Spring 2004

A Framework for Secure Group Key Management

Sahar Mohamed Ghanem
Old Dominion University

A FRAMEWORK FOR SECURE GROUP KEY MANAGEMENT

by

Sahar Mohamed Ghanem
M.Sc. Computer Science, December 1997, Alexandria University, Egypt
B.Sc. Computer Science, June 1994, Alexandria University, Egypt

A Dissertation Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirement for the Degree of

DOCTOR OF PHILOSOPHY

COMPUTER SCIENCE

OLD DOMINION UNIVERSITY
May 2004

Approved by:

Hussein Abdel-Wahab (Director)

James Leathrum (Member)

Kurt Maly (Member)

Ravi Mukkamala (Member)

Mohammad Zubair (Member)

ABSTRACT

A FRAMEWORK FOR SECURE GROUP KEY MANAGEMENT

Sahar Mohamed Ghanem
Old Dominion University, 2004
Director: Dr. Hussein Abdel-Wahab

The need for secure group communication is increasingly evident in a wide variety of governmental, commercial, and Internet communities. Secure group key management is concerned with the methods of issuing and distributing group keys, and the management of those keys over a period of time. To provide perfect secrecy, a central group key manager (GKM) has to perform group rekeying for every join or leave request. Fast rekeying is crucial to an application's performance that has large group size, experiences frequent joins and leaves, or where the GKM is hosted by a group member. Examples of such applications are interactive military simulation, secure video and audio broadcasting, and secure peer-to-peer networks. Traditionally, the rekeying is performed periodically for the batch of requests accumulated during an inter-rekey period. The use of a logical key hierarchy (LKH) by a GKM has been introduced to provide scalable rekeying. If the GKM maintains a LKH of degree d and height h, such that the group size $n \leq d^h$, and the batch size is R requests, a rekeying requires the GKM to regenerate $O(R \times h)$ keys and to perform $O(d \times R \times h)$ keys encryptions for the new keys distribution. The LKH approach provided a GKM rekeying cost that scales to the logarithm of the group size, however, the number of encryptions increases with increased LKH degree, LKH height, or the batch size. In this dissertation, we introduce a framework for scalable and efficient secure group key management that outperforms the original LKH approach. The framework has six components as follows. First, we present a **software model** for providing secure group key management that is independent of the application, the security mechanism, and the communication protocol. Second, we focus on a LKH-based GKM and introduce a **secure key distribution technique**, in which a

rekeying requires the GKM to regenerate $O(R \times h)$ keys. Instead of encryption, we propose a novel XOR-based key distribution technique, namely XORBP, which performs an XOR operation between keys, and uses random byte patterns (BPs) to distribute the key material in the rekey message to guard against insider attacks. Our experiments show that the XORBP LKH approach substantially reduces a rekeying computation effort by more than 90%. Third, we propose two novel **LKH batch rekeying protocols**. The first protocol maintains a balanced LKH (B^+-LKH) while the other maintains an unbalanced LKH (S-LKH). If a group experiences frequent leaves, keys are deleted form the LKH and maintaining a balanced LKH becomes crucial to the rekeying's process performance. In our experiments, the use of a B^+-LKH by a GKM, compared to a S-LKH, is shown to substantially reduce the number of LKH nodes (i.e., storage), and the number of regenerated keys per a rekeying by more than 50%. Moreover, the B^+-LKH performance is shown to be bounded with increased group dynamics. Fourth, we introduce a generalized **rekey policy** that can be used to provide periodic rekeying as well as other versatile rekeying conditions. Fifth, to support distributed group key management, we identify four **distributed group-rekeying protocols** between a set of peer rekey agents. Finally, we discuss **a group member and a GKM's recovery** after a short failure time.

ACKNOWLEDGMENTS

At fisrt, I thank *God* for enlightening my way and directing me to every success I have reached and may reach in future. I am truly blessed to have all the support to complete this dissertation.

My deepest gratitude and appreciation are due to Dr. Hussein Abdel-Wahab for his continued guidance and support throughout this work. I am indebted to him for long hours of motivating discussion, constructive feedback, and thorough review of this dissertation. In addition, I would like to extend my thanks to all my committee members: Dr. Kurt Maly, Dr. Mohammad Zubair, Dr. Ravi Mukkamala, and Dr. James Leathrum for their fruitful feedback concerning this dissertation.

Special thanks are due to my husband Ayman for his encouragement, motivating support, patience, and sincere opinions.The biggest thank you is due to my daughter Rana and my son Mohamed for being super good during my extended hours of work.

My utmost thanks for my Mother and Father for their unconditional love and support. Finally, I would like to express my deepest gratitude to my sisters, Maha, Nagia, and Thanaa and my friend Samya for being there whenever I needed.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

CHAPTER I

INTRODUCTION

Many emerging technologies, such as web technology and low cost high performance desktops have provided both the inspiration and the motivation of a wide range of applications, for which securing data transmission is an important requirement. Although secure point-to-point communications have been predominant so far, the need for secure group communication is increasingly evident in a wide variety of government, commercial, and Internet communities. Secure group communication is becoming the basis for a growing number of applications such as war gaming, law enforcement, disaster relief, stock quotes distribution, news feeds, software updates, live multi-party conferencing, shared work space, distributed interactive simulation, Internet video transmission, and on-line video games. Some of these applications engage in one-to-many communication while others involve many-to-many communication. Different group applications and different application contexts will need different security services.

In secure group communication, just as in point-to-point communication, the privacy, integrity, availability, and authenticity of a group service must be protected. However, a group security concerns are considerably more involved than those regarding point-to-point communication. In secure group communication, dealing with common issues of message authentication and confidentiality becomes much more complex. In addition, other concerns arise, such as access control, and dynamic group membership [4], [31]. Secure group communication is usually categorized by the Internet Engineering Task Force (IETF) as secure multicast communication. The IP multicast model [18] uses the notion of a group of members associated with a given group address. A sender simply sends a message to this group address and the network replicates the message and forwards the copies to group members located throughout the network.

The journal model for this dissertation is the IEEE/ACM Transactions on Networking.

Secure group communication has three major core areas: secure group policy, secure group data transfer, and secure group key management [29]. A secure group policy provides the definition, implementation and maintenance of policies governing the various mechanisms of group security, such as key dissemination, access control, updating (rekeying) of the group shared keys, and the actions taken when certain keys are compromised. Secure group data transfer is concerned with providing secure group traffic techniques such as the methods used to ascertain the authenticity of a piece of data and the methods used to establish data confidentiality. Secure group key management is concerned with the methods of issuing and distributing group keys and the management of those keys over period of time, e.g. updating (rekeying) the existing group key(s) under certain conditions following the prescribed policies.

In this dissertation, we present our view and efforts in developing software framework for providing secure group key management that is efficient, scalable, reliable, and independent of the application, the security mechanism, and the communication protocol.

1.1 Overview

Before the widespread use of the computer, *information security* was provided by physical and administrative means. With the introduction of the computer, the need for automated tools for protecting files and information stored on the computer became evident. The generic name for such tools is *computer security*. The introduction of distributed systems and the use of networks and communications facilitate carrying data between computers. *Network security* measures are needed to protect data during their transmission. There are no clear boundaries between these three forms of security.

By viewing the function of the computer system as providing information, there is a flow of information from a source to a destination, and the attacks could be classified as passive attacks, or active attacks. Passive attacks are usually called eavesdropping, monitoring, or interception, and its goal is to obtain information that is being transmitted. The attacks could be the release of message content, or traffic analysis. They are very difficult to detect. Thus, network security emphasis is on preventing them rather than detecting their occurrence. Active attacks involve modification of the data stream or the

creation of a false one. There are four active attack categories. *Masquerade* (impersonating) in which one entity pretends to be a different entity; *Replay* in which passive capture of a data units is followed by subsequent retransmission to produce an unauthorized effect; *Modification* of the message (alteration, delay, or reorder); *Denial of service* that prevents the normal use of a service. It is difficult to absolutely prevent active attacks. The goal of a network security system is to detect them and possibly recover from any resulting disruption or delays.

The following are the defined network security services:

- *Authentication* that assures the recipient that the message is from the source that it claims to be.
- *Access control* to limit and control the access of information to authorized users.
- *Confidentiality* (privacy) is the protection of transmitted data from passive attacks, so it is accessible only for authorized users.
- *Integrity* that assures the recipient that any modification of a transmitted message is done only by authorized users.
- *Non-repudiation* is to prevent neither the sender nor the receiver from denying a transmitted message.
- *Anonymity* when the identity of the sender of a message is secret.
- *Service availability* is the detection and recovery from attacks that result in the loss or reduction in availability of elements of a distributed system.

Many emerging technologies, such as low-cost high performance desktop, video and audio processing equipment, and high-speed transmission and switching will enable real-time information exchange among *group* of participants. A new generation of distributed group applications will take advantage of these technologies and provide many network-based services. Many of these applications will require security provisions for session management and information transmission. War gaming, stock quotes distribution, news feeds, distributed interactive simulation, live multi-party conferencing, and on-line video games are just some of these group applications that require multiparty exchange of data, voice, and video among a large number of simulated and real participants. Group communication has many varying characteristics such as group size, member characteristics (i.e., computing power and available bandwidth), membership dynamics,

expected group lifetime, number of senders, and volume and type of traffic. A group security service should address the different requirements of different group characteristics in addition to being scalable, reliable, and independent of security objective, technology, and communication protocol [10].

Cryptography techniques can be used to provide authentication, confidentiality, sender non-repudiation, and message integrity. The use of cryptography necessitates the distribution of shared group key(s). The nature of group communication presents a challenge when trying to provide secure group key management. Secure group key management addresses issues such as how to generate a group key, how to securely distribute the group key, how to revoke membership of leaving members, i.e., preventing leaving members from access to future group communication (*perfect forward secrecy*), how to prevent joining members from access to past group communication (*perfect backward secrecy*), and how to periodically refresh the group key [65].

Extending point-to-point protocols for distributing a group key is not scalable. For example, setting up a group of symmetric keys with the assistance of a centralized group key manager (GKM), where the GKM is used for authenticating and distributing the group key to group members. Such protocol will involve encrypting the relevant message n times, for a group of n members, which is not scalable. The primary design goal of a secure group key management is to be scalable and make efficient use of processing, bandwidth, and storage requirements for a GKM and a group member.

Secure group key management is a relatively recent field of research that is related to two classical problems namely secure broadcast and contributory group key agreement. In secure broadcast a sender wishes to broadcast a secret (group key) by a single transmission (that is received simultaneously by many receivers) to some subset of his receivers. Proposed solutions that are based on the mathematical Chinese Remainder Theorem [15] or polynomial interpolation [7] are either of theoretical interest where their security is not studied, or not efficient for large group sizes. Contributory group key agreement is usually based on a generalization of Diffie-Hellman (DH) key agreement protocol to a group [37], [61]. DH allows two individuals to agree on a shared key, even though they can only exchange messages in public. Group DH protocols are contributory key agreement protocols that generally require sending several messages and the group

key is generated and distributed after several rounds. These protocols are suitable for small size peer groups, but not suitable for one-to-many type of applications, or applications with heterogeneous environments where group members' computation power and bandwidth varies. Since the rekeying delay is very large, group DH protocols are not suitable for highly dynamic or large groups.

Secure group communication is usually categorized by the Internet Engineering Task Force (IETF) as secure IP multicast communication. Hardjono et al [29] propose a reference framework and problem areas for secure IP multicast protocol suites and define the functional building blocks for such protocol suites. Three problem areas are defined, namely, multicast data handling, keying material management, and multicast security policies. Multicast data handling covers problems concerning the security-related treatments of multicast data by the sender and the receiver that includes multicast data encryption, group authentication, source authentication, and data integrity. Management of the keying material (i.e., cryptographic key belonging to a group) is concerned with the secure distribution and refreshment of keying material along with their associated state and parameters. Multicast security policies cover aspects of policy in context of multicast security that include policy creation, high-level policy translation, and policy representation. Secure IP multicast provides security throughout the network layer and routing protocols, and might require trust in intermediate routers.

Iolus [49] is the first system to address the group key management scalability problem by noticing that the security association must be dynamic in case of group communication, changing as group membership varies. Iolus's approach to provide scalability introduces the notion of a secure distribution tree that is composed of a number of smaller secure multicast subgroups arranged in a hierarchy to create a single virtual secure group. Scalability is achieved by having each subgroup relatively independent. Each subgroup has its own subgroup keying and there is no global group key. Several other proposals adopt a distributed group key management to solve the group key management scalability problem, e.g. [21], [64].

Wong et al. [67] present a different approach to improve the scalability of group key distribution. Instead of a hierarchy of group security agents, they employ a hierarchy of keys namely Logical Key Hierarchy (LKH). It is assumed that there exists a trusted and

secure GKM responsible for group access control and key management using a LKH. The LKH keys are distributed to group members while attempting to localize (as much as possible) the effects of a rekeying event. The LKH approach gained a lot of interest, and several other techniques have been built on top of it to improve the rekeying computation, communication, or storage requirements [20].

1.2 Motivation and Objective

Secure group communication is becoming the basis of a wide variety of applications in many government, commercial, and Internet communities. Secure group key management is concerned with securely issuing and distributing a shared group key to group members. In order to ensure perfect secrecy, the shared group key needs to be changed and redistributed (rekeyed) as group members join or leave the group. Rekeying when a member joins (leaves) the group, used to provide perfect backward (forward) secrecy, prevents the member from accessing previous (future) group communication. Usually, there exists a dedicated group manager (GKM) responsible for such group key (*GK*) management issues. In terms of scalability, group rekeying presents a challenging problem when trying to revoke a membership such that a leaving group member would not have future access to the group communication.

A very fast rekeying is crucial to the performance of an application that has large group size, experiences frequent joins and leaves, or the GKM is hosted by a group member because of the required computational effort. For example, a distributed interactive military simulation that requires the exchange of communication between groups of tens of thousands of participants. A second example, is a content-based publish subscribe system such as stock quotes distribution, and secure broadcasting of audio and video, where a central server experiences frequent join and leave requests. A third example is a secure group of few hundred participants, where the GKM is hosted by a group member such as in peer-to-peer networks, mobile ad-hoc networks, or grid computing environments.

The simplest protocol is for the GKM to maintain the *GK* and a shared key with every group member. Rekeying for a new member joining the group requires the GKM to change the *GK*, encrypt it with its previous version and send it to old group members, and

encrypt it with the new member shared key and send it to him. Rekeying to revoke a membership (i.e., leaving member) requires the GKM to change the *GK*, encrypt it individually with each shared key and send it to the corresponding member. When revoking a membership, the GKM can no longer use the previous *GK* that is known to the leaving (evicted) member. This protocol requires two encryptions to provide perfect backward secrecy, but requires n encryptions to provide perfect forward secrecy for a group of n members. This protocol is not scalable since it scales linearly with the group size.

The logical key hierarchy (LKH) [67] provides a scalable approach and requires the GKM to maintain a hierarchy (tree) of keys of degree d. The root of the hierarchy is *GK*, the leaf nodes are the members shared keys, and the other keys (known as key-encrypting-keys KEKs) are used to provide scalable rekeying. Every group member holds the keys that fall on the path from his shared key leaf node to the root. If a new member joins the group, his shared key is inserted in the hierarchy and all the keys he will be holding are changed and redistributed. If a group member leaves the group, his shared key is deleted from the hierarchy and all the keys he was holding are changed and redistributed.

For example, the LKH of degree $d = 3$, shown in Fig. 1, is maintained by a GKM for a group of 9 members (a keys is indexed by the members' numbers whose holding it). Rekeying after inserting K_9 (member joins) requires the GKM to change K_{7-8} to be K_{7-9} and the group key K_{1-8} to be K_{1-9}, and to perform the following 4 encryptions[1] for the new keys distribution: $\{K_{7-9}\}K_{7-8}$, $\{K_{7-9}\}K_9$, $\{K_{1-9}\}K_{1-8}$, and $\{K_{1-9}\}K_9$. While rekeying after removing K_9 (member leaves) requires the GKM to change K_{7-9} to be K_{7-8} and the group key K_{1-9} to be K_{1-8}, and to perform the following 5 encryptions for the new keys distribution: $\{K_{7-8}\}K_7$, $\{K_{7-8}\}K_8$, $\{K_{1-8}\}K_{1-3}$, $\{K_{1-8}\}K_{4-6}$, and $\{K_{1-8}\}K_{7-8}$. In general, for a group of n members and a balanced LKH of degree d, rekeying after a member joins would require GKM to perform on the average $2 \times \log_d n$ encryptions and rekeying after a member leaves would require GKM to perform on the

[1] The notation {M}K implies that the message M is encrypted with the key K.

average $d \times \log_d n$ encryptions. A group member stores $\log_d n$ keys and has to perform at most $\log_d n$ decryptions for a rekeying.

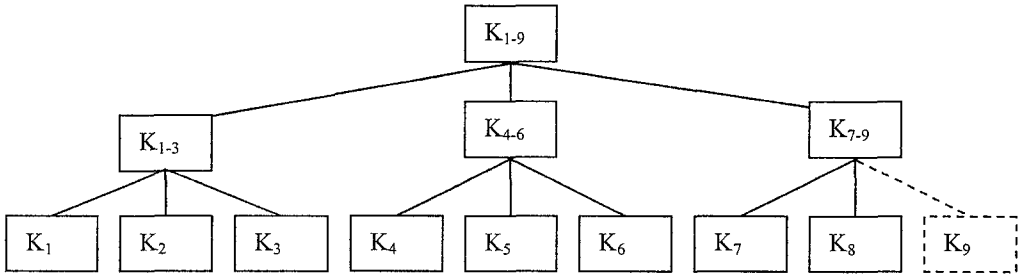


Fig. 1. A Logical Key Hierarchy of degree $d = 3$ for a group of 9 members.

Traditionally, group rekeying is performed periodically for the accumulated join and leave requests (i.e., batch of updates) during an inter-rekey period. If the GKM maintains a LKH of degree d and height h, such that $n \leq d^h$, and the batch size is R requests, a rekeying requires the GKM to regenerate $O(R \times h)$ keys and to perform $O(d \times R \times h)$ keys encryptions for the new keys distribution. The encryption-based LKH approach provided a rekeying cost that scales to the logarithm of the group size, however, the number of encryptions performed by a GKM increases with increased LKH degree, LKH height, or the batch size, and can be more than the simple approach's number of encryptions (i.e, n encryptions).

Many researchers introduced new techniques for group rekeying on top of LKH attempting to reduce compuation, communication, or storage cost for a GKM or a group members. While Chang et al. [13] achieve reduction in a GKM storage, their approaches allow members to collaborate or collude and break the system easily. The use of one-way function to reduce communication cost is suggested by Balenson et al. [2], which might increase the computation effort and the rekeying delay. The use of pseudo-random function to reduce communication-storage parameters is suggested by Canetti et al. [11],

which constraints key generation to applying pseudo random function which makes it hard to choose the session key form chosen weak keys.

The objective of our work is to provide a framework for secure group key management that outperforms the original LKH approach in terms of a rekeying computation effort for all application scenarios. The framework has to be secure, scalable, efficient, reliable, and independent of the application, the security mechanism, and the communication protocol.

The main component of the framework is the key distribution technique. The main drawback of the LKH approach is that rekeying requires the use of encryption/decryption that will delay the process. Many real-time applications require very fast rekeying so that it is not disruptive to their performance. In addition, the LKH approach has two different procedures for rekeying in case of a member joining or leaving the group. Having two un-symmetric rekeying protocols makes it more complex for batch processing, where a rekeying is performed after a sequence of requests of members joining and/or leaving the group (i.e., batch of updates). As previously noted, the other approaches built on top of LKH either increase the computation effort or are more vulnerable than the original LKH approach. Our objective is to introduce a key distribution technique, on top of LKH, that requires much less computation effort and symmetric in both rekeying cases. In addition, the new technique should be as secure as the original LKH and does not introduce any significant increase in the communication or the storage requirements.

While the use of LKH is becoming standard practice as a group key management technique, and many researchers assume a balanced LKH (i.e., all leaf nodes are at the same level) for their cost estimates. To the best of our knowledge, no LKH maintenance algorithms have been proposed for any LKH degree that keeps it balanced all the time. Our objective is to provide LKH insertion and deletion algorithms and the associated rekeying protocol(s) that maintain the LKH of any degree balanced at all times.

Since the group rekeying latency is large, it is not practical to apply such process after each member joins or leaves the group. Instead, a *batch rekeying process* should be applied for a sequence of members joining and/or leaving the group. The rekeying process could be triggered periodically or when a certain condition is satisfied such as the

batch size exceeding a certain limit. Our objective is to extend the developed balanced LKH algorithms and protocols for individual updates to a batch of updates.

A central key manager becomes a central point of both congestion and failure. For a scalable reliable framework, our design has to provide both central and distributed secure group key management mechanisms. In addition, it is essential to incorporate a *recovery mechanism* for a key manager and a group member after short times of failure. The mobile computing paradigm is an example where frequent short disconnection times may occur, due to handoffs.

1.3 Contributions

First, we presented a new generic software model for providing secure group communication. The model identifies five main components along with main functionality and interactions. The identified components are authentication manager, group key manager, rekey manager and the corresponding rekey client, group rekey channel, and cryptographic utility manager [25]. Then, we extended Java™ Security with an application-programming interface (API) that can be used to provide group key manager, rekey manager, and rekey client functionality as suggested by our model. Our secure group key management framework is independent of the application, the security mechanism, and the communication protocol. The group key management framework requires addressing the following issue: group key distribution, rekey protocol, batch rekeying, distributed group key management, and group key manager recovery. We briefly present our approach to resolve the aforementioned issues highlighting our contributions.

A Key Distribution Technique

We focused on the rekey manager/rekey client protocol that uses a Logical Key Hierarchy (LKH) in order to provide scalable group key distribution. Similar to the original LKH, we assume the rekey manager (re)-generates any key independent of all other keys including its old version. Then, the rekey manager sends a rekey message to all group members. The rekey message is received by the rekey client component, and contains a rekey packet for every new key. The rekey client chooses which rekey packets

to process and update his set of keys according to other guiding message information (e.g., the location of the new keys).

The original LKH approach encrypts a new key with either other key or its previous version. Instead, we proposed a novel XOR-based key distribution technique namely XORBP. The proposed approach uses an XOR operation between keys to reduce the computation effort, and uses random byte patterns (BP) to distribute the key material in a fixed size rekey packet to protect against insider attacks [24]. Compared to the encryption approach, our technique provides symmetric rekey protocols in both cases of group member joining and leaving. In addition, our experiments have shown that XORBP can achieve more than 90% reduction in the rekey message construction time, compared to the encryption-based key distribution technique, for the same LKH degree. For example, consider a news broadcast GKM that supports a group of size n = 60,000, where up to 100 listeners could join in a sec, a listener stays tuned for few minutes, and a one block encryption consumes 1 msec. Using the original encryption-based LKH, where $d = 4$, $h = 8$, and R = 1000, the rekey manager's rekey message construction time requires 32 sec. Using the suggested XORBP LKH approach, a rekey message construction time is reduced to 3.2 sec.

On the other hand, XORBP increases LKH storage, member storage, and the rekey communication cost (message size). Due to the un-symmetry of the encryption protocol, increasing the LKH degree with such protocol reduces the join rekey computation cost while increases the leave rekey computation cost. Using the symmetric XORBP key distribution technique and increasing LKH degree would not have the same constraint. The symmetry of XORBP protocol allows the use of a larger degree LKH, which reduces LKH storage, member storage, and rekey communication cost compared to a smaller degree LKH.

LKH Maintenance and Rekey Protocols

The research literature lacks practical LKH maintenance algorithms as well as algorithms for keeping it balanced. Keeping a LKH balanced is crucial to the performance of group rekeying especially for highly dynamic groups. We proposed two novel protocols for establishing and maintaining a LKH (by a rekey manager) with any

degree as key nodes are inserted and deleted while group members join and leave the group. In addition, we detailed the rekey message format and construction in different LKH insertion and deletion scenarios as well as the different rekey client updates to maintain a group member set of keys. One protocol adopts a balanced LKH while the other adopts an unbalanced LKH that is developed for comparison.

Our protocols are based on the rekey manager assigning a unique member identification (individual ID) that will be used as a group member sort and search value. Individual identifications are sent in the rekey message to guide its processing, so they better be randomly generated (not from any names, IP address, or any other true individual identification) to prevent the possibility of traffic analysis. In our protocols, the LKH plays a dual role as a key tree and an easily searchable data structure (using an individual ID) for the member individual material (name, IP address, key, ... etc).

Our first protocol maintains a LKH as a search tree (S-LKH) using the individual IDs. We adapt the search tree algorithms to accommodate the constraint that group individual materials are entries in the leaf nodes, while the internal nodes contain key-encrypting-keys (KEKs). Our second protocol maintains a LKH as a balanced B^+ search tree (B^+-LKH) that has the same structure as S-LKH but guarantees that the LKH is balanced after every node insertion or deletion. B^+ search trees have an extra constraint that all allocated nodes have to be at least half full to reduce the required tree allocated memory (storage). On the other hand, B^+-LKH maintenance introduces complexity and extra overhead in the rekey process.

We have performed empirical experiments to compare the rekey performance of S-LKH versus B^+-LKH for different group sizes and LKH degrees. For individual rekeying (i.e., rekey after every join or leave request) the use of B^+-LKH results in an increase in the average number of rekey packets and the average number of encrypted keys compared to S-LKH. On the other hand, a B^+-LKH has smaller height, and introduces a decrease in the maximum number of encrypted keys. The maximum number of encrypted keys identifies the minimum period that has to be elapsed between two rekeyings. Furthermore, a B^+-LKH requires much less allocated nodes (i.e., storage) compared to S-LKH. The reduction of the number of allocated nodes using B^+-LKH reaches 50% of the number of nodes for the same degree S-LKH for a highly dynamic group. A complete

LKH of degree d and height h contains $(d^h - 1)/(d-1)$ nodes ($\sum_{i=0}^{h-1} d^i$), and can fit a group of size $n \leq d^h$. A leaf node contains d individual keys, while an internal node contains d key-encrypting-keys. For the aforementioned example, a GKM for 60,000 group members and a LKH of degree d = 4, the B^+-LKH number of allocated nodes is estimated to be 42,000, (form our experiment, when d = 4, the B^+-LKH number of allocated nodes = $0.7 \times n$). On the other hand, if a S-LKH is used, the LKH number of allocated nodes could increase to more than 84,000 for a highly dynamic group.

Batch Rekeying

As previously mentioned, individual rekeying is not practical. For example, if the inter-arrival time of group members at the start of a session is very small, a new group key might be issued (by the rekey manager) before the previous key version has reached (or has been used by) the group members. A simple solution is periodic rekeying that suggests rekeying after a fixed period of time that is large enough to avoid the above problem. Periodic rekeying will require a rekeying for a batch of updates (i.e, accumulated join and leave requests during this period). Periodic rekeying doesn't take into account the batch size or the request delay. We have extended our protocols to support batch processing.

First, we introduced a generalized rekey policy based on three main parameters that determine the triggering condition for the rekeying process. The three parameters are batch size, maximum request delay (i.e., time between receiving the request and the start of rekeying), and the minimum inter-rekey period (i.e., minimum period that has to be elapsed between two consecutive rekeyings). The application has the flexibility of using all or some of the rekey policy parameters as a deciding factor for triggering the rekey process. The application type determines what blend of parameters is taken into consideration. We detailed the designed rekey policy definition and presented a software object design for secure group key management.

Next, we extended S-LKH and B^+-LKH rekey protocols for a batch of updates. For individual rekeying we concluded that the use of B^+-LKH introduces major LKH storage savings and slightly increases the rekey cost. Our experiments for batch of updates show

that using B^+-LKH with large batch size and/or high dynamic groups substantially reduces the rekey cost by more than 50% when compared to S-LKH. For example, assuming a balanced LKH (B^+-LKH) the number of regenerated keys in the above example is estimated to be 8,000 keys, while if an unbalanced S-LKH is used, the number of regenerated keys can increase to more than 16,000 keys (and therefore doubles the LKH estimated rekeying times). In addition, our experiments demonstrate that B^+-LKH performance is stable (bounded) for highly dynamic groups while S-LKH performance deteriorates as the group dynamics increase. Such S-LKH instability is due to the fact that the minimum number of children of a node is one while B^+-LKH nodes need to be at least half full.

Distributed Group Key Management

To extend the scalability and the reliability of our model, we introduced four cooperating protocols of distributed group key management between peer rekey agents. In a group of peer rekey agents, every agent manages a subset of the group members and participates equally in generating and distributing the group key (known to all group members). We show that the protocol with the minimal overhead is that one rekey agent at a time generates and distributes the group key to all members. We provide the design details of the LKH maintained at every agent for the different cooperation scenarios.

If any rekey agent is required to update all group members of a new group key, a naïve approach is that every agent maintains (replicates) the group LKH. Instead, we proposed the creation of agents' LKH (A-LKH) that reduces the replicated LKH size, and the number of maintained keys at a group member. Moreover, we discussed two different approaches for maintaining A-LKH namely dynamic A-LKH and static A-LKH. The first approach, dynamic A-LKH, allows a flexible agent join and leave but has a drawback of (sometimes) updating (some) group members when a rekey agent joins or leaves the agents' group. While, in the second approach, static A-LKH, the maximum number of rekey agents has to be known before starting the session and updating A-LKH is transparent to all group members.

Group Key Manager Recovery

Finally, we suggested a recovery protocol of a group key manager (agent) after a short time of failure. Although the group key manager state (e.g., LKH) could be recovered by collecting the state stored at all group members (and rekey agents), we introduced the use of a log file to facilitate such recovery in case of member failures or inconsistency. The logging system avoids writing any key or revealing any random number generator information. The log file is used to recover the last rekey policy, the rekey scheduler state, and the shape of LKH (without keys). The group members participate in the recovery phase by sending at least one encrypted recovery message to their rekey manager. The recovery message sent by a group member contains his set of maintained keys. Noticing that many LKH keys are stored by more than one group member (e.g., the group key is maintained by all group members), we introduced a key selection technique for group members to reduce the number of sent keys in a recovery message while allowing the group key manager to retrieve all LKH keys. The proposed logging and recovery mechanism is secure and easy to implement. The recovery of a group member after short time of failure can be treated as the member leaving the group then joining later. If no rekeying is initiated between the leave and join requests, the group member state is refreshed (i.e., sending him the same set of keys he was holding). In this case, refreshing a group member optimizes the rekey process by reducing the number of changed keys. Such refreshing requires the group member to provide his individual ID and key.

In summary our contributions can be summarized as follows:

1) A generic software model for secure group key management that identifies the main components and their functionalities and interaction. Extending Java™ security with an API that can be used to provide the group key manager, the rekey manager, and the rekey client functionality suggested in our model.
2) A simple key distribution technique XORBP that can be used with the Logical Key Hierarchy (LKH) approach for group key management. Our experiments show that, compared to the original encryption technique for key distribution, XORBP has

symmetric rekey procedures for join and leave, and can achieve more than 90% reduction in the rekey message construction time [24].

3) Two LKH protocols for group individual rekeying (i.e., after each join or leave request) that details the LKH insertion and deletion algorithms, and the rekey message format and construction performed by a group rekey manager. In addition, the protocols detail the rekey client updates performed by the component that receives the rekey message at a group member. Our first protocol adopts an unbalanced LKH (S-LKH) while the other adopts a balanced LKH (B^+-LKH). Our experiments show that B^+-LKH reduces the required LKH storage while slightly increases the individual rekeying cost compared to S-LKH. The reduction of the number of allocated nodes using B^+-LKH reaches 50% of the same degree S-LKH for a highly dynamic group [25].
4) For batch processing (sequence of join and/or leave requests): first, we formalized a definition of a flexible rekey policy that has three main parameters: batch size, maximum request delay, and minimum inter-rekey period. Then, we provided a simplified view of the software objects used to provide secure group key management. Next, we extended the above two protocols (S-LKH and B^+-LKH) to support batch rekeying. Our experiments for batch of updates show that using a balanced LKH (B^+-LKH) with large batch size and/or high dynamic group substantially reduces the rekey computation and communication cost by more than 50% when compared to an unbalanced LKH (S-LKH). In addition, our experiments show that B^+-LKH performance is stable (bounded) for highly dynamic groups while S-LKH performance deteriorates as the group dynamics increases.
5) We introduced four cooperating protocols of distributed group key management between a group of peer rekey agents, and detailed the maintained LKH and the group rekey overhead for each model. We introduced the use of agents' LKH (A-LKH) to reduce the size of the replicated LKH maintained at each agent over a naïve approach (used in two of the above protocols). In addition, we proposed two techniques for A-LKH maintenance, one allows a transparent agent join or leave to group members and the other is not transparent (group members might be affected/notified).

6) Finally, we proposed a logging and recovery mechanism for the group key manager and the rekey manager. The proposed technique is secure and easy to implement. Group members participate in the recovery of their group key manager by sending one recovery message (in most cases). A key selection technique is proposed for a group member to reduce the size and overhead of the recovery message. In addition, we discussed the recovery of a group member after a short time of failure.

1.4 Outline

The rest of this dissertation is organized as follows. Chapter II presents related work to secure group communication and secure group key management. Chapter III introduces the software model for secure group key management and presents the new key distribution technique XORBP. In addition, the experimental results for comparing XORBP key distribution technique with the encryption-based technique are presented. In Chapter IV, we detail the designed rekey protocols. The first protocol adopts an unbalanced LKH (S-LKH) while the second protocol adopts a balanced LKH (B^+-LKH). We present the rekey message format, the LKH data structure, the LKH maintenance algorithms along with the rekey message construction for both protocols, and the rekey client update procedures for B^+-LKH protocol. Moreover, the experimental results for comparing the two protocols for individual rekeying are presented. In chapter V, we introduce a rekey policy definition and implementation, and highlight the extension of B^+-LKH rekey protocol for batch processing. Furthermore, the experimental results for comparing S-LKH and B^+-LKH protocols for batch rekeying are presented. Chapter VI presents the extended model for distributed group key management. In addition, we discuss the recovery of a group member after a short time of failure as well as the proposed recovery protocol for the group key manager. Finally, chapter VII concludes this dissertation summarizing our contributions and presenting ideas for future extensions.

CHAPTER II

RELATED WORK

In chapter I, we identified the main requirements and issues for providing secure group key management. In a general model, there is a group manager responsible for generating and distributing a group key to all group members. The group manager is also responsible for changing and redistributing (i.e., rekeying) the group key when it deems necessary. The group key has to be changed to prevent new (old) group members from accessing previous (future) group communication. Secure group key management has to be scalable and reliable. A major scalability problem occurs when a rekeying is performed to revoke a group membership. A naïve solution allows the group manager to perform n encryptions to distribute a new group key to a group of n members.

In this chapter, we present relevant related work to the secure group key management problem. First, we present two (classical) problems similar to group key distribution. Section 2.1 presents the secure broadcasting problem, while section 2.2 presents the contributory group key agreement problem. As previously noted, secure group communication is categorized by IETF as secure multicast. Section 2.3 summarizes the IEFT group key management standard. In addition, we summarize the recent research work for secure group key management. The approaches for solving the scalability problem, identified above, can be categorized as physical distributed management (section 2.4) and the logical key hierarchy approach (section 2.5). Moreover, section 2.6 summarizes several related topics to secure group communication such as multicast IPsec, group policy, group access control, group data-origin authentication, and rekey transport protocols. Finally, section 2.7 summarizes this chapter.

2.1 Secure Broadcasting

Secure broadcast is motivated by the main property of a broadcast channel, that is a single transmission from a source station can be received simultaneously by many destination stations. Secure broadcast is defined as the sender wishing to broadcast a

secret to some subset of his receivers. Meanwhile, the sender does not perform a separate encryption either of the secret or of a single key with which to protect the secret, for each of the intended recipients.

Secure lock [15] proposes the locking concept and a secure lock implementation based on the Chinese remainder Theorem. The proposed scheme is efficient only when the number of users in a group is small, since the time to compute the lock and the length of the lock (hence the transmission) is proportional to the number of users.

Berkowis [7] provided a generalized model for a predefined scheme for secure broadcasting that uses polynomial interpolation for secret sharing. The general model assumes each receiver has a unique pseudo-share (secret) with the sender. The sender broadcast a set of shares, while each subscribed receiver adds his pseudo-share, as a possible share, to the received shares. If that pseudo-share is an actual share he recovers the secret, and if it is not he doesn't recover the secret. Some examination of the security of his scheme is still necessary. Gong [26] tries to add authentication, integrity check, and freshness assurance to the message of a modified version of the polynomial method.

Fait and Naor [22] introduce theoretical measures for the qualitative and quantitative assessment of the encryption schemes designed for broadcast transmissions. The work considers a scenario where there is a center and a set of users. The center provides the users with pre-arranged keys when they join the system. At some point the center wishes to broadcast a message to a dynamically changing privileged subset of the users. The obvious solution is to give every user its own key and transmit an individually encrypted message to every member of the privileged class. This requires a very lengthy transmission. The other simple solution is to provide every possible subset of users with a key. This requires every user to store a huge number of keys. The authors provide solutions, which are efficient in the two measures, transmission length, and storage at the user's end. In addition, the schemes should be computationally efficient. The security parameter was defined to be the length of the key. Another defined parameter is the number of users that have to collude so as to break the scheme. For a given parameter k, a k-resilient scheme should be resilient to any subset of k users that collude and any disjoint subset of any size of privileged users.

2.2 Contributory Group Key Agreement

There are two types of group key agreement, centralized or contributory. In centralized techniques, the entire key generation is performed by a single entity (which actually translates into key distribution, not key agreement). On the other hand, in a contributory key agreement, each group member makes an independent contribution to the group key. The contributory key agreement model is usually based on a generalization of Diffie-Hellman (DH) key agreement protocol to a group [37], [61]. DH is a public-key system that allows two individuals to agree on a shared key, even though they can only exchange messages in public. Group DH generally require sending several messages, exchanges, and the key is generated and distributed after several rounds. These protocols are suitable for small size peer groups. While they are not suitable for one-to-many (one sender and many receivers) type of applications, applications with a heterogeneous environment where member computation power and bandwidth varies. In addition, since the rekey latency (delay) is very large, they are not suitable for highly dynamic and/or large groups where frequent re-keying is necessary.

2.3 Standardized (IETF) Group Key Management

The Group Key Management Protocol (GKMP) [35], [36] is an application level protocol, independent of the underlying communication protocol. The creation and distribution of the group key require assignment of roles. The two primary roles are those of key distributor and member. The protocol identifies what functions the individual hosts perform in the protocol. The controller initiates the creation of the key, forms the key distribution messages, and collects acknowledgement of key receipt from the receiver. The member waits for a distribution message, decrypt, validate, and acknowledges the receipt of the new key.

Baugher et al. [6] present a group key management architecture for multicast security that is based upon the group controller model with a single group owner as the root-of-trust. The group owner designates a group controller for member registration and rekey. The framework and the architecture allow for a modular and flexible design of group key

management protocols for variety different settings that are specialized to application needs.

Hardjono et al [29] propose a reference framework and problem areas for secure IP multicast protocol suites and define a breakdown to functional building blocks for such protocol suites. They define three problem areas: multicast data handling, management of the keying material, and multicast security policies. Group key management building blocks following the reference framework are described in [28], [33].

2.4 Distributed Group Key Management

Ensemble [57] is a group communication system built at Cornell University, and is a descendant from an earlier system named Hours, that is descendant from the Isis system. The system allows processes to create process groups in which scalable reliable FIFO-ordered multicast and point-to-point communication are supported. A process group coherently binds together many processes into one entity. Processes may dynamically join and leave a group. Ensemble is a user-level library linked to an application, and is divided into many layers each implementing a simple protocol. Stacking together these layers, the user may customize the system to suite its needs. All members in a group must have the same stack to communicate. Ensemble group communication has inherently limited scalability, and scales to 100 members. Rodeh et al. [57] describe the security protocols and infrastructure of Ensemble. A completely distributed and fault-tolerant algorithm for the management of Ensemble group keys (arranged as LKH) is described in [56].

Iolus [49] is a scalable, general-purpose framework that can be used for either secure multicasting or multicast key management. Iolus discards the idea of large flat secure multicast group and replaces it with the notion of a secure distribution tree that is composed of multiple smaller secure multicast subgroups arranged in a hierarchy. Together these subgroups form a single virtual secure multicast group. The glue that holds the subgroups together consists of the Group Security Agents (GSAs) that manage each subgroup. The GSAs cooperate to invisibly deliver all multicast data securely to

each of the subgroups, thereby creating a single secure multicast image for the senders and receivers.

Versakey [64] is a middleware framework for secure multicasting. The framework presents three closely related schemes for key distribution and management, ranging from tightly centralized to completely distributed. The framework also provides a set of efficient transitions from one scheme to another. All approaches organize the space of keys that will eventually be assigned to group members in a unique way, without actually generating the keys before they are needed.

DISEC [21] proposes a distributed key management scheme for many-to-many secure group communication. The framework uses one-way function trees for key distribution and management. DISEC proposes a localized ID assignment scheme thereby eliminating the need for a centralized group controller. Each member generates its own key thereby contributing a secret towards the computation of the root key. In addition, DISEC doesn't have a single point of control, attack, or failure.

2.5 Logical Key Hierarchy

Wong et al. [67] present a novel solution to the scalability problem of group key management. They introduce key graphs and its special type, a key tree, to specify secure groups. It is assumed that a trusted and secure key server is responsible for group access control and key management, and the key server uses key graphs for group key management. A key graph is a directed acyclic graph with two types of nodes, u-nodes representing members and k-nodes representing keys. A member is given key k if and only if there is a directed path from u-node u to k-node k in the graph. In addition, they present three rekeying strategies, user oriented, key oriented, and group oriented join/leave protocols based on these strategies. The strategies are scalable to large groups with frequent joins and leaves. In particular, the average server processing time per join/leave increases linearly with the logarithm of group size. The key tree is widely used and known as a logical key hierarchy (LKH).

Representing a binary LKH as a one-way function trees (OFTs) is introduced in [2]. In comparison with LKH, OFT algorithm reduces half the number of bits broadcast by

the manager per add or evict operation. The OFT has the option of member contributions to the entropy of the common communication key. On the other hand, OFT raises some interesting questions about the security of function iterates, and that of bottom-up one-way function trees.

Key management using a Boolean function minimization technique, introduced in [13], is similar to the LKH scheme in the sense that it uses smart distribution of keys to achieve good scaling. However, instead of using a fixed hierarchy of keys, they dynamically generate the most suitable key hierarchy by composing different keys. The paper focuses explicitly on the problem of cumulative member removal and proposes a scheme that can be used to find the minimum number of messages required to distribute the new keys to the remaining group members. An advantage of their scheme is that the controller has to maintain only $O(\log_2 n)$ keys as opposed to $O(n)$, where n is the number of members in the group. Due to the minimal number of auxiliary keys that this key management maintains, it may be susceptible to collusion attacks. In a collusion attack, a set of members previously removed from the group collude and by combining their sets of keys may be able to obtain the current valid set of keys, thereby being able to continue unauthorized receipt of group communication.

Loptsiech et al. [46] describes a key management mechanism for group communication sessions that is based on the "Subset-Difference" algorithm. The Subset-Difference algorithm is especially suitable for stateless receivers. Its main advantage over LKH is that it requires to transmit only $2 \times r$ keys instead of $2 \times r \times \log_2 n$ keys in order to revoke r users from a set of n users, regardless of the coalition size, while maintaining a single decryption at the user's end. In return, it requires every receiver to store $\log_2(2 \times n)$ keys instead of $\log_2 n$ keys. The receiver needs to employ 1 decryption for every rekeying event plus $\log_2 n$ applications of a pseudo-random generator. Chen and Dondeti [14] study the advantage and applicability of statefull and stateless rekeying algorithms to different applications. An analytically comparison is presented of the storage cost and the rekeying cost of LKH and the Subset-Difference revocation algorithm in immediate and batch rekeying scenarios.

Canetti et al. [11] present a rekeying protocol for wide range of efficiency requirement with respect to several parameters. An upper bound is deduced in the tradeoff between storage and communication parameters In addition, lower bounds are presented on the tradeoff between communication and user storage. Moreover, the proposed scheme is shown to be almost optimal with respect to these lower bounds. The security of their scheme can be reduced to the strength or the security of the pseudo-random function used in the computation. Repeated applications of a pseudo-random function, to the input will make it difficult (for the group controller) to guarantee that the root key is not from a weak key space.

Another improved LKH algorithm, LKH+2, is proposed in [55], where a group manager can use keys already in the tree to drive new keys. LKH+2 achieves $K \times \log_2 n$ message size for leave operations, where K is the size of a key.

Selck et al. [58] present a modification to the LKH scheme where the new approach proposes an organization of the LKH trees with the respect to the members' compromise probabilities instead of keeping a balanced tree, in a spirit similar to data compression techniques such as Huffman and Shannon-Fano coding.

2.6 Additional Secure Group Communication Issues

In this section we present the following additional secure group communication issues: multicast IPsec, group policy, group data-origin authentication, rekey transport protocols, and secure multicast services.

2.6.1 Group/Multicast IP Security (IPsec)

IPsec [41] is designed to provide interoperable, cryptographically based security services for IPv4 and IPv6. These services are provided at the IP layer, offering protection for IP and/or upper layer protocols (e.g. TCP, UDP, ICMP, etc). These objectives are met through the use of two traffic security protocols, the Authentication Header (AH) [42] and the Encapsulating Security Payload (ESP) [43], and through the use of cryptographic key management procedures and protocols. These mechanisms are

designed to be algorithm-independent with a specified standard set of default algorithms to facilitate interoperability in the global Internet.

IPsec security services can be provided between a pair of communicating hosts, between a pair of communicating security gateways, or between a security gateway and a host. The protection offered is based on requirements defined by a Security Policy Database (SPD) established and maintained by a user or system administrator. Packets are selected for one of three processing modes based on IP and transport layer header information matched against entries in the SPD. Each packet is either afforded IPsec security services, discarded, or allowed to bypass IPsec.

Afforded IPsec packets (use of AH and/or ESP) make use of *Security Associations* (SAs). SA is a simplex connection that affords security services to the traffic carried by it. The Security Association Database (SAD) contains parameters that are associated with each active SA to specify the security services to be provided, protocols to be employed, and algorithms to be used. The Internet Security Association and Key Management Protocol (ISAKMP) [47] defines the procedures and packet formats to establish, negotiate, modify and delete security associations (SAs). Theses formats provide a consistent framework for transferring key and authentication data which is independent of the key generation technique, encryption algorithm and authentication mechanism. The Internet Key Exchange (IKE) [32] is an ISAKMP to negotiate, and provide authenticated key material for security associations in a protected manner.

Extending IPsec to support secure (multicast) groups is not standardized, however, there are several drafts try to extend IPsec to such support. Canetti et al. [9] propose an architecture for secure IP multicast that mimics the IPsec architecture, and re-uses exiting IPsec mechanisms wherever possible.

The IPsec ESP provides a set of security services that include data origin authentication, which enables an IPsec receiver to validate that a received packet originated from a peer-sender in a pair-wise SA. However, for secure IP multicast groups, ESP supports only "group authentication" and does not support data-origin authentication. Multicast ESP (MESP) [5] is an extension of the ESP transform for multicast data-origin authentication. Canetti et al. [12] propose another MESP transform

in addition to an Application MESP (AMESP) that is designed to work in the application/transport layer.

Similar to ISAKMP, the Group Secure Associate Key Management Protocol (GSAKMP) [34] defines the message passing requirements to provide mechanisms to disseminate group policy, perform access control decisions during group establishment, generate group keys, recover from the compromise of group members, delegate group security functions, and destroy the group. In GSAKMP group responsibilities are decomposed into authorized roles. Roles are defined for Group Owner, Group Controller, SubGroup Controller, and Member.

2.6.2 Group Policy

Security policy is a statement of the rules enforced by security mechanisms. Policies can be described by whom they cover and by what they cover. Group security policy can be static or it can be dynamic and tailored to the requirements of the group.

Hardjono et al. [30] define group security policy expressed in the form of policy token or policy certificate. It describes the elements that make-up an instance of group policy and explains the intended functions of each element.

The Antigone framework [48] provides an interface for the definition and implementation of a wide range of secure group policies. Policies are implemented by the composition and configuration of a defined set of mechanisms that provide the basic services needed for secure groups. Antigone provides mechanisms for providing the following functions; authentication, member join, session key and group member distribution, application messaging, failure detection, and member leave.

The Dynamic Cryptographic Context Management (DCCM) [19] provides a policy-based security for large (100,000 members), dynamically changing groups of participants. In DCCM, groups at all levels have policies. These policies are represented, negotiated, managed, and an unambiguous set of mechanisms and configuration (called a cryptographic context) is created to make particular interactions possible subject to these policies.

2.6.3 Multicast Group Access Control

Multicast communication provides one-to-many and many-to-many communication [18]. There are a number of available multicast routing protocols that provide the efficient transport mechanisms of multicast by routing packets with one group destination address to multiple recipients. A host uses the Internet Group Membership Protocol (IGMP) to notify the routing system that it should deliver packets for a particular multicast group to this host. Gong and Shacham [27] discuss threats, requirements for security, and some trade-offs between scalability and security. They outlined the fundamental security issues in building a trusted multicast facility such as protecting traffic, controlling participation, and restricting access of unauthorized users.

IGMP operates in a different portion of the network from the multicast routing protocol. IGMP operates between hosts and edge routers. Moffaert and Paridaens [50] discuss security aspects in IGMPv3. Coan et al. [16] propose an application-level secure multicast technique that addresses some of the limitations of end-to-end secure multicast. The technique has a defense against denial-of-service attacks by using a secure extension to IGMP. Ballardie [3] describes how a Core Based Tree (CBT) multicast protocol can provide for secure joining of a CBT group tree.

Gothic [39] is an architecture for providing group (receiver) access control. Gothic is composed of two systems: the group policy management system and the group member authorization system.

2.6.4 Group Data-Origin (Source) Authentication

The problem of stream authentication is solved for the case of one sender and one receiver. The sender and receiver agree on a secret key, which is used in conjunction with a message authenticating code (MAC) to ensure the authenticity of each packet. In case of multiple receivers, however, the problem becomes much harder to solve, because a symmetric approach would allow anyone holding a key (that is, any receiver) to forge packets. Alternatively, the sender can use digital signatures to sign every packet with its private key. This solution provides adequate authentication, but digital signature are prohibitively inefficient.

Wong and Lam [68] present a chaining technique for signing/verifying multiple packets using a single signing/verification operation. Gennaro and Rohatgi [23] present two solutions to the problem of authenticating digital streams. The first one is for the case of a finite stream, which is entirely know to the sender. The second case is for a potentially infinite stream, which is not known in advance to the sender.

TESLA [53] is a secure sender authentication mechanism for multicast data streams. It provides authentication of individual data packets, regardless of the packet loss rate. In addition, TESLA features low overhead for both the sender and the receiver, and does not require per-receiver state at the sender. For TESLA to be secure, the sender and the receiver are required to be loosely time synchronized. Loosely time synchronized means that the synchronization does not need to be precise, but the receiver musk now an upper bound on the dispersion (the maximum clock offset). Perrig et al. [54] propose several substantial modifications and improvements to TESLA.

2.6.5 Reliable Group Rekey Transport Protocols

Group re-keying involves two operations – key encoding and key distribution. The key-encoding phase involves generating a set of encrypted keys that have to be transmitted to the members of the group. The key distribution phase is concerned with packing these encrypted keys into packets and delivering the packets to the members of the group in a scalable, reliable, and timely manner. Although reliable multicast transport protocols such as RMP [66] can be used for reliable delivery of such packets, the reliable key delivery problem has some characteristics that can be exploited to design custom protocols that are more light-weight in nature. Possible tailored solutions to the reliable group key distribution problem are presented in [60] and [70].

2.6.6 Other Secure Multicast Service

The SecureRing [44] group communication protocols provide reliable ordered message delivery and group membership services despite faults caused by modifications to the programs of a group member following illicit access to, or capture of, a group member (called Byzantine faults).

Non-repudiation is a proof of delivery that the receiver did indeed receive data when they might deny reception. Using the Nark scheme [8], each multicast receiver can reliably prove whether any fragment of the data hasn't been delivered or wasn't delivered in time. Further, each receiver's data can be subject to an individual watermarked audit trail. This provides a deterrent against a receiver giving away or re-selling either the keys or the decrypted data.

2.7 Summary

In this chapter, we presented relevant related work to the secure group key management problem. We presented two classical problems related to the group key distribution problem: secure broadcasting and contributory group key agreement. In addition, we summarized the IETF's group key management standard. Furthermore, we presented the different approaches for distributed group management. Moreover, we summarized the logical key hierarchy (LKH) approach for scalable group key distribution, and several variations. Finally, we presented a summary of other related topics such as multicast IPsec, group policy, group access control, group data-source authentication, and rekey transport protocols.

CHAPTER III

XORBP: A NOVEL GROUP KEY DISTRIBUTION TECHNIQUE

In this chapter, we present the contributed software model for providing (central) secure group communication. The model identifies the main software components along with their functionalities and interactions. We focus on the details of the rekey manager that generates the shared group key and distributes it to all group members. A rekey (change of group key) is necessary when a member joins the group to prevent him from accessing group communication sent before he joined (such operation is denoted *join rekey*). A rekey is also necessary when a member leaves the group to prevent him from accessing further group communication (such operation is denoted *leave rekey*). We highlight the two traditional rekey management techniques namely star and logical key hierarchy (LKH). The traditional group key management systems used to encrypt a newly generated key with other key (such as the key's previous version or a group member key) before distributing it to group members. We demonstrate the drawbacks of encryption-based key distribution techniques (KDT) such as having a non-symmetric join and leave rekey costs, and being not scalable when used with star or high degree LKH key management. Moreover, we present our novel XOR-based KDT, namely XORBP. The proposed approach uses bit XOR operation between keys to reduce the computation effort, and random *byte pattern*s (denoted BPs) to distribute the key material in a fixed size rekey packet. We demonstrate that XORBP is symmetric in the join and leave rekey operations. Furthermore, we empirically study and compare the cost of the encryption-based and XORBP KDTs. Our experiments have shown that XORBP can achieve up to 87% reduction in the rekey time compared to an encryption-based KDT.

The rest of the chapter is organized as follows. Section 3.1 presents a generic software model for secure group communication. Section 3.2 discusses star and LKH rekey management techniques, and studies the properties of the traditional encryption-based KDT. Section 3.3 introduces XORBP the proposed group key distribution technique. Section 3.4 demonstrates how XORBP can be used with LKH. Section 3.5 presents scenarios and comparison of the new key distribution technique versus the

traditional approaches. Section 3.6 analyses and compares the cost estimates of XORBP versus the encryption-based KDT. Section 3.7 presents the experimental results confirming the analyzed estimates. Finally, the chapter is concluded in section 3.8.

3.1 Secure Group Key Management Components

Fig. 2 illustrates the designed model of the software components for secure group key management. The *authentication manager* is responsible for ensuring the identity of the group members defined according to the *group policy*. The authentication manager could receive a request from a group member to join the group, or could be in charge of inviting the members to join the group. Afterwards, it applies an authentication protocol (using long-term keys) to decide whether to accept or reject a member. In addition, it negotiates the session parameters, such as the protocols and implementation used, and establishes a session *individual key* with every new member. Moreover, the authentication manager could be in charge of ending a member's participation in the session, according to a defined policy, a request from the member himself, or due to detected member communication failure.

The authentication manager notifies the *group key manager* (GKM) of every member removal, and every new member addition along with that member's individual key. GKM applies a group *rekey policy*, as to when to change the *group key* (*GK*). Different policies determine whether rekeying is necessary when a member is added and/or removed, or whether it is performed periodically. In addition, the rekey policy could determine the batch size (number of added and/or removed members), or the rekey period. For example, a rekey startup policy configures the group key manager to wait a certain amount of time before starting the creation of the group key to avoid a startup implosion scenario. When a rekey is necessary, GKM asks the *rekey manager* to generate new *GK* along with the *rekey message* RM to be sent (broadcast) to all group members for such *GK* update.

In our model, we assume when a new member joins the group he receives an *initial key message* that is sent through his *private channel*. Afterwards, the rekey manager sends (broadcasts) a RM to all group members (including the new member), through a *group rekey channel* that updates *GK*. When a group member leaves (or is evicted from) the group, only one RM is sent to the remaining group members. The group rekey

channel implementation should guarantee message reliability, integrity, freshness, and source authentication. In addition, it should synchronize *GK* between all group members.

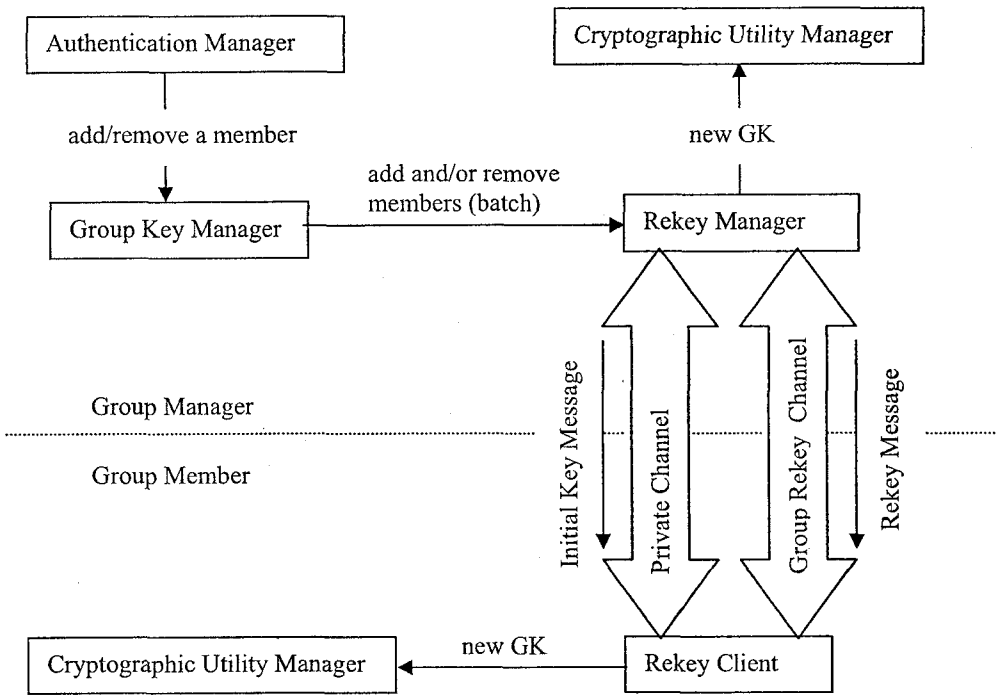


Fig. 2. Secure group key management software components.

Note that the authentication manager, the group key manager, and the rekey manager could be (all or some) software components running on the same machine, or could be software components running on different machines and communicating through network channels and protocols.

The *rekey client* is the group member component that receives RMs and maintains *GK*. Both the rekey manager and the rekey client immediately notify a *cryptographic utility manager* with a change of *GK*. The cryptographic utility manager is responsible for providing different group security services to the application. The cryptographic utility

manager has an Application Program Interface (API) that is used by the application to provide different security services. The cryptographic utility manager could derive several group keys (from the shared *GK*) for different uses, such as group encryption, message integrity, and authentication. Note that, the cryptographic utility manager is needed at the group manager if it will act as a group member.

3.2 Traditional Rekey Manager

The tradition approaches for providing central group rekey management either uses a star key management or a logical key hierarchy (LKH). Both approaches use encryption-based key distribution technique as explained next.

3.2.1 Star Rekey Manager

A *star* rekey manager for a group of n members maintains one group key *GK*, and n individual keys one for every group member. Every group member i maintains two keys, *GK* and his own individual key K_i.

If a new member $(n+1)$ joins the group, the rekey manager changes (regenerates) *GK* to be *GK'*, and sends a RM that has two encrypted[2] keys $[\{GK'\}GK, \{GK'\}K_{n+1}]$. The first encryption is the new *GK* (*GK'*) encrypted with the previous group key, and is decrypted by old group members to retrieve *GK'*. The second encryption is *GK'* encrypted with the new member individual key K_{n+1}, and is decrypted by the new member to retrieve *GK'*.

When member n leaves the group, the rekey manager sends a RM that has $(n-1)$ encryptions of the new group key $[\{GK'\}K_i, 1 \le i \le (n-1)]$. Each individual encryption is decrypted by the associated group member's key to retrieve the new group key.

Fig. 3 illustrates an example of the keys maintained by a star rekey manager for 9 members. If a new member joins the group and his individual key K_9 is to be inserted, *GK* is regenerated, and a RM that has two encrypted keys $[\{GK'\}GK, \{GK'\}K_9]$ is

---

[2] $\{M\}K$ denotes the message M is encrypted with the key K.

constructed and distributed to group members. If that member leaves the group, his individual key K_9 is deleted, a new group key GK'' is regenerated, and a RM is constructed and distributed to group members. In this case, the RM has 8 encrypted keys $[\{GK''\}K_1, \{GK''\}K_2, \{GK''\}K_3, \{GK''\}K_4, \{GK''\}K_5, \{GK''\}K_6, \{GK''\}K_7, \{GK''\}K_8]$.

We can conclude this technique does not provide a scalable RM construction cost since the cost (time and size) when a member leaves the group increases linearly with the group size.

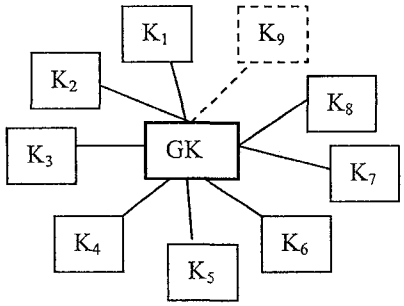


Fig. 3. The keys maintained by a star rekey manager for 9 members.

3.2.2 Logical Key Hierarchy (LKH) Approach for a Rekey Manager

A LKH rekey manager maintains one group key *GK*, an individual key for every group member, and a set of key-encrypting keys (KEKs) used for scalable rekeying. A LKH of a specified degree *d* is constructed such that *GK* is the root of the hierarchy, and every individual key represents a leaf node. Fig. 4 illustrates a LKH of degree $d = 3$ and height $h = 2$ for 9 members, where the root node represents *GK* and the leaf nodes represent the members' individual keys.

Every group member holds the set of keys at the nodes that fall in the path that leads to the root, starting from his individual leaf node key. To guarantee perfect backward secrecy, if a new member joins the group his individual key is inserted in the hierarchy

and all the keys on the path from his individual key leaf node to the root are regenerated. Similarly, to guarantee perfect forward secrecy, if a member leaves the group, his individual key is deleted from the hierarchy and all the keys he was holding are regenerated. In both cases, the rekey manager needs to construct a RM that contains such keys update. The constructed RM will contain a *rekey packet* for every new (regenerated) key.

For example, in Fig. 4 if a new member joins the group and his individual key $K_{3.3}$ is to be inserted, two keys need to be regenerated K_3 and *GK*. The RM in this case has two rekey packets for the two new keys, $[\{K_3^{'}\}K_3, \{K_3^{'}\}K_{3.3}]$ and $[\{GK^{'}\}GK, \{GK^{'}\}K_3^{'}]$. The first rekey packet has two encryptions of $K_3^{'}$, the first encryption is decrypted by old group members (who maintain K_3) to retreive $K_3^{'}$, and the second encryption is decrypted by the new member's individual key to retrieve $K_3^{'}$. Similarly, the second rekey packet has two encryptions of *GK*, the first encryption is decrypted by all old group members to retreive *GK'*, and the sencod encryption is for the new member (after he gets $K_3^{'}$). On the other hand, if that member leaves the group and his individual key $K_{3.3}$ is to be deleted, the same two keys need to be regenerated. The RM in this case has the two rekey packets $[\{K_3^{''}\}K_{3.1}, \{K_3^{''}\}K_{3.2}]$ and $[\{GK^{''}\}K_1, \{GK^{''}\}K_2, \{GK^{''}\}K_3^{'}]$. The new keys $K_3^{''}$ and $GK^{''}$ can no longer be encrypted with their previous versions since the leaving member already knows them, instead every new key is encrypted individually by each sibiling node key (after deleting the individual key node of the leaving member).

We can see that when using LKH and inserting $K_{3.3}$ the RM has 4 encrypted keys compared to only 2 in the star rekey manager. On the other hand, when deleting $K_{3.3}$ the RM has 5 encrypted keys compared to 8 in the star rekey manager.

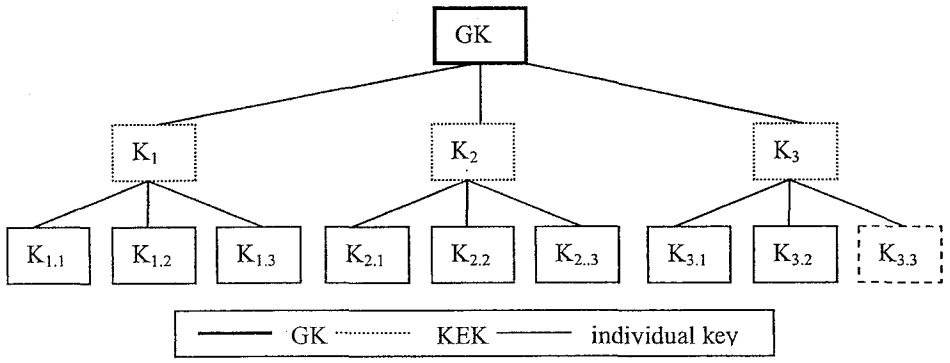


Fig. 4. A LKH of degree $d = 3$ and height $h = 3$ for a group of 9 members.

3.2.3 Encryption for Key Distribution

The traditional technique for distributing the group key in the above two rekey managers (star and LKH) is the use of encryption. The star rekey manager, for a group of n members, performs 2 key encryptions when a new member joins the group and performs $(n-1)$ key encryptions when a member leaves the group to construct a RM that updates *GK*. The leave rekey cost, using star rekey manager, increases linearly with the group size n increase.

A rekey manager that maintains a LKH of degree d and height h, for a group of n members, performs (at most) $(2 \times h)$ key encryptions when a new member joins the group and performs (at most) $(d \times h - 1)$ key encryptions when a member leaves the group. If the LKH is a complete tree for n members then the height $h = \log_d n$. The rekey cost is logarithmic in the group size n in both the join and leave cases, which is a scalable solution. Although the use of LKH provides a scalable group rekey solution, the cost of join and leave rekeyings are not symmetric. In addition, increasing the degree of the hierarchy d that decreases its height h and leads to a decrease of the join rekey cost while increases the leave rekey cost. For example, for a group of size $n = 512$ members, a LKH of degree $d = 2$ is of height $h = 9$ (assuming it is constructed as a complete tree), while a LKH of degree $d = 8$ is of height $h = 3$. In the first case, $d = 2$, the join rekey cost

is 18 encrypted keys and the leave rekey cost is 17 encrypted keys. On the other hand, when $d = 8$ the join rekey cost is 6 encrypted keys (1/3 the first case) and the leave rekey cost is 23 encrypted keys (4/3 the first case, and 4 times the join case). Wong et al. proved that the optimal LKH degree is 4 when encryption is used [67].

3.3 XORBP: A Novel Group Key Distribution Technique

Brute force techniques used to guess a key have to search on the average half the key space. Unless plain text is provided, the analyst must be able to recognize plain text as plain text. If the message is just plain text in English, then the result pops out immediately (although the task of recognizing English would have to be automated). If the message is some more general type of data, such as a "numerical" data, the problem becomes even more difficult to automate.

From the above observation, we can notice that all techniques that encrypt the new *GK* by any other key (previous *GK*, individual key, or KEK) do unnecessary work, and the same security can be achieved with much less computation effort. The new proposed computation method will use bit XOR operation between two keys instead of encrypting one with the other. The XOR operation is sufficient to protect the key material from outsider attacks (members outside the group) but doesn't protect individual key material from insider attacks (members inside the group). Hence, to protect from insider attacks, we suggest distributing the key material in random byte patterns (BPs) in a fixed size rekey packet.

3.3.1 Why XOR

Assume $C = A \oplus B$, where A and B are keys of size k bits[3]. The XOR operation has the following properties:

- Easy computation.
- The output C is always the same size as the two inputs (k bits). This property is not valid in addition and subtraction operations (e.g. in TABLE I: $11+11=110$).

[3] The symbol $\oplus$ denotes a logical XOR operation while & denotes logical AND operation.

- Reversible easy computation, i.e. knowing A and C, we can uniquely and easily calculate B. Unique reversible computation is not valid in AND and OR operations (e.g. in TABLE II: $10 \& 11 = 10 \& 10 = 10$).
- All output values are uniformly distributed in the output space. The output matrix size is $(2^k \times 2^k = 2^{2k})$, every output value in the range $[0:(2^k - 1)]$ appears 2^k times (see TABLE III). This property is not valid in all other simple operations (AND, OR, addition, or subtraction).
- Every output value can be generated with 2^k combinations. That is, knowing C only 2^k guesses are needed to know A and/or B. This property is not valid in all other simple operations.

TABLE I.

A + B, WHERE A AND B ARE 2 BITS LONG

+	00	01	10	11
00	00	01	10	11
01	01	10	11	100
10	10	11	100	101
11	11	100	101	110

TABLE II.

A & B, WHERE A AND B ARE 2 BITS LONG

&	00	01	10	11
00	00	00	00	00
01	00	01	00	01
10	00	00	10	10
11	00	01	10	11

TABLE III.

A ⊕ B, WHERE A AND B ARE 2 BITS LONG

⊕	00	01	10	11
00	00	01	10	11
01	01	00	11	10
10	10	11	00	01
11	11	10	01	00

The last two properties of XOR operation makes $(A \oplus B)$ as secure as $\{A\}B$, to all members who don't know both A and B. Performing XOR operation between the keys solves the problem of protecting the key material from outsider attacks. But this operation doesn't protect the key material from insider attacks. For example, if we have a group of two members X and Y, each one has his own individual key that is known only by him and by the rekey manager. Let the individual key of X is K_X, and the individual key of Y is K_Y. Assume the rekey manager needs to send them the group key *GK*. Previous methods used to broadcast a rekey packet that contains $[\{GK\}K_X, \{GK\}K_Y]$, the group key encrypted with every member individual key. Every member reads his own part in the packet, and decrypts it to retrieve the group key. Members outside the group can't learn any key material, and members inside the group can't learn each other keys.

Alternatively, the new method suggests sending a rekey packet that contains $[GK \oplus K_X, GK \oplus K_Y]$. We can see that Y who knows K_Y can retrieve *GK* easily, but also can retrieve K_X, since he can read $(GK \oplus K_X)$ and thus $(GK \oplus K_X) \oplus GK = K_X$. Hence, adding a security barrier to insider attacks is essential. The suggested method to protect from insider attacks is to distribute the key material in random byte pattern BP in the broadcast rekey packet. Every BP is known only by the rekey manager and by the individuals similar to K_X and K_Y. Every BP specifies a unique byte numbers in a fixed size rekey packet.

For example if all keys are of size 3 bytes, and the rekey packet size is 260 bytes. Assuming $BP_X = \{200,120,79\}$ and $BP_Y = \{110,205,55\}$, the suggested technique for distributing GK to the two members, is to distribute the 3 bytes of $(GK \oplus K_X)$ in BP_X packet bytes numbered 200, 120, and 79 respectively, and distribue the 3 bytes of $(GK \oplus K_Y)$ in BP_Y packet bytes numbered 110, 205, and 55 respectively.

3.3.2 Protection from Insider Attacks

If the key length is k bits, the key space that is searched by attackers has 2^k different key values. Let $K = \lceil k/8 \rceil$ be the size of the key in bytes. For a group of n members, the rekey manager that uses encryption (encrypt GK with every member individual key) to broadcast GK will send a rekey packet of size $(n \times K)$ bytes. The rekey manager that uses XOR and BP for distributing GK for n members should broadcast a rekey packet of size $(n \times K + E)$ bytes, i.e. the rekey packet contains E extra bytes.

The worst-case insider attack is that $(n-1)$ colluding members trying to guess the BP (and therefore the key) of the remaining member. A group of $(n-1)$ members can exclude $((n-1) \times K)$ bytes from the packet that contain their own versions of the key. The remaining are $(K + E)$ bytes, and they are trying to select K ordered bytes. In this case E is estimated so that they have a search space larger than or equal to the search space of the protected key. We can see that there is an E extra bytes increase in the message size, and this increase is the price paid for reducing the computation form n encryptions to simple XOR operations. The inequality $P_K^{(K+E)} \geq 2^k$ is used to estimate the extra bytes size[4].

Note that if we distribute the key material GK in the byte patterns BPs instead of the XORed keys $(GK \oplus K_X)$ and $(GK \oplus K_Y)$ the rekey packet will contain a repeated byte patterns of GK and that will make it easier (less permutation) for attackers to make a guess. If GK is distributed in "bit patterns" instead of byte patterns that will solve the repeated byte patterns problem but will increase the size of the data needed to keep the

---

[4] $P_a^b = \frac{b!}{(b-a)!}; c! = c \times (c-1) \times \ldots \times 1$

patterns. For example: If we have a key of size K bytes, and a rekey packet of size S bytes. To decode a "byte" location in the rekey packet s bits are needed such that $s = \lceil \log_2 S \rceil$. To decode a "bit" location in that rekey packet (s + 3) bits are needed. The data size for a key and byte pattern BP is = $(K + \lceil K \times s/8 \rceil)$ bytes. Using a bit pattern (there is no need for the key), the data size = $K \times (s+3)$ bytes. The data size required using byte pattern BP is less than the data size required using bit pattern for all positive values of s $(K \times (s+3) > (K + \lceil K \times s/8 \rceil)$ for all $s > (-16/7))$. On the other hand, using bit patterns might decrease the required extra bytes E and therefore the message size. If bit patterns are to be used, the rekey message extra bytes E can be estimated by solving the inequality $P_k^{(k+8\times E)} \geq 2^k$.

3.3.3 Extra Bytes Adjustment

The insider attack by one member in which he can exclude his own BP (K bytes), and make a guess for any other BP has a search space size equals to $P_K^{((n-1)\times K+E)}$. The insider attack by m members has a search space of size $P_K^{((n-m)\times K+E)}$. In the worst-case insider attack by $(n-1)$ members, the search space size is $P_K^{(K+E)}$, in which E is estimated to make the search space size greater than or equal to the search space of the protected key to have the same security achieved by encryption.

The extra bytes E depends only on the key size K, and does not depend on the group size n. We can further reduce E if the insider attacks are rare, or the cost of protecting the key from insider attacks (represented in E) is greater than the benefits gained from that protection. For example, if the keying material is changing frequently, we can assume that the lifetime of the key is shorter than the time required for making a correct guess using a reasonable cost machine. We can estimate a reasonable search space size Q from the lifetime of the key and calculate the extra bytes E such that $P_K^{(K+E)} \geq Q$, where Q is the reduced search space size.

In addition, using XOR operation instead of encryption helps in protecting the individual key material from the "*known plain text attack*". Known plain text attack, is an attack in which plain text and its corresponding cipher text are known by the attacker. It

has been proven that in some cases this information can reduce the search space for the key. Using encryption methods, an insider can build a database from the rekey messages RMs that contains pairs of (*GK*, *GK* encrypted by other individual key(s)). This database can help him in guessing the other individual key(s). The database can grow quickly if the group has frequent joins and leaves and therefore frequent RMs are sent. This type of attack does not exist when using XORBP. Using the above observation, if the known plain text attack reduces the search space size from 2^k to Q. Then extra bytes E is chosen such that $P_K^{(K+E)} \geq Q$, where Q is the reduced search space size.

3.4 Logical Key Hierarchy and XORBP

The following is a summary of the terminology used. The group key is *GK*. All key sizes are K bytes. The LKH degree is d. The group size (number of members) is n. The LKH height h for n members is $h = \lceil \log_d n \rceil$, assuming it is constructed as a complete tree.

The key data maintained by a rekey manager is a LKH of degree d and height h. Using XORBP, each non-root node key (KEK or individual key) is accompanied by a byte pattern BP. Each member is assigned a leaf node in the LKH, which contains his individual data (key, BP, ...etc). In addition, every member knows his leaf node *position* in the LKH, and holds all the entries of the LKH in the path from his leaf node up to the root.

It is assumed there is only one join or one leave at a time in which *GK* and all LKH entries (keys and BPs) that are held by that member need to be regenerated (batch rekeying for a set join and/or leave requests is discussed in chapter V). The rekey manager broadcasts a rekey message RM for every join or leave rekey. The RM contains a *message-identifier*, a rekey packet for every new key, and an encoded BP for every new BP. The message-identifier is the leaf node position of the member who caused the rekey either because he joined or left the group.

3.4.1 XORBP Rekey Packet Construction

The rekey message RM contains a rekey packet for every newly generated key, *GK* or KEK. A rekey packet contains the new key distribution information and is targeted to a corresponding set of members that should hold that key. No other member in the group or outside the group should be able to *easily* retrieve any key information from the rekey packet.

A XORBP rekey packet size is $S = d \times K + E$ bytes; where d is LKH degree, K is the key size in bytes, and E is the estimated extra bytes (section 3.3.3). A XORBP rekey packet constructed for a new key K_u at LKH node u contains multiple versions of K_u XORed with the keys at the LKH sibling nodes of u. If u is the key node path starting from the root, then uv is a path to a sibiling node of u. If the number of sibilings for node u is e (e is less than or equal to LKH degree d) there exists e sibling nodes determined by the path uv, $1 \le v \le e$, where each node contains (K_{uv}, BP_{uv}). All sibling nodes of a node u will be denoted (K_{u*}, BP_{u*})

The rekey packet is constructed for the new K_u such that for every v, the BP_{uv} bytes in the rekey packet contains $(K_u \oplus K_{uv})$. The remaing empty bytes in the rekey packets contains dummy (randomly generated) bytes.

3.4.2 Encoded Byte Pattern

When generating a new key, its corresponding BP needs to be regenerated too. Similar to the key, the newly generated BP needs to be sent in the rekey message RM to the group members who should hold it. As previously described, a new key will be distributed in a rekey packet of size S bytes such that $S = d \times K + E$, where d is LKH degree, K is the key size in bytes, and E is extra bytes. Each sibling node of the distributed key node should have a unique byte pattern BP that specifies unique K bytes in the rekey packet of S bytes.

Guaranteeing unique BPs can be implemented by maintaining an array R of Booleans of size S with every key, every entry in R corresponds to a byte in the rekey packet. Initially all array entries are set to *true*, a *true* value means the byte is free (i.e. not assigned to any sibling key node) while a *false* value means the byte is already assigned

and can't be assigned to any other sibling. When regenerating a BP, the old K bytes (old BP byte numbers) have to be freed (i.e. marked *true* in R) and then new free K bytes are selected (and marked *false* in R). The generation of a random BP will require the generation of K random numbers in the range [0:S-1]. Since the maximum number of node siblings is d, this would guarantee at least $(K+E)$ free bytes for a new BP of K bytes. If any of the randomly generated byte numbers is not free (locked up in R), the nearest free byte is chosen instead.

Similar to a key, a new BP can't be sent plain in a RM, instead it is encoded so that it can be retrieved only by the targeted members (members who maintains its corresponding key). The encoding of a newly generated BP is performed using its corresponding newly generated key. The new BP is first represented as a string of bits then XORed with the corresponding generated key. The bit representation of BP might be of shorter or longer length than the key. If it is shorter than the key, it is XORed with the first same-length bits of the key. If it is longer than the key, the key bits are repeated fully or partially (one or more times) until the exact length is reached.

For example, if the key size K is 3 bytes (24 bits), and the rekey packet size S is 260 bytes. Since S equals 260, 9 bits are enough to represent a byte number in the range [0:259]. A BP can be represented by a string of length $3\times 9 = 27$ bits that is approximated to 4 bytes when sent in a RM. For a key $K = \overbrace{11011011}^{3rd-byte}\overbrace{00001001}^{2nd-byte}\overbrace{01001100}^{1st-byte}$, and BP = (200, 79, 120) that can be represented as string of bits $BP = \overbrace{001111000}^{120(9bits)}\overbrace{001001111}^{79(9bits)}\overbrace{011001000}^{200(9bits)}$, the encoded BP is calculated as follows:

$$\begin{array}{l} \overbrace{001111000}^{120}\overbrace{001001111}^{79}\overbrace{011001000}^{200} \oplus \\ \underbrace{100}_{1st-3bits}\ \underbrace{11011011}_{3rd-byte}\ \underbrace{00001001}_{2nd-byte}\ \underbrace{01001100}_{1st-byte} \\ \hline \underbrace{101}_{4th-byte}\ \underbrace{00111011}_{3rd-byte}\ \underbrace{00011010}_{2nd-byte}\ \underbrace{01101100}_{1st-byte} \end{array}$$

. Note that the first three bits of the first key byte is repeated to reach the exact BP bit string size.

3.4.3 Simple Case: LKH of Height $h = 1$

A LKH of height $h = 1$ can fit a group of maximum size n such that $n \leq d$. The RM distributed by the rekey manager contains one rekey packet for a newly generated *GK* per every join or leave. The key data maintained by the rekey manager is shown in Fig. 5.

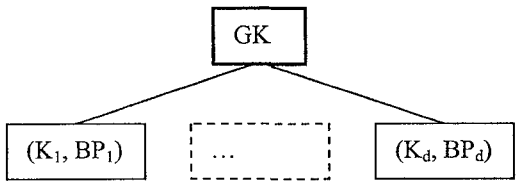


Fig. 5. A LKH of degree d and height $h = 1$.

The rekey procedure for a join or a leave of a member X whose individual key is K_X :

1. Determine the leaf node *position* for member X individual data to be inserted/deleted. The member position will be used as a message-identifier.
2. **If** (X is joining) **then** {Select free BP_X and send it to member X through his private channel along with his leaf node position; Insert the individual leaf node (K_X, BP_X) into LKH.} **else** {Delete the individual leaf node (K_X, BP_X) from LKH.}.
3. Generate new *GK*.
4. Construct a rekey packet for the new *GK* using (K_*, BP_*) (as described in section 3.4.1).
5. Send a rekey message RM that contains the constructed rekey packet to all group members.

3.4.4 Another Simple Case: LKH of Height $h = 2$

A LKH of height $h = 2$ can fit a group of maximum size n, such that $d < n \leq d^2$. Using a LKH of height 2, group members can be virtually viewed as arranged in d partitions (at most), and every partition contains d members (at most). A LKH of height 2 maintained by a rekey manager is shown in Fig. 6. For any partition p, all members at that partition hold the same partition key K_p and the same partition BP BP_p that are used in constructing a new GK rekey packet. Moreover, each member X holds his individual key K_{pX} and his individual BP BP_{pX} that are used in constructing a new partition key K_p rekey packet. For every join or leave rekeying, the rekey message RM distributed by the rekey manager contains two rekey packets one for a new K_p and the other is for a new GK. In addition, RM contains one encoded new BP BP_p.

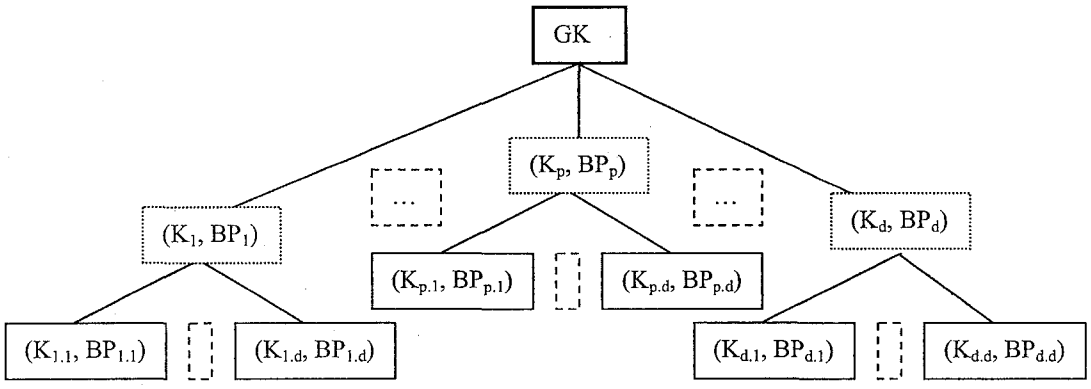


Fig. 6. A LKH of degree d and height $h = 2$.

The rekey procedure for a join or a leave of a member X whose individual key is $K_{p.x}$:

1. Determine the position, and therefore the partition p, where member X individual leaf node will be inserted/deleted. The member position will be used as a message-identifier.
2. **If** (X is joining) **then** {Select free $BP_{p.x}$ and send it to member X through his private channel along with his leaf node position; Insert the individual leaf node $(K_{p.x}, BP_{p.x})$ into LKH.} **else** {Delete the individual leaf node $(K_{p.x}, BP_{p.x})$ from LKH.}.
3. Generate new partition p node entries K_p and BP_p.
4. Construct a rekey packet for the new K_p using the individual leaf nodes $(K_{p.*}, BP_{p.*})$. (as descibed in section 3.4.1)
5. Encode the new BP_p with the new K_p (as described in section 3.4.2).
6. Generate new *GK*.
7. Construct a rekey packet for the new *GK* using partition nodes (K_*, BP_*)
8. Send a rekey message RM that contains the two rekey packets (from step 4 and 7) and the encoded BP (from step 5) to all group members.

Is changing the partition BP_p necessary? Yes it is necessary, and we will show this by example, assuming everybody knows the rekey procedure. If member X joins the group in partition A then he will hold K_A and BP_A. Assume X left the group for a while and joined it again in different partition B. Assume K_A is changed to K'_A when X left but BP_A is the same, and X (who knows the procedure) is able to memorize BP_A. A member X can retrieve the new K'_A when he joins partition B because he knows *GK* (member of the group), and he can read $(GK \oplus K'_A)$ at the same BP_A. If X left the group again and no other member joined or left partition A, X can retrieve the new *GK* and access further information by knowing BP_A and K'_A (note that he was a member of partition B, and K_B is changed once he left but K'_A is the same).

3.4.5 General Rekey Procedure by the Rekey Manager

Assume the inserted/deleted leaf node immediate parent position is $l_h l_{h-1} ... l_3 l_2$ as illustrated in Fig. 7, where l_h decodes a child node position of the root node (child at level h), l_{h-1} decodes a child node position of the node determined by l_h (child at level (h-1)), ..., l_2 decodes a child node position of the node determined by l_3 (child at level 2) and is the immediate parent of the leaf node that contains the member data (individual key and individual BP) among at most (d-1) other individual members data nodes.

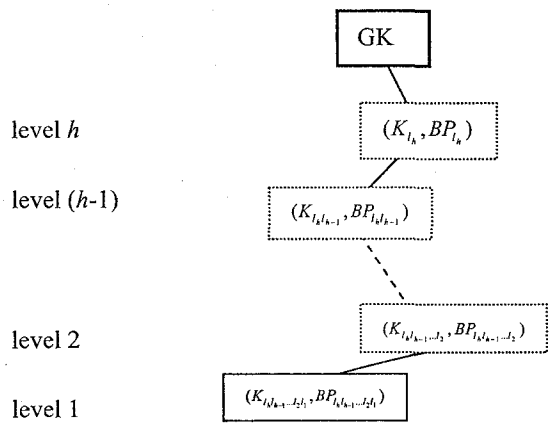


Fig. 7. The path to a leaf node in a LKH of height h.

The rekey procedure for a join or a leave of a member X:

1. Determine the leaf node position of member X individual data to be inserted/deleted, and therefore determine all the LKH nodes entries that need to be regenerated from the root node to the leaf node immediate parent. Assuming the position is $l_h l_{h-1} ... l_3 l_2$, and the LKH entries are GK, (K_{l_h}, BP_{l_h}), $(K_{l_h l_{h-1}}, BP_{l_h l_{h-1}})$, ..., and $(K_{l_h l_{h-1} ... l_2}, BP_{l_h l_{h-1} ... l_2})$. The position is used as the RM message-identifier.

2. **If** (X is joining) **then** {Select free individual BP $BP_{l_h l_{h-1}...l_2 l_1}$, and send it to member X along with his position through his private channel. Insert the member individual leaf node $(K_{l_h l_{h-1}...l_2 l_1}, BP_{l_h l_{h-1}...l_2 l_1})$ at the first level of LKH.} **else** {Delete the member individual leaf node $(K_{l_h l_{h-1}...l_2 l_1}, BP_{l_h l_{h-1}...l_2 l_1})$ from the first level of LKH.}
3. **For** every LKH entry at level $i = 2$ **to** h {
   a. Generate new key $K_{l_h l_{h-1}...l_i}$ and select new BP $BP_{l_h l_{h-1}...l_i}$.
   b. Construct a rekey packet for the new $K_{l_h l_{h-1}...l_i}$ using $(K_{l_h l_{h-1}...l_i *}, BP_{l_h l_{h-1}...l_i *})$ nodes at level (i-1) (as described in 3.4.1).
   c. Encode the new $BP_{l_h l_{h-1}...l_i}$ with the new $K_{l_h l_{h-1}...l_i}$ (as described in 3.4.2).}
4. Generate new *GK*.
5. Construct a rekey packet for the new *GK* to all members using level h keys and BPs (K_*, BP_*).
6. Send a RM to all members that contains the message-identifier, all constructed rekey packets, and all encoded BPs.

Note that, the rekey procedure is almost symmetric for both join and leave cases. Note also that all if all LKH new entries are generated at once (step 3.a and step 4), constructing the rekey packet and encoding BP for each new entry can be performed in parallel (step 3.b-c and step 5).

3.4.6 How Group Members Retrieve the New Keys and the New Byte Patterns

Not all rekey packets and all encoded BPs should be read and processed by all group members. Since every members knows his position and the RM includes the identification $l_h l_{h-1}...l_3 l_2$ (the position of the member who joined/left). Every member can select the rekey packets and the encoded BPs to process. The RM contains h rekey packets and (h-1) encoded BPs.

For every i, where $1 \le i < h$, the ith rekey packet and the ith encoded BP contain a new key and a new BP for a node at the $(i+1)^{th}$ level. This data should be retrieved by at most d^i members who have their position matches $l_h l_{h-1}...l_{i+1}$ *. The h^{th} rekey packet contains the new *GK*, and should be retrieved by all n (at most d^h) members

3.5 Scenarios and Comparison

In this section, we compare the group rekeying performance when the traditional key management approaches (star and encryption-based LKH) are used, and when the proposed approach (XORBP LKH) is used. Two examples are used to demonstrate such approaches, a group of members joining a subscription News broadcast server, and a group of peer-to-peer machines communicating securely.

We assume the group rekeying is performed periodically by a GKM that will learn the join and leave requests right before a rekeying process is initialized. The GKM will perform some time-consuming operations, e.g., random number generation and encryptions, before a rekeying, if any, and delay the rest of the operations, e.g., encryptions, until the exact requests are known. The following are the three approaches under consideration.

Approach 1: **The traditional star key management approach**. The GKM changes the group key and encrypts it individually for every group member. This approach requires the GKM to regenerate one key and to perform O(n) keys encryptions to provide perfect forward secrecy for a group of n members, i.e., the rekeying cost scales linearly with the group size. A group member performs 1 decryption to retrieve the group key. In this approach, the GKM can regenerate a new group key and encrypt it with the every group member individual keys right after a rekeying is committed and before learning the next requests.

Approach 2: Encryption-based LKH. The use of a LKH by a GKM provides a scalable group rekeying that scales to the logarithm of the group size. If the LKH degree is d, and its height is h ($n \leq d^h$), the GKM is required to regenerate O(h) keys and to perform O($d \times h$) keys encryptions to provide perfect forward secrecy after single leave request. However, for a batch of R requests, the GKM is required to regenerate O($R \times h$) keys and to perform O($R \times d \times h$) keys encryptions. A group member is required to perform at most h decryptions to retrieve the new group key (more than the required cost by the star approach).

Approach 3: **XORBP LKH**: The new key distribution technique, XORBP, is used with the LKH approach to provide a more scalable and efficient group rekeying that

doesn't require any encryption/decryption to be performed by the GKM or by any group member. Similar to the encryption-based LKH approach, for a batch of R requests, the GKM is required to regenerate $O(R \times h)$ keys, in addition, the GKM random number generation overhead is increased. Since no encryption is used, this approach reduces a rekeying time to 10% of the encryption-based LKH rekeying value, for the same LKH. Most of this time is spent in random number generation. The three approaches offer the same security capabilities.

3.5.1 A News Broadcast Server Example

Consider a News broadcast server that encrypts its broadcast using a group key that is handed to every newly joined member. Assume that the total number of connected group members at any point of time is 30,000 and the used encryption algorithm requires 1 msec for a single encryption/decryption. In addition, assume the server changes the group key periodically every 30 sec, and the average number of leave requests is 100 and the join requests are 50 in the 30 sec inter-rekey period. Consequently, a newly joined member might have to wait at most 30 sec before being able to decrypt the broadcast, and a leaving member might be able to decrypt the broadcast for maximum of 30 sec after he leaves.

Star key management: The GKM is required to perform 30,000 key encryptions which consume 30 sec. A group member only has to perform 1 decryption to extract the group key that consumes 1 msec. The GKM can start encrypting a new group key with every group member key before a rekeying. When he learns of the requests, he will throw away the encryptions performed for the leaving members (100 encryptions) and has to perform encryptions for the newly joined members (50 encryptions). The GKM needs the whole inter-rekey period to perform the encryptions. The traditional star key management has a problem in the following cases: the need to support a larger group size, the use of a more time consuming encryption standard, and the 30 sec maximum request delay is not acceptable.

Encryption-based LKH: If the LKH degree d = 4, height h = 10 due to nodes insertion and deletion, and the number of LKH new keys (rekey sub-tree size) for the 150 requests is 100 keys. The total number of key encryptions at the server = $100 \times 4 \times 10 = 4$

sec (compared to 30 sec in the star approach). A group member has to perform at most 10 decryptions that is 10 msec (every 30 sec). A group member decryption cost is increased compared to the star approach.

The GKM can perform all random number generation before learning the exact requests. However, most of the encryptions (if not all) has to be performed after learning the exact join and leave requests. This can be a drawback of LKH if we need to reduce the time after learning the requests and the start of the rekeying. However, if the rekeying is performed frequently, the total time spent by the GKM performing encryptions is a better cost measure.

The rekey cost for LKH with encryption based KDT increases in the following cases:

1) The LKH degree is increased.
2) The number of requests is increased. For example, 1000 new keys need to be distributed (instead of 100), in this case, the cost of LKH is worse than the star approach and requires 40 sec of GKM encryption time.
3) The LKH height is increased due to nodes insertion and deletion (i.e., maintaining a balanced LKH greatly affect the number of new keys/encryptions).

XORBP LKH: The rekeyig time is reduced to 10% of the above values that is 400 msec for the GKM and 1 msec for a group member (every 30 sec). The number of new keys for a single request is O(h). The total number of new keys for the 150 request is $100 \times 10 = 1000$ keys. The rekeying time doesn't increase with the LKH degree increase and slightly increases with larger number of requests or an unbalanced LKH since no encryption is performed (i.e., more XOR operations are performed). A group member can have the minimum cost achieved using the star key management.

Similarly, the GKM can perform random number generation before learning the requests. Compared to the star approach, the GKM can achieve better performance after learning the requests, since in the star approach the GKM has to perform encryptions for the newly joined members.

3.5.2 A Secure Peer-to-Peer Network Example

Consider a secure peer-to-peer network for a group of 1000 members (machines), and 1 msec encryption/decryption standard. If one machine (member) is hosting the GKM

and there are 50 member join/leave requests every 30 sec. A new group key will be issued every 30 sec. The following are the rekeying time costs for the considered approaches.

Star key management: the server encryption time = 1 sec; a group member decryption time = 1 msec (every 30 sec).

Encryption-based LKH: the server encryption time = $50(requests) \times 4(d) \times 4(h) = 800$ msec; a group member decryption time = 4 msec.

XORBP LKH: the server time cost = 80 msec; a group member time cost is less than 1 msec.

We can observe that the increase in the number of requests in the encryption-based LKH approach could lead to a worse performance than the star approach. The group member hosting the GKM prefers the minimum overhead approach that doesn't affect (disrupt) the application.

Similarly, performing pre-operations could reduce the time between knowing the requests and the actual rekeying in the star approach over the encryption-based LKH approach. The encryption-based LKH approach is better than the star approach if the total server (light-weight) time spent performing encryptions are compared. The XORBP LKH approach outperforms the other two approaches in both cases.

3.6 Cost Analysis and Estimates

The parameters to the cost equations are the key size K and its corresponding search space size Q (used in estimating the extra bytes E), and the LKH degree d.

3.6.1 How to Select the Key Size

It is usually assumed that group members are sharing a symmetric encryption key. Using symmetric cryptography usually achieves faster encryption/decryption than asymmetric cryptography. The key size is dependent on the used encryption algorithm. The two widely used symmetric key encryption algorithms are "Data Encryption Standard" (DES) that uses 56 bits key and "International Data Encryption Algorithm" (IDEA) that uses 128 bits key [40]. While an IDEA key is encoded in 16 bytes $(128/8 = 16)$, a DES key is encoded in 8 bytes (7 bits in every byte contain part of the

key $(7 \times 8 = 56)$ and the 8^{th} bit in every byte is used for parity check). Another widely used version of DES is called triple DES (or DES-EDE) that uses 3 DES keys [40].

Assuming the maximum key search space size $Q = 2^k$, the estimated extra bytes E for the above three algorithms are as follows:

- For DES, the extra bytes E can be estimated from $P_8^{8+E} \geq 2^{56}(7.2e^{16})$; where $E = 127$ bytes satisfies the ineqality $(P_8^{135} \approx 8.9e^{16})$.
- For IDEA, the extra bytes E can be estimated from $P_{16}^{16+E} \geq 2^{128}(3.4e^{38})$; where $E =$ 264 bytes satisfies the inequality $(P_{16}^{280} \approx 9.2e^{38})$.
- For triple DES, the extra bytes E can be estimated from $P_{24}^{24+E} \geq 2^{168}(3.7e^{50})$, where E =116 bytes satisfies the inequality $(P_{24}^{140} \approx 3.9e^{50})$.

3.6.2 How to Select the Degree of the Hierarchy

If it is desired to keep the rekey packet size S less than 1500 bytes to fit in one UDP packet (Ethernet network), in which case, a rekey packet can be sent without fragmentation. The degree d can be calculated using the equation $S = d \times K + E$, where K is determined form the used encryption algorithm and E is the estiamted extra bytes.

Increasing d will decrease h, and therefore will decrease the computation cost at the rekey manager and at every group member if XORBP is used as a KDT. On the other hand, increasing d increases the LKH node size as well as the rekey packet size S. Moreover, selecting d such that the byte pattern BP is represented in an exact size of bytes will omit adding extra bits (section 3.4.2). The LKH degree d can take into consideration the disk block size if the LKH will be stored on disk. In such case, it is better to keep each node in one disk block for easier access.

3.6.3 Cost Estimation

The following are the cost estimates of LKH key management approach used with XORBP KDT. Assuming the rekey packet size $S = d \times K + E$, where d is the LKH degree, K is the key size in bytes, and E is the estimated extra bytes. Let $s = \lceil \log_2 S \rceil$ that is s bits are needed to identify a byte location in a rekey packet. If the group size is n, and

LKH height is h ($h = \lceil \log_d n \rceil$ for a balanced LKH); the maximimun group size for the same height LKH $Max\_n = d^h$ (assuming the LKH is complete). The analytical cost estimates is as follows:

- Byte pattern size: $BPS = \lceil s \times K / 8 \rceil$ bytes.
- LKH root node (GK) size = K bytes.
- LKH non-root node size (NS) that contains a key and BP $NS = K + BPS$ bytes.
- A LKH (of degree d and height h) storage size can be estimated by adding the nodes' sizes at all levels. The required LKH storage (LKHS) for a group of size n smaller than Max_n can be estimated as

$$LKHS = \frac{n}{Max\_n} \times (K + \sum_{i=1}^{h} d^i \times NS) = \frac{n}{Max\_n}(K + \frac{d \times (Max\_n - 1) \times NS}{(d-1)}) \text{ bytes}$$

- A group member holds the LKH root node entry (GK), and (at most) h non-root nodes entries. The required group member storage (MS) can be estimated as $MS = K + h \times NS$ bytes.
- The rekey message RM contains (at most) h rekey packets and (h-1) encoded byte patterns BPs. The maximum RM size (RMS) can be estimated as $RMS = h \times S + (h-1) \times BPS$ bytes.
- Maximum number of newly generated keys = h per a rekey.
- Maximum number of newly generated BPs is (h-1). Therfore, the maximim number of randomly generated byte locations[5] can be estimated as = $(h-1) \times K$ per a rekey.
- If e_i is the number of sibilings at the ith level of a newly generated key node at level (i + 1). The rekey packet for that key contains ($e_i \times K$) bytes of key material (XORed keys), while the remaing bytes are filled with randomly generated bytes. The total number of the randomly generated bytes can be estiamted as = $(\sum_{i=1}^{h} S - e_i \times K)$ per a rekey.
- As previously explained in section 3.4.6, for every rekey message RM, there are (at most) d members who will process all h rekey packets, d^2 members who will process

(h-1) rekey packets, d^3 members who will process (h-2) rekey packets, ..., d^h (all members) who will process one packet (for GK). The average number of rekey packets processed by a group member can be estimated as = $1+\frac{\sum_{i=1}^{h-1} d^i}{Max\_n} = 1+\frac{(Max\_n-d)}{Max\_n\times(d-1)}$ per a rekey.

In a rekey message RM, there exists an encoded BP that corresponds to every rekey packet except for GK. Similar to the rekey packet, the encoded BP is processed by the same group members that process the corresponding rekey packet. From the above estimate, the average number of encoded BPs processed by a group member = $\frac{(Max\_n-d)}{Max\_n\times(d-1)}$ per a rekey.

For a LKH with encryption-based KDT, the byte pattern size (BPS) is equal to 0 in LKH storage (LKHS) and member storage (MS) estimates as given above. Let Enc_K be the size of an encrypted key in bytes. In such case, the rekey message RM has two different sizes: join RMS ($jRMS = 2\times h\times Enc\_K$) bytes, and leave RMS ($lRMS = (d\times h-1)\times Enc\_K$) bytes. Moreover, when an encryption-based KDT is used the only randomly generated numbers are the new keys.

Comparing the cost of XORBP versus encryption-based KDTs when used with the same degree (d) LKH and for the same group size n: from the above analytical cost estimates, XORBP introduces an increase in LKH node size and therfore an increase in the LKH storage (LKHS) and member storage (MS). In addition, the rekey message size (RMS) is subject to increase depending on the encrypted key size. On the other hand, the use of XOR operations between keys, instead of encryption, promises a substantial decrease in the rekey message construction time as well as the rekey processing time by a group member.

For example, for a group of size n = 4096 that uses DES encryption, key size K = 8 bytes, encrypted key size Enc_K = 16 bytes, and the (larger) estimated extra bytes E =

---

[5] A byte location is a number in the range [0:S-1], where S is the rekey packet size.

127 bytes. Fig. 8, Fig. 9, and Fig. 10 illustrate the analytical cost estimates for XORBP ("x" prefix) versus encryption ("e" prefix). The LKH degree is increased by 4 starting from 4 to 32. Note that: the same figures are obtained by trying different group sizes. Fig. 8. illustrates LKH storage (LKHS) and Fig. 9 illustrates member storage (MS) for both KDTs. As expected, the use of XORBP increases the storage requirement for the rekey manager and the rekey client. We can observe that LKHS and MS are slightly decreasing with the degree increase, and xLKHS and xMS are almost double eLKHS and eMS, respectively, for the same LKH degree.Fig. 10 illustrates the rekey message size (RMS) for the two encryption cases of join (ejRMS) and leave (elRMS) and for XORBP (xRMS). We can observe that when using encryption, the join RMS (ejRMS) is slighlty decreasing with LKH degree increase, while the leave RMS (elRMS) is linearly increasing with LKH degree increase. Similary, increasing LKH degree linearly increases elRM construction time (number of encryptions). Such leave rekey cost linear increase with LKH degree makes it unfeasible to use larger degree LKH with encryption-based KDT. On the other hand, when XORBP is used, xRMS (and therfore the construction time) is symmetric for the join and leave cases. As shown in Fig. 10, xRMS is larger than elRMS for all LKH degrees, but smaller/comparable to elRMS for larger LKH degrees (xRMS has a nonliner relation with LKH degree).

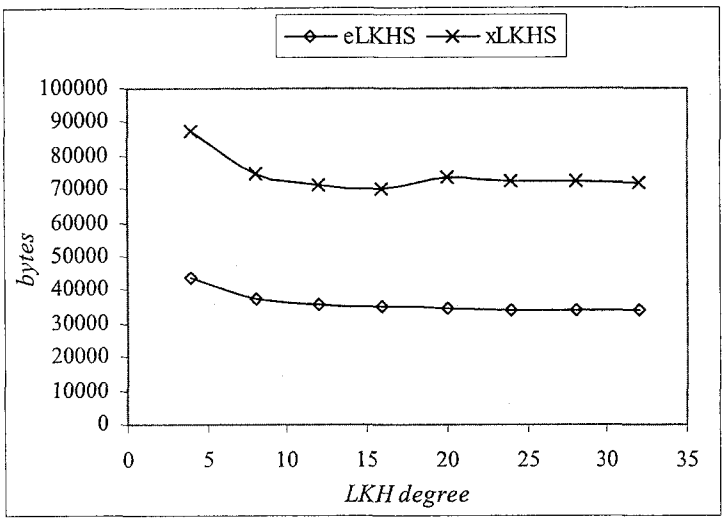


Fig. 8. Comparison of estimated LKH storage (LKHS) when used with encryption-based versus XORBP KDTs.

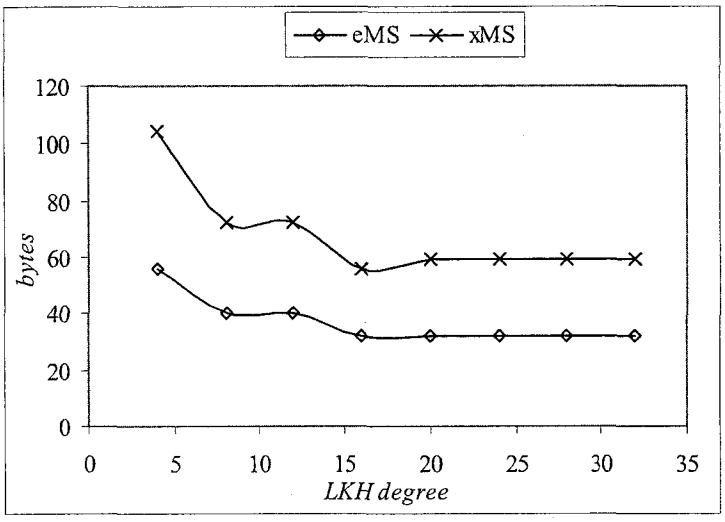


Fig. 9. Comparison of estimated LKH member storage (MS) when used with encryption-based versus XORBP KDTs.

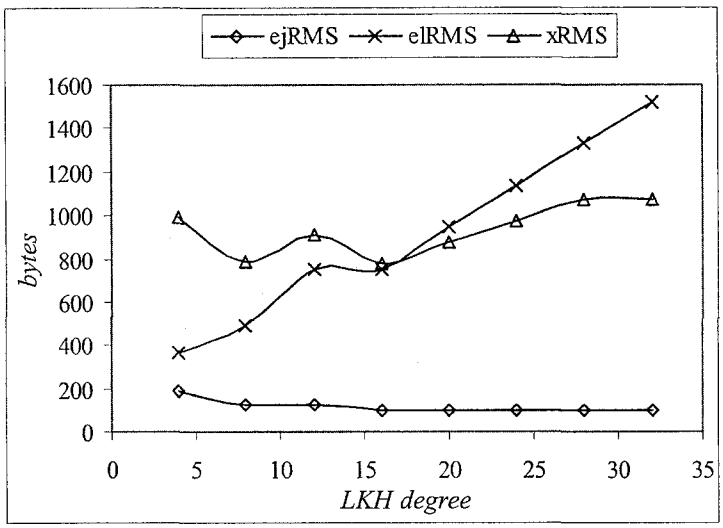


Fig. 10. Comparison of estimated LKH rekey message size (RMS) when used with encryption-based versus XORBP KDTs.

3.7 Experimental Results

We have implemented an initial prototype for the secure group key management components (section 3.1) extending Java™ security [62]. The implementation provides both star and LKH rekey managers. In addition, both encryption-based and XORBP KDTs are available with the use of LKH rekey manager. Moreover, two LKH maintenance algorithms and rekey protocols are available. One protocol adopts an unbalanced LKH while the other adopts a balanced LKH. Chapter IV provides the details of such algorithms and protocols.

We performed experiments to illustrate and compare the rekey message RM construction time in different cases. All experiments ran on the same machine: Sun Ultra-250 with processor speed of 400 MHz, main memory of 2 GB, and operating system Solaris 2.8. In the following experiments: a LKH rekey manager uses the unbalanced LKH algorithms. The group size is increased from 32 to 2048 in multiple of 2 (unless otherwise stated). For each group size, 100 LKHs are constructed by a sequence of

member additions. For every constructed LKH, 10 readings of RM construction time are measured for 5 join rekeyings and 5 leave rekeyings (one join followed by one leave 5 times). Next, the LKH join/leave RM construction time, for that group size, is considered as the average of the 500 readings.

The following experiments study and compare RM construction time as follows: 1) star versus LKH approach for group key management; 2) effect of increasing LKH degree when used with encryption-based or XORBP KDT; 3) effect of increasing the encryption time (i.e. more complex encryption standard) on the saving of RM construction time when XORBP KDT is used over the encryption-based KDT; 4) effect of using secure random number generation on XORBP KDT; and 5) comparing the estimated and measured rekey costs.

3.7.1 Star Versus LKH Key Management Approaches

The first experiment compares RM construction time for star rekey manager versus LKH rekey manager. Both managers are using encryption-based KDT with DES encryption. LKH degree is 4, and the group size increases from 32 to 256 in multiple of 2. Fig. 11 illustrates RM construction time for both managers in both the join and leave rekey cases. For star rekey manager sJoin and sLeave are the RM construction time in the join and leave rekeyings respectively. For LKH rekey manager eJoin(4) and eLeave(4) are the RM construction time in the join and leave rekeyings respectively, where 4 identifies LKH degree. We can observe that sLeave increases linearly with the group size increase and therefore star rekey manager does not provide scalable rekeying. The experiment confirms that using star rekey manager is not practical even for small group sizes.

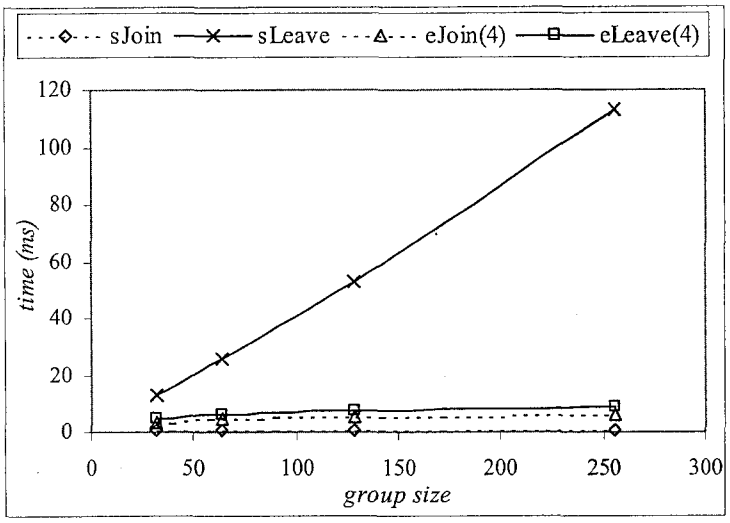


Fig. 11. Comparison of RM construction time in for star versus LKH key management approaches.

3.7.2 Increasing LKH Degree

The second experiment shows the effect of increasing LKH degree on a LKH rekey manager that uses encryption-based versus XORBP KDTs. The encryption algorithm is DES with extra bytes E = 127 bytes. The experiments are performed for LKH of degree 4 and 16. Fig. 12 illustrates the results when encryption-based KDT is used. We can observe that increasing LKH degree decreases the join rekey cost (eJoin(16) is 47% of eJoin (4)) while increasing the leave rekey cost (eLeave(16) is 135% of eLevae(4)). Such result confirms our analysis that the use of higher degree LKH (more than 4) with encryption-based KDT is not practical. Fig. 13 illustrates the results when XORBP KDT is used. We can observe that increasing LKH degree decreases both the join and leave rekey costs (xJoin(16)/xLevae(16) is 66% of xJoin(4)/xLeave(4)). Such result confirms our analysis that increasing LKH degree with XORBP KDT decreases both join and leave rekey costs.

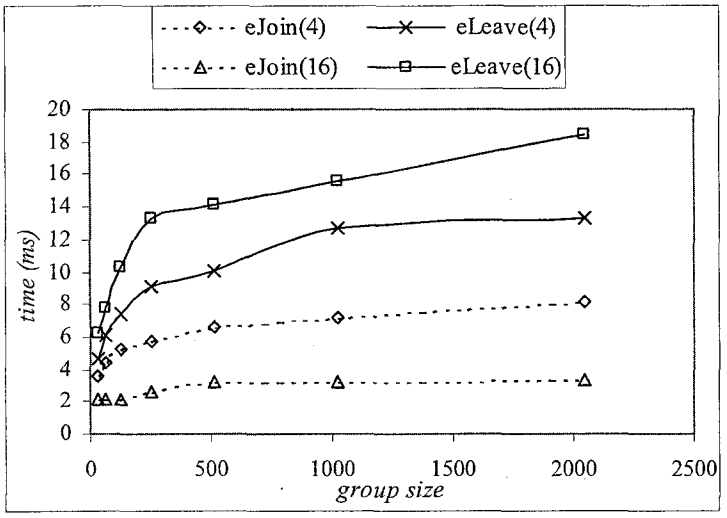


Fig. 12. Effect of LKH degree increase (d = 4 versus d = 16) on RM construction time when encryption-based KDT is used.

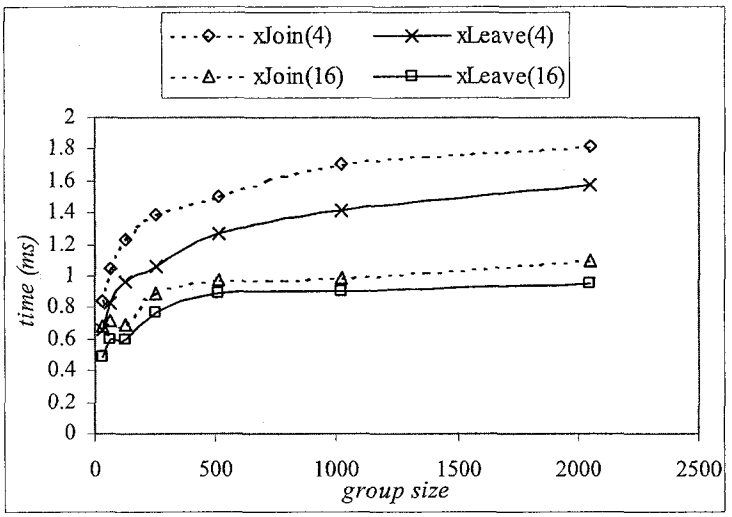


Fig. 13. Effect of LKH degree increase (d = 4 versus d = 16) on RM construction time when XORBP KDT is used.

3.7.3 Increasing Key Size

The third experiment shows the RM construction time saving for the same degree LKH when XORBP KDT is used versus encryption-based KDT for different encryption standards. LKH degree is 4, and the group size increases form 32 to 4096 in multiple of 2. Fig. 14 illustrates the results when DES encryption algorithm is used (extra bytes E = 127). We can observe that the use of XORBP KDT decreases both the join and leave rekey costs when compared to encryption-based KDT (xJoin is 23% of eJoin, and xLeave is 12% of eLeave). Fig. 15 illustrates the results when triple DES encryption algorithm is used (extra bytes E = 116). Triple DES key size is 3 times DES key size and performing a triple DES encryption is more time consuming than DES. Similarly, the use of XORBP reduces the rekey cost (xJoin is 40% of eJoin, and xLeave is 20 of % eLeave). Note that Fig. 14 and Fig. 15 demonstrate XORBP KDT symmetric rekey cost for both join and leave rekey cases, and the un-symmetry of the encryption-based KDT. Comparing RM construction time saving when DES is used versus tripe DES, we can observe that the time saving of XORBP is increased when used with smaller key size encryption protocol. When DES is used xJoin is 23% of eJoin while when triple DES is used xJoin is 40% of eJoin (i.e. when DES is used join RM construction time saving achieves 77%, while if triple DES is used the saving is reduced to 60%). Similarly, when DES is used leave RM construction time saving achieves 87%, while if triple DES is used the saving is reduced to 80%. Such saving is because larger key size introduces more random number generation for larger byte patterns and rekey packets' filling bytes. Random number generation is an expensive operation in terms of computation time but not as much as encryption.

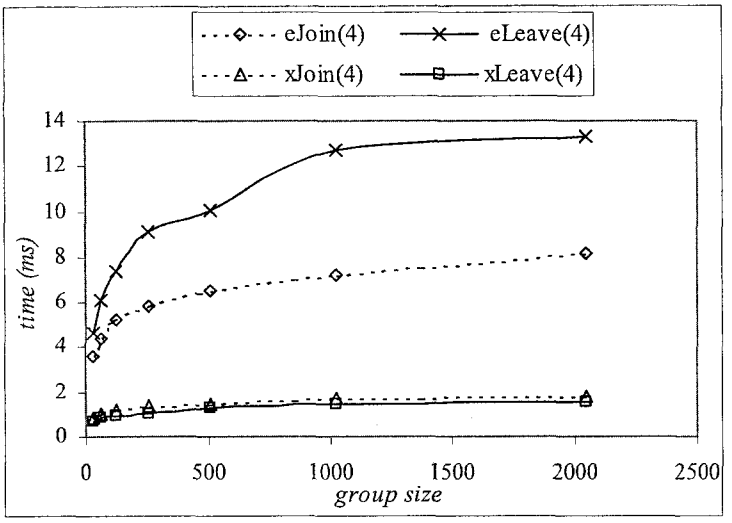


Fig. 14. Comparison of RM construction time when used with DES encryption-based versus XORBP KDTs.

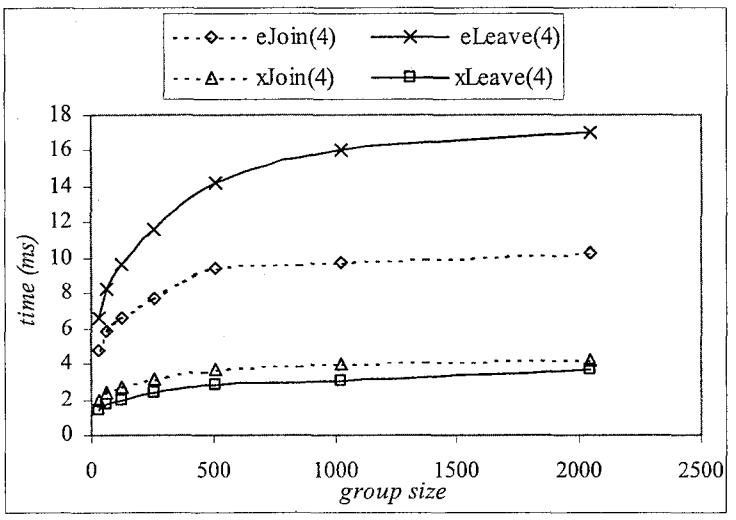


Fig. 15. Comparison of RM construction time when used with triple DES encryption-based versus XORBP KDTs.

3.7.4 Secure Random Number Generation

The use of XORBP introduces extra random number generation (section 3.6.3). The key generation in the above experiments is performed using *javax.crypto.KeyGenerator* class, while other random numbers and bytes are generated using *java.util.Random* class [62]. In addition, the above experiments perform un-optimized random byte generation, i.e. when a rekey packet is instantiated it is filled with newly generated random bytes then some of those bytes are overwritten with XORed keys (those bytes shouldn't be generated in the first place). Moreover, when encoding a BP, an unnecessary extra random byte is usually generated to augment the rest of the unused byte of the encoded BP.

Secure random number generation is more expensive than the usual (un-secure) random number generation. This experiment is performed using *java.security.SecureRandom* class that uses "SAH1PRNG" algorithm instead of *java.util.Random* class [62]. The same experiment as in section 3.7.3 for DES is repeated with the new random generation class while key generation uses the same *javax.crypto.KeyGenerator* class. The same code that performs un-optimized random byte generation is used. From our experiments, it is estimated that *SecureRandom* generation consumes 2.5 times the time of the same *Random* generation.

The experiment shows RM construction time saving for the same degree LKH when XORBP KDT is used versus an encryption-based KDT. LKH degree is 4, and the group size increases form 32 to 4096 in multiple of 2. Fig. 16 illustrates the results when DES encryption algorithm is used (extra bytes $E = 116$). Similarly, the use *SecureRandom* with XORBP KDT decreases both the join and leave rekey costs versus encryption-based KDT (xJoin is 56% of eJoin, and xLeave is 31% of eLeave). Comparing the saving with the results shown in Fig. 14, the join RM construction time saving is reduced from 77% using *Random* to 44% using *SecureRandom*. The leave RM construction time saving is reduced from 87% using *Random* to 69% using *SecureRandom*.

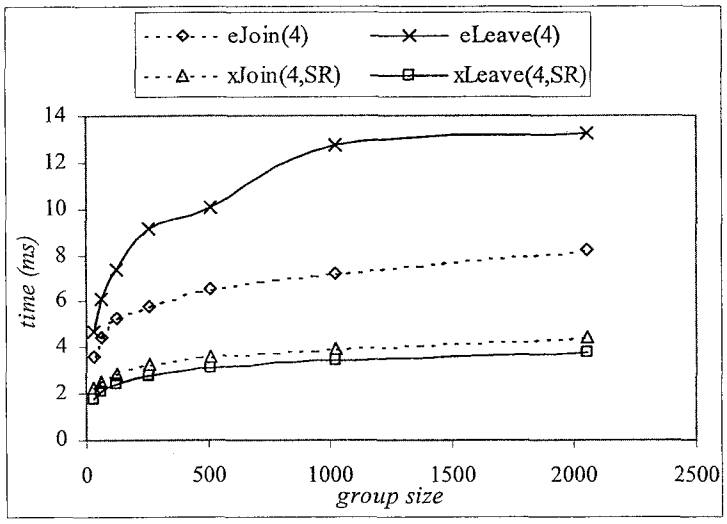


Fig. 16. Comparison of RM construction time when used with DES encryption-based KDT versus XORBP KDT that uses secure random number generation.

3.7.5 Estimated and Measured Costs

This experiment compares the estimated and measured LKH height for different LKH degrees and group size n = 4096. The LKH degree is increased from 4 to 32 by step 4 (i.e., 4, 6, 12, ..., 32), and the rekey manager uses XORBP KDT. For every LKH degree, n members are added and the LKH height and the number of allocated nodes are recorded. As previously mentioned, the experiments in this chapter adopt unbalanced LKH maintenance algorithms. The average of 500 readings is plotted.

Fig. 17 shows that the measured LKH height is usually larger than the estimated height for smaller LKH degrees, and almost the same for larger LKH degrees. Fig. 18 and Fig. 19 illustrate the difference between the required member storage (MS) and rekey message size (RMS) respectively using the measured and estimated LKH heights. Similar to the height, usually the measured MS and RMS have slight increase from the estimated values. Fig. 20 illustrates the difference between the required LKH storage (LKHS) of the measured and estimated values. Unlike MS and RMS, there is in the average 60%

increase in the measured LKHS over the estimated LKHS. Such increase is due to the use of unbalanced LKH maintenance algorithms. Such increase is expected to get higher with either a group size or group dynamics increases. Group dynamics determines the join and leave patterns.

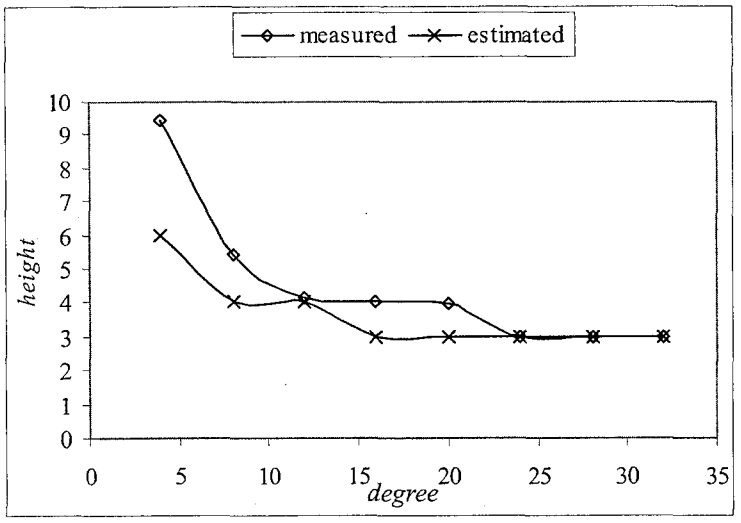


Fig. 17. Comparison of measured and estimated LKH height for a group of size $n = 4096$.

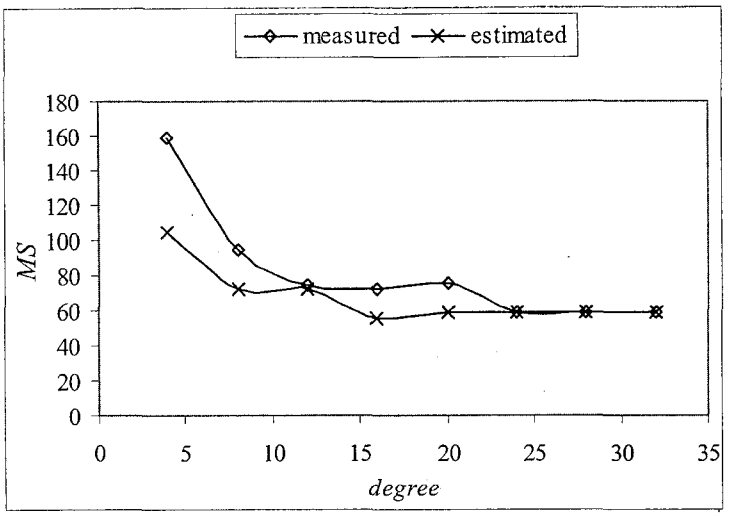


Fig. 18. Comparison of measured and estimated member storage (MS) for a group of size $n = 4096$.

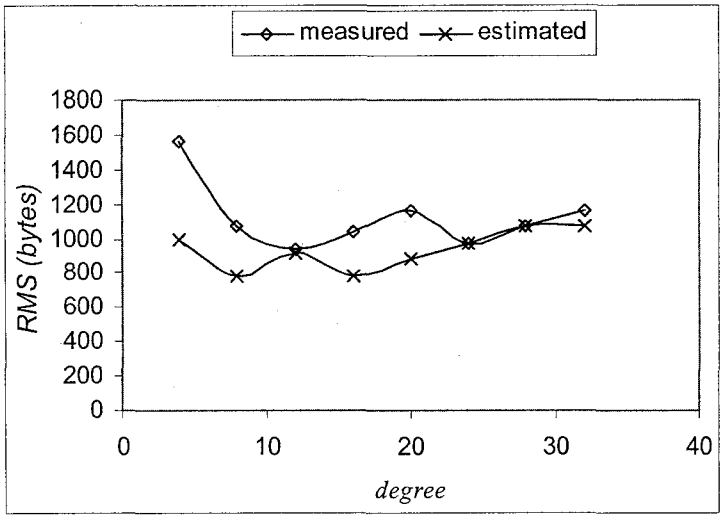


Fig. 19. Comparison of measured and estimated rekey message size (RMS) for a group of size.$n = 4096$.

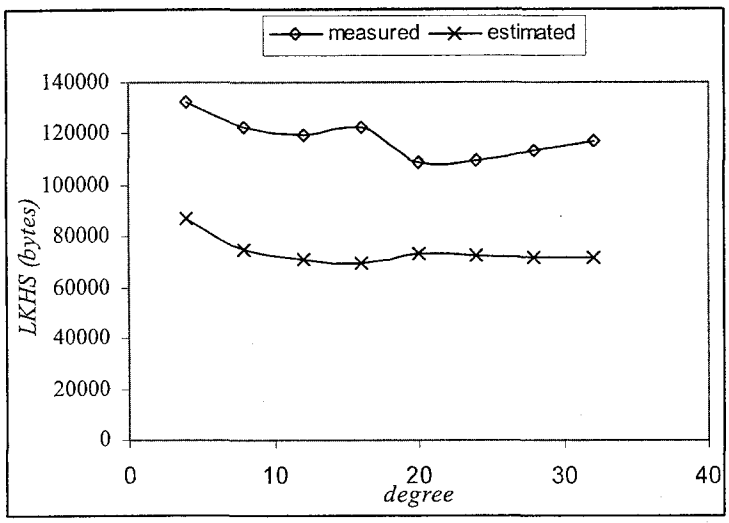


Fig. 20. Comparison of measured and estimated LKH storage (LKHS) for a group of size $n = 4096$.

3.8 Conclusion

In this chapter, we introduced a software model for secure group key management, where the main components along with their functionalities and interactions were identified. Concentrating on secure group key management, we highlighted two traditional rekey manager approaches for group rekeying, namely star and logical key hierarchy (LKH). The star key management approach is a simple approach that doesn't provide scalable leave rekeying since the leave rekey cost increases linearly with the group size. The LKH approach provided a scalable join and leave rekeying. Using the LKH approach, both join and leave rekeyings scales linearly with the logarithm of the group size. On the other hand, the LKH approach has un-symmetric rekeying procedures for join and leave cases and doesn't scale well with LKH degree increase. The original LKH key distribution technique (KDT) for a newly generated key in a rekey message is to encrypt a new key either with another key or with its previous version (encryption-based KDT).

We introduced XORBP, a new KDT that can be used with the LKH approach. XORBP performs a simple XOR operation between keys instead of encryption and distributes the key material in random byte patterns (BPs) in a fixed size rekey packet for every new key. The rekey message contains a rekey packet for every new key that is targeted to a set of group members that should hold that key. The use of XORBP provided symmetric rekey procedures for join and leave rekeyings. In addition, it substantially reduces the rekey time. On the other hand, the use of XORBP increases the required LKH storage, member storage, and the rekey message size. In addition, XORBP introduces extra random number generation when compared with encryption-based KDT.

We derived analytical cost estimates of XORBP KDT, and performed empirical experiments to compare its performance versus encryption-based KDT. Our experiments show that, increasing LKH degree when used with encryption-based KDT increases the un-symmetry of join and leave rekey costs, which makes the use of an LKH degree greater than 4 not practical. Using XORBP as KDT and increasing LKH degree allows the decrease of join and leave rekey costs. Using XORBP KDT versus encryption-based KDT, with the same degree LKH, can achieve 90% savings in the rekey message construction time. Using XORBP KDT with higher degree LKH (compared to lower degree LKH) provides extra savings in all cost metrics: storage, time, and communication. Finally, our experiments, using unbalanced LKH maintenance algorithms, show that there exists a slight increase in the measured LKH height, member storage, and rekey message size over the estimated values. On the other hand, the experiments show that the measured LKH storage for small group size has a 60% increase over the estimated value. Such undesirable increase motivates us to develop balanced LKH maintenance algorithms and protocols as explained in chapter IV.

CHAPTER IV

LOGICAL KEY HIERARCHY REKEY PROTOCOLS

As previously mentioned in chapter III, for secure group key management, there exists a (central) rekey manager that maintains a logical key hierarchy (LKH) for scalable rekeying (change of group key, *GK*, due to either new group member addition or group member removal). The rekey manager sends a rekey message (RM) to all group members for every group rekeying. The rekey message contains a rekey packet for every new LKH key. The rekey client is the group member component that maintains a set of LKH keys (including *GK*), and receives and process RMs for such keys update.

In this chapter, we propose two techniques for a rekey manager to maintain a LKH, and the associated rekey protocols. One technique adopts an unbalanced LKH (denoted S-LKH) while the other adopts a balanced LKH (denoted B^+-LKH). We detail the LKH node structure, and the RM format and construction for all scenarios of LKH node insertion and deletion. In addition, we present the rekey client processing for different RM types. We performed empirical experiments to compare the rekey performance of S-LKH protocol versus B^+-LKH protocol for different group sizes and LKH degrees. The B^+-LKH protocol causes a small increase in the average number of rekey packets, and the average number of encrypted keys in a RM when compared to the S-LKH protocol. However, in chapter V we show that introducing batch rekeying (rekeying for several members addition and/or removal) results in a reduction in the B^+-LKH case. On the other hand, the use of B^+-LKH decreases LKH height and the maximum number of encrypted keys in a RM when compared to S-LKH. The expected maximum rekey time is used in adjusting the minimum inter-rekey period that has to be elapsed between two consecutive rekeyings. Moreover, the use of B^+-LKH reduces the number of allocated nodes for a LKH (up to 50% reduction) when compared to S-LKH.

This chapter is organized as follows. Section 4.1 presents motivation and overview of the new techniques and protocols. Section 4.2 presents S-LKH node structure, RM format, and S-LKH maintenance algorithms along with RM construction. Section 4.3

presents B^+-LKH maintenance algorithms along with RM construction, and algorithms analysis. Section 4.4 details B^+-LKH rekey client processing when receiving a RM to update the maintained set of keys. Section 4.5 presents performance evaluation experiments and results. Finally, section 4.6 concludes the chapter.

4.1 Motivation and Overview

A LKH is maintained by a rekey manager to provide scalable rekeying. A balanced LKH is a key tree where all leaf nodes are at the same distance (level) from the root. Keeping a LKH balanced is very important to the performance of group rekeying especially for highly dynamic groups (many joins and leaves). Several researchers assume a balanced LKH when estimating and analyzing the cost of group rekeying [11], [67]. Keeping a LKH balanced is a crucial issue. However, the literature lacks practical LKH maintenance algorithms as well as algorithms for keeping a LKH of any degree balanced all the time [51] [52]. As concluded in chapter III, when an unbalanced LKH is used, there is always an increase in the measured LKH height over the estimated value. The increase in LKH height leads to a small increase in member storage, and rekey message size over the estimated values. Nevertheless, there is a substantial increase in the allocated LKH storage over the estimated value (the increase achieves 60% as shown in section 3.7.5).

The proposed LKH maintenance algorithms require the rekey manager to assign a unique identification for every group member, namely individual ID. For example, an individual ID could be a randomly generated number. Individual IDs are used in constructing the LKH and are sent in a RM to guide its processing (by a rekey client). Using LKH keys or true member identification (such as name or IP address) as IDs makes the rekey protocol vulnerable to traffic analysis. Since individual IDs are part of a RM, true IDs can be used to reveal the LKH structure and group members information.

Our proposed LKH maintenance techniques provide a dual LKH purpose, as a key tree and as an easily searchable data structure for individual material (ID, key, ...etc). The first proposed technique maintains a LKH as a search tree [63], denoted S-LKH, using individual IDs as searched values. A search tree is not balanced and is used to provide sort and search algorithms for a set of searched values. In a search tree, any value

is located only once at any tree node (internal or leaf). We adapt the traditional search tree algorithms to accommodate the constraint that a group member individual material (ID, key, ...etc) is always an entry in a leaf node. The S-LKH internal nodes contain key-encrypting-keys (KEKs). We detail S-LKH node structure and maintenance algorithms that show how a S-LKH grows (shrinks) when an individual entry is inserted (deleted) into a leaf node. In addition, the algorithms show how a RM is constructed for different insertion and deletion scenarios. The S-LKH maintenance algorithms are applicable for any LKH of degree $d \geq 2$.

The second proposed technique maintains LKH as a balanced B^+ search tree [63], denoted B^+-LKH, that has the same structure as S-LKH but guarantees that a LKH is balanced after every node insertion or deletion. B^+ search trees have an extra constraint that all allocated nodes have to be at least half full to reduce the required LKH storage (allocated memory space). B^+-LKH maintenance introduces complexity and extra overhead in RM construction and in the rekey client processing. We detail B^+-LKH maintenance algorithms along with RM construction for different insertion and deletion cases. In addition, we detail the rekey client RM processing (for key updates) for different RM types.

4.2 S-LKH: A LKH as a Search Tree

In a binary search tree, each node N contains a single search value v and points to two sub-trees (children). The left sub-tree (child) contains all the search values in the tree rooted at N that are less than or equal to v, and the right sub-tree (child) contains all the search values in the tree rooted at N that are greater than v. A multi-way search tree of degree d is a general tree in which each node has d or fewer children (sub-trees) and contains one fewer search values than it has children. That is, if a node has four children, it contains three search values. The search tree is constructed such that the search values are sorted in an ascending order in each node. In addition, the searched values are sorted across all nodes.

A rekey manager that maintains LKH as a S-LKH is required to provide a unique individual identification, ID, for every new member. S-LKH is constructed as a search tree for those individual IDs. An individual ID can be a newly generated random number.

Using LKH keys as sort/search values will reduce an insider attack search space. For example, colluding group members can specify a smaller search space for LKH keys by revealing their keys and positions to each other. Individual IDs are sent in a RM to guide the rekey client processing. Using true member identification such as name or IP address as an individual ID makes the protocol vulnerable to traffic analysis. Generating IDs as random numbers prevents both the insider attack and the traffic analysis problems.

Similar to a search tree, a S-LKH internal node has at least one child, while a S-LKH leaf node has no children. The proposed S-LKH maintenance algorithms adapt the traditional search tree algorithms to the constraint that an individual material (ID, key, ...etc) is always an entry in a leaf node. Consequently all searched IDs are entries in leaf nodes while the internal nodes contain replicas of certain IDs that are used as an index to guide the search for leaf entries' IDs.

4.2.1 S-LKH Node Structure

In a S-LKH of degree d, the node size e is the number of entries in a node such that $1 \le e \le d$. The leaf node structures is $[(K_1, ID_1),(K_2, ID_2),...,(K_e, ID_e)]$, where the pair (K_i, ID_i) is an individual entry that contains an indiviudal key K_i and an indiviudal ID ID_i among other individual information such as name, IP address,..etc (not shown). The individual IDs are the sorted/searched values used in constructing S-LKH and are unique through all leaf nodes. The internal node structure is $[(K_1, P_1), ID_1, (K_2, P_2),..., ID_{e-1}, (K_e, P_e)]$, where the pair (K_i, P_i) is a child node entry in which K_i is a KEK and P_i is a pointer to the (internal or leaf) child node. The internal nodes' IDs are replicas of certain leaf IDs and are choosen to guide the search.

A leaf node entry insertion requires a pair (K_i, ID_i) of the individul key and ID in addition to other individual material (not illustrated). While, an internal node entry insertion requires an ID, except for the first insertion, and a child node. An internal node entry is created to contain the pair (K_i, P_i), where K_i is a newly generated KEK and P_i is a pointer to the child node. Internal node IDs are inserted between childern entries as shown in the internal node structure above. A S-LKH is constructed such that for every internal node, the first entry P_1 points to a child node whose every ID_i entry $ID_i \le ID_1$,

the last entry P_e points to a child node whose every ID_i entry $ID_{e-1} < ID_i$, and every other P_j points to a child node whose every ID_i entry $ID_{j-1} < ID_i \leq ID_j$. In addition, all entries of a leaf node are sorted in ascending order by their IDs. Fig. 21 illustrates a S-LKH structure maintained by a rekey manager. The rekey manager maintains two entities: the group key *GK* and *root* the pointer to the S-LKH root node.

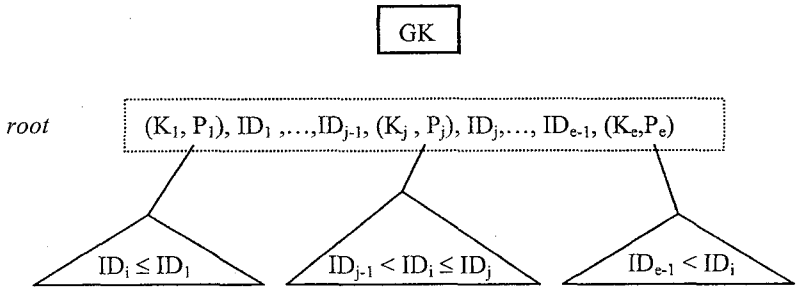
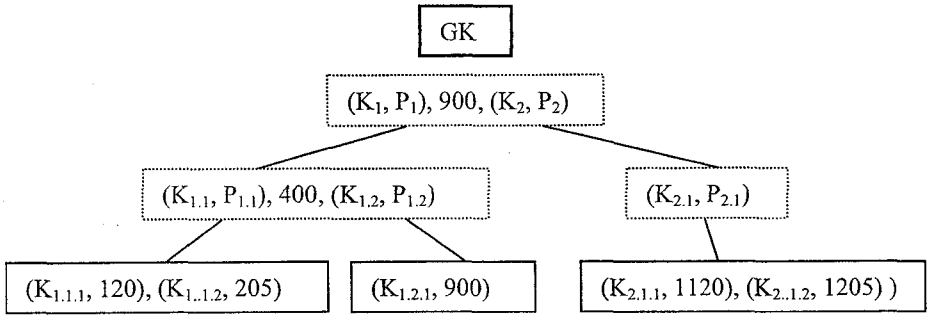


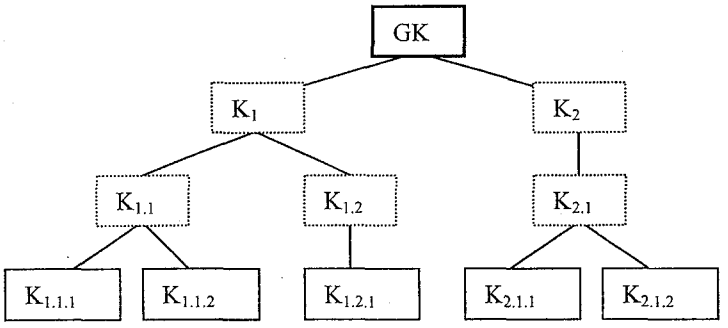
Fig. 21. A S-LKH structure.

A S-LKH provides dual purpose as a key tree and as an easily searched data structure for individual material. A S-LKH has two views, the *key view* that shows the corresponding key tree (LKH), and the *search view* that shows the search tree for individual IDs. For example, Fig. 22(a) is a S-LKH of degree $d = 2$ and height $h = 3$ for a group of size $n = 5$; Fig. 22(b) is the S-LKH key view, and Fig. 22(c) is the S-LKH search view.

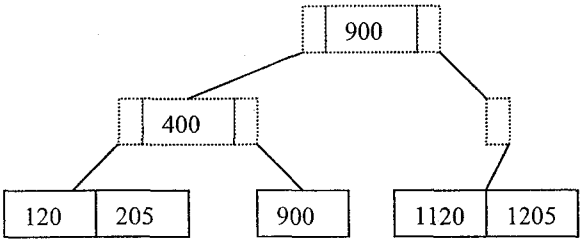
When S-LKH is used with XORBP key distribution technique (KDT) (chapter III), every key entry in an internal node or in a leaf node is associated with a byte pattern (BP). The BP will be allocated (generated) when the entry is first inserted. In the remainder of this chapter, we assume the use of encryption-based KDT, and explain the changes, if any, when XORBP is used.



(a) The S-LKH nodes.



(b) The S-LKH key view.



(c) The S-LKH search view.

Fig. 22. A S-LKH of degree $d = 2$ and height $h = 3$ for a group of size $n = 5$.

4.2.2 S-LKH Rekey Message Format

Fig. 23 depicts the rekey message RM format used by a S-LKH rekey manager. Fig. 23(a) illustrates the initial key message sent to a group member before receiving any RMs and is used to initialize his state (ID, position, LKH height, and LKH degree). Where ID is the member unique identification assigned by the rekey manager, and *position* is an encoded LKH position of the individual leaf entry. The *individual BP* is sent only if XORBP is used as a KDT. Fig. 23(b) illustrates the RM format, which is sent to all group members for every rekeying, where *SEQ* is a sequential number that indicates RM number starting from 1 for the first message, *type* is the message type that could be ADD if the rekey is due to new member addition or REMOVE if the rekey is due to group member removal (other types will be introduced later when the algorithms are presented), *position* is the encoded LKH position of inserted/deleted leaf node entry, *level* specifies the distance between the root node and the effected leaf node, ID is the inserted/deleted leaf entry ID, and a *RekeyPacket* is constructed for every new key.

If an encryption-base KDT is used, the *RekeyPacket*, shown in Fig. 23(c), contains several encryptions of a new key (*encKey*). Each *encKey* is targeted to a different set of group members. On the other hand, if a XORBP KDT is used, the *RekeyPacket*, shown in Fig. 23(d), contains a fixed length of bytes (size is *S* bytes as estimated in section 3.3.3) and an encoded BP *encodedBP* for the associated BP as explained in section 3.4.2. Note that, *GK* is not associated with a BP and a rekey packet for *GK* doesn't contain an encoded BP.

ID	Position	LKH height	LKH degree	Individual BP

(a) Individual (initial) key message.

SEQ	Type	Position	Level	ID
$RekeyPacket_1$, $RekeyPacket_2$, ...				

(b) Rekey Message (RM).

$encKey_1$, $encKey_2$, ...

(c) Encryption rekey packet.

S bytes	encodedBP

(d) XORBP rekey packet.

Fig. 23. The format of messages used by a S-LKH rekey manager.

4.2.3 Rekey Packet Construction

For an internal node entry (K_A, P_A) in an internal node N, there are two types of constructed rekey packets for a newly generated $K_A^{'}$. The first type is *addRekey* packet that is constructed after an insertion of an entry to the internal/leaf node A, where node A is the child of N pointed to by P_A (node A for *GK* is *root* node). The second type is

rmvRekey packet that is constructed after the deletion of an entry from node A. The rekey packets are constructed by calling the methods *addRekey*(A) and *rmvRekey*(A) provided by every internal node N (and by *GK*). Note that the inserted/deleted entry could be directly in node A or indirectly in the path of one of its children.

If an encryption-based KDT is used, the *addRekey* packet contains two encryptions of $K_A^{'}$ $[\{K_A^{'}\}K_A, \{K_A^{'}\}K_B]$ where K_B is the new entry key in node A. The *rmvRekey* packet contains e enctryptions of $K_A^{'}$ where e is the number of entries in node A $[\{K_A^{'}\}K_i, \forall K_i \in A]$.

Note that for the first time a newly created key is distributed, a *rmvRekey* has to be constructed since no previous version of the key exists. In addition, if an operation performs both insertion and deletion to node A, a *rmvRekey* packet has to be constructed for $K_A^{'}$ (the key previous version can not be used since some entries are deleted).

If XORBP KDT is used, an internal node entry (K_A, BP_A, P_A) contains a BP that has to be regenerated along with the key $K_A^{'}$. Both *addRekey* and *rmbRekey* packets construction is symmetric and uses node A entries as described in section 3.4.1. Every XORBP rekey packets, except for *GK*, contains an *encodedBP* for $BP_A^{'}$ using $K_A^{'}$ as descibed in section 3.4.2.

4.2.4 S-LKH Algorithm for New Group Member Addition

Fig. 24 is the S-LKH new member addition algorithm, *AddMember*, where the new group member has a unique identification *memberID* and an individual key *memberKey*. The algorithm details how the S-LKH of degree *d* rooted at node *root* is growing while adding the new member entry as well as how the individual key message *initMsg* and the RM *rekeyMsg* are constructed for the different addition cases (RM type). There are three possible RM (*rekeyMsg*) types ADD, SPLIT, and INCREASE as will be explained next.

Initially, the S-LKH rooted at *root* node is searched by *memberID* for the appropriate *position* in a leaf node N for the new member entry. The *lookup* method searches the S-LKH rooted at *root* node guided by *memberID* and returns the appropriate *position* for the individual entry to be inserted, in addition, it returns all visited nodes in *nodeStack*, (where the first pushed node is *root* and the last pushed node is the leaf node that should

contain the new individual entry). Then, the new entry is inserted where there are three cases. The first case occurs if the leaf node N has space for the new entry (number of entries less than *d*), a simple insert is performed and *rekeyMsg* type is set to ADD[6]. Note that, if XORBP KDT is used the *individualBP* filed in *initMsg* message is assigned after the leaf node insertion is performed (Fig. 23 (a)).

The other two cases occur if the leaf node N is full (has *d* entries). If N is full, a new leaf node *newNode* is allocated and N entries (including the new one) are split equally between the two nodes (N and *newNode*). If the number of entries (*d* + 1) doesn't split equally between the two nodes (odd number), we keep one more entry in N than *newNode*. The *newNode* is to be the right neighbor of N. The *splitInsert* method returns an ID that is the maximum ID value in node N after the split. An internal node entry (KEK and pointer) that points to *newNode* should be inserted in the parent of N (to the right of N entry). There are two cases for that insertion according to whether the parent node is full or not.

The second addition case occurs when the parent of N has space for a new entry, the *newNode* entry is inserted and *rekeyMsg* type is set to SPLIT. The third addition case occurs if the parent of N is full, a new internal node *newParent* is allocated to be the parent for the two children N and *newNode*. The pointer at the parent node that was pointing to N should be replaced to point to *newParent* instead and *rekeyMsg* type is set to INCREASE. The last case leads to an increase of S-LKH height only if the distance between *root* and N (denoted *level* in the algorithm) equals to (*h*-1). Note that, the underlined code highlights the assignment of the constructed rekey packets to *rekeyMsg* fields. Also note that, *rmvRekey* packets are constructed for the newly created KEKs and for KEKs that experience deletion in the associated node.

The number of rekey packets in *rekeyMsg* is (*level*+1), (*level*+2), and (*level*+3) in the cases of ADD, SPLIT, and INCREASE, respectively. The first two packets in the cases of SPLIT & INCREASE are *rmvRekey* packets while all other packets are *addRekey* packets. Please see appendix A for examples of the different new group member addition cases.

---

[6] ← denotes an assignment to multiple fields in a message.

```
Method AddMember(memberID, memberKey)
Globals: root, h, d, GK;
Returns: initMsg, rekeyMsg;

if (h equals 0) then { root = AllocateNew LeafNode(); h = 1; }
(position, nodeStack) = root.lookup (memberID);
level = nodeStack.size() –1; N = nodeStack.pop();
initMsg ← (memberID, position, h, d);
rekeyMsg ← (position, memberID, level);
if (N.size() < d) then { N.insert (memberKey, memberID); rekeyMsg.type = ADD; }
 else { newNode = AllocateNew LeafNode();
        ID = N.splitInsert(memberKey, memberID, newNode);
        parent = nodeStack.pop();
        if ((level > 0) and (parent.size() < d))
         then { parent.insert(ID, newNode); decrement level;
                rekeyMsg.type = SPLIT;
                rekeyMsg← (parent.rmvRekey(N), parent.rmvRekey(newNode)); }
         else
         { newParent = AllocateNew InternalNode();
          newParent.insert(null, N); newParent.insert(ID, newNode);
          rekeyMsg.type = INCREASE;
          rekeyMsg←(newParent.rmvRekey(N), newParent.rmvRekey(newNode));
          if (level equals (h-1)) then increment h;
          if (level equals 0) then root = newParent;
              else parent.replace(N, newParent); }
      }
for (i = 0 to (level-1))
   { prevN = N; N = nodeStack.pop();
     rekeyMsg ←N.addRekey(prevN); }
rekeyMsg ← GK.addRekey(root);
return initMsg, rekeyMsg;
```

Fig. 24. The S-LKH new group member addition and RM construction algorithm.

4.2.5 S-LKH Algorithm for Group Member Removal

Fig. 25 is the S-LKH group member removal algorithm, *RemoveMember*, that details how the S-LKH rooted at node *root* is shrinking after the removal of a group member entry as well as how the RM (*rekeyMsg*) is constructed for the different removal cases (RM type). There are two possible RM *rekeyMsg* types REMOVE, and DECREASE. The removed member is identified by his unique *memberID*.

Initially, the S-LKH rooted at node *root* is searched by *memberID* to determine the *position* of the leaf entry at node N to be deleted. The first removal case occurs when the leaf node N, after the deletion, contains one or more entries, the *rekeyMsg* type is set to REMOVE. The second case occurs when node N, after the deletion, has no more entries. In this case, node N entry (KEK and pointer) has to be deleted from its parent node. If the parent after the deletion has no more entries, its entry has to be deleted from its parent, and so on. The deletion could propagate to upper nodes and stops when it reaches the first non-empty node. The *rekeyMsg* type is this case is set to DECREASE and could lead to the decrease of LKH height h if it the deleted leaf node is the only node that has distance equals to (h-1) from the root. The height h might be decreased by more than one if more nodes are deleted. The number of rekey packets in *rekeyMsg* is (*level* + 1). Please see appendix A for examples of the different group member removal cases.

```
Method RemoveMember(memberID)
Globals: root, h, d, GK;
Returns: rekeyMsg;

(position, nodeStack) = root.lookup (memberID);
rekeyMsg ← (position, memberID);
level = nodeStack.size() –1;
N = nodeStack.pop(); N.delete(memberID);
if (N.size() > 0) then rekeyMsg.type = REMOVE;
 else { while (N.size() equlas 0)
          if (level equals 0) then { decrement level; free root; h = 0; breakWhile; }
            else { decrement level; prevN = N; N = nodeStack.pop();
                   N.delete(prevN); }
        h = root.getHeight();
        rekeyMsg.type = DECREASE;
      }
rekeyMsg.level = level;
for (i = 0 to (level-1))
 { prevN = N; N = nodeStack.pop();
   rekeyMsg ←N.rmvRekey(prevN); }
if (root does-not-equal null) then rekeyMsg ← GK.rmvRekey(root);
return rekeyMsg;
```

Fig. 25. The S-LKH group member removal and RM construction algorithm.

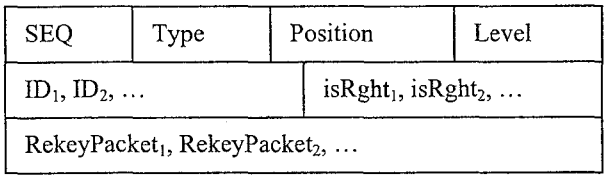
4.3 B^+-LKH: A LKH as a B^+ Search Tree

A B^+-LKH rekey manager maintains a balanced LKH adapting B^+ search tree insertion and deletion algorithms [63], [38]. A B^+-LKH is a S-LKH that has the same node structure shown in Fig. 21. A B^+ search tree of degree d is subject to two constraints, the first is all its leaf nodes are on the same level (i.e. balanced), and the second is all allocated nodes except the root are at least half full. The root node size is at least 2, while all other nodes' sizes are at least $\lfloor d/2 \rfloor$ that will be denoted *Min_d*.

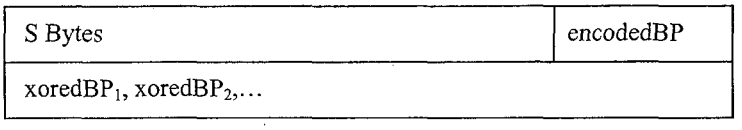
Maintaing a B^+-LKH introduces complexity and extra overhead in RM construction as well as in the rekey client processing. B^+-LKH algorithms are suitable for any LKH of degree *d* greater than or equal to 4. When *d* equals 2 or 3 *Min_d* is 1 (and so is S-LKH) and using B^+-LKH algorithms introduces exta overhead versus S-LKH.

4.3.1 B^+-LKH Rekey Message Format

The initial key message and many fields in RM are similar to the messages explained in section 4.2.2 for S-LKH protococl. Fig. 26 illustrates the changes to the messages format used by a B^+-LKH rekey manager. Fig. 26(a) is RM format that contains several IDs, and several boolean (bit) values *isRght*, where *isRght* is a Boolean value that indicates either "is right" or "is left" that is used with some message types as will be explained later when introducing the B^+-LKH *RemoveMember* algorithm. Fig. 26(b) is a XORBP rekey packet that contains several *xoredBP*s. A *xoredBP* is constructed with two same (bit) length BPs XORed, and is used with some message types as will be expaliend later when introucing the B^+-LKH *RemoveMember* algorithm.



(a) Rekey Message (RM).



(b) XORBP rekey packet.

Fig. 26. The format of messages used by a B^+-LKH rekey manager.

4.3.2 B$^+$-LKH Rekey Packet Construction

B$^+$-LKH algorithms use the same rekey packet construction introduced in section 4.2.3 for S-LKH protocol. In addition, there are two remove related operations to uphold the second B$^+$ search tree constraint that all nodes are at least half full. The first operation is *shift*, in which one entry is shifted from a node to one of its neighboring nodes. The second operation is *merge*, in which all entries in an underflow node (its size becomes less than *Min_d*) are merged (moved) to one of its neighboring nodes and the empty node is deleted.

A new rekey packet construction is needed for the *merge* operation and is called *mrgRekey*(A, *isRight*), where *isRight* is a boolean value if "*true*" that means A is the right neighbor of the deleted node and if "*false*" that means A is the left neighbor of the deleted node. Similar to *addRekey* and *rmvRekey* packet construction methods, *mrgRekey* is provided by the internal node N that contains the entry (K_A, P_A) for its child node A. The encryption-based rekey packet for the new $K_A^{'}$ contains *Min_d* encrypted key $[\{K_A^{'}\}K_A, \{K_A^{'}\}K_i, 1 \le i < Min\_d]$ where K_i is a merged entry key, and *isRight* determines which keys are merged. If *isRight* equals to *true*, the first *Min_d* entries are merged from the left neigbor node, and if equals to *false* the last *Min_d* entries are merged from the right neighbor node.

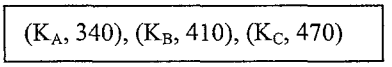
The XORBP rekey packets are constructed the same way for all packet types (*addRekey*, *rmvRekey* and *mrgRekey*). If XORBP is used as a KDT technique the shifted/merged entries' BP is subject to change due to the possible occupation of the assigned bytes. When an entry is shifted/merged a new BP is allocated. The new bit represented BP has to be sent in the rekey packet XORed with its bit represented previous value as a *xoredBP* illustrated in section 4.3.1. The rekey packet contains one *xoredBP* if there is a shifted entry to node A, and contains (*Min_d* - 1) *xoredBP*s if there are (*Min_d* - 1) merged entries to node A.

4.3.3 B$^+$-LKH Algorithm for New Group Member Addition

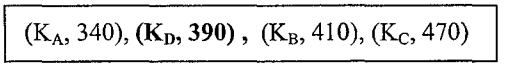
Adding a new group member leads to the insertion of a new entry in a leaf node, and might lead to insertions in one or more internal nodes. First we will present the different

insert operations in a leaf node and in an internal node, followed by the member's addition and RM construction algorithm (*AddMember*).

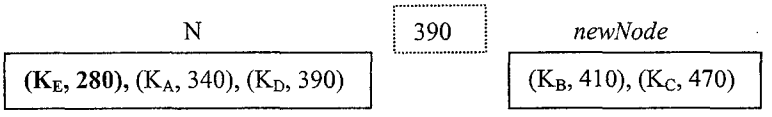
For a leaf node there exists two possible insert operations namely *insert* and *splitInsert*. Fig. 27 is an example that illustrates a leaf node N in a LKH of degree $d = 4$ ($Min\_d = 2$) after the two insert operations. Every leaf entry represents a member individual key and his unique ID. Fig. 27(a) shows the original leaf node N that has 3 entries. Fig. 27(b) shows the leaf node N after *insert* (K_D, 390) is performed (N contains maximum number of entries 4). Fig. 27(c) shows the leaf node N after *splitInsert* (K_E, 280) is performed. A new empty leaf node *newNode* is allocated and passed to this method call, and an ID is returned that will be inserted in an internal node in the upper level. Note that the entries are sorted by their IDs, and the last ID in N is returned after moving half of its entries to *newNode*. The two leaf nodes contain at least *Min_d* entries.



(a) Original leaf node N.



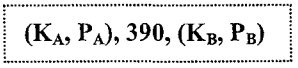
(b) Leaf node N after *insert* (K_D, 390)



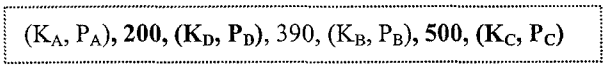
(c) Leaf nodes N and *newNode*, and the returned ID after *splitInsert* (K_E, 280).

Fig. 27. An example of different leaf node insertions in a B^+-LKH of degree $d = 4$.

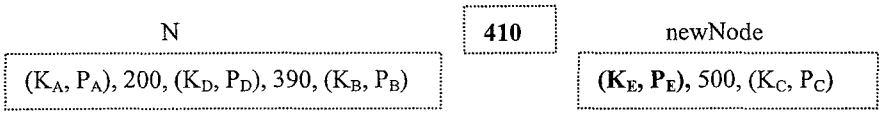
For an internal node, there are three possible insert operations namely *firstInsert*, *insert*, and *splitInsert*. Fig. 28 is an example that illustrates the three insert operations in an internal node N in a LKH of degree $d = 4$. The internal node insert operations are passed a child node (A, B, C, D, or E) and a pointer to these nodes is created in the entry. In addition, a newly created KEK is generated for every child node. The *firstInsert* operation, when the node is empty, is passed two child nodes. Fig. 28(a) shows the internal node N after *firstInsert*(A, 390, B) is performed. Fig. 28 (b) shows the internal node N after the *insert*(500, C) is performed, then *insert*(200, D) is performed, that makes the node full (has 4 entries). Fig. 28(c) shows the internal node N after *splitInsert*(410, E) is performed, where a new internal node *newNode* is passed to this method call and an ID is returned.



(a) Internal node N after *firstInsert* (A, 390, B).



(b) Internal node N after *insert* (500, C), then *insert* (200, D).



(c) Internal nodes N and *newNode*, and the returned ID after *splitInsert* (410, E).

Fig. 28. An example of different internal node insertions in a B^+-LKH of degree $d = 4$.

Fig. 29 is the B^+-LKH member addition and RM construction algorithm, *AddMember*, that details how the B^+-LKH rooted at node *root* is growing while adding new members entries as well as how the individual key message *initMsg* and RM *rekeyMsg* are constructed for different addition cases (RM type). There are three possible RM (*rekeyMsg*) types ADD, SPLIT, and INCREASE as will be explained next. The added member has a unique ID *memberID* and an individual key *memberKey*.

Initially, the B^+-LKH rooted at node *root* is searched by *memberID* for the appropriate *position* in a leaf node N for the new member entry. The first addition case occurs when the leaf node N has space for the new entry, a simple insertion is performed, and *rekeyMsg* type is set to ADD. The other two addition cases occur if the leaf node is full. If the leaf node is full a new leaf node *newNode* is allocated and the entries of N are split between the two nodes (N and *newNode*). An internal entry (KEK and pointer) has to be inserted for *newNode* at the parent of N and to its right. If the parent of N is not full a simple internal node insertion is performed. While if the parent node is full a new internal node is allocated and the entries of that parent are split between it and the new allocated node, and so on the split could propagate to upper levels. Note that, after *splitInsert* method is called the parent of nodes *prevN* and *prevNew* (denoted *prvNprnt* and *prvNwPrnt*, respectively) could be either *N* or *newNode*, and are assigned by that method call.

The second addition case occur when the split propagates until it reaches an internal node that has space for the new entry and the *rekeyMsg* type is set to SPLIT. The third addition case occurs when the split propagates to the root node leading to an increase of LKH height, and the *rekeyMsg* type is set to INCREASE. Please see appendix A for examples of the different new group member addition cases.

```
Method AddMember(memberID, memberKey)
Globals: root, h, d, GK;
Returns: intMsg, rekeyMsg;

if (h equals 0) then { root = AllocateNew LeafNode(); h = 1; }
(position, nodeStack) = root.lookup (memberID); level = h –1; N = nodeStack.pop();
initMsg ← (memberID, position, h, d); rekeyMsg ← (position, memberID);
if (N.size() < d) then { N.insert (memberKey, memberID); rekeyMsg.type = ADD; }
 else { done = false; newNode = AllocateNew LeafNode();
        ID = N.splitInsert(memberKey, memberID, newNode);
        while (level > 0)
        { decrement level; prevN = N; N = nodeStack.pop();
          if (N.size() < d)
            then { N.insert(ID, newNode); rekeyMsg.type = SPLIT;
                   rekeyMsg ← (level, ID, N.rmvRekey(prevN), N.rmvRekey(newNode));
                     done = true; breakWhile; }
            else { prevNew = newNode; newNode = new InternalNode();
                   rekeyMsg ←(ID);
                  (ID, prevNprnt, prevNewPrnt) = N.splitInsert(ID, prevNew, newNode);
                  rekeyMsg ← prevNprnt.rmvRekey(prevN);
                 rekeyMsg ← prevNewPrnt.rmvRekey(prevNew); }
        }
        if (not done )
          then { root = AllocateNew InternalNode(); root.firstInsert(N, ID, newNode);
                 rekeyMsg.type = INCREASE; increment h;
                  rekeyMsg ← (ID, root.rmvRekey(N), root.rmvRekey(newNode)); }
      }
for (i = 0 to (level-1))
 { prevN = N; N = nodeStack.pop();
  rekeyMsg ← N.addRekey(prevN); }
rekeyMsg ← GK.addRekey(root);
return initMsg, rekeyMsg;
```

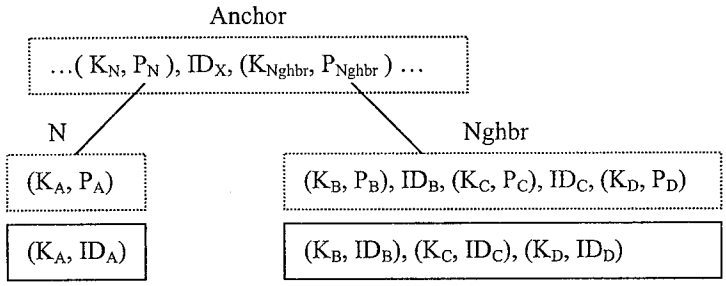
Fig. 29. The B^+-LKH new group member addition and RM construction algorithm.

4.3.4 B^+-LKH Algorithm for Group Member Removal

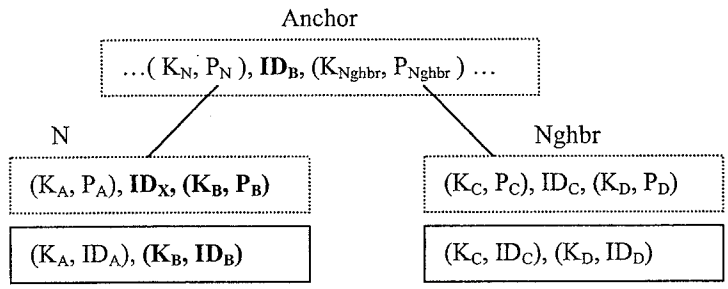
Removing a group member leads to the deletion of his entry from a leaf node and possibly the deletion of one or more internal node entries. The deletion of an entry could be simple that does not lead to the violation of not being half full or it could need extra overhead to uphold the B^+ constraint that all nodes are at least half full. Keeping the B^+-LKH balanced and keeping the nodes half full need two possible remove-related operations *shift* and *merge*, both operations apply to two neighboring siblings (of the same parent) nodes, N and its right or left neighbor *Nghbr*. The best neighbor for a node N (if the two exists) is the one with greater size (i.e. has more entries). If the two sibling neighbors have the same size, the right one is chosen. Note that, the first child of a node has only a right sibling, while the last child of a node has only a left sibling and the only neighbor is the best neighbor. The best neighbor is chosen from the two possible neighbors (if exists) of a node N, that have the same anchor, such that it has enough entries to avoid the more expensive *merge* operation. The original B^+ search tree algorithms impose no restriction on choosing a neighbor that has the same anchor and we avoided such choice because of its potential and complex change to the tree, and hence increased cost of the rekey operations [38]. For example if the best neighbor to a node doesn't have the same parent, two parent entries for the two nodes need to be rekeyed (regeneration of the key).

If the deletion of an entry at node N causes an underflow, i.e. its size becomes (*Min_d* –1), a *shift* or *merge* operation is essential to keep it at least half full. The shift operation moves an entry from *Nghbr* to N, where *Nghbr*'s size is more than *Min_d*. The merge operation moves all entries of N to *Nghbr* and deletes node N, where *Nghbr*'s size is exactly *Min_d*.

Fig. 30 and Fig. 31 are examples that depict the possible *shift* operations from right and left neighbors, respectively, in a B^+-LKH of degree $d = 4$. The minimum number of entries in a node is 2. The examples illustrate the nodes before and after the operation in the two cases of the nodes (N and Nghbr) being leaf or internal nodes. In addition, the examples illustrate how the ID is adjusted in the anchor node. The shift method call is provided by the anchor node and returns an ID that will be sent in the RM.

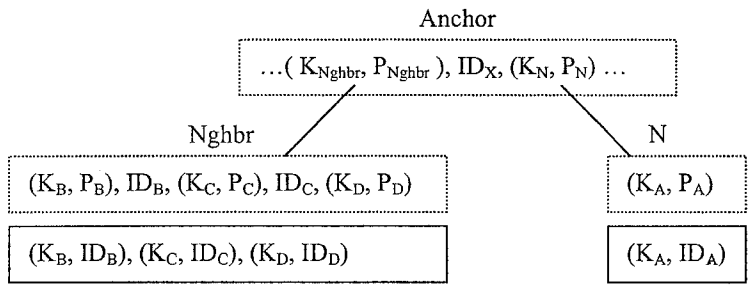


(a) Node N before *shift* from right neighbor Nghbr.

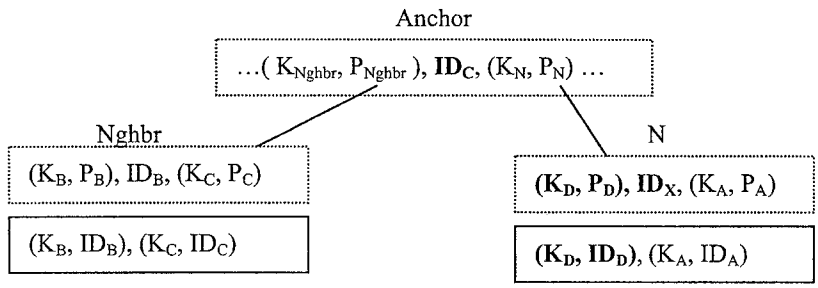


(b) Node N after *shift* from right neighbor Nghbr (ID_B is returned).

Fig. 30. An example of B^+-LKH internal/leaf node right *shift* operation.



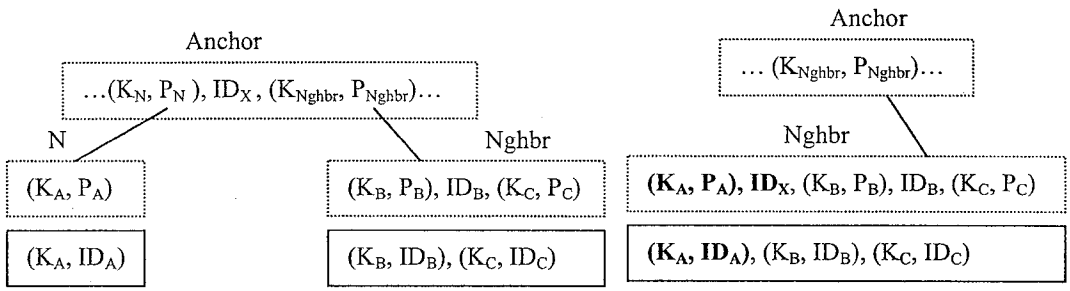
(a) Node N before *shift* from left neighbor Nghbr.



(b) Node N after *shift* from left neighbor Nghbr (ID_C is returned).

Fig. 31. An example of B^+-LKH internal/leaf node left *shift* operation.

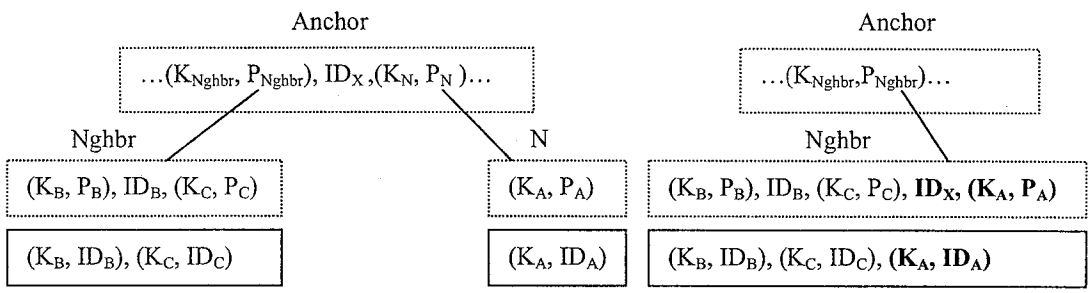
Fig. 32, and Fig. 33 are examples that depict the possible *merge* operations from right and left neighbors respectively in a B^+-LKH of degree $d = 4$. The minimum number of entries in a node is 2. The examples illustrate the nodes before and after the operation in the two cases of the nodes (N and Nghbr) being leaf or internal nodes. In addition, the examples illustrate how the anchor node is adjusted. The merge method call is provided by the anchor node and returns the deleted ID that will be sent in the RM.



(b) Node N before *merge* with right neighbor Nghbr.

(a) Node N after *merge* with right neighbor Nghbr (ID_X is returned).

Fig. 32. An example of B^+-LKH internal/leaf node right *merge* operation.



(b) Node N before *merge* with left neighbor Nghbr.

(a) Node N after *merge* with right left Nghbr (ID_X is returned).

Fig. 33. An example of B^+-LKH internal/leaf node left *merge* operation.

Fig. 34. illustrates the B^+-LKH group member removal and RM construction algorithm, *RemoveMember*, that details how a B^+-LKH is shrinking wile removing a group member entry. The removed member is identified by his unique ID *memberID*. Initially, the B^+-LKH rooted at node *root* is searched by *memberID* for the *position* of the deleted entry in the leaf node N. While searching for the entry the *lookup* method looks for the best neighbor of each node and pushes it in *nodeStack* as well as a flag is pushed in *isRghtStack* that determines if it is the right or the left neighbor. The deletion of an entry form a leaf node could introduce further deletions in upper level nodes that could propagate up to root or stops at lower level. The deletion of a member entry has four different cases, i.e. four different RM types, and those are REMOVE, MERGE, SHIFT, and DECREASE.

After the entry is deleted from leaf node N, node N is checked to see if it is at least half full or not. If node N contains at least *Min_d* entries *rekeyMsg* type is set to REMOVE. If node N underflows the best neighbor *Ngbgr* (that is popped from the stack) is checked to see if we could *shift* an entry from it (has more than *Min_d* entries) or a *merge* is essential (has exactly *Min_d* entries). If *shift* is possible, an entry is shifted form *Nghbr* to N, the deletion propagation stops, and *rekeyMsg* type is set to SHIFT. If *Nghbr* has exactly *Min_d* entries then the entries of node N are merged (moved) to *Nghbr* node. In this case, the internal node entry at the parent node (*anchor*) that was pointing to node N has to be deleted. If the anchor (parent) didn't underflow after that deletion the merge stops and *rekeyMsg* type is set to MERGE. If the anchor underflows its neighbor is checked for *shift* or *merge* operation, and so the deletion could propagate to upper level nodes. If the deletion propagates to root node and merged its only two children nodes, B^+-LKH height is reduced by 1 and *rekeyMsg* type is set to DECREASE. Please see appendix A for examples of the different group member removal cases.

```
Method RemoveMember(memberID)
Globals: root, h, d, Min_d, GK;
Returns: rekeyMsg;

(position, nodeStack, isRghtStack) = lookup (memberID);
rekeyMsg ← (position, memberID); level = h –1; N = nodeStack.pop();
N.delete(memberID);
if ((N.size() ≥ Min_d) or ((N equals root) and (N.size() > 0)))
 then rekeyMsg.type = REMOVE;
 else
  { done = false;
    while (level > 0)
    { decrement level; anchor = nodeStack.pop(); isRght = isRghtStack.pop();
      Nghbr = anchor.getNghbr(N, isRght);
      if (Nghbr.size() > Min_d)
       then
       { ID = anchor.shift(N, Nghbr); rekeyMsg.type = SHIFT;
       rekeyMsg← (level, ID, isRght, anchor.rmvRekey(N), anchor.rmvRekey(Nghbr));
         done = true; breakWhile; }
       else
        { ID = anchor.merge(N, Nghbr); N = anchor;
          rekeyMsg←(ID, isRght, anchor.mrgRekey(Nghbr, isRght));
          if ((N.size() ≥ Min_d) or ((N equals root) and (N.size() > 1)))
            then { rekeyMsg.type = MERGE; rekeyMsg← (level); done = true;
                   breakWhile; }
    }
  }
    if (not done )
      then { if (N equals root) then free root; else  root = N.childAt(0);
             rekeyMsg.type = DECREASE; decrement h; }
  }
for (i = 0 to (level-1))
  { prevN = N; N = nodeStack.pop();
    rekeyMsg ← N.rmvRekey(prevN); }
if (rekeyMsg.type() does-not-equal DECREASE)
 then rekeyMsg ← GK.rmvRekey(root);
return rekeyMsg;
```

Fig. 34. The B^+-LKH group member removal and RM construction algorithm.

4.3.5 Algorithms Analysis

Analyzing *AddMember* and *RemoveMember* algorithms for a B^+-LKH of height h, TABLE IV illustrates RM's (shown in Fig. 26) different field sizes for all group member addition and removal cases (RM type), where RM level equals L. TABLE V illustrates the different rekey packet sizes when encryption-based or XORBP KDT is used, where *Enc_K* is the encrypted key size in bytes, and S is the XORBP rekey packet size. As previously mentioned, for a B^+-LKH of degree d, *Min_d* is the minimum number of non-root node entries that is equal to $\lfloor d/2 \rfloor$.

TABLE IV

RM FIELD SIZE FOR B^+-LKH OF HEIGHT h, AND RM'S LEVEL L

RM type	"ID" length	"isRght" length	Number of *addRekey* packets (nA)	Number of *rmvRekey* packets (nR)	Number of *mrgRekey* packets (nM)
ADD	1	0	h	0	0
SPLIT	$h-L$	0	$L+1$	$2\times(h-L-1)$	0
INCREASE	$h+1$	0	1	$2\times h$	0
REMOVE	1	0	0	h	0
MERGE	$h-L$	$h-L-1$	0	$L+1$	$h-L-1$
SHIFT	$h-L$	$h-L-1$	0	$L+3$	$h-L-2$
DECREASE	h	$h-1$	0	0	$h-1$

TABLE V

REKEY PACKET SIZE FOR ENCRYPTION-BASED AND XORBP KDTS

	Encryption-based	XORBP
addRekey packet size	$2 \times Enc\_K$	S
rmvRekey packet size	$e \times Enc\_K$ where $Min\_d \le e \le d$ (e is the number of children for that key entry node)	S
mrgRekey packet size	$Min\_d \times Enc\_K$	S
Number of keys generated	$nA + nR + nM$	$nA + nR + nM$
Number of encoded BPs (A BP is K numbers in the range [0:S -1])	0	$nA + nR + nM - 1$
Number of *xored BPs*	0	$nM \times (Min\_d - 1)$ for MERGE and DECREASE $nM \times (Min\_d - 1) + 1$ for SHIFT 0 otherwise

4.4 B^+-LKH Rekey Client Processing

The rekey client is the software component at every group member that receives RMs and updates the client maintained set of keys. The rekey client initially receives *initMsg* that initializes the variables *ID*, *position*, *h*, *d*, and *Min_d* (calculated from *d*). The *position* is represented as an array of size *h*, where *position*[0] identifies the child node number of LKH root node. In addition, the rekey client maintains a list of keys *keyList* of size (*h* + 1), where its first element (entry number 0) is his individual key and its last element (entry number *h*) is *GK*. When the client receives *initMsg* he inserts his individual key in a newly created *keyList*. Then the client keeps receiving *rekeyMsg* to update his keys.

Updating *keyList*[i] from a rekey packet depends on whether the KDT is encryption-based or XORBP. If an encryption-based KDT is used, selecting the key *encKey* to be

decrypted depends on the rekey packet type (*addRekey*, *rmvRekey*, or *mrgRekey*) and *position*[i]. The selected *encKey* is decrypted either with its previous version, or with *keyList*[i-1]. On the other hand if XORBP is used, updating *keyList*[i] is symmetric for all packet types and uses *keyList*[i-1] and its associated BP to get the new version of the key. For every updated key, except *GK*, the associated BP is updated from the *encodedBP* in the same rekey packet. The individual BP that is associated to the individual key (*keyList*[0]) is sent in the initialization message *initMsg*.

When the rekey client receives *rekeyMsg* he compares his *position* with *rekeyMsg.position* to decide on the starting matching level (*match*) where he should start updating his *keyList*. For example, if the member individual is in the leaf node that has the inserted/deleted entry *match* will be 2. If the member individual entry is in a leaf node that has the same parent of the directly affected leaf node *match* will be 3, and so on. If *position* has no intersection with *rekeyMsg position* then *match* is set to (h + 1). The following code fragment illustrates how to adjust *match*. Note that, *match* equals 1 only at the new individual (i.e., his rekey client software component).

```
match = -1;
for (i = 0 to (h-1))
    if (position[i] equals rekeyMsg.position[I])
      then match = i;
          else breakFor;
match = h – match;
if (match equals 1) then match = 2;
```

After deciding on *match*, the update procedure is triggered by *rekeyMsg.type* and executed to update *keyList*. There are six different update procedures according to *rekeyMsg* type. The following is the *Simple* update procedure called when *rekeyMsg.Type* equals ADD or REMOVE. A group member whose *match* equals 2 and his ID is greater than the inserted/deleted ID experiences a change in his individual leaf node position. This individual leaf position is incremented by 1 if a new individual entry is inserted and is decremented by 1 if an individual entry is deleted. In addition, a group member updates

his *keyList* from the corresponding rekey packets according to his *match*. A group member whose *match* equals 2 updates all h keys, while a group member whose *match* equals (h + 1) updates only one key (*GK*). Please see Appendix B for detailed rekey client update procedures and an example.

```
if ((match equals 2) and (ID > rekeyMsg.ID[0]))
    then if (rekeyMsg.type equals ADD)
            then increment position[h-1];
            else decrement position[h-1];
for (i = (match - 2) to (h – 1))
    keyList.update(i + 1, rekeyMsg.packet[i]);
```

4.5 Experimental Results

We have implemented the rekey manager and the rekey client in Java™[[62]. Both S-LKH and B^+-LKH protocols are available for use with an encryption-based or XORBP KDT. In the following experiments, we compare the performance of an unbalanced LKH (S-LKH) versus a balanced LKH (B^+-LKH). First, an experiment is performed to study the frequency of the different rekey message (RM) types in both add and remove rekeyings. Second, the simulated group dynamics in the experiments is explained. Third, an experiment is performed to compare S-LKH and B^+-LKH rekey costs. Fourth, an experiment is performed to study the effect of LKH degree and group dynamics on S-LKH and B^+-LKH rekey costs and storage.

4.5.1 Frequency of Different Addition and Removal Cases

This experiment illustrates the frequency of different RM types in the addition and the removal rekey cases for both S-LKH and B^+-LKH protocols. The LKH degree d is increased from 2 to 10. For every LKH degree, the group size n increases from 32 to 2048 in multiples of 2. For every d and n, 10 LKHs are constructed by a sequence of n member additions then n member removals. A new unique random ID is generated for every new member. The removed member is randomly chosen from the existing

members. For every constructed LKH, the frequency of different RM types is recorded. We have noticed that the frequency of each RM type depends on LKH degree and doesn't depend on the group size.

Fig. 35 and Fig. 36 illustrate the frequency of different RM types for S-LKH protocol in the addition and removal rekey cases, respectively. We can observe that the frequency of the simplest rekey cases (ADD & REMOVE) increases with LKH degree increase, and are occurring more than 80% of the time for a LKH degree greater than 8.

Fig. 37 and Fig. 38 illustrate the frequency of different RM types for B^+-LKH protocol in the addition and removal rekey cases, respectively. Similarly, we can observe that the simplest rekey cases are occurring more than 80% of the time for a LKH degree greater than 8. In addition, the most expensive rekey cases (INCREASE & DECREASE) are occurring less than 1% of the time for any LKH degree.

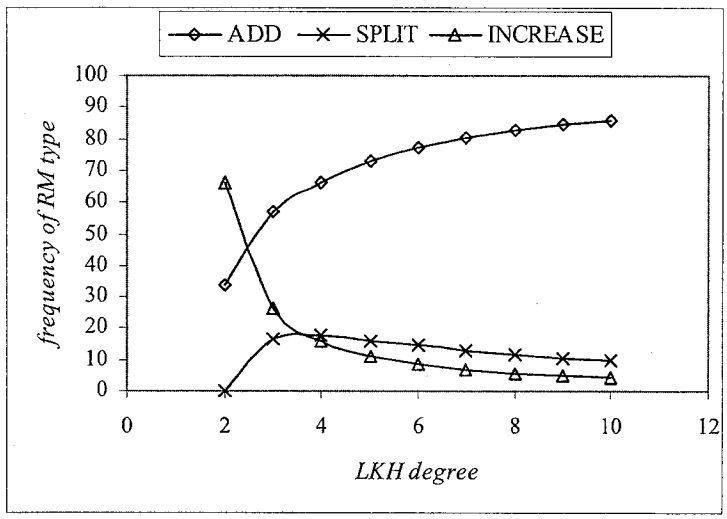


Fig. 35. Frequency of add RM type for the S-LKH protocol.

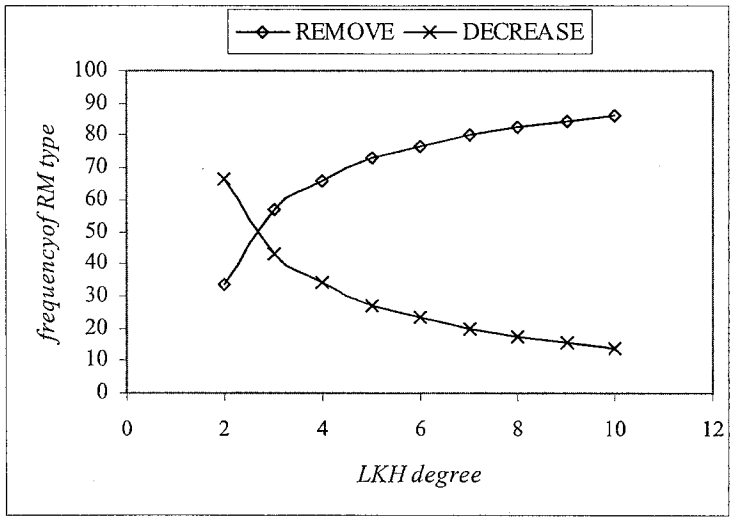


Fig. 36. Frequency of remove RM type for the S-LKH protocol.

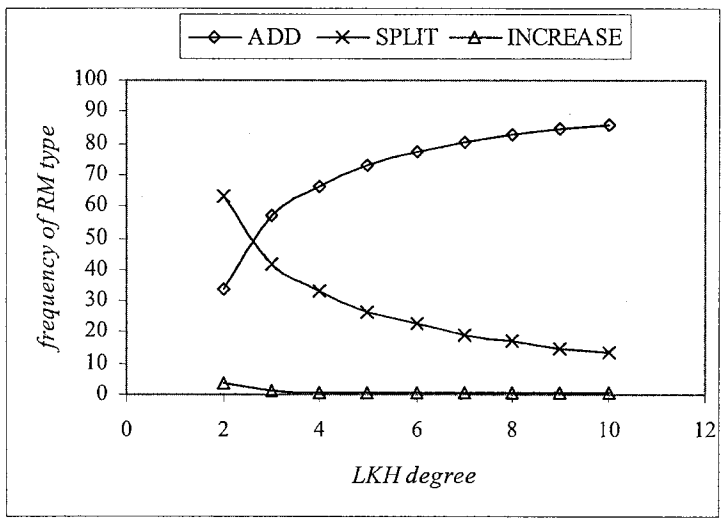


Fig. 37. Frequency of add RM type for the B^+-LKH protocol.

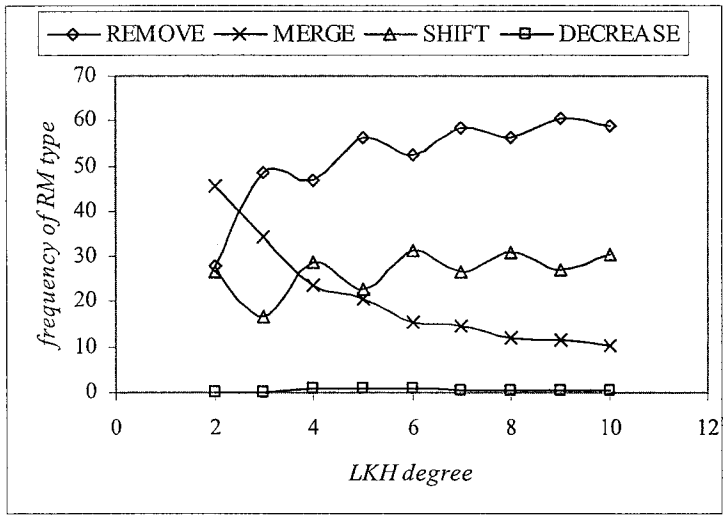


Fig. 38. Frequency of remove RM for the B^+-LKH protocol.

4.5.2 Group Dynamics

To simulate group dynamics, a LKH is constructed by a sequence of aN member additions followed by a sequence of rN member removal. The group size $n = aN - rN$, and the group dynamic ratio gdr is defined to be $gdr = rN / aN$. If the group is static (i.e., no member is removed) $gdr = 0$. For $gdr = 0.4$, the group size is 60% of the added members (i.e., n = 60% aN). To have a group of size $n > 0$, gdr value has to be in the range [0, 1[.

When an encryption-based KDT is used, the rekey message cost is measured as the total number of encrypted keys in a RM (in all rekey packets). On the other hand, when XORBP KDT is used, the rekey message cost is measured as the number of rekey packets in a RM.

In the following experiments, we compare the rekey performance of S-LKH versus B^+-LKH for the same LKH degree, group size, and group dynamic ratio. For every protocol, and the parameters (d, n, gdr), we construct 100 LKHs. For every constructed LKH, its height and the number of allocated nodes (LKH storage) are recorded. Then, 10

readings for rekey message cost in both add and remove rekey cases (i.e., a remove member followed by add member 10 times) are recorded. The plotted number of allocated nodes is the average of 100 readings, and the plotted rekey message cost is the average of 1000 reading.

4.5.3 S-LKH and B$^+$-LKH Rekey Cost

This experiment compares the behavior of add and remove rekey costs for S-LKH versus B$^+$-LKH protocols in terms of number of rekey packets and number of encrypted keys. The experiment is performed for LKH degree $d = 4$, group size $n = 8192$, and $gdr = 0.4$.

Fig. 39 and Fig. 40 illustrate, for both protocols, the frequency of the different values obtained for the number of rekey packets in a RM in add and remove rekeyings, respectively. We can observe the symmetry between the two figures (i.e., add and rekey symmetric cost in terms of the number of rekey packets in a RM). In addition, we can observe that using the S-LKH protocol, the number of rekey packets in a RM spans a wider range of values when compared to the B$^+$-LKH protocol.

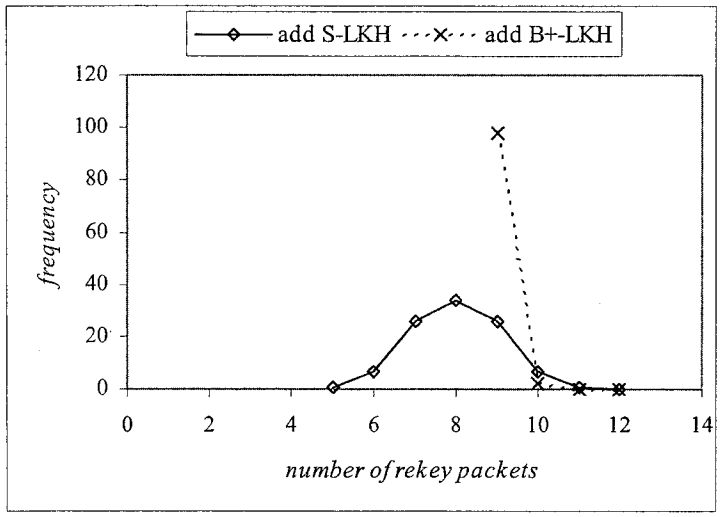


Fig. 39. Frequency of number of rekey packets in add rekey message.

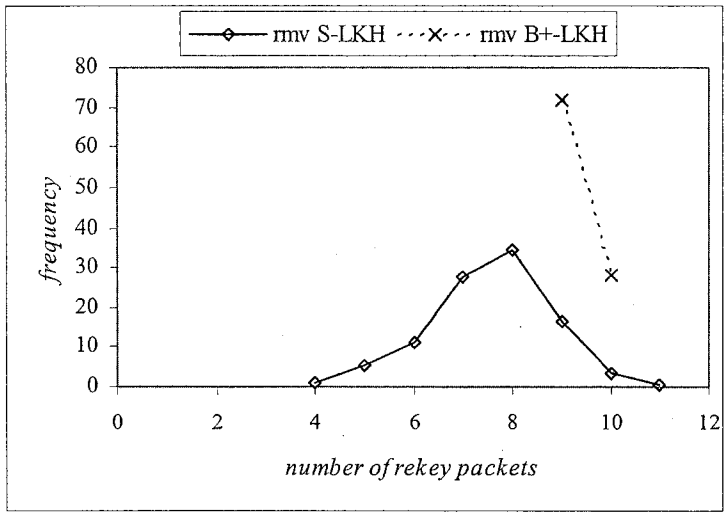


Fig. 40. Frequency of number of rekey packets in remove rekey message.

Fig. 41 and Fig. 42 illustrate, for both protocols, the frequency of the different values obtained for the number of encrypted keys in a RM in add and remove rekeyings, respectively. We can observe the un-symmetry between the two figures. Similarly, the rekey cost in terms of the number of encrypted keys spans a wider range of values when used with the S-LKH protocol compared to B^+-LKH. The S-LKH wider range of cost values is due to the un-balanced LKH that implies the existence of leaf nodes at different levels from the root node.

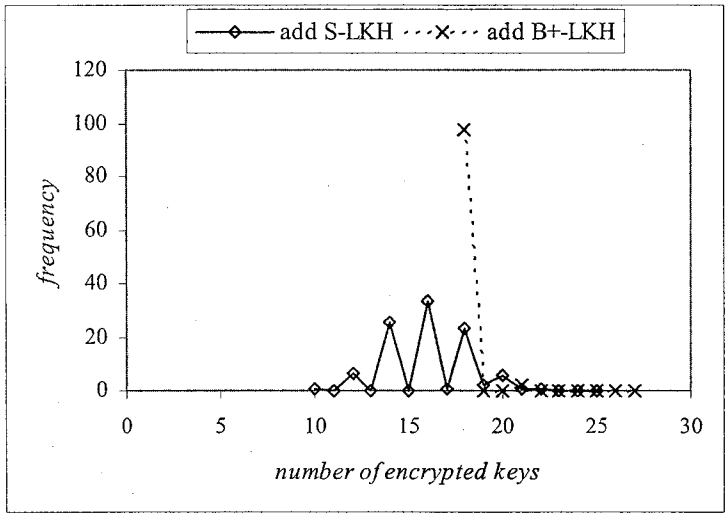


Fig. 41. Frequency of number of encrypted keys in add rekey message.

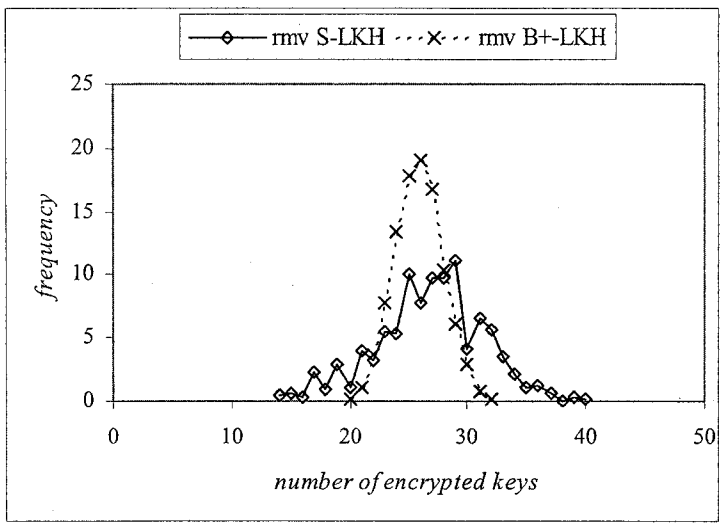


Fig. 42. Frequency of number of encrypted keys in remove rekey message.

TABLE VI summarizes the different rekey cost metrics for S-LKH versus B^+-LKH protocols when $d = 4$, $n = 8192$, and $gdr = 0.4$. TABLE VII summarizes the results when $d = 4$, smaller group size $n = 512$, and $gdr = 0.4$. TABLE VIII summarizes the results when LKH is having larger degree $d = 8$, large group size $n = 8192$, and $gdr = 0.4$.

From the previous results, we can conclude that the rekey cost maintains the same behavior for all group sizes and LKH degrees. The use of B^+-LKH protocol increases the average number of rekey packets and the average number of encrypted keys in a RM when compared to S-LKH protocol. On the other hand, the use of B^+-LKH decreases the average LKH height, the number of allocated nodes, and the maximum number of encrypted keys. The maximum number of encrypted keys (or the rekey packets) is used in estimating the minimum time that has to be elapsed between two consecutive rekeyings.

TABLE VI

S-LKH VERSUS B^+-LKH REKEY COST FOR ($d = 4$; $n = 8192$; $gdr = 0.4$)

	S-LKH	B^+-LKH
Average LKH height.	**11.09**	**9**
Average LKH number of allocated nodes.	**6485.72**	**5703.754**
AddMember average number of rekey packets.	8	9.028
AddMember maximum number of rekey packets.	12	12
RemoveMember average number of rekey packets.	7.542	9.282
RemoveMember maximum number of rekey packets.	11	10
AddMember average number of encrypted keys.	16.041	18.084
AddMember maximum number of encrypted keys.	25	27
RemoveMember average number of encrypted keys.	26.81	25.835
RemoveMember maximum number of encrypted keys.	**40**	**32**

TABLE VII.

S-LKH VERSUS B^+-LKH REKEY COST FOR ($d = 4$; $n = 512$; $gdr = 0.4$)

	S-LKH	B^+-LKH
Average LKH height.	**7.4**	**6.11**
Average LKH number of allocated nodes.	**404.007**	**353.559**
AddMember average number of rekey packets.	5.782	6.151
AddMember maximum number of rekey packets.	9	8
RemoveMember average number of rekey packets.	5.306	6.412
RemoveMember maximum number of rekey packets.	8	8
AddMember average number of encrypted keys.	11.637	12.343
AddMember maximum number of encrypted keys.	19	18
RemoveMember average number of encrypted keys.	17.912	17.873
RemoveMember maximum number of encrypted keys.	**29**	**23**

TABLE VIII.

S-LKH VERSUS B^+-LKH REKEY COST FOR ($d = 8$; $n = 8192$; $gdr = 0.4$)

	S-LKH	B^+-LKH
Average LKH height.	**6.16**	**6**
Average LKH number of allocated nodes.	**3108.134**	**2168.812**
AddMember average number of rekey packets.	5.171	6.001
AddMember maximum number of rekey packets.	7	7
RemoveMember average number of rekey packets.	5.141	6.338
RemoveMember maximum number of rekey packets.	7	7
AddMember average number of encrypted keys.	10.352	12.007
AddMember maximum number of encrypted keys.	19	19
RemoveMember average number of encrypted keys.	32.986	30.488
RemoveMember maximum number of encrypted keys.	**45**	**39**

4.5.4 Effect of Group Dynamics and LKH Degree

If encryption-based KDT is used the optimal LKH degree is 4, and the total number of encrypted keys in a RM is the rekey cost metric. When XORBP KDT is used, the number of rekey packets in a RM is used as a rekey cost metric. In this experiment, we study how the group dynamics and LKH degree affect the number of rekey packets (for XORBP KDT) in a RM and the number of allocated nodes in LKH (LKH storage). As we concluded from the previous experiment (section 4.5.3), add and remove rekey costs are symmetric in terms of the number of rekey packets in a RM.

The group size $n = 512$, and LKH degree is increased from 4 to 32 in increments of 4 (i.e. 4, 8, 12, ..., and 32). Fig. 43 illustrates, for S-LKH and B^+-LKH protocols, the average number of rekey packets in a RM for static group ($gdr = 0$). Fig. 44 illustrates the results when $gdr = 0.4$. We can observe that the B^+-LKH protocol introduces a slight increase in the average number of rekey packets in a RM over S-LKH protocol. Comparing Fig. 43 and Fig. 44, we can conclude that this increase is slightly affected by the group dynamics. Note that this increase is for individual rekeying (i.e. single add or remove rekey). In chapter V, we present batch rekeying for a sequence of add and/or remove requests. For batch processing, the B^+-LKH protocol rekey cost outperforms the S-LKH protocol.

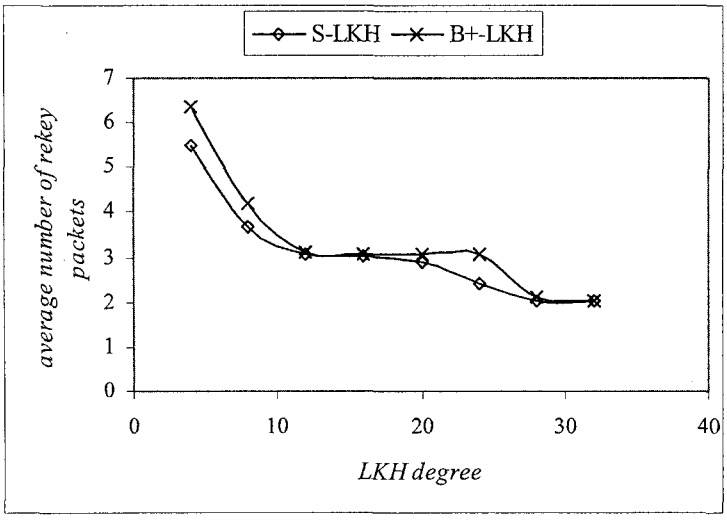


Fig. 43. Average number of rekey packets in a RM, where $gdr = 0$, and $n = 512$.

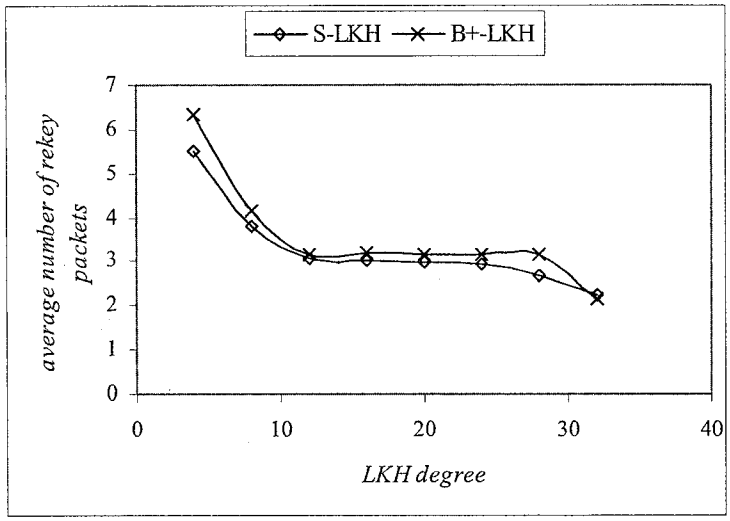


Fig. 44. Average number of rekey packets in a RM, where $gdr = 0.4$, and $n = 512$.

S-LKH increases the number of LKH allocated nodes when compared to B^+-LKH (section 4.5.3). If the number of allocated nodes for S-LKH and B^+-LKH are *sLKHS* and *bLKHS*, respectively. The S-LKH percentile increase in the number of allocated nodes can be calculated as $inc = (sLKHS - bLKHs) \times 100 / bLKHS$. Fig. 45 illustrates *inc* for group size n = 512, and gdr = 0, gdr = 0.2, and gdr = 0.4. We can observe that, the increase *inc* has a non-linear relation with the LKH degree. Howerver, *inc* increases with the increase of group dynamics. Fig. 46 illustrates *inc* when the group size n = 8192, and group dynamics ratio is 0, and 0.4. Similarly, the S-LKH percentile increase (*inc*) in allocated storage over B^+-LKH increases with the increase of group dynamics and attains 80% for gdr = 0.4. We have noticed that *inc* peaks when the group size (n) is near an exact power of d. For example when n = 512 *inc* peaks at d = 8 (8^3 = 512), and d = 24 (24^2 = 576), and when n = 8192 *inc* peaks at d =20 (20^3 = 8000). In this case, the B^+-LKH maintenance algorithms keeps much less number of nodes than the S-LKH ones.

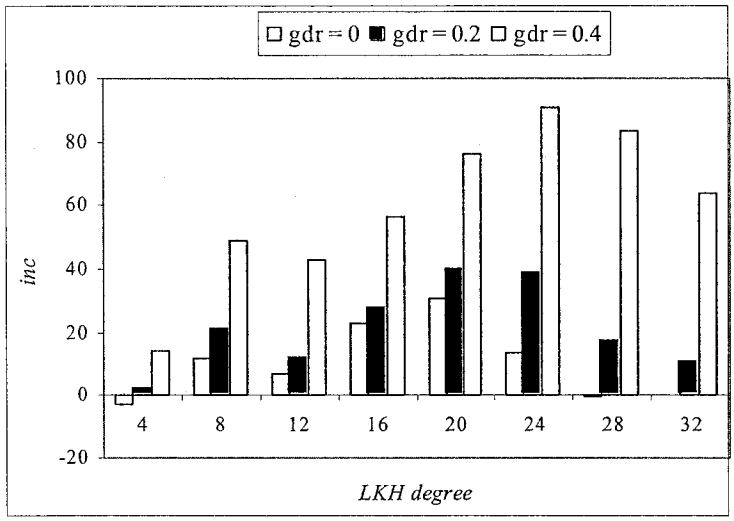


Fig. 45. S-LKH average number of nodes increase over B^+-LKH, where $n = 512$.

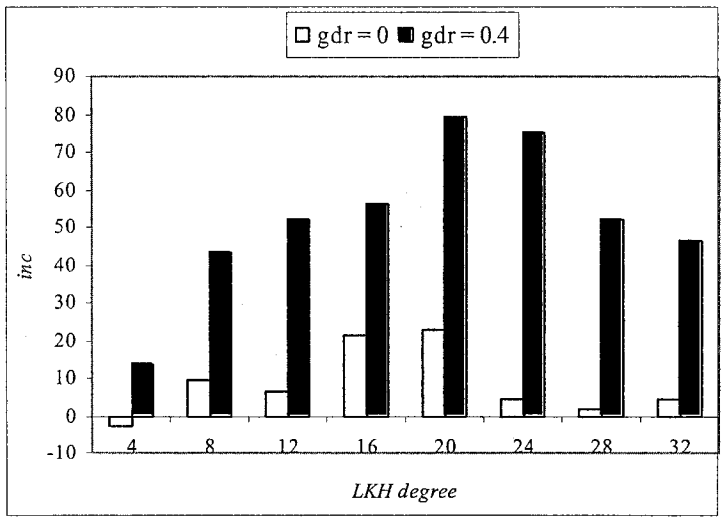


Fig. 46. S-LKH average number of nodes increase over B^+-LKH, where $n = 8192$.

4.6 Conclusion

In this chapter, two novel techniques for LKH maintenance and their associated rekey protocols are presented. The new techniques are based on the rekey manager assigning a unique individual identification (ID) for each group member. In both techniques, the LKH plays a dual role as a key tree and as an easily searchable data structure for individual material (ID, key, IP address, name, ...etc) using individual IDs. The proposed techniques detail the LKH node structure, the rekey message format, the LKH insertion and deletion algorithms along with the rekey message construction for different insertion and deletion scenarios. Moreover, the rekey client processing to different rekey message types is presented. The first technique, denoted S-LKH, maintains LKH as unbalanced search tree using individual IDs as search values. The traditional search tree insertion and deletion algorithms are adapted to the constraint that individual materials are always entries in leaf nodes. The second technique, denoted B^+-LKH, maintains LKH as a balanced search tree that has the same structure as S-LKH. In addition, a B^+ search tree has two additional constraints. The first constraint is, all leaf nodes are always at the same distance from the root (i.e. balanced). The second constraint is, all non-root node are always at least half full. These constraints introduce complexity and extra overhead in the rekey message and the rekey client processing.

We performed empirical experiments to study and compare the behavior of S-LKH and B^+-LKH protocols. The first experiment concludes that the frequency of the simplest RM types (simple insertion and deletion scenarios) increases with LKH degree increase for both protocols. The frequency of the simplest RM types is more than 80% for LKH degree greater than 8. For B^+-LKH protocol, the frequency of the most expensive RM type is less than 1% for any LKH degree. Other experiments illustrate that the use of B^+-LKH protocol increases the average number of rekey packets and the average number of encrypted keys (if encryption-based KDT is used) in a RM over S-LKH. On the other hand, the use of B^+-LKH decreases LKH height, the maximum number of encrypted keys in a RM, and the number of LKH allocated nodes (LKH storage). The S-LKH increase over B^+-LKH in the number of allocated nodes increases with increased group dynamics and attains more than 80% for highly dynamic groups (current group size = 60% number of added members).

In chapter IV, the rekey is performed for one group member addition or removal. In chapter V, batch rekeying for more than one group member addition and/or removal is introduced. For batch rekeying, B^+-LKH protocol rekey cost outperforms S-LKH protocol.

CHAPTER V

BATCH PROCESSING OF GROUP REKEYING

In chapter IV, we focused on individual rekeying, i.e. rekeying after each join and leave request. Individual rekeying is not a practical solution. For example, if the inter-arrival time (time between two join requests) of group members at the start of a session is very small; the inter-rekey time (time between two consecutive rekeyings) will be consequently very small and a new group key might be issued by the rekey manager before the previous key version has reached (or has been used by) the group members. Periodic rekeying has been suggested to alleviate this problem [45], [59], [69]. Periodic rekeying suggests rekeying after a fixed period of time that is large enough to avoid the above problem. Periodic rekeying requires a rekeying for a batch of requests, i.e., for accumulated join and leave requests during this period. Researchers suggested that the expiration of a rekey period triggers the rekeying process. Such approach does not take into account the batch size or the join/leave request delay during the rekey period.

This chapter introduces a generalized rekey policy definition based on three main parameters that determine the triggering condition for the rekeying process. The three main parameters are batch size, maximum join or leave request delay (time between receiving the request and the start of the rekeying process), and the minimum inter-rekey period (a minimum period of time that has to be elapsed between two consecutive rekeyings). The defined rekey policy provides versatile configuration options to the rekey triggering condition. The rekey policy can be simply used to provide periodic rekeying as well as other complex rekeying conditions as configured by the application. In addition, a simplified view of the software objects that are used to provide secure group key management is presented. Moreover, the batch rekey message format and construction are presented. When LKH key management is used, individual rekeying requires generating and distributing a set of keys that fall in a LKH path from an inserted/deleted leaf node to the root. On the other hand, batch rekeying requires generating and distributing a set of keys that compose a sub-tree of the original LKH. The rekey sub-tree

is composed of the individual LKH paths of the inserted and/or deleted leaf nodes to the root. The batch rekey sub-tree construction for the B^+-LKH protocol is detailed.

For individual rekeying, the use of B^+-LKH protocol introduces major LKH storage (number of allocated nodes) savings and slightly more rekey processing than the use of S-LKH protocol (see section 4.5.3). In this chapter, it will be demonstrated, through empirical experiments that using the B^+-LKH protocol for batch rekeying substantially reduces rekey processing overhead when compared to the S-LKH protocol with large batch sizes and/or high group dynamics. In addition, our experiments show that B^+-LKH rekey performance is stable (bounded) for highly dynamic groups while S-LKH rekey performance deteriorates as the group dynamics increases. Such S-LKH instability is due to the fact that the minimum number of node entries is one, while for B^+-LKH nodes have to be at least half full.

This chapter is organized as follows: Section 5.1 presents the motivation for introducing the rekey policy parameters. Section 5.2 details the proposed rekey policy definition. Section 5.3 presents a simplified view of the secure group key management software objects. Section 5.4 illustrates the batch rekey message, and the general batch rekeying process performed by a rekey manager that maintains S-LKH or B^+-LKH. Section 5.5 presents experimental results that compare S-LKH versus B^+-LKH protocols for batch rekeying. Finally, the chapter is concluded in section 5.6.

5.1 Motivation

Changing the group key is very expensive in terms of processing time, and bandwidth consumption. According to the software model introduced in section 3.1, the rekeying process time has three major time components: 1) RM construction by the rekey manager; 2) RM transmission from the rekey manager to all group members through a reliable group rekey channel; 3) RM processing by a rekey client. The rekey cost (time and bandwidth) at the rekey manager depends mainly on the group size, the key management protocol, the rekey manager processing power, the network bandwidth and delay, and the rekey transport protocol. The existence of a central group key manager (and a rekey manager) allows heterogeneous members' environments and the client processing is minimized. On the other hand, the group key manager is receiving the

group members' requests to join and leave the group, and is responsible for rekeying the group when it deems necessary. Periodic batch processing is introduced as a practical solution for frequent group rekeying [45], [59], [69]. For batch of requests, the rekey manager generates one RM that includes group keys' updates due to a set of group members joining and/or leaving the group. Almeroth and Ammar [1] demonstrate that for different group applications, the inter-arrival time and member joining duration are exponential in nature. Simple periodic rekeying does not take into account the possibility of no join or leave requests accumulating during a rekey period. Consequently, the proposed batch rekey policy has three main parameters, minimum inter-rekey period, batch size, and maximum request delay.

The minimum time between two consecutive rekeyings, denoted inter-rekey period, has to be greater than the expected (maximum) time needed to rekey the group. Otherwise, a new group key will be issued before its previous version is ever used. The need for the group key manager to guarantee minimum time interval between two consecutive rekeyings makes it essential to process a batch of requests. Moreover, to avoid a group startup implosion it is required to delay the initial creation of the group key for a suitable time period. The initial creation of the group (key) is processed as a batch processing for multiple new members addition.

LKH batch rekeying requires updating a set of keys that compose a sub-tree of the original LKH. The rekey sub-tree is constructed from all the added/removed leaf node paths to the root. Li et al. [45] show that, for a group of size n and LKH of degree 4 (optimal LKH degree for encryption-based KDT), if an all add requests batch size is greater than $n/2$ or an all remove requests batch size is greater than $n/4$ the use of LKH key management is worse than the use of star key management (chapter III). In both cases, the number of encrypted keys in a RM is equal to or greater than the group size n. That necessitates taking the batch size into consideration when designing a rekey policy.

The maximum request delay is defined to be the maximum time to be elapsed from the group key manager receiving the request and the start of the rekeying process. The maximum request delay is a major security concern. This delay determines the maximum period a group member will wait after he joins the group before being able to receive any group communication. Moreover, this delay determines the maximum period a group

member will be able to keep receiving the group communication after he leaves the group.

We can observe that simple periodic rekeying only guarantees a fixed time interval between two rekeyings but doesn't take into consideration the batch size and/or the maximum request delay. For some applications one of the above parameters might be of more interest and easier to estimate while the others are irrelevant or hard to estimate. For example, a cable network application might require a maximum request delay of 2 days that triggers the rekeying process, i.e. members wait at most 2 days to be added/removed to the network. Another example is a video conferencing application that requires a minimum inter-rekey period of 1 minute and a maximum request delay of 3 minutes.

The rekey policy parameters can be estimated from the group characteristics (the above time components), and other resource constraints such as the allowed usage of processing power and/or bandwidth. For example, the rekeying process might be allowed only 10% of the machine processing power, and no more than 5 kbps of bandwidth usage.

The necessity of changing the group key because of a new member joined the group (perfect backward secrecy/PBS), or a member left the group (perfect forward secrecy/PFS) depends on the application. For example, for a group of students meeting in a virtual classroom there is no need to change the group key when a member joins the group late (he is allowed to join from the start). On the other hand, for members joining a video-on-demand provider it is essential to change the group key when a new member joins or leaves the group. Note that, if the application only requires perfect backward secrecy, a simple non-LKH protocol can be used. The use of an LKH protocol is essential when perfect forward secrecy is required, and that is our concern.

5.2 Rekey Policy Definition

The group key manager is configured by the group *rekey policy* as to when the group rekeying should be performed. The rekey policy determines the timing of both the initial group key creation and the further rekeying condition. It is assumed that the rekey policy is static for simpler design and analysis. A dynamic adjustment to the policy parameters is left for future research.

The group key manager accumulates the requests in a *batch*. As previously mentioned in chapter III, the requests are inserted in the batch as messages are received from the authentication manager to add, remove, or refresh group members, namely Add(M), Remove(M), and Refresh(M), respectively, where M is a member identification, e.g., his name. The member *refresh* request is introduced to allow an easy recovery of a group member after short time of failure (please see chapter VI for more details). Refreshing a group member, assumes the group member temporarily lost his set of keys and requires sending him the same set of keys he was holding (as if he newly joined) without regenerating those keys. The accumulated requests are removed from the *batch* when a rekeying is initiated. The S-LKH and B^+-LKH protocols assume the rekey manager generates a unique ID for every group member that is used as a search value in constructing LKH. The request identification M is assumed to be different than ID (M might be used to generate the ID). The member identification M is required to be unique in the *batch*, while it can be replicated throughout LKH individual entries (each entry will have different ID).

The first policy parameter is the *rekey condition* (RC) that has one of four possible values: PBS for perfect backward secrecy, PFS for perfect forward secrecy, PBaFS for perfect backward and forward services, and NONE when no secrecy is required. Note that, if RC equals PBS or NONE there is no need to use an LKH protocol, but we allow their use with an LKH protocol for dynamic policy changes (e.g., used only during part of a session). In addition, if there is no change of keys due to a batch of requests (e.g., RC is PFS and the batch contains only add requests), the rekey manager still needs to construct a rekey message RM that updates the group members of changes about positions (due to the new individual entries insertions), newly created keys, and/or removed keys.

The second set of parameters determines the timing of the first group key creation, and has two components *initASize* and *initMaxDelay*. The third set of parameters determines the timing of the following rekeyings, and has three components *rekeyBatchSize*, *rekeyMinWait*, and *rekeyMaxDelay*.

The *batchSize* (*initASize* or *rekeyBatchSize*) determines the rekeying condition according to *rekeyMinWait* and *maxDelay* (*initMaxDelay* or *rekeyMaxDelay*) values as will be described in section 5.3, and its minimum value is one. We assume that a value of

zero for *minWait* or *maxDelay* means this parameter is undetermined (not important to the application). The *maxDelay* (if greater than zero) is the maximum delay a request can be held in the *batch* before start of rekeying. The *minWait* (if greater than is zero) is the minimum time that has to be elapsed between two consecutive rekeyings. Note that, $maxDelay \geq minWait$.

The *batchSize* parameter is compared to the current *batch* size, denoted *BS*. The batch size, *BS*, could simply be the total number of requests inserted in the *batch*, or a weighted sum of every request type as in equation (1). Where *AS*, *RS*, and *FS* are the number of entries in the *batch* (size) of Add, Remove, and Refresh requests respectively. And *a*, *r*, and *f* are the different weights of the different request types. The weights are policy parameters, e.g., if RC equals PBS it might be of interest to give more weight to member removal requests than any other requests.

$$BS = a \times AS + r \times RS + f \times FS \qquad (1)$$

In summary, the following are the rekey policy parameters:

- RC: the rekey condition that has four possible values: PBS, PFS, PBaFS, and NONE.
- *a, r, & f*: weights used for batch size *BS* computation.
- *initASize*, & *initMaxDelay*: initial batch size (all add requests) and initial maximum request delay that are used to specify the time of the initial group key creation.
- *rekeyBatchSize*, *rekeyMinWait, & rekeyMaxDelay*: batch size, minimum inter-rekey period, and maximum request delay that are used to specify the time of further rekeyings.

Where the minimum allowed value for *initASize* and *rekeyBatchSize* is one, and *rekeyMaxDelay* has to greater than or equal to *rekeyMinWait*.

The application has the flexibility of using all or some of the policy parameters as a deciding factor for triggering the rekey process. The type of the application determines what blend of parameters is taken into consideration. For example, an application that requires periodic rekeying every 3 minutes will have the following rekey policy:

- RC = PBaFS: backwards and forward secrecy are both required.

- $a = r = f = 1$: all request types (add, remove and refresh) have the same weight.
- *initASize* = *rekeyBatchSize* = 1: there is at least one request in the *batch* for the group key creation or a rekeying to be initiated.
- *initMaxDelay* = 5 minutes: wait 5 minutes after the first group member joins before creating the group key.
- *rekeyMinWait* = *rekeyMaxDelay* = 3 minutes: guarantee minimum inter-rekey period of 3 minutes, and maximum request delay of 3 minutes. In this case, if the requests' inter-arrival time is less than or equal to 3 minutes, a rekey will be periodically initiated every 3 minutes.

5.3 Group Key Management Software Design

Fig. 47 illustrates a simplified view of the software objects designed to provide secure group key management and their main interactions. A GroupKeyManager object is instantiated using instances of the RekeyPolicy (*rekeyPolicy*) and the RekeyManager (*rekeyManager*) as parameters. A RekeyManager object is instantiated with the rekey manager configuration such as use of B^+-LKH or S-LKH rekey protocol, LKH degree, and use of XORRBP or encryption-based KDT. A GroupKeyManager instantiates a Batch (*batch)*, Timer (*timer*), and Scheduler (*scheduler*) objects. The different objects' functionalities are as follows:

- The *RekeyPolicy* object provides methods for accessing (and setting) the policy parameters.
- The *RekeyManager* object maintains the group LKH and applies the rekey protocol. The RekeyManager provides the *rekey*(*batch*) method that takes the *batch* of requests as a parameter and constructs the rekey message RM and sends it to all group members. Moreover, the *rekey* method sets the *rekeyTime* to the time when the rekeying is started, empties the *batch*, and sets *minWaitFlag* to *false*, where *rekeyTime and minWaitFlag* are variables maintained by the *scheduler.*
- The *Batch* object provides methods for adding, removing, and accessing request messages, in addition to methods for configuring the batch size computation and a method to get the current batch size *size*().

- The *Timer* object provides a timed call to the *RekeyManager*'s method *rekey*(*batch*), where a thread is initialized when *timer's* method *start*(*TS*, *PRD*) is called to wait for certain ($sleepTime = TS - PRD - currentTime()$) before calling the *rekey* method, where *TS* is a time stamp of an action, and *PRD* is a period that has to be elapsed before initiating the rekey starting from *TS*. In addition, *timer* provides a method for interrupting and canceling the current waiting thread (*stop*()), if such thread is running. Moreover, *timer* provides a method that gets the current time-stamp *timeStamp*().
- The *Scheduler* object provides *checkRekey()* method that uses the *rekeyPolicy* to decide on the rekey triggering condition

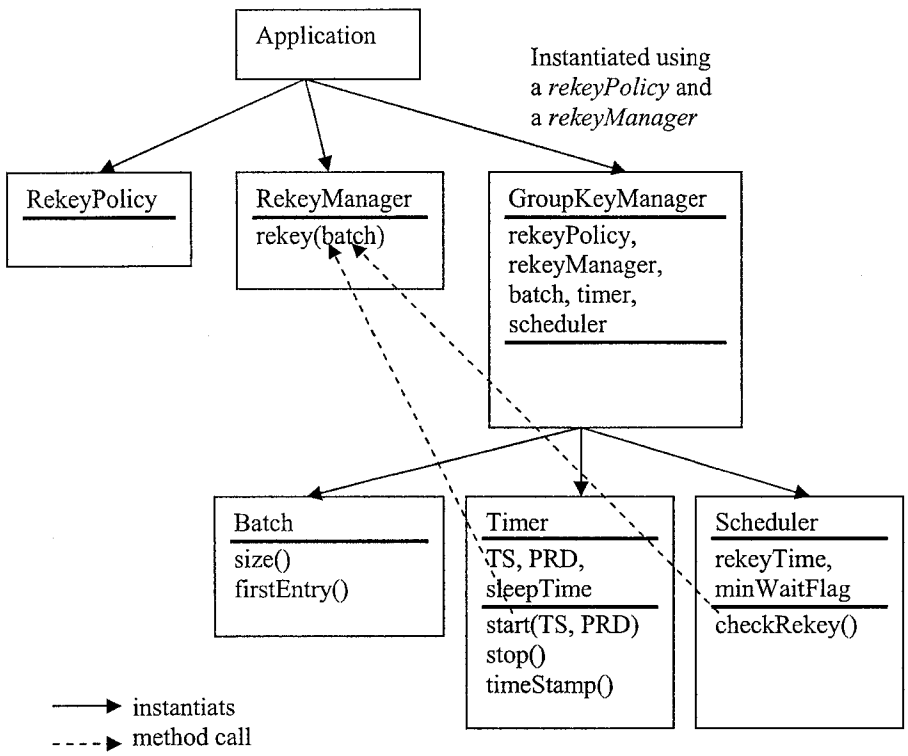


Fig. 47. Simplified view of the main group key management software objects.

When a *groupKeyManager* receives a request message (through a method call), it inserts the request in the *batch* after it is stamped with the current time-stamp, followed by a call to the scheduler's *checkRekey*() method. If the received request is Remove(M) and the *batch* contains Add(M) or Refresh(M) request, the old request is deleted and the new request is not inserted (e.g., when a member is removed a short time after he joined the group and before a rekey is initiated). If the received request is Add(M) and the *batch* contains Remove(M) request, the Remove(M) request is deleted and a Refresh(M) is inserted (e.g., when a group member recovers after short time of failure). The member identification M identifies a unique request in the *batch.* It is assumed that the group key manager will not receive a re-add request of an existing group member, or a remove request of a nonexistent member.

The scheduler that uses the *rekeyPolicy* to trigger a batch rekeying process has three different states as follows:

- ***minWait* = *maxDelay* = 0.** In this case, the batch rekeying is initialized as soon as the *batch* size reaches the *batchSize* determined by the rekey policy.
- ***maxDelay* > 0 and *minWait* = 0**. In this case, if the arrival rate of requests accumulates *batchSize* requests in the *batch* before *maxDelay* expires for the first *batch* request (the oldest), then the batch rekeying is initiated immediately. Otherwise, batch rekeying is initiated as soon as *maxDelay* expires for the oldest *batch* entry.
- ***maxDelay* ≥ *minWait* > 0**. In this case, if there is a slow arrival rate (accumulation) of requests in the *batch*, then *maxDelay* controls when the rekeying is initiated (batch size never reaches *batchSize*). On the other hand, if there is a fast arrival rate of requests in the *batch,* then *minWait* controls the minimum inter-rekey period by holding the rekeying process for a while when the *batch* size quickly reaches *batchSize.*

5.4 Rekey Sub-Tree Construction

In LKH group key management protocols, batch rekeying requires updating (generating and distributing) a set of LKH keys that compose a LKH sub-tree (denoted rekey sub-tree). The rekey sub-tree is composed of all LKH keys that fall on the paths of the inserted/deleted leaf nodes to the root. The rekey sub-tree size is the number of LKH keys that needs to be updated and therefore it represents the rekey cost.

Assuming the batch rekeying is initiated for a *batch* of requests, where the number of Add requests is *AS*, the number of Remove requests is *RS*, and the number of Refresh requests is *FS*. To reduce the rekey cost, the rekey sub-tree construction should minimize the rekey sub-tree size. There are three batch LKH update cases for such minimization as follows:

- ***AS* = *RS***. Every new individual leaf entry replaces a removed individual leaf entry in the LKH. In this case, every new group member will be assigned the same individual ID of a removed group member.
- ***AS* > *RS***. The *RS* removed individual entries are replaced by *RS* new individual entries, then the rest of the new individual entries are inserted into LKH. In this case, the number of newly added individual entries to LKH is *nA*, where $nA = AS - RS$.
- ***AS* < *RS***. The *AS* new individual entries replace *AS* removed individual entries, then the rest of the removed individual entries are deleted from LKH. In this case, the number of deleted individual entries from LKH is *nR*, where $nR = RS - AS$.

The LKH rekey sub-tree, denoted *rekeyTree,* is constructed to contain the keys that are affected by the replacement, the insertion, or the deletion of the updated leaf entries. In addition, *rekeyTree* contains the keys to be sent to the refreshed members (*FR* requests in the batch). An inserted, deleted or refreshed leaf entry LKH *position* determines the set of keys that are inserted in the rekey sub-tree. For example, Fig. 48 illustrates a B^+-LKH and batch of 4 add requests, 2 remove requests, and 2 refresh requests. The two remove requests positions are marked "Rplc" for replacing by 2 add requests, the other 2 add requests positions are marked "Add", and the 2 refresh requests positions are marked "Rfrsh". The key nodes that are inserted in the rekey sub-tree for such batch of updates are grayed. Note that, a new key node "$K_{3.3}$" is inserted to the original LKH to

accommodate the new entries. Please consult appendix C for the detailed B^+-LKH *rekeyTree* construction example.

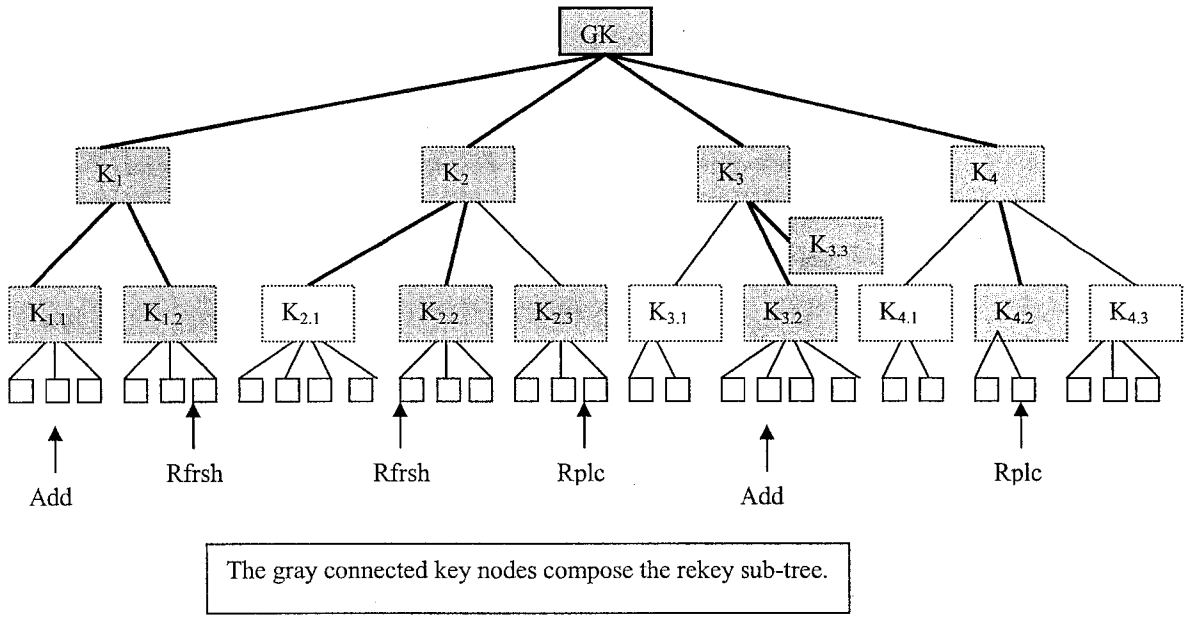


Fig. 48. An Example of a B^+-LKH, a batch of requests, and a rekey sub-tree.

There are four possible values of the rekey condition RC in a rekey policy that require LKH key changes as follows:

- PBS: a new member shouldn't be able to recover previous group keys (before he joins).
- PFS: a removed member shouldn't be able to recover new group keys (after he leaves).
- PBaFS: both above conditions should be satisfied
- NONE: no secrecy is required but LKH maintenance is necessary.

The *rekeyTree* is a LKH sub-tree that contains all LKH keys that need to be updated (i.e., regenerated and distributed to group member) for a batch of requests and the rekey condition RC determined from the rekey policy should always be satisfied. According to RC value, a *rekeyTree* key node is either unlabeled or labeled by one of three labels "A", "GA", and "GR". If XORBP KDT is used, the rekey packet is constructed the same way for all labeled keys as described in section 3.4.1. If encryption-based KDT is used, the key label determines how a rekey packet for distributing that key is constructed. If encryption-based KDT is used, the rekey packet is constructed for a *rekeyTree* key node, according to its label, as follows:

- No label: no rekey packet is constructed for that key.
- "A": construct a rekey packet that contains the key encrypted with every child key inserted in the *rekeyTree*.
- "GA": regenerate the key then construct a rekey packet that contains the newly generated key encrypted with its previous version, and with every child key inserted in the *rekeyTree*.
- "GR": generate the key then construct a rekey packet that contains the newly generated key encrypted with every child key in the original LKH.

Leaf key nodes inserted in the *rekeyTree* are always not labeled, (no packets are constructed for them) but they are used if their immediate parent is labeled "A" or "GA" as described above. When inserting a key in *rekeyTree* that already exists its label could be upgraded. The possible labels have the following precedence from lower to higher ("no label" < "A" < "GA" < "GR"). If the inserted key node (that already exists) is marked with a lower precedence label then it is upgraded, otherwise it is kept unchanged.

5.4.1 Rekey Message for a Batch of Requests

The format of the batch rekey message (RM) is illustrated in Fig. 49, where Addsize, RemoveSize, and RefreshSize is the number of Add, Remove, and Refresh requests in the batch, respectively. Other message fields are explained next.

SEQ	Add size	Remove size	Refresh size
$ReplacedPosition_1$, $ReplacedPosition_2$,...			
$RefreshedPosition_1$, $RefreshedPosition_2$,...			
$Add/RemoveHeader_1$, $Add/RemoveHeader_2$,...			
$RekeyPacket_1$, $RekeyPacket_2$,...			

Fig. 49. The batch rekey message (RM) format.

The following is the general procedure for constructing the rekey sub-tree (*rekeyTree*) and batch RM for *batch* of requests (for both S-LKH and B^+-LKH rekey protocols).

1. The *rekeyTree* root is initialized to contain the group key *GK* with no label.
2. Find the *position* of every replaced entry (added leaf entry in place of a removed leaf entry), replace the leaf entry in the original LKH and insert all the LKH keys in the path of that *position* in the *rekeyTree*. The leaf key node has no label, while the *label* of all internal key nodes (including the root that contains GK) depends on the policy rekey condition, RC as follows.

```
if (RC equals PBS) then label = "GA";
    else if (RC equals NONE) then label ="A";
            else label = "GR";
```

   In addition, an initial key message *initMsg* is constructed for every new member that contains his ID, position, LKH height, and LKH degree. Every replaced entry *position* is appended to the batch RM in the ReplacedPosition filed shown in Fig. 49.
3. Find the *position* of every refreshed entry, refresh the entry in the original LKH (update the individual entry changed data) and insert all the keys in the path of that *position* in the *rekeyTree* with the internal key node labeled "A". In addition, an initial key message *initMsg* is constructed for every refreshed member that contains

his ID, position, LKH height, and LKH degree. Every refreshed entry *position* is appended to batch RM in RefreshedPosition field shown in Fig. 49.

4. If the number of added entries *nA* is greater than zero (i.e., *nR* =0). For every added individual entry, add the individual leaf entry to the original LKH without any key generation and rekey packets construction. The S-LKH or B^+-LKH *AddMember* method is called without new keys generation or rekey packets construction (underlined code in Fig. 24 and Fig. 29). In batch rekeying, the *AddMember* method returns the *initMsg* and the header of the *rekeyMsg* (all fields except the rekey packets) that is appended to RM shown in Fig. 49. Insert all keys corresponding to such leaf entry insertion to the *rekeyTree* according to the rekey condition RC, and *position*, *type*, and *level* from the header of the *rekeyMsg*. Please consult appendix C for B^+-LKH *rekeyTree* labeled insertion of key nodes.
5. If the number of removed entries *nR* is greater than zero (*nA* =0). For every removed individual entry, remove the individual leaf entry from the original LKH without any keys generation or rekey packets construction. The S-LKH or B^+-LKH *RemoveMember* method is called without keys generation or rekey packets construction (underlined code in Fig. 25 and Fig. 34). In batch rekeying, the *RemoveMember* method returns the header of the *rekeyMsg* that is appended to RM shown in Fig. 49. Insert all keys corresponding to such leaf entry deletion to the *rekeyTree* according to the rekey condition RC, and *position*, *type*, *level*, and *isRight* array (only in B^+-LKH) from the header of the *rekeyMsg* Please consult appendix C for B^+-LKH *rekeyTree* labeled insertion of key nodes.
6. Send the above constructed *initMsg*s to all newly added members, and refreshed members. Construct the batch RM (shown in Fig. 49) that will be sent to all group members. The batch RM contains the positions of the replaced and refreshed entries, and the headers of the added/removed leaf entries. In addition, a rekey packet is constructed for every key node in the *rekeyTree* according to its label. The *rekeyTree* is parsed in post-order when constructing the rekey packets where the children of a node are visited before their parent.

5.5 Experimental Results

The following experiments are performed to compare the performance of S-LKH and B^+-LKH batch rekey costs with change of group dynamics (section 5.5.1), and change of LKH degree (section 5.5.2) for the same group size and batch size. A batch rekey cost is represented as the number of rekey packets in that batch rekey message (RM). If XORBP KDT is used, the number of rekey packets in a RM is a good rekey cost metric (all packets constructed the same way). If encryption-based KDT is used, each rekey packet contains a varying number of encrypted keys. The minimum number of encrypted keys in such rekey packet is 2, and the maximum is the LKH degree *d*.

The group dynamics is as defined in chapter IV. For a specified LKH degree *d*, group size *n*, and group dynamic ratio *gdr*, the LKH is constructed by adding aN members then removing rN members such that $n = aN - rN$ and $rN / aN = gdr$. In the following experiments, the batch size represents the number of replaced and/or refreshed leaf entries, while we assume the number of added and removed entries are zeros. For a constructed LKH, a batch rekeying is initialized with the specified batch size where the replaced and/or refreshed entries positions are randomly chosen. The following figures plot the average of 10 readings of the number of rekey packets in a RM (very small variance is noticed).

5.5.1 Effect of Group Dynamics

This experiment illustrates the effect of increasing the group dynamics on batch rekeying performance for both S-LKH and B^+-LKH protocols. The following figures show three horizontal lines *n*/2, *n*/*d* and average (*n*/2, *n*/*d*). Such lines help in analyzing the rekey cost for encryption-based KDT. The line *n*/2 marks the number of rekey packets in the best scenario for which the performance of LKH is the same as the performance of a star key management (*n* encrypted keys) described in chapter III, where each rekey packet contains exactly 2 encrypted keys. The performance of an LKH key management protocol with encryption-based KDT is worse than the star key management for all points above this line. The line *n*/*d* marks the number of rekey packets in the worst-case scenario (i.e., each rekey packet contains exactly *d* encrypted keys). The average line marks the average case scenario. The performance of an LKH key

management with encryption-based KDT is better than a star key management for all points under the n/d line.

This experiment illustrates the rekey cost of B^+-LKH versus S-LKH, where LKH degree is 4 and group size n = 8192, for different batch sizes and group dynamics. The batch sizes are 10%n, 20%n, ..., and 100%n. Fig. 50 illustrates the rekey cost for B^+-LKH (denoted B+) versus S-LKH (denoted S) when the group is static (gdr = 0). We can observe that for a degree 4 LKH and static group, the use of B^+-LKH introduces an increase in the rekey cost when compared to S-LKH rekey cost. In addition, we can observe that the average rekey performance of a LKH with encryption-based KDT and large batch size (more than 30% n) is worse than the use of star key management. Fig. 51 illustrates the rekey cost for the same LKH degree and same group size when the group dynamics is increased to gdr = 0.5. We can observe that B^+-LKH exhibits almost the same rekey performance of S-LKH for small batch sizes, and outperforms S-LKH when the batch size increases. Moreover, we can observe that for higher group dynamics, the average rekey performance of a degree 4 LKH and encryption-based KDT is better than star key management for smaller batch sizes (less than 20% n).

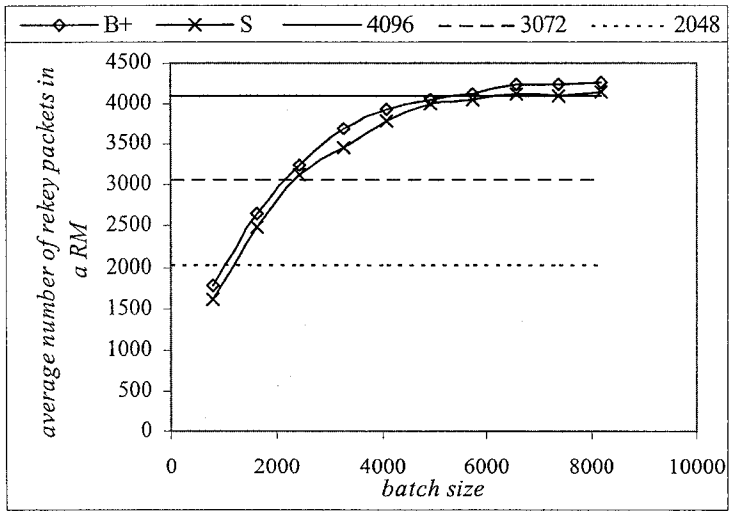


Fig. 50. B^+-LKH versus S-LKH rekey cost for $d = 4$, $n = 8192$, and $gdr = 0$.

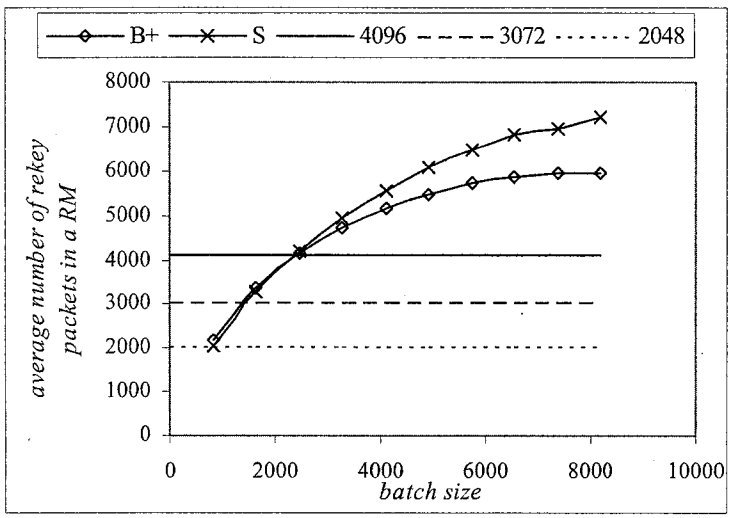


Fig. 51. B^+-LKH versus S-LKH rekey cost for $d = 4$, $n = 8192$, and $gdr = 0.5$.

Fig. 52 illustrates the performance of a degree 4 S-LKH rekey cost with the group dynamics increase, where *gdr* = 0, 0.2, 0.4, and 0.5, for different batch sizes and group size n = 8192. We can observe that S-LKH rekey cost increases with the group dynamics increase. Fig. 53 illustrates the performance of a degree 4 B^+-LKH rekey cost with the group dynamics increase for different batch sizes. We can observe that with the group dynamics increases, there is a smaller increase in B^+-LKH rekey cost compared to S-LKH rekey cost increase.

Performing the same experiment for degree 8 S-LKH and B^+-LKH. Fig. 54 illustrates the S-LKH rekey performance for the different group dynamics, and Fig. 55 illustrates the B^+-LKH rekey performance for the different group dynamics. We can observe that, for larger LKH degrees (more than 4), B^+-LKH rekey cost outperforms S-LKH rekey cost in all cases of batch sizes and group dynamics. In addition, from Fig. 54, we can observe that the average rekey cost of a degree 8 S-LKH with encryption-based KDT outperforms star key management for only small group dynamics (*gdr* = 0, 0.2) or small batch sizes (less than 30% n). On the other hand, from Fig. 55, we can observe that the average rekey cost of a degree 8 B^+-LKH with encryption-based KDT outperforms star key management for all batch sizes (up to 100% n) and all group dynamics. Moreover, we can observe that increasing the group dynamics for B^+-LKH protocol leads to a bounded increase in the rekey cost, while for S-LKH protocol the increase in the rekey cost steadily increases with the group dynamics.

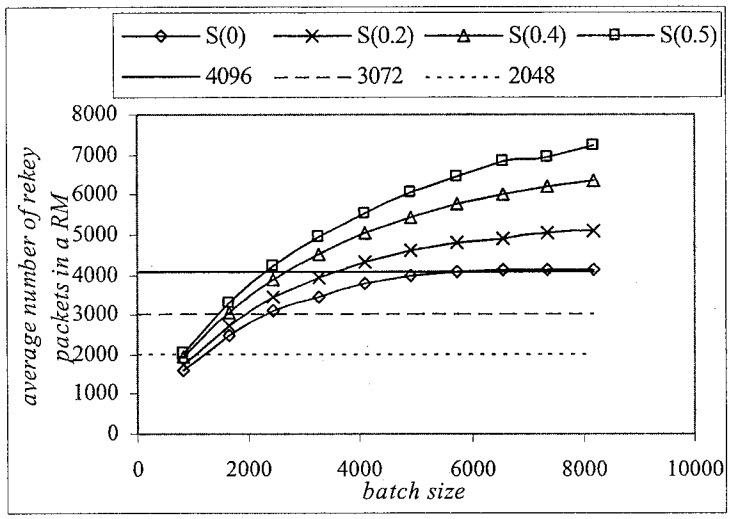


Fig. 52. Degree 4 S-LKH rekey cost (*gdr* = 0, 0.2, 0.4, 0.5).

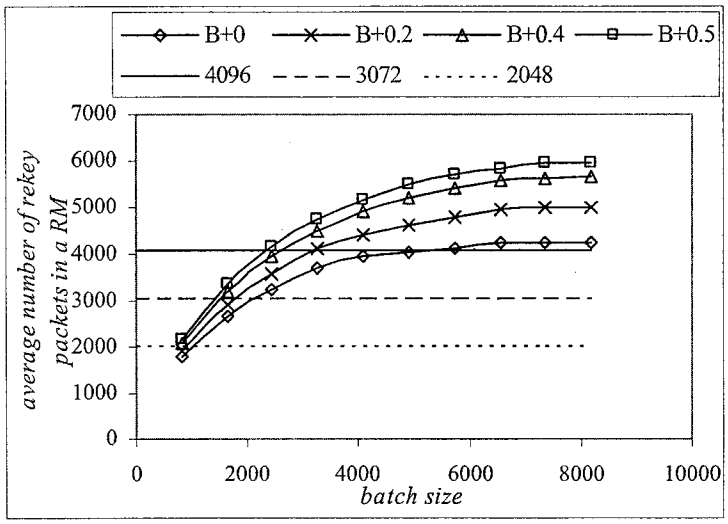


Fig. 53. Degree 4 B^+-LKH rekey cost (*gdr* = 0, 0.2, 0.4, 0.5).

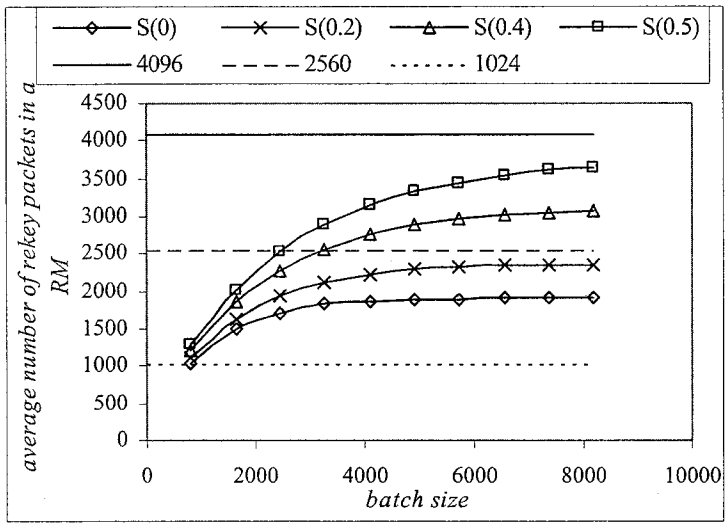


Fig. 54. Degree 8 S-LKH rekey cost (*gdr* = 0, 0.2, 0.4, 0.5).

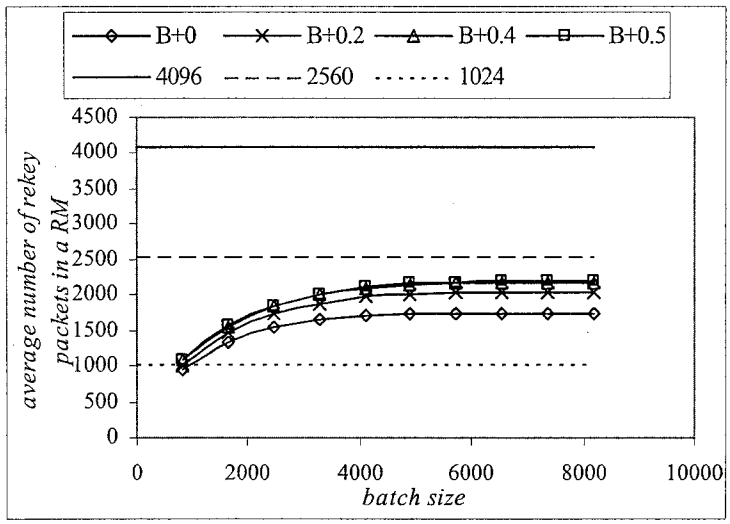


Fig. 55.Degree 8 B^+-LKH rekey cost (*gdr* = 0, 0.2, 0.4, 0.5).

5.5.2 Increasing LKH degree

In the previous experiment, we concluded that B^+-LKH rekey cost outperforms S-LKH rekey cost (for all batch sizes and group dynamics) for LKH degrees greater than 4. In this experiment, we study the effect of increasing LKH degree on the rekey cost represented as the average number of rekey packets in a RM. The LKH degree is increased from 4 to 32 in increments of 4.

First, the experiment is performed for group size n = 1024 and batch size 102 (10%n). Fig. 56 illustrates the change of S-LKH rekey cost with change of LKH degree for different group dynamics, where *gdr* = 0, 0.2, 0.4, and 0.5. We can observe that, the rekey cost is decreasing with LKH degree increase, while increasing with the group dynamics increase. Similarly, Fig. 57 illustrates the change of B^+-LKH rekey cost with change of LKH degree for different group dynamics. We can observe that, the rekey cost increase due to increased group dynamics is more bounded compared to S-LKH rekey cost increase (Fig. 56).

Assuming for the same parameters {*d*, *n*, *gdr*, and batch size} the S-LKH rekey cost is *cS* and the B^+-LKH rekey cost is *cB*. The S-LKH rekey cost percentile increase over B^+-LKH rekey cost (denoted *rci*) is calculated as $rci = (cS - cB) \times 100 / cB$. Fig. 58 illustrates the rekey cost percentile increase (*rci*) with change of LKH degree and different group dynamics. We can observe that the S-LKH rekey cost is greater than the B^+-LKH rekey cost for all LKH degrees greater than 4 (*rci* is greater than zero). The S-LKH rekey cost percentile increase (*rci*) peaks at certain LKH degrees, and usually increases with LKH degree increase and group dynamics increase (*rci* attains more than 50% when d = 12 and *gdr* = 0.5).

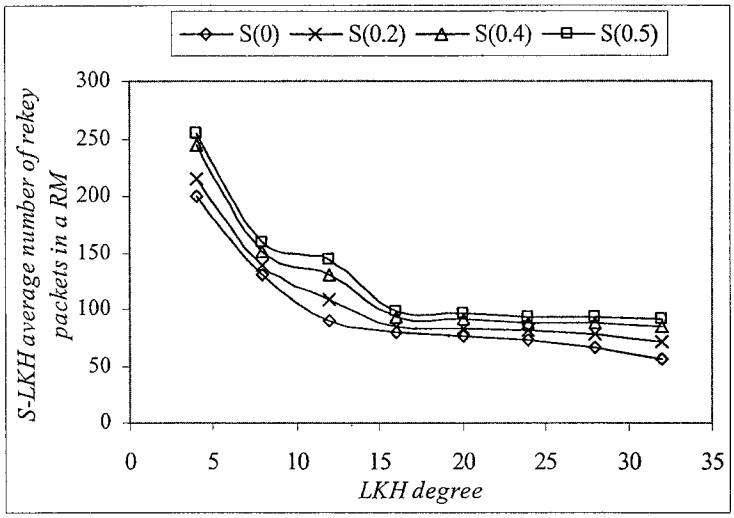


Fig. 56. A S-LKH rekey cost for different group dynamics (*gdr* = 0, 0.2, 0.4, 0.5).

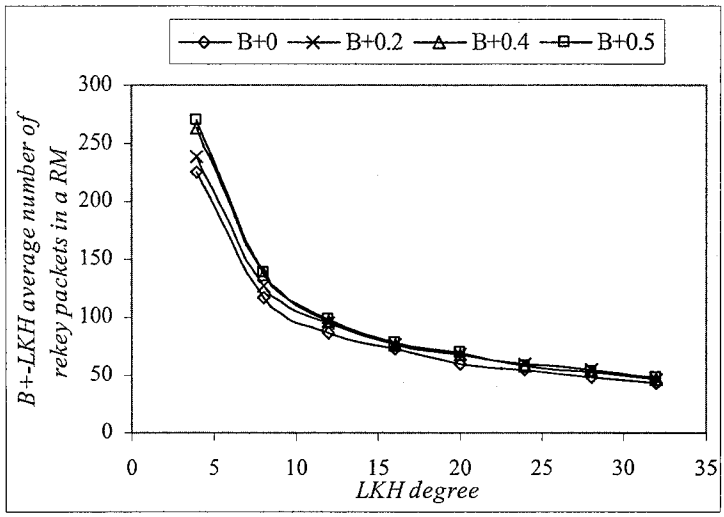


Fig. 57. A B^+-LKH rekey cost for different group dynamics (*gdr* = 0, 0.2, 0.4, 0.5).

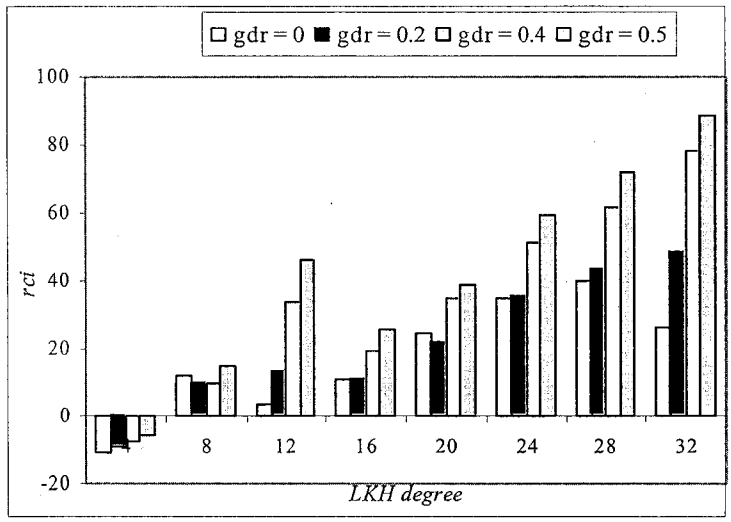


Fig. 58. A S-LKH rekey cost percentile increase (*rci*) over B^+-LKH, where n = 1024 and batch size = 102.

Next, the same experiment is performed with larger group size n = 8192 and the batch size is 819 (10% n). Fig. 59 illustrates the S-LKH rekey cost percentile increase over B^+-LKH rekey cost (*rci*) with change of LKH degree for four different group dynamics. Similarly, we can observe that the use of S-LKH introduces extra rekey cost over B^+-LKH for all LKH degrees greater than 4. This rekey-cost increase (*inc*) increases with the group dynamics increase. In addition, we can observe that this increase peaks at certain LKH degrees depending on the group size and batch size (peaks at different LKH degrees than what is shown in Fig. 58). The LKH degree that has a peak increase of S-LKH rekey cost over B^+-LKH rekey cost is the same for all group dynamics (for the same group size and batch size).

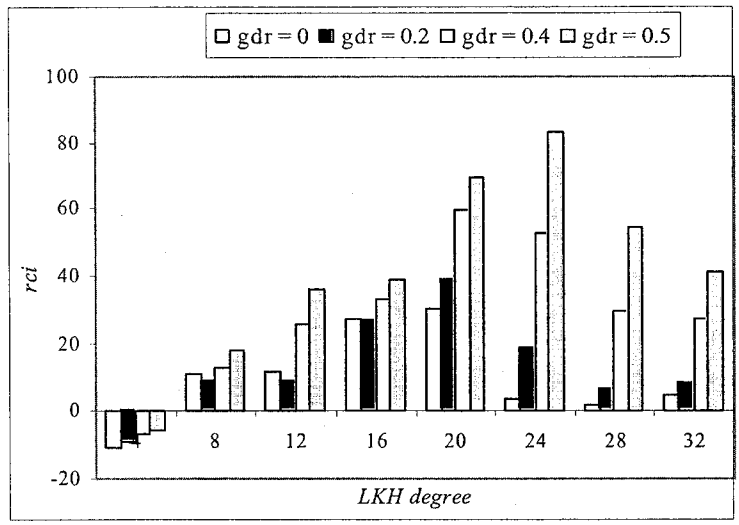


Fig. 59. A S-LKH rekey cost percentile increase (*rci*) over B^+-LKH, where $n = 8192$ and batch size = 819.

5.6 Conclusion

Researchers have suggested periodic rekeying to alleviate the problem of having very small inter-rekey period. A very small time between two consecutive rekeys does not allow a group key to be established and used by all group members. Periodic rekeying makes it essential to process a batch of requests. While periodic rekeying with a period greater than the rekey time solves the problem, it does not take into consideration the batch size, or the maximum request delay. In addition, simple periodic rekeying doesn't take into account the possibility of no requests being accumulated during an inter-rekey period.

In this chapter, a general and flexible rekey policy is presented. The defined rekey policy takes into account three parameters: minimum inter-rekey period, batch size, and maximum request delay. The policy has the flexibility of triggering the batch rekeying

process using all or a combination of these parameters. A simplified view of the software objects designed to provide secure group key management is presented. In addition, the batch rekey message (RM) and its construction in both S-LKH and B^+-LKH protocols is illustrated. Finally, experiments are performed to demonstrate that the B^+-LKH protocol introduces major rekey cost savings (less number of rekey packets) for a batch of requests compared to the S-LKH protocol. The B^+-LKH batch rekey savings compared to S-LKH increase with the increase of batch size or the group dynamics. In addition, we concluded that maintaining a balanced LKH (as a B^+-LKH) guarantees a bounded behavior with the increase of the group dynamics, while the performance of an unbalanced LKH (S-LKH) deteriorates with the increase of group dynamics.

CHAPTER VI

DISTRIBUTED GROUP REKEYING AND RECOVERY

In chapter III, we introduced a software model for secure group key management. We focused on the case of a central rekey manager that maintains the group key and performs group rekeying, when it deems necessary, according to a defined rekey policy. It is assumed that the rekey manager maintains a logical key hierarchy (LKH) for scalable rekeying. The existence of one rekey manager makes it a central point for both congestion and failure. Deploying a distributed set of rekey agents that equally share the load of group rekeying provides a more reliable and scalable solution. In addition, in applications which exhibit short failure time or disconnection times, e.g., mobile ad-hoc networks, a recovery mechanism is crucial to refresh the state of the group key management process. In this chapter, we discuss two enhancements to our group key management framework: distributed group rekeying and the recovery of a group key manager and a group member.

The chapter is organized as follows. Section 6.1 presents the distributed group rekeying protocols. Section 6.2 presents the proposed recovery mechanism for a group key manager/agent and discusses a group member recovery. Finally, section 6.3 concludes the chapter.

6.1 Distributed Group Rekeying

In this section, we present four cooperation protocols for distributed group rekeying between peer rekey agents. It is assumed that each rekey agent is capable of managing a subset of group members, and participating in the group rekeying process. We show that the rekey protocol with minimal overhead is that one rekey agent at a time generates and distributes a new group key to all group members. In addition, we detail the logical key hierarchy (LKH) maintained at a rekey agent for the different cooperation scenarios. If any rekey agent is required to distribute a group key to all group members, a naïve key management approach is that every rekey agent maintains (replicates) the group LKH. Instead, we propose the creation of agents' LKH (denoted A-LKH) that reduces the

replicated LKH size (compared to the naïve approach), and the number of maintained keys at a group member. Moreover, we discuss two different approaches for maintaining A-LKH namely dynamic A-LKH and static A-LKH. The dynamic A-LKH approach has a drawback of (sometimes) updating (some) group members for a rekey agent join or leave. On the other hand, the static A-LKH approach allows a transparent rekey agent join or leave for all group members, although the maximum number of rekey agents has to be known before starting a session.

The rest of this section is organized as follows. Section 6.1.1 is an overview of the distributed group rekeying approach between a group of rekey agents. Section 6.1.2 defines four different cooperation protocols between the rekey agents. Section 6.1.3 details the LKH maintained at a rekey gent for the different cooperation scenarios. Section 6.1.4 discusses the two different approaches for maintaining A-LKH.

6.1.1 Distributed Group Rekeying Overview

A distributed set of cooperating rekey agents provides a more scalable and reliable group rekeying than a central rekey manager. If an agent fails during a group session, other agents can assume its role and update the failed agent's subgroup members about group key changes (if allowed). In addition, a new agent can recover the state of a failed agent (recovery is discussed in section 6.2).

Consider a set of peer rekey agents, i.e., all agents have the same authority and capability of accessing, generating, and distributing the group key as well as any LKH key. Since all rekey agents have a full group rekey authority, there is no need to rekey the group (change GK) when an agent joins or leaves the rekey agents' group. A leaving agent is voluntarily relinquishes its responsibility (due to network disconnection or failure), but is still allowed access to further agents' communication. On the other hand, an evicted agent is not allowed any access to future agents or group communication. In this model, evicting a rekey agent is very expensive and would require recreating the group without that agent.

A rekey agent is responsible of managing a subset of the group members. Fig. 60 illustrates a rekey agents' group that manages a group of members, where every agent manages a different subset of the group members. At any point of time, there is one agent

who acts as the leader of the rekey agents' group, denoted LA. The LA is a rekey agent that is responsible for coordinating many group actions, such as the initiation of the group rekeying process. In addition, the group rekeying is performed for the LA's subgroup membership changes (i.e., members join and/or leave). An agent that exhibits a change in its subgroup membership has to nominate itself to be the leader to perform a rekeying. If there is only one rekey agent (in the rekey agents' group), it is assumed to be the LA until other agents join. Being a LA should be circulated fairly among all rekey agents. Choosing a leader among a group and guaranteeing there is only one leader at a time is the classical distributed systems mutual exclusion problem [17].

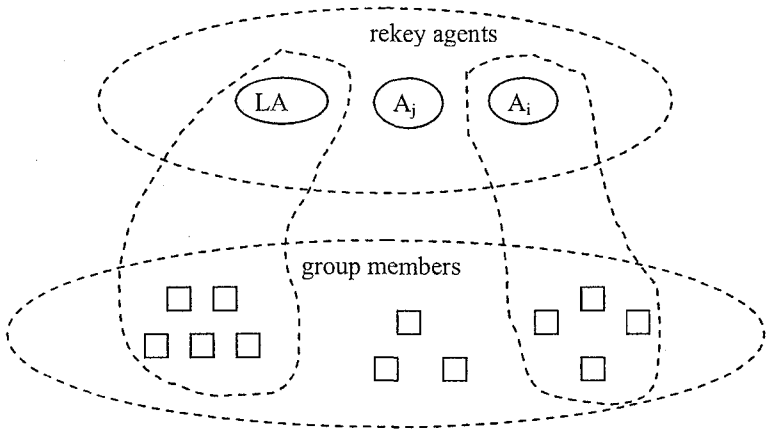


Fig. 60. Rekey agents and group members.

A rekey agent can join the rekey agents' group at any time (usually before the start of a group session). Initially, a rekey agent broadcasts its desire to join the agents' group to an agent-group channel prompting a response from the LA. The LA provides the initial status and (LKH) information. In addition, the LA informs other rekey agents of the new agent joining. A rekey agent is assumed to be active before any member joins its subgroup.

When a group member joins, he is assigned one rekey agent to be under its supervision (each member is supervised by only one rekey agent). There are several approaches for a client to select one sever among a distributed set of servers as follows:

- The client contacts a directory server (could be the authentication manager) who directs him to his server according to a load balancing or a route optimization technique.
- All servers addresses are published and the client chooses the nearest to his location (in the network sense), at random, or any other selection criteria.
- The servers are inserted in subnets, and the clients contact their subnet server.
- A client can send his request to a servers' channel, all servers receive the request but only one will respond according to a specified policy decision (for example, the leader or the nearest to the member's network location).

In the following cooperation models, if all agents are required to participate in generating a key (group key or other), a key agreement protocol (KAP) is needed. The existing technique known as group Deffie-Hellman [61] defines different protocols for such key agreement. The Deffie-Hellman protocol for two members requires two messages to be exchanged between the two parties, whereas group Deffie-Hellman protocols require several rounds and exchanges between all parties.

6.1.2 Rekey Agents Cooperation Protocols

The main function of a central rekey manager is to generate the group key (GK) then distribute it to all group members (G). In distributed rekey management between a group of m rekey agents, every agent A_i is responsible of managing a subgroup SG_i, such that $\bigcup_{i=1}^{m} SG_i = G$. The group rekeying is performed for the elected LA's subgroup membership changes. Other agents' subgroup membership changes are not incorporated is such rekeying. There are four group rekeying cooperation protocols between a group of rekey agents in terms of key generation and distribution, namely, all generate and all distribute; all generate and one distributes; one generates and all distributes; one generates and one distributes. The following are the four possible rekey agents' cooperation scenarios.

6.1.2.1 All Generate and All Distribute

All agents participate in generating a new GK through a key agreement protocol (KAP) and participate in distributing it to the group members. The following is the all-generate-and-all-distribute rekey protocol. First, the LA sends *StartRekey* message (command) to all other agents to start the KAP. Second, the KAP proceeds until all agents agree on the new GK. The KAP might require several rounds and message exchanges. Finally, every agent (including the leader) distributes the new GK to its subgroup members. It is essential that, a rekey agent signs the GK distribution message so that the group members are able to authenticate its source.

$LA \rightarrow A_i$: StartRekey

$A_i \rightarrow A_j$: KAP messages

$A_i \rightarrow SG_i$: GK

6.1.2.2 All Generate and One Distributes

All agents participate in generating a new GK then the LA distributes it to all group members. The following is the all-generate-and-one-distribute rekey protocol[7]. The first two steps generate new GK through KAP. Then, the LA distributes it to all group members. This protocol eliminates the signature overhead performed by each rekey agent to GK distribution message in the all-generate-and-all-distribute rekey protocol. Only the LA signs the GK distribution message sent to all group members.

$LA \rightarrow A_i$: StartRekey

$A_i \rightarrow A_j$: KAP messages

$LA \rightarrow G$: GK

[7] $X \rightarrow Y$: M, denotes X sends Y a message M.

6.1.2.3 One Generates and All Distribute

The LA generates a new GK, and all agents participate in distributing it. The following is the one-generate-and-all-distribute rekey protocol in two steps. First, the LA sends a *StartRekey* message to every agent along with the newly generated GK. Second, every agent (including the LA) distributes the new GK to its subgroup members. This protocol eliminates the KAP phase.

LA → A_i: StartRekey, GK

A_i → SG_i: GK

6.1.2.4 One Generates and One Distributes

The LA generates and distributes a new GK to all agents and to all group members. This is the minimal overhead rekey protocol that reduces the overhead incurred in both the KAP phase and the GK distribution message signature required if all agents are participating in the rekeying process. Note that, the rekey agents are taking turns in being the LA.

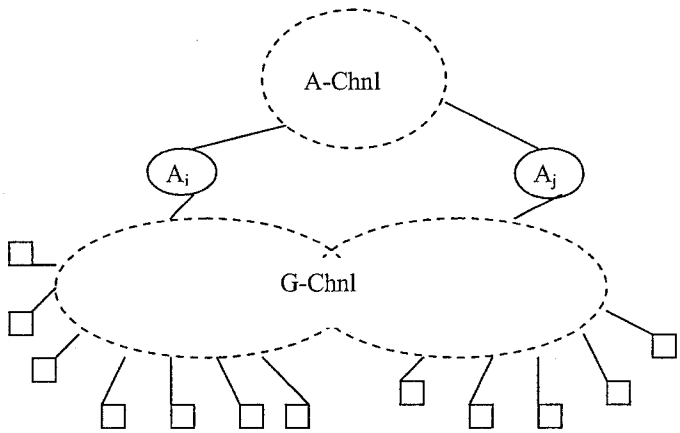
LA → A_i & G: GK

6.1.2.5 Comparison of Distributed Group Rekeying Protocols

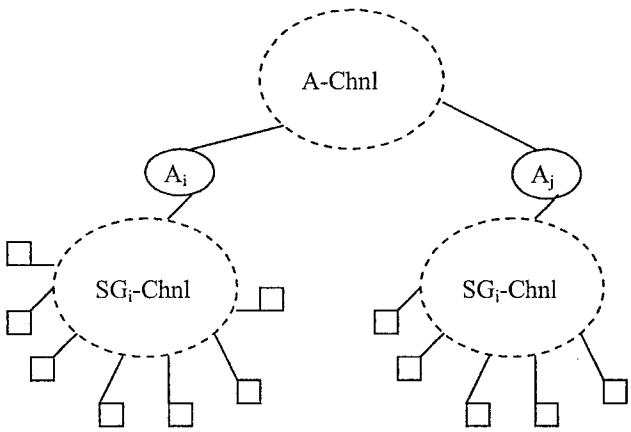
We can observe that the first rekey protocol that allows all rekey agents to participate in generating and distributing the group key in every rekeying requires the maximum overhead of both the key agreement protocol phase and the signature of GK distribution message performed by every rekey agent. The second rekey protocol that allows all rekey agents to participate in generating the group key, but the LA distributes it to all group members eliminates the signature of GK distribution message for all other agents. The third rekey protocol that allows the LA to generate a new GK, then every rekey agent distributes it to a subset of group members reduces the overhead incurred in the key agreement protocol phase that requires exchange of several messages. The fourth rekey protocol that suggests the LA generates and distributes a new GK to other rekey agents and all group members provides a minimal overhead rekey protocol for faster rekeying

process. The fairness in participating in the rekeying process between all rekey agents can be guaranteed through the leader selection mechanism.

In all cooperation scenarios, it is assumed that all rekey agents are communicating through an agent secure group channel (A-Chnl). In addition, all rekey agents and all group members are communicating through a secure rekey channel, G-Chnl, as illustrated in Fig. 61(a). In all-agents-distribute rekey protocols, every rekey agent instead can have its own independent subgroup rekey channel, SG-Chnl, as illustrated in Fig. 61(b).



(a) All members join the same group rekey channel (G-Chnl).



(b) Each subgroup members join different subgroup rekey channel (SG-Chnl).

Fig. 61. Communication channels between the rekey agents and the group members.

6.1.3 Distributed Group LKH Maintenance

For a group of n members managed by m rekey agents, the subgroup managed by a rekey agent is assumed to be of size (n/m). A logical key hierarchy (LKH) is used to provide scalable GK distribution. The rekey agents' cooperation model determines the LKH maintained at every agent. We will illustrate the LKH maintained at a rekey agent and the keys maintained by its subgroup members for the two different GK distribution cases: 1) all agents participate in distributing a new GK each to its subgroup members; 2) one agent at a time (the LA) distributes a new GK to all group members and to other rekey agents.

We will illustrate different LKHs of degree $d = 2$, where the group size $n = 32$, managed by 4 rekey agents (i.e., $m = 4$), and a rekey agent subgroup size is 8 members.

6.1.3.1 All Agents Distribute

In all-agents-distribute rekey protocols, every rekey agent participates in distributing a new GK to its subgroup members. It is sufficient for an agent A_i to maintain a LKH for its subgroup SG_i. There is no need for the rekey agents to share (replicate) any key information other than GK. In this case, every group member maintains his individual key and in the average $\log_d(n/m)$ keys, where n is the group size, m is the rekey agents' group size, and d is the LKH degree.

When $n = 32$ and $m = 4$, Fig. 62 illustrates the LKH (of height $h = 3$) maintained at a rekey agent for 8 members, where GK is the only replicated key at every rekey agent. A group member maintains 4 keys including his individual key and GK.

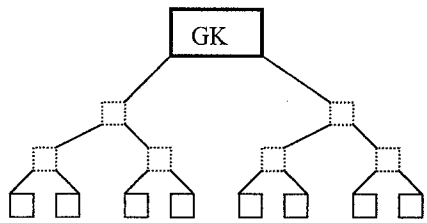


Fig. 62. A subgroup LKH of degree 2 for 8 members.

6.1.3.2 One Agent Distributes

In one-agent-at-a-time-distributes rekey protocols, the LA distributes a new GK to all other agents and to all group members. The naïve key management solution is every rekey agent maintains a fully replicated LKH for all group members. In this case, a group member maintains his individual key and in the average $\log_d(n)$ keys, where n is the group size, and d is the LKH degree.

When n = 32 and m = 4, Fig. 63 illustrates the group LKH (of height h = 5) for 32 members that is replicated at every rekey agent. A group member maintains 6 keys including his individual key and the group key.

The naïve solution requires a full replication of the group LKH rooted at GK. Alternatively, we suggest a more replication conservative solution. In the new approach, a rekey agent A_i maintains its subgroup LKH rooted at a rekey agent individual key AK_i. In addition, all agents replicate an agents' LKH (denoted A-LKH) rooted at GK. The leaf nodes of A-LKH are the agent keys AKs. The A-LKH and the subgroup LKHs are either having the same degree or having different degrees. Note that, A-LKH keys are replicated and known to all rekey agents including the agents' (individual) keys AKs. In this approach, a group member maintains an extra set of A-LKH keys starting from his agent individual key to GK. A group member maintains his individual key and in the average $\log_{d1}(n/m)$ subgroup LKH keys and $\log_{d2}(m)$ A-LKH keys, where n is the group size, m is the rekey agents' group size, *d1* is the subgroup LKH degree, and *d2* is the A-LKH degree. This approach allows any rekey agent to distribute a new GK to all group members but reduces the replicated LKH size at a rekey agent and the number of keys maintained at a group member when compared to the naïve solution.

When n = 32 and m = 4, Fig. 64 shows the A-LKH (of height 2) and the subgroup LKH (of height 3) maintained at agent A_1, where $d1 = d2 = 2$. A group member maintains 6 keys: his individual key, 2 subgroup-LKH KEKs, an agent key AK_1, 1 A-LKH KEK, and GK.

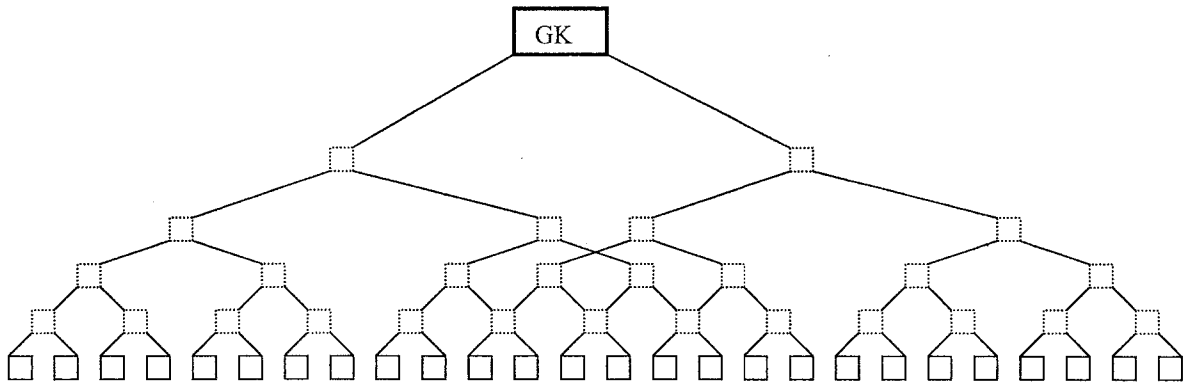


Fig. 63. A group LKH of degree 2 for 32 members.

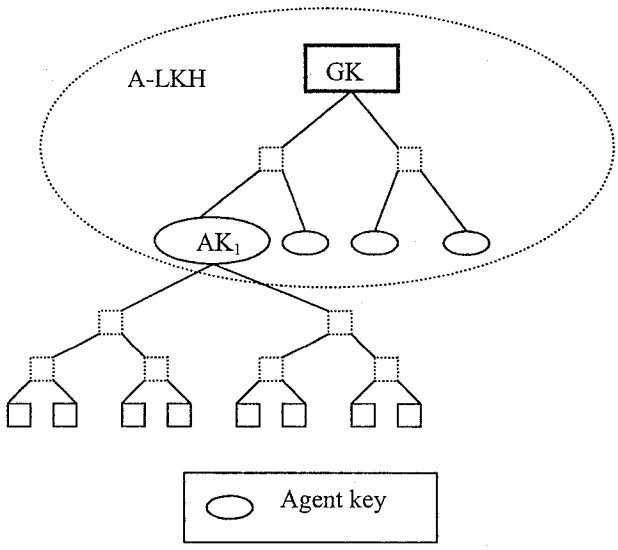


Fig. 64. An A-LKH and subgroup LKH maintained at rekey agent A_1 for 32 members.

6.1.4 Agents' LKH (A-LKH) Maintenance

The agents' LKH (A-LKH) is fully replicated at all rekey agents. There are two approaches for A-LKH maintenance: dynamic or static. In the dynamic approach, the A-

LKH is dynamically built up as the rekey agents join the agents group. In the static approach, the A-LKH is initiated to be of fixed static size that could accommodate the maximum number of rekey agents as they join. A newly joined agent contacts the LA to get the latest version of A-LKH. The A-LKH replica should be consistently updated at all agents through the agents' group communication channel. In the following sections, the two A-LKH maintenance approaches will be presented in detail, in addition to how an A-LKH key can be generated.

6.1.4.1 Dynamic A-LKH

In the dynamic A-LKH maintenance approach, the first rekey agent to start creates GK and its subgroup LKH. The A-LKH contains only GK, and it is considered the first agent individual key. The A-LKH dynamically grows as other rekey agents join the agents' group. There is no need to regenerate an existing A-LKH key (including GK) as agents join (the whole A-LKH is known to all agents). When an agent joins the agents' group, A-LKH keys are created to accommodate the new agent individual key (leaf A-LKH node). The LA notifies other rekey agents to update their replicated A-LKH. The new agent creates and maintains its subgroup LKH rooted at its newly created agent key.

In the dynamic A-LKH approach, creating a new A-LKH key requires updating (some) group members. As previously mentioned, evicting an agent is not valid (section 6.1.1). When an agent leaves, its individual key is deleted from A-LKH (that might lead to the deletion of other A-LKH keys). Similarly, when a rekey agent leaves, there is no need to regenerate an existing A-LKH key. The deletion of an A-LKH key requires updating (some) group members. This model has the drawback of sometimes affecting some group members as A-LKH keys are created or deleted.

Fig. 65 is an example that demonstrates the sequence of A-LKH key creation for 4 rekey agents, where A-LKH degree is 2. In Fig. 65(a), the first agent A_1 creates A-LKH that contains GK. In Fig. 65(b), the second agent A_2 joins, AK_1 and AK_2 are created. In this case, AK_1 should be sent to the subgroup members managed by A_1 (assuming no members have joined A_2 yet). In Fig. 65(c), the third agent A_3 joins, and K_1, K_2, and AK_3 are created. In this case, K_1 should be sent to the subgroup members managed by A_1 and

A_2. In Fig. 65(d), the fourth agent A_4 joins, and AK_4 is created. In this case, none of the group members is updated for such join.

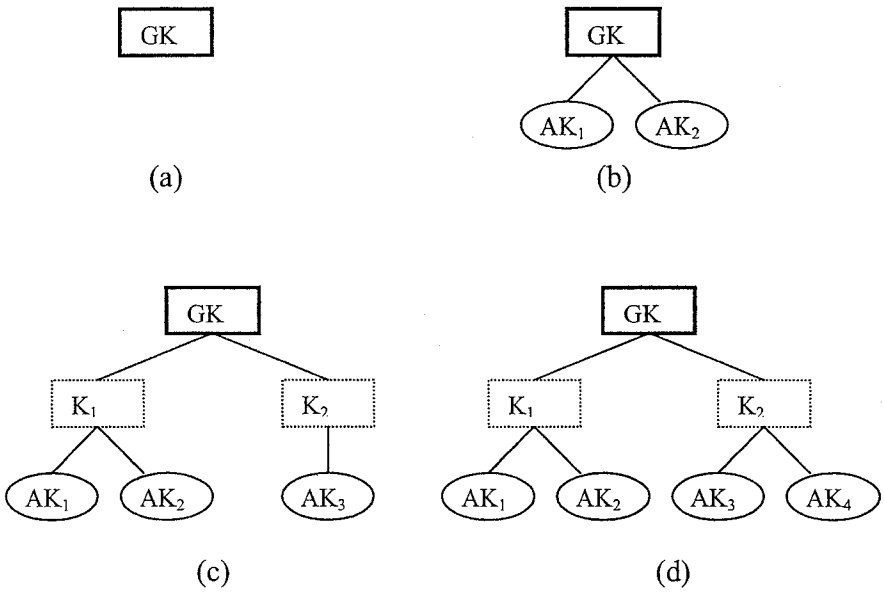


Fig. 65. Sequence of a dynamic A-LKH, key creation for 4 rekey agents.

6.1.4.2 Static A-LKH

The static A-LKH maintenance approach provides a transparent rekey agent join and leave for all group members, i.e., no members are updated for an agent join or leave. The first agent to start creates an empty (no keys) A-LKH that can accommodate a specified maximum number of agents (leaf nodes). It generates its own agent key AK (in a A-LKH leaf node), GK (A-LKH root node), and all the keys in the path between its AK and GK. When other agent joins, a newly generated AK is inserted into an empty A-LKH leaf node, and other A-LKH keys are generated as needed. When an agent leaves, only its AK is deleted (A-LKH leaf node is marked empty) allowing other agent keys to be inserted.

There is no need to regenerate an existing A-LKH keys as agents join or leave. The static A-LKH maintenance approach has a drawback that the maximum number of rekey agents has to be known before starting a session.

Fig. 66 is an example that shows the sequence of A-LKH key generation for 4 agents, where A-LKH degree is 2 and the maximum number of rekey agents is 4. In Fig. 66(a), the first agent to join generates AK_1, K_1, and GK. In Fig. 66(b), the second agent joins and AK_2 is generated. In Fig. 66(c), the third agent joins, and AK_3 and K_2 are generated. In Fig. 66(d), the fourth agent joins and AK_4 is generated.

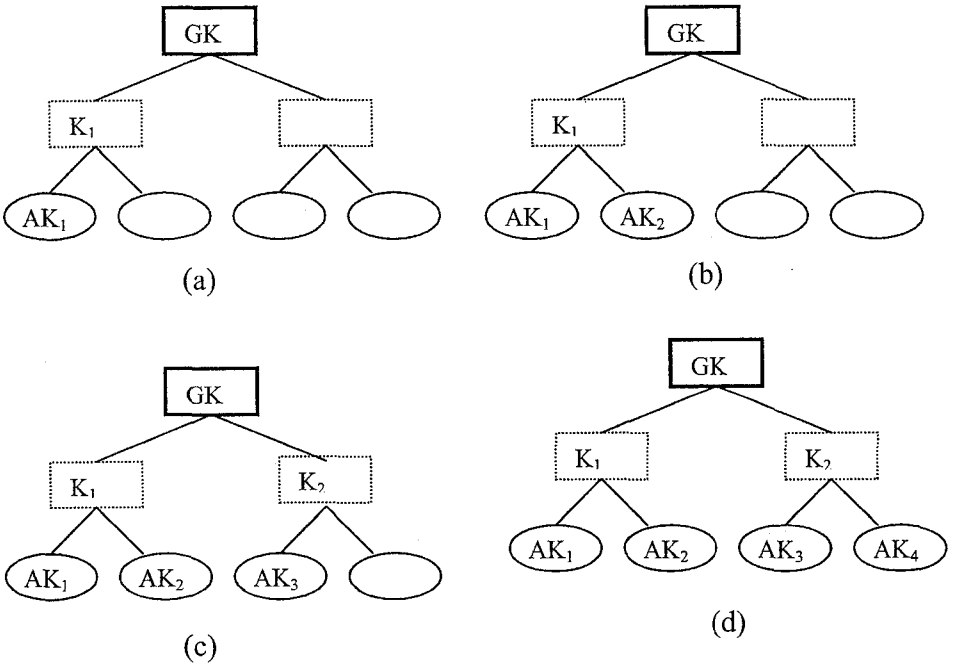


Fig. 66. Sequence of a static A-LKH key generation for 4 rekey agents.

6.1.4.3 A-LKH Key Generation

An A-LKH key to be used the first time by an agent is created by one of the following methods:

1) Creation by the leader agent (LA)
2) Creation by the new agent (NA) itself
3) Creation by both the LA and the NA through KAP
4) Creation by all agents through KAP

The following are the protocols for the four aforementioned cases.

Creation by the LA

The LA sends the NA the updated A-LKH after creating/generating the required keys. At the same time, the LA sends an update A-LKH message to all other agents.

LA → NA: A-LKH; LA → A_i: Update A-LKH

Creation by the NA

The LA sends the NA the A-LKH before the creation of any new key. The NA updates A-LKH and sends the update to all agents including the LA. This protocol requires two messages to be sent in sequence.

LA → NA: A-LKH

NA →A_i: Update A-LKH

Creation by both the LA and the NA

The LA sends the NA the A-LKH along with its share in the newly generated keys. Then, the NA sends back its share in the newly generated keys to the LA. Both the LA and the NA update A-LKH with the new keys. Then, the LA sends an update A-LKH to all other agents. This protocol requires three messages to be sent in sequence.

LA → NA: A-LKH, new-keys-share

NA → LA : new-keys-share

LA → A_i : Update A-LKH

Creation by all agents

The LA sends the NA the A-LKH, and sends to all other agents a *StartRekey* message for the required A-LKH keys (at least one). All agents (including the LA and the NA) exchange messages for the new keys generation. After the KAP proceeds all agents will be able to establish the same updates to A-LKH.

LA → NA: A-LKH; LA → A: StartRekey

A_i → A_j: KAP messages

We can observe that the first protocol is the simplest (fastest) since updating an A-LKH requires the LA to send two messages at the same time, one to the NA and one to the other rekey agents. However, choosing a protocol for A-LKH key creation/generation can be a group policy decided by the application.

6.2 Group Key Manager Recovery

In this section, the recovery of a group key manager and a rekey manager after short failure time is discussed. It is assumed that, the rekey manager is a software entity maintained by the group key manager, i.e., the rekey manager fails and recovers as a component of the group key manager. Such recovery process is concerned with the recovery of the last state of the rekey policy, the rekey scheduler, and the LKH. One approach to recover the state a failed group key manager is to have an independent full replica(s) of its state that assumes responsibility upon its failure. The drawback of this approach is the extra overhead needed to keep all replicas consistent all the time. Instead, we assume that the group key manager state is not replicated. We are concerned with the state recovery of a central group key manager that maintains group LKH as well as a group key agent that maintains a subgroup LKH and possibly an agents' LKH (A-LKH) after a short failure time, e.g., due to a server restart. Although the recovery of a LKH could be performed using the state stored at group members, we introduce the use of a log file that facilitates such recovery in case of members' failure or inconsistencies. The proposed logging and recovery mechanism is secure and easy to implement. The logging system avoids writing any key or revealing random number generator information. Group

members participate in the recovery of their key manager/agent by sending at least one encrypted recovery message. The recovery message sent by a group member contains his maintained list of keys. We introduce a key selection technique for a group member to reduce the number of keys sent in the recovery message while allowing the group key manager to retrieve all LKH keys. To the best of our knowledge, this topic has not been previously investigated in the research community.

The rest of this section is organized as follows. Section 6.2.1 is an overview of the proposed recovery system. Section 6.2.2 illustrates the proposed group key manager logging system. Section 6.2.3 introduces the recovery key used by the group members in the recovery of their manager. Section 6.2.4 details the group key manager recovery procedure. Section 6.2.5 demonstrates the group member recovery message and introduces a key selection technique that reduces the overhead in constructing the recovery message.

6.2.1 Recovery Overview

The authentication manager that maintains the group policy is assumed to be implementing an independent fault tolerance mechanism. In addition, the authentication manager is assumed to store the group requests (add, remove, and refresh) sent to the group key manager until a rekeying is successfully ended (i.e., committed). Moreover, the authentication manger either keeps the group requests or denies all or some types of those requests during the group key manager failure.

The recovery of a group member after failure could be treated as him leaving the group and joining at a later time. If the group member failure is for a very short time and the leave request is not processed (waiting in a batch of requests), when the join request is received, the group member state is refreshed instead. As previously mentioned in chapter IV, refreshing a group member state assumes the member lost his maintained set of keys, and requires sending him the same keys as if he newly joined. However, the rekey manager doesn't change LKH keys for refreshing a group member. Such refreshing optimizes the rekeying process by reducing the number of the newly generated LKH keys. The mobile computing paradigm is an example where frequent short disconnection times may occur, due to frequent handoffs.

In distributed group rekeying, if the distributed agents' cooperation protocol allows any agent to distribute a new GK to all group members, the failed agent subgroup members will be notified by the changes of GK during their agent failure period. On the other hand, if the distributed agents' cooperation protocol allows every agent to distribute a new GK to its subgroup members only, the failed agent subgroup members could store the un-interpreted group messages (due to lack of GK updates) during their agent failure and proceed interactively after its recovery. In this case, if the agent failure is for a very short time, its subgroup members might be able restore communication appropriately. Otherwise, a failed agent subgroup members might loose interactivity with the session.

As previously mentioned in chapter III, the group rekey channel provides a reliable group communication (multicast) protocol that assures a group member has received the rekey message (RM). A RM send method call (through the rekey channel) is assumed to return successfully even if RM didn't reach some (or all) group members due to their failures. The new GK is guaranteed to reach the group member by the rekey channel.

The rekey scheduler and the leader selection mechanism guarantee that there are no nested rekeyings (i.e., no start-rekey is issued before the previous rekeying is committed).

6.2.2 Group Key Manager Logging

The group key manager is configured through the rekey policy to schedule the group rekeying events while receiving requests (from the authentication manager) to add, remove, and refresh group members. When group rekeying deems necessary, the rekey manager is notified to issue a rekey message (RM) and send it to the group members. The proposed recovery mechanism assumes the group key manager is maintaining a log file. The log file is written to permanent storage (disk) periodically and forcefully at certain checkpoints, so that any type of failure does not affect it. Note that, we are not considering disk or catastrophic failures.

Writing a LKH key to the log file is crucial and requires encryption that is time (and processing) consuming. In addition, the keys are subject to change in a rekeying process, and the most recent version of a key is the only needed version after the recovery. The recovery mechanism avoids writing keys to the log file. Moreover, the randomly generated numbers (such as keys, IDs, or byte patterns (BPs); see chapter IV) could be

regenerated if the used pseudo random number generator and its initialization are revealed to an intruder. It is crucial to store the initial pseudo random generator state (e.g., its seed) that would allow the generation of the exact sequence of random numbers. The recovery mechanism avoids storing such random number generators state information.

In summary, a group key manager/agent writes a time stamped entry to the log file in the following cases:

- Initialization entry that is used to restore the employed protocols, implementations, and policies,
- Receiving a message to add, remove, or refresh a group member,
- Before initiating a rekeying process (i.e., the leader agent (LA) in a distributed group rekeying model) a *Start-Rekey* entry is written,
- After committing an initiated (by itself) rekeying process, a *Commit-Rekey* entry is written,
- When committing a rekying process (i.e., not the LA in a distributed group rekeying model), a *Rekey* entry is written,
- Change of rekey policy, and
- A LKH signature at specified checkpoints.

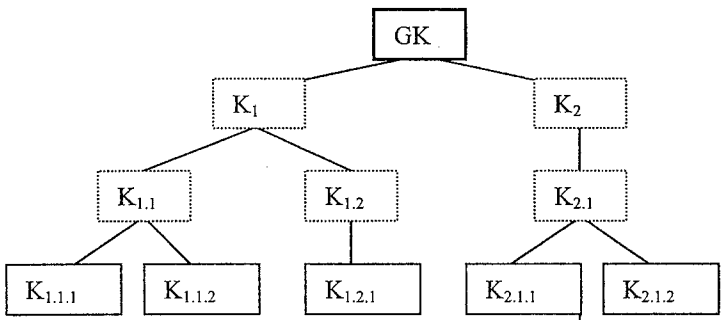
The log file is forcefully written to the permanent storage in the following cases:

- Initialization,
- Committing an initiated rekeying process, and
- A LKH signature written at specified checkpoints.

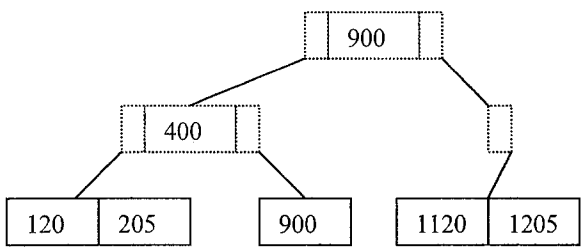
A checkpoint is introduced to facilitate the recovery process. The checkpoint could be scheduled periodically or after certain number of committed rekeyings. At a checkpoint, the rekey manager (governed by the group key manager) writes the LKH signature to the log file, and forcefully writes the log file to the permanent storage. The LKH should be checked to have updates since the previous checkpoint. In a distributed group rekeying, the agents' LKH (A-LKH) is not written to the log file since it is fully replicated at all agents and could be easily recovered. In the group key manager recovery, the LKH

signature determines the shape of the LKH, the number of entries at each node, and the guiding IDs. The following is the LKH signature of the LKH illustrated in Fig. 67. The LKH is parsed in pre-order and the IDs are written in order with the symbol "(" used to group a single node's entries.

T: LKH-Signature [((120, 205), 400, (900)), 900, ((1120, 1205))]



(a) The S-LKH key view.



(b) The S-LKH search view.

Fig. 67. A group LKH at a checkpoint time.

6.2.3 Recovery Key

A group member participates in the group key manager/agent recovery by sending some of his maintained keys as will be explained in section 6.2.5. For privacy purpose, a group member sends to the group key manager the recovery messages encrypted by a recovery key. The group key manager should be able to decrypt such messages, while no other group member should possess such capability. Using the group key, GK, as a recovery key, is not suitable, since the group members are aware of it. Instead, a group member either uses his individual key or a group key manager public key.

If the authentication manager stores the group members' individual keys, the group key manager contacts it at the beginning of a recovery process to obtain such keys (among other information). The group key manager recovers the group members' individual keys before receiving any recovery message from them. In this case, every group member uses his individual keys as a recovery key (to encrypt the recovery messages).

On the other hand, if the authentication manager doesn't store the group members' individual keys, a recovery key is needed. The recovery key has to be in the form of private key and public key pair. The recovery key could be a long-term key or a session recovery key. The private key is kept securely at the authentication manager or at the group key manager system. The public key is handed to every group member right after he joins the group to use as a recovery key.

6.2.4 Group Key Manager Recovery

The recovery of a group key manager/agent implies the restoration of the latest group/subgroup LKH, policy, scheduler state, and agents' LKH (if applicable). It is assumed that contact information to the authentication manager and other group key agents are recoverable (one could be through the other). The group key manager recovery process proceeds as follows:

1. Inspect the following log file entries: *Initialization* entry to reinitialize itself and the rekey manager; last *Rekey-Policy* entry to restore the rekey policy and adjust the scheduler; last *LKH-Signarure* entry to reestablish the group LKH structure.

2. Apply all committed rekeyings' changes to the LKH, i.e., insert and delete LKH nodes that took effect after last signature. Note that, without writing *LKH-Signature* the LKH could be restored by redoing all insertions and deletions form the beginning of the log file.
3. Contact the authentication manager for changes in the rekey policy (if allowed). In addition, the group key manager retrieves the stored requests at the authentication manager. If the last *Start-Rekey* entry in the log file is not followed by a *Commit-Rekey,* it is implied that the group key manager crashed during a rekeying. Although the exact scheduler state can't be recovered, the group key manager schedules a rekeying as soon as possible after LKH full recovery.
4. Contact the agents' group for latest agents' LKH (A-LKH), and the committed rekeyings during the failure period to adjust the sequential number SEQ. If all agents are not available during this recovery (e.g., all failed) and some rekeyings have been performed, the recovering agent subgroup members will provide partial construction of A-LKH and that will allow the recovering agent to proceed normally.
5. Send a recovery request to group members to send back their maintained list of keys to fully restore LKH keys (see section 6.2.5).

6.2.5 Group Member Recovery Message

A group member sends to his group key manager/agent one recovery message upon receiving a recovery request. The recovery message contains his individual LKH leaf entry position, his individual ID, last SEQ, and the maintained list of keys. In addition, the recovery message is encrypted using the recovery key as explained in section 6.2.3.

As previously illustrated for LKH keys, an individual key is maintained by one group member, GK is maintained by all group member, a KEK is maintained by a subset of the group members. If every group member sends all his maintained list of keys in the recovery message to the group key manager, GK and KEKs will be sent several times (e.g., GK will be sent by all group member). Instead, we propose an enhancement to the above protocol that allows group members to send a partial list of their maintained keys. Allowing only one group member only to send a recovered key is crucial if that member

fails. On the other hand, if all the members that maintain a key have failed, the group key manager will not be able to recover such key but will be able to proceed without it.

The proposed LKH keys recovery protocol provides a fair group member key selection that allows a group member to choose a partial list of his maintained keys to send to a recovering group key manager. In addition, it allows the group key manager to retrieve all keys in one round if no member fails. If some members fail during their group key manger recovery, LKH keys recovery might take two rounds as follows:

1st round

- The group key manager sends a recovery request to all group members.
- A group member sends an encrypted recovery message that contains his individual key (if not recovered from the authentication manager), his individual ID, and his LKH leaf entry position. Note that, if a group member didn't send a recovery message in the first recovery round, he is detected as failed by the group key manager.
- If the group member's individual entry falls on the path of the first child of a key node (determined from his LKH leaf entry position that equals to 1), send that key in the first round recovery message. The maximum number of keys a group member can send in a recovery message is half the LKH height, starting from the key on the 2nd LKH level (i.e., without his individual key).

2nd round

- If the group key manager didn't recover a LKH key (KEK) at the first round due to members' failure, a recovery request message is sent specifying the missing set of keys and the next existing neighbors (to the failed members) to send it.
- The specified group members send the specified keys.

We suggest that the above key selection algorithm is fair since an individual entry LKH position is determined from his randomly assigned individual ID. The probability of a group member sending a certain key is independent from any other key, and is equal to the probability of holding a key that exists in a first entry of a node that is equal to $1/d$, where d is the LKH degree.

For example, in the group key manager recovery process of the LKH of height 3 illustrated in Fig. 67, a group member will send a first round recovery message that

includes his ID, LKH leaf entry position, and at most 3 keys (assuming half 3 is 2) including his individual key. The five recovery messages sent by the five group members to the group key manager in the format (ID, LKH position, keys) are as follows: (120, 1.1.1, $K_{1.1.1}$, $K_{1.1}$, K_1), (205, 1.1.2, $K_{1.1.2}$, K_1, GK), (900, 1.2.1, $K_{1.2.1}$, $K_{1.2}$, GK), (1120, 2.1.1, $K_{2.1.1}$, $K_{2.1}$, K_2), and (900, 2.1.2, $K_{2.1.2}$, K_2).

6.3 Conclusion

Distributed group rekeying between a set of peer rekey agents provides a more scalable and reliable secure group key management compared to the central rekey manager approach. In this chapter, four group rekeying cooperation protocols between a distributed set of rekey agents, in terms of group key generation and distribution mechanism, are proposed. It is demonstrated that, the minimal overhead rekey protocol is when one rekey agent at a time generates and distributes a new group key to all agents and group members. In addition, the LKH maintained at a rekey agent in the two cases of new group key distribution are discussed. The first case is that each agent distributes the new group key to its subgroup members. The second case is that one rekey agent at a time distributes a new group key to all group members. The naïve solution in the latter case is that every rekey agent fully replicates the group LKH. Alternatively, we proposed the construction and replication of smaller size agents' LKH (A-LKH). The proposed approach reduces the replicated LKH size at each rekey agent and the number of keys maintained by a group member. Furthermore, we identified two approaches of such agents' LKH maintenance namely dynamic A-LKH and static A-LKH. The dynamic A-LKH approach has the drawback of affecting group members (by inserting or deleting keys) as agents join or leave the agents' group. The static A-LKH approach guarantees a transparent rekey agent join and leave but requires the specification of the maximum number of rekey agents before starting a session.

Moreover, a logging mechanism for the recovery of a group key manager/agent state after short failure time is presented. The logging includes all events that change the group key manager state but avoid writing any security revealing information such as keys. Group members participate in the recovery of their manager by sending an encrypted recovery message that includes a sub-list of their maintained keys. A fair group member

key selection technique is proposed to reduce the number of sent keys in a recovery message.

CHAPTER VII

CONCLUSION AND FUTURE EXTENSIONS

In this chapter, we conclude the dissertation by summarizing our motivation, objectives, contributions, and the performance of our proposed framework for secure group key management. Furthermore, we discuss a list of possible future extensions to our work in the context of secure group communication, and secure group key management.

7.1 Conclusion

Secure group communication is quickly becoming the adopted standard in many applications spanning diverse areas. Throughout the dissertation, we focused on secure group key management, which deals with group key (*GK*) issues such as establishing, distributing, and maintaining that key over the period of the group existence. To provide perfect secrecy, group rekeying (change of *GK*) has to be performed for every group member joining or leaving the group. Group rekeying is a challenging problem especially for large group sizes or highly dynamic groups.

The simplest group rekeying protocol is performed with the help of a trusted and secure group key manager. The group key manager maintains *GK*, and performs a group rekeying when it deems necessary according to a defined rekey policy. In a group rekeying process, a new *GK* is generated and distributed to group members such that a joining (leaving) member is not allowed access to previous (future) group communication. A very fast rekeying is crucial to the performance of an application that has large group size, experiences frequent joins and leaves, or the group key management is hosted by a group member because of the required computation effort. Traditionally, newly generated keys are encrypted for secure distribution to group members. Such technique is denoted encryption-based key distribution technique (KDT). There are two approaches for group key management, the star key management and the logical key hierarchy (LKH) approach. In the star key management, the group key manager performs 2 keys encryptions for join rekeying and n keys encryptions for leave rekeying, where n

is the group size. This approach is not scalable since leave rekeying scales linearly with the group size. In the LKH approach, if the LKH degree is d, the group key manager performs on the average $2 \times \log_d n$ keys encryptions for join rekeying, and $d \times \log_d n$ keys encryptions for leave rekeying. The LKH provids a scalable group rekeying, and is becoming the standard approach for group key management. However, when encryption-based KDT is used with LKH, there are two un-symmetric rekey protocols for join and leave rekeying. Such unsymmetric property makes increasing the LKH degree result in a decrease of the join rekey cost and an increase of the leave rekey cost. In this case, the optimal LKH degree is estimated to be 4.

Traditionally, group rekeying is performed periodically for the accumulated join and leave requests (i.e., batch of updates) during an inter-rekey period. In the star key management approach, the group key manager is required to regenerate one key and to perform O(n) key encryptions for a rekeying, where n is the group size. If the group key manager maintains a LKH of degree d and height h, such that $n \leq d^h$, and the batch size is R requests, a rekeying requires the group key manager to regenerate $O(R \times h)$ keys and to perform $O(d \times R \times h)$ keys encryptions. The encryption-based LKH approach provided a rekeying cost that scales to the logarithm of the group size, however, the number of encryptions performed by a GKM increases with increased LKH degree, LKH height, or the batch size, and can be more than the star approach's number of encryptions.

The objective of our work is to provide a framework for secure group key management that outperforms the original encryption-based LKH for all application scenarios. The framework has to be secure, efficient, scalable, reliable, and independent of the application. The group key management framework addresses the following issues: secure group communication software model, key distribution technique, rekey protocols, batch rekeying, distributed group rekeying, and recovery. We briefly present our approach to resolve the aforementioned issues highlighting our contributions.

Secure group communication software model. We presented a generic software model for providing secure group communication. The model identifies five main components as follows: authentication manager, group key manager, rekey manager and the corresponding rekey client, rekey channel, and cryptographic utility manager. The model is designed to isolate the group key management components and illustrate the

functionalities and interactions of other components. We have extended Java™ security with an application-programming interface (API) that can be used to provide group key manager, rekey manager, and rekey client functionalities as designed in our model.

Key distribution technique. We focused on the rekey manager that uses a LKH for scalable rekeying. We proposed a novel XOR-based KDT, namely XORBP. The proposed approach performs an XOR operation between keys to reduce the computation effort, and uses a random byte patterns (BP) to distribute the key material in a fixed size rekey packet (for every new key). The use of LKH and XORBP KDT provides symmetric rekey protocols in both cases of join and leave rekeyings.

We derived analytical cost estimates of XORBP and performed empirical experiments to compare its performance with the encryption-based KDT for the same degree LKH. The use of XORBP doubles the required LKH storage, the required member storage, and the number of randomly generated bits per a rekeying. The XORBP rekey message size is comparable to the encryption-based leave rekeying message size. On the other hand, the use of XORBP substantially reduces the rekey message construction time. Our experiments have shown that XORBP achieves up to 90% reduction in the rekey message construction time. In addition, contrary to the encryption-based KDT, increasing the LKH degree, when XORBP is used, reduces both join and leave rekeying cost. Such property allows the use of a larger degree LKH, which reduces the LKH storage, the member storage, and the rekey message size when compared to a smaller LKH degree. The analytical cost estimates assume that the LKH is balanced, while the experiments are performed using an un-balanced LKH. Such experiments show that there is a slight increase in the measured member storage and the rekey message size over the analytical values, but the measured LKH storage has a 60% increase over the analytical value.

Rekey Protocols. As group members join or leave the group, LKH nodes (keys) will be inserted or deleted. While, many researchers assume a balanced LKH when estimating the group rekeying cost, the literature lacks practical LKH protocols that maintain a balanced LKH of any degree all the time. We proposed two novel protocols for establishing and maintaining a LKH of any degree. One protocol adopts an unbalanced LKH while the other adopts a balanced LKH. The protocols assume that the rekey manager assigns a unique individual identification (ID) to every group member. For both

protocols, we detailed the LKH structure, the rekey message format, and the rekey processing at a rekey manager and at a rekey client for different scenarios of LKH keys insertion and deletions.

The first protocol, denoted S-LKH, maintains LKH as a search tree using the individual IDs. The second protocol, denoted B^+-LKH, maintains LKH as a balanced B^+ search tree that has the same structure as S-LKH. B^+ search tree insertion and deletion algorithms guarantee that the LKH is balanced after each node (key) insertion or deletion. In addition, B^+ search trees have an extra constraint that all allocated nodes have to be at least half full to reduce the allocated LKH storage (memory). On the other hand, B^+-LKH maintenance introduces complexity and extra overhead to the rekey process.

We have performed empirical experiments to compare the performance of S-LKH and B^+-LKH rekey protocols. The experiments show that, for both protocols, the frequency of the simple insertion and deletion scenarios increases with LKH degree increase. In addition, for B^+-LKH the frequency of the most expensive operation is less than 1% for any LKH degree. For individual rekeying (i.e., a rekeying after one group member joins or leaves), the use of B^+-LKH results in an increase in the average number of rekey packets (i.e., newly generated keys) and the average number of encrypted keys (measured when encryption-based KDT is used) when compared to S-LKH. On the other hand, a B^+-LKH has a smaller height and introduces a decrease in the expected maximum rekey time. The expected maximum rekey time identifies a minimum time period that has to be elapsed between two consecutive rekeyings. Furthermore, a B^+-LKH requires much less allocated nodes. The reduction of the number of allocated nodes using B^+-LKH reaches 50% of the same degree S-LKH for a highly dynamic group.

Batch Rekeying. Individual rekeying for a single join or leave request is not a practical solution. Instead, researchers suggested periodic rekeying to be performed for a batch of requests accumulated during an elapsed period. We have extended S-LKH and B^+-LKH protocols to support batch rekeying.

We introduced a generalized rekey policy definition that has three main parameters: minimum inter-rekey period, maximum request delay, and batch size. The defined policy can be used to provide simple periodic rekeying as well as other complex rekeying conditions as configured by the application. A simplified design of the software objects

used to provide secure group key management is presented. For batch rekeying, the newly generated keys compose a sub-tree of the original LKH. We illustrated how the rekey manager constructs the rekey sub-tree in both rekeying protocols and how the rekey tree is used in constructing the rekey message sent to group members for such keys updates.

We performed experiments to compare the batch rekeying performance of S-LKH and B^+-LKH protocols. Our experiments show that, the batch rekeying performance of a rekey protocol that uses LKH of degree 4 and encryption-based KDT is better than star key management only for small batch sizes (less than 20% n). In addition, our experiments show that using B^+-LKH for large batch sizes or highly dynamic groups substantially reduces the rekey cost when compared to S-LKH. In addition, B^+-LKH performance is shown to be stable (bounded) for highly dynamic groups while S-LKH performance deteriorates as the group dynamics increase.

Distributed group rekeying. To extend the scalability and the reliability of our model, we introduced four cooperation group rekeying protocols between a group of peer rekey agents. We illustrated that the protocol with the minimal overhead is that one rekey agent, at a time, generates and distributes a new group key to all group members. Detailed LKH maintenance in the different cooperation protocols are presented. In addition, the use of an agents' LKH (denoted A-LKH) is introduced to facilitate a new GK distribution by a rekey agent to all group members. The use of A-LKH minimizes the replicated LKH size at every rekey agent as well as the number of maintained keys at a group member. Finally, two approaches for A-LKH establishment are presented. The first is the dynamic A-LKH approach that is flexible but (some) group members might be updated for a rekey agent joining or leaving the agents' group. The second is the static A-LKH approach that requires the specification of the maximum number of rekey agents before starting a group session but provides transparent agents join and leave for group members.

Recovery. Finally, we proposed a logging and recovery mechanism for the group key manager/agent and the rekey manager/agent. The logging system is secure and easy to implement. Group members participate in the recovery of their manager by sending an encrypted recovery message when requested. The group member recovery message

contains his individual material and his maintained set of keys. We proposed a key selection technique to reduce the number of keys sent in the recovery message. In addition, we discussed the recovery of a group member after a short failure time.

In conclusion, the designed software model provides group key management components that are independent of the application, the security mechanism, and the communication protocol. The proposed XORBP KDT if used with the LKH approach achieves further reduction to the group rekeying computation cost and provides a more efficient and scalable solution than the encryption-based KDT. The proposed unbalanced LKH rekey protocol (S-LKH) can be used for any LKH degree. While, the proposed balanced LKH rekey protocol (B^+-LKH) is practical for a LKH of degree greater than 3. A B^+-LKH requires much less storage than S-LKH. In addition, the use of a B^+-LKH when compared to a S-LKH substantially reduces the batch rekeying cost for large batch sizes or highly dynamic groups and exhibits a bounded performance with increased group dynamics. Moreover, the proposed rekey policy offers versatile triggering conditions for the batch rekeying process including simple periodic batch rekeying. Furthermore, distributed group rekeying enhances the scalability of the group key management framework. Finally, the group key manager and the group member's recovery mechanism add reliability to the framework.

7.2 Future Extensions

The secure group key management framework can be extended as follows:

1) Adapting the proposed LKH rekey protocols to constrained LKH key generation mechanisms such as the use of a hash function. In our work, it is always assumed LKH keys are freshly randomly generated. Such constrained key regeneration techniques are used to reduce the group rekeying cost (i.e., number of randomly generated bits, rekey message size, etc...). Unfortunately, constrained key generation could be less secure.
2) Providing a dynamic rekey policy. Such dynamic rekey policy would require investigating the possibility of having conflicting policy decisions applied to the (short) time interval between two consecutive rekey policies.

3) Investigating distributed group rekeying where more than one rekey agent is experiencing a change in its subgroup membership. In this case, performing a group rekeying is similar to performing a distributed nested transaction that requires distributed concurrency control.
4) Experimenting with batch group rekeying for real application scenarios and different group sizes. The experiments would compare the batch group rekeying performance of S-LKH and B^+-LKH rekey protocols. Group applications have two benchmark scenarios. First, one sender and large group of receivers such as video broadcasting. Second, small group of peer group members such as a conferencing application where any member can be a sender.
5) Experimenting with the distributed group rekeying protocols for real application scenarios. The experiments would compare the different protocols overhead, and compare the proposed distributed architecture with other distributed secure group management architectures such as Iolus [49].
6) Implementing the proposed group key manager recovery technique and performing experiments to study its characteristics. The experiments will compare the time and overhead required for a group key manager recovery using the proposed selective logging technique and a full logging technique. A full logging technique would allow logging the LKH keys.
7) Experimenting with group member recovery in applications exhibiting short failure time such as mobile clients.
8) Perform an analytical study of the proposed key selection technique used by a group member in the construction of his group key manager recovery message.
9) Refining the implementation of the group key manager/agent, the rekey manager, and the rekey client as designed in the proposed framework. The finished product is a set of packages that extend Java™ security and can be used by secure group communication applications. The packages design will revolve around two Java™ security design principles: implementation independence and interoperability, and independence and extensibility.
10) Integrating the proposed XORBP key distribution technique and the S-LKH and B^+-LKH rekey protocols with the work of the IETF secure multicast group.

11) Investigating other secure group communication issues such as a group policy definition and implementation for the authentication manager, and a reliable group rekeying transport protocol for implementing the rekey channel.

REFERENCES

[1] K. C. Almeroth and M. H. Ammar, "Multicast Group Behavior in the Internet's Multicast Backbone (MBone)," *IEEE Communications Magazine*, vol. 35, no. 6, pp. 224-229, June 1997.

[2] D. Balenson, D. McGrew, and A. Sherman, "Key Management for Large Dynamic Groups: One-Way Function Trees and Amortized Initialization," Internet Draft (Work in Progress), Internet Engineering Task Force, draft-balenson-groupkeymgmt-oft-00.txt, Feb. 1999.

[3] A. Ballardie, "Scalable Multicast Key Distribution," Request For Comments 1949 (Experimental), Internet Engineering Task Force, May 1996.

[4] T. Ballardie and J. Crowcroft, "Multicast-Specific Security Threats and Counter-Measures," in *Proc. of the 1995 Symposium on Networks and Distributed System Security (SNDSS'95)*, San Diego, CA, USA, Feb. 1995, pp. 2-16.

[5] M. Baugher, R. Canetti, P. Cheng, and P. Rohatgi, "MESP: A Multicast Framework for the IPsec ESP," Internet Draft (Work in Progress), Internet Engineering Task Force, draft-ietf-msec-mesp-01.txt, Mar. 2003.

[6] M. Baugher, R. Canetti, and L. Dondeti, "Group Key Management Architecture," Internet Draft (Work in Progress), Internet Engineering Task Force, draft-ietf-msec-gkmarch-01.txt, Oct. 2001.

[7] S. Berkovits, "How to Broadcast a Secret," *Advances in Cryptography: Proc. of EUROCRYPT'91, Lecture Notes in Computer Science,* Springer-Verlag, Berlin, Germany, vol. 547, pp. 535-541, Apr. 1991.

[8] B. Briscoe and I. Fairman, "NARK: Receiver-Based Multicast Non-repudiation and Key Management," in *Proc. of 1st ACM Conference on E-commerce (EC'99)*, Denver, CO, USA, Nov. 1999.

[9] R. Canetti, P-C. Cheng, D. Pendarakis, J. R. Rao, P. Rohatgi, and D. Saha, "An Architecture for Secure Internet Multicast," Internet Draft (Work in Progress), Internet Engineering Task Force, draft-irtf-smug-sec-mcast-arch-00.txt, Feb. 1999.

[10] R. Canetti, J. Garay, G. Itkis, D. Micciancio, M. Naor, and B. Pinkas, "Multicast Security: A Taxonomy and Efficient Constructions," in *Proc. of the 1999 IEEE Conference On Computer Communications (INFOCOM'99)*, New York, NY, USA, Mar. 1999, vol. 2, pp. 708-716,

[11] R. Canetti, T. Malkin, and K. Nissim, "Efficient Communication-Storage Tradeoffs for Multicast Encryption," *Advances in Cryptography: Proc. of EUROCRYPT'99, Lecture Notes in Computer Science,* Springer-Verlag, Berlin, Germany, vol. 1592, pp. 459-474, 1999.

[12] R. Canetti, P. Rohatgi, and P-C. Cheng, "Multicast Data Security Transformations: Requirements, Considerations, and Proposed Design," Internet

Draft (Work in Progress), Internet Engineering Task Force, draft-irtf-smugdata-transforms-00.txt, June 2000.

[13] I. Chang, R. Engel, D. Kandlur, D. Pendarakis, and D. Saha, "Key Management for Secure Internet Multicast using Boolean Function Minimization Techniques," in *Proc. of the 1999 IEEE Conference On Computer Communications (INFOCOM'99)*, New York, NY, USA, Mar. 1999, vol. 2.

[14] W. Chen and L. R. Dondeti, "Performance Comparison of Stateful and Stateless Group Rekeying Algorithms," in *Proc. of the 4th International Workshop on Networked Group Communication (NGC'02)*, Boston, MA, USA, Oct. 2002.

[15] G. Chiou and W. Chen, "Secure Broadcasting Using the Secure Lock," *IEEE Transactions on Software Engineering*, vol. 15, no. 8, pp. 929-934, Aug. 1989.

[16] B. Coan, V. Kaul, S. Narain, and W. Stephens, "HASM: Hierarchical Application-Level Secure Multicast," Internet Draft (Work in Progress), Internet Research Task Force, draft-coan-hasm-00.txt, Nov. 2001.

[17] G. Colouris, J. Dollimore, and T. Kindberg, Distributed Systems: Concepts and Design, 2nd Edition, New York, NY: Addison-Wesley, 1994.

[18] S. E. Deering, "Host Extensions for IP Multicasting," Request For Comments 1112 (Proposed Standard), Internet Engineering Task Force, Aug. 1989.

[19] P. T. Dinsmore, D. M. Balenson, M. Heyman, P. S. Kruus, C. D. Scace, and A. T. Sherman, "Policy-Based Security Management for Large Dynamic Groups: An Overview of the DCCM Project," in *Proc. of the 2000 DARPA Information Survivability Conference and Exposition (DISCEX'00)*, Hilton Head, SC, USA, Jan. 2000, pp. 64–73.

[20] L. R. Dondeti, S. Mukherjee, and A. Samal, "Survey and Comparison of Secure Group Communication Protocols," Technical Report, University of Nebraska-Lincoln, Lincoln, NE, USA, June 1999.

[21] _____, "DISEC: A Distributed Framework for Scalable Secure Many-to-many Communication," in *Proc. of the 5th IEEE International Symposium on Computers and Communications (ISCC'00)*, Antibes, France, July 2000.

[22] A. Fiat and M. Naor, "Broadcast Encryption," *Advances in Cryptography: Proc. of CRYPTO'93, Lecture Notes in Computer Science,* Springer-Verlag, Berlin, Germany, vol. 773, pp. 480-491, Aug. 1993.

[23] R. Gennaro and P. Rohatgi, "How to Sign Digital Streams," *Advances in Cryptography: Proc. of CRYPTO'97, Lecture Notes in Computer Science,* Springer-Verlag, Berlin, Germany, vol. 1294, pp. 180-197, Aug. 1997.

[24] S. Ghanem and H. Abdel-Wahab, "A Simple XOR-based Technique for Distributing Group Key in Secure Multicasting," in *Proc. of the 5th IEEE International Symposium on Computers and Communications (ISCC'00)*, Antibes, France, July 2000, pp. 398-403.

[25] _____, "A Secure Group Key Management Framework: Design and Rekey Issues," in *Proc. of the 8th IEEE International Symposium on Computers and*

Communications (ISCC'03), Kemer-Antalya, Turkey, June-July 2003, pp. 797-802.

[26] L. Gong, "New Protocols for Third-Party-Based Authentication and Secure Broadcast," in *Proc. of* the 2nd *ACM Conference on Computer and Communications Security*, Fairfax, VA, USA, Nov. 1994, pp.176-183.

[27] L. Gong and N. Shacham, "Elements of Trusted Multicasting," in *Proc. of the 1994 IEEE International Conference on Network Protocols (ICNP'94)*, Boston, MA, USA, Oct.1994, pp. 23–30.

[28] T. Hardjono, B. Cain, and N. Doraswamy, "A Framework for Group Key Management for Multicast Security," Internet Draft (Work in Progress), Internet Engineering Task Force, draft-ietf-ipsec-gkmframework-03.txt, Aug. 2000.

[29] T. Hardjono, R. Canetti, M. Baugher, and P. Dinsmore, "Secure IP multicast: Problem Areas, Framework, and Building Blocks," Internet Draft (Work in Progress), Internet Engineering Task Force, draft-irtf-smug-framework-00.txt, Dec. 1999.

[30] T. Hardjono, H. Harney, P. McDaniel, A. Colegrove, and P. Dinsmore, "Group Security Policy Token," Internet Draft (Work in Progress), Internet Engineering Task Force, draft-ietf-msec-gspt-01.txt, Nov. 2001.

[31] T. Hardjono and G. Tsudik, "IP Multicast Security: Issues and Directions," Technical Report, Annales de Telecom, Paris, France, July-Aug. 2000, pp. 324-334.

[32] D. Harkins and D. Carrel, "The Internet Key Exchange (IKE)," Request For Comments 2409 (Proposed Standard), Internet Engineering Task Force, Nov. 1998.

[33] H. Harney, M. Baugher, and T. Hardjono, "GKM Building Block: Group Security Association (GSA) Definition," Internet Draft (Work in Progress), Internet Engineering Task Force, draft-irtf-smug-gkmbb-gsadef-00.txt, Feb. 2000.

[34] H. Harney, A. Colegrove, E. Harder, U. Meth, and R. Fleischer, "Group Secure Association Key Management Protocol," Internet Draft (Work in Progress), draft-harney-sparta-gsakmp-sec-02.txt, Internet Engineering Task Force, June 2000.

[35] H. Harney and C. Muckenhirn, "Group Key Management Protocol (GKMP) Specification," Request For Comments 2093 (Experimental), Internet Engineering Task Force, July 1997.

[36] _____, "Group Key Management Protocol (GKMP) Architecture," Request For Comments 2094 (Experimental), Internet Engineering Task Force, July 1997.

[37] I. Ingemarsson, D. T. Tang, and C. K. Wong, "A Conference Key Distribution System," *IEEE Transactions on Information Theory*, vol. 28, no. 5, pp. 714-720, Sep. 1982.

[38] J. Jannink, "Implementing Deletion in B^+-Trees," *ACM Special Interest Group on Management of Data (SIGMOD) Record,* vol. 24, no. 1, pp. 33-38, May 1995.

[39] P. Judge and M. Ammar, "Gothic: A Group Access Control Architecture for Secure Multicast and Anycast," in *Proc. of the 2002 IEEE Conference On Computer Communications (INFOCOM'02)*, New York, NY, USA, June 2002.

[40] C. Kaufman, R. Perlman, and M. Speciner, Network Security: Private Communication in a Public World, Englewood Cliffs, NJ: Prentice Hall, 1995.

[41] S. Kent and R. Atkinson, "Security Architecture for the Internet Protocol," Request For Comments 2401 (Proposed Standard), Internet Engineering Task Force, Nov. 1998.

[42] _____, "IP Authentication Header," Request For Comments 2402 (Proposed Standard), Internet Engineering Task Force, Nov. 1998.

[43] _____, "IP Encapsulating Security Payload (ESP)," Request For Comments 2406 (Proposed Standard), Internet Engineering Task Force, Nov. 1998.

[44] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith, "The SecureRing Protocols for securing group communication," in *Proc. of the 31st IEEE Hawaii International Conference on System Sciences*, Kona, Hawaii, Jan.1998, pp. 317–326.

[45] X. Li, Y. Yang, M. Gouda, and S. Lam, "Batch Rekeying for Secure Group Communications," in *Proc. of the 10th World Wide Web Conference (WWW10),* Hong Kong, China, May 2001, vol. 3, pp. 525-534.

[46] J. Lotspiech, M. Naor, and D. Naor, "Subset-Difference based Key Management for Secure Multicast," Internet Draft (Work in Progress), Internet Engineering Task Force, draft-irtf-smug-subsetdifference-00.txt, July 2001.

[47] D. Maughan, M. Schertler, M. Schneider, and J. Turner, "Internet Security Association and Key Management Protocol (ISAKMP)," Request For Comments 2408 (Proposed Standard), Internet Engineering Task Force, Nov. 1998.

[48] P. McDaniel, A. Prakash, and P. Honeyman, "Antigone: A Flexible Framework for Secure Group Communication," in *Proc. of the 8th USENIX Security Symposium*, Washington, D.C., USA, Aug. 1999, pp 99–114.

[49] S. Mittra, "Iolus: A Framework for Scalable Secure Multicasting," in *Proc. of the 1997 ACM SIGCOMM,* Cannes, France, Sep. 1997, vol. 27, no. 4, pp. 277-288.

[50] A. V. Moffaert and O. Paridaens, "Security Issues in Internet Group Management Protocol version 3 (IGMPv3)," Internet Draft (Work in Progress), Internet Engineering Task Force, draft-irtf-gsec-igmpv3-security-issues-00.txt, Dec. 2001.

[51] M. J. Moyer, J. R. Rao, P. Rohatgi, "Maintaining Balanced Key Trees for Secure Multicast," Internet Draft (Work in Progress), Internet Engineering Task Force, draft-irtf-smug-key-tree-balance-00.txt, June 1999.

[52] J. Pegueroles and F. Rico-Novella, "Balanced Batch LKH: New Proposal, Implementation and Performance Evaluation," in *Proc. of the 8th IEEE International Symposium on Computers and Communications (ISCC'03)*, Kemer-Antalya, Turkey, June-July 2003.

[53] A. Perrig, R. Canetti, B. Briscoe, J. D. Tygar, and D. Song, "TESLA: Multicast Source Authentication Transform," Internet Draft (Work in Progress), Internet Engineering Task Force, draft-irtf-smug-tesla-00.txt, Nov. 2000.

[54] A. Perrig, R. Canetti, D. Song, and J. D. Tygar, "Efficient and Secure Source Authentication for Multicast," in *Proc. of the 2001 Network and Distributed System Security Symposium (NDSS'01)*, San Diego, CA, USA, Feb. 2001, pp. 35–46.

[55] S. Rafaeli, L. Mathy, D. Hutchison, "LKH+2: An Improvement on the LKH+ Algorithm for Removal Operations," Internet Draft (Work in Progress), Interent Engineering Task Force, draft-rafaeli-lkh2-oo.txt, Jan. 2002.

[56] O. Rodeh, K. Birman, and D. Dolev, "Optimized Group Rekey for Group Communication Systems," Technical Report TR99-1764, Department of Computer Science, Cornell University, USA, Aug. 1999.

[57] O. Rodeh, K. Birma, M. Hayden, Z. Xiao, and D. Dolev, "Ensemble Security," Technical Report TR98-1703, Department of Computer Science, Cornell University, USA, Sep. 1998.

[58] A. Selck, C. McCubbin, and D. Sidhu, "Probabilistic Optimization of LKH-based Multicast Key Distribution Schemes," Internet Draft (Work in Progress), Internet Engineering Task Force, draft-selcuk-probabilistic-lkh-00.txt, Jan. 2000.

[59] S. Setia, S. Koussih, S. Jajodia, E. Harder, "Kronos: A Scalable Group Re-Keying Approach for Secure Multicast," in *Proc. of the 2000 IEEE Symposium on Security and Privacy*, Berkeley, CA, USA, May 2000, pp. 215-228.

[60] S. Setia, S. Zhu, and S. Jajodia, "A Comparative Performance Analysis of Reliable Group Rekey Transport Protocols for Secure Multicast," in *Proc. of the 2002 Performance,* Rome, Italy, Sep. 2002, pp.21-41.

[61] M. Steiner, G. Tsudik, and M. Waidner, "CLIQUES: A New Approach to Group Key Agreement," in *Proc. of the 18th IEEE International Conference on Distributed Computing Systems (ICDCS'98)*, Amsterdam, The Netherlands, May 1998, pp. 380-387.

[62] Sun Microsystems, Inc., Java Security Documentation [Online]. Available http://java.sun.com/security/

[63] A. M. Tenenbaum and M. J. Augenstein, Data Structures Using Pascal, 2nd Edition, Englewood Cliffs, NJ: Prentice-Hall, 1986.

[64] 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. 8, pp. 1614-1631, Aug. 1999.

[65] D. Wallner, E. Harder, and R. Agee, "Key Management for Multicast: Issues and Architecture," Request for Comments 2627 (Informational), Internet Engineering Task Force, June 1999.

[66] B. Whetten, T. Montgomery, and S. Kaplan, "A High Performance Totally Ordered Multicast Protocol," *Theory and Practice in Distributed Systems:*

International Workshop, Lecture Notes in Computer Science, Springer-Verlag, Berlin, Germany, no. 938, pp. 33–57, Apr. 1995.

[67] C. K. Wong, M. Gouda, and S. S. Lam, "Secure Group Communications using Key Graphs," in *Proc. of ACM SIGCOMM'98,* Vancouver, Canada, Sep. 1998, pp. 68-79.

[68] C. K. Wong and S. S. Lam, "Digital Signatures for Flows and Multicasts," Technical Report TR-98-15, Department of Computer Sciences, University of Texas at Austin, TX, USA, July 1998.

[69] Y. R. Yang, X. S. Li, X. R. Zhang, and S. S. Lam, "Reliable Group Rekeying: A Performance Analysis," in *Proc. of ACM SIGCOMM'01*, San Diego, CA, USA, Aug. 2001, pp. 27-38.

[70] X. B. Zhang, S. S. Lam, D. Y. Lee, and Y. R. Yang, "Protocol Design for Scalable and Reliable Group Rekeying," in *Proc. of SPIE conference on Scalability and Traffic Control in IP Networks*, Denver, CO, USA, Aug. 2001, vol. 4526.

APPENDIX A

EXAMPLES OF S-LKH AND B$^+$-LKH REKEY PROTOCOLS

This appendix contains two examples for S-LKH and B$^+$-LKH rekey protocols. The examples illustrate the rekey message sent in different group member addition and removal scenarios. The initial key message (*initMsg*) format is {ID, position, height, degree}. The rekey message (*rekeyMsg*) format for S-LKH rekey protocol is {SEQ, type, position, level, ID, RekeyPacket$_1$, RekeyPacket$_2$, ...}. The rekey message format (*rekeyMsg*) for B$^+$-LKH rekey protocol is {SEQ, type, position, level, (ID$_1$, ID$_2$, ...), (isRight$_1$, isRight$_2$, ...), RekeyPacket$_1$, RekeyPacket$_2$, ...} where *isRight* values are T for true and F for false. Note that maintaining SEQ is not shown in the algorithms (trivial). Note also that, the rekey message *level* filed is not assigned in all cases.

A LKH key is identified by its LKH position and that position is changing due to insertion/deletion of node entries. An *addRekey* packet is identified by a key and the directly/indirectly inserted entry number in the associated child node. If encryption-based KDT is used, such *addRekey* packet contains the new version of the key encrypted by its previous version and by that specified child node key entry. For example *addRekey*($K_{2.1}$, 2) packet is $[\{K'_{2.1}\}K_{2.1}, \{K'_{2.1}\}K_{2.1.2}]$. A *rmvRekey* packet is identified by a key. If encryption-based KDT is used, such *rmvRekey* packet contains the new key encrypted by every key in the associated child nod. For example *rmvRekey*($K_{2.1}$) packet is $[\{K'_{2.1}\}K_i, K_i \in node(P_{2.1})]$, where $node(P_{2.1})$ is the node pointed to by the pointer $P_{2.1}$.

S-LKH Examples

Fig. 68(a) illustrates the initial nodes of a S-LKH of degree 4 that is used to demonstrate the three member addition and the two member removal scenarios. The S-LKH is constructed using S-LKH AddMember (Fig. 24) and RemoveMember (Fig. 25) algorithms. The S-LKH height $h \geq 3$ (part of the tree is not expanded). For all other figures the S-LKH search view is used to illustrate the changes to the initial S-LKH.

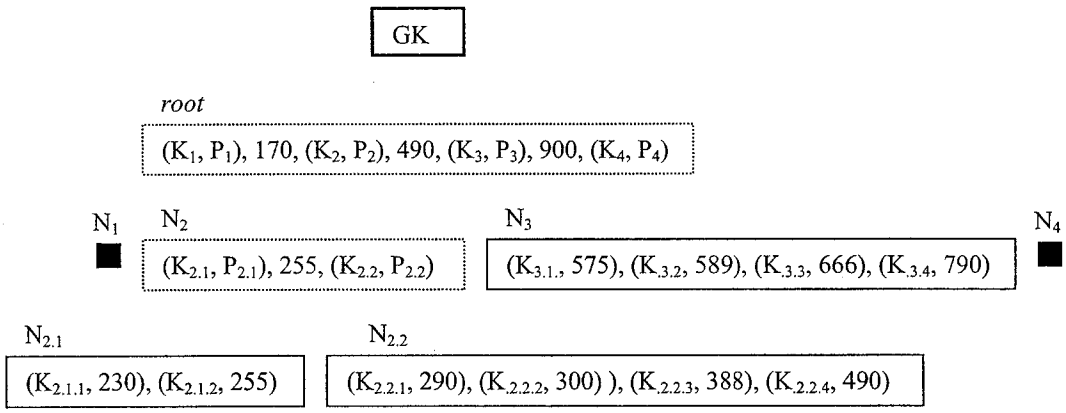
Fig. 68(b) is the S-LKH search view after AddMember(240, K_X) is performed. The returned *initMsg* is {240, 2.1.2, *h*, 4} and the returned *rekeyMsg* is {SEQ, ADD, 2.1.2, 2, 240, *addRekey*($K_{2.1}$, 2), *addRekey*(K_2, 1), *addRekey*(*GK*, 2)}.

Fig. 68(c) is the S-LKH search view after AddMember(420, K_Y) is performed. The returned *initMsg* is {420, 2.2.4, *h*, 4} and the returned *rekeyMsg* is {SEQ, SPLIT, 2.2.4, 2, 420, *rmvRekey*($K_{2.2}$), *rmvRekey*($K_{2.3}$), *addRekey*(K_2, 3), *addRekey*(*GK*, 2)}.

Fig. 68(d) is the S-LKH search view after AddMember(609, K_Z) is performed. The returned *iniMsg* is {609, 3.3, *h*, 4} and the returned *rekeyMsg* is {SEQ, INCREASE, 3.3, 1, 609, *rmvRekey*($K_{3.1}$), *rmvRekey*($K_{3.2}$), *addRekey*(K_3, 1), *addRekey*(*GK*, 3)}.

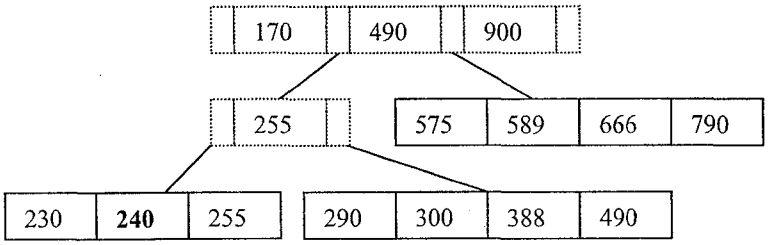
Fig. 68(e) is the S-LKH search view after RemoveMember(666) is performed. The returned *rekeyMsg* is {SEQ, REMOVE, 3.2.1, 2, 666, *rmvRekey*($K_{3.2}$), *rmvRekey*(K_3), *rmvRekey*(*GK*)}.

Fig. 68(f) The following figure is the S-LKH after RemoveMember(790) is performed. The returned *rekeyMsg* is {SEQ, DECREASE, 3.2.1, 1, 666, *rmvRekey*(K_3), *rmvRekey*(*GK*)}.

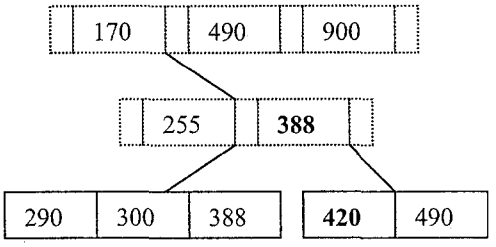


(a) The S-LKH initial nodes.

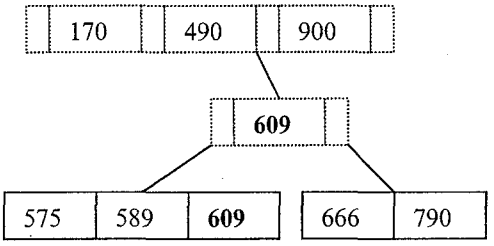
Fig. 68. A S-LKH member addition and removal examples.



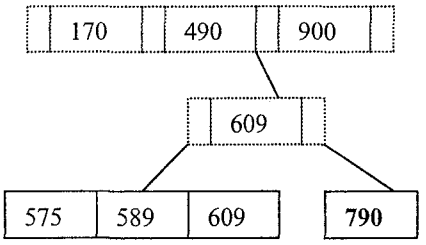
(b) The S-LKH search view after AddMember(240, K_X) is performed.



(c) The S-LKH search view after AddMember(420, K_Y) is performed.

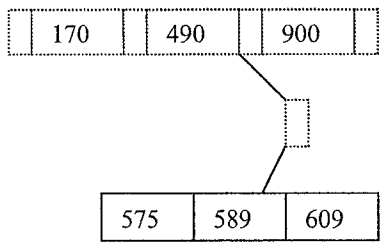


(d) The S-LKH search view after AddMember(609, K_Z) is performed.



(e) The S-LKH search view after RemoveMember(666) is performed.

Fig. 68. (Continued)



(f) The S-LKH search view after RemoveMember(790) is performed.

Fig. 68. (Continued)

B$^+$-LKH Example

Fig. 69(a) illustrates a B$^+$-LKH of degree $d = 4$ and height $h = 3$ constructed using the B$^+$-LKH AddMember (Fig. 29) and RemoveMember (Fig. 34) algorithms. Note that, parts of the tree are not expanded but the maintenance algorithms guarantees that all nodes are at the same level, so the height h of that B$^+$-LKH is 3. The B$^+$-LKH is used in demonstrating the different B$^+$-LKH member addition and removal scenarios.

Fig. 69(b) is the B$^+$-LKH search view after performing AddMember(600, K_X) followed by AddMembr (790, K_Y). The first returned *initMsg* is {600, 3.1.2, 3, 4} and the first returned *rekeyMsg* is {SEQ, ADD, 3.1.2, -, (600), -, *addRekey*($K_{3.1}$, 2), *addRekey*(K_3, 1), *addRekey*(*GK*, 3)}. Then the second returned *initMsg* is {790, 3.2.3, 3, 4} and the second returned *rekeyMsg* is {SEQ, ADD, 3.2.3, -, (790), -, *addRekey*($K_{3.2}$, 3), *addRekey*(K_3, 2), *addRekey*(*GK*, 3)}.

Fig. 69(c) is the B$^+$-LKH search view after AddMember(770, K_Z) is performed. The returned *intiMsg* is {770, 3.2.2, 3, 4} and the returned *rekeyMsg* is {SEQ, SPLIT, 3.2.2, 1, (770, 786), -, *rmvRekey*($K_{3.2}$), *rmvRekey*($K_{3.3}$), *addRekey*(K_3, 2), *addRekey*(*GK*, 3)}.

Fig. 69(d) is the B$^+$-LKH search view after AddMember(590, K_W) is performed. The returned *initMsg* is {590, 3.1.2, 3, 4} and the returned *rekeyMsg* is {SEQ, INCREASE, 3.1.2, -, (590, 600, 786, 786), -, *rmvRekey*($K_{1.3.1}$), *rmvRekey*($K_{1.3.2}$), *rmvRekey*($K_{1.3}$),

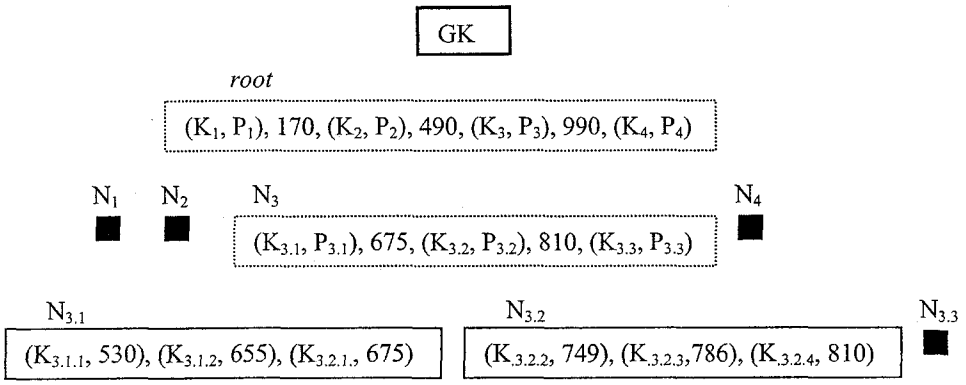
rmvRekey($K_{.2.1}$), *rmvRekey*(K_1), *rmvRekey*($K_{.2}$), *addRekey*(*GK*, 1)}. The B^+-LKH height *h* becomes 4.

Fig. 69(e) is the B^+-LKH search view after RemoveMember(990) is performed (an expansion of extra part of the tree is shown). The returned *rekeyMsg* is {SEQ, REMOVE, 2.1.2.3, -, (990), -, *rmvRekey*($K_{2.1.2}$), *rmvRekey*($K_{2.1}$), *rmvRekey*(K_2), *rmvRekey*(*GK*)}.

Fig. 69(f) is the B^+-LKH search view after RemoveMember(817) is performed. The returned *rekeyMsg* is {SEQ, SHIFT, 2.1.2.1, 0, (817, 810, 990), (F, T, F), *mrgRekey*($K_{2.2.1}$, F), *mrgRekey*($K_{2.2}$, T), *rmvRekey*(K_2), *rmvRekey*(K_1), *rmvRekey*(*GK*)}.

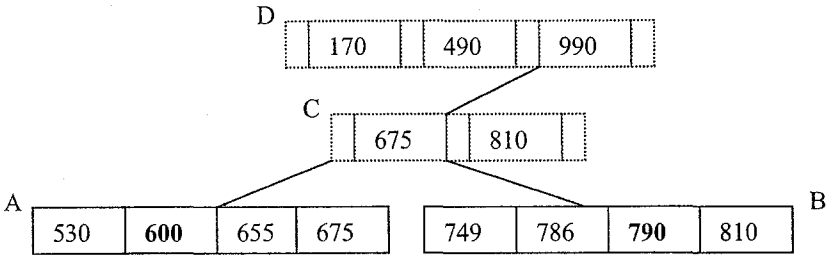
Fig. 69(g) illustrates an expansion of P_1 sub-tree. Fig. 69(h) is the B^+-LKH search view after RemoveMember(380) is performed. The returned *rekeyMsg* is {SEQ, MERGE, 1.2.3.1, 2, (380, 230), (F), *mrgRekey*($K_{1.2.2}$, F), *rmvekey*($K_{1.2}$), *rmvRekey*(K_1), *rmvRekey*(*GK*)}.

Fig. 69(i) is is the B^+-LKH search view after RemoveMember(100) is called. The returned *rekeyMsg* is {SEQ, DECREASE, 1.1.1.2, -, (100, 100, 170, 490), (T, T, T), *mrgRekey*($K_{1.1}$, T), *mrgRekey*(K_1, T), *mrgRekey*(*GK*, T)}.

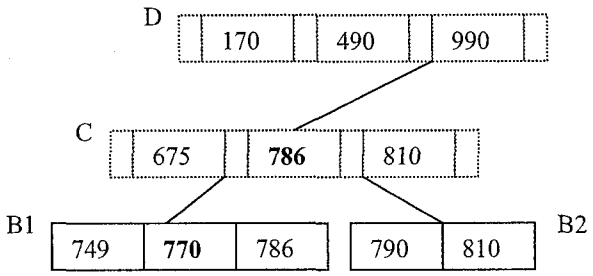


(a) The B^+-LKH initial nodes.

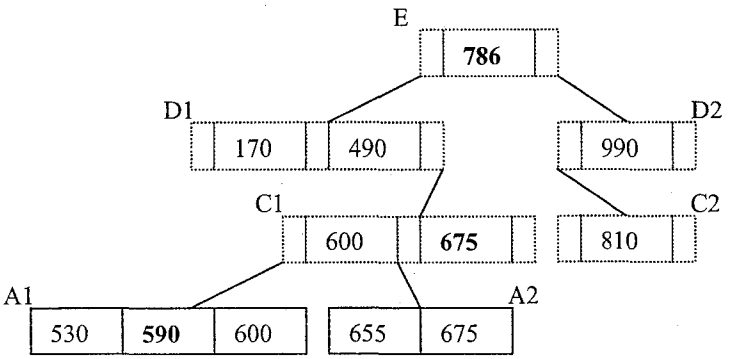
Fig. 69. A B^+-LKH member addition and removal examples.



(b) The B^+-LKH search view after AddMember(600, K_X) and (790, K_Y) are performed.

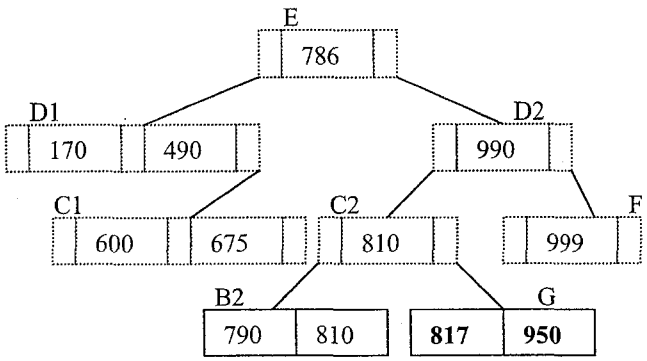


(c) The B^+-LKH search view after AddMember(770, K_Z) is performed.

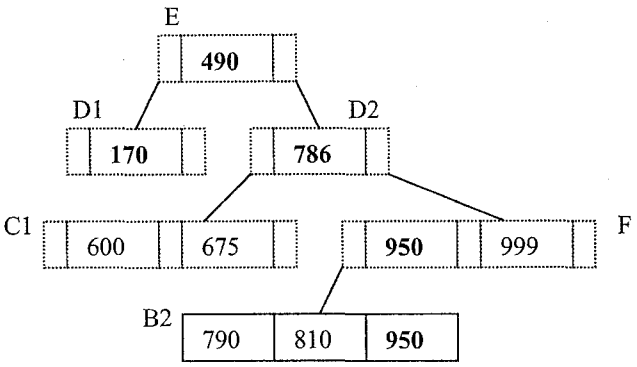


(d) The B^+-LKH search view after AddMember(590, K_W) is performed.

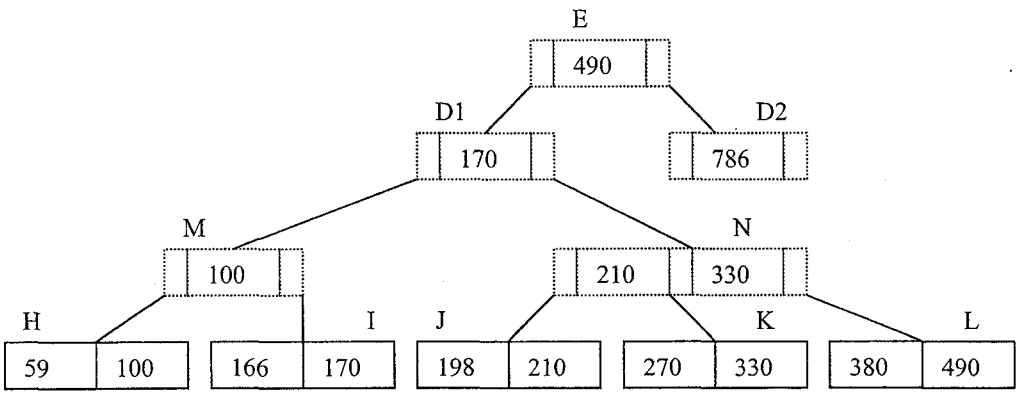
Fig. 69. (Continued)



(e) The B^+-LKH search view after RemoveMember(990) is performed.

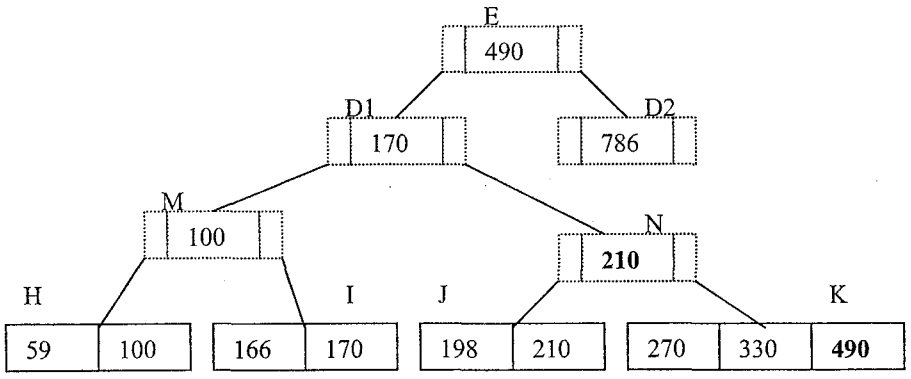


(f) The B^+-LKH search view after RemoveMember(817) is performed.

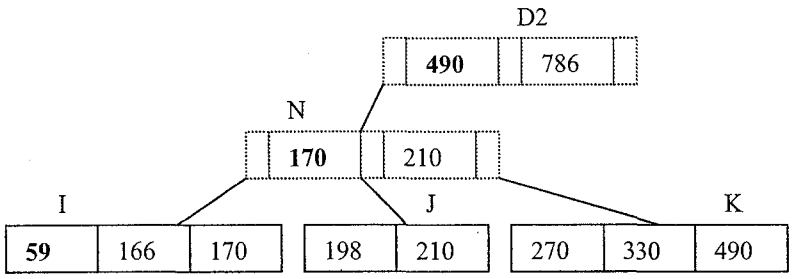


(g) The B^+-LKH expansion of P_1 sub-tree.

Fig. 69. (Continued)



(h) The B^+-LKH search view after RemoveMember(380) is performed.



(i) The B^+-LKH search view after RemoveMember(100) is performed.

Fig. 69. (Continued)

APPENDIX B

B$^+$-LKH REKEY CLIENT PROCESSING

```
Method Rekey(rekeyMsg)
Globals: h, d, Min_d, ID, position, keyList, KDT,
         rekeyPos, level, match, isRight, isRNghbr, isLNghbr, S;

{ rekeyPos = rekeyMsg.position; level = rekeyMsg.level; match = -1;
 for (I = 0 to (h-1)) if (position[I] equals rekeyPos[I]) then match = I; else breakFor;
 match = h – match; if (match equals 1) then match = 2;
 X = h + 1-match; isRght = rekeyMsg.isRght[match-3];
 isRNghbr = isRght and (position[X] equals (rekeyPos[X]+1));
 isLNgbr = (not isRght) and (position[X] equals (rekeyPos[X]-1));
 if (match < (h-level+2)) then S = h-level; else S = match-1;
 IF (rekeyMsg.type)
 { equals ADD or REMOVE: Simple();
  equals SPLIT: Split();equals INCREASE: Increase();
  equals MERGE: Merge(); equals SHIFT: Shift(); equals DECREASE: Decrease();}
}/***/
Method Loop1(startI, endI, adjust)
{ for (I = startI to endI) keyList.update(I + adjust, rekeyMsg.packet[I]);
}/***/
Method Loop2(startI, endI, adjust)
{ for (I = startI to endI)
   { if (ID > rekeyMsg.ID[I]) then increment position[h-1-I];
     packetNo = 2* I;
     if (position[h-1-I] > Min_d)
        then { increment position[h-1-I] by (Min_d+1); packetNo = packetNo+1; }
     keyList.update(I + adjust, rekeyMsg.packet[packetNo]); }
}/***/
Method Loop3(startI, endI, adjust)
{ for (I = startI to endI)
    { if (ID > rekeyMsg.ID[I]) then decrement position[h-1-I];
      if (KDT equals XORBP)
         then keyList.updateBP(I, rekeyMsg.xoredBP[I, position[h -1-I]]);
      if (not rekeyMsg.isRght[I]) then increment position[h-1-I] by Min_d;
     keyList.update(I + adjust, rekeyMsg.packet[I]); }
}
```

Fig. 70. The B$^+$-LKH rekey client Rekey(), Loop1(), Loop2(), and Loop3() methods.

```
Method Simple()
{ if ((match equals 2) and (ID > rekeyMsg.ID[0]))
     then if (rekeyMsg.type equals ADD) then increment position[h-1];
               else decrement position[h-1];
  Loop1(match-2, h-1, 1);
}/***/
Method Split()
{ Y = h-level-2;
  Loop2(match-2, Y, 1);
  if (((match-2) < (h-level)) and (ID >rekeyMsg.ID[Y+1]))
          then increment position[level];
  Loop1(Y+S, Y+h, -Y);
}/***/
Method Increase()
{ increment h; Loop2(match-2, h-1, 1);
}/***/
Method Decrease()
{ if ( match ≥ 3)
    then { if (isRNghbr) then increment position[X+1] by (Min_d –1);
            if (isRNgbr or isLNghbr) then Loop1(match-3, match-3, 1); }
  Loop3(match-2, h-2, 1);
  decrement h ; free position[0]; keyList.free(h+1);
}
```

Fig. 71. The B^+-LKH rekey client Simple(), Split(), Increase(), and Decrease() methods.

```
Method Merge()
{ if ( 3 ≤ match < (h-level+2))
   then { if (isRNghbr ) then increment position[X+1] by (Min_d-1);
          if (isRNghbr or isLNghbr ) then Loop1(match-3, match-3, 1); }
  Loop3(match-2, h-L-2, 1);
  if (((match-2) < (h-level)) and (ID >rekeyMsg.ID[h-level-1]))
       then decrement position[level];
  Loop1(S-1, h-1, 1);
}/***/
Method Shift()
{ if ( 3 ≤ match < (h-level+1))
    then { if (isRNghbr) then increment position[X+1] by (Min_d-1);
           if (isRNghbr or isLNghbr) then Loop1(match-3, match-3, 1); }
  if (match equals (h-level+1))
   then if ( isRNghbr or isLNghbr)
     then { if (isRght and (position[level] equals (rekeyPos[level]+1)))
              then decrement position[level+1];
            if ((isRght and (position[level+1]<1)) or
               (not isRght and (ID > rekepMsg.ID[match-2])))
            then { if (isRght)
                     then { decrement position[level];
                            increment position[level+1] by Min_d; }
                     else { increment position[level]; position[level+1]=1; }
                   if (KDT equals XORBP)
                     then keyList.updateBP(match-3, rekeyMsg.xoredBP(match-3, 1));
                   Loop1(match-3, match-3, 1);
                 } else Loop1(match-2, match-2, 0);
          }
  Loop3(match-2, h-level-3, 1);
  if (match < (h-level-1))
    then { if (rekeyMsg.isRght[h-level-2] and (ID > rekeyMsg.ID[h-level-2]))
             then decrement position[level+1];
           Loop1(h-level-2, h-level-2, 1); }
  Loop1(S, h, 0)
}
```

Fig. 72. The B^+-LKH rekey client Merge(), and Shift() methods.

Example

This example illustrates B^+-LKH rekey client processing for the returned *rekeyMsg* {SEQ, DECREASE, 1.1.1.2, -, (100, 100, 170, 490), (T, T, T), *mrgRekey*($K_{1.1}$, T), *mrgRekey*(K_1, T), *mrgRekey*(GK, T)} in the last step in the B^+-LKH example in appendix A. Initially, all rekey clients maintains h = 4, and when *rekeyMsg* is received they will execute the Decrease() method. All rekey clients will adjust h to be equal to 3 and *keyList* size will be 4 after the method is executed.

Note that *keyList.update*(*key_number, rekeyMsg.packet*[*packet_number*]) will be shortened to *KLU*(*key_number, packe_ number*). We will trace the rekey client *position* and the updated keys for four members with *match* = 2, 3, 4, and 5.

The group member whose ID = 50 has *match* = 2 and *position* = 1.1.1.1. The rekey client executes Loop3(0, 2, 1) {(KLU(1, 0); KLU(2, 1), KLU(3, 2)} and *position* becomes 1.1.1.

The group member whose ID = 166 has *match* = 3 and *position* = 1.1.2.1. The rekey client executes the condition with X = 2 and *isRight* = T {*position* =1.1.2.2, KLU(1, 0)} and then executes Loop3(1, 2, 1){*position* =1.1.1.2, KLU(2, 1); KLU(3, 2)} and *position* becomes 1.1.2.

The group member whose ID = 198 has *match* = 4 and *position* =1.2.1.1. The rekey client executes the condition with X = 1 and *isRight* =T {*position* =1.2.2.1, KLU(2, 1)}, then executes Loop3(2, 2, 1) {*position* =1.1.2.1, KUL(3, 1)}, finally *position* becomes 1.2.1.

The group member whose ID = 530 has *match* = 5 and *position* = 2.1.1.1. The rekey client executes the condition with X = 0 and *isRght* =T {*position* = 2.2.1.1, KLU(3, 2)} and *position* becomes 2.1.1.

APPENDIX C

B^+-LKH REKEY SUB-TREE LABELED INSERTION

This appendix details the B^+-LKH rekey sub-tree (*rekeyTree*) labeled insertion of key nodes. A key node is inserted into *rekeyTree* in one of four ways namely, *insert*, *insertSplit*, *insertMerge*, and *insertShift*. The simple insertion *insert*(N, RC, *type*) inserts the key node N labeled according to the policy determined rekey condition RC and the rekey message *type* that is either ADD or REMOVE as shown in TABLE IX. Another form of simple insertion is *insert*(N, label) that inserts the key node N with the specified label, and *insert*(N) that inserts the key node N with no label.

TABLE IX

LABEL OF KEY NODE N FOR SIMPLE RM TYPES: ADD & REMOVE

RC\type	ADD	REMOVE
NONE	"A"	-
PBS	"GA"	-
PFS	"A"	"GR"
PBaFS	"GA"	"GR"

The *insertSplit*(N, RC) of a key node N inserts two key nodes N1 and N2 to the *rekeyTree* for two nodes that result of node N spliting. Every internal key, *GK* or KEK, has an LKH internal entry that contains a pointer to its child node, where the child node for *GK* is *root* node. A split key node N means the child node pointed to by that key internal entry is split. Let N be the node specified to be split to two nodes N1 and N2, where N1 is chosen from the two nodes such that it contains the newly inserted entry and N2 is its neighbor that share entries previously inserted in N. Initially, the key for both

nodes N1 and N2 entries will contain the key that was in N and at least N1 key will be regenerated. The label of the two key nodes when inserted in the *rekeyTree* is specified according to RC as shown in TABLE X. Note that if N exists in the *rekeyTree*, N1 and N2 both will start with N label that could be upgraded. Also note that if RC is NONE, N1 will be labeled "GA" to guarantee the generation (creation) of that key (which initially contained the same key as N2) although "A" would be suitable otherwise.

TABLE X

LABELS OF KEY NODES N1 AND N2 FOR A SPLIT KEY NODE

RC	N1	N2
NONE	"GA"	-
PBS	"GA"	-
PFS	"GR"	"GR"
PBaFS	"GR"	"GR"

The *insertMerge*(N, *isRight*, RC) of key node N inserts the key node N1 in the r*ekeyTree* that is merged with N. The key node N1 is determined from *isRight* value (right or left neighbor). If N already exists in the *rekeyTree*, it is deleted first then N1 is inserted. Inserting N1 implies inserting all merged children entries with no label or with their label if any existed in the *rekeyTree*. The key node N1 will be labeled according to RC as shown in TABLE XI.

TABLE XI

LABEL OF MERGED KEY NODE N TO N1

RC	N1
NONE	"A"
PBS	"GA"
PFS	"A"
PBaFS	"GA"

The *insertShift*(N, *isRight*, RC) of key node N inserts two key nodes N and N1 in the *rekeyTree*. The node N1 is the N neighbor determined from *isRight* value where an entry is shifted from N1 to N. Both N and N1 nodes are labeled according to RC as shown in TABLE XII.

TABLE XII

LABEL OF SHIFTED KEY NODES FROM N1 TO N

RC	N	N1
NONE	-	-
PBS	"GA"	-
PFS	"GR"	"GR"
PbaFS	"GR"	"GR"

A leaf entry *position* in a B^+-LKH is represented by an array of size LKH height h, where each array entry specifies a child position in the path that leads to the leaf entry. There are $(h + 1)$ keys specified from the *position* $p_1 p_2 ... p_h$ as follows: the group key *GK*, (h-1) KEKs K_{p_1}, $K_{p_1 p_2}$, ..., and $K_{p_1 p_2 ... p_{h-1}}$, and a leaf (individual) key $K_{p_1 p_2 ... p_h}$. Assuming the keys specified by certain *position* are the keys of *key* array of size $(h + 1)$.

The *rekeyTree* insertion of such *key* array depends on the parameters RC, *type*, *level*, and *isRight* array. Fig. 73 illustrates the insertion of the *key* array to *rekeyTree*, for all possible rekey message types and policy-determined rekey condition (RC).

```
if ((type equals ADD) or (type equals REMOVE))
 then { for (I = 1 to h) rekeyTree.insert(key [I], RC, type);
        if (type equals ADD) then rekeyTree.insert(key[h+1]); }

if (type equals SPLIT)
 then { for (I = 1 to (level +1)) rekeyTree.insert(key[I], RC, ADD);
        for (I = (level+2) to h)  rekeyTree.insertSplit(key[I], RC);
        rekeyTree.insert(key[h+1]);}

if (type equals  INCREASE)
 then { rekeyTree.insert(key[1], RC, ADD);
        rekeyTree.insert(K1, "GR");
        rekeyTree.insert(K2, "GR");
        for (I =2 to h) rekeyTree.insertSplit(key[I], RC);
        rekeyTree.insert(key[h+1]); }


if (type equals MERGE)
 then { for (I = 1 to (level+1)) rekeyTree.insert(key[I], RC, REMOVE);
        for (I = (level+2) to h)
              rekeyTree.insertMerge(key[I], isRight[h+1-I], RC); }

if (type equals SHIFT)
 then { for (I = 1 to (level+1)) rekeyTree.insert(key[I], RC, REMOVE);
        rekeyTree.insertShift(key[level+2], isRight[h-level-1], RC);
        for (I = (level+3) to h)
             rekeyTree.insertMerge(key[I], isRight[h+1-I], RC); }

if (type equals DECREASE)
 then { rekeyTree.delete(GK);
        for (I = 1 to (h –1)) rekeyTree.insertMerge(key[I], isRight[h-I], RC); }
```

Fig. 73. Labeled insertion of *key* array to a B^+-LKH rekey sub-tree.

Example

For the B^+-LKH key view shown in Fig. 74, where degree $d = 4$, height $h = 3$, and group size $n = 29$. If RC is PBaFS, and a rekeying has been initiated for *batch* of requests that contains 4 Add requests, 2 Remove requests, and 2 Refresh requests as shown in the figure. The 2 removed entries' positions are marked "Rplc" for replacement by 2 added entries, the other 2 added entries' positions are marked "Add", and the 2 refreshed entries' positions are marked "Rfrsh".

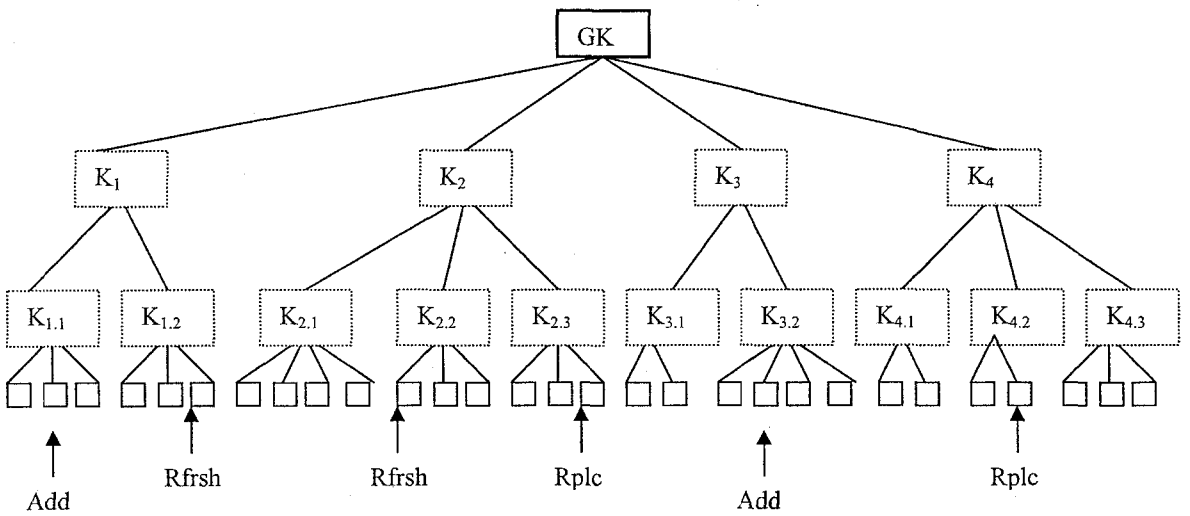


Fig. 74. A B^+-LKH key view and a batch of requests.

A LKH leaf entry position is determined by a path that starts from the root node and specifies the child node number in all nodes in the path that leads to that leaf node. The positions of the two refreshed individuals' entries are 1.2.3, and 2.2.1 (Rfrsh marked nodes). The two removed individuals' leaf entries will be replaced by two new individuals' leaf entries (i.e., a new member will be assigned the same ID of a removed member). The two replaced entries are at positions 2.3.3 and 4.2.2 (Rplc marked nodes).

The other two new individuals' leaf entries are assigned two newly generated IDs and inserted into the original LKH. From a new individual ID the position of his individual leaf entry is determined (Add marked positions). The *rekeyTree*, shown in Fig. 75, is constructed for the replaced, refreshed, and added entries as follows:

1. Replacing the leaf entry at position 2.3.3 leads to *rekeyTree* insertion of the key nodes GK, K_2, and $K_{2.3}$ labeled "GR", and the leaf key $K_{2.3.3}$ with no label.
2. Replacing the leaf entry at position 4.2.2 leads to *rekeyTree* insertion of the key nodes GK, K_4, and $K_{4.2}$ labeled "GR", and the leaf key node $K_{4.2.2}$ with no label. Note that GK is inserted before with the same label.
3. Refreshing the entry at position 1.2.3 leads to *rekeyTree* insertion of the key nodes GK, K_1, and $K_{1.2}$ labeled "A", and the leaf key node $K_{1.2.3}$ with no label. Note that GK is inserted before with higher ranked label.
4. Refreshing the entry at position 2.2.1 leads to *rekeyTree* insertion of the key nodes GK, K_2, and $K_{2.2}$ labeled "A", and the leaf key node $K_{2.2.1}$ with no label. Note that GK, and K_2 are inserted before with higher ranked labels.
5. The randomly generated ID_a for the first added individual positions his entry at 1.1.2, where RM *type* for such insertion is ADD. Inserting that leaf entry leads to *rekeyTree* insertion of the key nodes GK, K_1, and $K_{1.1}$ labeled "GA" and the leaf key node $K_{1.1.2}$ with no label. Note that GK is already inserted before with higher ranked label and K_1 is already inserted before with "A" label that is upgraded to "GA".
6. The randomly generated ID_b for the second added individual positions his entry at 3.2.2, where RM *type* of such insertion is SPLIT and *level* is 1. Inserting that leaf node leads to *rekeyTree* insertion of the key nodes GK, and K_3 labeled "GA". For the split node $K_{3.2}$ where N1 (that has the new entry) is $K_{3.2}$ and N2 is $K_{3.3}$, both N1 and N2 will be inserted labeled "GR". The leaf key node $K_{3.2.2}$ is inserted with no label.

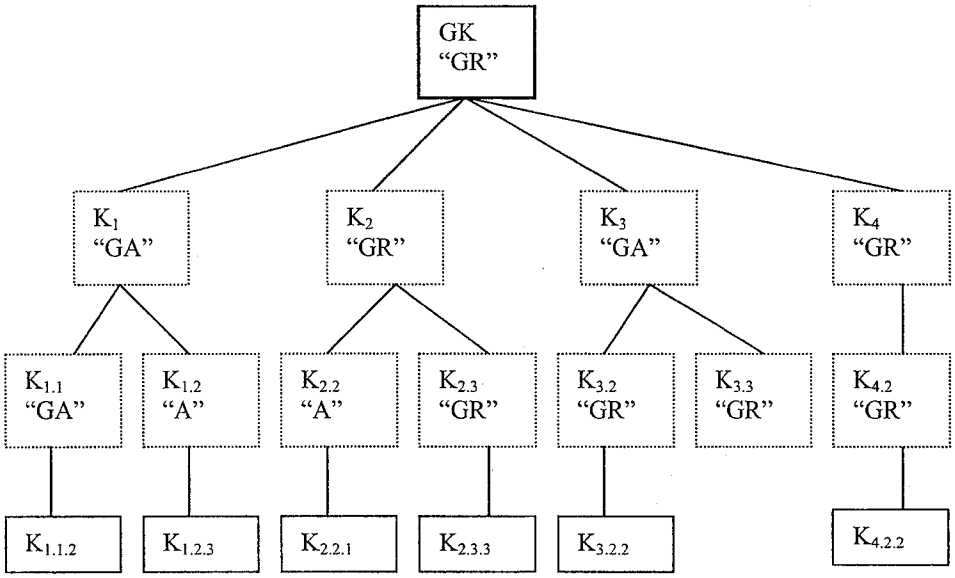


Fig. 75.The B^+-LKH rekey sub-tree constructed for batch of 8 requests.

The batch RM for such batch of requests contains:

- Two replaced positions 2.2.3, and 4.2.2
- Two refreshed positions 1.2.3, and 2.2.1
- Two individual RM headers {type = ADD, position = 1.1.2, ID_a} and {type = SPLIT, position = 3.2.2, level = 1, (ID_b, ID_c)}
- The rekey packets constructed for all labeled keys in the *rekeyTree* each according to its label. The *rekeyTree* is parsed in post-order generating the rekey packets for the keys in the following order: $K_{1.1}$, $K_{1.2}$, K_1, $K_{2.2}$, $K_{2.3}$, K_2, $K_{3.2}$, $K_{3.3}$, K_3, $K_{4.2}$, K_4, GK.

If encryption-based KDT is used, the rekey packets are as follows:

- For the two "A" labeled keys are: $[\{K_{1.2}\}K_{1.2.3}]$ and $[\{K_{2.2}\}K_{2.2.1}]$
- For the three "GA" labeled keys are: $[\{K'_{1.1}\}K_{1.1}, \{K'_{1.1}\}K_{1.1.2}]$, $[\{K'_1\}K_1, \{K'_1\}K_{1.1}, \{K'_1\}K_{1.2}]$; and $[\{K'_3\}K_3, \{K'_3\}K_{3.2}, \{K'_3\}K_{3.3}]$

- For the "GR" labeled key $K_{2.3}$ is $[\{K'_{2.3}\}K_{2.3.1}, \{K'_{2.3}\}K_{2.3.2}, \{K'_{2.3}\}K_{2.3.3}]$. All other "GR" labeled key (K_2, $K_{3.2}$, $K_{3.3}$, $K_{4.2}$, K_4, GK) are constructed the same way: a new key version is generated and encrypted with all its children keys (at the original LKH).

APPENDIX D

ACRONYMS

BP	Byte Pattern
DES	Data Encryption Standard
GK	Group Key
GKM	Group Key Manager
KAP	Key Agreement Protocol
KDT	Key Distribution Technique
KEK	Key Encrypting Key
LKH	Logical Key Hierarchy
PBS	Perfect Backward Secrecy
PFS	Perfect Forward Secrecy
PBaFS	Perfect Backward and Forward Secrecy
RM	Rekey Message

VITA

Sahar Mohamed Ghanem was born in Alexandria, Egypt, on May 24th, 1972. She received her Bachelor of Science in Computer Science and Automatic Control from The Faculty of Engineering, Alexandria University, Egypt, in June 1994. She worked as a Teaching Assistant for the Department of Computer Science at The Arab Academy for Science and Technology from September 1994 to May 1997. She worked as a Teaching Assistant for the Computer Science and Automatic Control Department at The Faculty of Engineering, Alexandria University, Egypt, from June 1997 to December 1997. In December 1997, she received her Master of Science degree from the Department of Computer Science at The Faculty of Engineering, Alexandria University, Egypt. She started working on her Ph.D. Degree in Computer Science at Old Dominion University, Virginia, in January 1998. During the course of her Ph.D. work, she co-authored ten scientific papers and technical reports.

Permanent address: Department of Computer Science
Old Dominion University
Norfolk, VA 23529-0162
USA