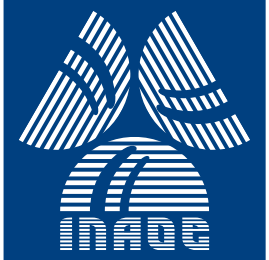

Efficient mechanism for key management in multi-session environments

por

José Roberto Pérez Cruz

Tesis sometida como requisito parcial para obtener el grado de

MAESTRO EN CIENCIAS EN EL ÁREA DE CIENCIAS COMPUTACIONALES

en el

Instituto Nacional de Astrofísica, Óptica y Electrónica
Noviembre 2009
Tonantzintla, Puebla

Supervisada por:

Dr. Saúl Eduardo Pomares Hernández, INAOE
Dr. Gustavo Rodríguez Gómez, INAOE

Abstract

The Internet 2 deployment introduces new communications capabilities, like multi-party collaboration, high scale multimedia assembly and multicast communication. Considering such capabilities, the research concering security is facing new challenges. One of such challenge, is to create secure multi-session frameworks to ensure the confidentiality of exchanged information. In a multi-session environment, there are several users joined at two or more work sessions simultaneously. The confidentiality in these environments can be achieved using cryptographic methods. Unfortunately, the key management, which is necessary for such environments, creates two problems: a high complexity in key distribution and a high storage cost.

This thesis proposes as its main contribution, an efficient decentralized multi-session key management mechanism for dynamic multimedia group communication, characterized by the use of an independent key per ciphered packet. Our solution proposes a functional architecture that exploits the overlapping of the user sessions to reduce the redundancy in key distribution. The proposed architecture makes use of two key generation strategies: a key derivation technique to reduce the rekey overhead and a pseudorandom number generator that allows the users to generate an independent key per ciphered packet. The pseudorandom number generator allows the users to generate independent keys for each transmitted packet. This characteristic enables the system to support the delay, the loss and transposition of packets.

Resumen

La implementación de Internet 2 ha introducido nuevas capacidades en las comunicaciones, como la colaboración multipartita, el ensamble de multimedia a gran escala y comunicaciones multicast. Considerando tales capacidades, la investigación en el ámbito de la seguridad enfrenta nuevos retos. Uno de ellos es crear ambientes de trabajo multi-sesión seguros para garantizar la confidencialidad de la información intercambiada. En los ambientes multi-sesión existe una gran cantidad de usuarios involucrados con dos o más sesiones de trabajo de manera simultanea. La confidencialidad en dichos ambientes se puede lograr usando métodos criptográficos. Desafortunadamente la gestión de las llaves, necesaria para tales métodos, crea dos problemas: una alta complejidad en la distribución y un alto costo de almacenamiento.

En esta tesis se propone, como principal contribuación, un mecanismo descentralizado basado en el paradigma de comunicación en grupo, para la gestión de llaves en ambientes multi-sesión. Dicha solución propone una arquitectura funcional que aprovecha el traslape en la sesiones de los usuarios para reducir la redundancia en la distribución de las llaves. La arquitectura propuesta emplea dos estrategias de generación de llaves: una técnica de derivación y un generador de números pseudoaleatorios. Con la técnica de derivación se logra reducir el costo involucrado en la actualización de las llaves, mientras que con el generador de números pseudoaleatorios los usuarios pueden generar llaves independientes para cada paquete cifrado, habilitando al sistema para soportar el retraso, la pérdida o la transposición de paquetes.

Contents

Notation

The notation used in this document is summarized in Table 1.

Table 1: General notation

Symbol	**Meaning**
KEK	Key Encryption Key
TEK	Transfer Encryption Key
DEK	Data Encryption Key
SK	Session Key
BK	Blinded Key
KDC	Key Distribution Center
LKH	Logical Key Hierarchy
KEK-tree	hierarchy of KEKs
rKEK	the key located in the root vertex of a KEK-tree
SG	Service Group (group of users that have access to the same resources)
DG	Data Group (group of users that have access to a unique data stream)
OG	Overlapping Group (group of users with overlapped sessions)
OGC	Overlapping Group Controller
n	number of users in the system
s	number of sessions in the system
t	identifier of an OG and its KEK-tree
OG_t	overlapping group related to the KEK-tree t

Continued on next page

Table 1 – continued from previous page

Symbol	Meaning
OGC_t	overlapping group controller responsible of the OG_t
n_t	number of users in an OG_t
$m(s)$	maximum number of OGs
m_0	number of OGs in the system
d	degree of a KEK-tree
$l_d(n_t)$	length of the branches of a tree and is either L or $L+1$, where $L = \lfloor log_d(n_t) \rfloor$
i, j	indices of KEKs
$K_{i,j}^t$	KEK of the tree t
Ω_t	set of SKs related to an OG_t
SK_h	session key related to the session h ($1 \leq h \leq s$)
BK_h	blinded key related to the SK_h
b_h	order of an algebraic group
p, q, p_h, q_h	large prime integers
k', K', SK'	the apostrophe indicates that the key has been updated
$\{K_i\}_{K_j}$	indicates that the key K_i is encrypted by the key K_j
$\rightarrow$	indicates the transmission of a message
$KDC \rightarrow u_n : \{K_i\}_{K_j}$	indicates that the KDC sends to user u_n the key K_i encrypted by the key K_j
$\equiv$	denote a congruence modulo n ($a \equiv b \mod n$ if $a - b$ is divisible by n)
$\oplus$	denote a direct sum, that is a construction which combines several modules into a new larger module

Continued on next page

Table 1 – continued from previous page

Symbol	Meaning
$\triangle$	denote the symmetric difference of two sets, in other words is the set of elements which are in one of the sets, but not in both

Chapter 1

Introduction

1.1 Motivation

The Internet 2 enables new capabilities, such as multi-party collaboration, high-scale multimedia assembly and multicast communication. The aim of these new capabilities is to develop communication platforms where there may be a high-scale associativity of users communicating by using multimedia data (audio, video, text, still images, etc.). To achieve this, the communication platforms contemplate the support of heterogeneous data management (transmission of discrete and continuous data) which must satisfy certain properties in order to not degrade the quality of service [SWM+01]. Specifically, for continuous data transmission, the communication should support the delay, the loss and the transposition of packets.

1.2 Problem description

For the reasons mentioned above, the new communication platforms are facing new challenges about security research. One such challenge is to create secure environments where several applications and several users maintaining work sessions in two or more applications can exist simultaneously. A description of a multi-session environment is given by the following example.

Suppose that there are several users on the Internet using some or all of the following three applications: an interactive meeting through session 1, an iTV show through session 2, and a multimedia forum through session 3. Some users coincide in one, two

or three of these applications, but there are many other users that do not coincide at all among the applications used (see Figure 1.1).

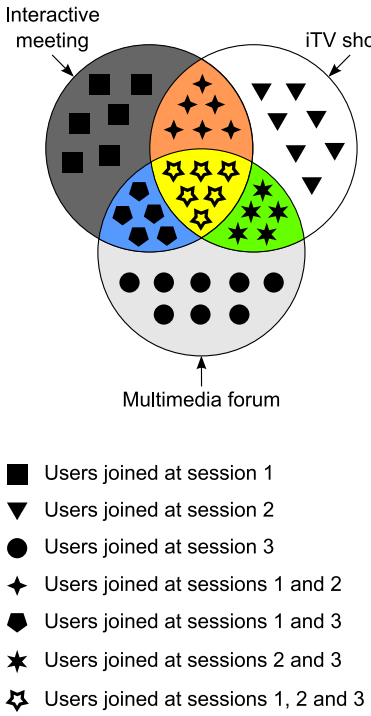

Figure 1.1: Multi-session scenario with three different applications

As shown in Figure 1.1, users in orange, green and blue areas coincide in two applications respectively, while users in the yellow area coincide in all the applications. Users that coincide in some applications are said to have an *overlapping* in their sessions. With this scenario, users involved in sessions 1 and 3 may exchange information with users associated with these sessions, but users associated only with session 3 should not have access to the information exchanged in session 1 and vice versa. In other words, the exchanged information should be confidential, which means that the information can be accessed only by entities or groups of authorized entities [HDP03, FK04, MVO96].

A practical way to assure the confidentiality is with the use of cryptographic methods along with a selective key distribution technique [RH03]. Unfortunately, the key management in multi-session environments presents two main problems: a high complexity in key distribution and a high storage cost. These problems arise because users have

to store independent keys for each joined session, and the number of keys required for rekeying depends on the number of users in each session.

Besides the problems associated with key management in multi-session environments, special requirements must be met to preserve the quality of service for continuous data transmission. Since communication channels are not necessarily reliable nor ordered, there is no guarantee that all information is received correctly [CQN+02]. Therefore, systems should support the delay, loss and transposition of packets. For these requirements, it is desirable to use independent keys per transmitted packet, so that the lost, delayed or transposed packet has no adverse effects on the quality of service. Furthermore, if there are as many keys as transmitted packets, the level of confidentiality would be even greater.

Currently, some solutions exist that are designed for environments where there are several users with different access privileges or which are associated with different work sessions [SL03, SL04, WOCG07, DML04, GLLC05, RLK05]. Unfortunately, such solutions are not designed to support dynamic formation and decomposition of groups since they rely on complex architectures to organize users and keys.

1.3 Proposed solution

In this thesis, we propose as main contribution, an efficient decentralized multi-session key management mechanism for dynamic multimedia group communication, which is characterized by the use of an independent key per ciphered packet. Our mechanism uses a functional architecture that exploits the overlapping present in the user's sessions, creating groups composed of users with the same memberships[1] to reduce the key distribution redundancy. Furthermore, the proposed architecture is designed to support the dynamic formation and decomposition of groups.

[1]In this context, a membership is the state of a user of being a member of a *communication group* (see Section 2.1 for a definition). Being a member of a communication group, a user has access privileges to some programs or resources.

Along with the architecture, our mechanism uses two key generation strategies: a derivation technique and a pseudorandom number generator. Through the derivation technique, each formed group is organized into an independent hierarchy to manage the auxiliary keys used in the rekeying, while it allows the members of each group to derive the auxiliary keys by themselves, without the Key Distribution Center (KDC) having to generate, encrypt and distribute all the keys. With the pseudorandom number generator, our mechanism allows the users to generate independent keys for each transmitted packet. By the way in which independent keys are generated, unlike many current solutions where users can only decrypt a specific stream in a 1 to n communication, in our solution, users can encrypt and decrypt different streams in an n to n communication.

1.4 Methodology

In order to achieve the objective of this thesis, the following methodology was proposed.

1. **Design of an organizational architecture.** This concerns the design of a structure that allows the organization of the auxiliary keys, needed by the users of a system to perform the rekeying. In a multi-session environment, a user involved with several sessions must handle several keys, which may create a significant increment in storage and communication costs, respect to the current solutions about multi-group key management, if a good strategy is not used. For this reason, it is necessary to take advantage of the coincidences and overlapping in user memberships to reduce storage and distribution costs, eliminating redundancy in managed keys. Therefore, for a multi-session environment, the architecture must organize users according to their memberships, forming groups with users who have the same memberships.

2. **Development of a mechanism for dynamic rekeying.** This refers to the development of a key distribution scheme based on the organizational architecture. A key distribution scheme consists on the key generation methods and strategies used

by the group controllers and users to perform the rekeying and the transmission of packets. The key distribution scheme depends entirely on the organizational architecture because the architecture establishes the correspondence between users and keys. Therefore, the key generation methods and strategies must be appropriate to be adapted to the organizational architecture.

3. **Development of a centralized key management mechanism.** Since a centralized approach has less restrictions than others, it is a good environment to develop a first solution that uses the achieved functional architecture. With the centralized mechanism, the first rekey algorithms are developed. Such algorithms are used to handle the join, leave or switch of users in the system.

4. **Development of a decentralized key management mechanism.** The achieved centralized scheme has to be extended in order to solve the inherent problems of a centralized approach. The main problem of a centralized approach is to have a system with a single point of failure (a single group controller). Consequently, if the group controller fails, the entire group is affected. Furthermore, as the group controller has to handle all the groups, the centralized mechanism is scalable only in the number of users, but it is not scalable in the number of groups.

5. **Performance analysis.** In order to demonstrate the efficiency of the mechanisms, the storage and communication costs have to be calculated. With the calculated costs, the mechanisms must be compared to other solutions in order to corroborate that the proposed solutions are really efficient.

1.5 Document organization

This document is organized as follows:

In chapter two, the main concepts involved with key management for secure group communications are introduced.

In chapter three, a survey of key management for secure group communications is presented. The survey includes the multi-group case, where an extension is proposed to the current taxonomy of key management mechanisms.

In chapter four, the proposed mechanisms for multi-session key management are explained. First, a centralized multi-session key management mechanism is defined, where the functional architecture and the rekey algorithms are detailed. Second, in order to achieve a more scalable mechanism, a decentralized mechanism is defined. The decentralized mechanism extend the functional architecture achieved in the centralized mechanism.

In chapter five, the performance analysis of the proposed mechanisms is presented. Communication and storage costs of the two defined mechanism are calculated. In addition, costs are compared with other similar solutions.

Finally, the conclusions of this work are summarized in chapter six.

Chapter 2

Fundamentals

2.1 Secure group communications

Today, one of the fastest growing communication models is the multicast, since it allows a more efficient dispatch of messages on each link of a network. In a multicast communication, several receivers are associated with an address so that when a user needs to transmit a message to a set of users, the user only needs to send the message once to a single address. Thus, multicast allows the reduction in bandwidth. In this context, the set of receivers associated with an address is called communication group [HDP03].

Group confidentiality. When a communication group is established, it is expected that within it exists some confidentiality when the participants exchange information; therefore it is necessary to establish access control policies to ensure that only authorized entities can access the group and the information exchanged. A practical way to restrict access to information, is the use of cryptography with a selective distribution mechanism of cryptographic keys [HDP03, RH03].

In this environment, a cryptographic algorithm takes a piece of information, such as a group's message, and using an appropriate key, it performs some transformations to encrypt the original data, making impossible to retrieve such information, unless the corresponding decryption algorithm and the corresponding key were used [MVO96, FS03]. This makes it possible to establish private multicast sessions, where each message is encrypted with an algorithm and a selected key; thus, only the group members can recover such information. In this context, the selected key is called the *group key*.

Update of group keys. Besides the need to protect the information exchanged between the group members, it is necessary to provide a mechanism to renew the group key (*rekey*) when an entity joins or leaves the group [RH03]. The rekey mechanism is performed by a key manager.

When an entity joins the group, the system must guarantee that the new member cannot access the information exchanged before its arrival, such warranty is called *backward secrecy*. Similarly, when an entity leaves the group, the system must guarantee that the member that leaves the group cannot access the information exchanged after its departure, such warranty is called *forward secrecy* [RH03].

Backward secrecy preservation often has a different complexity than *forward secrecy* preservation. To preserve the *backward secrecy*, at user arrival, a new group key is chosen, and by using a multicast communication the new group key is distributed among the old members, while for the new user, the key is sent by unicast. *Forward secrecy* preservation is often more complex because when a user leaves the group, it knows the encryption method and some of the keys used to decrypt information, so there are cases in which a single multicast transmission of the new key does not offer a good security level. There are several approaches that propose a solution to the *forward secrecy* preservation. Some of the first solutions proposed the distribution of the new keys using $O(n)$ unicast transmissions that were extremely costly [HM97b, HM97a, MJMR99]. In some of the recent approaches, derivation strategies have been proposed, with which users can calculate the keys by themselves, making it necessary to transmit a simple message to request the rekey.

2.1.1 Multi-group environments

When some entities, participating in a multicast communication, belong to two or more groups simultaneously, a multi-group environment is established. In a multi-group communication, *backward secrecy* and *forward secrecy* must be guaranted in each of the groups involved, although the members can interact with each other. This implies that

in a multi-group transmission, only the members of the destination group can access the information exchanged, making such information inaccessible to the rest of the entities involved with the user, but that are outside of the destination group.

In a multi-group environment, confidentiality must be maintained within each group. For this, users have to store a set of keys for each group to which they are associated. Consequently, key management in a mult-group environment may become redundant, especially if the access control services are performed individually in each group. A practical way to solve the problem of redundancy in key management is by taking advantage of the concurrency that some users may have in the same groups. In this context, when two or more users concur in the same groups, it is said that they have an "overlapping " in their memberships. In this way, the efficiency and the scalability of a system depends on the way that the overlapping in user memberships is managed.

2.2 Group key managers

Depending on the allowed level of association, the key management mechanisms can be generally classified in one of two categories: single-group key management mechanisms and multi-group key management mechanisms.

Problems related to the key management in a single-group environments have been a topic widely studied, evidence of this are the works of Rafaeli-Hutchison [RH03] and Challal et al. [CS05, YCS05]. These works have agreed to define a taxonomy for the different key management mechanisms for single-group environments, which consists of three main categories:

- **Centralized scheme**. In this scheme, a single entity is responsible for controlling the whole group, assuming the responsibility for generating and distributing the keys, reducing the computational cost and storage requirements in the rest of the group members. In addition, with this scheme, the enter and exit of members is accomplished easily while maintaining the synchrony of the system. The prob-

lem with this scheme is that if the central entity fails, the whole group stops its operation.

- **Decentralized scheme**. In this scheme the responsibility for managing a group is divided among several entities, forming subgroups, making the user's enter and exit transactions being undertaken by several entities, while minimizing the problem of concentrating the work into a single entity. The main problem with these schemes is the complexity involved in synchronization.

- **Distributed scheme**. This scheme is characterized by the lack of a group controller, so that all members work together to generate the keys. The main problem with this scheme is that the synchronization time increases significantly, so the enter and exit of users become complex.

Problems related to the key management in multi-group environments, as a special case of secure group communications, is a topic that has been recently explored, nevertheless some important solutions have emerged. According to existing solutions, key management mechanisms for multi-group environments can be classified into two main categories:

- **Centralized scheme**. In this category, those mechanisms where a central entity controls all the groups in the system are considered, assuming the responsibility for generating and distributing the keys. In addition, the central controller maintains complex structures to organize the keys and users of the system.

- **Decentralized scheme**. In this category, the mechanisms where there is more than one entity responsible for controlling the groups are considered.

A detailed description of the works of each category is presented in Chapter 3.

2.3 Key generation

A cryptographic key is a piece of information as a sequence of numbers or letters. For this reason, it must meet certain requirements to provide an acceptable security level. In this way, the security of a key is mainly determined by the computational difficulty to be discovered and its randomness. There are certain mathematical functions, which by their nature, can be used to generate robust cryptographic keys.

Some of the most common functions used for key generation are: one-way functions, trapdoor one-way functions, Hash functions and pseudorandom number generators. For a more detailed description refer to [MVO96].

2.3.1 One-way functions

The one-way functions are mathematical functions that have the property of being easy to compute but unfeasible to invert.

Definition. A function f defined from a set X to a set X is called one-way function if $f(x)$ is easy to compute for all $x \in X$, but for "essentially all" elements $y \in Im(f)$, it is computationally unfeasible to find any $x \in X$ such that $f(x) = y$.

In this context, terms "easy" and "unfeasible" refer to the capability of an algorithm to compute a certain function in a polynomial time, depending on the size of the entry.

2.3.2 Trapdoor one-way functions

Definition. A trapdoor one-way function is a one-way function $f : X \to Y$ with the additional property that, given some extra information, called the trapdoor information, it becomes feasible to find for any given $y \in Im(f)$, an $x \in X$ such that $f(x) = y$.

In this way, finding the inverse function of a trap without knowing the extra information is a computationally difficult problem.

Trapdoor one-way functions are used mainly in public key cryptographic algorithms, such as the techniques for key exchange Diffie-Hellman and RSA.

2.3.3 Hash functions

Definition. A hash function is a computationally efficient function mapping binary strings of arbitrary length to binary strings of some fixed length, called *hash-values*.

A hash function h is usually chosen in such way that it is computationally unfeasible to find two different inputs that have a common hash value, and it is computationally impossible to find a pre-image of x such as $h(x) = y$.

Hash functions are used mainly to create authentication codes, such as a digital signature, or to create verification codes, which are used to verify the integrity of data received in a transmission.

2.3.4 Pseudorandom number generators

To consider a number or sequence to be random, the set where it is taken should be an equiprobable space; in other words, every element must have the same chance of being elected and that the choice of one does not depend on the choice of another. This implies that choosing a random number depends entirely on the *Hazard*, so to obtain a source of true randomness, it must involve physical media. In order to overcome such adversity, most existing methods have been devised to build deterministic pseudorandom sequences, from a small random sequence called *seed*. In this way, the generated sequences appear to proceed from a real random sources, for anyone who does not know the generation method. For practical purposes, the generation algorithm is often disclosed, but the seed is kept secret.

Therefore, a pseudorandom number generator algorithm can be defined as a deterministic process which takes as input a random number s, called the seed, and has as output a sequence of *quasi*-random numbers of length n.

Pseudorandom number generators, are used mainly in stream ciphers, where information is encrypted in small fragments in real time.

Applications that use this kind of cryptosystems are those that handle continuous data streams, such as telephone calls or video transmissions.

Chapter 3

State of the art

3.1 Taxonomy of key management mechanisms

This thesis proposes an extension to the taxonomy previously defined through the works of Rafaeli- Hutchison [RH03] and Challal et al. [CS05, YCS05], to consider the special case of the key management in multi-group environments. The resulting taxonomy is shown in Figure 3.1, which is based on the organization proposed by Challal et al.

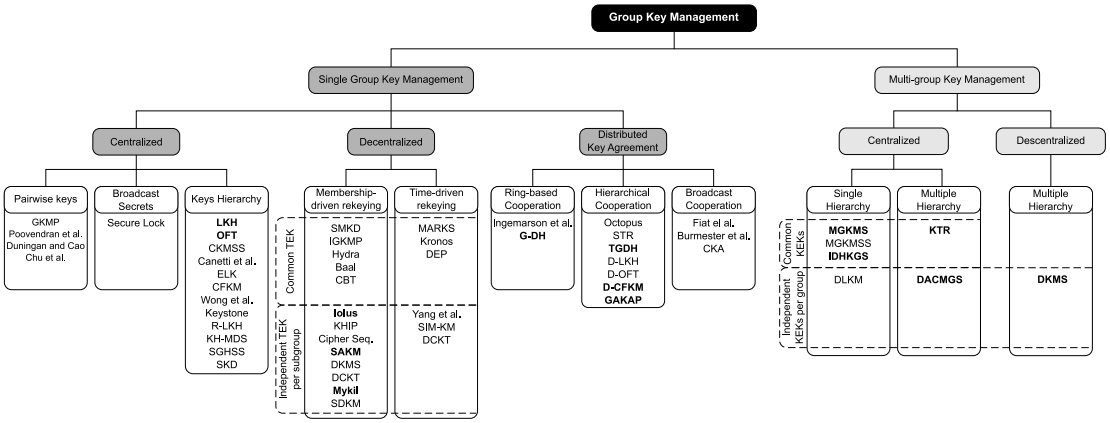

Figure 3.1: Taxonomy of key management mechanisms for secure group communications

In this thesis we will discuss only the main classical contributions that continue to be a reference as well as some of the most recent contributions; therefore, not all the mechanisms and schemes shown in Figure 3.1 are explained, although they have been considered in the taxonomy, to maintain a frame of reference. Mechanisms and schemes shown in Figure 3.1 that are not discussed in this document can be found in the works of Rafaeli-Hutchison [RH03] and Challal et al [CS05, YCS05].

3.2 Key management for single group environments

3.2.1 Centralized key management

In a centralized system, an entity called as the Group Controller (GC) is responsible for controlling the whole group. The GC is used as a Key Distribution Center (KDC) because it generates and distributes the keys to the whole group. The KDC does not depend on any other entity to perform the access control and key distribution. However, precisely such skill makes this central entity a potential single point of failure: if the KDC has problems, the whole group will be affected. If the KDC stops working, the group privacy will be compromised since the keys in which the group privacy are based on, will not be generated, updated nor distributed [RH03].

Despite the problems that might present a centralized mechanism, there are certain advantages, such as the reduction of storage and computational costs in the rest of the group members. Furthermore, the input and output of users becomes easier, increasing the scalability of the group.

The efficiency of a centralized key management mechanism can be measured considering the following parameters [RH03, CS05]:

- *Communication cost.* Refers to the number of messages required to update the keys when users join or leave.

- *Computational cost.* This parameter relates to the amount of computations required by the group members and the KDC to manage the keys.

- *Storage cost.* Refers to the number of keys that group members and the KDC need to keep.

- *Preservation of backward and forward secrecy.* It refers to the system's ability to maintain confidentiality of exchanged information, even when users join or leave the group.

- *Collusion.* When certain members leave the group, they should not be able to interact to derive the *group key.*

3.2.1.1 LKH

In the mechanisms based on Logical Key Hierarchy (LKH) [WGL98, DW99], the KDC maintains a tree of keys, where the root is the Transfer Encryption Key (TEK), and the rest of the vertices are Key Encryption Keys (KEKs) (see Figure 3.2). Particularly, the KEKs of the leaves are the individual keys of the group members, and each one of those keys is known only by the KDC and the corresponding user.

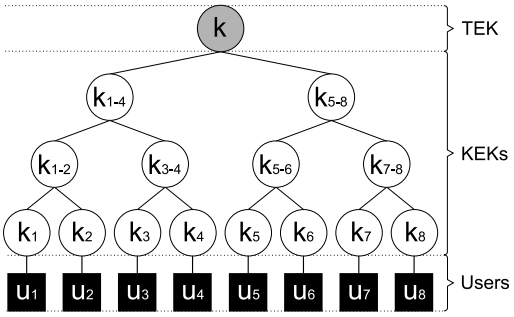

Figure 3.2: Logical key hierarchy architecture

With such organization, each group member has to store all the keys located along the tree branch where it is joined, from its individual key to the TEK.

When a new user joins the group, a new leaf is added to the tree and is associated with the individual key of that user. In order to preserve the *backward secrecy*, all the keys in the vertices of the branch of the new leaf are updated by the KDC. To distribute the updated keys, the KDC encrypts each key with the keys related to its children.

The user join process is illustrated by the Figure 3.3. When user u_3 joins the group, the KEK k_3 and its corresponding leaf are added to the tree, compromising the keys k_{3-4}, k_{1-4} and k. Therefore, the KDC distributes the updated keys as follows: k' is encrypted with k'_{1-4} and k_{5-8}, k'_{1-4} is encrypted with k_{1-2} and k'_{3-4}, and k'_{3-4} with k_3 and k_4.

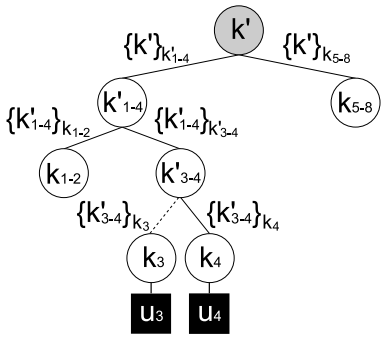

Figure 3.3: User join process in LKH scheme

When a user leaves the group, the KDC removes the corresponding leaf and updates the keys in the affected branch in order to preserve the *forward secrecy*. As for the user join process, each updated key is encrypted with the keys related to its children and the rekey message excludes the key of the departed user. The user leave process is illustrated by Figure 3.4: when user u_3 leaves the group, the KEK k_3 and its corresponding leaf are removed from the tree, compromising the keys k_{3-4}, k_{1-4} and k. Therefore, the KDC distributes the updated keys as follows: k' is encrypted with k'_{1-4} and k_{5-8}, k'_{1-4} is encrypted with k_{1-2} and k'_{3-4}, and k'_{3-4} with k_3.

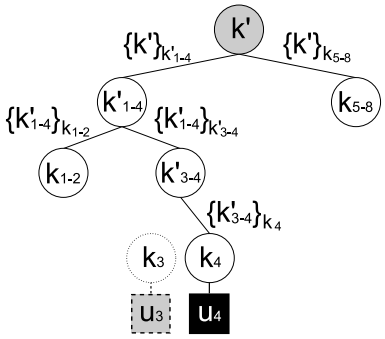

Figure 3.4: User leave process in LKH scheme

3.2.1.2 OFT

One-way Function Tree (OFT) [SM03] is a solution based on binary key trees, that reduces the communication cost of the LKH scheme. Using a OFT, the group members can calculate the keys of its branch, reducing the information sent by the KDC.

In the OFT, the set of keys located along a user's branch, from the individual key to the TEK, is called *ancestor set*. Unlike the LKH scheme, with the OFT, each group member stores its individual key and the keys located in the sibling vertices of its *ancestor set*. The set of such keys is called *sibling set* (see Figure 3.5). Each key in the *ancestor set* is *blinded* using a one-way function $g(\cdot)$.

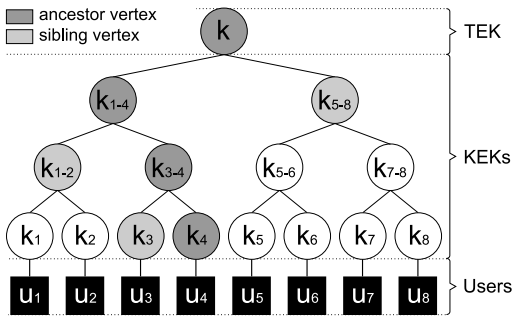

Figure 3.5: Ancestor and sibling sets of member u_4

Thus, each user can compute the keys of its branch using the following formula:

$$k_i = f(g(k_{left(i)}), g(k_{right(i)})) \tag{3.1}$$

where $f(\cdot)$ is a mixing function, $k_{left(i)}$ and $k_{right(i)}$ denote respectively the left and right children of vertex i, that are *blinded* using a one-way function $g(\cdot)$.

When a rekeying operation is needed, the KDC just sends the updated KEKs of the affected branch, blinding each key with the one-way function $g(\cdot)$. Figure 3.6 the KEKs sent out by the KDC for a rekey process.

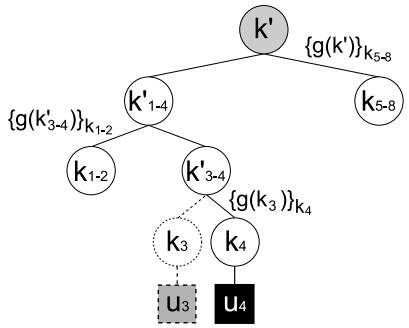

Figure 3.6: Necessary encryptions when u_3 joins the group

Considering the tree shown in Figure 3.6. If a new user u_3 joins the system, the KDC sends the blinded keys: $g(k'_{1-4})$ encrypted with k_{5-8}, $g(k'_{3-4})$ encrypted with k_{1-2} and $g(k_3)$ encrypted with k_4. Therefore, the new KEKs can be calculated by every member as:

$$k'_{3-4} = f(g(k_3), g(k_4))$$
$$k'_{1-4} = f(g(k_{1-2}), g(k'_{3-4}))$$
$$k' = f(g(k'_{1-4}), g(k_{5-8}))$$

3.2.1.3 SGHSS

The Subgroup Hierarchy with Secret Sharing (SGHSS) [NPKKI07] was designed to reduce the communication cost in the rekey operation involved with the leave process, allowing those group members to calculate the keys by themselves. With the SGHSS, the KDC maintains a tree, where the root is the TEK, the internal vertices are KEKs, and unlike other schemes, each leaf is a key shared by the members of a subgroup.

For the generation of keys of the different levels of the LKH, the KDC uses a modular exponentiation over a finite field. Thus, the KDC distributes some secrets among the users, and such secrets are used by the users to calculate the TEK and the key of their subgroup.

A secret is defined as the exponent α that satisfies the following property: let p be a large prime and let g be the primitive element of the multiplicative group $\mathbb{Z}_p^*$. It is computationally difficult to determine α given g and $g^\alpha \mod p$.

In a group there are two kinds of secrets: the *member secrets* and the *server secrets*. The *member secret* α_j^i, assigned to user i of the subgroup j, is selected under the condition that $2 \leq \alpha_j^i \leq p-2$ and $gdc(\alpha_j^s, p-1) = 1$. The *server secret*, assigned to subgroup j is selected under the condition that $2 \leq \alpha_j^s \leq p-2$. Using both secrets, the key of the

subgroup j is computed by:

$$K_j \equiv g^{\alpha_j^1 \alpha_j^2 ... \alpha_j^m \alpha_j^s} \mod p \tag{3.2}$$

In this way, each key located in an internal vertex is generated by the multiplication of the exponents of its children, and each subgroup key is generated by the multiplication of the exponents of the users. Furthermore, the inverse value of each *member secret* is used to exclude a user when it leaves the group.

Figure 3.7 shows a group divided in 3 subgroups with m users and a subgroup with $m-1$ users.

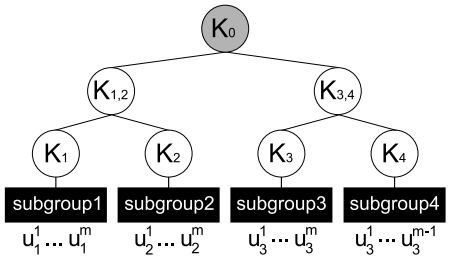

Figure 3.7: Key hierarchy for 4 subgroups

For the example of Figure 3.7, the subgroup keys K_1, K_2, K_3, K_4, the intermediate keys $K_{1,2}$, $K_{3,4}$ and the TEK K_0 are computed by:

$$\begin{aligned}
K_1 &\equiv g^{\alpha_1^1 ... \alpha_1^m \alpha_1^s} \mod p \\
K_2 &\equiv g^{\alpha_2^1 ... \alpha_2^m \alpha_2^s} \mod p \\
K_3 &\equiv g^{\alpha_3^1 ... \alpha_3^m \alpha_3^s} \mod p \\
K_4 &\equiv g^{\alpha_4^1 ... \alpha_4^{m-1} \alpha_4^s} \mod p \\
K_{1,2} &\equiv g^{(\alpha_1^1 ... \alpha_1^m \alpha_1^s)(\alpha_2^1 ... \alpha_2^m \alpha_2^s)} \mod p \\
K_{3,4} &\equiv g^{(\alpha_3^1 ... \alpha_3^m \alpha_3^s)(\alpha_4^1 ... \alpha_4^{m-1} \alpha_4^s)} \mod p \\
K_0 &\equiv g^{(\alpha_1^1 ... \alpha_1^m \alpha_1^s)(\alpha_2^1 ... \alpha_2^m \alpha_2^s)(\alpha_3^1 ... \alpha_3^m \alpha_3^s)(\alpha_4^1 ... \alpha_4^{m-1} \alpha_4^s)} \mod p
\end{aligned}$$

Considering the example of the group shown in Figure 3.7, if a new user joins the group, the KDC determines the subgroup that accommodates the new member. If subgroup 4 is selected, the KDC assigns the individual key $K_{p_m^4}$ and the secret α_4^m to the new user (u_4^m), and calculates the inverse value α_4^{-m}. Then, the KDC updates the *server secret* α_4^s, the keys K_4, $K_{3,4}$ and K_0, using α_4^m and the updated *server secret* as follows:

$$
\begin{aligned}
K_4' &\equiv g^{\alpha_4^1 ... \alpha_4^{m-1} \alpha_4^m \alpha_4^{s'}} \mod p \\
K_{3,4}' &\equiv g^{(\alpha_3^1 ... \alpha_3^m \alpha_3^s)(\alpha_4^1 ... \alpha_4^{m-1} \alpha_4^m \alpha_4^{s'})} \mod p \\
K_0' &\equiv g^{(\alpha_1^1 ... \alpha_1^m \alpha_1^s)(\alpha_2^1 ... \alpha_2^m \alpha_2^s)(\alpha_3^1 ... \alpha_3^m \alpha_3^s)(\alpha_4^1 ... \alpha_4^{m-1} \alpha_4^m \alpha_4^{s'})} \mod p
\end{aligned}
$$

Finally, the KDC unicasts to u_4^m, the keys K_4', $K_{3,4}'$ and K_0' and the inverse values of the other members of the subgroup 4 ($\alpha_4^{-1}, ..., \alpha_4^{-m-1}$), encrypting the message with $K_{p_m^4}$. For the rest of group members, the KDC multicasts (α_4^{-m}, K_4') encrypted with K_4, $K_{3,4}'$ encrypted with $K_{3,4}$ and K_0' encrypted with K_0.

In the case when user u_4^m decides to leave the group, the KDC just has to notify the remaining members of subgroup 4 about the abandonment, and multicasts the inverse α_4^{-m} to the others subgroups. Thus the KDC multicasts α_4^{-m} encrypted with K_3 and $K_{1,2}$, respectively. Thus, the KDC and the remaining members update the keys as follows:

$$
\begin{aligned}
K_4' &\equiv (K_4)^{\alpha_4^{-m}} \equiv g^{\alpha_4^1 ... \alpha_4^m \alpha_4^{-m} \alpha_4^s} \equiv g^{\alpha_4^1 ... \alpha_4^{m-1} \alpha_4^s} \mod p \\
K_{3,4}' &\equiv (K_{3,4})^{\alpha_4^{-m}} \equiv g^{(\alpha_3^1 ... \alpha_3^m \alpha_3^s)(\alpha_4^1 ... \alpha_4^{m-1} \alpha_4^s)} \mod p \\
K_0' &\equiv (K_0)^{\alpha_4^{-m}} \equiv g^{(\alpha_1^1 ... \alpha_1^m \alpha_1^s)(\alpha_2^1 ... \alpha_2^m \alpha_2^s)(\alpha_3^1 ... \alpha_3^m \alpha_3^s)(\alpha_4^1 ... \alpha_4^{m-1} \alpha_4^s)} \mod p
\end{aligned}
$$

3.2.1.4 SKD

The Shared Key Derivation protocol (SKD) [LLL05, LHLL09] uses a tree similar to LKH to organize the users involved in the group. The novelty in SKD is that keys located in the internal vertices are used by the users to compute or *derive* the new keys

by themselves, avoiding the KDC the need to generate and distribute all the keys in a rekey operation.

In SKD, the KDC and the group members use a one-way function $f(\cdot)$, called the key derivation function, to compute the new keys, using previous keys.

In the key tree used in SKD, the vertices are denoted as $x_{i,j}$, where i is the level of the vertex and j is the most left position relative to the level i (see Figure 3.8).

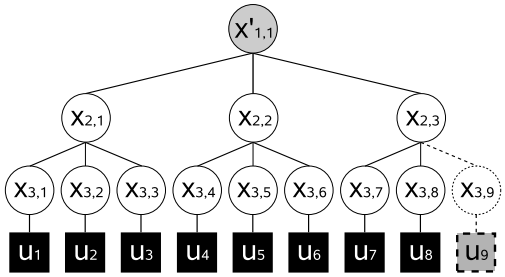

Figure 3.8: SKD architecture

When a new user joins the group, the KDC assigns a new vertex to store the individual key of the user. As the key tree must be balanced, the insertion of a new vertex depends on whether the subtree chosen for insertion is full or not. Assuming that x_{h,p_h} is the root vertex of subtree y_{h,p_h}, the insertion of a new vertex is performed in one of two ways:

- if y_{h,p_h} is not full, the new vertex $x'_{h+1,p_{h+1}}$ is inserted
- if y_{h,p_h} is full, a child of x_{h,p_h}, ie the vertex $x_{h+1,p_{h+1}}$ is moved to a lower level, becoming the vertex $x'_{h+2,s_{h+2}}$ and its old position is replaced by a new intermediate vertex $x'_{h+1,p_{h+1}}$, which will be the new parent of $x'_{h+2,s_{h+2}}$ and the new inserted vertex $x'_{h+2,p_{h+2}}$

When the new vertex is inserted, all the compromised keys are updated by:

$$k'_{i,p_i} = f(k_{i,p_i}) \tag{3.3}$$

where k_{i,p_i} is the KEK previous to the rekeying, called the derivation key of vertex x_{i,p_i}.

If the chosen subtree was full and the creation of a new intermediate vertex $x'_{h+1,p_{h+1}}$ was needed, the key of the new intermediate vertex is calculated by:

$$k'_{h+1,p_{h+1}} = f(k_{h+2,s_{h+2}} \oplus k_g) \tag{3.4}$$

where $k_{h+2,s_{h+2}} = k_{h+1,p_{h+1}}$ is the derivation key, corresponding to the previous vertex $x_{h+1,p_{h+1}}$, while k_g is the TEK ($k_{1,1}$) called the *salt value*, that is used to ensure that the derived key is different to the other previously generated keys.

For the rekeying, the KDC just sends a message, informing the current members about the recent user addition. As the new user does not know the KEKs, the KDC unicasts the involved KEKs, encrypting the message with the individual key of the new user.

When a user leaves the group, the KDC removes the corresponding vertex and updates the KEKs of the affected branch. As the key tree must be balanced, the removal of the vertex is performed in the following way: if the vertex x_{h,p_h}, the root of subtree y_{h,p_h} stays with at least two children, the key x_{h,p_h} is replaced by the only child, $x'_{h,p_h} = x_{h+1,p_{h+1}}$. In any case, the new KEKs k'_{i,p_i} are computed by:

$$k'_{i,p_i} = f(k_{i+1,s_{i+1}} \oplus k_{i,p_i}) \tag{3.5}$$

where $k_{i+1,s_{i+1}}$ is the derivation key, and the previous key k_{i,p_i} is used as *salt value*. As the derivation keys only benefit users in subtree $y_{i+1,s_{i+1}}$, the KDC multicasts the updated keys to the members who cannot derive the keys.

3.2.1.5 Summary

In Tables 3.2, 3.3 and 3.4, the mechanisms and schemes discussed in previous sections are compared. Table 3.2 shows a comparison of the communication costs and indicates the security services offered by each solution. Table 3.3 shows a comparison of storage costs. Finally, Table 3.4 shows a comparison of the computational cost.

The notation used in Tables 3.2, 3.3, and 3.4 is described in Table 3.1:

Table 3.1: Notation used in Tables 3.2, 3.3 and 3.4

Symbol	Meaning
n	number of users in the group
m	number of subgroups
d	degree of the tree
F	one-way function
Ex	modular exponentiation
R	pseudorandom number generation
E	encryption operation
D	decryption operation
K	size of a key in bits
N	size of a notification message
S	size of a secret

Table 3.2: Comparison of the centralized key management mechanisms by communication cost

Scheme/ Mechanism	Secrecy		Secure against coll.	Communication cost		
				Join		Leave
	back	fore		multicast	unicast	
LKH	Yes	Yes	Yes	$(2\log_d(n)-1)K$	$(\log_d(n)+1)K$	$(d\log_d(n)+1)K$
OFT	Yes	Yes	Yes	$(\log_2(n)+1)K$	$(\log_2(n)+1)K$	$2(\log_2(n)+1)K$
SGHSS	Yes	Yes	Yes	$\log_2\lceil n/m\rceil K+S$	$\log_2\lceil n/m\rceil K+\lceil n/m\rceil S$	$\log_2\lceil n/m\rceil S$
SKD	Yes	Yes	Yes	$\log_2(n)N$	$\log_d(n)K$	$(d-1)\log_d(n)K$

Table 3.3: Comparison of the centralized key management mechanisms by storage cost

Scheme/ Mechanism	Storage cost	
	KDC	users
LKH	$\frac{(dn-1)K}{(d-1)}$	$\log_d(n)K$
OFT	$(2n-1)K$	$\log_2(n)K$
SGHSS	$2\lceil n/m\rceil K+nS$	$\log_2(\lceil n/m\rceil+(m-1))K$
SKD	$\frac{(dn-1)K}{(d-1)}$	$\log_d(n)K$

From Table 3.2, we can notice that the most efficient solutions are the SGHSS scheme and the SKD protocol, which combine a key hierarchy with a derivation technique, decreasing the amount of messages sent out by the KDC.

In the SKD protocol, the derivation technique allows the KDC to only transmit the keys that some users cannot derive, along with the information of the affected branch, which has a smaller size than a key.

Table 3.4: Comparison of the centralized key management mechanisms by computational cost

Scheme/ Mechanism	Join		Leave	
	KDC	users	KDC	users
LKH	$\log_d(n)(2E+R)$	$\log_d(n)D$	$\log_d(n)(dE+R)$	$\log_d(n)D$
OFT	$\log_2(n)(2E+2F)+2R$	$\log_2(n)(D+F)$	$\log_2(n)(E+2F)$	$\log_2(n)(D+F)$
SGHSS	$\log_2\lceil n/m\rceil E+\log_2\lceil n/m\rceil Ex$	$\log_2\lceil n/m\rceil D$	$\log_2\lceil n/m\rceil E+\log_2\lceil n/m\rceil Ex$	$\log_2\lceil n/m\rceil Ex$
SKD	$\log_d(n)(E+F)+R$	$\log_d(n)(F)$	$\log_d(n)((d-1)E+F)$	$\frac{\log_d(n)((d-1)D+F)}{d}$

SGHSS offers the best multicast communication cost because the KDC divides the group in to subgroups, where several users share a KEK, which is derived from secrets, decreasing the amount of keys sent out by the KDC.

3.2.2 Decentralized key management architectures

The decentralized key management approaches are designed to reduce the problems associated with centralized schemes, where a single entity is responsible for controlling the whole group. In a decentralized scheme, the group is divided into small subgroups such that there is a controller for each subgroup. In this way, if a controller fails, the whole group is not affected. Depending on how the decentralized scheme is defined, its efficiency can be determined by some aspects [RH03, CS05], such as:

- *Decentralized controller.* Refers to the independence of the subgroup controllers (SGCs). In other words, the SGCs do not depend on a central controller.

- *Key independence.* Refers to the capability that the keys are not compromised with previous keys.

- *Local rekey.* Refers to the capability that the membership changes in a subgroup, be managed by the SGC locally, without affecting other subgroups (*1 affects n* phenomenon).

- *Data transformation.* In the schemes where there are independent TEKs per subgroup, each SGC has to translate (decrypt with a key and encrypt with another

key) the information sent to other groups. Such task could generate a work overload in SGCs.

- *Preservation of backward and forward secrecy.* Refers to the capability of the system to mantain the privacy of the information even when the subgroups topology changes.

3.2.2.1 Iolus

Iolus framework [Mit97] is based on dividing the group into subgroups, which are organized through a hierarchy called *secure distribution tree.*

Each subgroup in the secure distribution tree maintains an independent subgroup TEK in order not to affect other groups when a user joins or leaves the system.

In the Iolus framework, each subgroup is managed by entities called *group security agents* (GSAs). The GSAs are responsible for connecting the subgroups, forming a hierarchy of subgroups. Thus, in a Iolus system there are two kinds of GSAs: the *group security controller*(GSC) and the *group security intermediaries* (GSIs).

The GSC maintains the control in the highest level of the hierarchy, being the root of the tree formed by all the GSIs. Thus, the GSC is responsible for the security of the whole group. The GSIs are reliable servers, authorized by the GSC to function as proxies of the GSC or of its parent GSI. The organization of the subgroup hierarchy is shown in Figure 3.9.

Each GSI forms a bridge between the subgroups, receiving information from its parents and multicasting such information among its child subgroups. For such reason, the GSIs must perform information translations.

In Iolus, a new group is launched when a server is started as a GSC, which is provided with an access control list (ACL) used to either allow or deny the association of users to the system. At the same time, when the group is launched, several entities can begin to function as GSIs.

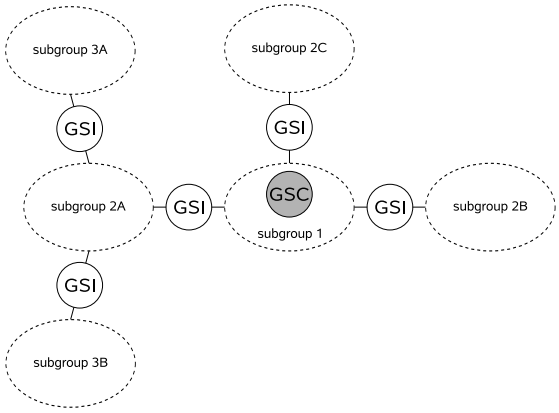

Figure 3.9: Secure distribution tree organization

In order to join the group, users must locate an appropriate GSA and send a join request. As the GSA receives a join request, it assigns an individual key to the user. The GSA uses the user's individual key to transmit the subgroup TEK.

The departure of a user may be for two reasons: the user request for it or a GSA decides to expel the user. In any case, the GSA has to perform a rekey in the subgroup.

3.2.2.2 SAKM

In the Scalable and Adaptative Key Management approach (SAKM) [CBB04], the group is split into clusters of subgroups, based on the dynamism of memberships.

SAKM starts a group with one common TEK, and dynamically splits the group into clusters with different local TEKs. The split aims to minimize the translation cost and the overhead produced by the rekey.

With the SAKM architecture, a group is organized into multiple subgroups, arranged in a tree structure (see Figure 3.10). Each subgroup is controlled by a SAKM Agent, which is responsible for the local key management. A SAKM Agent could be in two possible states: active or passive. An active SAKM Agent uses an independent TEK for its subgroup, and thus has to translate received messages before forwarding them to local members. A passive SAKM Agent uses the same TEK as its parent subgroup and just forwards received messages to local members without translation. Thus, the whole

SAKM Agents state induces a partition of the subgroups into a set of clusters. Each cluster is composed of a set of subgroups that share the same TEK. The cluster's root agent is an active agent and all internal agents are passive. Messages are translated only at the cluster's root.

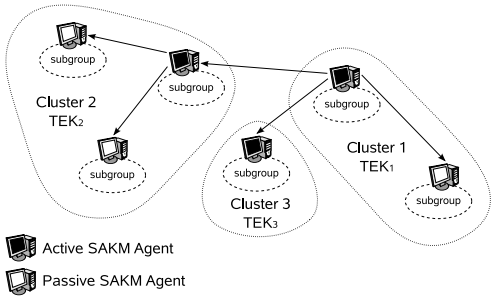

Figure 3.10: SAKM architecture

SAKM follows a probabilistic model to perform an estimation of the arrival of users, and using such estimation, the state of the agents is established. Therefore, periodically, the SAKM agents exchange dynamism information about their subgroups. Based on this information, each agent estimates two costs: the overhead cost induced if the agent becomes active (translation cost) and the cost induced if the agent becomes passive (the *1 affects n* overhead). Comparing the two costs, a SAKM Agent decides whether to become active or passive. If an agent becomes passive, it merges with its parent cluster, so it uses its parent TEK. If an agent becomes active, it forms a new separate cluster, so it uses an independent local TEK.

The SAKM architecture is open, so every SAKM Agent is free to use any rekey strategy. However, two rekey strategies can be considered: a root agent sharing a TEK with all the subgroup users (*n root/leaf pairwise*) or a key hierarchy (LKH).

With the *n root/leaf pairwise* strategy, when a user joins the group, the SAKM Agent multicasts the new TEK to the old members, encrypting the message with the previous TEK, while for the new user, the SAKM Agent unicasts the TEK, encrypted with a key previously agreed between two entities (individual key). If a user leaves the group,

the SAKM Agent unicasts to each member the new TEK encrypted with the respective individual keys.

With the LKH strategy, at a join or leave event, the SAKM updates the keys of the compromised tree branch, including the TEK.

3.2.2.3 Mykil

Mykil [HM03] is a scheme based in Iolus and LKH, so the group is divided into subgroups, called areas, which form a subgroup hierarchy.

In Mykil, each area is organized in a LKH structure, called *auxiliary key tree*, whose root is an entity called Area Controller (AC). The AC is the entity responsible for maintaining the *auxiliary key tree*, so it manages the join and leave events, and all the used keys, including a local TEK, which is independent to the TEKs of other areas.

As an AC can be a member of another area, each area can be linked with another, using the links of its AC, forming a tree of areas (see Figure 3.11).

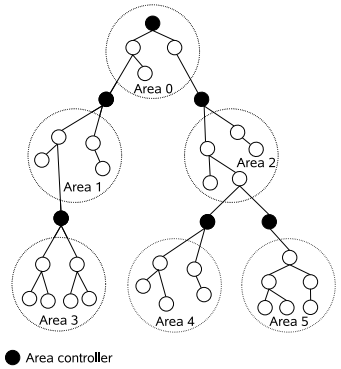

Figure 3.11: Mykil architecture

If a user belonging to an area needs to send data to a user belonging to another area, must encrypt the message with the local TEK, and send it to its AC. The AC must translate the information with the TEK of the destination area.

A group begins when an area is created, ie, when an entity is designed as the root of an *auxiliary key tree*. Thus, the root of the first formed area will be the root of the tree formed by all the group areas.

When a user requires to join the group, it chooses an appropriate AC and unicasts a join request. When the request is received, the AC performs a rekey strategy similar to the LKH scheme.

In Mykil, each AC maintains a balanced tree of 4th degree, so at a join event, the AC must ensure that the tree has empty leaves to store the key of the new member. If the tree is full, the AC must create a new level, splitting the leaf vertex of the most-left position.

For the leave event, as for the join event, Mykil follows a LKH strategy, with the difference that the tree pruning is not performed. As the join process is cheaper when there are empty leaves in the tree, at the user departure, the involved vertex is not removed, just remains as available.

3.2.2.4 Summary

Based on the characteristics listed at the beginning of Section 3.2.2, Table 3.5 gives a comparison of the architectures previously discussed.

Table 3.5: Comparison of the decentralized key management architectures

Architecture	Controllers		Key independence	Local rekey	Data transformation	*1 affects n* phenomenon
	SGCs	KDC				
Iolus	Yes	Yes	Yes	Yes	Yes	No
SAKM	Yes	No	Yes	Yes	Yes	No
Mykil	Yes	Yes	Yes	Yes	Yes	No

As it can be noticed, although decentralized architectures use SGCs, there are some schemes that require a KDC, as is the case of Iolus and Mykil. In such cases, the KDC is the entity responsible for beginning the distribution structure, designating reliable entities to function as subgroup controllers. In these cases, the KDC does not interfere in the rekey operations, as long as subgroup controllers begin to operate.

It is clear that those schemes that use a TEK per subgroup eliminate the *1 affects n* phenomenon, if the rekeying is performed locally and the changes in user organization do not impact the structure of the subgroups.

While managing a TEK per subgroup can eliminate the *1 affects n* phenomenon, this means that the SGCs have to perform data translations, which induce some delay in communications. In this way, SAKM defines some strategies to reduce the effects caused by the data translations.

3.2.3 Distributed key management mechanisms

The distributed key management mechanisms are characterized by having no group controller. Thus, the group key can be generated through the contribution of all members, each contributing their own quota for the calculation of the key.

In a distributed scheme several considerations should be taken, including ensuring that mechanisms for the generation of keys are available to all group members, and also ensuring that the processing time and communication requirements have linearly increased in terms of number of members. Furthermore, a distributed mechanism requires each user to be aware of the group membership list to make sure that the protocols are robust. Some parameters can be used to evaluate the distributed mechanisms [RH03, CS05], such as:

- *Number of rounds.* Refers to the number of iterations among the group members required to generate a key.

- *Number of messages.* Refers to the number of messages, exchanged among the users and which are needed to generate a key. In some cases, the exchanged messages produce unbearable delays that grow along with the group.

- *Processing during setup.* Refers to the computations needed during the setup time. Setting up the group requires most of the computation involved in maintaining the group because all members need to be contacted.

- *DH key.* Identify whether the mechanism uses Diffie-Hellman protocol (DH) [DH76] to generate the keys.

3.2.3.1 GDH

Group Diffie-Hellman exchange (GDH) [STW96] is an extension of the Diffie-Hellman protocol that supports group operations.

With GDH, the group agrees on a pair of primes (q and α) and starts calculating in a distributed way the intermediate values. The first member calculates the first value (α^{x_1}, where x_1 is a random secret generated by the first member) and sends it to the next member. Each subsequent member receiving the set of intermediary values raises them using its own secret number, generating a new set. A set generated by the ith member will have i intermediate values with $i-1$ exponents and a cardinal value containing all exponents.

Thus, the last member can easily calculate the group key k from the cardinal value: $k = \alpha^{x_1 x_2 x_3 \ldots x_n} \mod q$. The last member raises all intermediate values to its secret value and multicasts the whole set to all group members. Upon receiving this message, each group member u_i extracts its respective intermediate value and calculates k by exponentiation of the ith value to x_i . The setup time and the length of messages are linear in terms of the number of group members since all members must contribute to generate the group key.

3.2.3.2 D-CFKM

D-CFKM [WCS+99] is characterized by the use of a Flat Table (FT) instead of a tree structure to organize the keys and the users of a group.

In D-CFKM, both users and keys have a unique ID. The user ID can be derived from its network address, and the bits of such ID determine the keys that the user can access.

Each key is addressed through a *key selector*, which consists of two fields: the *version field* and the *revision field*, arranged as shown in Figure 3.12.

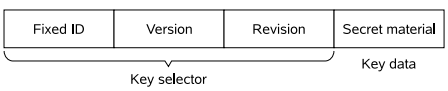

Figure 3.12: Key structure

The *version field* is increased to indicate that a new key has been generated or when a user leaves the group. The *revision field* is increased to indicate that the key must be calculated using a one-way function or when a user joins the group.

The FT has an entry to store the TEK and $2w$ entries to store the KEKs, where w is the number of bits in the IDs of each user. Thus, in the FT, there are two keys for each bit b of the user ID, each one associated with the two values $v \in \{0, 1\}$ that the bit can take. The key associated with bit b having the value v is referred to as $K_{b.v}$. Thus, a user that has the ID 0110 can access the keys $TEK_{0.0}$, $TEK_{1.1}$, $TEK_{2.1}$ and $TEK_{3.0}$. Figure 3.13 shows the assignation of the keys through a FT.

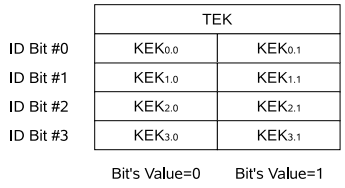

Figure 3.13: Flat ID assignment

The formation of a group starts when the first user creates its own keys (the TEK and the $2w$ KEKs) and begins to emit messages that contain the keys, their identifiers, their version and revision numbers, as well as the address of the creator of such keys. With the transmission of such messages, the first user indicates that it is acting as a *key holder* and that a set of keys agree. Each user involved in the agreement acts as a *key holder* until it receives a message that it has transmitted, indicating that the key has been agreed upon.

In the D-CFKM scheme, each user performs user admission functions and other management functions. When a new user requires to join the group, it waits to receive packets containing information about the geometry of the FT and the address of some users. The new user chooses a user and contacts it to establish a shared secret, and then sends the corresponding TEK and KEKs, using a secure channel. The user that receives the new user must increase the *revision field* in order to notify the remaining users about the joining of the new user and the corresponding rekey.

To expel a user, a *key holder* updates the TEK and the TEKs involved with the removed user. Then, the *key holder* increases the *version field* of the affected keys and multicasts a message to notify that a rekey is needed.

3.2.3.3 TGDH

The tree Group Diffie-Hellman scheme (TGDH) [Per99, KPT00] is characterized by the use of a key binary tree together with the Diffie-Hellman key exchange.

Each vertex in the tree is denoted as $\langle l, v\rangle$, where l refers to the level in the tree and $0 \leq v \leq 2^l - 1$ (see Figure 3.14). Each vertex $\langle l, v\rangle$ is associated with a key $K_{\langle l,v\rangle}$ and a *blinded key* $BK_{\langle l,v\rangle} = f(K_{\langle l,v\rangle})$, where the function $f(\cdot)$ is a modular exponentiation, analogous to the one used in Diffie-Hellman protocol.

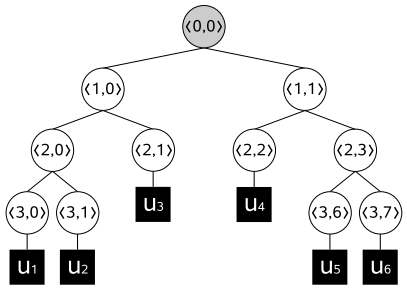

Figure 3.14: Notation for a tree

Each leaf vertex is associated with the user session key $K_{\langle l,v\rangle}$, which is randomly generated. In addition, each user knows all the keys involved in its branch, from $\langle l, v\rangle$ to $\langle 0, 0\rangle$. Each key can be calculated by:

$$\begin{aligned} K_{\langle l,v\rangle} &= (BK_{\langle l+1,2v+1\rangle})^{K_{\langle l+1,2v\rangle}} \mod p \\ &= (BK_{\langle l+1,2v\rangle})^{K_{\langle l+1,2v+1\rangle}} \mod p \\ &= \alpha^{K_{\langle l+1,2v\rangle} K_{\langle l+1,2v+1\rangle}} \mod p \\ &= f(K_{\langle l+1,2v\rangle} K_{\langle l+1,2v+1\rangle}) \end{aligned}$$

Whenever a user membership changes, the remaining users must update the key tree, thereby requiring that the communication among group members be provided with *View Synchrony* [FLS97].

Some group members can purchase the responsibility of generating and broadcasting the keys, when assuming the role of *sponsor*. An *sponsor* will be the user that manages the join and leave events at some point.

When a user needs to join the group, it must send a join request, containing its own blinded key $BK_{\langle 0,0 \rangle}$. When the current members of the group receive the request, the insertion point in the tree must be determined. This will be the right-most vertex, which balances the tree. The *sponsor* will be the user, located in the right-most position in the tree, and will be responsible for creating the new vertex, updating the key tree and computing the new group key, by using the necessary blinded keys. After the *sponsor* computes the new group key, it broadcasts such key.

When a user leaves the group, the *sponsor* will be the user located in the right-most position in the affected branch. At the user leave, each group member must update its key tree, eliminating the vertex of the removed user. For the new formation, the parent vertex of the removed user is replaced with its sibling vertex. Finally, the *sponsor* must compute all the affected keys and then broadcast the updated keys to the remaining group.

3.2.3.4 GAKAP

The GAKAP method [BBB04] is based on the TGDH scheme, proposing the use of a full-balanced key tree and eliminating fusion of the whole tree. In this scheme, the tree management depends on a threshold of the group activity, which keeps the tree full-balanced. This threshold is given by a probabilistic value, which defines the tree model for key management and key distribution.

Group definition is done in three phases. The first phase is performed by an entity named “initial controller” which publishes the opening of a multicast session. After

starting the multicast session for a specific period of time, the initial controller will receive association requests sent by other entities and will create a list of participants. Using such list, the controller will build the first distribution key tree, which is broadcasted among users. The second phase consists of i rounds, where at most $n/2^i$ group members are sponsors (n is the number of participants in the group, $1 \leq i \leq h$, and $h = log_2 n$). The sponsors calculate keys for the tree's branch where they are associated, going from the leaves to the tree's root, broadcasting the intermediate keys in the hierarchical structure. During the last phase, corresponding to round $h + 1$, each member calculates the *group key* and the initial probability of group activity.

If an entity requires to be incorporated into the group, it will broadcast an association request to the members of the group, including its blinded key BK. As in the case of getting the request, each group member defines the place in the tree where the new member will be inserted, which will correspond to the first empty vertex found when crossing the tree from left to right. The position assigned to the vertex associated to the new tree member will be the identification value (ID), as shown in Figure 3.15.

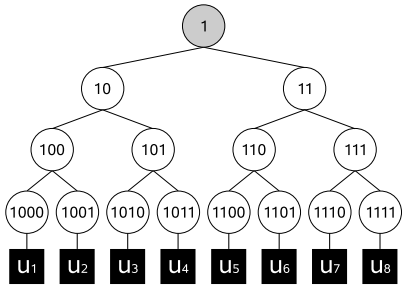

Figure 3.15: Indentification of users and keys in the GAKAP method

The sibling's new member vertex will be the sponsor, updating and broadcasting the compromised BKs. It will also send the new key tree to the new group member. Using the keys sent by the sponsor, all members will be able to calculate the *group key*.

When a member decides to leave the group, it will broadcast a leave request, including its ID. Each group member will update its key tree, eliminating the key corresponding to the leaving member and all BKs from its sponsors path. In addition, the sponsor

will erase all keys in its branch to the tree's root, and it will calculate the required keys, broadcasting the BKs to the remaining users. Using the keys sent by the sponsor, all members will be able to calculate the *group key*.

3.2.3.5 Summary

In this section, three important protocols based on a distributed scheme were presented (G-DH, D-CFKM and TGDH). Also, one of the most recent contributions in this area, the GAKAP method, was discussed. Table 3.7 compares such protocols, based on the criteria discussed at the beginning of Section 3.2.3.

The notation used in Table 3.7 is described in Table 3.6.

Table 3.6: Notation used in Table 3.7

Symbol	Meaning
n	number of users in the group
i	user index
E	encryption operation
D	decryption operation
Ex	modular exponentiation

Table 3.7: Comparison of distributed key management mechanisms

Mechanism/ Method	No. of rounds	No. of messages		DH Key	Setup	
		multicast	unicast		Leader	Others
G-DH	n	n	$n-1$	Yes	n.a.	$(i+1)Ex$
D-CFKM	n	0	$2n-1$	No	$(i-1)E$	iD
TGDH	$\log_2(n)$	$\log_2(n)$	0	Yes	n.a.	$(\log_2(n)+1)Ex$
GAKAP	$\log_2(n)$	$\log_2(n)$	1	Yes	n.a.	$(\log_2(n)+1)Ex$

As can be noticed in Table 3.7, the best communication costs are obtained when combining a Diffie-Hellman exchange-key protocol with a LKH structure. This is true despite the fact that computational costs of each entity are increased.

Also, notice that the measures obtained from the TGDH protocol and the GAKAP method are very similar. However, it is important to point out that the principal difference between them is presented in association events: under the TGDH protocol,

each sponsor transmits the whole tree, while during the GAKAP method, the sponsors transmit only the set of modified keys.

It is clear that the main problem in a distributed scheme is the synchronization that must be achieved among users of a group. The success in this synchronization depends on the fact that key updates do not affect the right transmission in the information.

3.3 Key management for multi-group environments

Due to advancements in communications schemes of the type *many-to-many*, some collaborative environments have been designed where there are entities simultaneously associated to more than one group. Some applications of this type contain heterogeneous data flows or flows with many data layers, which makes mandatory for users to have different privileges for information access. On the other hand, there are applications allowing multi-party collaboration, where users may be involved with more than one work session.

In this type of environments, each group must preserve privacy in its internal information, using different security schemes. An entity involved with several groups must handle several keys, which may create a significant increment in storage cost if a good strategy is not used. On the other hand, the dynamics in the rekey process may create additional difficulties with updates in memberships, due to the fact that user's associations, disassociations or switching among groups may be non-monotonic.

Despite of the previously named difficulties, it is possible to take advantage of the coincidences and overlaps in user memberships to reduce storage and distribution costs, eliminating redundancy in managed keys. Based on this, efficiency in a key management scheme for multi-groups environments may be measured using the following parameters [SL03]:

- *Communication costs.* It refers to the number of bytes used by KDC or secondary servers to distribute keys among users in different groups.

- *Storage requirement.* It refers to the amount of keys used to code information that must be stored by the involved entities in the system.

- *Distribution redundancy.* Depending upon distribution redundancy and the way in which keys and users are organized in the system, membership's overlap will define redundancy level in managed keys. In some schemes, a change in user membership directly affects all system users (*1 affects n*), creating an additional cost in communication and resulting in the need of some users to renew their keys even if they do not require it.

The next sections discuss the main key management schemes for multi-group environments, emphasizing structures and distribution strategies for each case.

3.3.1 Centralized multi-group key management

3.3.1.1 MGKMS

The Multi-group key management scheme (MGKMS) [SL03, SL04] focuses on the transmission of streams with multiple layers or multiple-object types, where different privilege levels are required.

In MGKMS, users are grouped according to privileges or services to which they may have access. There are two possible types of groups: data groups (DG) and service groups (SG). A data group is a set of users that have access to a unique data stream by a multicast session. A service group is a set of users that have access to same resources, that is, the same data or privileges layers, and there are no membership overlaps. Figure 3.16 shows properties of service and data groups.

MGKMS defines a centralized hierarchical architecture, named *integrated key graph*, to organize all user keys and to exploit overlaps among user memberships, in order to improve efficiency in access control. An integrated key graph is made with several trees, following the next three steps:

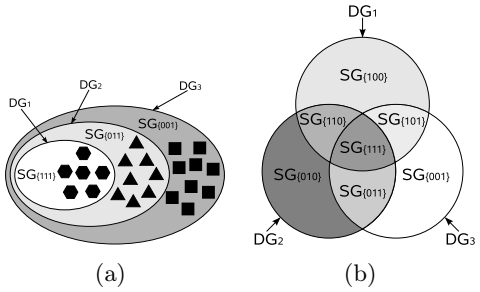

Figure 3.16: Data groups and service groups in (a) multi-layer and (b) multi-object environments

Step 1: For each SG S_i, a sub-tree is built whose leaves vertices are users at S_i, and whose root is associated with a key defined by K_i^S. These sub-trees are called *SG-subtrees*.

Step 2: For each DG D_m, a sub-tree is built, whose root is the key of DG K_m^D and whose leaves are the roots of SG sub-trees holding users with the same resources privileges. These sub-trees are called *DG-subtrees*.

Step 3: A graph is generated by connecting leaves of the *DG-subtrees* and roots of the *SG-subtrees*.

Figure 3.17 presents the three steps for the generation of the integrated key graph.

This multi-group integrated keys graph may be seen as M overlapped trees, each having a key K_m^D as a root and users in DG D_m as leaves. In this structure, the keys for a user in a SGS are the keys of the path defined from it to the roots of *DG-subtrees*.

In addition to association and disassociation, users may switch their memberships, changing their privilege levels. A centralized entity (KDC) will be responsible for updating the engaged keys in any of these operations. For example, if a user u_k with a set of keys ϕ_i moves from the SG-subtree S_i to a new location in the SG-subtree S_j, the KDC performs a rekeying by doing the following:

- first keys are updated in $\bar{\phi_i} \cap \phi_j$ using one-way functions, with some procedure similar to the one defined by the D-CFKM scheme.

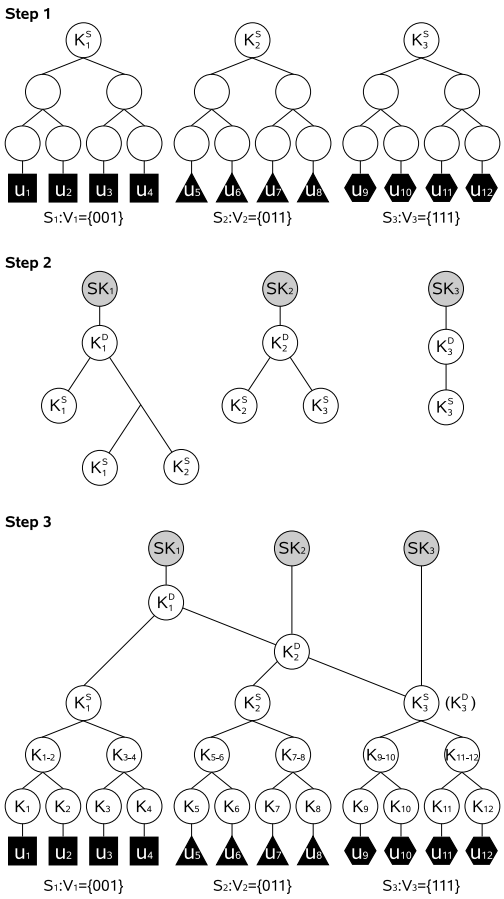

Figure 3.17: Building the integrated key graph

- then, the KDC generates new keys at $\bar{\phi}_i \cap \phi_j$ and distributes them in the graph compromised path, from leaves to root, using children's keys at each level.

Taking as example the structure defined in Figure 3.17, if user u_8 moves from SG S_2 to SG S_1, the vertex K_4 at SG S_1 will be divided to accommodate users u_4 and u_8, generating a new vertex for key K_{4-8} while vertex K_7 at SG S_2 will be up one level and will substitute the vertex of the key K_{7-8}. The KDC will generate keys K'_{3-4} and $K_1^{S'}$ using previous keys and will increment its *revision number* as in the D-CFKM scheme, for users 1, 2, 3 and 4 to calculate new keys using one-way functions. In addition, the KDC will generate keys K'_{4-8}, $K_2^{S'}$, $K_2^{D'}$ and SK'_2 and will distribute them among the corresponding users, using the following messages:

$$\left\{K'_{4-8}\right\}_{K_8}, \left\{K'_{4-8}\right\}_{K_4}, \left\{K_2^{S'}\right\}_{K_{5-6}}, \left\{K_2^{S'}\right\}_{K_7}, \left\{K_2^{D'}\right\}_{K_2^{S'}}, \left\{K_2^{D'}\right\}_{K_3^S}, \left\{SK'_2\right\}_{K_2^{D'}}$$

3.3.1.2 IDHKGS

The ID-Based Hierarchical Key Graph Scheme (IDHKGS) [WOCG07] uses an integrated graph defined in MGKMS for key management. Its difference with respect to the MGKMS is that in the IDHKGS, each key is identified by a unique ID, which is used by each user to deduce the set of keys stored by other users.

The ID of each key consists of a pair of integers, assigned by KDC according to the type of tree where key belongs to, that is, a SG-subtree or DG-subtree. For the SG-subtree, ID of each key is given by pair $\langle i, m\rangle$, where i is a prime number ($i = 2, 3, 5, ...$) which identifies the SG to which the key belongs to, and m ($m \geqq 0$) corresponds to the key position in the SG-subtree. The position of each key in a SG-subtree is enumerated from the root (K_i^S) from up to bottom and left to right, being root ID $\langle i, 0\rangle$.

The SG key's ID's follow this rule: the vertex associated to key $k_{\langle i,\lfloor(m-1)/2\rfloor\rangle}$ is the parent of the vertex associated to key $k_{\langle i,m\rangle}$.

In the section corresponding to the DG-subtree, if a vertex associated to a key K_m^D has two children with associated keys IDs $\langle j_1, n_1\rangle$ and $\langle j_2, n_2\rangle$ respectively, the ID for key K_m^D is pair $\langle j, n\rangle$, where $j = mcm(j_1, j_2)$ and $n = max(j_1, j_2)$. If a vertex associated to a key K_m^D has only one child, its ID is the couple $\langle j_1, -1\rangle$. Figure 3.18 shows key identifications in an integrated graph.

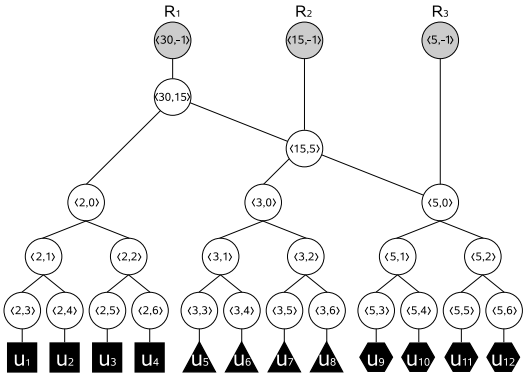

Figure 3.18: Keys identification in a integrated graph

For the case where a user u requests to join the group SG_i, the KDC inserts a vertex associated to u at the end of one of the shortest paths of the SG-subtree. The KDC

assigns ID $\langle i, m\rangle$ to the key associated to user u and broadcasts the ID $\langle i, m\rangle_J$ (where J shows that it is a new user join), and the maximum ID of current keys stored at SG_i, $\langle i, n_k\rangle$. When a user receives $\langle i, m\rangle_J$ and $\langle i, n_k\rangle$, it will be able to derive new and past keys ($k' = f(k)$) using a one-way function. If SG_i is full when u joining takes place, the first leftmost vertex associated to key $k_{\langle i,n\rangle}$ is divided. As a consequence of such division, the previous key identified with $\langle i, n\rangle$, will be identified with ID $\langle i, l\rangle$, making the vertex associated to the new $k'_{\langle i,n\rangle}$ the parent of vertices associated to keys $k_{\langle i,l\rangle}$ and $k_{\langle i,m\rangle}$. The new key $k'_{\langle i,n\rangle}$ will be calculated with the one-way function $k'_{\langle i,n\rangle} = f(k_{\langle i,l\rangle} \oplus k_{\langle i,0\rangle})$.

For the case when a user u leaves group SG_i, all keys in the set ϕ_u are updated. The KDC will broadcast the ID of the key associated to user u, $\langle i, n\rangle_L$ (with L showing that it is the case of a user leave). When the rest of the users get the message, they calculate new keys using the one-way function $k' = f(k \oplus k_1)$, where k_1 will be the first key that is not in the path of the disassociated user, as traveling the graph bottom-up. KDC should only code and send keys being associated to users that cannot calculate keys by themselves.

When a user u moves from SG_i to SG_j, the operation is considered as if the user first leaves SG_i and then joins SG_j. The KDC will broadcast the user key ID departing from SG_i $\langle i, n\rangle_{SL}$ and from the user being added to SG_j $\langle i, m\rangle_{SJ}$ (SJ and SL denote a user switching) as well as the maximum ID from current stored keys at SG_i, $\langle i, n_k\rangle$. After getting the corresponding messages, users of each SG perform operations for rekeying.

3.3.1.3 DACMGS

Dynamic Access Control for Multi-privileged Groups Scheme (DACMGS) [DML04] has been designed to handle multiple groups with high dynamism when being created and decomposed. This is different from schemes based on an integrated key graph, proposed by MGKMS, that are more efficient in environments where the number of groups is fixed and where data streams scales just in one dimension.

In the DACMGS, each service group (SG) makes a sub-tree, where each leaf vertex represents users and each root vertex is associated to a set of TEK's called Access Key set (AK set). Each AK set is a subset of the set of TEKs (TEK set). The sub-tree of each SG will have an extra vertex, a level below the root vertex, which will be associated to a KEK, named service root key (SRK). The rest of the middle vertices will be associated to other KEKs. Figure 3.19 shows the architecture for DACMGS.

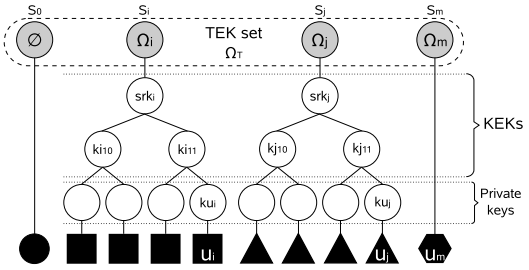

Figure 3.19: DACMGS architecture

Let Ω_T be the set of TEKs. Each SG S_i will be associated to an AK set Ω_i, $\Omega_i \subseteq \Omega_T$. Each user associated to a SG S_i will store a private key, a set of KEKs and an AK set Ω_i.

In the DACMGS, the SG S_0 represents a virtual group, associated to AK set $\Omega_0 = \emptyset$ with no access privileges, to whom all users with no membership will be associated. Therefore, user join is equivalent to a switch from SG S_0 to some SG S_i. Similarly, a disassociation is handled as a shift of a user from SG S_i to SG S_0.

As with the D-CFKM scheme, each TEK and each KEK are referenced by a unique ID, which consists on a *revision field* and a *version field*, with a difference in the fact that the DACMGS scheme uses both fields for rekeying. In this case, the value of the *revision field* is incremented each time that a key is recalculated, while the value of the *version field* is incremented when the the KDC transmits a new secret.

As may be anticipated, both user join and user leave are managed as a switch, therefore in any case, rekeying is done in the same way. Suppose that a user u_i is switched from SG S_i to SG S_j, rekeying consists of four steps:

Step 1: The KDC updates all keys involved in the path affected by the departure of user u_i, in the sub-tree corresponding to SG S_i.

Step 2: The KDC generates a new secret ck_s, updating keys at $\Omega_i \cap \bar{\Omega}_j$, using a one-way function $k' = H_{ck_s}(k)$ and incrementing the value of the *version field* in such keys. Finally, the KDC transmits a rekeying message $\{ck_s\}_s\, rk_l$ to all SGs S_l, including S_i, such that $\Omega_l \cap (\Omega_i \cap \bar{\Omega}) \neq \emptyset$, so that the affected users may be able to derive keys presenting an increment in the *version field*.

Step 3: The KDC selects a right position in sub-tree S_j for user u_i. If the sub-tree is partly full, only one vertex will be added to the structure. If the sub-tree is full, a leaf vertex will be selected to be divided and a new level in the tree will be created. Once the storage in the sub-tree is solved, the KDC will update all keys in the affected path, using past keys and the one-way function $k' = H(k)$, and then sends all keys to user u_i, coding them with the corresponding private key. In order that the rest of users derive new KEKs, the KDC will increment only the value of the *revision field* in each affected key.

Step 4: Similar to step 2, the KDC will update all keys at $\bar{\Omega}_i \cap \Omega_j$ and will increment the value of *revision field* for each key so that the rest of the users may derive new keys.

3.3.1.4 KTR

The Key Tree Reuse scheme (KTR) [GLLC05] is useful in environments where an entity maintains subscriptions simultaneously to several programs (membership to several groups). It is based on two principles: (1) Users associated to several groups can be stored in a key structure with shared keys; (2) Previous keys may be reused to reduce costs in rekeying without compromising the security of the system. In this KTR scheme, a structure named *key forest* is defined, which consists of several Shared-key trees (SKT), where users are allocated according to their memberships. Each tree represents a user

group sharing their membership with one or several groups, and they are composed by KEKs. The root of each tree is connected to the TEKs of existing groups, according to memberships acquired by users represented in each tree. Figure 3.20 shows a tree with shared keys.

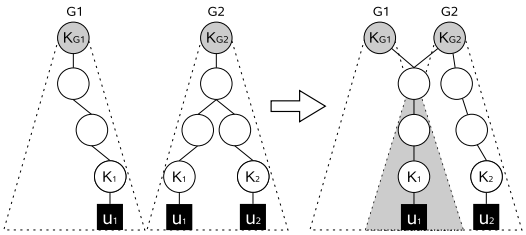

Figure 3.20: A Shared-key tree

In addition to the shared keys, key forests contain a multi-level structure, called *root graph*, made by several vertices acting as mediators among vertices corresponding to TEKs and tree roots of trees with shared keys. Vertices in root graph are used to avoid that, facing the departure of a user related to a TEK and consequently its update. KEKs related to all users associated to the n trees connected with such TEK may have to be updated. In this way, each TEK will be related to only two intermediate vertices or two trees. Similarly, each vertex in the root graph will have at most two links, increasing the number of intermediate vertices as the number of trees associated to a TEK is incremented. Figure 4.1 shows the structure of a key forest.

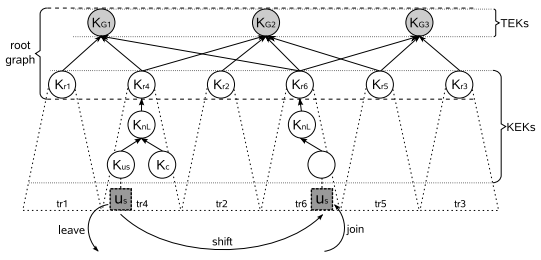

Figure 3.21: Key forest

The KTR scheme considers three activities where it is mandatory to perform rekeying: join, leave and switch.

When a user leaves the system, the server will perform rekeying in the affected trees, following the LKH scheme.

For join and shwitch, the KTR proposes a key reuse scheme, which consists on updating only such keys that may compromise the security of the system. Such keys, called critical keys, are those located in a path traced from some tree to a TEK and able to break the backward secrecy. The server will decide when a key is critical, based on the lifetime of that key.When a join or switch event occurs, the server will select the best path to store the user. Such path must contain the least number of critical keys in order to be able to perform an efficient rekeying and to reuse the highest number of keys.

3.3.2 Decentralized multi-group key management

3.3.2.1 DKMS

The Distributed Key Management Scheme (DKMS) [RLK05] is a solution designed to solve the problem associated to multi-privileges groups facing problems associated to schemes based on integrated key graph, defined in the MGKMS scheme: decrement in the complexity associated with the construction of the integrated graph and elimination in redundancy related to rekeying operations.

In the DKMS, each SG is managed by a SG server, which will keep a key tree to manage SKs related to users of a SG.

The DKMS structure is composed of two parts: the DG, which contains all SG servers, and the SG, which includes all users associated to each SG. Figure 3.22 shows the structure proposed by the DKMS scheme.

The DKMS structure is built by three steps:

Step 1: The DG part is built; a SG server group (SGSG) is built, containing all SG servers. In this phase, each SG server is related to one SG key K_i^S.

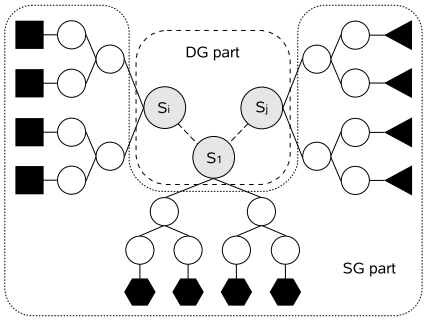

Figure 3.22: DKMS structure

Step 2: The SG part is built; for each SG S_i, a SG-sub-tree is built, whose root vertex will be a SG key K_i^S and leaves vertices will be associated to users related to SG S_i.

Step 3: The SG and DG part are combined, connecting SG keys to roots in each SG-sub-tree.

The DKMS scheme handles user switch as the departure of a user from a SG, and the joining of that user to another SG; therefore, only such operations are taken for rekeying.

During the joining of a new user u_k to a SG_i, the SG server S_i will select a position in the SG-sub-tree, suitable to house a new user, and it will update keys in the affected path, using one-way functions. Finally, the SG server S_i will send rekeying messages to users related to the affected path.

When the user u_k leaves a SG_i, the SG server S_i will update the SKs in the affected path in the SG-sub-tree. Then, the SG server S_i will exchange new SKs with some of the rest of the SG servers, coding them with the SGSG key. Finally, it will multicast the SK needed into the SG.

3.3.3 Summary

In this section some key management schemes for multi-group environments were presented. As one can notice, overlaps among user's memberships can be used to improve

the efficiency in distribution and storage. Table 3.9 compares the presented schemes with respect to such factors.

As commented in Section 3.3, the way in which membership overlaps are handled may create redundancy in key distribution, resulting in the fact that changes in a user membership affect all users. Table 3.10 shows schemes presenting this problem.

The notation used in Table 3.9 is described in Table 3.8. For the schemes presented in this section, the organization of keys and users is done by structures using two levels. In the first level, the users are organized according to their privileges or subscriptions (SG trees and trees of shared keys). In the second level, memberships are organized (DG tree and Root Graph). In order to make a fair comparison, in the notation shown in Table 3.9, the total of structures at first level is named $m(s)$, and the number of users in each structure in such level is named n_t. The second level structure is named common graph (CG). The server responsible for managing the keys of the system has to send all the keys in the CG, therefore, the number of keys sent by the responsible server is denoted as N_K.

Table 3.8: Notation used in Table 3.9 and Table 3.10

Symbol	Meaning
n	total number of users in the whole system
n_t	number of users grouped in a tree
s	number of resources or programs in the system
$m(s)$	number of service or program groups
$m(s_r)$	number of service or program groups related to a user
$\|\Omega_t\|$	number of keys related to a group
N_K	number of keys sent by a KDC to the $m(s_r)$ involved groups
K	key size
S_{sk}	size of revision value descriptor of a secret

In the case of the solutions based on the Integrated Key Graph , N_K is approximately two times the number of resources that the user can access ($N_k \approx 2|\Omega_t|$). In the case of DACMGS, DKMS and KTR, N_K is the number of keys sent by the KDC to the $m(s_r)$ involved groups, $N_K = \sum_{l=1}^{m(s_r)} |\Omega_t \cap \Omega_l|$, where $\Omega_t \cap \Omega_l$ is the number of keys shared by two groups.

Table 3.9: Comparison of key management mechanisms for multi-group environments by communication cost

Scheme	Communication cost				
	join		switch		leave
	multicast	unicast	multicast	unicast	
MGKMS	S_{sk}	$(\log_d(n_t)+N_K+1)K$	$(d\log_d(n_t)+N_K)K$	$(\log_d(n_t)+N_K+1)K$	$(d\log_d(n_t)+N_K)K$
IDHGKS	S_{sk}	$(\log_d(n_t)+N_K+1)K$	$((d-1)\log_d(n_t)+N_K)K$	$(\log_d(n_t)+N_K+1)K$	$((d-1)\log_d(n_t)+N_K)K$
DACMGS	0	$(\log_d(n_t)+1)K$	$d(\log_d(n_t)-1)K+N_KS_{ck}$	$(\log_d(n_t)+1)K$	$(d\log_d(n_t)-1)K+N_KS_{ck}$
DKMS	$d\log_d(n_t)+m(s_r)$	$(\log_d(n_t)+\lvert\Omega_t\rvert)K$	$(d\log_d(n_t)+m(s_r))K$	$(\log_d(n_t)+\lvert\Omega_t\rvert)K$	$(d\log_d(n_t)+m(s_r))K$
KTR	$2Km(s_r)m(s)$	$(\log_2(n_t)+\log_d(m(s_r))+N_K)K$	$2Km(s_r)m(s)$	$(\log_2(n_t)+\log_2(m(s_r))+N_K)K$	$2Km(s_r)m(s)$

Table 3.10: Comparison of key management mechanisms for multi-group environments by storage cost

Scheme	Storage cost			*1 affects n*
	KDC	SGSs	users	
MGKMS	$(\frac{d}{d-1}n+2s)K$	n.a.	$(\log_d(n_t)+N_K+1)K$	Yes
IDHGKS	$(\frac{d}{d-1}n+2s)K$	n.a.	$(\log_d(n_t)+N_K+1)K$	Yes
DACMGS	$(\frac{d}{d-1}n+s)K$	n.a.	$(\log_d(n_t)+\lvert\Omega_t\rvert)K$	No
DKMS	n.a.	$(\frac{d}{d-1}n_t+\lvert\Omega_t\rvert+1)K$	$(\log_d(n_t)+\lvert\Omega_t\rvert)K$	No
KTR	$(2(n+m(s)))K$	n.a.	$(\log_2(n_t)+\log_2(m(s_r))+N_K)K$	Yes

Considering schemes where a unique hierarchy is defined to manage keys, Table 3.9 shows that best costs are obtained by IDHGKS, which is based on an integrated key graph, defined by the MGKMS scheme. The main limitation in the mechanisms based on integrated key graph is the redundancy in distribution; the change in a membership of a user located at the lowest level in the DG sub-tree will affect users associated to the SG sub-trees at upper levels. Another problem found in this kind of schemes is related to the complexity in structure maintenance. In spite of that, this kind of schemes is very efficient in environments where the number of groups is fixed and data flows are scalable in only one dimension.

With respect to schemes based on multiple-key hierarchies, both DACMGS and DKMS solve the redundancy problem found in key distribution, appearing in schemes based on integrated key graph. These schemes obtain good results for storage and communication. It must be noted that the DKMS scheme is able to solve this problem using a decentralized scheme, making each service group be maintained by a KDC.

The KTR scheme is another important contribution, even though it does not solve the problem related to the *1 affects n* phenomenon. The KTR offers an adequate infrastructure for environments where memberships are related not only to access privileges for data streams, but also to applications or subscriptions of programs.

Chapter 4

Proposed mechanisms for multi-session key management

According to the proposed methodology, we present two mechanisms for key management in multi-session environments. The first mechanism is based on a centralized scheme; the reason is that this scheme has less restrictions than others, and therefore is suitable to design the main features of a key manager: key generation and key distribution. With the centralized mechanism, we solve the problem of the key distribution redundancy, which is the main characteristic of a multi-session environment. The second mechanism is an extension of the first mechanism and is based on a decentralized scheme. With the decentralized mechanism, we solve the problems related to a single point of failure, the scalability and the number of supported sessions of a system.

4.1 Centralized mechanism (MM-MSKMS)

We propose a mechanism named Multimedia Multi-session Key Management Scheme (MM-MSKMS). The proposed scheme uses a key forest structure, similar to the Key Management Graph defined in the Dynamic Access Control for Multi-privileged Groups Scheme (DACMGS) [DML04]. MM-MSKMS differs from the DACMGS in three main aspects: first, the key forest is combined with a key derivation technique to reduce the rekeying overhead; second, the users are enabled to transmit multimedia information generating independent keys for each packet, by using a pseudorandom number generator; and finally, the users can exchange streams among them in an n to n communication.

The fact that MM-MSKMS uses one key per packet allows the system to support the delay, the loss and the transposition of packets, since each packet is completely independent of the others, avoiding the need for the decryption to be done in a specific order.

4.1.1 Architecture

In the MM-MSKMS, the KDC maintains a key forest to organize the joined users according to their membership. The key forest will be composed by different key trees, each one associated with a group of users who have an exact match on their memberships. In other words, each tree represents a group of users who have a full overlapping in their sessions. Each group of users with overlapped sessions is named as *Overlapping Group* (OG). In order to use a general notation, even those groups where users are involved with a single session are called OG.

In a system where there are s sessions, there will be at most $m(s)$ Overlapping Groups, where $m(s)$ is determined by:

$$m(s) = \sum_{r=1}^{s} \left(\begin{array}{c} s \\ r \end{array} \right) \tag{4.1}$$

Thus, the key forest is formed by m_0 trees, with $1 \leq m_0 \leq m(s)$ trees (see Figure 4.1). Each tree is formed by two kinds of keys: the Key Encryption Keys (KEKs) and the Session Keys (SKs). KEKs are used as auxiliary keys for the rekeying operations. SKs are used to generate independent Data Encryption Keys (DEKs), which are used to encrypt and decrypt information related to sessions. In a system with s sessions, there are s SKs, one for each session.

KEKs are organized in balanced trees, where each key is denoted as $K^t_{i,j}$, where t indicates the OG associated with the tree ($1 \leq t \leq m_0$), i indicates the vertex level, and j indicates the most left position relative to the level i. KEKs located in the $K^t_{1,1}$

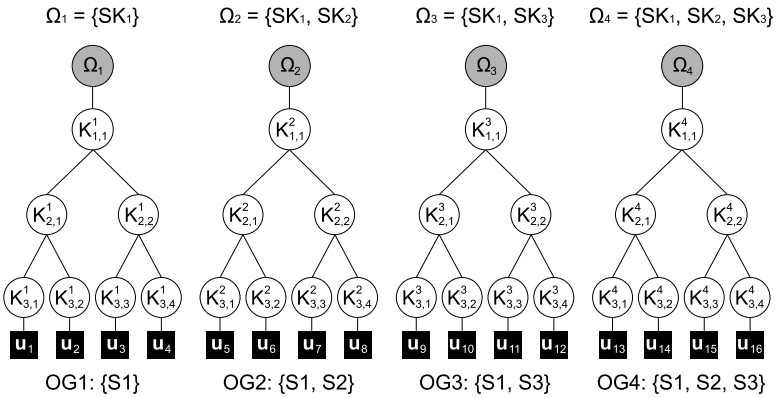

Figure 4.1: Key forest for 4 OGs related to three sessions

position are called root-KEKs (rKEKs) and are used to distribute the SKs, while KEKs in the lowest level are the individual keys of the OG members.

All the SKs associated with the sessions related to an OG_t form a set Ω_t, which is represented by an additional vertex in a level above of the KEK-tree.

With such key organization, each OG member must store all the keys along the tree path where it is joined, from its individual key to the rKEK, along with the SKs in the corresponding Ω_t. Unlike KEKs and SKs, DEKs are not stored by users because they are generated before the transmission of a packet.

An example of a key forest structure used to organize different OGs is shown in Figure 4.1. In such example, it is assumed that in the system there are 16 users grouped according to their memberships into 4 of the 7 possible OGs, related to three different sessions. Thus, the members of OG_1 have access to information of session S_1, using SK_1; the members of OG_2 have access to the information of sessions S_1 and S_2, using SK_1 and SK_2, the members of OG_3 have access to information of sessions S_1 and S_3, using SK_1 and SK_3, while the members of OG_4 have access to information of sessions S_1, S_2 and S_3, using SK_1, SK_2 and SK_3.

4.1.2 Key generation

As we mentioned in section 4.1.1, members in each OG store two kinds of keys: KEKs and SKs. KEKs are keys generated through derivation technique to avoid having the

KDC generate, encrypt and transmit all the keys related to the rekeying. SKs are keys designed to allow the users to generate independent DEKs and transmit multimedia packets, using one key per packet.

4.1.2.1 KEKs generation

For the generation of KEKs, our mechanism uses a key derivation technique similar to the one defined in the SKD protocol [LLL05, LHLL09]. With this technique, each user can compute the new KEKs using a function $f(\cdot)$ and previous keys. Thus, the KDC only has to transmit the keys which some users cannot derive, decreasing the computational effort in the KDC and the use of bandwidth. For key derivation, function $f(\cdot)$ could be a one-way function, a pseudo-random number generator or a trap-door function.

4.1.2.2 SKs generation

Each SK_h $(1 \leq h \leq s)$ is a packet formed by variables used by the Blum Blum Shub algorithm (BBS) [BBS86]. The BBS algorithm has the purpose of generating a pseudo-random number series free of patterns that can be discovered with any reasonable amount of calculations. With some bits of each of the generated numbers, issuers construct an individual DEK to encrypt a packet. Receptors can recover any DEK generating the corresponding number series from the packet index and a *seed*. Thus, users can use individual keys to encrypt each transmitted packet.

With the BBS algorithm, each number is generated by:

$$x_{k+1} = (x_k)^2 \mod b \tag{4.2}$$

where $b = pq$, being p and q two large primes congruent with 3 mod 4 ($p \equiv 3 \mod 4$ and $q \equiv 3 \mod 4$).

To start the number generation, a *seed* must be chosen; such seed can be a random number x_0 that is a relative prime with b. Knowing x_0, any user can compute the kth generated number using the equation:

$$x_k = x_0^{(2^k \mod ((p-1)(q-1)))} \mod b \tag{4.3}$$

Thus, each key SK_h will be a packet formed by the variables p_h, q_h, x_{0_h} and b_h, which will be computed by the KDC.

4.1.2.3 DEKs generation

Generating independent DEKs. When a user starts the transmission of multimedia information, for each packet P_r, that user locally generates an independent DEK in the following way:

1: Using the variables of $SK_h = \{p_h, q_h, x_{0_h}, b_h\}$ and equation (4.2), the user generates a pseudo random number series of k elements, depending upon the required key size (for example, if the system uses AES-128, then one series of 64 elements should be generated)

2: At most $log_2 log_2 b_h$ bits of each generated number are taken to form k bit sequences

3: DEK is formed concatenating the k bit sequences

Finally, each packet is encrypted with the generated DEK, using the specified cypher algorithm.

Recovering independent DEKs. When a user receives a packet P_r, that user will use the packet index to recover the DEK to decrypt information in the following way:

1: Using the variables of $SK_h = \{p_h, q_h, x_{0_h}, b_h\}$ and equation (4.3), the user locally generates a pseudo random number series of k elements, starting at element $(r-1)k+1$

2: At most, $log_2 log_2 b_h$ bits of each generated number are taken to form k bit sequences

3: The corresponding DEK is created concatenating the k bit sequences

Finally, each packet is decrypted with the DEK, using the specified cypher algorithm.

4.1.3 Rekey operations

Rekeying operations must be started by the KDC when a membership change takes place. We understand as a membership change when a user joins an OG or leaves any OG in order to leave the whole system or simply to change its OGs and its joined sessions.

4.1.3.1 User join

When a user requests to join the system, the KDC decides which OG must hold that user, according to its requested sessions. Then, the KDC randomly generates an individual key for the new member and sends it through a secure channel. Moreover, the KDC updates the compromised KEKs and SKs.

Updating KEKs. The KDC assigns a new vertex in the KEK-tree to store the individual key of the new member. As each KEK-tree maintained by the KDC must be balanced, each new vertex is inserted in the shortest paths of the KEK-tree.

Assuming that K^t_{v,j_v}, the root vertex of KEK subtree X^t_{v,j_v} is the last internal vertex on the joined path:

- if X^t_{v,j_v} is not full, the new vertex $K'^t_{v+1,j_{v+1}}$ is inserted
- if X^t_{v,j_v} is full, the left most vertex $K^t_{v+1,j_{v+1}}$ is moved to a lower level, becoming the new vertex $K'^t_{v+2,j_{v+2}}$ and its old position is replaced by a new intermediate vertex $K'^t_{v+1,j_{v+1}}$. Thus $K'^t_{v+1,j_{v+1}}$ will be the parent of $K'^t_{v+2,j_{v+2}}$ and the vertex associated with the individual key of the new user, $K^t_{v+2,j_{v+2}+1}$.

In both cases, all the new KEKs, found in unchanged vertices between the vertex asociated with the key of the new user and $K^t_{1,1}$, are computed by:

$$K'^t_{i,j_i} = f(K^t_{i,j_i}) \tag{4.4}$$

where K^t_{i,j_i} is the previous KEK of that position, named as derivation key.

If the new intermediate vertex $K'^{t}_{v+1,j_{v+1}}$ is inserted, the new KEK is computed by:

$$K'^{t}_{v+1,j_{v+1}} = f(K'^{t}_{v+2,j_{v+2}} \oplus K^{t}_{1,1}) \tag{4.5}$$

where $K'^{t}_{v+2,j_{v+2}}$, the previous KEK $K^{t}_{v+1,j_{v+1}}$ is the derivation key, while the rKEK $K^{t}_{1,1}$, named the *salt* value, is used to ensure that the derived key is different even when the same derivation key is used since $K^{t}_{1,1}$ will be different each time.

For the remaining OG members, the KDC multicasts a message to inform the position of the new user. Thus, each user can compute the necessary KEKs. As the new user does not know the KEKs involved with its path, the KDC sends to it a unicast message with the related keys.

Consider the OG_2 of the system shown in Figure 4.1. Suppose that user u_{17} joins the system. The KDC moves the vertex associated with the individual key of user u_5 to a lower level, and replaces that position with the new vertex $K'^{2}_{3,1}$. With this modification, the new intermediate vertex is the parent of $K^{2}_{4,1}$ and $K^{2}_{4,2}$ (see Figure 4.2), where $K^{2}_{4,2}$ is the individual key of the new user.

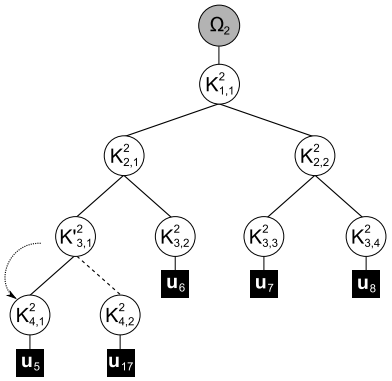

Figure 4.2: An example of user join

The compromised KEKs are recomputed by:

$$K'^{2}_{1,1} = f(K^{2}_{1,1})$$
$$K'^{2}_{2,1} = f(K^{2}_{2,1})$$
$$K'^{2}_{3,1} = f(K^{2}_{4,1} \oplus K^{2}_{1,1})$$

The new user u_{17} cannot derive the KEKs related to its path. For this reason, the KDC unicasts such keys in the following way:

$$KDC \rightarrow u_{17} : \{K'^{2}_{1,1}\}_{K'^{2}_{2,1}} \| \{K'^{2}_{2,1}\}_{K'^{2}_{3,1}} \| \{K'^{2}_{3,1}\}_{K^{2}_{4,2}}$$

Updating SKs. Finishing the updating of the corresponding KEKs, the KDC updates the SKs of the set Ω_t to preserve the backward secrecy.

To generate each SK_h, the KDC generates the necessary variables for the BBS algorithm: two primes p_h and q_h, congruent with 3 mod 4 and one number x_{0_h}, relative prime with $b_h = p_h q_h$, which will be the seed of the BBS generator. Thus, the new key SK_h is the packet $\{p_h, q_h, x_{0_h}, b_h\}$.

To finish the rekeying, the KDC multicasts the new SKs to all the involved OGs, encrypting each packet with the corresponding rKEKs.

In the example shown in Figure 4.2, as OG_2 members are involved in sessions S_1 and S_2, the KDC has to generate the new SK_1 and SK_2, which are elements of Ω_2. Finally, the KDC multicasts the new SKs to the OGs involved with these keys, using the corresponding rKEKs to encrypt those messages. The KDC transmits the new SKs

through the following messages:

$$KDC \rightarrow OG_1 : \{SK'_1\}_{K^1_{1,1}}$$

$$KDC \rightarrow OG_2 : \{SK'_1\}_{K^2_{1,1}} \| \{SK'_2\}_{K^2_{1,1}}$$

$$KDC \rightarrow OG_3 : \{SK'_1\}_{K^3_{1,1}}$$

$$KDC \rightarrow OG_4 : \{SK'_1\}_{K^4_{1,1}} \| \{SK'_2\}_{K^4_{1,1}}$$

The rekeying for the user join process is detailed in Algorithms 1 and 2.

Algorithm 1 User join algorithm on KDC's side

Input: join_request_message($user, \Theta$) /*Θ *is a set with the requested sessions**/
Output: Updated Keys
Ω_{new_user} =get_related_SKs_with(Θ)
t = choose_an_OG_where($\Omega_t = \Omega_{new\_user}$)
$user\_key$ = generate_key()
unicast($user\_key, user$)
$heigh$ = get_heigh_of(t)
(i, j) = get_last_iternal_vertex(t)
if subtree(i, j) is not full **then** /**verifies if the last internal vertex can hold a new vertex**/
    (i, j) = get_right_most_leaf($t, heigh + 1$)
    $K^t_{i,j+1} = user\_key$
else
    if $j < d^{heigh-1}$ **then** /**insert a new vertex under the next available vertex**/
        (i, j) = get_left_most_leaf($t, heigh$)
    else /**create a new KEK-tree level**/
        (i, j) = get_left_most_leaf($t, heigh + 1$)
    end if
    /**new intermediate key derivation**/
    $K^t_{i+1,d(j-1)+1} = K^t_{i,j}$
    $K^t_{i,j} = f(K^t_{i,j} \oplus K^t_{1,1})$
    $K^t_{i+1,d(j-1)+2} = user\_key$
end if
multicast(join_notification$(i + 1, d(j - 1) + 2), OG_t$)
$i = i - 1$
while $i > 0$ **do** /**update of the compromised KEKs**/
    $j = \lceil j/d \rceil$
    $K^t_{i,j} = f(K^t_{i,j})$
    $i = i - 1$
end while
unicast(updated_KEKs(), $user$)
for each $SK_h \in \Omega_t$ **do** /**update of the compromised SKs**/
    (p, q) =generate_two_primes_congruent_with(3 mod 4)
    x = generate_a_relative_prime_with($b = pq$)
    $SK_h = \{p, q, x, b\}$
end for
for each $\beta_g \in \{OG_l | \Omega_l \cap \Omega_t \neq \emptyset \wedge l \in [1, m_0]\}$ **do**
    multicast($\{\Omega_g \cap \Omega_t\}_{K^g_{1,1}}, \beta_g$)
end for

Algorithm 2 User join algorithm on user's side

Input: join_notification(i, j)
Output: Updated Keys
$x = i - 1$
$y = \lceil j/d \rceil$
*/*verifies if a new intermediate vertex has been inserted*/*
if local_user_individual_key()= $K^t_{x,y}$ **then**
    */*new intermediate key derivation*/*
    $K^t_{x+1,d(j-1)+1}$ =local_user_individual_key()
    $K^t_{x,y} = f(K^t_{x,y} \oplus K^t_{1,1})$
    $i = i - 1$
    $j = \lceil j/d \rceil$
end if
$i = i - 1$
while $i > 0$ **do** */*update of the compromised KEKs*/*
    $j = \lceil j/d \rceil$
    if local_user_holds($\{K^t_{i,j}\}$) **then**
        $K^t_{i,j} = f(K^t_{i,j})$
    end if
    $i = i - 1$
end while
*/*update of the compromised SKs*/*
wait_unitl_the_reception_of($\{\Omega_t\}_{K^t_{1,1}}$)
decrypt($\{\Omega_t\}_{K^t_{1,1}}$)

4.1.3.2 User leave

Updating KEKs. When a user leaves the system, the KDC removes the corresponding vertex in the KEK-tree of the affected OG and updates the compromised keys.

Assuming that K^t_{v,j_v} is the root vertex of the affected KEK subtree X^t_{v,j_v},the KEKs updating is performed in one of two ways:

- if K^t_{v,j_v} has at least two children, K^t_{v,j_v} is only updated
- if K^t_{v,j_v} has only a child, K^t_{v,j_v} is replaced by its child ($K'^t_{v,j_v} = K^t_{v+1,j_{v+1}}$)

In both cases, the new KEKs K'^t_{i,j_i} of the compromised path are computed using the previous keys K^t_{i,j_i} along with the left most key of the lower level $i+1$, located in the opposite path of the removed vertex, as follows:

$$K'^t_{i,j_i} = f(K^t_{i+1,j_{i+1}} \oplus K^t_{i,j_i}) \tag{4.6}$$

where $K^t_{i+1,j_{i+1}}$ is the derivation key and the previous key K^t_{i,j_i} is used as *salt* value.

For the remaining OG members, the KDC multicasts a message to inform the position of the removed user. Thus, each user can start the rekeying.

As the derivation strategy only benefits users in the opposite path of the removed vertex, the KDC has to send the updated KEKs to users that cannot derive those keys.

Consider the OG_2 of Figure 4.2. Assuming that the user u_{17} leaves the system, the KDC modifies the KEK-tree, moving the vertex $K^2_{4,1}$ to an upper level, replacing the vertex $K^2_{3,1}$ as shown in Figure 4.3.

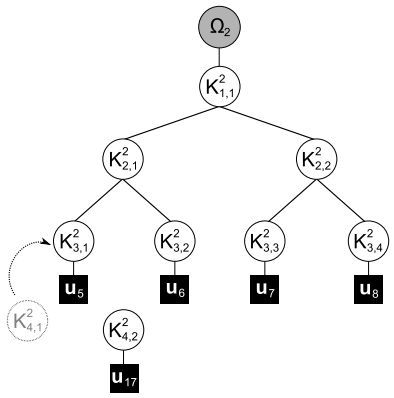

Figure 4.3: An example of user leave

The new KEKs are computed by:

$$K'^2_{1,1} = f(K^2_{2,2} \oplus K^2_{1,1})$$
$$K'^2_{2,1} = f(K^2_{3,2} \oplus K^2_{2,1})$$

Since not all users can derive the keys, the KDC sends the following messages to complete the updating process of KEKs:

$$KDC \rightarrow u_5 : \{K'^2_{2,1}\}_{K^2_{3,1}}$$
$$KDC \rightarrow u_5 - u_6 : \{K'^2_{1,1}\}_{K'^2_{2,1}}$$

Updating SKs. To finish the rekeying, the KDC updates the SKs of the set Ω_t to preserve the forward secrecy. For each $SK_h \in \Omega_t$, the KDC computes the corresponding values p_h, q_h, x_{0_h} and b_h, and then transmits the new SKs to all the involved OGs, encrypting each packet with the corresponding rKEKs.

In the example shown in Figure 4.3, to finish the rekeying, the KDC updates the SKs of the set Ω_2, and multicasts those keys to the members in the affected OGs through the following messages:

$$KDC \rightarrow OG_1 : \{SK_1'\}_{K_{1,1}^1}$$
$$KDC \rightarrow OG_2 : \{SK_1'\}_{K_{1,1}^2} \| \{SK_2'\}_{K_{1,1}^2}$$
$$KDC \rightarrow OG_3 : \{SK_1'\}_{K_{1,1}^3}$$
$$KDC \rightarrow OG_4 : \{SK_1'\}_{K_{1,1}^4} \| \{SK_2'\}_{K_{1,1}^4}$$

The rekeying for the user leave process is detailed in Algorithms 3 and 4.

4.1.3.3 User switch

When a user requires to leave an OG_y to join an OG_z, the KDC has to modify the KEK-trees of the affected OGs and update the compromised SKs.

The updating of KEKs is performed as described in Sections 4.1.3.1 and 4.1.3.2. For the KEK-tree of OG_y, the operations related to the user leave event will be performed, while for the KEK-tree of OG_z, the process will be similar to the user join event, with the only difference being that the KDC does not assign a new individual key to the user. The KDC only modifies the user key index in order to incorporate it into the new KEK-tree.

To finish the rekeying, the KDC updates the SKs that sets Ω_y and Ω_z do not have in common. In other words, the KDC updates the SKs in $\Omega_y \triangle \Omega_z$. The renewal of the SKs in the symmetric difference of Ω_y and Ω_z, is intended to ensure backward and

Algorithm 3 User leave algorithm on KDC's side

Input: leave_request_message(OG_t, i, j)
Output: Updated Keys
multicast(leave_notification(i, j), OG_t)
delete_vertex(i, j, t)
if number_of_children_of($K^t_{i-1,\lceil j/d \rceil}$) $= 1$ **then**
    $K^t_{i-1,\lceil j/d \rceil} = K^t_{i,j}$ /*move the key to a upper level*/
    $i = i - 1$
    $j = \lceil j/d \rceil$
end if
$y = i$
while $i > 1$ **do** /*update of the compromised KEKs*/
    (h, v) =get_left_most_sibling_of(i, j, t)
    $K^t_{i-1,\lceil j/d \rceil} = f(K^t_{h,v} \oplus K^t_{i-1,\lceil j/d \rceil})$ /*KEKs derivation*/
    $i = i - 1$
    $j = \lceil j/d \rceil$
end while
multicast(updated_KEKs(), users_that_cannot_derive())
for each $SK_h \in \Omega_t$ **do** /*update of the compromised SKs*/
    (p, q) =generate_two_primes_congruent_with(3 mod 4)
    x = generate_a_relative_prime_with($b = pq$)
    $SK_h = \{p, q, x, b\}$
end for
for each $\beta_g \in \{OG_l | \Omega_l \cap \Omega_t \neq \emptyset \wedge l \in [1, m_0]\}$ **do**
    multicast($\{\Omega_g \cap \Omega_t\}_{K^g_{1,1}}, \beta_g$)
end for

Algorithm 4 User leave algorithm on user's side

Input: leave_notification(i, j)
Output: Updated Keys
    $i = i - 1$
    $j = \lceil j/d \rceil$
while $i > 1$ **do** /*update of the compromised SKs*/
    (h, v) =get_left_most_sibling_of(i, j)
    if local_user_ holds($\{K^t_{i-1,\lceil j/d \rceil}, K^t_{h,v}\}$) **then**
        $K^t_{i-1,\lceil j/d \rceil} = f(K^t_{h,v} \oplus K^t_{i-1,\lceil j/d \rceil})$ /*KEKs derivation*/
    end if
    $i = i - 1$
    $j = \lceil j/d \rceil$
end while
/*update of the compromised SKs*/
wait_unitl_the_reception_of($\{\Omega_t\}_{K^t_{1,1}}$)
decrypt($\{\Omega_t\}_{K^t_{1,1}}$)

forward secrecy in each of the system's OGs, using the common SKs in order not to raise the rekeying overhead.

After the KDC computes the new SKs, those keys will be sent to all the memebers of the OGs involved with the SKs.

In order to exemplify the user switch process, consider the OG_2 and the OG_4 in Figure 4.1. Assuming that user u_{15} leaves the OG_4 in order to join the OG_2, first, the KDC modifies the KEK-tree of OG_4, removing the corresponding vertex of the individual user key. Then, the KDC modifies the KEK-tree of the OG_2 in order to asign a new vertex for the individual key of u_{15}. We illustrate this process in Figure 4.4.

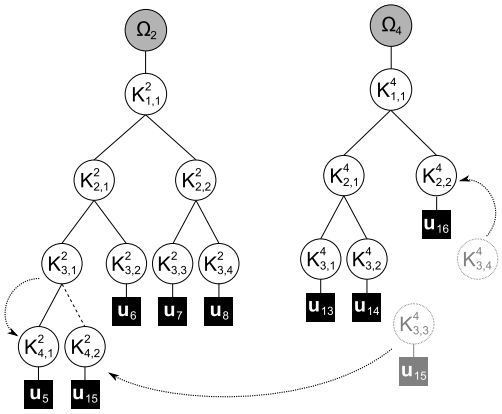

Figure 4.4: An example of user switch

Compromised KEKs are computed by:

$$
\begin{aligned}
K'^{4}_{1,1} &= f(K^{4}_{2,1} \oplus K^{4}_{1,1}) \\
K'^{2}_{1,1} &= f(K^{2}_{1,1}) \\
K'^{2}_{2,1} &= f(K^{2}_{2,1}) \\
K'^{2}_{3,1} &= f(K^{2}_{4,1} \oplus K^{2}_{1,1})
\end{aligned}
$$

As users u_{15} and u_{16} cannot derive the KEKs, the KDC sends those keys through the following messages:

$$
\begin{aligned}
KDC \rightarrow u_{16} &: \{K'^{4}_{1,1}\}_{K^{4}_{2,2}} \\
KDC \rightarrow u_{15} &: \{K'^{2}_{1,1}\}_{K'^{2}_{2,1}} \| \{K'^{2}_{2,1}\}_{K'^{2}_{3,1}} \| \{K'^{2}_{3,1}\}_{K^{2}_{4,2}}
\end{aligned}
$$

To finish the rekeying, the KDC computes the new SK_3, which is the key in $\Omega_2 \triangle \Omega_4$. Then the KDC transmits the new SK'_3 to all the members of OG_3 and OG_4, using the following messages:

$$KDC \rightarrow OG_3 : \{SK'_3\}_{K^3_{1,1}}$$
$$KDC \rightarrow OG_4 : \{SK'_3\}_{K^4_{1,1}}$$

4.2 Decentralized mechanism (DMM-MSKMS)

In the centralized mechanism stated in section 4.1, the responsibilities of the generation and distribution of the SKs are located in a single entity; thus, making the system to have a single point of failure: the KDC. If the KDC fails at any given moment, the security and the functionality of the whole system can be affected. In order to solve that problem, we propose an extension of the MM-MSKMS, named Decentralized Multimedia Multi-session Key Management Scheme (DMM-MSKMS). In the DMM-MSKMS each Overlapping Group (OG) can be controlled by an independent server. In this way, if one of the independent servers fails, only the users of the related OG are affected without compromising the remaining system.

4.2.1 Architecture

The DMM-MSKMS inherits the organization of keys and users of the MM-MSKMS, with the difference that in the DMM-MSKMS each OG is controlled by an independent server named Overlapping Group Controller (OGC). Thus, each tree of the key forest is held by an OGC. Therefore, in a system where there are s sessions, there will be at most $m(s)$ OGCs (see equation 4.1).

In addition to the key forest, in the DMM-MSKMS a second level of key management is defined and which is used to organize the OGCs and the sessions, as shown in Figure 4.5.

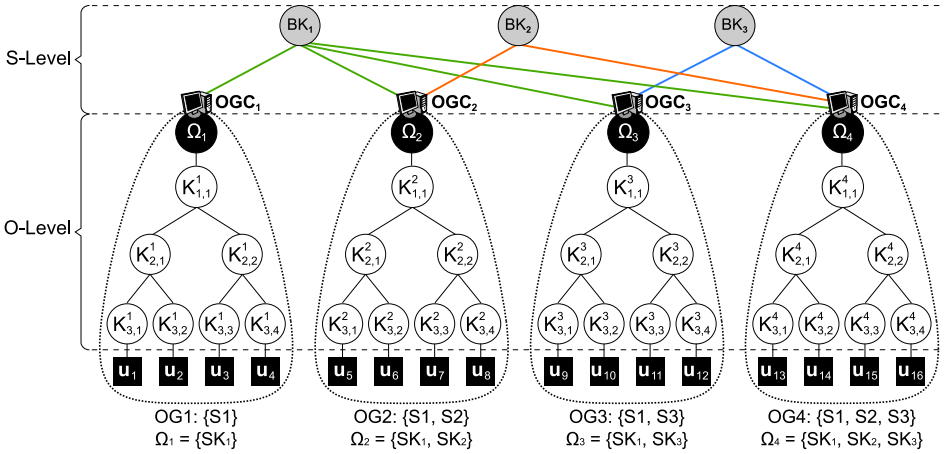

Figure 4.5: Decentralized organization for 4 OGs related to three sessions

As shown in Figure 4.5, the DMM-MSKMS architecture consists of two key management levels: the S-Level and the O-Level. In the S-Level, the OGCs that share an SK agree on a common key to encrypt the interchanged messages among them at the SKs updating. The agreed keys are called Blinded Keys (BKs), and there will be one BK for each SK. In the O-Level, each OGC manages the SKs and the KEKs related to an OG in a similar way in which the KDC does it in the MM-MSKMS. Therefore, each OGC needs to store all the KEKs involved in its tree, the SKs related to its OG and the BKs needed to encrypt the SKs.

In the DMM-MSKMS as in the MM-MSKMS, each OG member must store all the keys located along the tree path where it is joined, from its individual key to the rKEK, along with the SKs related to its OG. Also OG members do not need to store the DEKs, since such keys are generated instantly at the beginning of the transmission of packets.

4.2.2 Key generation

In this section we focused only in the BKs generation, because in DMM-MSKMS the SKs, the KEKs and the DEKs are generated in the same manner as in the MM-MSKMS (see section 4.1.2).

4.2.2.1 BKs generation

The BKs are keys generated in a contributory way, using the extended version of the Diffie-Hellman key exchange (GDH.2 protocol) proposed in [STW96] that supports group operations.

With the GDH.2 protocol, n entities of a work group, agree *a priori* on a cyclic group G, of order q, and a generator α of G. Then, the n entities contribute to collect n values, in a distributive fashion, to form a key in the following way:

1: Each entity randomly chooses a value $x_i \in G$, called *nonce*. For a work group of n entities, there will be n different *nonces* $(x_1, x_2, ..., x_n)$

2: With all the *nonces*, the entities form the value $\alpha^{x_1 x_2 \cdots x_n} \mod q$ in a contributory way. The first entity calculates the first value α^{x_1} and passes it to the next entity. When the second entity receives the value α^{x_1}, it raises that value to the power of its *nonce* (suppose x_2) and forms the set $\{\alpha^{x_1}, \alpha^{x_1 x_2}\}$, then passes it to the third entity. Each subsequent entity receives the set of intermediary values and raises them using its own nonce, generating a new set. A set generated by the ith entity will have i intermediate values with $i-1$ exponents and a *cardinal* value containing all the exponents. For example, if the fourth member receives the set:

$$\{\alpha^{x_2 x_3}, \alpha^{x_1 x_3}, \alpha^{x_1 x_2}, \alpha^{x_1 x_2 x_3}\}$$

generates the set

$$\{\alpha^{x_2 x_3 x_4}, \alpha^{x_1 x_3 x_4}, \alpha^{x_1 x_2 x_4}, \alpha^{x_1 x_2 x_3}, \alpha^{x_1 x_2 x_3 x_4}\}$$

where $\alpha^{x_1 x_2 x_3 x_4}$ is the *cardinal* value.

3: Finally, the last entity sets the group key to $\alpha^{x_1 x_2 \cdots x_n} \mod q$ and multicasts to the whole work group, the set of all the intermediate values received from the previous entity. Thus, any entity can extract its intermediate value and calculate the group key.

4.2.3 Rekey operations

In the DMM-MSKMS, two kinds of rekey operations can be performed: the rekeying of the S-Level and the rekeying of the O-Level. The rekeying of the S-Level is performed when an OGC joins or leaves the system, for which the OGCs must agree on the BKs, needed to encrypt the SKs, shared with the new OGC, using the GDH.2 protocol. The rekeying of the O-Level is performed by an OGC at the change of any user membership, in a similar manner as the KDC performs the rekeying in the MM-MSKMS.

4.2.3.1 Rekeying of the S-Level

OGC join. When a new OGC joins the system, the OGCs that share some SKs start a BK agreement for each of the compromised SKs. For each BK, the oldest related OGC will be the responsible entity for generating the first of the intermediate values. Thus the new OGC will be the responsible entity for generating the last *cardinal* value and to establish the BK.

After the OGCs have agreed on the BKs, the oldest related OGCs update the compromised SKs in order to maintain the backward secrecy in each OG. The OGCs responsible for generating the SKs will have to encrypt the SKs with the corresponding BKs and multicast the packet to all the related OGCs. At the receiving of the SKs, each OGC individually distributes those keys into their OGs, encrypting the messages with the corresponding rKEKs.

For example, considering the system shown in figure 4.6, if a new OGC (OGC_5) joins the system in order to control an OG_5, where users maintain work sessions with S_2 and S_3, the OGCs OGC_2, OGC_3, OGC_4 and OGC_5 have to agree on the new BK_2 and BK_3 respectively, and update the corresponding SKs.

To agree on the new BK_2, the OGC_2 calculates a value $\alpha_2^{x_2}$ and sends it to the OGC_4. The OGC_4 calculates the value $\alpha_2^{x_2x_4}$ and constructs the set $\{\alpha_2^{x_2}, \alpha_2^{x_4}, \alpha_2^{x_2x_4}\}$, sending it to the OGC_5. Finally, the OGC_5 calculates the value $\alpha_2^{x_2x_4x_5}$, sets the BK_2 to $\alpha_2^{x_2x_4x_5} \mod q$, and multicasts the set $\{\alpha_2^{x_4x_5}, \alpha_2^{x_2x_5}\}$.

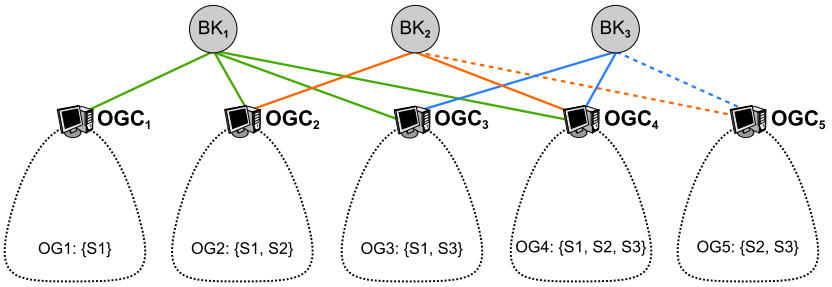

Figure 4.6: An example of OGC join

In a similar way, to agree on the new BK_3, the OGC_3 calculates a value $\alpha_3^{x_3}$ and sends it to the OGC_4. The OGC_4 calculates the value $\alpha_3^{x_3x_4}$ and constructs the set $\{\alpha_3^{x_3}, \alpha_3^{x_4}, \alpha_3^{x_3x_4}\}$, sending it to the OGC_5. Finally, the OGC_5 calculates the value $\alpha_3^{x_3x_4x_5}$, sets the BK_3 to $\alpha_3^{x_3x_4x_5} \mod q$ and multicasts the set $\{\alpha_3^{x_4x_5}, \alpha_3^{x_3x_5}\}$.

To finish the rekeying of the S-Level, the OGC_2 generates a new SK_2 and the OGC_3 generates a new SK_3, and both OGCs multicast those keys to OGC_4 and OGC_5. After receiving the new SKs, the involved OGCs multicast the corresponding SKs into their OGs, encrypting those keys with the corresponding rKEKs. The involved OGCs transmit the new SKs using the following messages:

$$OGC_2 \rightarrow OGC_4, OGC_5 : \{SK_2\}_{BK_2}$$

$$OGC_3 \rightarrow OGC_4, OGC_5 : \{SK_3\}_{BK_3}$$

$$OGC_2 \rightarrow OG_2 : \{SK_2\}_{K^2_{1,1}}$$

$$OGC_3 \rightarrow OG_3 : \{SK_3\}_{K^3_{1,1}}$$

$$OGC_4 \rightarrow OG_4 : \{SK_2, SK_3\}_{K^4_{1,1}}$$

$$OGC_5 \rightarrow OG_5 : \{SK_2, SK_3\}_{K^5_{1,1}}$$

The rekeying for the OGC join process is detailed in Algorithms 5 and 6.

OGC leave. When an OGC leaves the system, the remaining OGCs have to agree on the new BKs in order to maintain the forward secrecy. The BK agreement is performed excluding the OGC that left the system. As in the OGC join process, for each

Algorithm 5 OGC join algorithm in the current OGCs side

Input: OGC_join_notification(Θ)
Output: Updated Keys
Ω_{m_0+1} =get_related_SKs_with(Θ)
if local OGC_t holds $\{SK_h | SK_h \in \Omega_t \cap \Omega_{m_0+1} \wedge h \in [1,s]\}$ **then**
  for each BK_h related to a $SK_h \in \Omega_t \cap \Omega_{m_0+1}$ **do**
    if local OGC_t is the oldest entity related to BK_h **then**
      x_1 = choose_a_nonce_in(G)
      unicast($\alpha_h^{x_1}, next\_related\_OGC$)
    else
      wait_until_the_reception_of($\{\alpha_h^{\prod(x_i|i\in[1,t])} | t \in [1,m_0]\}$)
      x_t = choose_a_nonce_in(G)
      for each $\gamma \in \{\alpha_h^{\prod(x_i|i\in[1,t])} | t \in [1,m_0]\}$ **do**
        $\gamma = \gamma^{x_t}$
      end for
      unicast($\{\alpha_h^{\prod(x_i|i\in[1,t])} | t \in [1,m_0]\} \cup \{\alpha^{x_1 \cdots x_t}\}, next\_related\_OGC$)
    end if
  end for
  wait_until_the_reception_of($all\_the\_updated\_BKs$)
  $m_0 = m_0 + 1$
  for each $SK_h \in \Omega_t \cap \Omega_{m_0}$ **do**
    if local OGC_t is the oldest entity related to SK_h **then**
      (p,q) =generate_two_primes_congruent_with($3 \mod 4$)
      x = generate_a_relative_prime_with($b = pq$)
      $SK_h = \{p,q,x,b\}$
      multicast($\{SK_h\}_{BK_h}, related\_OGCs$)
    end if
  end for
  wait_until_the_reception_of($all\_the\_updated\_SKs$)
  multicast($\{\Omega_t \cap \Omega_{m_0}\}_{K_{1,1}^t}, OG_t$)
end if

Algorithm 6 OGC join algorithm in the new OGC side

Input: $(\{\alpha_l^{\prod(x_i|i\in[1,t])} | t \in [1,m_0]\})$
Output: Updated Keys
x_{new} = choose_a_nonce_in(G)
$BK_h = \alpha_h^{x_1 \cdots x_{new}}$
for each $\gamma \in \{\alpha_h^{\prod(x_i|i\in[1,t])} | t \in [1,m_0]\}$ **do**
  $\gamma = \gamma^{x_{new}}$
end for
multicast($\{\alpha_h^{\prod(x_i|i\in[1,new])} | new \in [1,m_0+1]\}$)
wait_until_the_reception_of($all\_the\_updated\_BKs$)
wait_until_the_reception_of($all\_the\_updated\_SKs$)
multicast($\{\Omega_{new}\}_{K_{1,1}^{new}}, OG_{new}$)

BK, the oldest related OGC will be the responsible entity for generating the first of the intermediate values. Therefore, the newest OGC will be the responsible entity for generating the last *cardinal* value and to establish the BK.

In the same manner as the OGC join process, the oldest related OGCs have to update the compromised SKs and distribute those keys among the involved OGCs, encrypting the messages with the corresponding BKs. Then each involved OGC distributes the updated SKs into their OGs, encrypting the messages with the corresponding rKEKs.

For example, considering the system shown in Figure 4.7, if the OGC_3 leaves the system, all the OGCs have to agree on the new BK_1 and BK_3, respectively, and update the corresponding SKs.

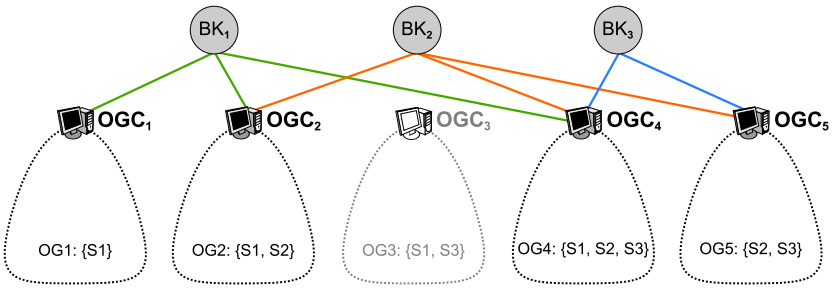

Figure 4.7: An example of OGC leave

To agree on the new BK_1, the OGC_1 calculates a value $\alpha_1^{x_1}$ and sends it to the OGC_2. The OGC_2 calculates the value $\alpha_1^{x_1 x_2}$ and constructs the set $\{\alpha_1^{x_1}, \alpha_1^{x_2}, \alpha_1^{x_1 x_2}\}$, sending it to the OGC_4. Finally, the OGC_4 calculates the value $\alpha_1^{x_1 x_2 x_4}$, sets the BK_1 to $\alpha_1^{x_1 x_2 x_4} \mod q$ and multicasts the set $\{\alpha_1^{x_2 x_4}, \alpha_1^{x_1 x_4}\}$.

To agree on the new BK_3, the OGC_4 calculates a value $\alpha_3^{x_4}$ and sends it to the OGC_5. In this case, the OGC_5 simply sets the BK_3 to $\alpha_3^{x_4 x_5} \mod q$ and transmits $\{\alpha_3^{x_5}\}$.

To finish the rekeying of the S-Level, the OGC_1 generates a new SK_1 and multicasts that key to OGC_2 and OGC_4, while the OGC_4 generates a new SK_3 and sends that key to OGC_5. After receiving the new SKs, the involved OGCs multicasts the corresponding SKs into their OGs, encrypting those keys with the corresponding rKEKs. The involved

OGCs transmit the new SKs using the following messages:

$$OGC_1 \rightarrow OGC_2, OGC_4 : \{SK_1\}_{BK_1}$$

$$OGC_4 \rightarrow OGC_5 : \{SK_3\}_{BK_3}$$

$$OGC_1 \rightarrow OG_1 : \{SK_1\}_{K^1_{1,1}}$$

$$OGC_2 \rightarrow OG_2 : \{SK_1\}_{K^2_{1,1}}$$

$$OGC_4 \rightarrow OG_4 : \{SK_3\}_{K^4_{1,1}}$$

$$OGC_5 \rightarrow OG_5 : \{SK_3\}_{K^5_{1,1}}$$

The rekeying for the OGC leave process is detailed in Algorithm 7.

Algorithm 7 OGC leave algorithm

Input: OGC_leave_notification($OGC_{leave}, \Omega_{leave}$)
Output: Updated Keys
if local OGC_t holds $\{SK_h | SK_h \in \Omega_t \cap \Omega_{leave} \wedge h \in [1, s]\}$ **then**
  for each BK_h related to a $SK_h \in \Omega_t \cap \Omega_{leave}$ **do**
    if local OGC_t is the oldest entity related to BK_h **then**
      x_1 = choose_a_nonce_in(G)
      unicast($\alpha_h^{x_1}, next\_related\_OGC$)
    else
      wait_until_the_reception_of($\{\alpha_h^{\prod(x_i|i\in[1,t])} | t \in [1, m_0] \wedge i \neq leave\}$)
      x_t = choose_a_nonce_in(G)
      for each $\gamma \in \{\alpha_h^{\prod(x_i|i\in[1,t])} | t \in [1, m_0] \wedge i \neq leave\}$ **do**
        $\gamma = \gamma^{x_t}$
      end for
      unicast($\{\alpha_h^{\prod(x_i|i\in[1,t])} | t \in [1, m_0] \wedge i \neq leave\} \cup \{\alpha^{x_1 \cdots x_t}\}, next\_related\_OGC$)
    end if
  end for
  wait_until_the_reception_of($all\_the\_updated\_BKs$)
  $m_0 = m_0 - 1$
  for each $SK_h \in \Omega_t \cap \Omega_{leave}$ **do**
    if local OGC_t is the oldest entity related to SK_h **then**
      (p, q) =generate_two_primes_congruent_with(3 mod 4)
      x = generate_a_relative_prime_with($b = pq$)
      $SK_h = \{p, q, x, b\}$
      multicast($\{SK_h\}_{BK_h}, related\_OGCs$)
    end if
  end for
  wait_until_the_reception_of($all\_the\_updated\_SKs$)
  multicast($\{\Omega_t \cap \Omega_{leave} | h \in [1, s]\}_{K^t_{1,1}}, OG_t$)
end if

4.2.3.2 Rekeying of the O-Level

As we mentioned above, the rekeying of the O-Level is performed individually by an OGC, when a user joins or leaves an OG in order to leave the whole system or when a user simply change its OG and its joined sessions. Thus, the rekeying process performed by an OGC is quite similar to the rekeying process performed by the KDC in the MM-MSKMS. The main difference is that when an OGC updates an SK, it must also transmit that key to the other OGCs.

User join. When a new user requests to join the system, the responsible OGC for controlling the OG involved with the requested sessions will be the entity that processes the user join.

Assuming that the OGC_t is responsible for attending the join request, the user join process is performed as follows:

1: The OGC_t updates the corresponding KEK-tree in the same manner as the KDC in the MM-MSKMS (see section 4.1.3.1).

2: The OGC_t updates the SKs of the set Ω_t in the same manner as the KDC in the MM-MSKMS (see section 4.1.3.1).

3: At the SKs updating, the OGC_t also multicasts the updated SKs to the related OGCs, encrypting the messages with the corresponding BKs.

As in the MM-MSKMS, the members in the OG_t derive the compromised KEKs by themselves after the OGC's join notification.

The rekeying for the user join process is detailed in Algorithms 8 and 9.

User leave. When a user leaves the system, the involved OGC must update the compromised KEKs and SKs.

Assuming that the OGC_t is responsible for the affected OG, the user leave process is performed as follows:

Algorithm 8 User join algorithm on OGC's side

Input: join_request_message($user, \Theta$) /*Θ *is a set with the requested sessions*/*
Output: Updated Keys
Ω_{new_user} =get_related_SKs_with(Θ)
if local OGC_t holds $\Omega_t = \Omega_{new\_user}$ **then**
  $user\_key$ = generate_key()
  unicast($user\_key, user$)
  $height$ = get_height_of(t)
  (i, j) = get_last_internal_vertex(t)
  if subtree(i, j) is not full **then** /*verifies if the last internal vertex can hold a new vertex*/
    (i, j) = get_right_most_leaf$(t, height + 1)$
    $K^t_{i,j+1} = user\_key$
  else
    if $j < d^{height-1}$ **then** /*insert a new vertex under the next available vertex*/
      (i, j) = get_left_most_leaf$(t, height)$
    else /*create a new KEK-tree level*/
      (i, j) = get_left_most_leaf$(t, height + 1)$
    end if
    /*new intermediate key derivation*/
    $K^t_{i+1,d(j-1)+1} = K^t_{i,j}$
    $K^t_{i,j} = f(K^t_{i,j} \oplus K^t_{1,1})$
    $K^t_{i+1,d(j-1)+2} = user\_key$
  end if
  multicast(join_notification$(i + 1, d(j - 1) + 2), OG_t$)
  $i = i - 1$
  while $i > 0$ **do** /*update of the compromised KEKs*/
    $j = \lceil j/d \rceil$
    $K^t_{i,j} = f(K^t_{i,j})$
    $i = i - 1$
  end while
  unicast(updated_KEKs(), $user$)
  for each $SK_h \in \Omega_t$ **do** /*update of the compromised SKs*/
    (p, q) =generate_two_primes_congruent_with(3 mod 4)
    x = generate_a_relative_prime_with($b = pq$)
    $SK_h = \{p, q, x, b\}$
    multicast($\{SK_h\}_{BK_h}, related\_OGCs$)
  end for
  multicast($\{\Omega_t\}_{K^t_{1,1}}, OG_t$)
end if

Algorithm 9 User join algorithm in user's side

Input: join_notification(i, j)
Output: Updated Keys
$x = i - 1$
$y = \lceil j/d \rceil$
*/*verifies is a new intermediate vertex has been inserted*/*
if local_user_individual_key()= $K^t_{x,y}$ **then**
    */*new intermediate key derivation*/*
    $K^t_{x+1,d(j-1)+1}$ =local_user_individual_key()
    $K^t_{x,y} = f(K^t_{x,y} \oplus K^t_{1,1})$
    $i = i - 1$
    $j = \lceil j/d \rceil$
end if
$i = i - 1$
while $i > 0$ **do** */*update of the compromised KEKs*/*
    $j = \lceil j/d \rceil$
    if local_user_holds($\{K^t_{i,j}\}$) **then**
        $K^t_{i,j} = f(K^t_{i,j})$
    end if
    $i = i - 1$
end while
*/*update of the compromised SKs*/*
wait_until_the_reception_of($\{\Omega_t\}_{K^t_{1,1}}$)
decrypt($\{\Omega_t\}_{K^t_{1,1}}$)

1: The OGC_t updates the corresponding KEK-tree in the same manner as the KDC in the MM-MSKMS (see section 4.1.3.2).

2: The OGC_t updates the SKs of the set Ω_t in the same manner as the KDC in the MM-MSKMS (see section 4.1.3.2).

3: At the SKs updating, the OGC_t also multicasts the updated SKs to the related OGCs, encrypting the messages with the corresponding BKs.

The remaining members of the OG_t that can derive the compromised KEKs compute such keys by themselves after the OGC's leave notification. For the members that cannot derive the compromised KEKs, the OGC_t has to send the new KEKs.

The rekeying for the user leave process is detailed in Algorithms 10 and 11.

Algorithm 10 User leave algorithm on OGC's side

Input: leave_request_message(i, j)
Output: Updated Keys
multicast(leave_notification$(i, j), OG_t$)
delete_vertex(i, j, t)
if number_of_children_of$(K^t_{i-1,\lceil j/d \rceil}) = 1$ **then**
    $K^t_{i-1,\lceil j/d \rceil} = K^t_{i,j}$ /*move the key to a upper level*/
    $i = i - 1$
    $j = \lceil j/d \rceil$
end if
$y = i$
while $i > 1$ **do** /*update of the compromised KEKs*/
    (h, v) =get_left_most_sibling_of(i, j, t)
    $K^t_{i-1,\lceil j/d \rceil} = f(K^t_{h,v} \oplus K^t_{i-1,\lceil j/d \rceil})$ /*KEKs derivation*/
    $i = i - 1$
    $j = \lceil j/d \rceil$
end while
multicast(updated_KEKs(), users_that_cannot_derive())
for each $SK_h \in \Omega_t$ **do** /*update of the compromised SKs*/
    (p, q) =generate_two_primes_congruent_with$(3 \mod 4)$
    x = generate_a_relative_prime_with$(b = pq)$
    $SK_h = \{p, q, x, b\}$
    multicast$(\{SK_h\}_{BK_h}, related_OGCs)$
end for
multicast$(\{\Omega_t\}_{K^t_{1,1}}, OG_t)$

Algorithm 11 User leave algorithm in user's side

Input: leave_notification(i, j)
Output: Updated Keys
    $i = i - 1$
    $j = \lceil j/d \rceil$
while $i > 1$ **do** /*update of the compromised SKs*/
    (h, v) =get_left_most_sibling_of(i, j)
    if local_user_ holds$(\{K^t_{i-1,\lceil j/d \rceil}, K^t_{h,v}\})$ **then**
        $K^t_{i-1,\lceil j/d \rceil} = f(K^t_{h,v} \oplus K^t_{i-1,\lceil j/d \rceil})$ /*KEKs derivation*/
    end if
    $i = i - 1$
    $j = \lceil j/d \rceil$
end while
/*update of the compromised SKs*/
wait_until_the_reception_of$(\{\Omega_t\}_{K^t_{1,1}})$
decrypt$(\{\Omega_t\}_{K^t_{1,1}})$

User switch. The user switch process is the only one that differs from the processes performed in the MM-MSKMS, because in the DMM-MSKMS, the user that asks to change its OG, must inform the current OGC about the desired changes in its memberships, before its departure. Thus, the current OGC negotiates with the destiny OGC, the SKs that they need to update.

Assuming that a user needs to leave the OG_y to join an OG_z, the user switch process is performed as follows:

1: The user requests the OGC_y to leave the OG_y in order to change its memberships, indicating which sessions it wants to join.

2: The OGC_y sends to the OGC_z a message informing about the switch process, the SKs that it holds, and the user's ID.

3: The OGC_z responds to the OGC_y through a message, indicating which SKs are in the symmetric difference of Ω_z and Ω_y ($\Omega_z \triangle \Omega_y$).

4: When the OGC_y receives the response from the OGC_z, it updates the compromised KEKs in the same manner as the KDC in the MM-MSKMS, at the user leave process (see section 4.1.3.2). Then, the OGC_y will wait a time τ so that the OGC_z notices the join of the switched user to the OG_z.

5: When the OGC_z receives the join request from the switched user, it notifies the OGC_y about the user join and updates the compromised KEKs in the same manner as the KDC in the MM-MSKMS at the user join process (see section 4.1.3.1). Then, the OGC_z updates the SKs in $\Omega_z \cap (\Omega_z \triangle \Omega_y)$ and multicasts the updated SKs into its OG, using the corresponding rKEKs to encrypt the message. The OGC_z also multicasts the updated SKs to the related OGCs, encrypting the messages with the corresponding BKs.

6: When the OGC_y receives the notification from the OGC_z, about the user join, it updates the SKs in $\Omega_y \cap (\Omega_z \triangle \Omega_y)$. If after a time τ, the notification from the

OGC_z is not received, the OGC_y updates all the SKs in the set Ω_y. The OGC_y multicasts the updated SKs into its OG, using the corresponding rKEKs, and to the related OGCs, using the corresponding BKs.

The exchange of information between the OGCs involved in the user switch process is intented to ensure the forward secrecy in the case where a user wouldn't finish the switch process. Thus, the time τ involves the period that a user takes to leave an OG and to join another, a time during which some vulnerability in the forward secrecy could be allowed. The variable τ should be specified by the system, and will depend on the network conditions and the join/leave activity. In systems where a high join/leave activity exists, the time τ can be discarded, because the dynamics of the group composition allows the frecuent rekey.

In order to show the rekeying of the O-Level, we use an example of the user switch process.

Considering the system shown in the Figure 4.8, if the user u_{15} requires to leave the OG_4 to join the OG_2, that user sends a switch request to the OGC_4, informing that it will change its sessions, from $\{S_1, S_2, S_3\}$ to $\{S_1, S_2\}$.

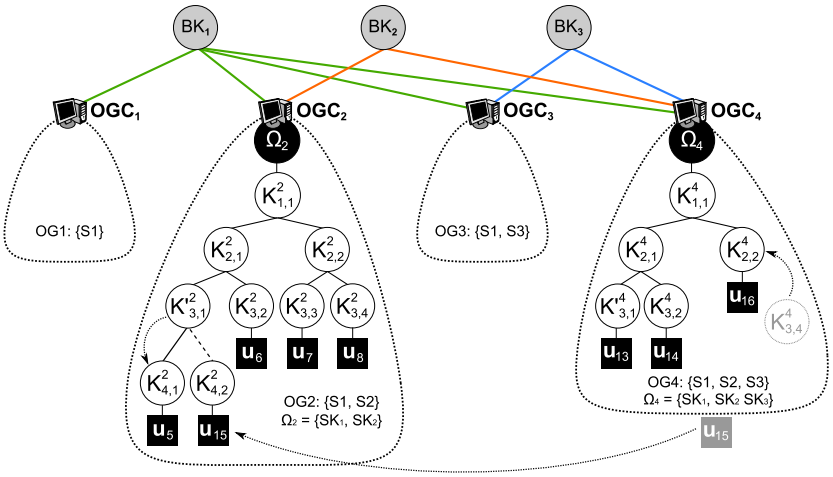

Figure 4.8: An example of user switch

When the OGC_4 receives the switch request, sends a messages to the OGC_2, informing about the switch process and indicating the SKs that it holds. Then the OGC_2

responds with a message indicating that the only SK in the symmetric difference between Ω_z and Ω_y is SK_3.

When the OGC_4 receives the response from the OGC_2, it removes the vertex associated with the individual user key and modifies the KEK-tree as shown in Figure 4.8. Then, the OGC_4 computes the new rKEK by:

$$K'^{4}_{1,1} = f(K^{4}_{2,1} \oplus K^{4}_{1,1})$$

As u_{16} cannot derive the rKEK, the OGC_4 sends that key as follows:

$$OGC_4 \rightarrow u_{16} : \{K'^{4}_{1,1}\}_{K^{4}_{2,2}}$$

Then, the OGC_4 waits for the user join notification from the OGC_2.

When the OGC_2 receives the join request from u_{15}, it sends a messages to the OGC_4 informing about the user join. Then, the OGC_2 modifies the KEK-tree in order to assign a new vertex for the individual key of u_{15} (see Figure 4.8). The OGC_2 updates the compromised KEKs by:

$$\begin{aligned} K'^{2}_{1,1} &= f(K^{2}_{1,1}) \\ K'^{2}_{2,1} &= f(K^{2}_{2,1}) \\ K'^{2}_{3,1} &= f(K^{2}_{4,1} \oplus K^{2}_{1,1}) \end{aligned}$$

As u_{15} cannot derive the KEKs, the OGC_2 sends those keys as follows:

$$OGC_2 \rightarrow u_{15} : \{K'^{2}_{1,1}\}_{K'^{2}_{2,1}} \| \{K'^{2}_{2,1}\}_{K'^{2}_{3,1}} \| \{K'^{2}_{3,1}\}_{K^{2}_{4,2}}$$

As the SK_3 is the only key in the symmetric difference between Ω_2 and Ω_4, the OGC_2 does not have to update any SK.

If the OGC_4 receives the user join notification from the OGC_2, it calculates a new SK_3 and multicasts that key to the members of the OG4 and to the related OGCs

(OGC_1, OGC_2 and OGC_3), using the following messages:

$$OGC_4 \rightarrow OG_4 : \{SK_3\}_{K^4_{1,1}}$$
$$OGC_4 \rightarrow OGC_1, OGC_2, OGC_3 : \{SK_3\}_{BK_3}$$

The rekeying for the user switch process is detailed in Algorithms 12 and 13.

Algorithm 12 User switch algorithm on the side of the responsible OGC of the abandoned OG

Input: leave_request_message(i, j, Θ) /*Θ *is a set with the requested sessions**/
Output: Updated Keys
Ω_z =get_related_SKs_with(Θ)
unicast(switch_notification$(\Omega_y, u_ID), OGC_z$)
wait_until_the_reception_of($\Omega_y \triangle \Omega_z$)
multicast(leave_notification$(i, j), OG_y$)
delete_vertex(i, j, y)
if number_of_children_of($K^y_{i-1,\lceil j/d \rceil}$) $= 1$ **then**
    $K^y_{i-1,\lceil j/d \rceil} = K^y_{i,j}$ /*move the key to a upper level*/
    $i = i - 1$
    $j = \lceil j/d \rceil$
end if
$y = i$
while $i > 1$ **do** /*update of the compromised KEKs*/
    (h, v) =get_left_most_sibling_of(i, j, t)
    $K^y_{i-1,\lceil j/d \rceil} = f(K^y_{h,v} \oplus K^y_{i-1,\lceil j/d \rceil})$ /*KEKs derivation*/
    $i = i - 1$
    $j = \lceil j/d \rceil$
end while
multicast(updated_KEKs(), users_that_cannot_derive())
wait(τ)
if user_join_notification was received from OGC_z **then**
    for each $SK_h \in \Omega_y \cap (\Omega_z \triangle \Omega_y)$ **do** /*update of the compromised SKs*/
        (p, q) =generate_two_primes_congruent_with(3 mod 4)
        x = generate_a_relative_prime_with($b = pq$)
        $SK_h = \{p, q, x, b\}$
        multicast($\{SK_h\}_{BK_h}, related\_OGCs$)
    end for
    multicast($\{\Omega_y \cap (\Omega_z \triangle \Omega_y)\}_{K^y_{1,1}}, OG_y$)
else
    for each $SK_h \in \Omega_y$ **do** /*update of the compromised SKs*/
        (p, q) =generate_two_primes_congruent_with(3 mod 4)
        x = generate_a_relative_prime_with($b = pq$)
        $SK_h = \{p, q, x, b\}$
        multicast($\{SK_h\}_{BK_h}, related\_OGCs$)
    end for
    multicast($\{\Omega_y\}_{K^y_{1,1}}, OG_y$)
end if

Algorithm 13 User switch algorithm on the side of the responsible OGC of the joined OG

Input: switch_notification(Ω_y, u_ID)
Output: Updated Keys
unicast(switch_notification_response($\Omega_y \triangle \Omega_z$), OGC_y)
if a join_request_message is received from $user_{u\_ID}$ **then**
  unicast(user_join_notification(), OGC_y)
  $user\_key$ = generate_key()
  unicast($user\_key, user_{u\_ID}$)
  $height$ = get_height_of(z)
  (i, j) = get_last_iternal_vertex(z)
  if subtree(i, j) is not full **then** */*verifies if the last internal vertex can hold a new vertex*/*
    (i, j) = get_right_most_leaf($z, height + 1$)
    $K^z_{i,j+1} = user\_key$
  else
    if $j < d^{height-1}$ **then** */*insert a new vertex under the next available vertex*/*
      (i, j) = get_left_most_leaf($z, height$)
    else */*create a new KEK-tree level*/*
      (i, j) = get_left_most_leaf($z, height + 1$)
    end if
    */*new intermediate key derivation*/*
    $K^z_{i+1,d(j-1)+1} = K^z_{i,j}$
    $K^z_{i,j} = f(K^z_{i,j} \oplus K^z_{1,1})$
    $K^z_{i+1,d(j-1)+2} = user\_key$
  end if
  multicast(join_notification($i + 1, d(j - 1) + 2$), OG_z)
  $i = i - 1$
  while $i > 0$ **do** */*update of the compromised KEKs*/*
    $j = \lceil j/d \rceil$
    $K^z_{i,j} = f(K^z_{i,j})$
    $i = i - 1$
  end while
  unicast(updated_KEKs(), $user_{u\_ID}$)
  for each $SK_h \in \Omega_z \cap (\Omega_z \triangle \Omega_y)$ **do** */*update of the compromised SKs*/*
    (p, q) =generate_two_primes_congruent_with(3 mod 4)
    x = generate_a_relative_prime_with($b = pq$)
    $SK_h = \{p, q, x, b\}$
    multicast($\{SK_h\}_{BK_h}, related\_OGCs$)
  end for
  multicast($\{\Omega_z \cap (\Omega_z \triangle \Omega_y)\}_{K^z_{1,1}}, OG_z$)
end if

Chapter 5

Performance analysis

In this chapter we analyse the performance of the MM-MSKMS and the DMM-MSKMS, focusing on storage and communication overheads in order to demostrate the efficiency of our solution, compared against other similar solutions.

5.1 Analysis of the MM-MSKMS

5.1.1 Storage overhead

As we mentioned above, in the MM-MSKMS trees are used as storage structures to organize the keys and the members of the different OGs present in the system. Particularly, the trees used in this work can be viewed as a graph composed of a KEK-tree connected with an additional vertex used to store the SKs. Each KEK-tree is maintained as balanced as possible by positioning the joining users on the shortest paths.

Let n denote the number of users joined at the whole system and n_t the number of users involved in a tree (OG). We use $l_d(n_t)$ to denote the length of the branches of a tree of d degree. Since each KEK-tree is balanced and it is possible that not all the branches have the same length at some point, $l_d(n_t)$ is either L or $L+1$, where $L = \lfloor \log_d(n_t) \rfloor$. Particularly,

- the number of users who are on branches with length L is $d^L - \lceil \frac{n_t - d^L}{d-1} \rceil$,
- and the number of users who are on branches with length $L+1$ is $n_t - d^L + \lceil \frac{n_t - d^L}{d-1} \rceil$

Therefore, the total number of keys in a KEK-tree is determined by:

$$TK(n_t) = n_t + \frac{d^L - 1}{d-1} + \lceil \frac{n_t - d^L}{d-1} \rceil \tag{5.1}$$

As the KDC holds the s SKs related to the system sessions and maintains the KEK-trees of the $m(s)$ OGs, the total number of keys stored by the KDC is determined by:

$$TK_{KDC} = \sum_{t=1}^{m(s)} TK(n_t) + s \tag{5.2}$$

Each user joined at an OG_t has to store the $l_d(n_t)$ KEKs involved with its branch and the $|\Omega_t|$ SKs related to its OG. Thus, the total keys stored by each user is determined by:

$$TK_{u \in OG_t} = l_d(n_t) + |\Omega_t| \tag{5.3}$$

Assuming the worst case, where all the combinations of the s sessions exist, we can take $m(s)$ as a fixed value ($m(s) = m_0$) throughout the communication process. If we also assume that all the OGs have the same number of users ($n_t = n_0$), the number of users in the whole system is $n = m_0 \cdot n_0$. Using (5.2), the KDC's storage overhead is calculated as:

$$TK_{KDC} = m_0 \cdot TK(n_0) + s \tag{5.4}$$

Using (5.3), we have that the user's storage overhead is:

$$TK_{u \in OG_t} = l_d(n_0) + |\Omega_t| \tag{5.5}$$

From (5.1), we have that $\lim_{n_0 \to \infty} TK(n_0) = \frac{d}{d-1} n_0$. Therefore, as s is a fixed value throughout the communication process and $s \ll n_0$ when $n_0 \to \infty$, using (5.4) we can calculate the KDC's asymptotic storage overhead as:

$$TK_{KDC} \sim O(\frac{d}{d-1} m_0 \cdot n_0) = O(\frac{d}{d-1} n) \tag{5.6}$$

Since $|\Omega_t|$ is fixed for each OG, using (5.5) we can calculate the user's asymptotic storage overhead as:

$$TK_{u \in OG_t} \sim O(\log_d(n_0)) \tag{5.7}$$

5.1.2 Communication overhead

The communication overhead is determined by the number of messages transmitted at rekey operations. Therefore, we analyze the number of messages involved in the rekeying related to the join, leave and switch processes, in order to determine the communication overhead.

Let $m(s_r)$ be the number of OGs involved with the updated SKs in a rekey operation. According to the algorithm stated in Section 4.1.3.1, when a new user joins the system, the KDC unicasts to the new user a message with the $l_d(n_t)$ KEKs related to its branch and the $|\Omega_t|$ SKs related to its OG. Moreover, the KDC multicasts a join notification with the information of the join branch ($l_d(n_t)$ indices of the affected KEKs) to the remaining users, and also multicasts $m(s_r)$ messages to transmit the SKs to the involved OGs. Therefore, the number of messages sent out by the KDC is determined by:

$$M_{join} = m(s_r) + 2 \tag{5.8}$$

As each OG_t is related to $|\Omega_t|$ SKs, the total number of SKs sent by the KDC to the $m(s_r)$ involved groups in a rekey operation is determined by:

$$N_K = \sum_{l=1}^{m(s_r)} |\Omega_t \cap \Omega_l| \tag{5.9}$$

where Ω_t denotes the set of SKs related to an OG_t and Ω_l denotes the set of SKs related to an OG_l.

Therefore, the number of keys sent out by the KDC in the rekeying, needed for the user join process, is determined by the following equations:

$$NK_{Junicast} = l_d(n_t) + |\Omega_t| \tag{5.10}$$

$$NK_{Jmulticast} = N_K \tag{5.11}$$

where $NK_{Junicast}$ denotes the number of keys sent out in a unicast communication and $NK_{Jmulticast}$ denotes the number of keys sent out in a multicast way.

When a user leaves the system, the KDC multicasts $(d-1)l_d(n_t)$ messages with the KEKs of the affected branch to the users which cannot derive them, a message with a leave notification, and also multicasts $m(s_r)$ messages with the updated SKs to the involved OGs. Therefore, the number of messages sent out by the KDC is determined by:

$$M_{leave} = (d-1)l_d(n_t) + m(s_r) + 1 \tag{5.12}$$

As for the user leave process only the remaining users are involved, the KDC does not send any message in a unicast communication. Thus, using 5.9 the number of keys sent out by the KDC in the rekeying, needed for the user leave process, is determined by:

$$NK_{Lmulticast} = (d-1)l_d(n_t) + N_K \tag{5.13}$$

As the rekey for the user switch process involves the join and leave processes, from (5.8) and (5.12) we know that $(d-1)l_d(n_t) + 3$ messages are necessary to update the compromised KEKs, 2 messages to update the KEKs in the joined group and $(d-1)l_d(n_t) + 1$ to update the KEKs of the left group. Assuming that a user switches from OG_y to OG_z, the KDC has to update the SKs in $\Omega_y \triangle \Omega_z$. Let $m(s_r)$ be the number of messages to update such SKs, then the number of messages sent out by the KDC is determined by:

$$M_{switch} = (d-1)l_d(n_t) + m(s_r) + 3 \tag{5.14}$$

Using 5.9, 5.10, 5.11 and 5.13, the total number of keys sent out by the KDC in the rekeying, needed for the user swtich process is determined by:

$$NK_{Sunicast} = l_d(n_t) + |\Omega_t| \tag{5.15}$$

$$NK_{Smulticast} = dl_d(n_t) + N_K \tag{5.16}$$

In this case, N_K involves the number of SKs sent out by the KDC to the $m(s_r)$ groups, related to the $|\Omega_y \triangle \Omega_z|$ updated SKs.

The switch process involves the two basic rekey operations: join and leave; thus, we can use it to determine the highest bound of transmitted messages.

Assuming the worst case, which is when the user switch process involves an OG related to all the s sessions, and an OG related only with one session, we have that $m(s_r) = m(s-1)$. Furthermore, if we also assume that all the OGs have the same number of users, $n_t = n_0$, using (5.14) the total number of messages involved in the rekeying process is given by:

$$M = (d-1)l_d(n_0) + m(s-1) + 3 \tag{5.17}$$

If $n_0 \to \infty$, as s is fixed throughout the communication process and $m(s-1) < n_0$, we can see that the asymptotic communication overhead is:

$$M \sim O(d \log_d(n_0)) \tag{5.18}$$

5.1.3 Comparison

In this section we compare the MM-MSKMS and the DACMGS because both schemes use a similar key organization. The comparison is focused on two measures: the storage overhead and the communication overhead.

In Table 5.1 we summarize the measurements, which are expressed in bits. These results are based on the results of DACMGS [DML04], and on the results obtained in Sections 5.1.2 and 5.1.1.

In Table 5.1, S_K denotes the KEK's and the TEK's size, S_{ck} denotes the size of a secret that is smaller than a KEK, while S_{sk} denotes the size of a SK. If we use a cryptosystem with KEKs of 128 bits, the size of the SKs should be $O(192)$ bits, using the BBS algorithm with 32-bit integers.

For a fair comparison, we compare DACMGS and MM-MSKMS breaking down the communication cost in the indivual costs of join, leave and switch processes. Moreover, the communication costs, shown in Table 5.1, include the size of the notification messages sent out by the KDC in the MM-MSKMS.

Table 5.1: Performance comparison for storage and communication overhead

		DACMGS	MM-MSKMS
Storage costs			
	KDC	$(\frac{d}{d-1}n + s)S_K$	$\frac{d}{d-1}nS_K + sS_{sk}$
	User	$(\log_d(n_t) + \lvert\Omega_t\rvert)S_K$	$\log_d(n_t)S_K + \lvert\Omega_t\rvert S_{sk}$
Communication costs			
User join	Unicast	$(\log_d(n_t) + 1)S_K$	$\log_d(n_t)S_K + \lvert\Omega_t\rvert S_{sk}$
	Multicast	0	$\log_d(n_t) + N_K S_{sk}$
User leave	Unicast	0	0
	Multicast	$d(\log_d(n_t) - 1)S_K + N_K S_{ck}$	$(d-1)\log_d(n_t)S_K + N_K S_{sk}$
User switch	Unicast	$(\log_d(n_t) + 1)S_K$	$\log_d(n_t)S_K + \lvert\Omega_t\rvert S_{sk}$
	Multicast	$d(\log_d(n_t) - 1)S_K + N_K S_{ck}$	$d\log_d(n_t)S_K + N_K S_{sk}$

In addition to the costs, we note that both DACMGS as MM-MSKMS lack of the *1 affects n* phenomenon, unlike some of the solutions related to multi-group key management exposed in Chapter 3.

In Table 5.1 we can observe that DACMGS does not multicast any key in the user join process. The reason is that all the users in DACMGS work only as receivers entities

in a 1 to n communication. Thus the KDC does not have to transmit the new keys in the rekeying process because the new version of the keys are indicated in the received packets. Unlike the DACMGS, the MM-MSKMS is designed for n to n communications and each user is a transceiver entity; therefore, the KDC has to multicast some of the updated keys in the user join process to avoid inconsistencies in transmissions. However, the cost of MM-MSKMS does not differ significantly compared with the cost of DACMGS.

As can be noticed in Table 5.1, the MM-MSKMS and DACMG are schemes where the rekey overhead depends on the number of groups in which the keys are involved ($m(s_r)$). Both schemes are not scalable in the number of groups because the rekey overhead grows linearly with the number of groups in the system.

5.2 Analysis of the DMM-MSKMS

5.2.1 Storage overhead

As we mentioned in Section 4.2, in DMM-MSKMS each OGC holds the KEK-tree related to the OG that it manages. In addition, each OGC holds the SKs and BKs related to the sessions which it is involved. As each OGC_t holds $|\Omega_t|$ SKs and there is a BK for each SK, using (5.1) the total number of keys stored by each OGC is determined by:

$$TK_{OGC_t} = TK(n_t) + 2|\Omega_t| \tag{5.19}$$

From (5.1) also it is known that $\lim_{n_0 \to \infty} TK(n_0) = \frac{d}{d-1} n_0$. Furthermore, $|\Omega_t|$ is a fixed value for each OGC, throughout the communication process. If we assume that all the OGs have the same number of users, denoted by $n_t = n_0$, using (5.19) we can calculate the OGC's storage overhead asymptotically as:

$$TK_{OGC} \sim O(\frac{d}{d-1} n_0) \tag{5.20}$$

In the same manner as in MM-MSKMS, in DMM-MSKMS each user joined at an OG_t has to store the KEKs involved with its branch and the $|\Omega_t|$ SKs related to its OG. Thus, the total keys stored by each user in DMM-MSKMS is the same given by equation (5.3). Therefore, assuming the same number of users in all the OGs ($n_t = n_0$), asymptotically the amount of keys held by a user in the DMM-MSKMS is given by equation (5.7) ($TK_{u \in OG_t} \sim O(\log_d(n_0))$).

5.2.2 Communication overhead

In DMM-MSKMS there are two levels of key management, for this reason we analyze the amount of messages involved in the rekeying of each level separately.

5.2.2.1 Communication overhead in the S-Level

As we can notice in the algorithms stated in Section 4.2.3.1, when an OGC_t joins or leaves the system, the OGCs which are involved with the SKs in Ω_t have to agree on a BK for each of the compromised SKs.

Let $m(s_h)$ be the number of OGCs involved with a $SK_h \in \Omega_t$, the related BK is agreed through $m(s_h)$ rounds, where each involved OGC sends a message with a set of intermediate values to the next involved OGC, until the last involved OGC multicasts the new BK. Thus, a BK agreement needs $m(s_h)$ messages. In addition, each $SK_h \in \Omega_t$ must be updated so one message for each compromised SK is required in order to distribute such keys among the involved OGCs. At the receiving of the updated SKs, each of the involved OGC multicasts such keys to its OG. Let $m(s_{\Omega_t})$ be the number of OGCs involved with the compromised SKs. The number of messages needed for the rekeying of the S-Level is given by:

$$M_{S-Level} = \sum_{h=1}^{|\Omega_t|} m(s_h) + |\Omega_t| + m(s_{\Omega_t}) \tag{5.21}$$

Assuming the worst case, when the joining of a new OGC, which is involved with all the s sessions, complements the $m(s)$ OGs in the system, we have that $|\Omega_t| = s$ and

$m(s_{\Omega_t}) = m(s)$. Furthermore, as all the OGs are related to the SKs in Ω_t, there are the same number of OGCs involved with each $SK_h \in \Omega_t$, $m(s_h) = 2^{s-1}$. Thus, from (5.21), the number of messages for the rekeying of the S-Level is calculated asymptotically as:

$$M_{S-Level} \sim O(s(2^{s-1} + 1) + m(s)) \tag{5.22}$$

Although this cost is high, it does not affect the scalability of the system, since the addition or the removal of an OGC is handled without involving any user. Those events are handled only by the OGCs, in the setup phase of an OG. Furthermore, the addition and/or removal of an OGC are events that occur with a very low frequency.

5.2.2.2 Communication overhead in O-Level

The rekey overhead in the O-Level is determined by the amount of messages sent out by the OGCs, in the rekeying related to the join, leave and switch processes.

With the same assumptions as in Section 5.1.1, let $l_d(n_t)$ be the branches length of the branches of a tree of d degrees. According to the algorithm stated in Section 4.2.3.2, the number of messages sent out by an OGC_t in the user join process is determined by:

$$M_{join} = |\Omega_t| + 4 \tag{5.23}$$

which includes two messages unicasted to the new user with its individual key and the $l_d(n_t)$ KEKs of its branch, one message with the join notification ($l_d(n_t)$ indices of the affected branch) multicasted to the current users, one message used to multicast the updated Ω_t among the OG_t and the $|\Omega_t|$ messages multicasted to distribute the updated SKs among the related OGCs. Therefore, the number of keys sent out by an OGC_t in the user join process is determined by the following equations:

$$NK_{J_{unicast}} = l_d(n_t) + 1 \tag{5.24}$$

$$NK_{Jmulticast} = 2|\Omega_t| \tag{5.25}$$

According to the algorithm found in Section 4.2.3.2, the number of messages sent out by an OGC_t in the user leave process is determined by:

$$M_{leave} = (d-1)l_d(n_t) + |\Omega_t| + 2 \tag{5.26}$$

which includes up to $(d-1)l_d(n_t)$ messages employed to send the updated KEKs to the users that cannot derive it, $|\Omega_t|$ messages sent out to distribute the updated SKs among the related OGCs, one message with the leave notification sent to the remaining users and one message to distribute the updated Ω_t among the OG_t.

As for the user leave process only the remaining users are involved, the responsible OGC does not send any message in a unicast way. Thus the number of keys, sent out by an OGC_t in the user leave process is determined by:

$$NK_{Lmulticast} = (d-1)l_d(n_t) + 2|\Omega_t| \tag{5.27}$$

When a user switches from OG_y to OG_z, the involved OGCs have to update the SKs in $\Omega_y \cap (\Omega_z \triangle \Omega_y)$ and $\Omega_z \cap (\Omega_z \triangle \Omega_y)$, respectively. In addition, the involved OGCs exchange extra information to ensure the forward secrecy in the case where a user will not finish the switch process. As the user switch process involves the join and leave processes, from (5.23) and (5.26) we can calculate the number of messages sent out by the OGC_y and OGC_z as:

$$M_{switch} = (d-1)l_d(n_y) + |\Omega_y \cap (\Omega_z \triangle \Omega_y)| + |\Omega_z \cap (\Omega_z \triangle \Omega_y)| + 9 \tag{5.28}$$

which inlcudes the messages related to the join and leave events and the three messages exchanged by the involved OGCs to ensure the forward secrecy.

It is known that in the user switch process the two basic operations are involved: join and leave; however in the DMM-MSKMS the user switch process is performed by

two OGCs. An OGC updates the affected KEKs acording to the user join process while the other OGC updates the affected KEKs following the user leave process and both OGCs only update the SKs that do not have in common. For this reason the number of keys unicasted by an OGC is bounded by the number of keys involved in the user join process, while the number of keys multicasted by an OGC is bounded by the number of keys involved in the user leave process, given by equations 5.24 and 5.27.

Assuming the worst case, when a user switch process involves an OG related to all the s sessions, and an OG related only with one session, we have that the number of messages that are necessary to update the SKs is $|\Omega_y \cap (\Omega_z \triangle \Omega_y)| + |\Omega_z \cap (\Omega_z \triangle \Omega_y)| = s - 1$. Thus, from (5.28) the number of messages sent out by the involved OGCs in the user switch process is given by:

$$M_{switch} = (d-1)l_d(n_y) + s + 8 \tag{5.29}$$

Furthermore, if we assume that all the OGs in the system have the same number of users, $n_y = n_0$ and $n_0 \to \infty$, as s is a fixed value throughout the communication process and $s \ll n_0$ using (5.29), we can calculate the rekey overhead of O-Level asymptotically as:

$$M_{O-Level} \sim O(d \log_d(n_0)) \tag{5.30}$$

5.2.3 Comparison

In this section we compare the DMM-MSKMS and DKMS because both schemes use an independent server to handle a group, forming a similar key/user organization. The comparison is focused on two measures: the storage overhead and the communication overhead. For the communication overhead, we only compare the costs related to the O-Level because the costs related to the management of the servers in DKMS are not defined.

In Table 5.2 we summarize the measurements, which are expressed in bits and are based on the results given in [RLK05] and the results obtained in Sections 5.2.1 and

5.2.2. In Table 5.2, S_K denotes the size of the KEK and the TEK, while S_{SK} denotes size of the SK. As we mentioned above, if we use a cryptosystem with KEKs of 128 bits, the size of the SKs should be $O(192)$ bits.

For a fair comparison, we compare DKMS and DMM-MSKMS breaking down the communication cost in the indivual costs of join, leave and switch processes. Moreover, the communication costs, shown in Table 5.2, include the size of the notification messages sent out by an OGC in the DMM-MSKMS.

Table 5.2: Performance comparison for storage and communication overheads

		DKMS	DMM-MSKMS
Storage costs			
	Servers	$(\frac{d}{d-1}n_t + \lvert\Omega_t\rvert + 1)S_K$	$\frac{d}{d-1}n_t S_K + 2\lvert\Omega_t\rvert S_{SK}$
	User	$(\log_d(n_t) + \lvert\Omega_t\rvert)S_K$	$\log_d(n_t)S_K + \lvert\Omega_t\rvert S_{SK}$
Communication costs			
User join	Unicast	$(\log_d(n_t) + \lvert\Omega_t\rvert)S_K$	$\log_d(n_t)S_K + 1$
	Multicast	$d\log_d(n_t) + m(s_r)$	$\log_d(n_t) + 2\lvert\Omega_t\rvert S_{SK}$
User leave	Unicast	0	0
	Multicast	$(d\log_d(n_t) + m(s_r))S_K$	$(d-1)\log_d(n_t)S_K + 2\lvert\Omega_t\rvert S_{SK}$
User switch	Unicast	$(\log_d(n_t) + \lvert\Omega_t\rvert)S_K$	$\log_d(n_t)S_K + 1$
	Multicast	$(d\log_d(n_t) + m(s_r))S_K$	$(d-1)\log_d(n_t)S_K + 2\lvert\Omega_t\rvert S_{SK}$

In addition to the costs, we note that both DKMS as DMM-MSKMS lack of the *1 affects n* phenomenon, unlike some of the solutions related to multi-group key management exposed in Chapter 3.

In Table 5.2 we can notice that in DKMS as in MM-MSKMS, the communication overhead depends on the number of groups in which some keys are involved ($m(s_r)$). This is because in DKMS, the session keys are agreed on by the involved servers, increasing the cost linearly to the number of groups. Therefore, DKMS is not scalable in the number of groups.

In DMM-MSKMS, the involved servers previously agree on a key (BK) to encrypt each session key, so the rekey overhead depends on the number of sessions and not on

the number of groups. Furthermore, as each OGC handles only one tree, the number of messages needed to distribute the compromised keys is in the order of the logarithm of the number of users joined at OG (n_t). Therefore, DMM-MSKMS is scalable in the number of users and in the number of groups.

Another important characteristic is that DMM-MSKMS is designed for users who work as transceivers in an n to n communication, unlike DKMS, which is designed for users who work only as receivers in a 1 to n communication. Thus, in the DMM-MSKMS, users can exchange streams among them and are not only limited to decrypting a specific stream.

Chapter 6

Conclusions and future work

6.1 Summary

This thesis has presented an efficient multi-session key management mechanism for dynamic multimedia group communication, these mechanism is characterized by the use of an independent key per ciphered packet. Unlike other proposed solutions, in our solutions users not only can decrypt a specific stream, but can also exchange streams among them in an n to n communication.

According to the proposed methodology, we developed two mechanisms: the MM-MSKMS and the DMM-MSKMS.

The MM-MSKMS was based on a centralized approach. The MM-MSKMS architecture exploits the overlap of the user sessions in order to reduce the redundancy in key distribution. This architecture organizes the users according to their memberships forming groups with users who have the same memberships. The proposed architecture uses two key generation strategies: a key derivation technique to reduce the rekey overhead and a pseudorandom number generator that allows the users to generate independent key per ciphered packet. This last characteristic enables the system to support the delay, loss or transposition of packets.

The MM-MSKMS is scalable in the number of users, but is not scalable in the number of groups because the number of messages that the KDC needs to distribute the session keys linearly grows to the number of groups. Nevertheless, it is scalable in the number of users since the number of messages used to handle a user membership changes in the

order of the logarithm with respect to the number of users. Therefore, MM-MSKMS is an efficient mechanism for environments where there is a high scale associativity of users and there are a few sessions.

The DMM-MSKMS is an extension of MM-MSKMS and it is the main contribution of this thesis. With the DMM-MSKMS we solve some problems involved in the MM-MSKMS, which are: a single point of failure by concentrating the whole key management in the KDC, and the lack of scalability in the number of groups. Such problems were solved, using two levels of key management: a level to manage the sessions (S-Level) and a level to manage the overlapping in the memberships of the users (O-Level). In the S-Level, several servers agree on some keys to manage the session keys, allowing that the number of messages required to distribute the session keys depends on the number of sessions and not on the number of groups. In the O-Level each server, member of the S-Level, manages a group with users who have the same memberships. With such organization, the key management task is distributed among several entities, eliminating the single point of failure, which is present in the MM-MSKMS. Furthermore, the O-Level inherits the skill of the MM-MSKMS to handle the changes in memberships using a number of messages that is in the order of the logarithm with respect to the number of users. Therefore, this solution is appropriate for environments where there is a high scale associativity of users, and a large number of groups and sessions.

6.2 Future work

Some aspects, observed during the development of the proposed mechanisms, deserve further studies. However, we consider tree main aspects for future work:

- **Design of policies for OGs allocation.** In current DMM-MSKMS, we have not defined any way so that a system can be initiated in a centralized scheme and dynamically becomes a decentralized system. One way that we see is that such skill can be achieved in a manner that the system can be initialized by an OGC

acting as a KDC, using the MM-MSKMS, and following certain criteria, such the OGC can delegate to other servers the responsibility of handling some OGs.

- **Adaptative KEK-tree degree.** In some environments, there are groups with different association levels. For example, if there are some users in an interactive meeting and some of those users are subscribed in a forum, the number of users in both applications will be different. Since the users in the forum not necessarily have to be present every moment, the number of users online may be lesser. In such case having KEK-trees of the same degree may become inefficient. For this reason, it is necessary to look into a way to dynamically establish the degree of trees. The degree of trees may be established based on groups membership dynamism.

- **Verification of the performance in real-time environments.** In environments where the information has a short lifetime, long delays in information exchange, may affect the efficiency of a system. For this reason, emulation or simulation of our mechanism would verify whether our solution can be applied to real-time systems.

Bibliography

[BBB04] Andre Boumso, Boucif Amar Bensaber, and Ismail Biskri. Gakap, multicast key agreement protocol for ad hoc networks based on group activity probability. In *LCN '04: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, pages 700–704, Washington, DC, USA, 2004. IEEE Computer Society.

[BBS86] L Blum, M Blum, and M Shub. A simple unpredictable pseudo random number generator. *SIAM J. Comput.*, 15(2):364–383, 1986.

[CBB04] Yacine Challal, Hatem Bettahar, and Abdelmadjid Bouabdallah. Sakm: a scalable and adaptive key management approach for multicast communications. *SIGCOMM Comput. Commun. Rev.*, 34(2):55–70, 2004.

[CQN+02] Hao-hua Chu, Lintian Qiao, Klara Nahrstedt, Hua Wang, and Ritesh Jain. A secure multicast protocol with copyright protection. *SIGCOMM Comput. Commun. Rev.*, 32(2):42–60, 2002.

[CS05] Yacine Challal and Hamida Seba. Group key management protocols: A novel taxonomy, 2005.

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[DML04] Yongdong Wu Di Ma, Robert H. Deng and Tieyan Li. Dynamic access control for multi-privileged group communications. In *6th International*

Conference on Information and Communications Security (ICICS 2004), Lecture Notes in Computer Science (LNCS) 3269, pages 508–519, Berlin, Heidelberg, 2004. Springer-Verlag.

[DW99] R. Agee D. Wallner, E. Harder. Network working group d. wallner request for comments: 2627 e. harder category: Informational r. agee national security agency june 1999 key management for multicast: Issues and architectures, 199.

[FK04] Borko Furht and Darko Kirovski. *Multimedia Security Handbook*. CRC Press, Inc., Boca Raton, FL, USA, 2004.

[FLS97] Alan Fekete, Nancy Lynch, and Alex Shvartsman. Specifying and using a partitionable group communication service. In *ACM Transactions on Computer Systems*, pages 53–62, 1997.

[FS03] Niels Ferguson and Bruce Schneier. *Practical Cryptography*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[GLLC05] Qijun Gu, Peng Liu, Wang-Chien Lee, and Chao-Hsien Chu. Ktr: an efficient key management scheme for air access control. In *MOBIQUITOUS '05: Proceedings of the The Second Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services*, pages 499–501, Washington, DC, USA, 2005. IEEE Computer Society.

[HDP03] Thomas Hardjono, Lakshminath R. Dondeti, and Radia Perlman. *Multicast and Group Security*. Artech House, Inc., Norwood, MA, USA, 2003.

[HM97a] H. Harney and C. Muckenhirn. Group key management protocol (gkmp) architecture. *RFC 2094*, 1997.

[HM97b] H. Harney and C. Muckenhirn. Group key management protocol (gkmp) specification. *RFC 2093*, 1997.

[HM03] Jyh-How Huang and Shivakant Mishra. Mykil: A highly scalable key distribution protocol for large group multicast. In *IEEE GLOBECOM 2003*, Washington, DC, USA, 2003. IEEE Computer Society.

[KPT00] Yongdae Kim, Adrian Perrig, and Gene Tsudik. Simple and fault-tolerant key agreement for dynamic collaborative groups. In *CCS '00: Proceedings of the 7th ACM conference on Computer and communications security*, pages 235–244, New York, NY, USA, 2000. ACM.

[LHLL09] Jen-Chiun Lin, Kuo-Hsuan Huang, Feipei Lai, and Hung-Chang Lee. Secure and efficient group key management with shared key derivation. *Comput. Stand. Interfaces*, 31(1):192–208, 2009.

[LLL05] Jen-Chiun Lin, Feipei Lai, and Hung-Chang Lee. Efficient group key management protocol with one-way key derivation. In *LCN '05: Proceedings of the The IEEE Conference on Local Computer Networks 30th Anniversary*, pages 336–343, Washington, DC, USA, 2005. IEEE Computer Society.

[Mit97] Suvo Mittra. Iolus: a framework for scalable secure multicasting. *SIGCOMM Comput. Commun. Rev.*, 27(4):277–288, 1997.

[MJMR99] Josyula R. Rao Matthew J. Moyer and Pankaj Rohatgi. A survey of security issues in multicast communications. *IEEE Network*, 13(6):12–23, 1999.

[MVO96] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1996.

[NPKKI07] Alireza Nemaney Pour, Kazuya Kumekawa, Toshihiko Kato, and Shuichi Itoh. A hierarchical group key management scheme for secure multicast increasing efficiency of key distribution in leave operation. *Comput. Netw.*, 51(17):4727–4743, 2007.

[Per99] Adrian Perrig. Efficient collaborative key management protocols for secure autonomous group communication. In *In International Workshop on*

Cryptographic Techniques and E-Commerce (CrypTEC â99, pages 192–202, 1999.

[RH03] Sandro Rafaeli and David Hutchison. A survey of key management for secure group communication. *ACM Comput. Surv.*, 35(3):309–329, 2003.

[RLK05] Jie Li Ruidong Li and Hisao Kameda. Dynamic access control for multi-privileged group communications. In *International Conference on Computer Network and Mobile Computing 2005 (ICCNMC 2005), Lecture Notes in Computer Science (LNCS) 3619*, pages 539–548, Berlin, Heidelberg, 2005. Springer-Verlag.

[SL03] Yan Sun and K. J. R. Liu. Multi-layer key management for secure multimedia multicast communications. In *ICME '03: Proceedings of the 2003 International Conference on Multimedia and Expo*, pages 205–208, Washington, DC, USA, 2003. IEEE Computer Society.

[SL04] Yan Sun and K. J. R. Liu. Scalable hierarchical access control in secure group communications. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, pages 1296– 1306, Washington, DC, USA, 2004. IEEE Computer Society.

[SM03] Alan T. Sherman and David A. McGrew. Key establishment in large dynamic groups using one-way function trees. *IEEE Trans. Softw. Eng.*, 29(5):444–458, 2003.

[STW96] Michael Steiner, Gene Tsudik, and Michael Waidner. Diffie-hellman key distribution extended to group communication. In *CCS '96: Proceedings of the 3rd ACM conference on Computer and communications security*, pages 31–37, New York, NY, USA, 1996. ACM.

[SWM+01] Dapeng Wu Student, Dapeng Wu, Student Member, Yiwei Thomas Hou, Wenwu Zhu, Ya qin Zhang, Jon M. Peha, and Senior Member. Streaming

video over the internet: Approaches and directions. *IEEE Transactions on Circuits and Systems for Video Technology*, 11:282–300, 2001.

[WCS+99] Marcel Waldvogel, Germano Caronni, Dan Sun, Nathalie Weiler, Bernhard Plattner, and Student Member. The versakey framework: Versatile group key management. *IEEE Journal on Selected Areas in Communications*, 17:1614–1631, 1999.

[WGL98] Chung Kei Wong, Mohamed Gouda, and Simon S. Lam. Secure group communications using key graphs. In *IEEE/ACM Transactions on Networking*, pages 68–79, 1998.

[WOCG07] Guojun Wang, Jie Ouyang, Hsiao-Hwa Chen, and Minyi Guo. Efficient group key management for multi-privileged groups. *Comput. Commun.*, 30(11-12):2497–2509, 2007.

[YCS05] Abdelmadjid Bouabdallah Yacine Challal and Hamida Seba. A taxonomy of group key management protocols: Issues and solutions. In *Proceedings of World Academy of Science, Engineering and Technology*, volume 6, pages 5–19, Oslo, Norway, 2005.