SGs	Access Privilege
$SG_{(TH,SH)}$	$DG_{(TH,SH)}$, $DG_{(TH,SL)}$, $DG_{(TL,SH)}$, $DG_{(TL,SL)}$
$SG_{(TH,SL)}$	$DG_{(TH,SL)}$, $DG_{(TL,SL)}$
$SG_{(TL,SH)}$	$DG_{(TL,SH)}$, $DG_{(TL,SL)}$
$SG_{(TL,SL)}$	$DG_{(TL,SL)}$

In general, assume that a compressed bit stream supports N types of scalability, and the ith type has M_i layers. In this thesis, a simple scenario where $M_1 = M_2 = \cdots = M_N = M$ is considered, and it can be extended to the general scenario where $\{M_i\}$ are different. To simplify the notation, let $DG_{\mathbf{I}}$ with $\mathbf{I} = (i_1, \cdots, i_N)$ denote the data group that contains the i_jth ($1 \leq i_j \leq M$) layer for the jth type of scalability, where $i_j = 1$ corresponds to the base layer while $i_j = M$ is the highest enhancement layer. Correspondingly, $SG_{\mathbf{I}}$ denotes the service group that subscribes to a total of i_j ($1 \leq i_j \leq M$) layers for the jth type of scalability, including the base layer. That is, for the jth type of scalability, $i_j = 1$ means that users in this service group only receive the base layer, while $i_j = M$ indicates that users in this group receive all M layers. In the above Example 1 with $N = M = 2$, let temporal

and spatial scalability be the 1st and the 2nd types of scalability, respectively. Then, the notations for data groups $DG_{(TL,SL)}$, $DG_{(TL,SH)}$, $DG_{(TH,SL)}$ and $DG_{(TH,SH)}$ are simplified as $DG_{(1,1)}$, $DG_{(1,2)}$, $DG_{(2,1)}$ and $DG_{(2,2)}$, respectively. Correspondingly, the service groups $SG_{(TL,SL)}$, $SG_{(TL,SH)}$, $SG_{(TH,SL)}$ and $SG_{(TH,SH)}$ are simplified as $SG_{(1,1)}$, $SG_{(1,2)}$, $SG_{(2,1)}$ and $SG_{(2,2)}$, respectively. To achieve multilevel access control, each data group is encrypted by a session key $\mathbf{K}_s^{\mathbf{I}}$. In this paper, it is said that $\mathbf{K}_s^{\mathbf{I}}$ is associated with data group $DG_{\mathbf{I}}$ and service group $SG_{\mathbf{I}}$ when $\mathbf{K}_s^{\mathbf{I}}$ is used to encrypt data group $DG_{\mathbf{I}}$.

3.2 Key Tree Design

This thesis focuses on live broadcast applications, where the service provider broadcasts the video stream to a group of users and where a user has little control on the streaming session except the ability to join and leave [2]. In these applications, the content owner or the service provider can serve as KDC and manage the key update, and the centralized key management approach is used in this work. Following the work in [50], [51], the key tree design is divided into two parts: the SG subtree design and the DG subtree design.

3.2.1 Step 1 SG Subtree Design

It was shown in [31] that the ALX tree avoids the unbalanced tree structure that may often happen in the fixed-degree logic key trees due to frequent membership update, and that it helps to reduce the rekey overhead significantly. This makes the ALX key tree an ideal candidate for key management in live broadcast applications, where large variations in the total number of users is observed. Following the work in [31], to address the unbalanced key tree structures, the proposed key tree uses the ALX tree to manage users in each service group. Figure 3.2 shows an example of the four ALX SG subtrees built for the four service groups in the Example 1 in Section 3.1. Here, $a = 3$ and $l = 1$ are chosen for all four

subtrees. $\mathbf{K}_g$ is the group key shared by all users in the service group, and $\mathbf{K}_p$ denotes user's private key.

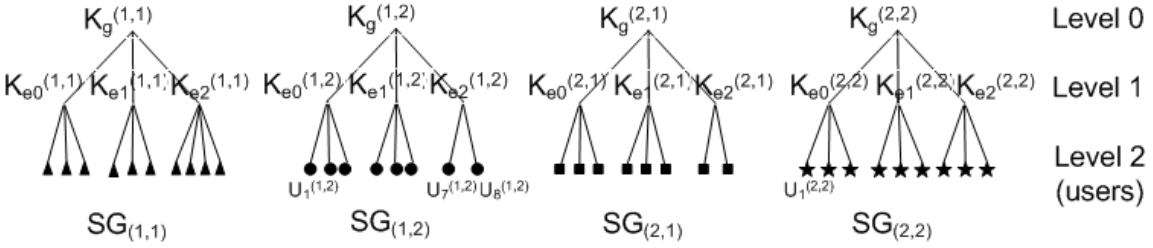

Fig. 3.2. SG Subtree design for **Example** 1. $a = 3$ and $l = 1$.

An important issue in the ALX logic key tree design is to select the optimal parameters a and l that minimize the number of rekey messages when a user joins/leaves the service. From the analysis in [31], the rekey overhead for a user join is much smaller than that for a user leave and can often be ignored. Therefore, in this thesis, the SG ALX trees is designed to minimize the rekey overhead for leaving users. Given a total of k users in a service group, assume that they are uniformly distributed in an ALX tree of degree a and level $l+1$, that is, all the nodes at level l have approximately the same number of children. The analysis in [31] showed that the average number of rekey messages for one user departure can be approximated by $C(k) = k/a^l + al - 1$. For a given k, all possible pairs of (a,l) are enumerated to find the optimal one that minimizes $C(k)$. The optimal pairs of (a^*,l^*) for different k are shown in Table 3.2. From Table 3.2, it is observed that $a^* = 3$ is the optimal tree degree when $15 \leq k \leq 25000$, and the optimal l^* remains the same for a wide range of k. For example, $l^* = 5$ for all $k \in [365, 1093]$. Thus, the optimal ALX tree structure remains unchanged even when the total number of users changes significantly. This is a desired feature for live broadcast applications with frequent and drastic membership change.

TABLE 3.2

The optimal values of (a, l) for different user numbers (k).

k	15-40	41-121	122-364	365-1093	1094-3280	3281-9842	9843-25000
(a^*, l^*)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)	(3,8)

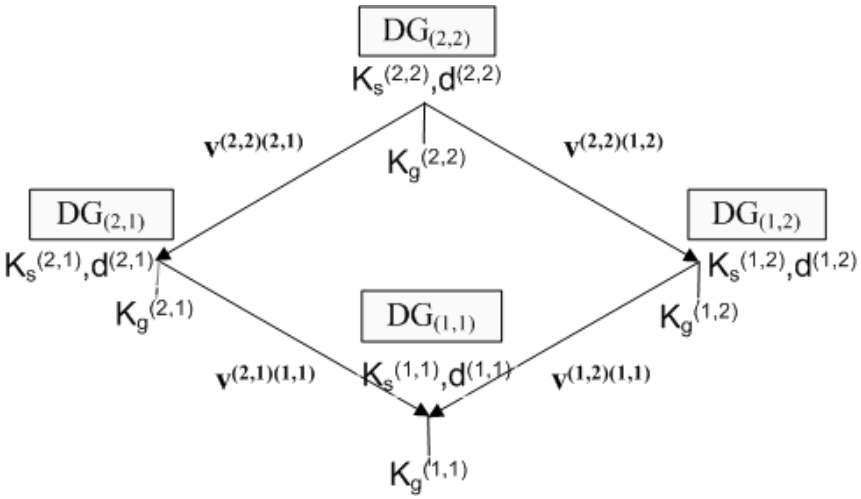

Fig. 3.3. DG Subtree design for **Example** 1.

3.2.2 Step 2 DG Subtree Design

A POSET Hash-based scheme was proposed in [52], [54] to address the data dependency between different layers and to facilitate efficient management of session keys. Given the data groups, the POSET Hash-based scheme first builds a corresponding Hasse diagram [53], [55]. Then, each vertex is assigned a secret session key and a public label. In addition, an edge value is assigned to each arrow connecting two vertices. Figure 3.3 is a POSET Hash-based DG subtree for Example 1 in Section 3.1. In Figure 3.3, the arrow connecting the two vertices $DG_{(1,2)}$ and $DG_{(1,1)}$ has a value $v^{(1,2),(1,1)}$ that satisfies

$v^{(1,2),(1,1)} = \mathbf{K}_s^{(1,1)} - H\left(\mathbf{K}_s^{(1,2)}, d^{(1,1)}\right)$. In this scheme, the session keys are known only to users who have authorization to the corresponding data groups, while the vertex labels $\{d^{\mathbf{I}}\}$ and the edge values $\{v^{\mathbf{I},\mathbf{J}}\}$ are public information. Note that in the POSET Hash-based DG subtree, given a higher layer's session key, all the lower layers' session keys can be derived using the public information $\{d^{\mathbf{I}}, v^{\mathbf{I},\mathbf{J}}\}$. For example, in Figure 3.3, with knowledge of $K_s^{(2,2)}$ for data group $DG_{(2,2)}$, a user can derive all the other session keys using $\mathbf{K}_s^{(2,1)} = v^{(2,2),(2,1)} + H\left(\mathbf{K}_s^{(2,2)}, d^{(2,1)}\right)$, $\mathbf{K}_s^{(1,2)} = v^{(2,2),(1,2)} + H\left(\mathbf{K}_s^{(2,2)}, d^{(1,2)}\right)$ and $\mathbf{K}_s^{(1,1)} = v^{(2,1),(1,1)} + H\left(\mathbf{K}_s^{(2,1)}, d^{(1,1)}\right)$ (or equivalently, $\mathbf{K}_s^{(1,1)} = v^{(1,2),(1,1)} + H\left(\mathbf{K}_s^{(1,2)}, d^{(1,1)}\right)$).

In the POSET scheme, vertex $\mathbf{I} = (i_1, \cdots, i_N)$ (or key $\mathbf{K}_s^{\mathbf{I}}$) is an ancestor of vertex $\mathbf{J} = (j_1, \cdots, j_N)$ (or key $\mathbf{K}_s^{\mathbf{J}}$) and vertex $\mathbf{J}$ (or key $\mathbf{K}_s^{\mathbf{J}}$) is a descendant of vertex $\mathbf{I}$ (or key $\mathbf{K}_s^{\mathbf{I}}$) if $\mathbf{K}_s^{\mathbf{J}}$ can be derived from $\mathbf{K}_s^{\mathbf{I}}$, that is, $1 \leq j_k \leq i_k$ for all $k = 1, \cdots, N$. Define $Q_{\mathbf{I}} \stackrel{\triangle}{=} \left\{\mathbf{K}_s^{\mathbf{J}=(j_1,\cdots,j_N)} | \mathbf{J} \in U_{\mathbf{I}}\right\}$ as the set including $\mathbf{K}_s^{\mathbf{I}}$ and all its descendants, where $U_{\mathbf{I}} \stackrel{\triangle}{=} \{\mathbf{J} : 1 \leq j_k \leq i_k, \forall k = 1, \cdots, N\}$. For service group $SG_{\mathbf{I}}$, $Q_{\mathbf{I}}$ includes all the session keys that it should know. For example, in Figure 3.3, for service group $SG_{(1,2)}$, $U_{(1,2)} = \{(1,2),(1,1)\}$ and $Q_{(1,2)} = \left\{\mathbf{K}_s^{(1,2)}, \mathbf{K}_s^{(1,1)}\right\}$. In addition, if an ancestor $\mathbf{K}_s^{\mathbf{I}}$ and a descendant $\mathbf{K}_s^{\mathbf{J}}$ are connected by an arrow in the Hasse diagram, that is, $\sum_{k=1}^{N} |i_k - j_k| = 1$, then $\mathbf{K}_s^{\mathbf{I}}$ is a parent of $\mathbf{K}_s^{\mathbf{J}}$. In Figure 3.3, $\mathbf{K}_s^{(2,2)}$ is an ancestor of $\mathbf{K}_s^{(2,1)}$, $\mathbf{K}_s^{(1,2)}$, and $\mathbf{K}_s^{(1,1)}$, but $\mathbf{K}_s^{(2,2)}$ is a parent of only $\mathbf{K}_s^{(2,1)}$ and $\mathbf{K}_s^{(1,2)}$.

3.2.3 Step 3 Integration of the SG Subtrees and the DG Subtree

The last step is to connect the roots of the SG subtrees and the corresponding leaves of the DG subtree and to construct the integrated key management scheme. Figure 3.4 illustrates the corresponding integrated POSET Hash-based key tree for Example 1 in Section 3.1. Note that in Figure 3.4, although a SG subtree $SG_{\mathbf{I}}$ is directly connected to only one vertex $\mathbf{I}$ in the DG subtree, users in $SG_{\mathbf{I}}$ not only have access to the session key $\mathbf{K}_s^{\mathbf{I}}$ associated with

vertex **I**, they can also use the public information to derive all its descendants $\mathbf{K}_s^{\mathbf{J}} \in Q_{\mathbf{I}}$.

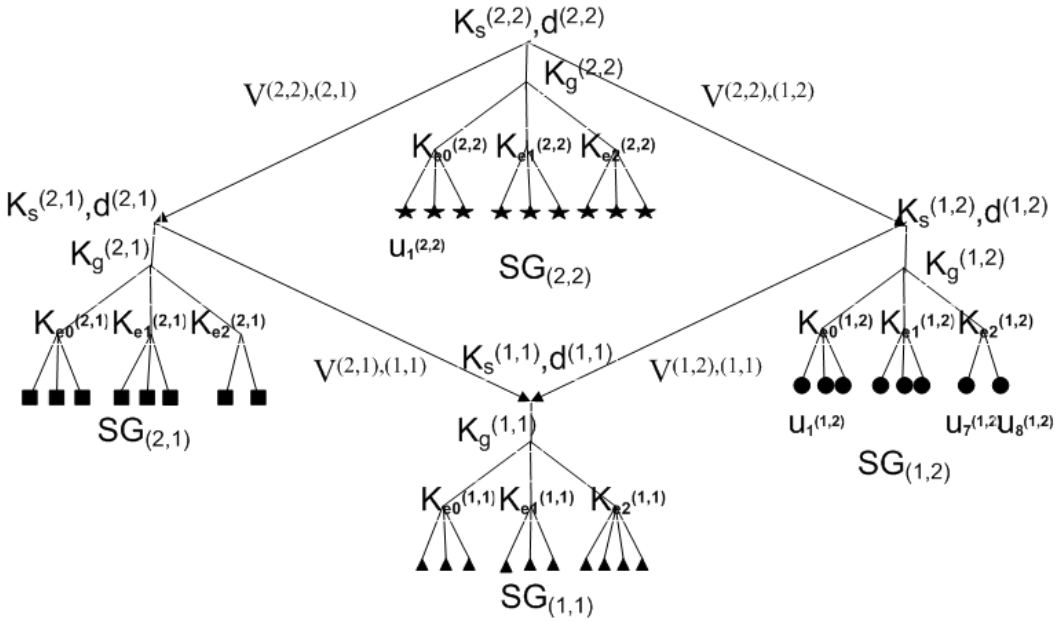

Fig. 3.4. POSET Hash-based key management scheme for **Example** 1.

3.3 User Dynamics in Live Broadcast Streaming

Prior works showed that live streaming applications share several common characteristics in their membership dynamics.

- *Large variation in the total number of users* is observed in many live streaming applications. For example, a study in [16] showed that a popular live streaming program had a maximum of 74000 users, while the minimum number of users was less than 10000.

- *Flash crowd* is the phenomenon that the peak arrival rate is much higher than the average arrival rate [18], [16], [17]. It is observed in all short streams (that have pre-determined broadcast time and duration, such as NBA games or talk shows) and

in 50% of the popular streams (whose largest number of concurrent users exceeds 1000). For example, a study in [16] showed that the peak arrival rate can be twice that of the usual arrival rate. Similarly, it is observed that the number of leaving users is large at the end of the programs.

- *High reconnection rate* is observed in those programs that are both short and popular. In these programs, users may leave the service due to excessively long waiting time or poor network conditions, and reconnect to the service within a short duration. For example, in a recent live baseball game that was broadcasted by Yahoo! Japan, at the beginning of the game, in each time unit, a maximum of 200 users rejoined the program within 2 minutes after their leaving [19].

- *A large number of short sessions* are observed, especially at the beginning of the live streaming programs. A work showed that for popular programs, over 55% of users stayed in the program for less than 5 minutes, and 30% of them stayed less than 1 minute in the program [17]. The work in [19] studied a live broadcast of a baseball game in Japan, and showed that the average number of short sessions (whose durations were shorter than 120 seconds) was around 170 per minute, and the peak number reached 400 short sessions per minute at the beginning of the program.

To summarize, the group membership may change drastically and frequently in live streaming applications, which poses challenges to efficient key management. This work addresses this frequent membership change in live streaming applications, and investigates efficient key management schemes, which offer the desired level of protection of multimedia content and minimize the rekey cost.

3.4 Performance Criteria

To measure the efficiency of key management schemes, the following criteria are used to measure the computation and communication rekey overhead:

- C_{en}: the average number of rekey messages that KDC needs to encrypt per second,
- C_{de}: the average number of rekey messages that a user needs to decrypt per second, and
- C_{msg}: the average number of rekey messages broadcasted by KDC per second, including both the messages with encrypted keys and those to update public information in POSET.

Chapter 4

Efficient Key Management in Multilevel Access Control

The previous chapter introduces the proposed key tree. The purpose of the key tree design is to support secure and efficient key update. The InS scheme and the MGS scheme use the encryption-based method to update both the SG and the DG subtrees and to securely transmit new keys to users. For the POSET Hash-based scheme, instead of using the traditional encryption-based method, this thesis proposes to use the public information in the POSET ($\{d^{\mathbf{I}}, v^{\mathbf{I},\mathbf{J}}\}$) and the hash function to update the session keys in the data group subtree. This is because hash functions introduce much lower computation cost than encryption and decryption, and thus can help significantly reduce the rekey overhead. In this work, two different applications are considered: those that require strict forward and backward security, and those that can tolerate a small amount of content leak.

4.1 Sequential Rekeying Scheme

For applications that cannot tolerate any content leak, sequential rekeying that is widely used in key management schemes [9], [21], [31], [23, 28], [51] is adopted. It updates keys immediately after each user join/leave and treats each reconnection as a user leave followed by an independent user join. It achieves strict forward and backward security, and provides an upper bound of the rekey overhead of key management schemes. With sequential rekeying, when a user joins/leaves the service, KDC immediately updates all the keys that he/she knows, including the keys in the service group subtree and those in the data group subtree.

4.1.1 Update of the Service Group Subtree

Following the work in [31], encryption-based method is used to update the keys in the service group subtree when there is a membership update. In the example in Figure 3.2, when user $u_8^{(1,2)}$ joins the service group $SG_{(1,2)}$, KDC puts him/her under the node labeled $\mathbf{K}_{e2}^{(1,2)}$ (besides $u_7^{(1,2)}$) in the ALX subtree, and two keys, $\mathbf{K}_g^{(1,2)}$ and $\mathbf{K}_{e2}^{(1,2)}$, need to be updated. When there is a joining user, an efficient way to update keys is to use a secure one-way function $f(\cdot)$ that is previously agreed upon by KDC and all users. For example, to update key $\mathbf{K}_g^{(1,2)}$, KDC and all users in the service group $SG_{(1,2)}$ calculate $\mathbf{K}_{g,new}^{(1,2)} = f\left(\mathbf{K}_{g,old}^{(1,2)}\right)$, where $\mathbf{K}_{g,old}^{(1,2)}$ and $\mathbf{K}_{g,new}^{(1,2)}$ are the old and the new versions of the group key $\mathbf{K}_g^{(1,2)}$, respectively. Similarly, KDC and user $u_7^{(1,2)}$ update $\mathbf{K}_{e2}^{(1,2)}$ through $\mathbf{K}_{e2,new}^{(1,2)} = f\left(\mathbf{K}_{e2,old}^{(1,2)}\right)$.

When user $u_8^{(1,2)}$ leaves the service, 2 keys in the SG subtree, $\mathbf{K}_g^{(1,2)}$ and $\mathbf{K}_{e2}^{(1,2)}$, need to be updated. To send the new key $\mathbf{K}_{e2}^{(1,2)}$ to user $u_7^{(1,2)}$, KDC multicasts the encrypted message $\{\mathbf{K}_{e2,new}^{(1,2)}\}_{\mathbf{K}_p^7}$. To update key $\mathbf{K}_g^{(1,2)}$, KDC sends three rekey messages $\{\mathbf{K}_{g,new}^{(1,2)}\}_{\mathbf{K}_{e0}^{(1,2)}}$, $\{\mathbf{K}_{g,new}^{(1,2)}\}_{\mathbf{K}_{e1}^{(1,2)}}$, and $\{\mathbf{K}_{g,new}^{(1,2)}\}_{\mathbf{K}_{e2,new}^{(1,2)}}$, which only enable the remaining users in the service group $SG_{(1,2)}$ to find the new key $\mathbf{K}_g^{(1,2)}$. Thus, when user $u_8^{(1,2)}$ leaves the service, a total

of 4 rekey messages are needed to securely update keys in the SG subtrees.

4.1.2 Update of $\{d^{\mathbf{I}}\}$

After updating the keys in the SG subtree, the next step is to update the keys in the data group subtree. Note that when a user joins/leaves, before updating the session keys in POSET, KDC and users first need to update the vertex labels $\{d^{\mathbf{I}}\}$. Otherwise, the joining/leaving user can easily derive old/new session keys himself/herself. For example, if $d^{(1,2)}$ is unchanged after user $u_8^{(1,2)}$ joins/leaves the service group $SG_{(1,2)}$, then the old and the new versions of $\mathbf{K}_s^{(1,2)}$ satisfy:

$$\mathbf{K}_{s,old}^{(1,2)} = v_{old}^{(2,2),(1,2)} + H\left(\mathbf{K}_s^{(2,2)}, d^{(1,2)}\right), \tag{4.1}$$

$$\text{and} \quad \mathbf{K}_{s,new}^{(1,2)} = v_{new}^{(2,2),(1,2)} + H\left(\mathbf{K}_s^{(2,2)}, d^{(1,2)}\right). \tag{4.2}$$

$\mathbf{K}_s^{(2,2)}$ in (4.1) and (4.2) are the same since there is no membership update in the service group $SG_{(2,2)}$. Therefore, if $d^{(1,2)}$ is not changed, the output of the hash function $H\left(\mathbf{K}_s^{(2,2)}, d^{(1,2)}\right)$ remains the same, and thus

$$\text{(4.2)-(4.1)} \Rightarrow \mathbf{K}_{s,new}^{(1,2)} - \mathbf{K}_{s,old}^{(1,2)} = v_{new}^{(2,2),(1,2)} - v_{old}^{(2,2),(1,2)}. \tag{4.3}$$

In (4.3), $v_{new}^{(2,2),(1,2)}$ and $v_{old}^{(2,2),(1,2)}$ are public values known by all users. For the joining user $u_8^{(1,2)}$, since he/she knows the new key $\mathbf{K}_{s,new}^{(1,2)}$, he/she can easily use (4.3) to calculate $\mathbf{K}_{s,old}^{(1,2)}$ and can access the content broadcasted before he/she joins. Similarly, for a leaving user $u_8^{(1,2)}$ who has the old key $\mathbf{K}_{s,old}^{(1,2)}$, he/she can also derive the new key.

In addition, when user $u_8^{(1,2)}$ joins/leaves the service group $SG_{(1,2)}$, $d^{(1,1)}$ also needs to be updated. This is because $\mathbf{K}_s^{(1,1)}$ has two parents $\mathbf{K}_s^{(2,1)}$ and $\mathbf{K}_s^{(1,2)}$, which means that there are two equations, $\mathbf{K}_s^{(1,1)} = v^{(2,1),(1,1)} + H\left(\mathbf{K}_s^{(2,1)}, d^{(1,1)}\right)$ and $\mathbf{K}_s^{(1,1)} = v^{(1,2),(1,1)} + H\left(\mathbf{K}_s^{(1,2)}, d^{(1,1)}\right)$, that can be used to derive $\mathbf{K}_s^{(1,1)}$. Assume that $d^{(1,1)}$ is not changed but $\mathbf{K}_s^{(1,2)}$ has been updated already. In this example, since $\mathbf{K}_s^{(1,2)}$ is updated, the new/leaving

user $u_8^{(1,2)}$ cannot use (4.3) to derive the new key $\mathbf{K}_s^{(1,1)}$. However, he/she can still derive the new key $\mathbf{K}_s^{(1,1)}$ using the public information $d^{(1,1)}$ and $v^{(2,1),(1,1)}$. Note that the old and the new versions of $\mathbf{K}_s^{(1,1)}$ satisfy:

$$\mathbf{K}_{s,old}^{(1,1)} = v_{old}^{(2,1),(1,1)} + H\left(\mathbf{K}_s^{(2,1)}, d^{(1,1)}\right), \tag{4.4}$$

$$\text{and} \quad \mathbf{K}_{s,new}^{(1,1)} = v_{new}^{(2,1),(1,1)} + H\left(\mathbf{K}_s^{(2,1)}, d^{(1,1)}\right). \tag{4.5}$$

With $H\left(\mathbf{K}_s^{(2,1)}, d^{(1,1)}\right)$ unchanged,

$$\text{(4.5)-(4.4)} \Rightarrow \mathbf{K}_{s,new}^{(1,1)} - \mathbf{K}_{s,old}^{(1,1)} = v_{new}^{(2,1),(1,1)} - v_{old}^{(2,1),(1,1)}. \tag{4.6}$$

Since $v_{old}^{(2,1),(1,1)}$ and $v_{new}^{(2,1)(1,1)}$ are public information, user $u_8^{(1,2)}$ can derive the old/new version of $\mathbf{K}_s^{(1,1)}$ using (4.6), which violates the forward/backward security requirement. Therefore, in the above example, even though there is no membership update in $SG_{(1,1)}$, $d^{(1,1)}$ has to be updated too.

In general, when a user joins/leaves a service group $SG_{\mathbf{I}}$, the public value of the associated $DG_{\mathbf{I}}$ and all of its descendants must be updated. A simple and efficient way to update $d^{\mathbf{I}}$ is to use a secure one-way function $f(\cdot)$ that is previously agreed upon by KDC and all users, and let $d_{new}^{\mathbf{I}} = f\left(d_{old}^{\mathbf{I}}\right)$. It introduces little communication and computation overhead that can often be ignored.

4.1.3 Session Key Update for a User Join

When a user joins the service, to ensure backward security, all session keys to be known by him/her should be updated. Similar to the update of the service group subtree, an efficient way to update session keys for a user join is to use the secure one-way function $f(\cdot)$.

For example, in Figure 3.4, when user $u_8^{(1,2)}$ joins the service group $SG_{(1,2)}$, KDC puts him/her under the node labeled $\mathbf{K}_{e2}^{(1,2)}$ in the $SG_{(1,2)}$ subtree, and updates 2 keys in the data group subtree: the new $\mathbf{K}_s^{(1,2)}$ should be known by service groups $SG_{(2,2)}$ and $SG_{(1,2)}$,

and all users in the service should have the new $\mathbf{K}_s^{(1,1)}$. First, the one-way function $f(\cdot)$ is used to update $d^{(1,2)}$ and $d^{(1,1)}$. Second, KDC and users in $SG_{(2,2)}$ and $SG_{(1,2)}$ use the one-way function to calculate $\mathbf{K}_{s,new}^{(1,2)} = f\left(\mathbf{K}_{s,old}^{(1,2)}\right)$, and similarly, KDC and all users calculate $\mathbf{K}_{s,new}^{(1,1)} = f\left(\mathbf{K}_{s,old}^{(1,1)}\right)$. The third step in the rekey process is to update the edge values. This is to ensure that for an arrow connecting a parent $\mathbf{K}_s^{\mathbf{I}}$ and a child $\mathbf{K}_s^{\mathbf{J}}$, the edge value satisfies $v^{\mathbf{I},\mathbf{J}} = \mathbf{K}_s^{\mathbf{J}} - H\left(\mathbf{K}_s^{\mathbf{I}}, d^{\mathbf{J}}\right)$ after the key update. In the above example, since $\mathbf{K}_s^{(1,2)}$ and $\mathbf{K}_s^{(1,1)}$ are updated, the values of three arrows that are connected to the two vertices, $v^{(2,2),(1,2)}$, $v^{(2,1),(1,1)}$ and $v^{(1,2),(1,1)}$, need to be updated. To update $v^{(2,2),(1,2)}$, KDC and users in $SG_{(2,2)}$ calculate $v_{new}^{(2,2),(1,2)} = \mathbf{K}_{s,new}^{(1,2)} - H\left(\mathbf{K}_s^{(2,2)}, d_{new}^{(1,2)}\right)$. KDC and users in $SG_{(2,2)}$ and $SG_{(2,1)}$ derive the new $v^{(2,1),(1,1)}$ using $v_{new}^{(2,1),(1,1)} = \mathbf{K}_{s,new}^{(1,1)} - H\left(\mathbf{K}_s^{(2,1)}, d_{new}^{(1,1)}\right)$. Similarly, KDC and users in $SG_{(2,2)}$ and $SG_{(1,2)}$ derive the new $v^{(1,2),(1,1)}$ using $v_{new}^{(1,2),(1,1)} = \mathbf{K}_{s,new}^{(1,1)} - H\left(\mathbf{K}_{s,new}^{(1,2)}, d_{new}^{(1,1)}\right)$. Finally, KDC unicasts to the joining user $u_8^{(1,2)}$ all the keys and public information that he/she needs.

In general, with sequential rekeying, for a user joining service group $SG_{\mathbf{I}}$, all keys in $Q_{\mathbf{I}}$, including the session key $\mathbf{K}_s^{\mathbf{I}}$ associated with $SG_{\mathbf{I}}$ and all its descendants, must be updated.

- First, for every session key $\mathbf{K}_s^{\mathbf{J}} \in Q_{\mathbf{I}}$, KDC and users find the corresponding vertex and use the one-way function $f(\cdot)$ to update its public value $d^{\mathbf{J}}$.
- Then, for every session key $\mathbf{K}_s^{\mathbf{J}} \in Q_{\mathbf{I}}$, KDC and users use the one-way function $f(\cdot)$ to update $\mathbf{K}_s^{\mathbf{J}}$.
- Finally, for every session key $\mathbf{K}_s^{\mathbf{J}} \in Q_{\mathbf{I}}$, KDC and users check all the arrows connecting $\mathbf{K}_s^{\mathbf{J}}$ and its parents. For each parent $\mathbf{K}_s^{\mathbf{M}=(m_1,\cdots,m_N)}$ where $\sum_{k=1}^{N} |m_k - j_k| = 1$, the public value of the corresponding arrow connecting $\mathbf{K}_s^{\mathbf{M}}$ and $\mathbf{K}_s^{\mathbf{J}}$ is updated as $v_{new}^{\mathbf{M},\mathbf{J}} = \mathbf{K}_{s,new}^{\mathbf{J}} - H\left(\mathbf{K}_s^{\mathbf{M}}, d_{new}^{\mathbf{J}}\right)$.

The computation cost of the one-way function $f(\cdot)$ and the Hash function $H(\cdot)$ is much smaller than that of encryption and decryption and can often be ignored. Thus, from the

above analysis, for a user join, the above proposed method introduces no overhead to update session keys in the data group subtree.

4.1.4 Session Key Update for a User Leave

In the example in Figure 3.4, when user $u_8^{(1,2)}$ leaves the service group $SG_{(1,2)}$, $Q_{(1,2)} = \left\{\mathbf{K}_s^{(1,2)}, \mathbf{K}_s^{(1,1)}\right\}$ in the POSET need to be updated. KDC first updates $\mathbf{K}_s^{(1,2)}$ that is associated with the service group $SG_{(1,2)}$, and lets $SG_{(2,2)}$ and $SG_{(1,2)}$ have the new $\mathbf{K}_s^{(1,2)}$. There are two possible methods to update $\mathbf{K}_s^{(1,2)}$ using public information. In the first algorithm,

- First, KDC and users update the vertex value $d^{(1,2)}$ using the one-way function, and $d_{new}^{(1,2)} = f\left(d_{old}^{(1,2)}\right)$.
- Then, KDC randomly generates a new key $\mathbf{K}_{s,new}^{(1,2)}$. To let the remaining users in service group $SG_{(1,2)}$ get the new key, KDC uses the newly updated $\mathbf{K}_{g,new}^{(1,2)}$ to encrypt $\mathbf{K}_{s,new}^{(1,2)}$ and sends a rekey message $\{\mathbf{K}_{s,new}^{(1,2)}\}_{\mathbf{K}_{g,new}^{(1,2)}}$.
- Finally, KDC updates and broadcasts the new edge value $v_{new}^{(2,2),(1,2)}$ through $v_{new}^{(2,2),(1,2)} = \mathbf{K}_{s,new}^{(1,2)} - H\left(\mathbf{K}_s^{(2,2)}, d_{new}^{(1,2)}\right)$, which enables users in $SG_{(2,2)}$ to derive $\mathbf{K}_{s,new}^{(1,2)}$.

In the second algorithm,

- First, KDC and users update $d^{(1,2)}$ using the one-way function $f(\cdot)$.
- Then, KDC keeps the edge value $v^{(2,2),(1,2)}$ unchanged. KDC and $SG_{(2,2)}$ calculate $\mathbf{K}_{s,new}^{(1,2)} = v^{(2,2),(1,1)} + H\left(\mathbf{K}_s^{(2,2)}, d_{new}^{(1,2)}\right)$, which enables users in $SG_{(2,2)}$ to have access to the new key.
- Finally, KDC sends a rekey message $\{\mathbf{K}_{s,new}^{(1,2)}\}_{\mathbf{K}_{g,new}^{(1,2)}}$ to the remaining users in $SG_{(1,2)}$.

Note that in both algorithms, the update of $d^{(1,2)}$ changes the output of the hash function $H\left(\mathbf{K}_s^{(2,2)}, d^{(1,2)}\right)$, and the secure hash function $H(\cdot)$ prevents the leaving user $u_8^{(1,2)}$ from

deriving the new session key $K_{s,new}^{(1,2)}$. Thus, both algorithms are secure. When considering the rekey cost of the two algorithms, to update $\mathbf{K}_s^{(1,2)}$, Algorithm 1 needs 1 rekey message to service group $SG_{(1,2)}$ and 1 message to update $v^{(2,2),(1,2)}$, while Algorithm 2 requires only 1 rekey message to service group $SG_{(1,2)}$. Since both algorithms are secure while Algorithm 2 introduces a smaller communication overhead, the second algorithm is used in this work.

The last step is to update $\mathbf{K}_s^{(1,1)}$ and let all users have the new session key to decrypt the data group $DG_{(1,1)}$. To update $\mathbf{K}_s^{(1,1)}$,

- First, KDC and all users update the vertex value $d^{(1,1)}$ using $d_{new}^{(1,1)} = f\left(d_{old}^{(1,1)}\right)$.
- Then, KDC uses the arrow connecting $\mathbf{K}_s^{(1,2)}$ and $\mathbf{K}_s^{(1,1)}$ to update $\mathbf{K}_s^{(1,1)}$ and calculates $\mathbf{K}_{s,new}^{(1,1)} = v^{(1,2),(1,1)} + H\left(\mathbf{K}_{s,new}^{(1,2)}, d_{new}^{(1,1)}\right)$. Note that service group $SG_{(2,2)}$ and $SG_{(1,2)}$ also have knowledge of $\mathbf{K}_{s,new}^{(1,2)}$. Thus, they can use the same method to calculate $\mathbf{K}_{s,new}^{(1,1)}$.
- Then, KDC encrypts $\mathbf{K}_{s,new}^{(1,1)}$ with $SG_{(1,1)}$'s group key $\mathbf{K}_g^{(1,1)}$ and sends the encrypted rekey message $\{\mathbf{K}_{s,new}^{(1,1)}\}_{\mathbf{K}_g^{(1,1)}}$ to the service group $SG_{(1,1)}$.
- Finally, to let users in the service group $SG_{(2,1)}$ get the new key, KDC uses the edge connecting $\mathbf{K}_s^{(2,1)}$ and $\mathbf{K}_s^{(1,1)}$, calculates and broadcasts the new edge value $v_{new}^{(2,1),(1,1)} = \mathbf{K}_{s,new}^{(1,1)} - H\left(\mathbf{K}_s^{(2,1)}, d_{new}^{(1,1)}\right)$.

Therefore, to update $\mathbf{K}_s^{(1,1)}$, KDC sends one encrypted rekey message to $SG_{(1,1)}$ and one message to update the edge value $v^{(2,1),(1,1))}$. Note that in Figure 3.3, $\mathbf{K}_s^{(1,1)}$ has two parents, $\mathbf{K}_s^{(2,1)}$ and $\mathbf{K}_s^{(1,2)}$. In the above example, the new key $\mathbf{K}_{s,new}^{(1,1)}$ is generated using $\mathbf{K}_s^{(1,2)}$ and the edge value $v^{(1,2),(1,1)}$, and then update the other edge value $v^{(2,1),(1,1)}$. An alternative way to generate the new key $\mathbf{K}_{s,new}^{(1,1)}$ is to use the other parent $\mathbf{K}_s^{(2,1)}$ and the corresponding edge value $v^{(2,1),(1,1)}$, and then update $v^{(1,2),(1,1)}$. Both methods give the same rekey cost,

and either can be selected to update $\mathbf{K}_s^{(1,1)}$ as long as KDC and all users use the same method.

In general, to ensure forward security, when a user leaves $SG_{\mathbf{I}}$, KDC updates the session keys in $Q_{\mathbf{I}}$ one by one. To update a key $\mathbf{K}_s^{\mathbf{J}} \in Q_{\mathbf{I}}$,

- First, KDC and users update the corresponding public label $d^{\mathbf{J}}$ using one-way function $d_{new}^{\mathbf{J}} = f\left(d_{old}^{\mathbf{J}}\right)$.
- If the vertex $\mathbf{K}_s^{\mathbf{J}}$ does not have any parent, KDC randomly generates a new session key $\mathbf{K}_{s,new}^{\mathbf{J}}$. Otherwise, KDC selects one of its parents $\mathbf{K}_s^{\mathbf{M}=(m_1,\cdots,m_N)}$ associated with service group $SG_{\mathbf{M}}$, where $\sum_{k=1}^{N}|m_k - j_k| = 1$, and calculates $\mathbf{K}_{s,new}^{\mathbf{J}} = v^{\mathbf{M},\mathbf{J}} + H(\mathbf{K}_s^{\mathbf{M}}, d_{new}^{\mathbf{J}})$. Then, KDC encrypts $\mathbf{K}_{s,new}^{\mathbf{J}}$ using $\mathbf{K}_g^{\mathbf{J}}$, and sends the encrypted rekey message to current users in $SG_{\mathbf{I}}$. Users in $SG_{\mathbf{M}}$ and $SG_{\mathbf{M}}$'s ancestors also use $v^{\mathbf{M},\mathbf{J}}$ and the same method to derive $\mathbf{K}_{s,new}^{\mathbf{J}}$.
- Finally, KDC checks all the other parents of $\mathbf{K}_s^{\mathbf{J}}$. For each parent $\mathbf{K}_s^{\mathbf{G}=(g_1,\cdots,g_N)}$where $\sum_{k=1}^{N}|g_k - j_k| = 1$, KDC calculates and broadcasts the new edge value $v_{new}^{\mathbf{G},\mathbf{J}} = \mathbf{K}_{s,new}^{\mathbf{J}} - H\left(\mathbf{K}_s^{\mathbf{G}}, d_{new}^{\mathbf{J}}\right)$. This enables users in service group $SG_{\mathbf{G}}$ to derive $\mathbf{K}_{s,new}^{\mathbf{J}}$.

To summarize, the proposed rekey scheme uses the secure hash function and public information to securely update the session keys while significantly reducing the rekey cost.

4.2 The Reconnection Scheme

As discussed in Section 3.3, high reconnection rate and large number of short sessions are observed in live streaming applications [19]. Sequential rekeying treats each reconnection as one leave and one independent user join to achieve strict forward security, which results in large rekeying overhead. In reality, many live broadcast applications do not require strict

forward and backward security, and can allow content leak up to T_l seconds. (The value of T_l is determined by the content owner and is application dependent.) With the relaxed requirement on forward security, KDC allows a leaving user to keep his/her keys (and thus access to the content) for up to T_l seconds after his/her leaving. In this scenario, if a user rejoins the same service group in less than T_l seconds after his/her leaving, KDC puts the rejoining user to the same leaf in the SG subtree and does not need to update the keys he/she previously had. Consequently, there is no cost associated with the rejoining user if he/she comes back within T_l seconds.

For applications that do not require strict forward security, similar to the work in [31], a wait-for-leaving (WFL) list is used to address the high reconnection rate. For each service group, a WFL list is built to record information of each leaving user, including the user's identity and his/her leaving time. In this work, a rejoining user's absent time is defined as the duration between his/her leaving and his/her rejoining of the *same* service group. A user is removed from the WFL lists when his/her absent time is larger than T_l or when he/she rejoins the same service group within T_l seconds after his/her leaving. In this work, it is assumed that KDC has sufficient computation capability and storage space to maintain the WFL lists.

The proposed reconnection scheme is as follows:

- When user $u_i^{\mathbf{I}}$ leaves the service group $SG_{\mathbf{I}}$, KDC records $u_i^{\mathbf{I}}$'s identity and leaving time $t_{u_i^{\mathbf{I}}}$ in the $\text{WFL}_{\mathbf{I}}$ list while not removing $u_i^{\mathbf{I}}$ from the corresponding ALX subtree. KDC still allows $u_i^{\mathbf{I}}$ to continue receiving the service in the interval from $t_{u_i^{\mathbf{I}}}$ to $t_{u_i^{\mathbf{I}}} + T_l$. Then, KDC checks all other users in all WFL lists, removes from the lists those whose absent time exceeds T_l, and update the keys using the sequential rekeying algorithm. For example, assume that user $u_j^{\mathbf{J}}$ is in the $\text{WFL}_{\mathbf{J}}$ list for service group $SG_{\mathbf{J}}$ at the current time $t_{u_i^{\mathbf{I}}}$. If $t_{u_i^{\mathbf{I}}} - t_{u_j^{\mathbf{J}}} \geq T_l$, $u_j^{\mathbf{J}}$ is removed from both the $\text{WFL}_{\mathbf{J}}$ list and the corresponding ALX subtree for service group $SG_{\mathbf{J}}$, and KDC updates all the keys $u_j^{\mathbf{J}}$

knows. If $t_{u_i^{\mathbf{I}}} - t_{u_j^{\mathbf{J}}} < T_l$, KDC keeps $u_j^{\mathbf{J}}$ in the key tree and the $\text{WFL}_{\mathbf{J}}$ list.

- If no user leaves the service in the past second, KDC updates all of the WFL lists and the key tree periodically (every second) in the same way as described above. This is to ensure that for each leaving user, the maximum content leak does not exceed T_l seconds.

- When user $u_i^{\mathbf{I}}$ joins the service group $SG_{\mathbf{I}}$, KDC first checks the $\text{WFL}_{\mathbf{I}}$ list. If $u_i^{\mathbf{I}}$ is already in the $\text{WFL}_{\mathbf{I}}$ list, then $u_i^{\mathbf{I}}$ is a rejoining user. KDC simply removes $u_i^{\mathbf{I}}$'s record from the $\text{WFL}_{\mathbf{I}}$ list and does not update the key tree. Otherwise, KDC treats $u_i^{\mathbf{I}}$ as a new user, and updates the keys using the one-way function.

For those live broadcast programs that allow content leak up to T_l seconds, if a leaving user rejoins the same service group in the same batch interval as he/she leaves, the proposed reconnection scheme does not relocate him/her to another leaf and does not update the keys he/she previously had. Consequently, there is no cost associated with such rejoining users.

4.3 The Batch Rekeying Scheme

A lot of live streaming applications have the flash crowd behavior and experience high membership update rate [18], [16], [19]. For such applications, immediate key update after each membership change may cause consecutive update of the same keys within a short time interval, and sequential rekeying may introduce significant rekey overhead [56]. For example, in Figure 3.4, assume that user $u_1^{(2,2)}$ in service group $SG_{(2,2)}$ leaves the service right after two users $u_1^{(1,2)}$ and $u_8^{(1,2)}$ leave the service group $SG_{(1,2)}$. Since all three users have access to $\mathbf{K}_s^{(1,2)}$ and $\mathbf{K}_s^{(1,1)}$, KDC and the remaining users have to update these two session keys three times within a short period.

In this work, for applications that do not require strict forward security, batch rekeying proposed in [56] is used to address redundant update of the same key. Here, sequential key update is till used for a user join so that a user can access the content immediately after he/she joins the service. Batch rekeying is only applied for leaving users only, and KDC periodically updates keys known to leaving users at predefined time called "batch moments". Let the batch interval, which is the period between two consecutive batch moments, equal to T_l. It guarantees that the content that is leaked to a leaving user does not exceed the maximum allowed T_l seconds. The proposed batch rekeying scheme is as follows:

- When user $u_i^{\mathbf{I}}$ leaves the service group $\mathrm{SG}_{\mathbf{I}}$, KDC puts him/her in the $\mathrm{WFL}_{\mathbf{I}}$ list, records his/her identity, and keeps him/her in the key tree.
- When user $u_j^{\mathbf{J}}$ joins the service group $\mathrm{SG}_{\mathbf{J}}$, KDC first checks the $\mathrm{WFL}_{\mathbf{J}}$ list. If $u_j^{\mathbf{J}}$ is already in the $\mathrm{WFL}_{\mathbf{J}}$ list, then $u_j^{\mathbf{J}}$ is a rejoining user. KDC removes $u_j^{\mathbf{J}}$ from $\mathrm{WFL}_{\mathbf{J}}$ list and does not update the key tree. Otherwise, KDC treats $u_j^{\mathbf{J}}$ as a new user, and updates the keys using the one-way function.
- At each batch moment, KDC checks the identities of all leaving users in all WFL lists, identifies all keys that they know, and updates them together. Then, KDC clears all WFL lists.

Consequently, keys known by the leaving users are only updated once in one batch interval. For the above example, only 1 rekey message is needed for each session key update, while with sequential rekeying, each session key needs to be updated 3 times in a short interval.

4.4 Grouped User Placement in the ALX Tree

With batch rekeying, for each service group, the locations of leaving users in the ALX key tree affect the performance of the key management scheme. In the extreme case where there

is a leaving user in every branch of the ALX subtree, all keys in the ALX subtree have to be updated at the next batch moment, and it introduces the largest possible rekey overhead. On the contrary, if all leaving users in a service group are in the same or neighboring subtrees of an ALX SG tree, KDC only needs to update a few keys at the next batch moment, and the rekey cost is much smaller. Thus, to further reduce the rekey overhead, a possible solution is to strategically place joining users in the key tree, such that in one batch interval leaving users in the same service group are close to each other in the ALX tree.

Prior works has shown that there are many short sessions in live broadcast streaming applications, especially at the beginning of the programs [17], [19]. For such *short-stay users* who stay in the service for only a few minutes, if they come at approximately the same time, they also tend to leave in approximately the same or adjacent batch intervals. To reduce the rekey cost when short-stay users leave the service, a grouped user placement scheme is proposed. For an ALX subtree with level $l+1$, for users who joins the same service group at approximately the same time, instead of randomly placing them in the ALX subtree, the proposed scheme puts them under the same or adjacent nodes at level l. For those short-stay users who join and leave the service at approximately the same time, grouping them in the same or neighboring subtrees helps increase the number keys that they share and thus helps reduce the rekey overhead.

Chapter 5

Performance Analysis and Simulation Results

The previous chapter introduces four key update schemes, and this chapter will provide a thorough performance analysis of the communication overhead, KDC encryption overhead and user's computation overhead to update keys, and show the simulation results for each key update algorithm.

From the above rekeying algorithms in Section 4, when a user joins, KDC only needs to broadcast a message with the indices of all keys that need to be updated, and the one-way function $f(\cdot)$ is used to ensure all users have the updated keys. When a user leaves the service, to ensure forward security, KDC needs to multicast the encrypted updated keys and the updated public values $\{d^{\mathbf{I}}, v^{\mathbf{I},\mathbf{J}}\}$. Thus, the rekey cost for a leaving user is much higher than that when a user joins the service, and this thesis focuses on the rekey overhead for leaving users only. For each service group $SG_{\mathbf{I}}$, following prior works [31], [16], users' arrival process is modeled as Poisson with mean $\lambda_{\mathbf{I}}$, and each user's service duration follows the exponential distribution with mean $1/\mu_{\mathbf{I}}$. $\alpha_{\mathbf{I}}$ is defined as the probability that a leaving user rejoins the same service group later. For a rejoining user, let the interval between

his/her leave and rejoin follow the exponential distribution with mean $1/\upsilon_{\mathbf{I}}$.

5.1 Performance Analysis of Sequential Rekeying

First, the performance of sequential rekeying is analyzed. From Section 4.1, sequential rekeying achieves strict forward and backward security.

5.1.1 Analysis of C_{en}

First, the KDC's computation cost to update the ALX SG subtrees is calculated, that is, the average number of rekey messages that KDC needs to encrypt per second to update the service group subtrees. For service group $SG_{\mathbf{I}}$ with a total of $k_{\mathbf{I}}$ users, following the above user join and leave model, the average number of leaving users per second approximately equals to $k_{\mathbf{I}}\mu_{\mathbf{I}}$, and the average number of arrivals per second is $\lambda_{\mathbf{I}}$. To analyze the performance of sequential rekeying, the scenario where $k_{\mathbf{I}} \gg k_{\mathbf{I}}\mu_{\mathbf{I}}$ and $k_{\mathbf{I}} \gg \lambda_{\mathbf{I}}$ and where the variation on the total number of users in $SG_{\mathbf{I}}$ within a second can be ignored is considered. Furthermore, it is assumed that the $k_{\mathbf{I}}$ users in $SG_{\mathbf{I}}$ are uniformly distributed in an ALX tree of degree $a_{\mathbf{I}}$ and level $l_{\mathbf{I}}+1$. In this scenario, following the analysis in [31], for a user leave in $SG_{\mathbf{I}}$, to update the corresponding service group ALX key tree, the number of rekey messages that KDC needs to encrypt is $C_{en,seq}^{SG_{\mathbf{I}}} \approx k_{\mathbf{I}}/a_{\mathbf{I}}^{l_{\mathbf{I}}} + a_{\mathbf{I}}l_{\mathbf{I}} - 1$.

Then, the KDC's computation cost to update the DG subtree is analyzed. When a user leaves $SG_{\mathbf{I}}$, the session key $\mathbf{K}_s^{\mathbf{I}}$ associated with service group $SG_{\mathbf{I}}$ and all its $\left(\prod_{j=1}^{N} i_j - 1\right)$ descendants need to be updated. That is, all session keys in $Q_{\mathbf{I}}$ have to be updated, and the proposed rekey algorithm uses one rekey message to update each new session key. Take Figure 3.4 as an example, when user $u_8^{(1,2)}$ leaves $SG_{(1,2)}$, the session keys $\mathbf{K}_s^{(1,2)}$ and its descendant $\mathbf{K}_s^{(1,1)}$ need to be updated. By using the proposed rekey algorithm, one rekey message is required to update each new session key. Thus, a total of 2 rekey messages,

$\{\mathbf{K}_{s,new}^{(1,2)}\}_{\mathbf{K}_{g,new}^{(1,2)}}$ and $\{\mathbf{K}_{s,new}^{(1,1)}\}_{\mathbf{K}_{g}^{(1,1)}}$, are sent to update $\mathbf{K}_{s}^{(1,2)}$ and $\mathbf{K}_{s}^{(1,1)}$. Therefore, when a user leaves $SG_{\mathbf{I}}$, a total of $\prod_{j=1}^{N} i_j$ rekeying messages are encrypted and sent by KDC to update the session keys in $Q_{\mathbf{I}}$.

To summarize, the average number of rekey messages that KDC needs to encrypt per user leave in $SG_{\mathbf{I}}$ is $\prod_{j=1}^{N} i_j + C_{en,seq}^{SG_{\mathbf{I}}}$. Given that $k_{\mathbf{I}}\mu_{\mathbf{I}}$ users leave $SG_{\mathbf{I}}$ in one second and there are M^N service groups, the average number of rekey messages to be encrypted per second is

$$C_{en,seq} \approx \sum_{\mathbf{I}=(i_1..i_N)} k_{\mathbf{I}}\mu_{\mathbf{I}} \left(\prod_{j=1}^{N} i_j + k_{\mathbf{I}}/a_{\mathbf{I}}^{l_{\mathbf{I}}} + a_{\mathbf{I}} l_{\mathbf{I}} - 1 \right). \tag{5.1}$$

5.1.2 Analysis of C_{de}

To calculate a user's computation cost C_{de}, first, the average number of rekey messages that a user has to decrypt per second to update the service group ALX key tree is analyzed. Assume that in each service group, users are uniformly distributed in the ALX subtree. When user $u_i^{\mathbf{I}}$ leaves the service group $SG_{\mathbf{I}}$, all the $l_{\mathbf{I}}+1$ key encryption keys in the corresponding ALX subtree that he/she knows (excluding $u_i^{\mathbf{I}}$'s private key) have to be updated. Let $\mathbf{K}_{u_i^{\mathbf{I}}}^{r}$ be the key on the rth level of the ALX subtree that user $u_i^{\mathbf{I}}$ knows. Under the assumption of uniform user distribution in the key tree, averagely speaking, a total of $k_{\mathbf{I}}/a_{\mathbf{I}}^{r}$ users, including $u_i^{\mathbf{I}}$, share $\mathbf{K}_{u_i^{\mathbf{I}}}^{r}$. Thus, when user $u_i^{\mathbf{I}}$ leaves service group $\mathrm{SG}_{\mathbf{I}}$, the remaining $k_{\mathbf{I}}/a_{\mathbf{I}}^{r}-1$ users need to decrypt the new key $\mathbf{K}_{u_i^{\mathbf{I}}}^{r}$. Considering all the $l_{\mathbf{I}}+1$ key encryption keys that need to be updated in the $SG_{\mathbf{I}}$ subtree when user $u_i^{\mathbf{I}}$ leaves, the total number of rekey messages decrypted by all users in $SG_{\mathbf{I}}$ can be approximated by $C_{de,seq}^{SG_{\mathbf{I}}} \approx \sum_{r=0}^{l_{\mathbf{I}}}(k_{\mathbf{I}}/a_{\mathbf{I}}^{r}-1)$.

Next, let us consider the user's decryption cost associated with the update of session keys. In the example in Figure 3.4, when user $u_8^{(1,2)}$ leaves $SG_{(1,2)}$, to receive the updated key $\mathbf{K}_{s}^{(1,2)}$, only users in the same service group $SG_{(1,2)}$ need to decrypt the rekey message $\{\mathbf{K}_{s,new}^{(1,2)}\}_{\mathbf{K}_{g,new}^{(1,2)}}$, while users in $SG_{(2,2)}$ can derive $\mathbf{K}_{s,new}^{(1,2)}$ themselves using public information

$d_{new}^{(1,2)}$ and $v^{(2,2),(1,2)}$. Similarly, to update $K_s^{(1,1)}$, only users in $SG_{(1,1)}$ have to decrypt the rekey message, and the rest of users can derive the new key themselves using public information. In general, when session key $\mathbf{K}_s^{\mathbf{I}}$ that is associated with $SG_{\mathbf{I}}$ is revealed, only users in the corresponding service group $SG_{\mathbf{I}}$ need to decrypt the rekey message, and users subscribing to higher-quality copies can calculate the new key using the predefined hash functions and public information. In addition, when a user leaves the service group $SG_{\mathbf{I}}$, the corresponding session key $\mathbf{K}_s^{\mathbf{I}}$ and all its $\left(\prod_{j=1}^{N} i_j - 1\right)$ descendants in $Q_{\mathbf{I}} = \left\{\mathbf{K}_s^{\mathbf{J}} | \mathbf{J} \in U_{\mathbf{I}}\right\}$ need to be updated. To update $\mathbf{K}_s^{\mathbf{I}}$, all the remaining $k_{\mathbf{I}} - 1$ users in $SG_{\mathbf{I}}$ have to decrypt the rekey message; while to update a descendant $\mathbf{K}_s^{\mathbf{J}} \in Q_{\mathbf{I}}$, all the $k_{\mathbf{J}}$ users in the corresponding service group $SG_{\mathbf{J}}$ have to decrypt the rekey message. Consequently, for a user leave in $SG_{\mathbf{I}}$, the total number of decryptions by all users to update the session keys is $\left(\sum_{\mathbf{J} \in U_{\mathbf{I}}} k_{\mathbf{J}}\right) - 1$.

Since each service group $SG_{\mathbf{I}}$ has approximately $k_{\mathbf{I}}\mu_{\mathbf{I}}$ user leaving per second, the average number of rekey messages that a user has to decrypt per second is

$$C_{de,seq} \approx \sum_{\mathbf{I}} \left[k_{\mathbf{I}}\mu_{\mathbf{I}} \left(\sum_{\mathbf{J} \in U_{\mathbf{I}}} k_{\mathbf{J}} - 1 + \sum_{r=0}^{l_{\mathbf{I}}} (k_{\mathbf{I}}/a_{\mathbf{I}}^{r} - 1) \right) \right] \Big/ \left(\sum_{\mathbf{I}} k_{\mathbf{I}} \right), \tag{5.2}$$

where the denominator is the total number of users in the service.

5.1.3 Analysis of C_{msg}

To analyze C_{msg}, the average number of rekey messages that KDC needs to send per second to update the service group subtree is first considered. From the analysis in Section 4.1.1, to update keys in the SG ALX tree, the encryption-based method is used and, therefore, the number of rekey messages sent by KDC is the same as the number of encryptions performed by KDC. Thus, for a user leave in service group $SG_{\mathbf{I}}$, $C_{msg,seq}^{SG_{\mathbf{I}}} = C_{en,seq}^{SG_{\mathbf{I}}} \approx k_{\mathbf{I}}/a_{\mathbf{I}}^{l_{\mathbf{I}}} + a_{\mathbf{I}} l_{\mathbf{I}} - 1$.

Here is the analysis of the communication overhead to update the session keys. Note that in the proposed scheme in Section 4.1.4, KDC not only sends encrypted new keys to

users, but also broadcasts messages to update the edge values, both of which should be counted when analyzing C_{msg}.

For session key $\mathbf{K}_s^{\mathbf{J}}$, let $n_{\mathbf{J}}$ denote the number of $\mathbf{K}_s^{\mathbf{J}}$'s parents. In a system that supports N types of scalability with M layers for each scalability type, given $\mathbf{J} = (j_1, \cdots, j_N)$, it can be shown that $n_{\mathbf{J}} = \sum_{k=1}^{N} Ind[j_k]$, where

$$Ind[i] = \begin{cases} 1 & \text{if } i < M \\ 0 & \text{if } i = M, \end{cases} \tag{5.3}$$

is an indicator function. $Ind[j_k] = 0$ when the associated service group $SG_{\mathbf{J}}$ receives all M layers for the kth type of scalability, and $Ind[j_k] = 1$ otherwise. Following the scheme proposed in Section 4.1.4, to update $\mathbf{K}_s^{\mathbf{J}}$, if $\mathbf{K}_s^{\mathbf{J}}$ has no parent (that is, $n_{\mathbf{J}} = 0$), then KDC sends one message with the encrypted new key to the associated service group $SG_{\mathbf{J}}$. If $n_{\mathbf{J}} > 0$, following Section 4.1.4, KDC sends one message with the encrypted new key $\mathbf{K}_s^{\mathbf{J}}$ to the associated service group $SG_{\mathbf{J}}$, and $n_{\mathbf{J}} - 1$ messages to update the edge values. Therefore, to update the session key $\mathbf{K}_s^{\mathbf{J}}$, KDC needs to send a total of $\max(1, n_{\mathbf{J}})$ messages.

For a user leave in $SG_{\mathbf{I}}$, all keys in $Q_{\mathbf{I}}$ need to be updated, and therefore, the total number of messages sent by KDC is $C_{msg,seq}^{DG_I} = \sum_{\mathbf{J} \in U_{\mathbf{I}}} \max(1, n_{\mathbf{J}})$. Assume that an average of $k_{\mathbf{I}}\mu_{\mathbf{I}}$ users leave service group $SG_{\mathbf{I}}$ in one second and there are a total of M^N service groups. With sequential rekeying, the average number of rekey messages sent by KDC per second is

$$C_{msg,seq} \approx \sum_{\mathbf{I}} \left[k_{\mathbf{I}}\mu_{\mathbf{I}} \left(k_{\mathbf{I}}/a_{\mathbf{I}}^{l_{\mathbf{I}}} + a_{\mathbf{I}} l_{\mathbf{I}} - 1 + \sum_{\mathbf{J} \in U_{\mathbf{I}}} \max(1, n_{\mathbf{J}}) \right) \right]. \tag{5.4}$$

5.1.4 Performance Comparison

To compare the performance of the proposed POSET hash-based key management scheme with that of the independent (InS) and the multi-group (MGS) key management schemes, a system that support N types of scalability with M layers for each type of scalability is

considered. Thus, there are a total of M^N data groups in the system. It is further assumed that all service groups have the same average number of users k and the same average user service time $1/\mu$. That is, $k_{\mathbf{I}} = k$ and $\mu_{\mathbf{I}} = \mu$ for all service groups $SG_{\mathbf{I}}$. Sequential key update is used for all three schemes.

Note that for all three key management schemes, the same ALX tree is used for the service group key tree, and thus, the rekey cost to update keys in the SG subtree is the same for all three key management schemes. Consequently, this work only compares the average rekey cost to update the DG key trees. With the above system setup, for the POSET hash-based scheme with sequential rekeying,

$$\begin{aligned} C_{en,seq}^{DG} &\approx k\mu \sum_{\mathbf{I}} \left(\prod_{j=1}^{N} i_j \right), \\ C_{de,seq}^{DG} &\approx k\mu \sum_{\mathbf{I}} \left(\sum_{\mathbf{J} \in U_{\mathbf{I}}} k - 1 \right) \Big/ \left(M^N k \right) \approx k\mu \sum_{\mathbf{I}} \left(\prod_{j=1}^{N} i_j \right) \Big/ \left(M^N \right) - \mu, \\ C_{msg,seq}^{DG} &\approx k\mu \sum_{\mathbf{I}} \left(\sum_{J \in U_{\mathbf{I}}} \max(1, n_{\mathbf{J}}) \right). \end{aligned} \tag{5.5}$$

For different values of N and M, the results are shown in Table 5.1.

From Table 5.1, the proposed POSET hash-based scheme introduces much smaller rekey cost than the independent and the multi-group scheme. For example, when $N = 1$ and $M = 3$, the proposed scheme helps reduce C_{en}, C_{de} and C_{msg} by 50% when compared with the two encryption-based schemes. In addition, the POSET Hash-based scheme reduces the rekey cost by a larger amount when N and M take larger values. For example, with $N = 1$ and $M = 2$, the proposed scheme helps reduce the rekey cost to approximately 60% when compared with the independent scheme. With $N = M = 3$, C_{en}, C_{de} and C_{msg} are reduced to approximately 7.3%, 7.4%, and 18%, respectively, when compared with the InS scheme.

In summary, the InS scheme manages keys in each service group independently, and thus the InS key tree is much simpler than the MGS scheme and the POSET Hash-based

TABLE 5.1

Performance comparison of different key management schemes.

	$C^{DG}_{en,seq}/(M^N k\mu)$			$C^{DG}_{de,seq}/(M^N k\mu)$			$C^{DG}_{msg,seq}/(M^N k\mu)$		
$\langle N,M\rangle$	InS	MGS	POSET	InS	MGS	POSET	InS	MGS	POSET
$\langle 1,2\rangle$	2.5	3.5	1.5	1.25	2.25	0.75	2.5	3.5	1.5
$\langle 1,3\rangle$	4.67	5.33	2	1.56	3	0.67	4.67	5.33	2
$\langle 2,2\rangle$	5.25	7.25	2.25	1.31	3.06	0.56	5.25	7.25	3.25
$\langle 2,3\rangle$	20.78	14.56	4	2.31	4.83	0.44	20.78	14.56	6.78
$\langle 3,2\rangle$	15.63	13.38	3.38	1.95	3.89	0.42	15.63	13.38	6.88
$\langle 3,3\rangle$	110.26	36.63	8	4.08	8.17	0.3	110.26	36.63	20.04

scheme. However, since it does not address the data dependency in scalable coding, it introduces the largest rekey overhead among these three key management schemes. To address the data dependency in scalable coding, the MGS scheme introduces one more layer of key encryption keys to connect session keys in the DG subtree, which helps reduce the rekey overhead. However, due to the extra layer of key encryption keys, user storage overhead increases when compared with the InS scheme. The proposed scheme, POSET Hash-based scheme, introduces public information $\{\mathbf{d},\mathbf{v}\}$ to connect session keys in the DG subtree, and reduces the rekey overhead to update session keys by using hash functions.

5.2 Performance Analysis of the Reconnection Scheme

From the above analysis, the POSET Hash-based scheme gives a much smaller rekey overhead than the InS and the MGS schemes. Therefore, in the following sections, only the POSET Hash-based scheme will be consider, and the sequential rekeying method is used

as the benchmark.

In the reconnection scheme, for a given T_l, "retry users" is used to denote those rejoining users who come back and join the same service group within T_l seconds after they leave. And the term "departure users" is used to denote all other leaving users, including those who never come back, users who join different service groups, and those rejoining users whose absent time is larger than T_l. In the dynamics model, each user who leaves service group $SG_{\mathbf{I}}$ has a probability of $\alpha_{\mathbf{I}}$ to rejoin the same service group, and the absent time of a rejoining user follows an exponential distribution with mean $1/\upsilon_{\mathbf{I}}$. So the probability that a leaving user rejoins the same service group $SG_{\mathbf{I}}$ within T_l seconds after he/she leaves is $P_{\mathbf{I}}^{re} = \alpha_{\mathbf{I}} \int_0^{T_l} \upsilon_{\mathbf{I}} e^{-t\upsilon_{\mathbf{I}}} dt = \alpha_{\mathbf{I}}(1 - e^{-T_l \upsilon_{\mathbf{I}}})$.

Given a total of $k_{\mathbf{I}}$ users in the service group $SG_{\mathbf{I}}$, this thesis considers the scenario where the total number of users is approximately the same within T_l seconds, and the change on $k_{\mathbf{I}}$ can be ignored. As such, the average number of departure users in $SG_{\mathbf{I}}$ per second is $(1 - P_{\mathbf{I}}^{re}) k_{\mathbf{I}} \mu_{\mathbf{I}}$. Note that the proposed reconnection algorithm only removes the rekey redundancies associated with the retry users, while KDC still updates all the keys that a departure user has. Therefore,

$$\begin{aligned}
\mathbf{C}_{en,retry} &\approx \sum_{\mathbf{I}} (1 - P_{\mathbf{I}}^{re}) k_{\mathbf{I}} \mu_{\mathbf{I}} \left(\prod_{j=1}^{N} i_j + k_{\mathbf{I}} / a_{\mathbf{I}}^{l_{\mathbf{I}}} + a_{\mathbf{I}} l_{\mathbf{I}} - 1 \right), \\
\mathbf{C}_{de,retry} &\approx \sum_{\mathbf{I}} \left[(1 - P_{\mathbf{I}}^{re}) k_{\mathbf{I}} \mu_{\mathbf{I}} \left(\sum_{\mathbf{J} \in U_{\mathbf{I}}} k_{\mathbf{J}} - 1 + \sum_{r=0}^{l_{\mathbf{I}}} (k_{\mathbf{I}} / a_{\mathbf{I}}^{r} - 1) \right) \right] \Bigg/ \left(\sum_{\mathbf{I}} k_{\mathbf{I}} \right), \\
\text{and } C_{msg,retry} &\approx \sum_{\mathbf{I}} \left[(1 - P_{\mathbf{I}}^{re}) k_{\mathbf{I}} \mu_{\mathbf{I}} \left(k_{\mathbf{I}} / a_{\mathbf{I}}^{l_{\mathbf{I}}} + a_{\mathbf{I}} l_{\mathbf{I}} - 1 + \sum_{J \in U_{\mathbf{I}}} \max(1, n_{\mathbf{J}}) \right) \right]. \qquad (5.6)
\end{aligned}$$

Compared with (5.1) - (5.4), for service group $SG_{\mathbf{I}}$, the proposed reconnection algorithm reduces the rekey overhead by a factor of $P_{\mathbf{I}}^{re}$. In the extreme case of $P_{\mathbf{I}}^{re} = 1$ for all possible $\mathbf{I}$, that is, all leaving users rejoin the same service group within T_l seconds after they leave, the rekey overhead is zero and no keys need to be updated. Also, from (5.6), when $\alpha_{\mathbf{I}}$ and T_l increase, $P_{\mathbf{I}}^{re}$ is larger (that is, the average number of departure users per

second is smaller), and the proposed reconnection scheme is more efficient.

To quantify the effectiveness of the proposed reconnection algorithm, η is defined as the ratio of the rekey overhead of the reconnection scheme over that of sequential rekeying. That is, $\eta_{en,retry} = \mathbf{C}_{en,retry}/\mathbf{C}_{en,seq}$, $\eta_{de,retry} = \mathbf{C}_{de,retry}/\mathbf{C}_{de,seq}$ and $\eta_{msg,retry} = \mathbf{C}_{msg,retry}/\mathbf{C}_{msg,seq}$. When η takes a smaller value, the reconnection scheme is more efficient. In addition, σ_η^2 is the variance of η.

To verify the correctness of the analysis, same as Example 1 in Section 3.1, it is assumed that there are four service groups whose user join/leave parameters are the same. For each service group, there are two types of users in this simulations: half of the users are *short-stay* users who stay in the service for a short duration with mean $1/\mu_s$, while the other half are *long-stay* users who stay in the service for a long time with mean $1/\mu_l$ [17]. For example, $(\lambda = 4, 1/\mu_l = 1500, 1/\mu_s = 240)$ means that for each service group, on average, there are 4 joining users in each second. Half of them will stay in the service for 1500 seconds on average, and the average staying time of the other half is 240 seconds. The total number of users in each service group is approximately 3400, and remains relatively unchanged in one second.

Figure 5.1 shows the analytical and the simulation results of the reconnection scheme for different user reconnection rate. The maximum allowed content leak is fixed as $T_l = 150$ seconds, and $\alpha = 0$ indicates that all leaving users leave the service permanently and there is no rejoining user. The simulation results in Figure 5.1(a) are the average rekey overhead based on 5000 simulation runs, and Figure 5.1(b) gives the variances of η for simulation results. From Figure 5.1(a), the reconnection scheme is more efficient when α is larger, and the rekey cost drops from 92% to 76% when α increases from 0.1 to 0.3. In addition, from Figure 5.1(a), the analytical and simulation results match very well. From 5.1(b), the variances of the simulation rekey overheads are small. For example, when $\alpha = 0.2$, $\sigma_{\eta_{en}}^2 < 10^{-6}$, $\sigma_{\eta_{de}}^2 < 10^{-3}$, and $\sigma_{\eta_{msg}}^2 < 10^{-6}$.

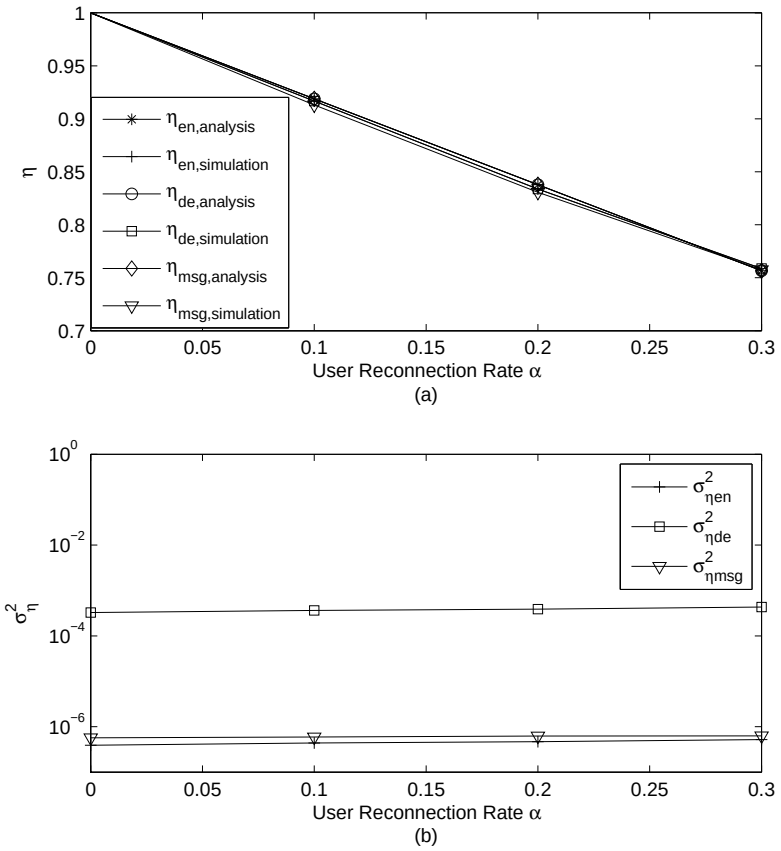

Fig. 5.1. Performance of the reconnection scheme. $\lambda = 4$, $1/\mu_l = 1500$, $1/\mu_s = 240$, $1/\upsilon = 90$ seconds, and $T_l = 150$ sec.

Then, the impact of the maximum allowed content leak on the efficiency of the reconnection scheme is studied. In Figure 5.2, $\alpha = 0.3$ is fixed, and $T_l = 0$ corresponds to the scenario where strict forward security is required. From Figure 5.2, the reconnection scheme is more efficient when the maximum allowed content leak is larger, and the rekey cost drops from 85% to 76% when T_l is increased from 40 seconds to 150 seconds. Again, the analytical and simulation results match very well. The variance of the simulation results are similar to those in Figure 5.1(b), and thus omitted here.

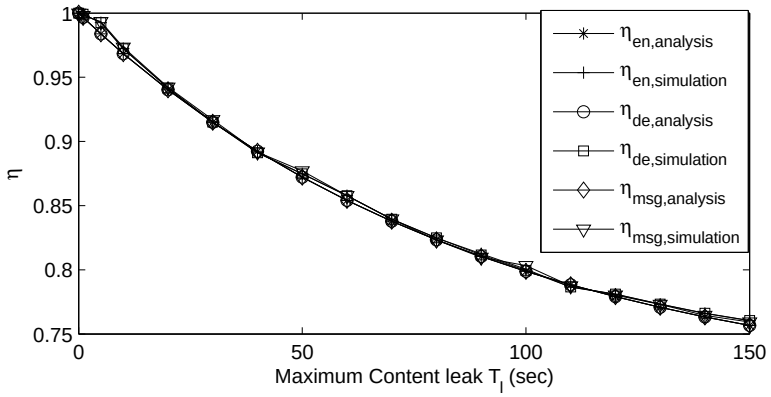

Fig. 5.2. Performance of the reconnection scheme. $\lambda = 4$, $1/\mu_l = 1500$, $1/\mu_s = 240$, $1/\upsilon = 90$ seconds, and $\alpha = 0.3$.

5.3 Performance Analysis of the Batch Rekeying Scheme

To analyze the performance of batch rekeying, the thesis considers the scenario where the number of users in service group $SG_{\mathbf{I}}$ remains approximately the same within one batch interval. Following the user join/leave model, in one batch interval of length T_l, the average number of joining users of service group $SG_{\mathbf{I}}$ is approximately $T_l\lambda_{\mathbf{I}}$, and the average number of leaving users in $SG_{\mathbf{I}}$ can be approximated by $T_l k_{\mathbf{I}}\mu_{\mathbf{I}}$.

For a user $u_i^{\mathbf{I}}$ who rejoins the service group $SG_{\mathbf{I}}$, let $t_{u_i^{\mathbf{I}}}$ be the time interval between the previous batch moment and $u_i^{\mathbf{I}}$'s leaving time, and let $\tau_{u_i^{\mathbf{I}}}$ be $u_i^{\mathbf{I}}$'s absent time (the time interval between $u_i^{\mathbf{I}}$'s leaving and rejoining). $t_{u_i^{\mathbf{I}}}$ and $\tau_{u_j^{\mathbf{I}}}$ follow exponential distributions with means $1/\mu_{\mathbf{I}}$ and $1/\upsilon_{\mathbf{I}}$, respectively. The probability that $u_i^{\mathbf{I}}$ rejoins the same service group before the next batch moment is

$$P'_{\mathbf{I}} \approx \alpha_{\mathbf{I}} P[t_{u_i^{\mathbf{I}}} + \tau_{u_i^{\mathbf{I}}} \leq T_l] = \alpha_{\mathbf{I}} \left(1 - \frac{\upsilon_{\mathbf{I}}}{\upsilon_{\mathbf{I}} - \mu_{\mathbf{I}}} e^{-T_l\mu_{\mathbf{I}}} + \frac{\mu_{\mathbf{I}}}{\upsilon_{\mathbf{I}} - \mu_{\mathbf{I}}} e^{-T_l\upsilon_{\mathbf{I}}}\right). \tag{5.7}$$

Note that if a leaving user rejoins the same service group before the next batch moment, his/her leaving incurs zero rekey cost. Therefore, for service group $SG_{\mathbf{I}}$, in each batch interval, the average number of departure users whose keys need to be updated at the next

batch moment (including those who never come back and those who rejoin the service after the next batch moment) approximately equals to $k'_{\mathbf{I}} \approx (1-P'_{\mathbf{I}})T_l k_{\mathbf{I}} \mu_{\mathbf{I}}$.

5.3.1 Analysis of C_{en}

i) To analyze C_{en}, first, the average number of messages that KDC needs to encrypt to update the service group subtree in one batch interval is considered. For batch rekeying with $k'_{\mathbf{I}}$ departure users, it considers the scenario where the departure users are uniformly distributed in the ALX key tree with degree $a_{\mathbf{I}}$ and level $l_{\mathbf{I}}+1$. It gives the upper bound of the rekey overhead for the batch rekeying scheme. In such a scenario, for a node at level j in the ALX tree, an average of $k'_{\mathbf{I}}/a^j_{\mathbf{I}}$ departure users are descendants of that node.

a) When $k'_{\mathbf{I}}/a^j_{\mathbf{I}} > 1$, that is, $0 \le j \le m_{\mathbf{I}}$ where

$$m_{\mathbf{I}} = \left\lfloor \log_{a_{\mathbf{I}}} k'_{\mathbf{I}} \right\rfloor \approx \left\lfloor \log_{a_{\mathbf{I}}} \left((1-P'_{\mathbf{I}})T_l k_{\mathbf{I}} \mu_{\mathbf{I}}\right) \right\rfloor, \tag{5.8}$$

each node at level j has at least one descendant who is a departure user, and all keys at level j have to be updated at the next batch moment. For example, if users $u_1^{(1,2)}$ and $u_8^{(1,2)}$ leave the service group $SG_{(1,2)}$ in Figure 3.4, $m_{(1,2)} = 0$, and only $\mathbf{K}_g^{(1,2)}$ at level 0 is known by both leaving users. To update $\mathbf{K}_g^{(1,2)}$ at the next moment, 3 rekey messages, $\{\mathbf{K}_{g,new}^{(1,2)}\}_{\mathbf{K}_{e0,new}^{(1,2)}}$, $\{\mathbf{K}_{g,new}^{(1,2)}\}_{\mathbf{K}_{e1}^{(1,2)}}$ and $\{\mathbf{K}_{g,new}^{(1,2)}\}_{\mathbf{K}_{e2,new}^{(1,2)}}$, are encrypted by KDC. In general, to update a key encryption key at level $0 \le j \le m_{\mathbf{I}}$ in the ALX tree, KDC needs $a_{\mathbf{I}}$ encrypted rekey messages, and there are a total of $\sum_{j=0}^{m_{\mathbf{I}}} a^j_{\mathbf{I}}$ key encryption keys in the upper $m_{\mathbf{I}}+1$ levels of the ALX subtree that need to be updated. Consequently, to update keys in the upper $m_{\mathbf{I}}+1$ levels in $\mathrm{SG}_{\mathbf{I}}$'s key tree, the total rekey messages that KDC encrypts are $C^{0,m_{\mathbf{I}}}_{en,batch} \approx a_{\mathbf{I}} \sum_{j=0}^{m_{\mathbf{I}}} a^j_{\mathbf{I}} = \sum_{j=0}^{m_{\mathbf{I}}} a^{j+1}_{\mathbf{I}} = a_{\mathbf{I}} \left(a^{m_{\mathbf{I}}+1}_{\mathbf{I}} - 1\right)/(a_{\mathbf{I}}-1)$.

b) When $k'_{\mathbf{I}}/a^j_{\mathbf{I}} < 1$, that is, $m_{\mathbf{I}} < j \le l_{\mathbf{I}}$, some of the nodes at level j do not have any departure users in their descendants. Take the above example where users $u_1^{(1,2)}$ and $u_8^{(1,2)}$ leave the service group $SG_{(1,2)}$, node $\mathbf{K}_{e1}^{(1,2)}$ in the $SG_{(1,2)}$ subtree does not need to be

updated. Under the assumption of uniform distribution of the $k'_{\mathbf{I}}$ departure users in the tree, among all $a_{\mathbf{I}}^j$ nodes at level j, $k'_{\mathbf{I}} < a_{\mathbf{I}}^j$ of them have descendants that are departure users. Thus, at level $j > m_{\mathbf{I}}$, $k'_{\mathbf{I}}$ key encryption keys need to be updated at the next batch moment. In the above example with $u_1^{(1,2)}$ and $u_8^{(1,2)}$ leaving the service group $SG_{(1,2)}$ subtree, only 2 keys, $\mathbf{K}_{e0}^{(1,2)}$ and $\mathbf{K}_{e2}^{(1,2)}$ at level 1 need to be updated.

To update a key encryption key at level $m_{\mathbf{I}}+1 \leq j \leq l_{\mathbf{I}}-1$ in the ALX tree, $a_{\mathbf{I}}$ rekeying messages are needed. To update the $k'_{\mathbf{I}}$ key encryption keys at level $l_{\mathbf{I}}$, the users' private keys is used to encrypt the new keys. Since each node at level $l_{\mathbf{I}}$ has approximately $k_{\mathbf{I}}/a_{\mathbf{I}}^{l_{\mathbf{I}}}$ children, an average of $k_{\mathbf{I}}/a_{\mathbf{I}}^{l_{\mathbf{I}}}$ rekey messages are needed to update one key at level $l_{\mathbf{I}}$. Consequently, to update keys at level $m_{\mathbf{I}}+1$ to level $l_{\mathbf{I}}$, KDC needs to encrypt and send $C_{en,batch}^{m_{\mathbf{I}}+1,l} \approx (1-P'_{\mathbf{I}})T_l k_{\mathbf{I}}\mu_{\mathbf{I}}[a_{\mathbf{I}}(l_{\mathbf{I}}-m_{\mathbf{I}}-1)+k_{\mathbf{I}}/a_{\mathbf{I}}^{l_{\mathbf{I}}}]$ rekey messages.

c) In summary, for service group $SG_{\mathbf{I}}$, the messages that KDC encrypts to update the SG key tree in each batch interval can be approximated by

$$\begin{aligned} C_{en,batch}^{SG_{\mathbf{I}}} &= C_{en,batch}^{0,m_{\mathbf{I}}} + C_{en,batch}^{m_{\mathbf{I}}+1,l} \\ &\approx a_{\mathbf{I}}\frac{a_{\mathbf{I}}^{m_{\mathbf{I}}+1}-1}{a_{\mathbf{I}}-1} + (1-P'_{\mathbf{I}})T_l k_{\mathbf{I}}\mu_{\mathbf{I}}\left[a_{\mathbf{I}}(l_{\mathbf{I}}-m_{\mathbf{I}}-1)+k_{\mathbf{I}}/a_{\mathbf{I}}^{l_{\mathbf{I}}}\right]. \end{aligned} \tag{5.9}$$

ii) Then the KDC's computation cost to update session keys in the DG subtree is analyzed. This work considers the scenario where at least one user leaves from the service group $SG_{(i_1=M,\cdots,i_N=M)}$ (which corresponds to the top node in the Hasse diagram) in each batch interval, and all session keys in the DG subtree have to be updated at the next batch moment. This gives the upper bound of the KDC's computation cost to update the DG key tree for batch rekeying. In such a scenario, at each batch moment, KDC needs to encrypt and send $C_{en,batch}^{DG} = M^N$ new session keys.

iii) Since there are M^N service groups in the service, with batch rekeying, the upper bound of KDC's computation overhead per second is

$$\overline{C}_{en,batch} \approx \frac{1}{T_l}\left(C_{en,batch}^{DG} + \sum_{\mathbf{I}} \mathbf{C}_{en,batch}^{SG_{\mathbf{I}}}\right)$$

$$\approx \quad \frac{M^N}{T_l} + \sum_{\mathbf{I}} \left\{ \frac{a_{\mathbf{I}}(a_{\mathbf{I}}^{m_{\mathbf{I}}+1}-1)}{T_l(a_{\mathbf{I}}-1)} + (1-P'_{\mathbf{I}})k_{\mathbf{I}}\mu_{\mathbf{I}} \left[a_{\mathbf{I}}(l_{\mathbf{I}}-m_{\mathbf{I}}-1) + k_{\mathbf{I}}/a_{\mathbf{I}}^{l_{\mathbf{I}}} \right] \right\}. \quad (5.10)$$

From (5.10), $\overline{C}_{en,batch}$ is a decreasing function of the batch interval T_l, and the average number of rekey messages that KDC encrypts per second is smaller when T_l is larger.

5.3.2 Analysis of C_{de}

The analysis of the upper bound of C_{de} is similar to that of C_{en}. First, the decryption cost associated with the service group key tree update is analyzed. Following the same analysis as the previous section, in one batch interval, for service group $SG_{\mathbf{I}}$, since all keys in the upper $m_{\mathbf{I}}+1$ levels need to be updated, all remaining users need to decrypt and update keys in the upper $m_{\mathbf{I}}+1$ levels that he/she knows. Hence, in each batch interval, every remaining user needs to decrypt $(m_{\mathbf{I}}+1)$ rekey messages to update the key encryption keys in the upper $(m_{\mathbf{I}}+1)$ levels. So, the average number of rekey messages all users decrypt to update keys in the upper $m_{\mathbf{I}}+1$ levels of $SG_{\mathbf{I}}$ is $C_{de,batch}^{0,m_{\mathbf{I}}} \approx k_{\mathbf{I}}(m_{\mathbf{I}}+1)$. Second, the update of keys at level $m_{\mathbf{I}}+1 \leq j \leq l_{\mathbf{I}}$ is considered. For a key at level j, an average of $k_{\mathbf{I}}/a_{\mathbf{I}}^j$ users share this key, and to update that key, the remaining $k_{\mathbf{I}}/a_{\mathbf{I}}^j - 1$ users need to decrypt the corresponding rekey message at the next batch moment. There are $k'_{\mathbf{I}}$ keys at level j that need to be updated, and therefore, to update keys at level $m_{\mathbf{I}}+1$ to $l_{\mathbf{I}}$ in the ALX subtree, the average number of rekey messages that all users need to decrypt per batch interval is $C_{de,batch}^{m_{\mathbf{I}}+1,l_{\mathbf{I}}} \approx k'_{\mathbf{I}} \left[\sum_{j=m_{\mathbf{I}}+1}^{l} (k_{\mathbf{I}}/a_{\mathbf{I}}^j - 1) \right]$. Finally, for each batch interval, for users in $SG_{\mathbf{I}}$ to update the corresponding SG key tree, the total number of decryptions performed by all users in $SG_{\mathbf{I}}$ is approximately $C_{de,batch}^{SG_{\mathbf{I}}} = C_{de,batch}^{0,m_{\mathbf{I}}} + C_{de,batch}^{m_{\mathbf{I}}+1,l_{\mathbf{I}}} \approx k_{\mathbf{I}}(m_{\mathbf{I}}+1) + k'_{\mathbf{I}} \sum_{j=m_{\mathbf{I}}+1}^{l_{\mathbf{I}}} (k_{\mathbf{I}}/a_{\mathbf{I}}^j - 1)$.

Then the user's decryption cost to update the DG key tree is analyzed. In one batch interval, when there is at least one user leaving from $\mathrm{SG}_{(M,\cdots,M)}$, all session keys in the DG tree need to be updated, and each reamining user has to update all the session keys that he/she has. For a user in $\mathrm{SG}_{\mathbf{I}}$, following the proposed scheme in Section 4.3, to find

the new version of the associated session key, he/she decrypts the corresponding rekey message. Then he/she uses public information to derive the rest of the session keys that he/she needs. For example, in Figure 3.4, if user $u_1^{(2,2)}$ leaves $SG_{(2,2)}$ during the batch interval, the remaining users in $SG_{(2,2)}$ decrypt the message $\{\mathbf{K}_{s,new}^{(2,2)}\}_{\mathbf{K}_{g,new}^{(2,2)}}$ to get the new $\mathbf{K}_s^{(2,2)}$, and use public information to derive the other three session keys. Hence, in one batch interval, the upper bound of the decryption overhead associated with session key update is one decryption per user and a total of $\sum_{\mathbf{I}} k_{\mathbf{I}}$ decryptions for all users.

Thus, with batch rekeying, the upper bound of the average number of rekey messages that one user needs to decrypt per second, can be approximated by

$$\begin{aligned}\overline{\mathbf{C}}_{de,batch} &\approx \frac{1}{T_l \sum_{\mathbf{I}} k_{\mathbf{I}}} \left(\sum_{\mathbf{I}} k_{\mathbf{I}} + \sum_{\mathbf{I}} C_{de,batch}^{SG_{\mathbf{I}}} \right) \\ &\approx \sum_{\mathbf{I}} \left[\frac{k_{\mathbf{I}}(m_{\mathbf{I}}+2)}{T_l} + (1-P'_{\mathbf{I}}) k_{\mathbf{I}} \mu_{\mathbf{I}} \sum_{j=m_{\mathbf{I}}+1}^{l_{\mathbf{I}}} (k_{\mathbf{I}}/a_{\mathbf{I}}^j - 1) \right] \Big/ \left(\sum_{\mathbf{I}} k_{\mathbf{I}} \right). \end{aligned} \tag{5.11}$$

From (5.11), $\overline{C}_{de,batch}$ is a decreasing function of the batch interval T_l.

5.3.3 Analysis of C_{msg}

In the proposed scheme, KDC sends two types of messages that contribute to C_{msg}, those that contain the encrypted new keys (both key encryption keys in the service group ALX key tree and the session keys in POSET), and those that are used to update the edge values. The number of rekey messages that contain the encrypted new keys is the same as $C_{en,batch}$, the number of rekey messages that KDC needs to encrypt. To analyze the number of messages that KDC sends to update the edge values, from Section 4.1.4, to update one session key $\mathbf{K}_s^{\mathbf{I}}$, KDC sends $\max(0, n_{\mathbf{I}} - 1)$ messages to update the edge values. Under the assumption that all session keys need to be updated at each batch moment, the number of messages that KDC sends per batch interval is upper bounded by $\sum_{\mathbf{I}} \max(0, n_{\mathbf{I}} - 1)$, and the

upper bound of $C_{msg,batch}$ can be approximated by

$$\begin{aligned}
&\overline{C}_{msg,batch} = \overline{C}_{en,batch} + \frac{1}{T_l}\sum_{\mathbf{I}} \max(0, n_{\mathbf{I}} - 1) \\
\approx\ & \frac{M^N}{T_l} + \sum_{\mathbf{I}} \left\{ \frac{a_{\mathbf{I}}(a_{\mathbf{I}}^{m_{\mathbf{I}}+1} - 1)}{T_l(a_{\mathbf{I}} - 1)} + (1 - P'_{\mathbf{I}}) k_{\mathbf{I}} \mu_{\mathbf{I}} \left[a_{\mathbf{I}}(l_{\mathbf{I}} - m_{\mathbf{I}} - 1) + k_{\mathbf{I}}/a_{\mathbf{I}}^{l_{\mathbf{I}}} \right] \right. \\
& \left. + \frac{\max(0, n_{\mathbf{I}} - 1)}{T_l} \right\}. \qquad (5.12)
\end{aligned}$$

To verify the correctness of the above analysis, Figure 5.3 compares the the upper bounds of the rekey cost and the simulation results of the batch rekeying scheme. The system setup is the same as that in Figure 5.1. From Figure 5.3, (5.10), (5.11) and (5.12) give tight upper bounds of the rekey overhead for batch rekeying. In addition, from Figure 5.3, batch rekeying helps significantly reduce the KDC's computation cost and the communication overhead. For example, with $T_l = 100$ seconds, batch rekeying helps reduce the KDC's encryption cost C_{en} to 27% when compared with sequential rekeying. In addition, batch rekeying is more efficient when the batch interval is larger. For example, when T_l increases from 50 to 150 seconds, $\eta_{en,batch} = C_{en,batch}/C_{en,seq}$ and $\eta_{msg,batch} = C_{msg,batch}/C_{msg,seq}$ are reduced from 35% to 22%, and $\eta_{de,batch} = C_{de,batch}/C_{de,seq}$ is lowered from 1% to 0.4%.

Under the same system setup, the efficiency of batch rekeying in the DG subtree and the SG subtree are investigated respectively. Figure 5.4(a) gives the analytical upper bound and the simulation results of KDC encryption overhead to update session keys in the DG subtrees. Figure 5.4(b) shows KDC computation overhead to update keys in the SG ALX subtrees, and Figure 5.4(c) is the overall rekey computation overhead at KDC. Batch rekeying reduces KDC computation overhead significantly in both subtrees, while it is more efficient to reduce KDC encryption overhead for session key update in the DG subtree. When $T_l = 100$ seconds, $\eta^{DG}_{en,batch} = 0.11\%$ while $\eta^{SG}_{en,batch} \approx 30\%$. In addition, from Figure 5.4(a), (b) and (c), when the number of users is large, KDC computation overhead of key encryption key update is dominant in overall encryption overhead at KDC. η^{batch}_{msg} is observed to

have the same trend as η_{en}^{batch} and thus omitted here.

Figure 5.5(a) and (b) show user's decryption overheads for session key update and key encryption key update with the same setup. Figure 5.5(c) is the overall decryption cost per user. Different from KDC computation overhead, user's rekey overheads in both the DG subtree and the SG subtrees fall to less then 1% when $T_l \geq 50$ seconds.

Then the impact of the membership update parameters on the efficiency of batch rekeying is studied. Same as in Figure 5.4, assume that the 4 service groups have the same set of parameters. The simulation first considers the scenario where users join and leave the service more frequently with a higher membership update rate, and select $\mathbf{R}_h \stackrel{\triangle}{=} (\lambda = 4, 1/\mu_s = 1500, 1/\mu_l = 240)$. That is, for each service group, the group size is around 3400 users, and on average, 4 new users join the service group per second. Then another set of parameters $\mathbf{R}_l \stackrel{\triangle}{=} (\lambda = 2, 1/\mu_s = 3000, 1/\mu_l = 240)$ is chosen, where for each service group, the group size is about 3200 users, and only 2 users join each service group per second on average. It corresponds to the scenario where users join and leave the service at a much lower rate. Figure 5.6 plots the simulation results of the rekey overhead for these two different scenarios. Here, $\alpha = 0.3$ and $1/\upsilon = 90$ are fixed, which are the same for all service groups. From Figure 5.6, batch rekeying is more efficient when users join and leave the service more frequently. For example, when $T_l = 100$ seconds, $\eta_{en,batch} = 0.27$, $\eta_{de,batch} = 0.04$ and $\eta_{msg,batch} = 0.25$ with $\mathbf{R}_h$, while $\eta_{en,batch} = 0.37$, $\eta_{de,batch} = 0.1$ and $\eta_{msg,batch} = 0.34$ if the membership update parameters are $\mathbf{R}_l = (\lambda = 2, 1/\mu_s = 3000, 1/\mu_l = 240)$.

5.4 Performance of Grouped User Placement

Figure 5.7 shows the performance of the grouped user placement for the POSET Hash-based scheme in Figure 3.4. Compared with random placement of joining users, grouped user placement can further help reduce the rekey cost, especially when the maximum al-

TABLE 5.2

Parameters of the user join/leave model for each service group

duration (total 7200s)	$\leq$ 900s	901-6600s	>6600s
dynamics $(\lambda, 1/\mu_l, 1/\mu_s)$	(6, 3500, 240)	(2, 1200, 240)	(0.5, 240, 240)

lowed content leak T_l is small. For example, with $T_l = 20$ seconds, grouped user placement can help further reduce η_{en} and η_{msg} by another 3%.

5.5 Simulation Results of A Popular Short-Duration Live Streaming

In this simulation, a popular short-duration live streaming of length 7200 seconds is considered. It is assumed that there are 4 data groups, same as in Figure 3.1. Time is divided into three non-overlapping periods to address different user behavior at the beginning, in the middle, and at the end of the program. There are two types of users in each service group: half of the users are *short-stay* users whose average staying time is $1/\mu_s$ and the other half are *long-stay* users with average staying time $1/\mu_l$. Assume that the membership update parameters $\{\lambda, \mu_l, \mu_s, \alpha, \upsilon\}$ are the same for all four service groups, and the parameters chosen for different types of users and for different periods of the program are given in Table 5.2. $\alpha = 0.3$ and $1/\upsilon = 90$ are fixed.

As all service groups have the same dynamics parameters, they have the same dynamics pattern. Figure 5.8 shows an example of the user number in each subgroup. From Figure 5.8, the total number of users per service group changes from a few dozen to a maximum of 3000, and in the interval between 900s and 6600s, the total number of users is around

2000 to 3000. Following the Table 3.2, an ALX subtree of degree $a = 3$ and $l = 6$ is used for each service group to maximize the performance.

Figure 5.9(a) shows the average simulation results of the proposed POSET hash-based key management scheme that uses the reconnection, batch rekeying and grouped placement methods, and Figure 5.9(b) shows the variance of the simulation results. All data in Figure 5.9 are calculated based on 72000 simulation runs. From Figure 5.9(a), when compared with sequential rekeying, the proposed scheme is more efficient when T_l is larger. When $T_l \geq 50$ seconds, the KDC's computation overhead C_{en} and the communication cost C_{msg} are reduced by more than 50%, and each user's computation cost C_{de} is reduced to less than 5%. This is consistent with the simulation results in the previous sections. Figure 5.9(b) shows that the variances of the simulation results are small. For example, when $T_l = 100$, $\sigma^2_{\eta_{en}} \approx 10^{-7}$, $\sigma^2_{\eta_{de}} \approx 10^{-4}$, and $\sigma^2_{\eta_{en}} \approx 10^{-8}$.

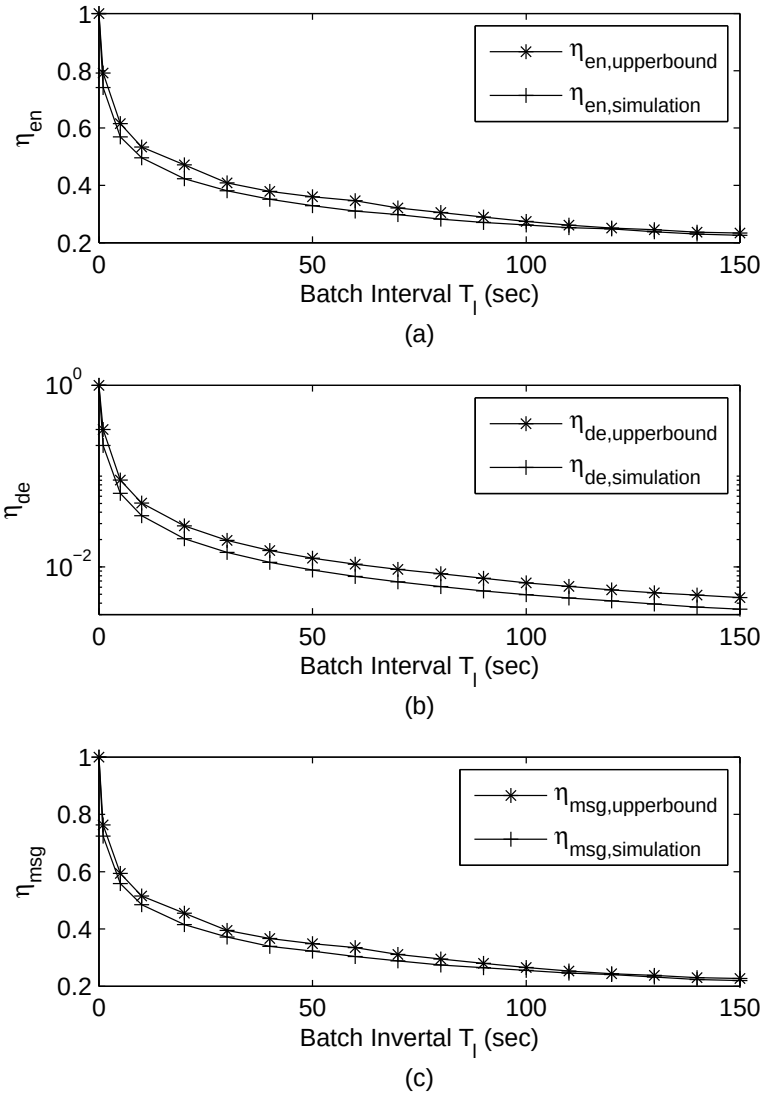

Fig. 5.3. Performance of batch rekeying. $\lambda = 4$, $1/\mu_l = 1500$, and $1/\mu_s = 240$. $1/\upsilon = 90$ seconds and $\alpha = 0.3$.

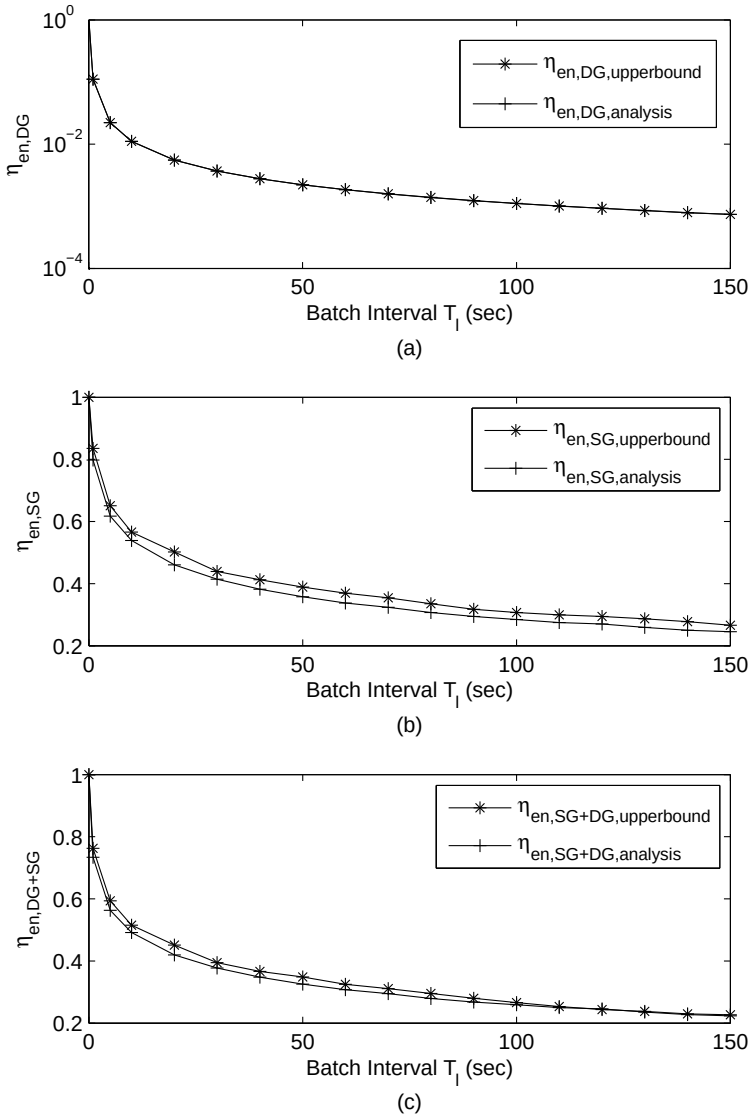

Fig. 5.4. KDC computation overhead with batch rekeying. $\lambda = 4$, $1/\mu_l = 1500$, and $1/\mu_s = 240$. $1/\upsilon = 90$ seconds and $\alpha = 0.3$.

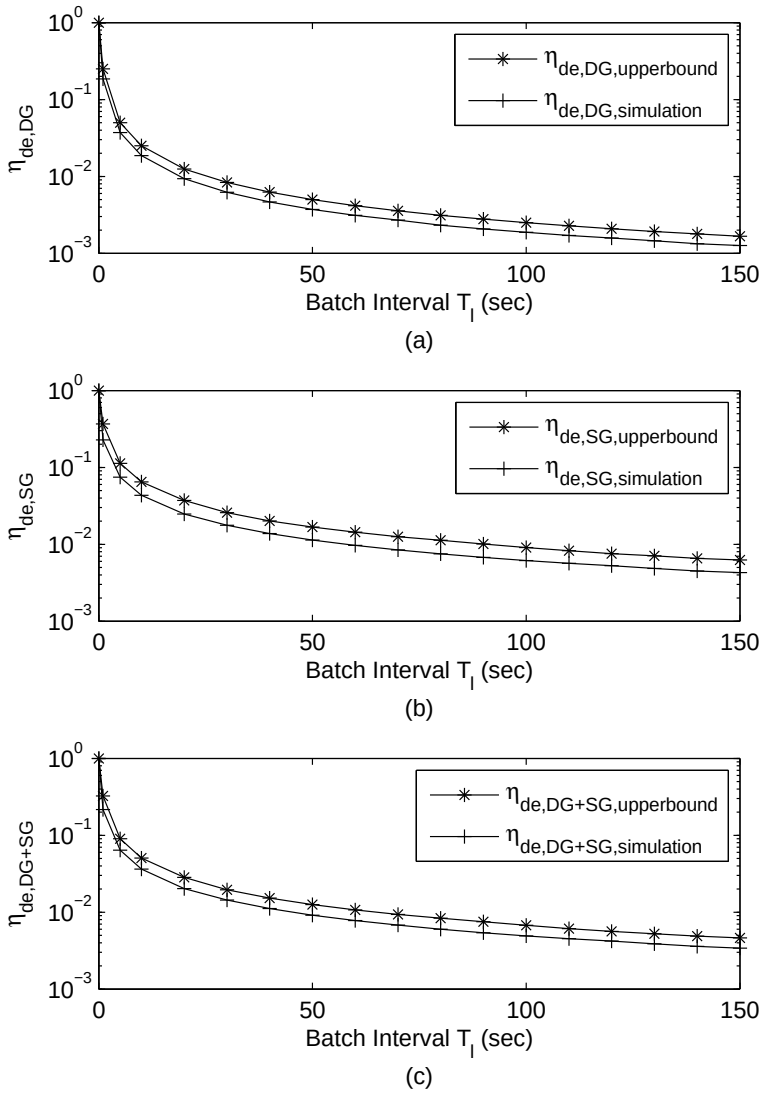

Fig. 5.5. User decryption overhead with batch rekeying. $\lambda = 4$, $1/\mu_l = 1500$, and $1/\mu_s = 240$. $1/\upsilon = 90$ seconds and $\alpha = 0.3$.

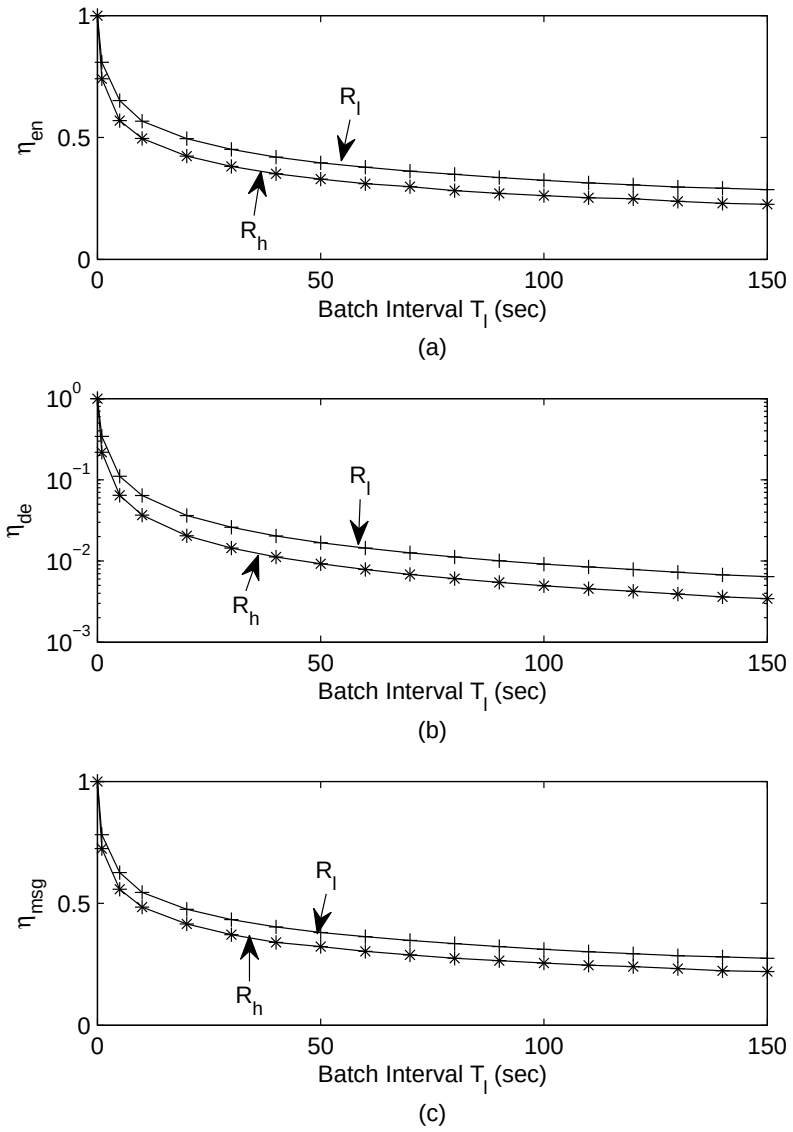

Fig. 5.6. Performance of batch rekeying for different membership update rates. $1/\upsilon = 90$ seconds and $\alpha = 0.3$.

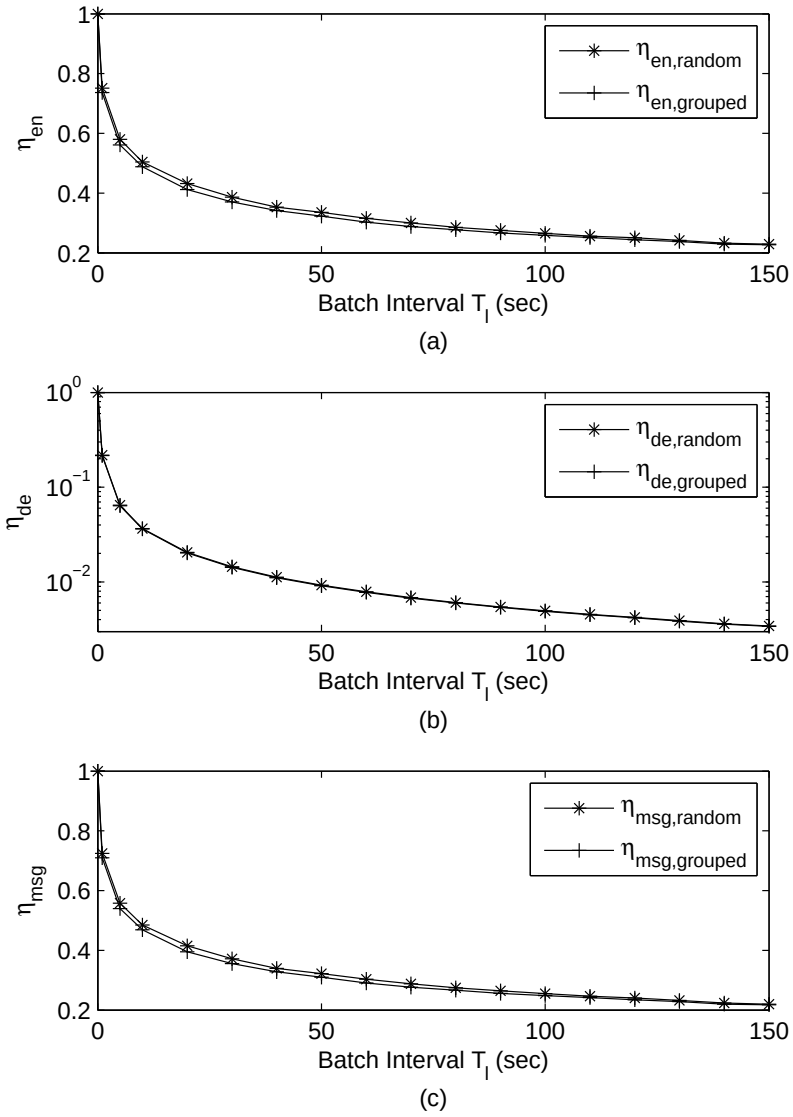

Fig. 5.7. Performance of the grouped user placement scheme. $\lambda = 4$, $1/\mu_l = 1500$, and $1/\mu_s = 240$. $1/\upsilon = 90$ seconds and $\alpha = 0.3$.

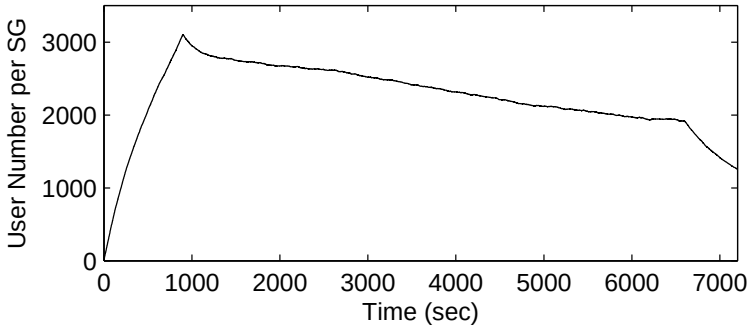

Fig. 5.8. Number of users in each service group in a short-duration live broadcast streaming application.

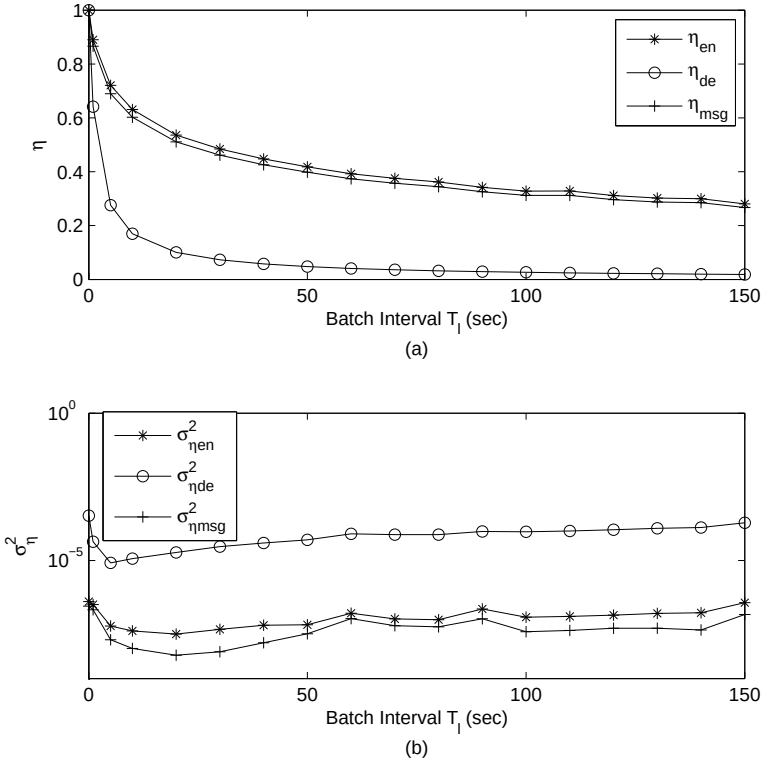

Fig. 5.9. Rekey overhead for a popular short-duration live broadcast streaming. $1/\upsilon = 90$ seconds and $\alpha = 0.3$.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis, a POSET Hash-based key management scheme for secure and efficient key update in scalable live broadcast applications is proposed. It explores the data dependency in scalable coding to reduce the rekey cost, and addresses frequent membership update in live streaming applications. The performance of the proposed scheme is analyzed and its efficiency in reducing the rekey cost is demonstrated.

This work first considers applications that require strict forward and backward security, and focuses on the design of data group key tree to address the data dependency in scalable video coding and to efficiently update the session keys. The data group key tree design uses the POSET Hash-based structure, and the proposed key management scheme uses public information and the hash function to update the session keys. The results show that when compared with the traditional encryption-based key update schemes, the proposed method can help reduce 50% to 90% of the rekey cost to update the session keys. Also, it reduces the rekey cost by a larger amount when scalable coding supports more types of scalability and more layers.

For applications that can tolerate a small amount of content leak, this work designs efficient key update schemes to address the frequent and drastic membership update in live streaming applications, and analyzes their performance. The reconnection scheme, the batch rekeying and the grouped user placement schemes are used to address the high reconnection rate, the flash crowd phenomenon, and the large number of short sessions, respectively. Both the analytical and simulation results show that the proposed schemes can help reduce the number of rekey messages and the KDC's computation cost by more than 50%, and lower the user's computation cost by more than 90%.

6.2 Future Work

The following directions would be explored in the future.

It is shown in this thesis that the rekey overhead can be reduced by exploring membership dynamics in live broadcast applications. The impact of user behavior on key management can be further investigated. For example, in scalable live broadcast applications, a user may switch from one service group to another. In the current work, a user switching is treated as a user leave followed by an independent user join, and all keys known to the switching user are updated. However, there is no need to update all those keys as the switching user may have access to the same layers. For example, in Figure 3.4, when user $u_8^{(1,2)}$ switches from $SG_{(1,2)}$ to $SG_{(2,2)}$, the session key $\mathbf{K}_s^{(2,2)}$ needs to be update using the one-way function to prevent $u_8^{(1,2)}$ from accessing the previous communication in $DG_{(2,2)}$, while the session keys $\mathbf{K}_s^{(1,2)}$ and $\mathbf{K}_s^{(1,1)}$ can remain the same since $u_8^{(1,2)}$ can still access the data groups $DG_{(1,2)}$ and $DG_{(1,1)}$. Therefore, a possible way to further reduce the rekey overhead is to investigate an efficient key update algorithm for user switch. In addition, previous works in [1], [29], [42] assumed that users' exact leaving times can be known when they join the service, and used this information to reduce the rekey cost. In the fu-

ture, this work can be extended to this scenario, and investigates how users' leaving time can help further reduce the POSET rekey cost.

In addition, a popular live broadcast application may have millions of users. For example, ESPN's digital media platforms (including ESPN.com and the ESPN mobile website) reported that there were over 2 billion total page views and more than 487 million total visits for the 2008-09 NBA regular season [57]. For such applications with large audience, a single KDC may not have sufficient computation power to handle all the key update. To address this issue, a possible solution is to follow the work in [58] and to group users into several subgroups depending on their geographical locations. Then a trusted agent is assigned to each subgroup to help update keys. The secure agent can be introduced to the proposed POSET scheme, and each SG can be divided into two parts: the agent key tree and the user key tree. In the agent key tree, the root node corresponds to the session key; and the leaves correspond to agents. In a user key tree, the root represents an agent and the leaves correspond to users. The KDC only updates the keys in the agent tree for each service group in the POSET scheme, and the agents manage keys in the user subtrees. Such a solution helps reduce the computation cost of KDC in popular live broadcast applications with millions of users.

Furthermore, this thesis focuses on centralized key management schemes where there is a KDC who generates and distributes secret keys. In reality, there are many applications where such a centralized entity cannot be found, for example, in wireless sensor networks. For such application, contributory POSET key management schemes can be investigated to securely and efficiently update keys.

References

[1] Y. Mao, Y. Sun, M. Wu, and K. J. R. Liu, "JET: Dynamic join-exit-tree amortization and schedualing for contributory key management," *IEEE/ACM Trans. on Networking*, vol. 14, no. 5, pp. 1128–1540, Oct. 2006.

[2] M. van der Schaar and P. A. Chou, *Multimedia Over IP and Wireless Networks*. Acedemic Press, 2007.

[3] Y. Mao and M. Wu, "A joint signal processing and cryptographic approach to multimedia encryption," *IEEE Trans. on Image Processing*, vol. 15, no. 7, pp. 2061–2075, July 2006.

[4] W. Zeng, J. Lan, and X. Zhuang, "Security for multimedia adaptation: Architectures and solutions," *IEEE Multimedia Magazine*, vol. 13, no. 2, pp. 68–76, Apr.-June 2006.

[5] B. Zhu, "Chapter: Multimedia encryption, multimedia security technologies for digital rights management," *Academic Press*, 2006.

[6] J. Wen, M. S. adn W. Zeng, M. Luttrell, and W. Jin, "A format-compliant configurable encryption framework for access control of video," *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 12, no. 6, pp. 545–556, June 2002.

[7] W. Zeng, J. Wen, and M. Severa, "Fast self-synchronous content scrambling by spatially shuffling codewords of compressed bitstreams," *in Proc. IEEE Int. Conf. Image Processing, 2002*, vol. 3, pp. 169–172.

[8] W. Zeng and S. Lei, "Efficient frequency domain selective scrambling of digital video," *IEEE Trans. on Multimedia*, vol. 5, no. 1, pp. 118–129, Mar. 2003.

[9] D. Balenson, D. McGrew, and A. Sherman, "Key management for large dynamic group: One way function trees and amortized initialization," *Internet Draft,draft-irtf-smug-groupkeymgmt-oft-00.txt*, 2000.

[10] M. J. Moyer, J. R. Rao, and P. Rohatgi, "A survey of security issue in multicast communications," *IEEE Networks*, vol. 13, no. 6, pp. 12–23, Nov.-Dec. 1999.

[11] S. Rafaeli and D. Hutchison, "A survey of key management for secure group communication," *ACM Computer Surveys*, vol. 35, no. 3, pp. 309–329, Sept. 2003.

[12] T. Wiegand, G. Sullivan, G. Biontegaard, and A. Luthra, "Overview of the H.264/AVC video coding standard," *IEEE Trans. on Circuits and System for Video Technology*, vol. 13, no. 7, pp. 560–576, July 2003.

[13] H. Schwarz, D. Marpe, and T. Wiegand, "Overview of the scalable video coding extension of the H.264/AVC standard," *IEEE Trans. on Circuits and System for Video Technology*, vol. 17, no. 9, pp. 1103–1120, Sept. 2007.

[14] Y. Wang, J. Ostermann, and Y. Zhang, "Video processing and communications," *Prentice Hall*, 2002.

[15] B. Zhu, Y. Yang, and T. Chen, "A DRM system supporting what you see is what you pay," *First International Conference on Digital Right Management:Technologies, Issues, Challenges and System*, 2005.

[16] K. Sripanidkulchai, B. Maggs, and H. Zhang, "An analisis of live streaming workload on the internet," *in Proc. ACM Internet Measurement Conference, 2004*, pp. 41–45.

[17] K. Sripanidkulchai, A. Ganjam, B. Maggs, and H. Zhang, "The feasibility of supperting large-scale live streaming applications with dynamic application end-points," *in Proc. 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 107–120.

[18] E. Veloso, V. Almeida, W. Meira, A. Bestavros, and S. Jin, "A hiearchical characterization of a live streaming media workload," *in Proc. 2nd ACM SIGCOMM Workshop on Internet measurment, 2002*, pp. 117–130.

[19] B. Li, G. Y. Keung, S. Xie, F. Liu, Y. Sun, and H. Yin, "An empirical study of flash crowd dynamics in a P2P-based live video streaming system," *in Proc. IEEE GLOBECOM, 2008*, pp. 1–5.

[20] H. Harney and C. Muckenhirn, "Group key management protocol (gkmp) architecture," *RFC 2094*, July. 1997.

[21] C. Wong, M. Gouda, and S. Lam, "Secure group communications using key graphs," *IEEE/ACM Trans. on Networking*, vol. 8, no. 1, pp. 16–30, Feb. 2000.

[22] D. Wallner, E.Harder, and R. Agee, "Key management for multicast: Issues and architecture," *Internet Draft,draft-wallner-key-arch-00.txt*, 1997.

[23] R. Torres, X. Sun, A. Walters, C. Rotaru, and S. Rao, "Enabling confidentiality of data delivery in an overlay broadcasting system," *in Proc. IEEE INFOCOM, 2007*, pp. 607–615.

[24] S. Zhu, S. Setia, and S. Jajodia, "Performance optimizations for group key management schemes," *in Proc. 23rd Int. Conf. distributed computering systems, 2003*, pp. 163–171.

[25] M. Waldvogel, G. Caronni, D. Sun, N. Weiler, and B. Plattner, "The versakey framework:versatile group key management," *IEEE Journal on Selected Areas In Communications*, vol. 17, no. 9, pp. 1–16, Sept. 1999.

[26] W. Trappe and L. C. Washington, *Introduction to Cryptography with Coding Theory*, 2nd ed. Prentice Hill, 2005.

[27] R. Canetti, J. Garay, G. Itkis, D. Micciancio, M. Naor, and B. Pinkas, "Multicast security: A taxonomy and some efficient constructions," *in Proc. IEEE INFORCOM 1999*, vol. 2, pp. 708–716.

[28] F. Zhou, J. Xu, L. Lin, and H. Xu, "Multicast key management scheme based on TOFT," *in Proc. 10th IEEE Int. Conf. on High Performance Computing and Commmuniaction, 2008*, pp. 1030–1035.

[29] G.Hao, N. V. Vinodchandran, B. Ramamurthy, and X. Zou, "A balanced key tree approach for dynamic secure group communication," *in Proc. 14th Int. Conf. Computer Communications and Networks, 2005*, pp. 345–350.

[30] R. Sedgewick, "Algorithms," *Addison-Wesley Publication*, 1988.

[31] Y. Sun, W. Trappe, and K. J. R. Liu, "A scalable multicast key management scheme for heterogeneous wireless networks," *IEEE/ACM Trans. on Networking*, vol. 12, no. 4, pp. 653–666, Aug. 2004.

[32] M. Burmester and Y. Desmedt, "A secure and efficient conference key distribution system," *in Proc. EUROCRYPT'94, 1994*, vol. 950, LCNS, pp. 275–286.

[33] M. Steiner, G. Tsudik, and M. Waidner, "Diffie-Hellman key distribution extended to group communication," *in Proc. ACM 3rd Conf. Computer and Communications Security, 1996*, pp. 31–37.

[34] W. Trappe, Y. Wang, and K. J. R. Liu, "Resource-aware conference key establishment for heterogeneous networks," *IEEE/ACM Trans. on Networking*, vol. 13, no. 1, pp. 134–146, Feb. 2005.

[35] ——, "Establishment of conference keys in heterogeneous networks," *in Proc. IEEE ICC, 2002*, pp. 2201–2205.

[36] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Trans. on Information Theory*, vol. 22, no. 6, pp. 644–654, Nov. 1976.

[37] K. Becher and U. Wille, "Communication complexity of group key distribution," *in Proc. 5th ACM conf. Computer and communications security, 1998*, pp. 1–6.

[38] O. Rodeh, K. Birman, and D. Dolev, "Optimized group rekey for group communication systems," *in Proc. ISOC Network and Distributed Systems Security Symposium, 2000*, pp. 275–286.

[39] Y. Kim, A. Perrig, and G. Tsudik, "Simple and fault-tolerant key agreement for dynamic collaborative group," *in Proc. 7th ACM Conf. Computer and Communications Security,2000*, pp. 235–224.

[40] L. R. Dondeti and S. Mukherjee, "Disec: A distributed framework for scalable secure many-to-many communicaitons," *in Proc. 5th IEEE Symp. Computing and Commmuniactions Security, 2008*, pp. 693–698.

[41] X. Gu, J. Yang, J. Yu, and J. Lan, "Join-tree-based contributory group key management," *in Proc. 10th IEEE International Conference on High Performance Computing and Communications, 2008*, pp. 564 – 571.

[42] X. Gu, Z. Cao, J. Yang, and J. Lan, "Dynamic contributory key management based on weighted-join-exit-tree," *in Proc. IEEE Military Communications Conference, 2008*, pp. 1 – 7.

[43] L. Eschenauer and V. D. Gligor, "A key management scheme for distributed sensor networks," *in Proc. 9th ACM Conference on Computerand Communication Security*, pp. 41–47, Nov. 2002.

[44] H. Chan, A. Perrig, and D. Song, "Random key predistribution schemes for sensor networks," *in Proc. 2003 IEEE Symposium on Security and Privacy*, pp. 197–213, 2003.

[45] I. Blake, G. Seroussi, , and N. Smart, "Advances in elliptic curve cryptography," *Cambridge University Press*, 2005.

[46] D. J. Malan, M. Welsh, and M. D. Smith, "A public-key infrastructure for key distribution in tinyos based on elliptic curve cryptography," *in Proc. First IEEE International Conference on Sensor and Ad Hoc Communications and Networks*, 2004.

[47] N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz, "Comparing elliptic curve cryptography and rsa on 8-bit cpus," *in Proc. 6th International Workshop on Cryptographic Hardware and Embedded Systems, 2005*.

[48] G. Gong, T. A. Berson, and D. R. Stinson, "Elliptic curve pseudorandom sequence generators," *in Proc. 6th Annual International Workshop, 2000*, pp. 34–48.

[49] X. Du, M. Guizani, Y. Xiao, and H. Chen, "A routing-driven elliptic curve cryptography based key management scheme for heterogeneous sensor networks," *IEEE Trans. on Wireless Communications*, pp. 1223–1229, Mar. 2009.

[50] S. Dexter, R. Belostotskiy, and A. M. Eskicioglu, "Multi-layer multicast key management with threshold cryptography," *IS&T/SPIE Symposium on Electronic Imaging, Security, Steganography, and Watermarking of Multimedia Contents VI Conference*, Jan. 2004.

[51] Y. Sun and K. J. R. Liu, "Hierarchical group access control for secure multicast communications," *IEEE/ACM Trans. on Networking*, vol. 15, no. 6, pp. 1514–1526, Dec. 2007.

[52] K. Frikken, M. Atallah, and M. Bykova, "Hash-based access control in an arbitrary hierarchy," *CERIAS Technical Report 2004-49, Purdue University*, Nov. 2004.

[53] B. Zhu, M. Feng, and S. Li, "Secure key management for flexible digital rights management of scalable codestreams," *IEEE 7th Workshop Multimedia Signal Processing, 2005*, pp. 1–4.

[54] S. Zhong, "A practical key management scheme for access control in a user hierarchy," *Computer and Security*, vol. 21, no. 8, pp. 750–759, Dec. 2002.

[55] B. Zhu, M. Feng, and S. Li, "An efficient key scheme for multiple access of JPEG 2000 and motion JPEG 2000 enabling truncations," *in Proc. 3rd IEEE Consumer Communications and Networking Conf., 2006*, vol. 2, pp. 1124–1128.

[56] X. S. Li, Y. R. Yang, M. G. Gouda, and S. S. Lam, "Batch rekeying for secure group communications," *in Proc. 10th ACM Int. World Wide Web Conf., 2001*, pp. 525–534.

[57] *http://www.espnmediazone.com/press_releases/2009_04_april/20090416_NBAAudienceGrowthAcrossESPNPlatforms.htm*.

[58] S. Mittra, "Iolus: A framework for scalable secure multicasting," *in Proc. ACM SIGCOMM'97, 1997*, pp. 277–288.