Group Key Management: Algorithms, Benchmarking, and Reconfigurable Architectures

Vom Fachbereich Informatik
der Technischen Universität Darmstadt
genehmigte

DISSERTATION

zur Erlangung des akademischen Grades eines
Doktor-Ingenieurs (Dr.-Ing.)

von

Dipl.-Ing. Abdulhadi Shoufan

aus Homs-Syrien

Referenten der Arbeit:	**Prof. Dr.-Ing. Sorin A. Huss**
	Prof. Dr.-Ing. Klaus D. Müller-Glaser
Tag der Einreichung:	**06.02.2007**
Tag der mündlichen Prüfung:	**30.03.2007**

Darmstadt 2007
D17

To Mother & Father
and to
Madieha

Zusammenfassung

IP Multicast ist eine effiziente Lösung für Gruppenkommunikation über das Internet. Sowohl die Serverressourcen als auch die Netzbandbreite werden durch diese neue Technologie entlastet. Spezielle Probleme und Herausforderungen entstehen allerdings, wenn die Gruppenkommunikation Sicherheitsanforderungen erfüllen muss. Ein wichtiger Aspekt bezieht sich auf die gemeinsame Nutzung eines Kommunikationsschlüssels. Dieser Schlüssel muss nämlich jedes Mal aktualisiert und verteilt werden, wenn die Gruppenzusammensetzung sich ändert. Dieser Prozess, der als *Rekeying* bezeichnet wird, wirft ein *Skalierbarkeitsproblem* für große dynamische Gruppen auf: Das Rekeying basiert auf rechenaufwendigen kryptographischen Operationen und erfordert die Übertragung von Rekeyingnachrichten. Das Skalierbarkeitsproblem zeichnet sich daher durch ein Rechenoverhead auf der Serverseite und durch ein Kommunikationsoverhead im Übertragungsnetzwerk aus.

In der Literatur wurden zahlreiche Architekturen, Algorithmen und Protokolle publiziert, die dieses Skalierbarkeitsproblem adressieren. Lösungen zur Optimierung der Rekeyingperformanz konzentrieren sich auf die Minimierung der Anzahl der erforderlichen kryptographischen Operationen und somit der Länge der Rekeyingnachrichten. Eine akzeptierte Strategie zur Reduzierung der Rekeyingkosten verwendet eine Stapelverarbeitung von Rekeyinganfragen, die innerhalb eines festgelegten Rekeyingintervalls gesammelt werden. Eine Spezifizierung der maximalen Länge dieses Intervalls fehlt jedoch in der Literatur bisher. Zu lange Rekeyingintervalle verursachen längere Wartezeiten für neue Mitglieder und längere Zugriffszeiten für Verlassende. Folglich ist die Stapelverarbeitung von Rekeyinganfragen stets verbunden mit einem Verlust an Dienstgüte einerseits und an der Systemsicherheit andererseits. Aufgrund der Neuigkeit und der Komplexität dieses Forschungsgebiets vermissen die präsentierten Lösungen eine einheitliche Methode zur Abschätzung der Rekeyingperformanz. In den meisten Fällen wird dadurch eine Evaluierung von verschiedenen Rekeyingalgorithmen enorm erschwert.

Die vorliegende Dissertation erörtert die oben erwähnten drei Probleme des Gruppenrekeying. Erstens wird eine Methode präsentiert, die die Probleme der Dienstgüte und der Sicherheit im Stapelrekeying adressiert. Diese Methode wird als *Ereignisgesteuertes Stapelrekeying* bezeichnet. Zweitens wird ein *Rekeyingbenchmark* eingeführt, der einen einheitlichen zuverlässigen Weg zur Abschätzung der Rekeyingperformanz verschiedener Rekeyingalgorithmen darstellt. Drittens werden drei innovative Hardware- und HW/SW-Architekturen zur Optimierung der Rekeyingperformanz präsentiert. Im

Unterschied zu bisherigen Lösungen wird die Rekeyingperformanz durch diese Architekturen nicht nur auf der algorithmischen Rekeyingebene optimiert, sondern auf der kyptographischen Ebene und auf der Plattformebene. Die neuen Architekturen werden als der *Real-Time Rekeying Processor*, der *Batch Rekeying Processor* und der *High-Flexibility Rekeying Processor* bezeichnet.

Preface

IP multicast is an efficient solution for group communication over the Internet, as both the sender resources and the network bandwidth are relieved with the aid of this emerging technology. However, this superiority suffers, when the group communication must fulfill some security requirements. An essential issue relates to sharing the communication key. Particularly, this key must be updated and securely distributed, every time the group membership changes. This process, which is denoted as *group rekeying*, raises a scalability problem in large dynamic groups: Rekeying is based on computationally extensive cryptographic operations and on the dissemination of rekeying messages. Thus, the scalability problem presents itself by a computation overhead on both the sender and the receiver sides, and by a communication overhead in the network.

Numerous architectures, algorithms, and protocols have been proposed in the literature to cope with this scalability problem. Related work on *optimizing rekeying performance* mostly concentrates on minimizing the number of required cryptographic operations and thus the length of the rekeying message. An accepted strategy to reduce rekeying costs utilizes batch processing of rekeying requests, which are summed up during a rekeying interval. However, a specification of the maximal length of this rekeying interval is not provided, so far. Too long rekeying intervals cause longer waiting times for new members and longer access times for removed ones. Consequently, a *problem of QoS and security* is associated with batch rekeying. Because of its novelty and complexity, the work on rekeying optimization lacks a unified way to estimate rekeying performance. In most cases, therefore, an *evaluation of different algorithms* is impossible.

The presented dissertation addresses the above three problems of group rekeying. Firstly, an approach, denoted as *Even-Driven Batch Rekeying*, is proposed to tackle the QoS and security problems caused by long rekeying intervals in batch rekeying. Secondly, to enable a reliable evaluation of rekeying algorithms, a *Rekeying Benchmark* is introduced, which provides a unified way to estimate the performance of different rekeying algorithms on the system level. Thirdly, three novel hardware and hardware/software architectures are presented for optimizing the rekeying performance. In contrast to related work, these architectures, denoted as the *Real-Time Rekeying Processor*, the *Batch Rekeying Processor*, and the *High-Flexibility Rekeying Processor*, optimize rekeying not only on the rekeying algorithm level, but also on the cryptography and platform levels.

Acknowledgments

This work developed during my activity as scientific assistant in the Integrated Circuits and Systems Laboratory at the University of Technology Darmstadt. Though a doctorial thesis is related to its author, I can not forget the great support of other people, estimable people.

A grateful word of thanks to

Prof. Dr. Sorin A. Huss, my advisor. Only with his directing, without to enforce, his advices, without to restrict, and with his energy, without to disturb, it was possible to accomplish this work in the best working atmosphere and in the right time.

Prof. Dr. Klaus D. Mueller-Glaser, my co-referee for taking the time and making the effort to read this work and comment it constructively.

All my colleagues for the nice time and the friendly cooperation, Maxim Anikeev, Tom Assmuth, Markus Ernst, Prih Hastono, Stephan Hermanns, Dan Honciuc, Elisabeth Hudson, Adeel Israr, Michael Jung, Andreas Kuehn, Joseph Laschgari, Ralf Laue, Steffen Klupsch, Stephan Klaus, Tim Sander, Maria Tiedemann, Juergen Weber, Song Yuan, and Kaiping Zeng.

All the students, who substantially supported the realizations presented in this thesis and other research work, Mujtaba Abrooy, Peter Bungert, Murtuza Cutleriwala, Zhaoming Dai, Torsten Hahn, Nico Hubert, Marcus Lindner, Dominik Litzinger, Felix Madlener, Joana Otetelisanu, Sven Rettig, Abdeloahid Tadoo, Bieanvenu Tatsi, and Tobias Teichner.

My teacher and role model Zaki Ramdoun.

My mother and my father for their love, generosity, patience, and stamina. Despite the modest means, they managed to bring four of five sons and the only daughter to complete study.

My wife Madieha for the wonderful years with her, for her advices and her comments regarding this work.

Syria, which produced me, and Germany, which refined me.

Content

1 Introduction

1.1 Overview

This chapter introduces to the scope of the presented dissertation, specifies its objectives, and outlines it. Section 1.2 illustrates some basics of IP multicast and its advantages. Section 1.3 briefly highlights the role of security in information systems and explains cryptographic methods as essential security means. Section 1.4 represents the problems resulting from applying security models of unicast communication to IP multicast. An essential issue relates to the group key management which is detailed as well in this section with an overview of related work. Section 1.5 lastly depicts the three main objectives of this work and outlines the methods and solutions to attain these objectives. The structure of the dissertation is provided in this concluding section, too.

1.2 IP Multicast

Various Internet applications rely on group communication in either one-to-many or in many-to-many mode. One way to support this communication relates to sending data packets from the sender to all receivers using the well-known unicast technique. However, this approach does not scale and results in overloading both the sender resources and the network. **Figure 1.1** represents an example for this communication mode, where the hosts H1 to H5 build a group and H1 is trying to deliver a data packet to other group members. As can be seen, the data packet is duplicated 17 times before it arrives the destination hosts. Note that the packets are transferred over the shortest path to destinations. This is achieved by the different network routers (R) which execute a routing algorithm based on the line cost values depicted in this figure. Some solutions hand over the task of data duplication from the sender host to a dedicated server, which is denoted as multipoint control unit (MCU) in the scope of video conferencing. By this means the sender resources are relieved but the network traffic does not ease as data packets are still transmitted in multiple copies.

A scalable group communication demands the dissemination of just one copy of the data packet over the network. IP Multicast represents a technique that fulfills this requirement as depicted in **Figure 1.2**. Based on its special address class a multicast data packet is recognized by multicast routers (MR), which duplicate this packet as necessary. Multicast IP-addresses belong to class-D, which assigns 28 bits (from a total of 32 bits) to identify different groups. Thus, IP multicast can support up to 250 million different groups. When

investigating this routing technology, two questions arise. Firstly, how do multicast routers learn the hosts belonging to some group? This question results from the fact that class-D addresses do not contain a field, which specifies the subnet to which a host belongs, as it is the case in other IP-address classes for unicast communication. Secondly, how does routing perform? These two questions are first answered in the next section briefly. Afterwards, some application scenarios of IP multicast are described and the associated difficulties in employing this communication technology are outlined.

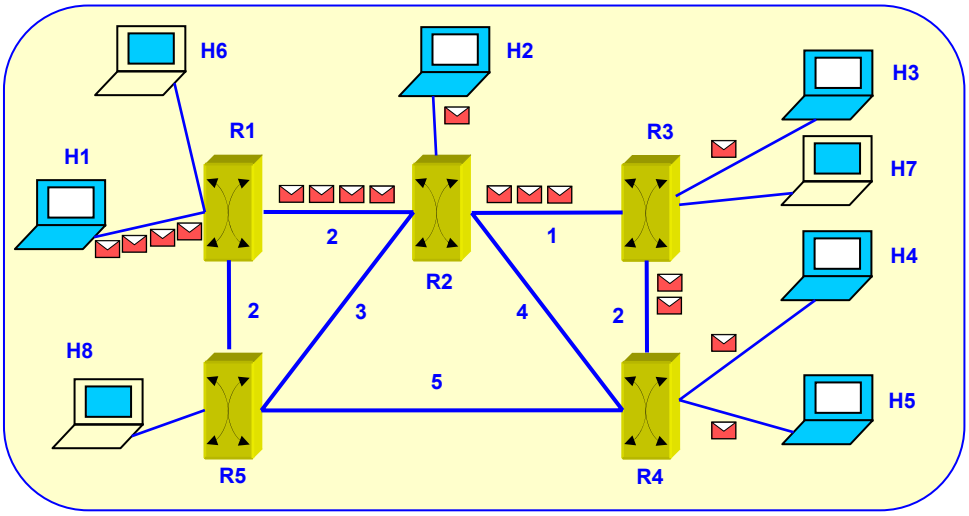


Figure 1.1. Example for unicast-based group communication

1.2.1 IP Multicast Protocols

For an efficient multicast routing each router must be kept informed about all the running multicast groups and about the belonging of other multicast routers to these groups. A multicast router is said to belong to some multicast group, if it has at least one host in its local network, which is a member of this group. This information is saved in form of a *multicast address table* (MAT). In the example shown in **Figure 1.2** the multicast routers MR1 to MR5 manage the MAT depicted in **Table 1.1**. In this example hosts H1 to H5 and H6 to H8 build two multicast groups, which are called A and B, respectively. The first entry in this table, for instance, indicates that the router MR1 has one or more hosts in its subnet, which are members in the multicast groups A and B. Remember that a multicast group is identified by its IP address. The establishment of the MAT is supported by two multicast protocols:

1. The *Internet group management protocol* (IGMP) which is executed between each multicast router and the hosts in its local network to check which of these hosts belong to which multicast group.
2. A routing protocol, e.g. the *distance vector multicast routing protocol* (DVMRP), which enables multicast routers to exchange data obtained from executing the IGMP

protocol, so that all routers have the up-to-date MAT. Note that a routing protocol is originally used to route useful data from a sender to all group members.

Besides the MAT, each multicast router manages a *routing table* (RT). In the DVMRP protocol, for instance, these tables contain information on the distance (D) to each other router and how to reach it. **Table 1.2** depicts the RTs for the example of **Figure 1.2**. The left-most table relates to MR1 and states, for instance, that the shortest path from MR1 to MR4 has a distance of 5 and that this router can be accessed through MR2.

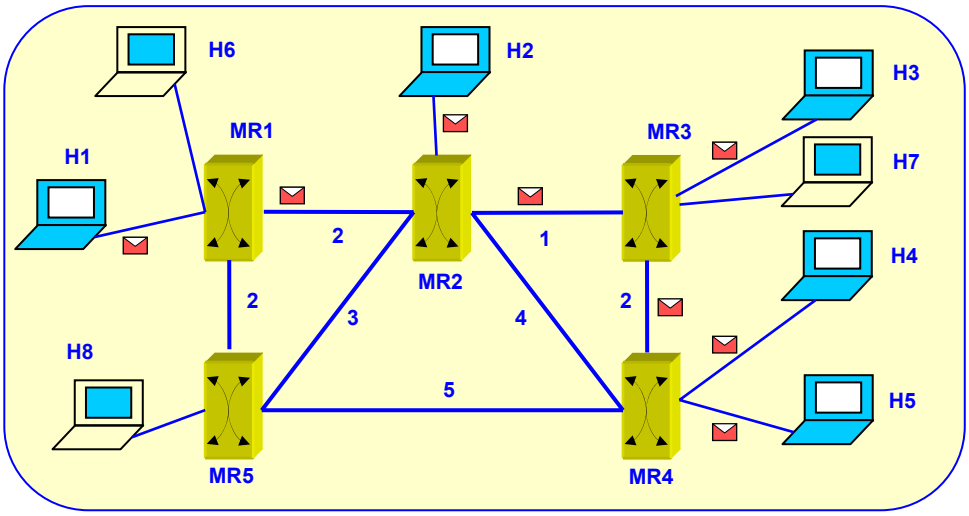


Figure 1.2. Example for multicast-based group communication

Both the IGMP and the DVMRP belong to the IP layer of the Internet layer model and were first defined as RFC (request for comment) in the late eighties. Current specifications of these protocols can be found as Internet Drafts in [Ca97] and [Pu98], respectively.

Table 1.1. MAT for the example of **Figure 1.2**

Multicast Router	Multicast address
MR1	A, B
MR2	A
MR3	A, B
MR4	A
MR5	B

Table 1.2. Routing tables for the example of **Figure 1.2**

MR1

MR	MR	D
1	1	0
2	2	2
3	2	3
4	**2**	**5**
5	5	2

MR2

MR	MR	D
1	1	2
2	2	0
3	3	1
4	3	3
5	5	3

MR3

MR	MR	D
1	2	3
2	2	1
3	3	0
4	4	2
5	2	4

MR4

MR	MR	D
1	3	5
2	3	3
3	3	2
4	4	0
5	5	5

MR5

MR	MR	D
1	1	2
2	2	3
3	2	4
4	4	5
5	5	0

1.2.2 IP Multicast Applications

The transport control protocol TCP of the transport layer supports only point-to-point services. Therefore, multicast applications must utilize the user datagram protocol UDP, which offers unreliable connectionless services on the transport layer. However, this does not restrict multicast to error-tolerant applications such as multimedia streaming. Reliable multicast applications can be supported, for example, by adding a special layer between the application and the transport layers, which emulates the TCP. Accordingly, multicast applications can be classified into four categories [Mi99]:

1. **Real-time multimedia applications,** e.g. video conferencing, internet radio, and multimedia events. These applications are error-tolerant but demand low timing jitter characteristics for correct synchronization.
2. **Real-time data applications,** such as the distribution of stock quotes and interactive gaming. The requirements of scalability and reliability for these applications differ depending on the particular data. Text news, for instance, must be delivered error-free to very large groups as a rule. Some latency in the transmission of these data, however, can be tolerated.
3. **Non real-time multimedia applications,** e.g. remote class rooms with high reliability but low or moderate scalability requirements.
4. **Non real-time data applications,** such as database replication and software distribution, with both high reliability and high scalability demands.

Despite its scalability and usability in a vast number of current and new Internet applications, IP multicast technology is not yet exploited to large extent. Several economic and technical reasons are responsible regarding both private and public networks. Some examples for the deployment barriers of IP multicast in private networks relate to the limited application software and administration tools supporting this technology. One essential obstacle in the usage of this communication technique over the public Internet is attributed to the fact that not all current IP routers are multicast-able.

For further reading on networking and IP multicast the following publications are recommended [Ha01], [Ta00], [Mi99], and [Go99].

1.3 Information Security

1.3.1 Threats, Requirements, and Solutions

An information system is regarded as secure if it ensures the delivery of the right information to the right party at the right time. The meaning of the terms *right information*, *right party*, and *right time* in the context of security is illustrated as follows. To ensure the delivery of right information, the representing data must be able to be checked on their integrity and their source. *Data integrity* means that no manipulation has been performed on these data en route. *Data source authentication* guarantees that data stem from the source claiming or denying to be the sender. The protection against denying is known as *non-repudiation* property of a secure system. Supplying information to the right party implicates the hiding of this information against unauthorized parties which requires applying mechanisms for *data confidentiality*. Providing data with integrity, authentication, and confidentiality is only reasonable if these data are available at the right time. *Data availability* is regarded as security goal, since several attack schemes aim this property.

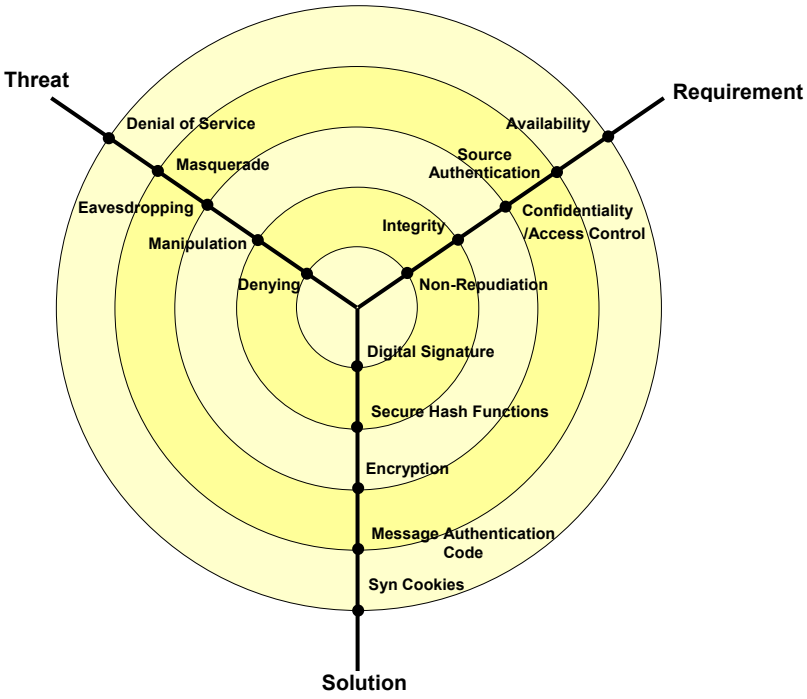


Figure 1.3. Visualization of security requirements, threats, and solutions

The decision on required security objectives for an information system and on the measures to attain these goals is a largely sophisticated task, which is defined as a *security policy* for that system. The specification of a security policy relies on an in-depth analysis of the criticality of the information to be exchanged, the data delivery system, and the threats it is exposed to. Receivers of Pay-TV data, for instance, do not need to verify that video

contents originate from the Pay-TV provider, as a rule. Thus, data source authentication is not a requirement in this case. In contrast, for a commercial site it is indispensable to verify the client identity before executing its order. Only by applying a non-repudiation strategy a possible denying can be disproved. Inspired by the graphical representation of Gajski's diagram [Ga92], the relation between security requirements, well-known threats, and possible countermeasures can be depicted schematically by means of a Y-Diagram as illustrated in **Figure 1.3**. The hierarchical representation reflects to large extent the dependency of the different security requirements. In this respect, data availability is an essential aspect, which all other requirements are based upon. The non-repudiation property, however, assumes data availability, authentication, and integrity. In some cases, furthermore, non-repudiation largely relies on data confidentiality.

1.3.2 Cryptography

Regardless of solutions to the availability requirement, all other countermeasures on the solution axis of **Figure 1.3** rely on cryptographic operations. Cryptography is defined in [Me96] as *the study of mathematical techniques related to aspects of information security such as confidentiality, data integrity, entity authentication, and data origin authentication.* The mathematical techniques are applied to data representing the information to be protected from eavesdropping, manipulation, etc. All cryptographic methods rely on using some information denoted as *key*. The way how to exchange and manage this key represents the most important criterion to classify cryptographic algorithms. According to this criterion two main categories are present: the symmetric-key and the public-key cryptography. In *symmetric-key cryptography* both communication parties use the same key, denoted as *symmetric key*, to apply cryptographic operations to encipher or decipher data. Cryptographic enciphering and deciphering are referred to as *encryption* and *decryption*, respectively. The Data Encryption Standard (DES) [Ni81] and the Advanced Encryption Standard (AES) [Ni01] are the most known examples for this cryptography class. Besides encryption and decryption for the purpose of confidentiality, symmetric-key cryptography may be employed to realize secure hash functions and message authentication code, which support data integrity and source authentication, respectively, see **Figure 1.3.** For the functionality of symmetric-key cryptography, however, a secret key must be agreed and delivered over a secure channel. This hard requirement is avoided by the *public-key cryptography*, which exploits two different keys for encryption and decryption. These keys are denoted as the *public* and the *private key*. For a communication party A to send an encrypted message to another party B, it uses the public-key of B. Getting the encrypted message, B can decrypt it with its private key. The idea of public-key cryptography was first published in [Di76]. RSA [Ri78], ElGamal [El85], and Elliptic Curve Cryptography (ECC) [Ko87, Mi86] represent the most known public-key cryptosystems. As the mathematically inverse function of encryption, decryption is based on a relation between the private and public keys. The security of public-key cryptography is based on the fact that this relation is so complex that an extraction of the private key from the public one is impossible during defensible time. This is realized by using complex mathematical problems to generate key pairs such as the factorization of large prime numbers, e.g. in the case of RSA, or the determination of the discrete logarithm, e.g. in the cases of ElGamal and ECC. The security of public-key cryptography does not only rely on the complex relation between the private and the public key, but also on complicating the encryption and

decryption processes themselves. This fact negatively affects the performance of these algorithms and makes them unsuitable to encrypt large data under hard timing constraints. In practice, therefore, public-key cryptography is mainly used to process small amounts of data such as in the following two cases. First, public-key encryption can be used to agree on a symmetric key which can then be employed for secure communication based on symmetric-key encryption. The second application relates to the digitally signing of short data. In this case a sender uses its private key to sign a message and the receiver access the public key of the sender to verify that the message stems from this sender. Digital signing is a measure to ensure non-repudiation as depicted in **Figure 1.3**. Note that public-key cryptography depends on the availability of authentic public keys. The resources, policies, protocols, and procedures demanded to create, distribute, manage, and revoke public keys construct a framework denoted as *public-key infrastructure* (PKI). The task of distributing certified keys in PKI is assigned to an entity which is trusted by every one and referred to as *certification authority*. Consequently, *key management* represents an essential issue in both symmetric-key and public-key cryptography.

In the scope of this work several cryptographic algorithms are utilized. These include the Advance Encryption Standard (AES) as a symmetric-key encryption primitive, the secure Meyer hash function [Ma85], and the Message Authentication Code MAC [Is89]. Both the Meyer hash function and MAC are based on symmetric-key encryption. In addition, for generating secure keys, an algorithm specified in ANSI X9.17 is exploited, which also relies on symmetric-key cipher. For building digital signatures the Elliptic Curve Digital Signature Algorithm (ECDSA) [Ie00] is employed. All these algorithms are briefly described in Chapter 5 with focus on their hardware realization. For further reading on these topics it is referred to the related literature and to text books on security and cryptography, e.g. [Da01], [Ec06], [Me96], and [Sc96].

1.4 Secure Multicast

Various multicast applications demand data delivery under security conditions such as confidentiality, integrity, and data source authentication. A collaborative group of company employees, for instance, may use Internet conferencing to exchange information and need to keep their communication secret. Another example from the multimedia streaming field relates to a multicast Pay-TV scenario, where the access should only be granted to those members, who already have paid the charge. Applying security strategies to multicast communication poses special problems, which are unfamiliar in the one-to-one communication mode. These special issues are attributed to the following properties:

1. Secure communication is based on sharing a secret, i.e. a key, between communicating parties. The more parties learn the secret, the higher the risk to disclose it.
2. In general, multicast groups are large-scale and characterized by highly dynamic membership. To restrict access to authorized members, the group communication key must be changed after every change in the group membership. This increases the complexity of key management.
3. Based on the fact that symmetric key in one-to-one communication is shared by only two partners, data source authentication can be easily verified by encrypting these data

with the shared symmetric key. Unfortunately, this so-called message authentication code (MAC) can not be employed in secure multicast, since the secret key is shared by a large number of members.

4. As mentioned previously, to communicate securely, a security policy must be defined and negotiated between communication parties. Such an agreement in multicast mode is much more sophisticated than in the unicast case, because a security policy must be found which satisfies the requirements of many parties, instead of just two.

1.4.1 Secure Multicast Problem Areas

These and other difficulties in applying security issues to IP multicast have been recognized by research and industry institutions and resulted in establishing the research group *Secure Multicast Group* (SMuG) within the *Internet Research Task Force* (IRTF) in early 1998 [Sm98] and the working group *Multicast Security* (MSEC) within the *Internet Engineering Task Force* (IETF) in early 2000 [Ms00]. A main contribution of SMuG and MSEC relates to identifying the following three problem areas in secure multicast [Ha03]:

1. ***Secure multicast data handling:*** In this problem area secure data transmission including data confidentiality, integrity, and source authentication is treated. Investigating the adaptability of available security protocols to secure multicast presents an essential issue in this problem area. One example relates to the *Multicast Encapsulating Secure Payload* (MESP) as an extension to the known ESP protocol in IP security [Ca00]. Furthermore, in the scope of this area the concept of *group authentication* [Ca99] is introduced as a simple form of *data source authentication*. Some groups are trustful and only need to protect themselves against non-member parties. In this case, group authentication can be employed, which relies on simple symmetric-key cryptography. In contrast, if group members do not trust each other, an exact data source authentication must be used. This kind of authentication in multicast can not be performed on the base of symmetric-key cryptography as in the unicast case. Instead, solutions based on public-key cryptography such as digital signatures must be employed.

2. ***Management of keying material:*** This problem area concerns the generation and distribution of the group communication key. This process is performed both during registration in static groups and after every membership change in dynamic groups. The generation and distribution of a new communication key as an effect of joining new members or removing old ones in dynamic multicast groups is denoted as *Group Rekeying*. This process represents the focus of this dissertation and will be detailed in next section.

3. ***Multicast security policies:*** In secure multicast different members may have different capabilities and responsibilities. One task of the security policy is to define the roles of group members, e.g., as a sender, a receiver, or as a *group controller and key server* (GCKS). Besides, the security policy specifies, for instance, which encryption algorithm should be used for data confidentiality. As policy negotiation among large groups is in practice impossible, a mechanism for policy enforcement by the GCKS must be employed [Di00].

1.4.2 Group Rekeying

Consider the multicast Pay-TV system illustrated in **Figure 1.4**. A *video server* (VS) encrypts video packets with a *group key* k_g and sends them using IP-multicast. Members m_0, m_1, and m_3 aim to buy the service. For this purpose, they connect to a *registration and authentication server* (RAS), pay the charge, and get each an individual *identity key* k_0, k_1, and k_3. In a following step the RAS encrypts the group key k_g with each of the identity keys and sends it to the corresponding member per unicast. The notation $E_{k_a}(k_b)$ in **Figure 1.4** refers to the encryption of key k_b with key k_a. Each authorized member accordingly receives the encrypted k_g, decrypts it with its identity key and gets thereby the group key k_g. Members m_0, m_1, and m_3 in this scenario can now use k_g to decipher the encrypted video data and watch the movie. Member m_2, however, is excluded: he or she can download encrypted movies, but cannot enjoy them. Note that referring to m_2 as a group member relates to the multicast group, not to the secure multicast group. As mentioned before, joining a multicast group is loose and is performed between a host and the multicast router in the corresponding network based on the Internet Group Management Protocol (IGMP). The video provider in this scenario does not know who is currently a group member, therefore, it uses encryption to control access. The way how the registration works and the identity keys are distributed is not in the focus of this work. It is assumed that the task of generation and distribution of identity keys is covered by the RAS.

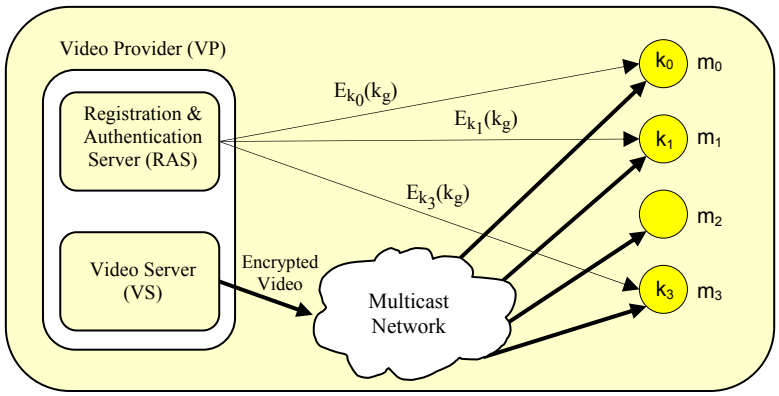


Figure 1.4. Pay-TV: Potential scenario for secure multicast

1.4.2.1 Scalability Problem

If m_2 decides to buy the service later on, then this member registers at the RAS and gets its own identity key k_2. To keep *backward access control*, i.e., to prevent m_2 from decrypting old videos, the RAS generates a new group key k_g^{new}, encrypts it with the current group key k_g, and multicasts it. By this means k_g^{new} becomes available to all current members of the group. In addition, the RAS encrypts k_g^{new} with k_2 and sends it to m_2 per unicast. Consequently, joining a new member in this scheme causes two encryptions on the server side, which is fairly acceptable. In contrast, the process of disjoining a member is highly inefficient. Assume for example that m_1 has to leave the group, because his or her

subscription period ended. To keep *forward access control*, i.e., to prevent m_1 from decrypting future video material, the RAS again has to generate a new group key, but this must NOT be encrypted with the current group key like in the join case. Instead, the RAS encrypts the new group key with each of the identity keys of the remaining members, i.e. with k_0, k_2, and k_3. Thereafter every remaining member gets the k_g^{new} encrypted with its identity key. In other words, disjoining a member from a group having n participants costs a total of $n-1$ encryptions on the server side.

Assuming that there are as much join as disjoin requests, then the average cost of a join/disjoin operation is nearly equal to $n/2$. Obviously, this scheme is not scalable for large groups. In the sequel this rekeying scheme is denoted as *simple rekeying*.

1.4.2.2 Related Work on Group Rekeying

Several solutions have been proposed in literature to cope with the scalability problem in multicast group rekeying. In the Iolus scheme [Mi97] the group is divided into several subgroups. Each subgroup is controlled by a trusted third-party proxy, whereas these proxies are controlled by the group owner. The rekeying within a subgroup occurs as in the simple scheme, which means that Iolus approach becomes unscalable for large subgroups, apart from the drawback of the need of trusted third-party agents. Similar decentralized rekeying schemes are presented in [Do00] and [Bi00]. MARKS [Br99] is a mechanism for efficient key distribution. In this mechanism the group controller knows each member's disjoin time and performs rekeying at fixed time instances, which restricts system dynamics. The Logical Key Hierarchy scheme (LKH) proposed in [Ha99] and [Wo00] allows an efficient rekeying for large groups without a-priori knowledge of group joins or leaves. Many improvements for LKH were proposed to enhance performance. Originally, LKH performs rekeying in real-time mode, i.e. rekeying is executed immediately for each join or disjoin request. One performance improvement relies on processing rekeying requests in batch. In this mode several requests are summed up during a rekeying interval and then processed simultaneously, see, e.g., [Li01], [Ji02], and [Ma04]. Other improvements to LKH deal with tree rebalancing to keep a logarithmic relationship between rekeying costs and group size, see, e.g., [Mo99], [Ra01], [Go04]. Because of its relevance for this work, the next section is dedicated to illustrating the LKH rekeying approach. Another rekeying scheme relies on One-way Function Trees (OFT). This scheme reduces the communication overhead for disjoin rekeying at the expense of additional computations on both the server and the member side [Ba00], [Sh03]. Recently, the interest in group key management exceeded the scope of typical IP multicast. Thus, several solutions for group key management are proposed for Ad-Hoc networks [Se04], [Li06], mobile multicast [Ro06], satellite multicast [Ho04], and wireless cellular networks [Um06].

1.4.2.3 Logical Key Hierarchy

The basic idea behind the LKH is to divide the group into hierarchical subgroups and to provide the members of each subgroup with a shared key, which is called the *help-key*. Consider the example illustrated in **Figure 1.5** for an eight-member group. In this model members m_0 and m_1 build a subgroup with the help-key $k_{0\text{-}1}$, members m_0, m_1, m_2, and m_3 build a larger subgroup, whose help-key is $k_{0\text{-}3}$. All members compose the largest subgroup

with the help-key $k_{0\text{-}7}$. This key represents the group key which is used to encrypt useful data. Consequently, a member now holds several instead of just two keys. These keys are the identity key k_d, which is known only to this member and to the server, the group key k_g known to all group members, and some help-keys $k_{x\text{-}y}$ corresponding to the subgroups, which the member belongs to. Member m_6, for example, has $k_d = k_6$, $k_g = k_{0\text{-}7}$ and two help-keys, which are $k_{6\text{-}7}$ and $k_{4\text{-}7}$.

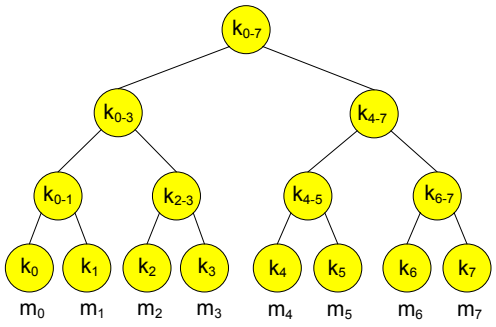


Figure 1.5. LKH example

Disjoin Rekeying

Assume that the member m_2 wants to leave the group. How many encryptions have to be computed by the server to rekey the group?

Except for the identity key all the keys held by m_2 (i.e. $k_{2\text{-}3}$, $k_{0\text{-}3}$ and $k_{0\text{-}7}$) have to be changed. After removing m_2, however, the help-key $k_{2\text{-}3}$ can be destroyed, as this key is only known by one member, i.e. m_3. Therefore, only two keys, $k_{0\text{-}3}^{new}$ and $k_{0\text{-}7}^{new}$ are generated, encrypted, and sent to the remaining members needing these keys. **Figure 1.6** represents the key tree after this processing:

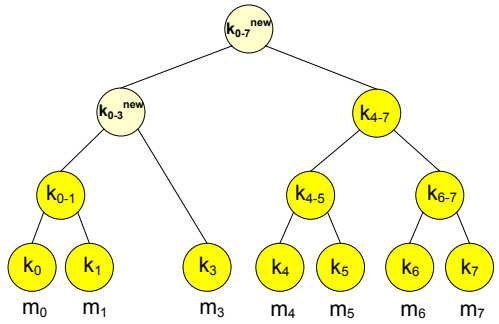


Figure 1.6. LKH example after disjoining m_2

Using the notation $E_{k_a}(k_b)$ to refer to a *rekeying submessage* representing the encryption of the key k_b with the key k_a, the server has to generate the following rekeying submessages in order to disjoin m_2:

$$E_{k_3}(k_{0\text{-}3}^{new}),\ E_{k_{0\text{-}1}}(k_{0\text{-}3}^{new}),\ E_{k_{0\text{-}3}^{new}}(k_{0\text{-}7}^{new}),\ E_{k_{4\text{-}7}}(k_{0\text{-}7}^{new})$$

A *Rekeying Message* is composed of all rekeying submessages and other related information, e.g., the *rekeying submessage identification* illustrated later on in this work.

The server has to compute just 4 encryptions to rekey the group. In contrast, in the simple scheme a total of 7 encryptions would be necessary. The gain of LKH becomes more obvious in the case of large group sizes. **Table 1.3** details this comparison.

Join Rekeying

Assume now that another member will be joined at the tree position of m_2. The new member will be called m_2, too. How many encryptions have to be performed by the server to rekey the group?

All keys from the join point of m_2 to the *root* (k_g) have to be updated and encrypted. The following rekeying submessages are constructed:

$$E_{k_3}(k_{2\text{-}3}^{new}),\ E_{k_2}(k_{2\text{-}3}^{new}),\ E_{k_{0\text{-}1}}(k_{0\text{-}3}^{new}),\ E_{k_{2\text{-}3}^{new}}(k_{0\text{-}3}^{new}),\ E_{k_{0\text{-}3}^{new}}(k_{0\text{-}7}^{new}),\ E_{k_{4\text{-}7}}(k_{0\text{-}7}^{new})$$

In contrast to the simple scheme, where just two encryptions would be needed for join rekeying, the LKH requires more encryptions. Nevertheless, considering both join and disjoin processes, the LKH is clearly superior to the simple scheme, as visible from **Table 1.3**. While the average encryption costs increase linearly with n in the simple scheme, LKH unveils a logarithmic dependence.

Table 1.3. LKH vs. simple rekeying scheme

	# Encryptions	
	Simple scheme	LKH
Join	2	$2*\log_2 n$
Disjoin	n-1	$2*(\log_2 n -1)$
Average value	***O(n)***	***O(log$_2$n)***

1.5 Work Objectives and Outline

This dissertation makes three main contributions to multicast group rekeying. Thus, solutions, algorithms, and architectures presented in this work can mainly be assigned to the second problem area in secure multicast, see Section 1.4.1. As rekeying results in multicasting rekeying messages, which must be authenticated to prevent manipulation, this work deals also with corresponding authentication issues which belong to the first problem area. **Figure 1.7** illustrates schematically the three objectives followed in this work as big bubbles surrounded by related solutions. This figure can be used as a reference in this dissertation.

1. ***QoS and Access Control Aware Batch Rekeying:*** As mentioned previously, batch rekeying aims at optimizing the rekeying performance by processing several requests at the same time. By this means some key generations and encryptions are saved. Refer to the discussed disjoin and join requests in the example of last section and to **Figure 1.5**

and **Figure 1.6**. Note that the separate handling of these requests required the generation of 5 new keys and the execution of 10 encryptions, in total. In contrast, if these two requests are processed simultaneously, then only 3 generations and 6 encryptions will be needed. This batch rekeying mode, however, demands that former requests have to wait on later ones. Consequently, new members must wait longer to be granted access, and members who must be removed keep access for longer time periods. Related work on batch rekeying assumes a fixed rekeying interval or defines a lower bound for this parameter. In this work an upper bound is introduced, which assures that the quality of service for joining members and the access control against leaving ones always remains within system specific limits. Besides the necessary metrics for this solution, an algorithm, denoted as *Event-driven Batch Rekeying*, is presented which considers this issue. Two simulation case studies illustrate the significance of the proposed method. Chapter 2 is dedicated to this work objective.

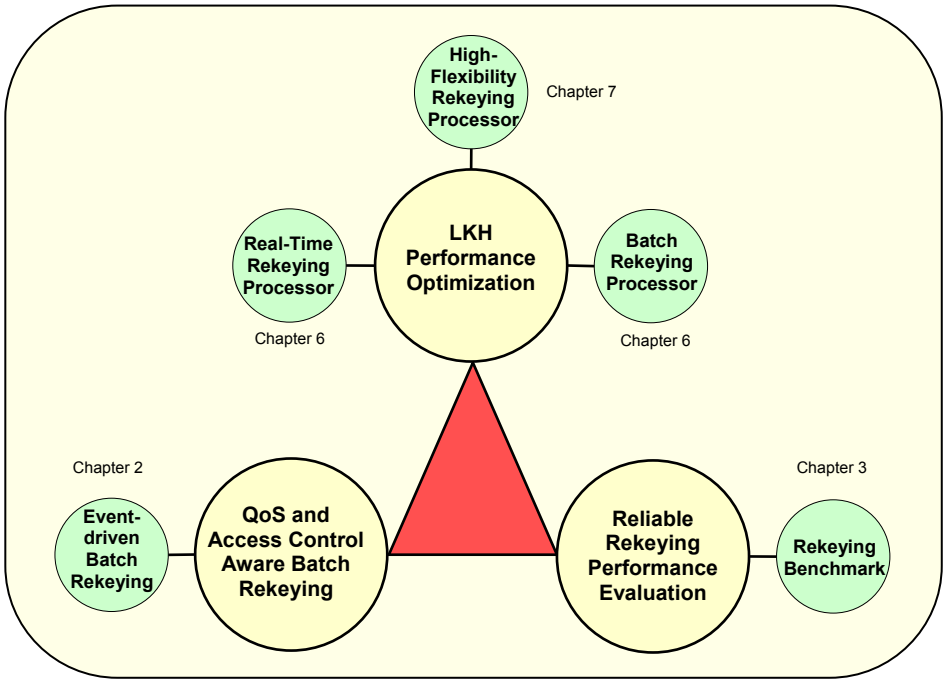


Figure 1.7. Work objectives and solution structure

2. ***Reliable Rekeying Performance Evaluation:*** The reader of related work on group rekeying misses up to now a way to compare proposed solutions to each other. This comparison is impeded by a wide spectrum of non-unified performance metrics and by largely different ways to estimate these metrics in literature. For a reliable rekeying performance evaluation this work presents a novel *Rekeying Benchmark* as a unified way for estimating unified metrics expressing rekeying performance. The reliable evaluation origins from defining new metrics to estimate performance, which are system-specific and independent of both rekeying algorithms and the underlying cryptographic operations and execution platforms. The rekeying benchmark is the subject of Chapter 3.

3. ***LKH Performance Optimization:*** All previous work on optimizing the performance of rekeying algorithms concentrates on reducing the number of time-consuming operations, which are needed to perform rekeying such as the number of key generations or the number of encryptions. In this work novel architectures are proposed, which optimize rekeying performance on a lower level. By means of hardware acceleration, not only the amount of cryptographic operations is reduced, but also the execution time of these operations. In the course of this work two hardware-only processors and one hardware/software processor were designed and implemented on reconfigurable platforms. These are the *Real-Time Rekeying Processor* (RTRP), the *Batch Rekeying Processor* (BRP), and the *High Flexibility Rekeying Processor* (HiFlexRP). Each of these architectures may be used as a coprocessor in the server environment of a multicast group owner, e.g. as a coprocessor for the registration and authentication server to accelerate rekeying in the Pay-TV scenario presented in **Figure 1.4**. Because of several similarities between the rekeying processors and to avoid repeating similar facts Chapter 4 highlights the employed implementation platforms and Chapter 5 describes the common features of these architectures. Chapter 6 then details both the RTRP and the BRP. Chapter 7 is devoted to the HiFlexRP.

Remarks and notation:

This work treats the key management problem in secure multicast with an emphasis on the server side. As for the network, only the dynamic group behavior is investigated which is reflected by member join and leave rates. Neither communication overhead nor protocol issues are considered. The dissertation can be read either in the order of its chapters or in a different way taking the following points into account. Chapter 3 is completely independent. Chapter 4 represents an introduction to reconfigurable architectures and some commercial devices and tools. Therefore, this chapter may be skipped by experts in this field. Chapter 6 and Chapter 7 depend strongly on Chapter 5, but are themselves independent of each other. Furthermore, Chapter 6 is slightly based on some points presented in Chapter 2.

As a quick reference, **Table 1.4** summarizes some important terms, which were presented in this chapter and will be used frequently in next chapters.

Table 1.4. Notation

Term	Meaning
Identity key k_d	*A key which is known to the server and one member m_d*
Group key k_g	*A key which is known to the server and to all group members. k_g is used to encrypt useful data*
Help-key $k_{x\text{-}y}$	*A key shared between the server and some group members. k_g is regarded as a special help-key*
Rekeying Submessage (RSM)	*An encrypted key k_b with a key k_a: $E_{k_a}(k_b)$*
Rekeying Message (RM)	*The set of all RSMs and some auxiliary data*

2 QoS and Access Control Aware Batch Rekeying

2.1 Overview

This chapter represents a solution to the batch rekeying problem resulting from long rekeying intervals. Section 2.2 describes the batch rekeying and compares it with the real time rekeying based on the LKH algorithm. The problems of batch rekeying are then presented and specified by new metrics in Section 2.3. Section 2.4 uses these metrics to provide different methods for optimizing batch rekeying. Section 2.5 represents two case studies and Section 2.6 concludes the chapter with some design hints for batch rekeying solutions.

2.2 Batch Rekeying

Group rekeying based on the algorithm of logical key hierarchy LKH presented in Section 1.4 features a real time characteristic. According to this algorithm each rekeying request is granted separately. However, a performance improvement can be achieved, if several requests are processed simultaneously. This processing mode is denoted as batch rekeying [Li01]. Different rekeying requests demand an update of several keys. Some of these keys, however, are likely to be processed several times if these requests are treated separately. The performance gain in batch rekeying relies on avoiding this multiple processing by reducing the number of updates of some key to one, maximally. To illustrate this point consider the left key tree in **Figure 2.1** and assume that the rekeying server receives three rekeying requests in the following order: *join* m_7, *disjoin* m_3, and *disjoin* m_0. The right key tree in **Figure 2.1** represents the state after performing rekeying for these three requests.

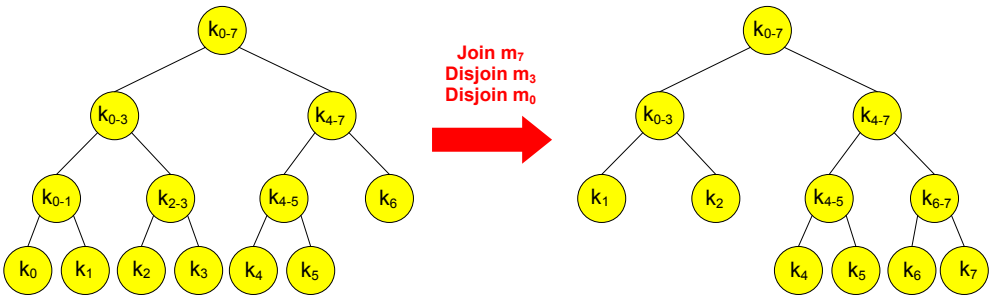


Figure 2.1. Batch rekeying example

Table 2.1 summarizes how often each of the help-keys is processed in both real-time and batch rekeying. Processing a key in this context means an update of this key and two following encryptions of it with the left and the right son keys. From this table it is obvious that processing rekeying requests in batch is more efficient than real-time rekeying.

Table 2.1. Batch vs. real-time rekeying for the previous example

	k_{0-1}	k_{2-3}	k_{4-5}	k_{6-7}	k_{0-3}	k_{4-7}	k_{0-7}
Real-time rekeying	0	0	0	1	2	1	3
Batch rekeying	0	0	0	1	1	1	1

Batch rekeying proceeds in two phases which are repeated frequently. **Figure 2.2** depicts this point:

1. *Marking:* In this phase rekeying requests are collected and the help-keys, which need to be processed, are marked. The marked keys build a so-called *rekeying subtree*.
2. *Processing:* In this phase all keys in the rekeying subtree are regenerated and encrypted by the corresponding keys to build the rekeying message.

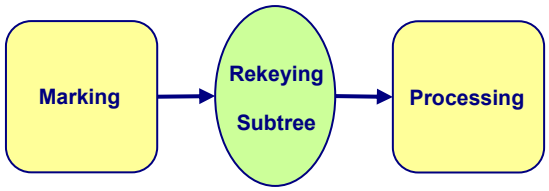


Figure 2.2. Batch rekeying

The marking is performed within regular time slots called *rekeying intervals T*. The processing takes differently long according to the built subtree. **Figure 2.3** depicts this situation, where BPT_i denotes the *batch processing time* of the *i-th* batch.

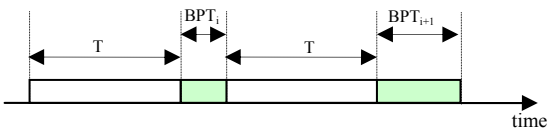


Figure 2.3. Timing in batch rekeying

2.3 Problems of Batch Rekeying

The analysis given in the previous section on batch rekeying performance is optimistic because it does not consider the waiting times of requests before they are served. The longer the rekeying interval the higher the probability for some requests to wait longer. Too long rekeying intervals in batch rekeying have two drawbacks with regard to *Quality of*

Service and *Access Control*. A new member, on the one hand, gets the group key in batch rekeying later than in real-time rekeying which means that only a worse QoS can be offered by batch rekeying. On the other hand, a leaving member remains to have a valid group key in batch mode for a longer time period than in immediate rekeying which corresponds to degradation in the access control. To quantify these items two new metrics are introduced: these are the *Join Batch Delay (JBD)* and the *Disjoin Batch Delay (DBD)*.

2.3.1 Join Batch Delay

Definition 2.1:

Join Batch Delay is defined as the additional waiting time for a joining member to get the group key in batch rekeying compared to the real-time case, see **Figure 2.4.** Within a rekeying interval T a join request appears delayed by t_{aJ} from the end point of last interval. An immediate processing of this request, i.e., without batching, would take t_{pJ}. However, through batch processing the corresponding member will be joined at the end of the processing of the *(i+1)-th* batch. Accordingly, JBD can be estimated using the following formula.

$$JBD = T + BPT_i + BPT_{i+1} - (t_{aJ} + t_{pJ}) \qquad (2.1)$$

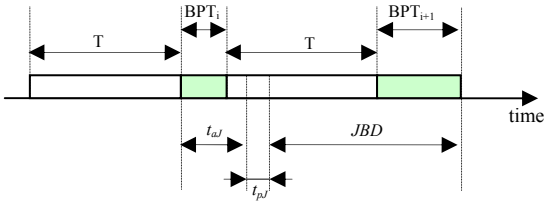


Figure 2.4. JBD in batch rekeying

2.3.2 Disjoin Batch Delay

Definition 2.2:

Disjoin Batch Delay is defined as the additional time needed to deactivate the help-keys and the group key of a leaving member in batch rekeying compared to the real-time case, see **Figure 2.5.** Similarly to JBD, DBD can be estimated as follows.

$$DBD = T + BPT_i + BPT_{i+1} - (t_{aD} + t_{pD}) \qquad (2.2)$$

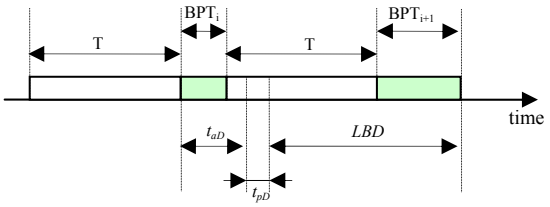


Figure 2.5. DBD in batch rekeying

Note 2.1:
Because of the analogies in the behavior of JBD and DBD, the analysis in next sections is sometimes limited to JBD to avoid repetition.

2.4 Optimized Batch Rekeying

As a rule, the processing time of a join request in real-time rekeying t_{pJ} is short compared to the other terms of equation (2.1) and can therefore be neglected. In addition, considering all joining members in one rekeying interval, the worst-case *JBD* must be investigated. This case occurs for the earliest join request in the interval, in other words for the join request with the minimal appearance time t_{aJ}^{min}. The *JBD* equation, accordingly, can be rewritten as follows.

$$JBD_{worst} = T + BPT_i + BPT_{i+1} - t_{aJ}^{min} \qquad (2.3)$$

In general, optimizing the QoS for joining members is based on minimizing JBD_{worst} which can be achieved by means of elimination or minimization of the contributing terms in (2.3). These terms are:

T: *Rekeying interval*

BPT_i: *Batch processing time of the current batch*

BPT_{i+1}: *Batch processing time of the next batch*

2.4.1 Optimized Cryptographic Algorithms and Platforms

The batch processing time *BPT* in (2.3) is mainly affected by four parameters:

1. *G:* Number of new keys needed for processing the corresponding subtree.
2. *E:* Number of encryptions needed for processing the corresponding subtree.
3. C_g: Cost of the generation of one key in time units *(generation cost factor)*.
4. C_e: Cost of one encryption in time units *(encryption cost factor)*.

Thus, *BPT* can be estimated as follows.

$$BPT = C_e \cdot E + C_g \cdot G \qquad (2.4)$$

The encryption/generation cost factors C_e and C_g depend, on the one hand, on the used algorithms for encryption and key generation. On the other hand, they are affected by the performance of the underlying platform. The more efficient the encryption/generation algorithms and the more high-performance the executing platform is, the smaller *BPT* and therefore the smaller *JBD* will be. The number of encryptions and generations, *E* and *G*, however, is much more complex to estimate because of its dependency on the current state of the key tree and on the following indeterministic factors:

1. The number of join/disjoin requests N_J/N_D in the corresponding rekeying interval, which is a function of the temporal request distribution and of the rekeying interval.
2. The logical (spatial) distribution of disjoin requests in the tree. (In contrast, join requests have no indeterministic contribution, because the server decides on the join points in the tree).

In general, both *E* and *G* increase with higher request rates, with higher dispersion of disjoin requests and with higher group sizes. To keep the number of encryptions/ generations small, an appropriate rekeying algorithm must be chosen, e.g., LKH [Wo00] or One-way Function Trees [Sh03], etc.

Optimizing the QoS and access control in batch rekeying by means of *BPT* minimizing can be classified as hard QoS/AC management, because it demands essential improvements of algorithms and/or platforms which is expensive and can not be performed in real time, as a rule.

2.4.2 Pipelined Batch Rekeying

In addition to performance enhancements, parallelizing the marking and processing phases of batch rekeying results in a smaller *JBD* and consequently a higher QoS. During the processing of the *i-th* batch the *(i+1)-th* subtree can be generated as depicted in **Figure 2.6**. The suffix *p* in JBD^p stays for pipelining. See **Figure 2.4** for a comparison.

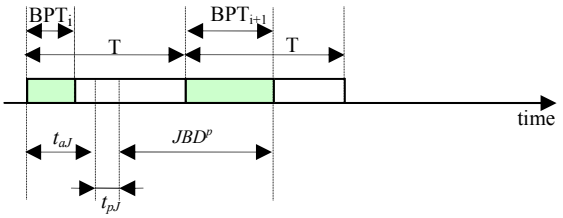


Figure 2.6. JBD in pipelined batch rekeying

From this figure it is obvious, that the contribution of BPT_i to *JBD* is eliminated.

$$JBD^p{}_{worst} = T + BPT_{i+1} - t_{aJ}{}^{min} \qquad (2.5)$$

Similarly to *BPT* minimizing, pipelining can be seen as hard QoS/AC management. This strategy, however, optimizes QoS/AC independent of the underlying algorithms for rekeying, encryption and key generation, and without relation to the network situation, which is mirrored by the request rate.

2.4.3 Event-driven Batch Rekeying

Related work on batch rekeying assumes either a constant rekeying interval, e.g. [Li01] and [Zh01], or defines a lower bound on this interval to limit communication overhead, e.g. [Ya01] and [Ji02]. For the purpose of QoS/AC improvement an upper bound on *T* has to be introduced as follows.

$$T < min\ \{T_{max1},\ T_{max2}\} \qquad (2.6)$$

T_{max1} and T_{max2} correspond to the maximal allowable join and disjoin batch delays in the system specification JBD_{max} and DBD_{max}, respectively. Using (2.5) T_{max1} can be estimated as follows. The pipelining suffix p is neglected, for clarity.

$$T_{max1} = JBD_{max} - BPT_{i+1} + t_{aJ}^{min} \quad (2.7)$$

Similarly, for T_{max2} the following can be written.

$$T_{max2} = DBD_{max} - BPT_{i+1} + t_{aD}^{min} \quad (2.8)$$

Secure multicast applications differ according to their sensitivity to the QoS and access control associated with batch processing. While Pay-TV, for example, emphasizes high QoS values and accepts as a rule some loss of access control, other applications, e.g., in military fields, do not tolerate any sacrifice of these parameters. Furthermore, the demands on QoS and access control can vary from time to time for the same application depending on some scenario-specific parameters. A Pay-TV provider, for example, can tolerate longer values of *DBD* to guarantee the required *JBD* at times of high join rates, e.g., shortly before starting the streaming of a sport event.

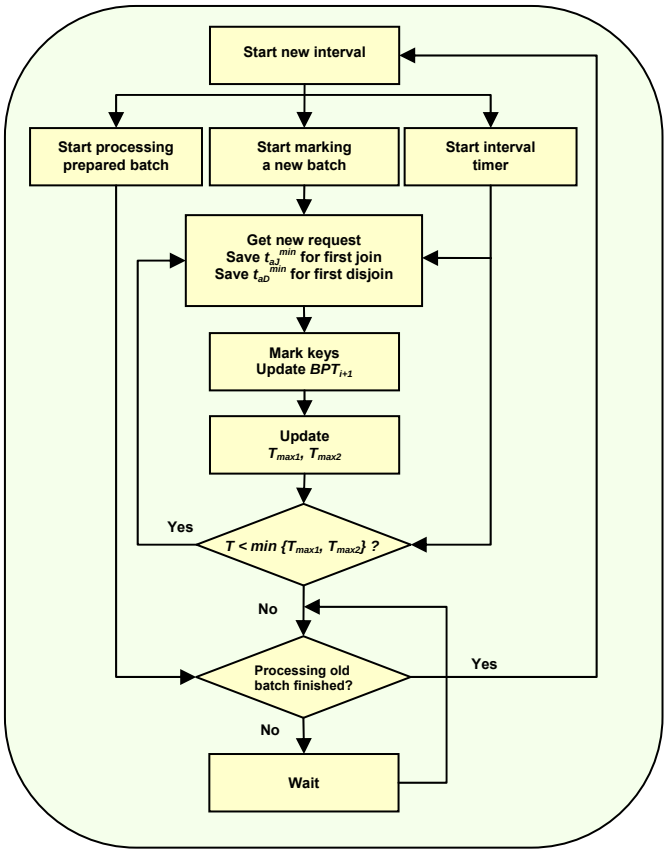


Figure 2.7. Pipelined event-driven batch rekeying

The application of (2.6) results in an *event-driven batch rekeying* which is activated by the events of exceeding T_{max1} and T_{max2}. These events cause the concluding of the current rekeying interval and the starting of a new one.

Figure 2.7 illustrates the new batch rekeying algorithm which supports both pipelining and event-driven operation modes to optimize QoS and access control. Note that, because of pipelining, a new interval can only be started if the processing of the old batch is already finished which is true in most cases. As depicted in **Figure 2.7,** event driven rekeying relies on estimating the batch processing time BPT_{i+1} during marking. This task assumes knowing the encryption and the generation cost factors C_e and C_g according to (2.4).

The Batch Rekeying Processor presented in Chapter 6 realizes an event-driven batch rekeying [Sh05]. In this hardware implementation the factors C_e and C_g are well defined and a real-time functionality guarantees an accurate estimation of BPT_{i+1} because of the common time base for all hardware modules. The module Batch Delay Monitor of this processor is integrated to a preprocessing unit which performs the marking to prepare the rekeying subtree.

In contrast to the other optimization strategies, event-driven rekeying provides a real-time control of QoS and access control in each rekeying interval and can, therefore, be classified as soft QoS/AC management which keeps QoS/AC within desirable values during the system operation.

2.5 Case Studies

Though the *JBD* behaviour given in (2.5) appears to be simple to evaluate, a comprehensive analysis of this quantity is almost impossible. This is particularly because of the highly complex dependencies of the batch processing time. To illustrate this point, this characteristic is represented as a set of functional relationships, where N_J and N_D refer to the number of join and disjoin requests summed up in a rekeying interval, respectively.

1. $JBD^p_{worst} = f_1 (T , BPT_{i+1} , t_{aJ}^{min})$
2. $BPT_{i+1} = f_2 (C_e , E , C_g, G)$
3. $C_e , C_g = f_3$ *(encryption/generation algorithms, platform)*
4. $E , G = f_4 (N_J , N_D ,$ *tree state, spatial request distribution in the tree)*
5. $N_J , N_D = f_5$ *(T, temporal request distribution)*

Note that the rekeying interval T affects JBD^p_{worst} not only directly according to f_1, but also indirectly corresponding to f_5 depending on the current distribution function of join/disjoin requests.

Almost all the related work on batch rekeying only consider the relation f_4 for E based on borderline cases. A few papers [Ya01, Zh03] investigate the relation f_5 assuming an exponential request distribution. Based on these approaches, a form for the function f_1 is derived, which can be evaluated by means of simulation. For this purpose, borderline conditions are introduced for the different relations f_1, f_2, f_4, and f_5 to ease the analysis. The

steps outlined in this derivation can be used as a general guideline for other cases and conditions.

1. $t_{aJ}^{min} = 0$, which means that the first join request appears at the beginning of a rekeying interval. Accordingly, JBD^{p}_{worst} can be written as follows.

$$JBD^{p}_{worst} = T + BPT_{i+1} \tag{2.9}$$

2. Binary trees: In this case a help-key needing to be updated is generated once and encrypted twice, thus,

$$E = 2 \cdot G \tag{2.10}$$

3. Based on the key generator specified in [An00], the generation of one key costs two encryptions, i.e.,

$$C_g = 2 \cdot C_e \tag{2.11}$$

Setting (2.10) and (2.11) in (2.4):

$$BPT_{i+1} = 2.C_e.E \tag{2.12}$$

4. Exponential distribution of the inter-arrival times of disjoin requests [Al96]. According to [Ya01, Zh01], the number of disjoin requests in a rekeying interval T is given by the following formula, where n denotes the group size. μ is the disjoin request rate.

$$N_D = n(1-e^{-\mu T}) \tag{2.13}$$

5. Balanced trees and an equal number of join and disjoin requests in the rekeying interval, $N_J = N_D$. The number of encryptions needed to process this batch can be calculated according to [Li01] as follows, where h represents the tree depth, $h = log_2 n$.

$$E = 2\sum_{i=0}^{h-1} 2^i \left(1 - \frac{\binom{n - n/2^i}{N_D}}{\binom{n}{N_D}}\right) \tag{2.14}$$

Setting (2.13) in (2.14), (2.14) in (2.12), and (2.12) in (2.9):

$$JBD^{p}_{worst} = T + 4C_e \sum_{i=0}^{h-1} 2^i \left(1 - \frac{\binom{n - n/2^i}{n(1-e^{-\mu T})}}{\binom{n}{n(1-e^{-\mu T})}}\right) \tag{2.15}$$

The resulting *JBD* is a function of the rekeying interval, the request rate, the encryption cost factor and the group size:

$$JBD^{p}_{worst} = f_1(T, \mu, C_e, n)$$

The following two case study simulations illustrate the hard and soft management of QoS introduced in the previous sections.

Case study 1

If the rekeying interval has a pre-specified value, e.g., for communication overhead reasons, then *JBD* can only be controlled by the encryption cost factor C_e for some group size n. **Figure 2.8** shows *JBD* as a function of the request rate μ with C_e as a parameter and with:

$$n = 524.288 \qquad T = 1\ sec$$

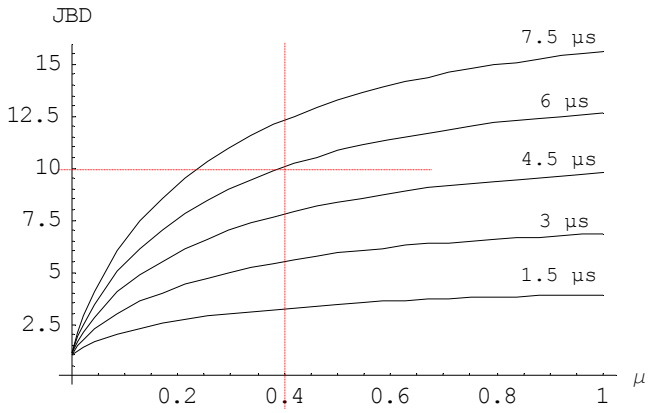


Figure 2.8. $JBD^{p}_{worst} = f(\mu, C_e)$; $T, n = const.$

The diagram of **Figure 2.8** can be viewed as a representation of the design space which can be used to decide on the minimal performance of the encryption algorithm and/or the minimal performance of the underlying platform. If the maximal acceptable join batch delay equals *10 sec*, for example, and the maximal request rate in the multicast group is $\mu = 0.4\ sec^{-1}$, then C_e must be chosen equal to $6.10^{-6}\ sec$ as a maximum. The sought value of C_e can then be obtained by means of selecting an appropriate encryption algorithm, or by using a sufficiently powerful runtime platform, or both.

Case study 2

For a given encryption algorithm and a given runtime platform the *JBD* can be optimized softly by the rekeying interval according to the request rate. **Figure 2.9** shows the simulation results for *JBD* as a function of T and μ with:

$$n = 524.288 \qquad C_e = 1.5.10^{-6}\ sec$$

By exploiting this representation it can be decided on the rekeying interval appropriate to some value of the request rate in order to satisfy the QoS specification. The algorithm presented in Section 2.4.3 provides without loss of generality an automatic selection of the rekeying interval according to both the network situation and the tree state in real-time.

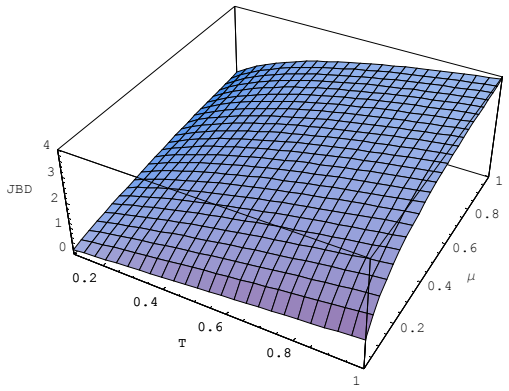


Figure 2.9. $JBD^{p}_{worst} = f(\mu,T)$; C_e, $n = const.$

2.6 Summary

In this chapter the problems of quality of service and access control associated with batch rekeying were investigated. The analysis has resulted in several improvement possibilities of the rekeying system which can be summarized in form of the following hints:

1. Arrange for a high-performance execution platform.
2. Provide efficient algorithms for encryption and key generation.
3. Provide efficient key management algorithms.
4. Exploit a pipelined batch rekeying algorithm.
5. Use event-driven batch rekeying with a variable rekeying interval.

Case studies have demonstrated the two generic methods of QoS management for batch rekeying: (1) hard, and (2) soft QoS management. Event-driven batch rekeying assumes given values for system-specific parameters JBD_{max} and DBD_{max}.

3 Rekeying Benchmark

3.1 Overview

This chapter presents a novel approach to performance evaluation of rekeying algorithms. First, Section 3.2 illustrates the problem of rekeying performance evaluation. Section 3.3 introduces the design concept of a rekeying benchmark as a solution to this problem. New performance metrics and relating simulation modes are then discussed in Section 3.4. The design of the rekeying benchmark and its components is detailed in Section 3.5. Section 3.6 depicts some implementation issues of the benchmark. Lastly, Section 3.7 illustrates the application of the benchmark by means of a simulation case study.

3.2 Rekeying Performance Evaluation Problem

A typical problem in scientific work relates to the analysis and evaluation of own results and comparing them with those of related work. This problem, on the one hand, can be attributed to the increasing number of scientific institutions and the vast publication possibilities such as journals, conferences and workshops. This trend hinders a comprehensive overview of the state-of-the-art situation in some scientific field. On the other hand, some research areas – because of their novelty, complexity, or both – lack a unified way to draw these comparisons. This situation, for instance, does not apply to the work on performance optimization of the new encryption algorithm AES. Since its release in 2001 by NIST [Ni01], an enormous amount of work is published. However, the recognized way to describe the performance of a block cipher in terms of throughput and latency allows for a reliable comparison between these solutions. On the contrary, such unified metrics are still missing for estimating the performance of rekeying algorithms, which is caused by both the novelty of this problem area and its complexity.

This complexity, however, did not only result in largely different metrics to express rekeying performance, but also in diverse ways of estimating these metrics. In this respect, the reader of proposed work on multicast group rekeying is not only confronted with different performance quantities, but also with various estimation methods such as analytical modeling, simulation based approaches, and real-time measurement using provisional prototypes. Each one of these techniques has specific constraints and drawbacks, which can be outlined in the following points:

1. ***Analytical approaches*** are always based on simplified models and relate to special cases such as full balanced trees. As a rule, rekeying performance can only be expressed by abstract numbers of some primitive operations, e.g. the number of encryptions, for borderline cases, e.g. a worst-case analysis or a best-case analysis.
2. ***Simulation based approaches*** are mostly used to prove a presented analytical investigation without model enhancement and without including sophisticated effects such as group dynamics.
3. ***Measurement approaches*** deliver results, which are strongly dependent of the deployed cryptographic primitives, their implementation, and of the platform they run on.

Furthermore, the performance of group rekeying is influenced by a couple of factors reflecting the group state and dynamics, on the one hand, and by some algorithm specific parameters such as the tree degree in LKH, on the other. Accordingly, two questions arise for performance estimation: which factors must be taken into consideration, and how should they be included, as variables or as parameters? Again, the largely different answers to these questions in related work make a large contribution to the performance evaluation problem. In summary, the difficulty of evaluating different rekeying algorithms is attributed to the following three points:

1. Non-unified performance estimation methods.
2. Non-unified consideration of the input quantities affecting the performance.
3. Non-unified definition of output metrics representing the performance.

Table 3.1 delivers a representative view of this situation in related work. Note that an input quantity can be considered either as a variable or as a parameter. This differentiation is needed when the corresponding performance metric is a function of several variables. For some estimation, a variable, which is kept constant, is called a parameter.

The diverse ways of looking at rekeying performance do not only obstruct an objective assessment of the corresponding algorithms, but also give an explanation of some inconsistencies in the conclusions drawn by some related work. The following two examples illustrate this point:

Example 3.1: Tree Degree

Though many publications on tree-based rekeying do not address the effect of tree degree, some work investigates its value, which results in optimized rekeying costs. While [Wo00] states a value of 4 as an optimal tree degree, [Go03] proves that trees with variable degree between 2 und 3 are more efficient.

Example 3.2: Tree Rebalancing

A lot of related work on LKH has commented that the logarithmic relation of rekeying costs to group size may be easily violated if the tree gets out of balance as an effect of multiple disjoin operations. In extreme cases rekeying costs can even grow linearly to the group size which makes a rebalancing of the tree indispensable.

Table 3.1. Dissimilarity in rekeying performance estimation in related work

Work	Performance estimation method	Performance estimation mode and constraints	Input Quantities		Performance metric
			Variable	Parameter	
[Li01]	Analytical	Worst-case Average costs	# Join, # disjoins, Tree degree Group size		# Encryptions
	Simulation	Worst-case Average costs	# Joins, # Disjoins 0-1000 0-4000	Tree degree 2,4,8,16,32 Group size 1024, 4096	# Encryptions
[Wa99]	Measurement	Group grows from 5000 to 20000 with 1% probability for join and 0.1 % for disjoin	Time 0-600 Sec 0-8000 Sec		# Messages per minute # Tree levels
[Sh03]	Analytical	Full balanced binary trees	Group size, Parameters for encryption, key generation, and hashing costs		Abstract cost per join/disjoin, per multiple joins/disjoins
[Wo00]	Analytical	Full balanced trees	Tree height and degree		# encryptions per request
	Measurement	Join rate = disjoin rate = 50%	Group size 0-8192	Tree degree 2-16	Request processing time
[Lu05]	Simulation	Join rate = disjoin rate = 50% n_0 = 10000. Worst-case, average cost	Operation number 0-10000		Rekeying message cost per 2000 operations
[Pe03]	Simulation	Statistically generated join/ disjoin patterns	Time 0-70 Min 0–700 Min	Batch period 0-40 Min 0–50 Min	Tree height
[Ng05]	Analytical	Worst-case, best-case analysis	Group size, tree degree, highest layer		# Keys per request
	Simulation	Full balanced trees	Group size 0-8192	# Cumulative layers	
[Am04]	Analytical		Group size # merging members # leaving members		# Exponentiation # Signatures # Verification
	Measurement	Average communication and client delay included	Group size 0-50	RSA module 512 Bit, 1024 Bit	Time per join/disjoin. Time per merging/partition
[Ch02]	Analytical	1 join / 1 disjoin	Group size All potential members Potential members not in the group currently		# Encrypted messages
	Simulation	Dedicated for some MBone sessions	Time 0-400 hours	Group sizes 4096, 64 K. Batch period 20-240 Min	
[Mi97]	Measurement		Payload size in bytes		Time

The first contribution on tree rebalancing has been made by Moyer [Mo99] who introduced two methods: an immediate and a periodic rebalancing. The immediate rebalancing results in worst-case rekeying costs of $4 \cdot log_2 n$ encryptions instead of $2 \cdot log_2 n$ in the case without rebalancing. For the periodic rebalancing, however, no information is provided about the overall performance. Moharrum et al. [Mo04] presented another method for rebalancing based on sub-trees. A comparison with the solution of Moyer is drawn, but not with the original LKH. Rodeh et al. [Ro00] applied rebalancing methods known in AVL trees to rebalance multicast key trees. However, no backward access control is guaranteed in this solution. Goshi et al. [Go03] proposed three algorithms for tree rebalancing. The analytical analysis in this work does include rebalancing costs and the simulation results only relate to equally likely join and disjoin behavior, which does not disturb the tree structure as a rule. The same remark applies to the simulation results provided by Lu [Lu05], who presented a rebalancing method without node splitting. Section 3.7 provides a detailed case study, which illustrates that rebalancing costs exceed the gain associated with it, which does not satisfy the usage of this tree management strategy.

This chapter presents a solution to the performance evaluation problem in multicast group rekeying [Sh07a]. The proposed benchmark provides a simulation environment, which can be used to evaluate different rekeying algorithms in a unified way. The study presented in this chapter is limited to the computational overhead on the server side of a group manager. This component, however, dominates the overall rekeying costs in many cases. An inclusion of communication costs and of the computational overhead on the client side is planed in future work.

3.3 Rekeying Benchmark Design Concept

3.3.1 Benchmark Abstraction Model

Rekeying presents a solution for group key management in secure multicast. As an essential step in the process of joining and removing members, rekeying performance directly influences the efficiency of this process with major effects on the system behavior. The faster a member can be removed the higher is the system security. The faster a member can be joined the higher is the system quality of service. The more efficient the rekeying the larger the groups, which can be supported and the more members may be joined and removed per time unit. Accordingly, the importance of rekeying performance estimation results from the significant effects of this performance on the system behavior in respect of the following items:

1. The *amount of quality of service*, which can be offered to a joining member.
2. The *amount of security* against a removed member.
3. *Scalability* in terms of supportable group sizes.
4. *Group dynamics* in terms of maximal supportable join and disjoin rates.

A representation of rekeying performance using these items allows a more understandable and reliable means to evaluate different rekeying algorithms. The advantage of this

presentation stems from the abstraction associated with it. This can be illustrated as follows. To enable a reliable evaluation of rekeying algorithms, metrics must be estimated, which are independent of these algorithms. Thus, an abstraction of the performance estimation from rekeying algorithms is required. **Figure 3.1** represents the task of evaluating rekeying algorithms as a four-layer abstraction model. The highest layer, denoted as the benchmark layer, takes the responsibility for performance evaluation of rekeying algorithms which are executed on the following rekeying layer. The introduction of the two lower layers, the cryptography and the platform layers, originates from the following analysis. The rekeying layer performs join and disjoin requests based on cryptographic operations such as encryption and digital signing. For each cryptographic primitive a wide selection is available. Taking symmetric-key encryption as an example, rekeying may employ DES, 3DES, AES, IDEA or other algorithms. The same rekeying algorithm behaves differently according to the utilized cryptographic primitives. Further more, the same cryptographic primitive features different performance according to the platform it runs on. This fact remains, even if public-domain libraries such as CryptoLib [Cl07] are utilized to realize cryptographic functions. Consequently, a reliable rekeying benchmark does not only rely on an abstraction from the details of the analyzed rekeying algorithms. Rekeying itself must be decoupled from the underlying cryptographic primitives and from the executing platform.

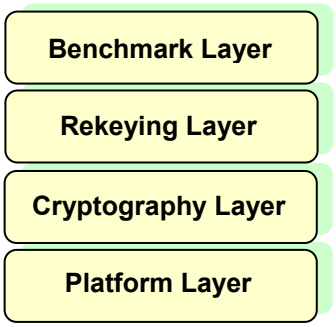


Figure 3.1. Rekeying benchmark abstraction model

The representation of the task of rekeying performance evaluation as an abstraction model provides several advantages and introduces essential design aspects for the benchmark:

1. Due to the abstraction of the benchmark from the rekeying task, a reliable und understandable mechanism for comparing different rekeying algorithms is provided.
2. The translation of rekeying costs into a system level permits the combination of these costs with other system costs such as those of user registration and authentication.
3. The separation of rekeying algorithms from the cryptographic layer and from the execution platform leads to a substantial acceleration of the evaluation process. This gain is based on the fact that rekeying algorithms to be evaluated do not need to execute any cryptographic algorithms. Instead, they just provide information on the required number of these operations. The actual rekeying costs are then determined by the

benchmark with the aid of timing parameters of the used primitives and the execution platform. This point will be detailed in the next section.

4. From the last point it is obvious that the demand for a reliable rekeying benchmark can not be fulfilled by real-time measurements on prototypes or final products, since these measurements can not be performed independently of the cryptographic primitives and the platform. Instead, for rekeying algorithms to be evaluated fairly and efficiently some kind of simulation has to be employed.

3.3.2 Benchmark Data Flow

A good understanding of the benchmark abstraction model can be delivered by investigating the data exchange between its different layers. **Figure 3.2** shows a refinement of this model which presents the data flow between the different layers based on the following aspects:

1. The rekeying layer receives rekeying requests and executes pseudo rekeying, which means that rekeying algorithms only decide on the cryptographic operations needed for these requests without executing them. This issue is illustrated by the gap between the rekeying and the cryptography layers.
2. The rekeying requests are delivered without any timing information. This means that the rekeying layer is not informed about the temporal distribution of the rekeying requests. This task is assigned to the benchmark layer.
3. The rekeying cost data provide information on the number of the needed cryptographic operations for each rekeying request or request batch.
4. The timing parameters hide the cryptographic primitives and the executing platform to provide a unified cost estimation, which can be used by the benchmark layer for all rekeying algorithms in the same way.
5. To estimate the cost of a rekeying request the benchmark sums the products of the rekeying cost data and the corresponding timing parameters.

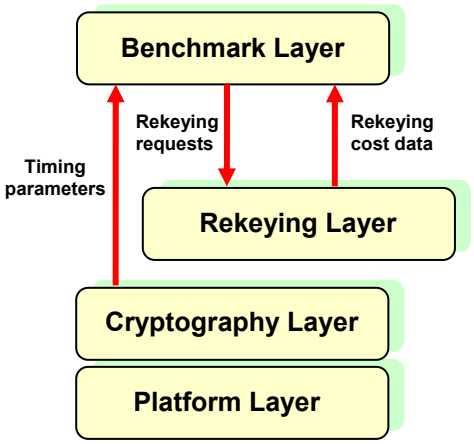


Figure 3.2. Data flow in the benchmark abstraction model

3.4 Rekeying Benchmark as a Simulation Environment

3.4.1 Cost metrics and Evaluation Criteria

As mentioned in the previous section it is necessary for the benchmark to translate rekeying costs into system-level metrics, which can be estimated for each rekeying algorithm and, thus, allow for a reliable comparison between different ones. In the following, these metrics are defined and some associated evaluation criteria are introduced. These criteria depict how the particular metric can be employed to evaluate different rekeying algorithms.

3.4.1.1 Rekeying Quality of Service (RQoS)

To define this metric two auxiliary quantities are introduced first:

Definition 3.1:

A *Required Join Time* T_J^{sys} specifies a rekeying system and is defined as the maximal allowable rekeying time needed to join a member.

Definition 3.2:

An *Actual Join Time* T_J specifies a join request and is defined as the sum of the waiting time W_J of the join request in the system queue and the rekeying time RT_J consumed by a rekeying algorithm to grant this request:

$$T_J = W_J + RT_J \tag{3.1}$$

Definition 3.3:

Rekeying Quality of Service RQoS specifies a join request and is defined as the difference between the required join time of the system and the actual join time of this request:

$$RQoS = T_J^{sys} - T_J \tag{3.2}$$

Evaluation Criterion 1:

- For a rekeying algorithm to join members correctly it must feature a RQoS which is equal to or higher than zero, i.e.

$$RQoS \geq 0 \tag{3.3}$$

- Considering two rekeying algorithms with $RQoS_1$ and $RQoS_2$, the algorithm with the higher RQoS delivers a better join behavior.

3.4.1.2 Rekeying Access Control (RAC)

Similarly to RQoS, the rekeying access control depends on two other quantities which are defined first:

Definition 3.4:

A *Required Disjoin Time* T_D^{sys} specifies a rekeying system and is defined as the maximal allowable rekeying time needed to disjoin a member.

Definition 3.5:

An *Actual Disjoin Time* T_D specifies a disjoin request and is defined as the sum of the waiting time W_D of the disjoin request in the system queue and the rekeying time RT_D consumed by a rekeying algorithm to grant this request:

$$T_D = W_D + RT_D \tag{3.4}$$

Definition 3.6:

Rekeying Access Control specifies a disjoin request and is defined as the difference between the required disjoin time of the system and the actual disjoin time of this request:

$$RAC = T_D^{sys} - T_D \tag{3.5}$$

Evaluation Criterion 2:

- For a rekeying algorithm to disjoin members correctly it must feature a RAC which is equal to or higher than zero, i.e.

$$RAC \geq 0 \tag{3.6}$$

- Considering two rekeying algorithms with RAC_1 and RAC_2, the algorithm with the higher RAC delivers a better member disjoin behavior.

3.4.1.3 Maximal Group Size n_{max}

This metric represents the maximal group size which can be supported without deterioration of the system requirements of QoS and access control.

Evaluation Criterion 3:

- For a rekeying algorithm to join and disjoin members correctly a maximal group size must be chosen which fulfills both criteria (3.3) and (3.6).
- Considering two rekeying algorithms, which fulfill criteria (3.3) and (3.6), the algorithm, that supports a higher n_{max}, features higher scalability.

3.4.1.4 Maximal Join and Disjoin Rates

The total rekeying time depends on group dynamics. The higher the request rate the higher is the rekeying algorithm occupancy and the higher is the probability for new requests to wait in the system queue. According to related work on modeling the multicast member dynamics, e.g. [Al96], the benchmark assumes the following model:

1. The arrival process of new members underlies a Poisson distribution. The inter-arrival times, accordingly, are exponentially distributed with the parameter λ, which indicates the number of join requests per time unit.
2. The duration of members in the group is a random variable which also features an exponential distribution with the parameter μ, which represents the number of disjoin requests per time unit.

Evaluation Criterion 4:

- For a rekeying algorithm to join and disjoin members correctly maximal join/disjoin rates λ_{max} /μ_{max} must be chosen, so that both criteria (3.3) and (3.6) are fulfilled.
- Considering two rekeying algorithms, which fulfill both criteria (3.3) and (3.6), the algorithm, that supports higher λ_{max} /μ_{max}, features higher dynamics.

3.4.2 Simulation Modes

The benchmark provides a way to verify the evaluation criteria defined above by means of simulation. For this purpose four simulation modes are proposed. The next sections provide a general description of these modes and their utilization goals. An in-depth description of the simulation process will be provided in Section 3.5. As a quick reference, **Figure 3.3** and **Table 3.2** provide an overview of the supported simulation modes and the associated parameters and settings.

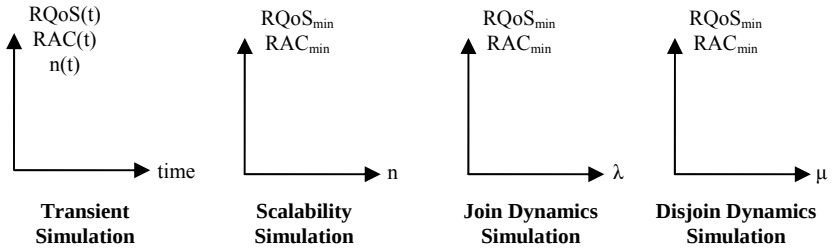


Figure 3.3. Simulation modes in the rekeying benchmark

Note 3.1:

The system parameter N_{max} given in **Table 3.2** represents the *desired* maximal group size. This parameter is required by some rekeying algorithms for set-up. It differs from n_{max}, which is the actually supportable group size by a rekeying algorithm, see Section 3.4.1.3.

3.4.2.1 Transient Simulation

This simulation estimates the current group size *n(t)*, the rekeying Quality of Service *RQoS(t),* and the rekeying access control *RAC(t)* as functions of time. With the help of this simulation mode the behavior of rekeying algorithms can be observed over long time periods and in some interesting intervals such as the ones shortly before and after an important event in multicast communication. For this purpose, the transient simulation allows the setting of an initial group size n_0, a join rate λ, a disjoin rate μ, and the desired simulation time t_{sim}. Like all other simulation modes, the transient simulation receives the

system parameters T_J^{sys} and T_D^{sys}, which refer to the required rekeying times from the system point of view. In addition, to estimate the rekeying times RT_J and RT_D, the timing parameters must be entered, see Definition 3.21. Similarly to the system parameters, the timing parameters are independent of the simulation mode, as can be seen in **Table 3.2**. The transient simulation builds the foundation for all the other three simulation modes.

Table 3.2. Simulation parameters and metrics

Simulation Modes	System Parameters	Timing Parameters	Group Parameters	Simulation Parameters	Variable	Output Metrics
Transient Simulation	T_J^{sys} T_D^{sys} N_{max}	C_g C_e C_h C_m C_s	n_0 λ μ	t_{sim}	time	RQoS(t) RAC(t) n(t)
Scalability Simulation			λ μ	T_o $[n_{start}\text{-}n_{end}]$ Δn	n	$RQoS_{min}$ RAC_{min}
Join Dynamics Simulation			n_0 μ	T_o $[\lambda_{start}\text{-}\lambda_{end}]$ $\Delta\lambda$	λ	$RQoS_{min}$ RAC_{min}
Disjoin Dynamics Simulation			n_0 λ	T_o $[\mu_{start}\text{-}\mu_{end}]$ $\Delta\mu$	μ	$RQoS_{min}$ RAC_{min}

3.4.2.2 Scalability Simulation

The importance of this simulation mode results from the significance of the scalability problem in group rekeying. The scalability simulation investigates the effect of the group size on the system behavior. Group size influences the *RQoS* and *RAC* through the rekeying time terms RT_J and RT_D in (3.1) and (3.4). Each group size n of a user-definable range serves as an initial group size for a new scalability simulation point. Scalability simulation differs from transient simulation in two points. First, for each scalability simulation point a transient simulation is started over a fixed observation interval T_o with the current group size as initial value (n_0) for the transient simulation. The second specialty of scalability simulation relates to estimating the worst-case values of the performance metrics *RQoS* and *RAC*, i.e. $RQoS_{min}$ and RAC_{min}. In other words, from all requests collected in the observation interval T_o only the join request with the worst RQoS and the disjoin request with the worst RAC are considered. The scalability simulation helps to estimate the maximal group size n_{max} which can be supported by some rekeying algorithm for certain group dynamics. The maximal group size n_{max} can be estimated graphically as the lower group size at the intersection of the curves of $RQoS_{min}$ and RAC_{min} with the x-axis. This issue is schematically illustrated in **Figure 3.4**.

3.4.2.3 Join Dynamics Simulation

High join rates result in short inter-arrival times of join requests and more rekeying computations. This causes longer waiting times for new join and disjoin requests. Accordingly, higher join rates does not only affect the rekeying QoS, but also the rekeying

access control because of the waiting time terms in (3.1) and (3.4). The join dynamics simulation represents a way to investigate theses dependencies. The user defines an initial group size n_0, a disjoin rate μ, and a fixed observation interval T_o. In addition, a simulation range for the join rate λ is entered. For each value of λ a transient simulation over T_o is started which is similar to the one described in the scalability simulation. With the help of join dynamics simulation the maximal allowable join rate λ_{max} for a rekeying algorithm can be estimated. λ_{max} corresponds to the lower λ-value of the intersection points of the curves of $RQoS_{min}$ and RAC_{min} with the x-axis.

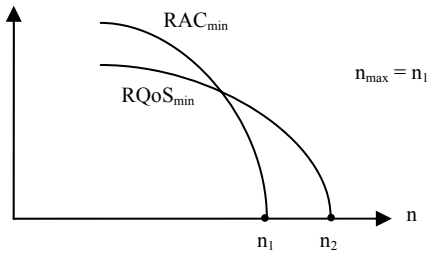


Figure 3.4. Scalability simulation to estimate n_{max}

Another interesting knowledge, which can be gained on the base of this simulation mode, relates to the investigation of the rekeying algorithm behavior in the case of unbalanced group dynamics. Unbalanced group dynamics means that the join request rate exceeds the disjoin request rate, or wise versa. The significance of this analysis is based on the fact that the performance of some rekeying algorithms differs considerably according to the request type dominating the group dynamics. While the Star Graph rekeying [Wo00], for instance, scales well for join requests, it performs largely inefficient in the case of high disjoin request rates. To investigate the effect of unbalanced group dynamics a vertical line is drawn at the join rate, which equals the defined disjoin rate μ, and the $RQoS_{min}/RAC_{min}$ behavior is observed on the both sides of this line. **Figure 3.5** schematically illustrates this point. The $RQoS_{min}$ of two rekeying algorithms A1 and A2 is estimated. For join rates, which are lower than the given disjoin rate, algorithm A1 features higher performance than algorithm A2. This advantage, however, decreases with increasing join rates, which becomes evident after exceeding the disjoin rate.

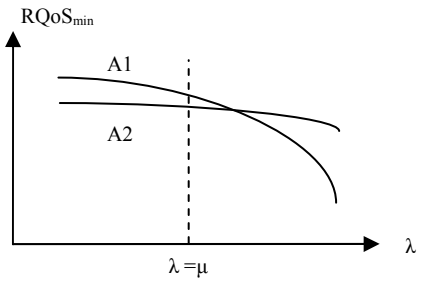


Figure 3.5. Join dynamics simulation to investigate unbalanced dynamics

3.4.2.4 *Disjoin Dynamics Simulation*

This simulation can be utilized to estimate the maximal disjoin rate μ_{max}. All other properties of the join dynamics simulation apply to this simulation and are not repeated here, for brevity.

3.5 Rekeying Benchmark Design

3.5.1 General Architecture

The rekeying benchmark is mainly composed of two interfaces and three components, as depicted in **Figure 3.6**:

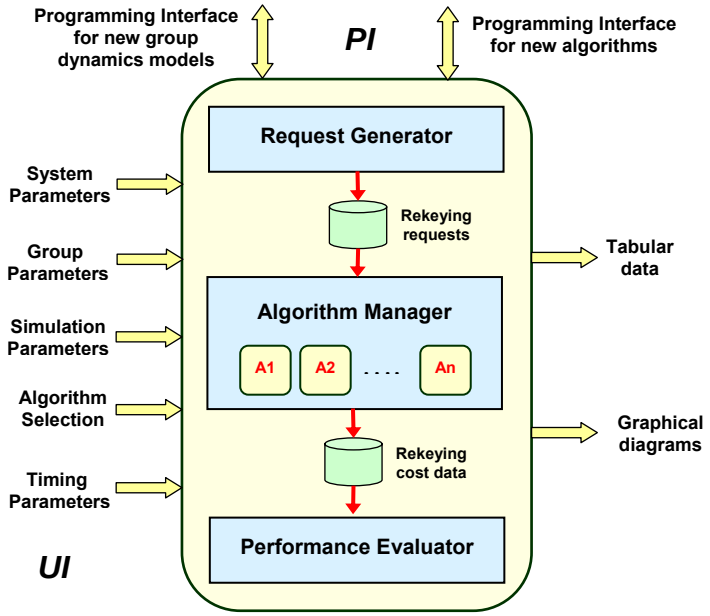


Figure 3.6. Rekeying benchmark architecture

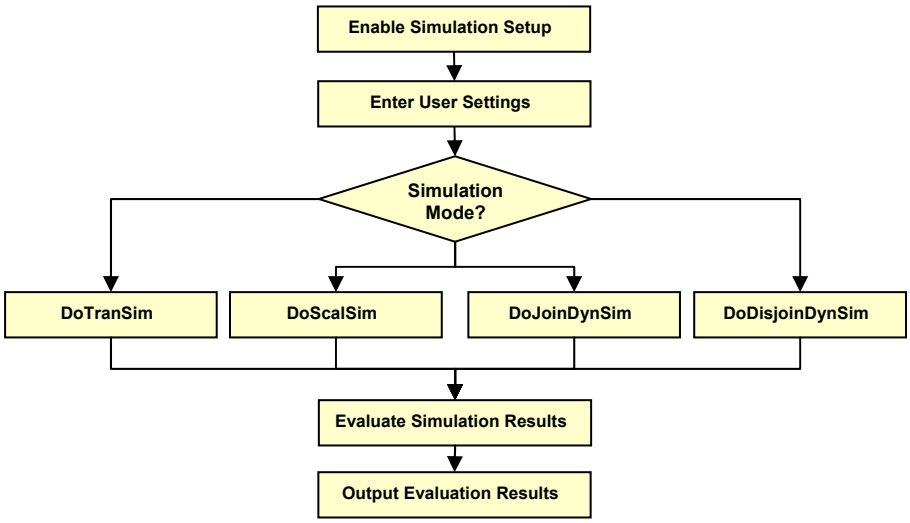
1. *User Interface (UI):* This interface enables benchmark users to evaluate different rekeying algorithms by selecting these algorithms and setting the desired parameters for the system, group, timing and simulation runs. Simulation results can be captured in a tabular form or graphically.
2. *Programming Interface (PI):* For designers of rekeying algorithms this interface enables the integration of new algorithms into the benchmark environment. In addition, groups with special dynamic behavior, which does not follow a Poisson distribution, can be supported with the aid of a special programming interface.
3. *Request Generator:* Depending on the entered group and simulation parameters the request generator builds a request list. An entry of this list keeps information on the request type, join or disjoin, the identity of the member to be joined or disjoined, and

the arrival time of this request. The request generator deals with the difficult task of modeling the group dynamics. Group dynamics includes two indeterministic contributions. The first component relates to the stochastic process of request arrivals. The second indeterministic contribution represents the task of choosing the member to be removed. This item results from the fact that most rekeying algorithms feature member-dependent rekeying performance. Removing a member located on a higher level in a key tree, for instance, does not cost as much as removing another, whose leaf belongs to a lower level. Choosing identities for joining members, in contrast, is deterministic because the group manager has control of the join point, as a rule.

4. *Algorithm Manager:* This component selects and configures the rekeying algorithms to be evaluated according to user settings. It coordinates the functions of the benchmark and the rekeying algorithms.
5. *Performance Evaluator:* Based on the rekeying cost data delivered from the rekeying algorithms, the entered timing parameters, and on the selected simulation, the rekeying performance of each algorithm in terms of RQoS and RAC is estimated and prepared for display.

The algorithm manager plays a central role in the benchmark architecture. Its functionality can be illustrated by the process described in **Algorithm 3.1**. After reading the user settings of the desired parameters, the simulation mode, and the algorithms to be evaluated, the algorithm manager executes the corresponding simulation process. Simulation processes on their part call the request generator and pass the rekeying requests to the selected rekeying algorithms. As a result, a simulation process provides abstraction rekeying costs, i.e. without timing information. This information is first supplied to the performance evaluator which combines the timing parameters with the abstract rekeying costs to determine the RQoS and RAC metrics.

Algorithm 3.1 Benchmark evaluation process



For clarity, next sections detail the three benchmark components in an order corresponding to the benchmark architecture as depicted **Figure 3.6**.

3.5.2 Request Generator

The request generator (RG) produces a *rekeying request list RRL(T)* by executing the *Request Generator Process* based on three subprocesses: the *Arrival Process*, which generates *join/disjoin arrival lists* $A_J(T)/A_D(T)$ and the *Join/Disjoin Identity Selection Processes*, which generate a member identity for a join/disjoin request. For a formal description, a terminology specific to the RG is presented first.

3.5.2.1 Request Generator Terminology

Definition 3.7:

A *Rekeying Request* is 3-tuple *(type, ID,* t_a*),* where *type* indicates the request type which can be join (*J*) or disjoin (*D*). *ID* represents the member identity to be joined (ID_J) or disjoined (ID_D). t_a describes the arrival time of a join request (t_{aJ}) or a disjoin request (t_{aD}) measured from the start point of the simulation run.

Definition 3.8:

A *Rekeying Request List over T*, RRL(T)*,* is an ordered set of rekeying requests, which arrive during a defined time interval *T*. The requests in the list are ordered according to their arrival times.

Example 3.3: RRL(T)

An RRL(T) can be represented in tabular form, **Table 3.3** depicts an example.

Table 3.3. Example for a rekeying request list RRL(T)

Request Type	Member Identity	Arrival Time (ms)
D	1099	0
J	50	0.1
J	178	2
D	22657	5.3

Definition 3.9:

A *join arrival list over T,* $A_J(T)$, is an ordered list of inter-arrival times, which relate to all join requests generated during a given time interval *T*:

$$A_J(T) = (\Delta t_J(1), \Delta t_J(2), \ldots \Delta t_J(i), \ldots , \Delta t_J(h)),$$

where $\Delta t_J(i)$ indicates the inter-arrival time of the *i-th* join request in the interval T and

$$\sum_{i=1}^{i=h} \Delta t_J(i) \leq T \qquad (3.7)$$

Definition 3.10:

A *Disjoin arrival list over T,* $A_D(T)$, is an ordered list of inter-arrival times, which relate to all disjoin requests generated during a given time interval *T*:

$$A_D(T) = (\Delta t_D(1), \Delta t_D(2), \ldots \Delta t_D(i), \ldots , \Delta t_D(k)),$$

where $\Delta t_D(i)$ indicates the inter-arrival time of the *i-th* disjoin request in the interval T and

$$\sum_{i=1}^{i=k} \Delta t_D(i) \leq T \qquad (3.8)$$

Definition 3.11:

A *member identity* (ID) is defined as a natural number which takes any value between *0* and $N_{max} - 1$, where N_{max} represents the maximal desired group size, see Note 3.1.

Definition 3.12:

From the view point of the request generator a *complete multicast group M* is defined as a set of all member identities.

$$M = \{ID(i)\},\ i = 0 \div (N_{max} - 1)$$

Definition 3.13:

A *joined multicast subgroup* (M_J) is defined as the subset of M which includes all given identities.

At the start of a simulation with initial group size n_0, M_J is defined as follows:

$$M_J = \{ID(i)\},\ i = 0 \div (n_0 - 1)$$

Definition 3.14:

A *potential multicast subgroup* (M_D) is defined as the subset of M which includes all free identities, i.e. the identities which can still be given to new members.

At the start of a simulation with initial group size n_0, M_D is defined as follows:

$$M_D = \{ID(i)\},\ i = n_0 \div (N_{max} - 1)$$

3.5.2.2 Request Generator Process (GenReqList)

This process generates a rekeying request list RRL(T) according to Definition 3.8. **Algorithm 3.2** illustrates this process as pseudo code. First, the arrival process GetArrivalLists is called to produce join and disjoin arrival lists $A_J(T)$ and $A_D(T)$, see Definition 3.9 and Definition 3.10. According to their inter-arrival times in these lists, the arrival times for the individual requests are then determined. Depending on the request type, the member identity is obtained by calling the processes GetJoinID or GetDisjoinID. Afterwards, the RRL(T) is updated by the new rekeying request 3-tuple. After processing all entries of $A_J(T)$ and $A_D(T)$, the RRL(T) is sorted by increasing arrival time. Note that the request generator is transparent to the simulation mode. Utilizing the generator for different simulation modes will be described later in the scope of the algorithm manager. Example 3.4 illustrates this code in more details.

Algorithm 3.2 GenReqList

Input: T
Output: RRL(T) -- *Definition 3.8*

1. GetArrivalLists(T) → $A_J(T)$ and $A_D(T)$ -- *Section 3.5.2.3*
2. i := 1, j := 1, t_{aJ} := 0, t_{aD} := 0;
3. **do**
4. **if** $\Delta t_J(i) \geq \Delta t_D(j)$ **then**
5. $t_{aD} := t_{aD} + \Delta t_D(j)$; -- *Equation (3.8)*
6. GetDisjoinID → ID_D -- *Section 3.5.2.5*
7. j := j + 1;
8. Add (D, ID_D, t_{aD}) into RRL(T)
9. **else**
10. $t_{aJ} := t_{aJ} + \Delta t_J(i)$; -- *Equation (3.7)*
11. GetJoinID → ID_J -- *Section 3.5.2.4*
12. i := i + 1;
13. Add (J, ID_J, t_{aJ}) into RRL(T)
14. **end if**
15. **while** (i≤h or j≤k) -- *h/k: number of $A_J(T)$/$A_D(T)$ entries*
16. Sort RRL(T)
17. **return** RRL(T)

Example 3.4: Request Generator Process

The following example illustrates **Algorithm 3.2**.

Input: A group of maximal 8 members. 5 members are currently joined as follows.

$M = \{0, 1, 2, 3, 4, 5, 6, 7\}$, $M_J = \{0, 1, 2, 3, 4\}$, $M_D = \{5, 6, 7\}$

See definitions 3.12, 3.13, and 3.14 for M, M_J and M_D, respectively.

Assume that calling the process GetArrivalLists(T) on some interval *T* results in:

$A_J(T) = (10, 25)$, $A_D(T) = (11, 5, 7)$

The contents of the inter-arrival time lists $A_J(T)$ and $A_D(T)$ indicate that during the given interval 2 join requests and 3 disjoin requests are collected, i.e. $h = 2, k = 3$. In addition, the requests feature the following inter-arrival times:

$\Delta t_J(1) = 10$, $\Delta t_J(2) = 25$, $\Delta t_D(1) = 11$, $\Delta t_D(2) = 5$, $\Delta t_D(3) = 7$.

In the first run of the do-while loop, the if-condition in **Algorithm 3.2** is false because $\Delta t_J(1) < \Delta t_D(1)$. Therefore, the first join request is processed by determining its arrival time in Step 10, which results in $t_{aJ} := 0 + 10 = 10$, as $t_{aJ} = 0$ initially. Assuming that executing

the process GetJoinID in Step 11 results in a member identity $ID_J = 5$, a first entry is written into the rekeying request list RRL(T), Step 13, as depicted in the first row of **Table 3.4** which represents the RRL(T) for his example.

In the second iteration the if-condition is true because $\Delta t_J(2) > \Delta t_D(1)$. Therefore, the next request to be written to the RRL(T) is of a disjoin type and has an arrival time $t_{aD} := 0 + 11 = 11$, as $t_{aD} = 0$ initially. Assuming that GetDisjoinID returns an ID_D which is equal to 3, the RRL(T) is extended by the second entry of **Table 3.4**.

In the third iteration the if-condition is also true because $\Delta t_J(2) = \Delta t_D(2)$. Therefore, the next request to be written to the RRL(T) is of a disjoin type and has an arrival time $t_{aD} := 11 + 5 = 16$. If GetDisjoinID returns an ID_D which is equal to *1*, the request list is updated by the third row of **Table 3.4**. The other two entries can be estimated in the same way.

Table 3.4. RRL(T) of Example 3.4

Request Type	Member Identity	Arrival Time (ms)
J	5	10
D	3	11
D	1	16
D	4	23
J	1	35

Figure 3.7 illustrates the relation between the inter-arrival times generated by the process GetArrivalList(T) and the estimated arrival times in the given example.

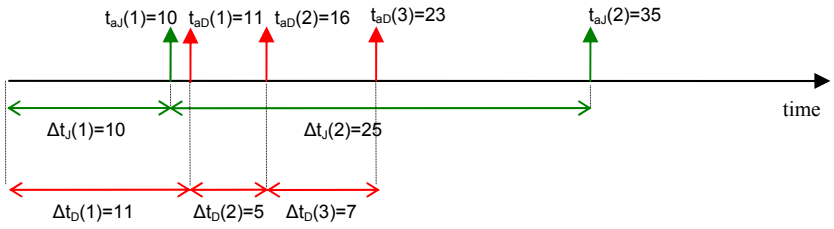


Figure 3.7. Arrival times and inter-arrival times for Example 3.4

After the request generation in this example, the joined and the potential multicast subgroups are given now as follows:

$M_J = \{0, 1, 2, 5\}$, $M_D = \{3, 4, 6, 7\}$

Note 3.2:

The benchmark prototype optionally allows the user to skip the request generator and to enter a rekeying request list freely. This provides a means to construct a RRL(T) independently of the distribution function of rekeying requests. Therefore, typical errors,

which result from choosing some distribution function based on network traffic analysis, can be avoided by using this approach.

3.5.2.3 Arrival Process (GetArrivalLists)

The arrival process fulfills the task of generating the join and disjoin arrival lists $A_J(T)$ and $A_D(T)$ according to Definition 3.9 and Definition 3.10, respectively. As default, request inter-arrival times are assumed to follow an exponential distribution for both the join and disjoin cases with the request rates λ and μ, respectively. The corresponding probability density functions (pdf) and the cumulative distribution functions (cdf) can be given by:

$$f_J(\Delta t_J) = \lambda e^{-\lambda \Delta t_J} \qquad F_J(\Delta t_J) = 1 - e^{-\lambda \Delta t_J} \tag{3.9}$$

$$f_D(\Delta t_D) = \mu e^{-\mu \Delta t_D} \qquad F_D(\Delta t_D) = 1 - e^{-\mu \Delta t_D} \tag{3.10}$$

To generate an exponentially distributed random variate – here the inter-arrival times – based on uniform random numbers in the interval [0 - 1], the inverse transformation technique can be used [Le04]. Accordingly, if *r* represents a random number between zero and one the inter-arrival time of a join or disjoin request can be estimated as follows:

$$\Delta t_J = -\frac{1}{\lambda} \ln(r) \tag{3.11}$$

$$\Delta t_D = -\frac{1}{\mu} \ln(r) \tag{3.12}$$

Algorithm 3.3 describes the arrival process as pseudo code. The process GetArrivalLists generates join requests, as long as the sum of their inter-arrival times lower than or equal to *T*. The same applies to disjoin requests.

Algorithm 3.3 GetArrivalLists

Input: T
Output: $A_J(T)$, $A_D(T)$ *-- Definitions 3.9 and 3.10*

1. $\Sigma\Delta t_J := 0$, $\Sigma\Delta t_D := 0$
2. **while** $\Sigma\Delta t_J \leq T$ **do**
3. Generate r
4. Determine Δt_J according to Eq. (3.11)
5. $\Sigma\Delta t_J := \Sigma\Delta t_J + \Delta t_J$;
6. Add Δt_J to $A_J(T)$
7. **while** $\Sigma\Delta t_D \leq T$ **do**
8. Generate r
9. Determine Δt_D according to Eq. (3.12)
10. $\Sigma\Delta t_D := \Sigma\Delta t_D + \Delta t_D$;
11. Add Δt_D to $A_D(T)$
12. **return** $A_J(T)$, $A_D(T)$

3.5.2.4 Join Identity Selection Process (GetJoinID)

To join a member the group manager can select any available identity number from the potential multicast subgroup M_D, see Definition 3.14. A possible strategy may rely on selecting the smallest available ID. This strategy naturally allows some order in the group management. This order, for example, represents itself in tree-based rekeying algorithms by filling in the tree gaps and, thus, keeping the tree balance to some extent. Accordingly, GetJoinID is a deterministic process which can be illustrated by the pseudo code in **Algorithm 3.4**. A selected ID_J must be added to M_J, i.e. to the set of given identities, see Definition 3.13.

Algorithm 3.4 GetJoinID

Input: n/a
Output: ID_J

1. Get ID_J^{min} from M_D -- *Definitions 3.14*
2. Add ID_J^{min} to M_J -- *Definitions 3.13*
3. **return** ID_J^{min}

3.5.2.5 Disjoin Identity Selection Process (GetDisjoinID)

In contrast to the join case, a rekeying system does not have prior knowledge of the member to be disjoined, in general. Therefore, selecting an identity for a leaving member from the joined multicast subgroup M_J, is a random process. Tow selection modes are proposed:

Trial Selection:

The leave identity ID_D can be modelled as a random variable which is uniform distributed in the general case. ID_D accepts any value of M_J. Calling the minimal and the maximal available identities in M_J as ID_D^{min} and ID_D^{max}, respectively, and assuming that the range $[ID_D^{min}, ID_D^{max}]$ is continuous, i.e. all identity numbers higher than ID_D^{min} and lower than ID_D^{max} are available in M_J, then an ID_D can be selected by applying the following formula:

$$ID_D = ID_D^{min} + (ID_D^{max} - ID_D^{min}) \cdot r, \qquad (3.13)$$

where r is a uniform zero-one random number. However, due to membership changes M_J does not have necessarily a continuous range of member identities. Therefore, applying (3.13) can result in an ID_D which does not belong to M_J. In this case a new zero-one random number is generated and a new ID_D is tried. This improper situation occurs more likely if the range $[ID_D^{min}, ID_D^{max}]$ is lightly occupied.

Certain Selection:

To avoid the problem of trial selection mentioned above, the ID_D's of M_J are associated with continuous successive indices from *0* to *m-1,* where *m* represents the number of all ID_D's in M_J. One way to achieve this is to save the elements of M_J as an array with the

index *i*. To select an ID_D, a uniform zero-one random number *r* is generated first. Then an index *i* is estimated using (3.14). In a last step, the ID_D is selected which is indexed by *i*.

$$i = m \cdot r \tag{3.14}$$

Accordingly, certain selection does not suffer from useless trying. However, it demands a re-indexing of the ID_D's after selecting one disjoin identity. The costs of this re-indexing increase with larger M_J.

Trial Selection vs. Certain Selection:

Selecting an appropriate selection mode with respect to efficiency is a hard problem which will be investigated in future work. For the purpose of the benchmark prototype, a switching strategy between the two selection modes is deployed. This switching depends on the occupancy of the joined multicast subgroup M_J. To describe this occupancy, the following concept is proposed.

Definition 3.15:

An *occupancy factor (OF)* is a real quantity which describes the occupancy of the joined multicast subgroup M_J and is given as follows:

$$OF = \frac{m}{1 + ID_D^{\max} - ID_D^{\min}}.100\% \tag{3.15}$$

The benchmark prototype switches to a trial selection for *OF* values larger than 50%, otherwise the certain selection mode is applied. **Algorithm 3.5** illustrates the disjoin identity selection process as pseudo code:

Algorithm 3.5 GetDisjoinID

Input: n/a
Output: ID_D

```
1.  Determine OF according to Eq. (3.15)
2.  if OF > 50% then
3.     do                                          -- trial selection
4.        Generate r
5.        Determine a potential ID_D according to Eq. (3.13)
6.     while (ID_D does not belong to M_J)
7.  else                                           -- certain selection
8.     Generate r
9.     Determine the index for an ID_D according to Eq. (3.14)
10.    Get corresponding ID_D
11.    Re-index M_J
12. Add ID_D to M_D
13. return ID_D
```

3.5.3 Algorithm Manager

This component acts as a coordinator in the benchmark and fulfills the main tasks of user interface management, algorithm control, and simulation execution. For this purpose the algorithm manager reads in the user settings and calls the request generator. It then passes the request list to the selected rekeying algorithms and collects the rekeying cost data. These data are then sent to the performance evaluator, see **Figure 3.6**. The benchmark functionality was presented as flow chart in **Algorithm 3.1** in brief. In this section the underlying simulation processes DoTranSim, SoScalSim, DoJoinDynSim, and DoDisjoinDynSim will be illustrated. For this, three basic concepts are introduced first.

Definition 3.16:

Abstract Rekeying Cost (ARC) is a 5-tuple *(G, E, H, M, S),* which specifies the costs of a rekeying request or request batch in terms of the amount of cryptographic operations needed to grant this request or request batch by a rekeying algorithm. The elements of the ARC are specified in **Table 3.5.**

Table 3.5. Abstract rekeying cost notation

ARC Element	Meaning
G	Number of generated cryptographic keys
E	Number of symmetric encryptions
H	Number of cryptographic hash operations
M	Number of message authentication code operations
S	Number of digital signatures

Definition 3.17:

A *Rekying Cost List* RCL(T) is a rekeying request list RRL(T), see Definition 3.8**,** which is extended by the abstract rekeying cost ARC for each request.

Example 3.5: RCL(T)

Table 3.6 shows an example for an RCL(T) which is an extension of the rekeying request list given in **Table 3.3.** This example results from executing the LKH algorithm with binary trees. This can be seen from the fact that each generated key is encrypted twice to determine the rekeying submessages. Note that the rekeying algorithm in this example does not apply group authentication, therefore, no message authentication codes are needed. Instead, rekeying submessages are hashed and the final hash value is signed once for each request.

Table 3.6. RCL(T) example

Request Type	Member Identity	Arrival Time (ms)	Rekeying Cost List RCL(T)				
			G	E	H	M	S
D	1099	0	6	12	12	0	1
J	50	0.1	3	6	6	0	1
J	178	2	8	16	16	0	1
D	22657	5.3	2	4	4	0	1

Because of its simultaneous processing of all rekeying requests arriving in an interval T, batch rekeying results in the same abstract rekeying costs for all requests of the RRL(T). A batch processing of the rekeying request list given in **Table 3.3** may result in the rekeying cost list shown in **Table 3.7.**

Table 3.7. RCL(T) example in batch rekeying

Request Type	Member Identity	Arrival Time (ms)	Rekeying Cost List RCL(T)				
			G	E	H	M	S
D	1099	0	20	40	40	0	2
J	50	0.1	20	40	40	0	2
J	178	2	20	40	40	0	2
D	22657	5.3	20	40	40	0	2

Definition 3.18:

A *Complex Rekeying Cost List* CRCL(T) is a set of rekeying cost lists generated over the same interval under different group conditions:

$$CRCL(T) = \{RCL_1(T), RCL_2(T), \ldots RCL_m(T)\}.$$

The concept of CRCL(T) is used to support the three complex simulation modes, the scalability, the join dynamics and the disjoin dynamics simulations. For these simulation modes an RCL(T) is generated for each n, λ or μ value in the desired simulation range, respectively.

Example 3.6:

For a scalability simulation with n_{start} = 1000, n_{end} = 2000 and Δn = 100 a CRCL(T) is produced, which contains 11 RCL(T).

Based on this terminology the different simulation processes can now be described. First the transient simulation is illustrated. Because of the large similarity between the other simulation modes, only the scalability simulation is presented, for brevity. An adapting of this description to a join/disjoin dynamics simulation is straightforward. For an overview, it is referred to **Algorithm 3.1**, which illustrates the context of these simulation processes in the overall benchmark process.

3.5.3.1 Transient Simulation Process (DoTranSim)

Algorithm 3.6 represents the process of transient simulation. Initially, the request generator process is resumed to generate a request list RRL(t_{sim}) over the entered simulation time t_{sim}. For each selected rekeying algorithm, the algorithm manager performs then two main steps. First, the rekeying algorithm is requested to initialize the group with n_0 members. n_0 can accept any value between 0 and N_{max}, see **Tabel 3.2.** Second, each rekeying request of RRL(t_{sim}) is sent to the rekeying algorithm, which then returns the corresponding abstract rekeying cost ARC for that request.

Algorithm 3.6 DoTranSim

Input: All settings for a transient simulation as given in **Table 3.2**.
Set of rekeying algorithms to be evaluated.
Output: A RCL(t_{sim}) for each rekeying algorithm *-- Definition 3.17*

1. GenReqList(t_{sim}) → RRL(t_{sim}) *-- Algorithm 3.2*
2. **for each** rekeying algorithm **do**
3. Initialize the group with n_0 members
4. **while** RRL(t_{sim}) is not empty **do**
5. Send a rekeying request to the algorithm
6. Get corresponding ARC *-- Definition 3.16*
7. Add ARC to RCL(t_{sim})
8. **return** RCL(t_{sim}) of all algorithms

3.5.3.2 Scalability Simulation Process (DoScalSim)

Algorithm 3.7 represents the process of scalability simulation. As mentioned in Section 3.2, this simulation mode relies on the transient simulation. For each group size value n of the desired simulation range [n_{start}, n_{end}], a transient simulation is performed over the entered observation time T_o. Recall that this simulation mode provides a rekeying cost list RCL(T_o) for each simulation point. Selecting the request with the maximal cost from RCL(T_o) is a task of the performance evaluator as will be seen in the next section.

Algorithm 3.7 DoScalSim

Input: All settings for a scalability simulation as given in **Table 3.2**.
Set of rekeying algorithms to be evaluated.
Output: A CRCL(T_o) for each rekeying algorithm *-- Definition 3.18*

1. **for each** rekeying algorithm **do**
2. n := n_{start};
3. **while** n ≤ n_{end} **do**
4. DoTranSim for T_o and n_0 := n → RCL(T_o) *-- Algorithm 3.6*
5. Add RCL(T_o) to CRCL(T_o)
6. n := n + Δn;
7. **return** CRCL(T_o) of all algorithms

3.5.4 Performance Evaluator

This component receives a set of RCL(T) or CRCL(T) and calculates the system metrics RQoS and RAC with respect to time, group size, or join/disjoin request rate, depending on the simulation mode. First, three concepts are introduced, which are necessary for a formal description of the functionality of this component.

Definition 3.19:

A *Performance Simulation Point* (PSP) is 3-tuple *(x, RQoS, RAC),* where *x* represents the variable to which the RQoS and RAC are related.

Depending on simulation mode x, RQoS and RAC are interpreted as already illustrated in **Table 3.2.** Recall that RQoS is not defined for a disjoin request. Similarly, RAC is not available for a join request.

Definition 3.20:

A *Rekeying Performance List* (RPL) is a set of performance simulation points.

$$RPL = \{PSP\} = \{(x_1, RQoS_1, RAC_1), (x_2, RQoS_2, RAC_2), \ldots\}$$

Definition 3.21:

A *Timing Parameter List* (TPL) is a 5-tuple $(C_g, C_e, C_h, C_m, C_s)$, where the tuple elements are defined as depicted in **Table 3.8**.

As mentioned in Section 3.3.2, the timing parameters reflect the performance of employed cryptographic algorithms and of the platform to the benchmark layer, which allows for a reliable evaluation of different rekeying algorithms. Timing parameters are entered by the user independently of the simulation mode as was illustrated in **Table 3.2.**

Table 3.8. Timing parameters meaning

ARC Element	Meaning
C_g	Cost of generating one cryptographic key in time units
C_e	Cost of one symmetric encryption in time units
C_h	Cost of one cryptographic hash operation in time units
C_m	Cost of one message authentication code in time units
C_s	Cost of one digital signature in time units

The performance evaluator executes processes, which combine a rekeying cost list RCL(T) or a complex rekeying cost list CRCL(T) with a timing parameter list TPL to produce a rekeying performance list PRL for each rekeying algorithm.

For each rekeying request in RCL(T)/CRCL(T) the actual join and disjoin time is determined according to equations (3.1) and (3.4), respectively. For this purpose, the rekeying time for a join or disjoin request consumed by a rekeying algorithm is estimated first according to:

$$RT_{J/D} = G.C_g + E.C_e + H.C_h + M.C_m + S.C_s \qquad (3.16)$$

Second, the waiting time of a request is estimated as follows:

$$W_{J/D} = \sum_{i=1}^{m} RT_i \qquad \text{for } m \geq 1 \qquad (3.17)$$

$$W_{J/D} = 0 \qquad \text{for } m = 0, \qquad (3.18)$$

where *m* represents the number of all requests waiting in the system queue or being processed at the arrival of the request at hand. Note that equation (3.17) is approximate since it does not consider the time part of RT_1 (the request being processed), which has passed before appearing the considered request.

Knowing the waiting times and the rekeying times, the actual rekeying times can be estimated using (3.1) and (3.4). Afterwards, RQoS and RAC can be calculated for a join or disjoin request according to (3.2) or (3.5), respectively.

3.5.4.1 Transient Evaluation Process (EvalTranSimResults)

In the case of a transient simulation the performance evaluator executes the process EvalTranSimResults according to **Algorithm 3.8**. For each join and disjoin request in the RCL(T), a performance simulation point PSP is determined. The symbol ∞ in the pseudo code indicates an undefined metric for the current state. For example, RQoS is not defined for a disjoin request. t_{aJ} and t_{aD} represent the arrival times of the corresponding join and disjoin requests, respectively. Remember that these time values are determined from the arrival lists by the request generator process according to **Algorithm 3.2**.

Algorithm 3.8 EvalTranSimResults

Input: A RCL(t_{sim}) for each rekeying algorithm, T_J^{sys}, T_D^{sys}
Output: A PRL for each rekeying algorithm

```
1.  for each RCL(t_sim) do
2.      for each request in RCL(t_sim) do
3.          Determine RT_J/D                    -- Equation 3.16
4.          Determine W_J/D                     -- Equation 3.17 or 3.18
5.          if request type = J then
6.              Determine T_J                   -- Equation 3.1
7.              Determine RQoS                  -- Equation 3.2
8.              PSP = (t_aJ, RQoS , ∞)
9.          else
10.             Determine T_D                   -- Equation 3.4
11.             Determine RAC                   -- Equation 3.5
12.             PSP = (t_aD, ∞, RAC)
13.         end if
14.     Add PSP to PRL
15. return PRL of all algorithms
```

3.5.4.2 Complex Evaluation Process (EvalComplexSimResults)

Other simulation modes deliver a CRCL(T). The performance evaluator generates one performance simulation point PSP for each RCL(T) of CRCL(T). The first element of the

PSP tuple represents a n, λ or μ value for scalability, join dynamics or disjoin dynamics simulation, respectively. The second element represents the minimal rekeying quality of service $RQoS_{min}$ of all join requests in the observation time for the corresponding n, λ or μ value. Similarly, the third element represents RAC_{min} of all disjoin requests. **Algorithm 3.9** depicts the process EvalComplexSimResults for evaluating non-transient simulation results. The symbol ∞ in this pseudo code indicates an initial very large value of the corresponding metric.

Algorithm 3.9 EvalComplexSimResults

Input: A $CRCL(T_o)$ for each rekeying algorithm, T_J^{sys}, T_D^{sys}
Output: A PRL for each rekeying algorithm

```
1.  for each rekeying algorithm do
2.      for each RCL(T_o) of CRCL(T_o)do
3.          RQoS_min := ∞, RAC_min := ∞;
4.          for each request in RCL(T_o) do
5.              Determine RT_J/D                          -- Equation 3.16
6.              Determine W_J/D                           -- Equation 3.17 or 3.18
7.              if request type = J then
8.                  Determine T_J                         -- Equation 3.1
9.                  Determine RQoS                        -- Equation 3.2
10.                 if RQoS < RQoS_min  then  RQoS_min := RQoS
11.             else
12.                 Determine T_D                         -- Equation 3.4
13.                 Determine RAC                         -- Equation 3.5
14.                 if RAC < RAC_min  then RAC_min := RAC
15.             end if
16.         PSP = (n/λ/μ, RQoS_min, RAC_min)
17.     Add PSP to PRL
18. return PRL of all algorithms
```

3.6 Implementation

The rekeying benchmark was implemented in Java using the Eclipse Environment [Ec05]. The software architecture consists of two main components: the Graphical User Interface (GUI) and the actual simulation kernel, as depicted in **Figure 3.8**. In this figure the benchmark software is illustrated as a simplified class diagram according to the Unified Modelling Language (UML) [Ke05]. Note the assigning of the different classes to two packages denoted as *gui* and *kernel*. *BenchmarkAndAlgorithmManager* represents the central class in the package *kernel* and includes the main function. This class is associated with the class *SimulationSettings,* which receives its attribute values from the GUI class *SimulationSetup*. The execution of the benchmark causes the opening of a framework,

where several simulations can be executed. This point is indicated by the association relation between the classes *SimulationSetup* and *MainFrame*. *Simulation* is an abstract class, which is inherited by two different simulation classes: *TransientSimulation* and *ComplexSimulation*. Note that the class *ComplexSimulation* is also abstract and builds the base class for the other three simulation classes *ScalabilitySimulation*, *JoinDynSimulation,* and *DisjoinDynSimulation*. The association relation between the classes *TransientSimulation* and *ComplexSimulation* reflects the fact that each complex simulation is based on a frequented execution of the transient simulation.

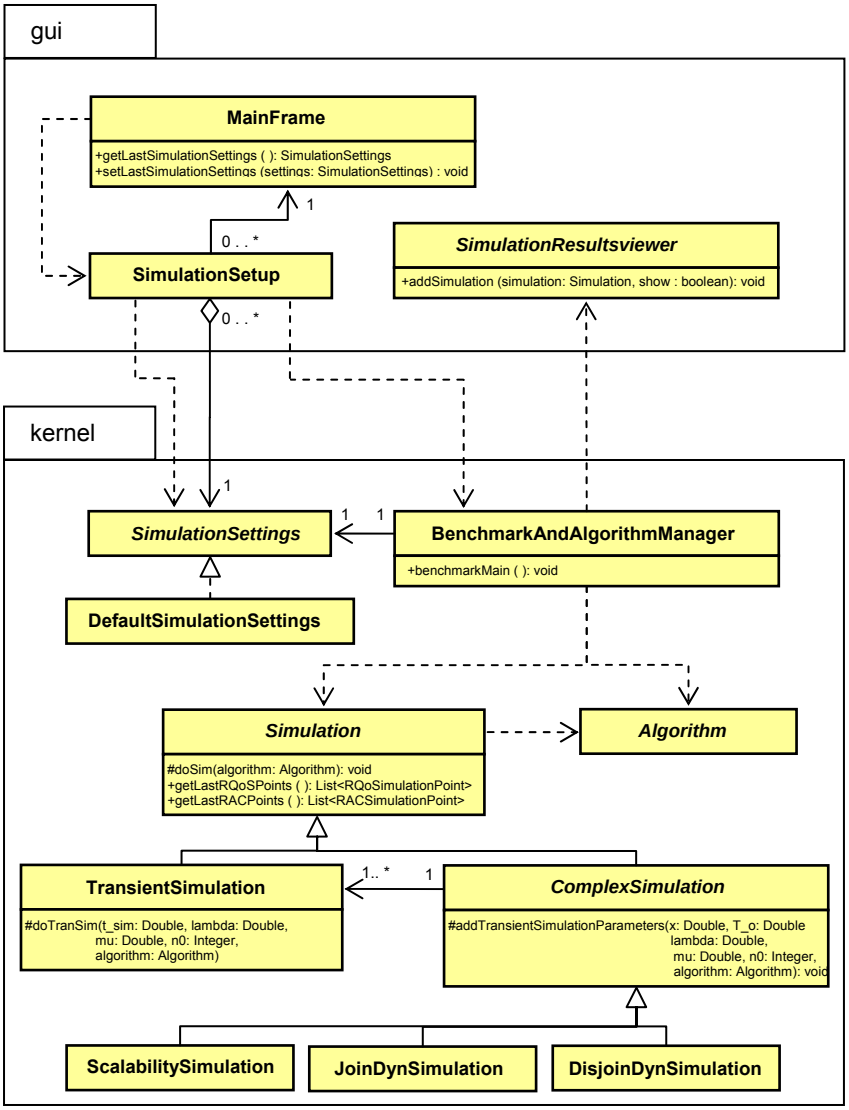


Figure 3.8. Rekeying benchmark class diagram

After program start, the simulation setup window displays default parameters. Changing these values is stored for a next simulation in the same session. See **Figure 3.9** for an overview of the simulation setup window. The set of all parameters belonging to one simulation are managed as a parameter list using the library class *LinkedHashMap*. This class is not shown in **Figure 3.8** for simplicity.

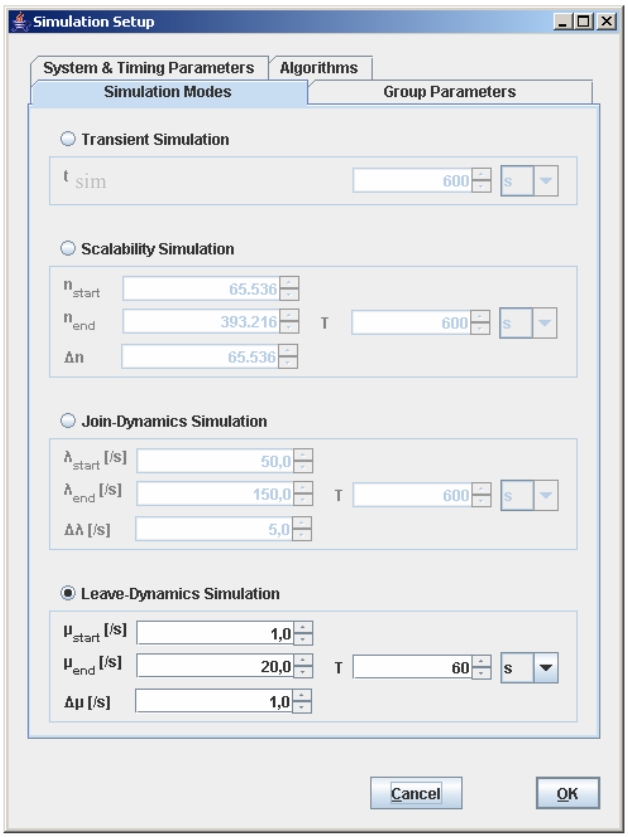


Figure 3.9. Simulation Setup window

3.7 Case Study (LKH Tree Rebalancing)

This case study relates to Example 3.2 presented at the start of this chapter, which depicted some divergence proposals in the literature regarding tree rebalancing for the logical key hierarchy algorithm. From investigating the related work given in this example it is obvious that a comprehensive analysis is needed to justify the employment of rebalancing, which is associated with additional rekeying costs resulting from shifting members between tree leaves. The rekeying benchmark offers this possibility by allowing a simultaneous evaluation of two LKH algorithms (with and without rebalancing) under complex conditions. Especially, the effect of disjoin rate is of interest in case of rebalancing, because

members leave the group in random manner, which disturbs the tree balance as a rule. Therefore, a disjoin dynamics simulation is performed under the following conditions:

$T_J^{sys} = T_D^{sys} = 100$ *ms*, $N_{max} = 65.536$,

$C_g = C_e = C_h = C_m = 1$ *µs*, $C_s = 15$ *ms*,

$n_0 = 4096$, $\lambda = 10\ s^{-1}$, $T_o = 1s$, $\mu_{start} = 1s^{-1}$, $\mu_{stop} = 20s^{-1}$, $\Delta\mu = 1s^{-1}$.

The simulation result clearly unveils that rebalancing degrades both RQoS and RAC values. This performance deterioration increases with an increasing disjoin rate. Simulation results are depicted in **Figure 3.10** and **Figure 3.11**.

The results of this case study unambiguously demonstrate that additional rekeying costs associated with rebalancing exceed the performance gain achieved by it. Consequently, rebalancing is not advantageous for LKH trees, at least under the given simulation conditions.

Currently, related work argues for rebalancing as a way to prevent tree degradation, which results in linear rekeying costs with respect to the group size in the rather extreme case of a very high disjoin rate. The main point, which is disregarded in this argumentation, is that the group size in such rare cases is very small and almost equal to the LKH tree height in the balanced case.

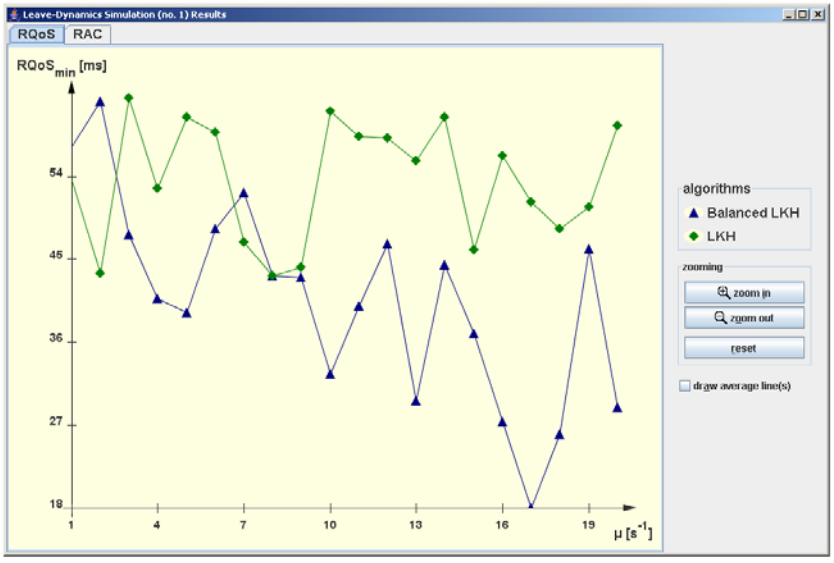


Figure 3.10. RQoS in rebalanced vs. non-rebalanced LKH

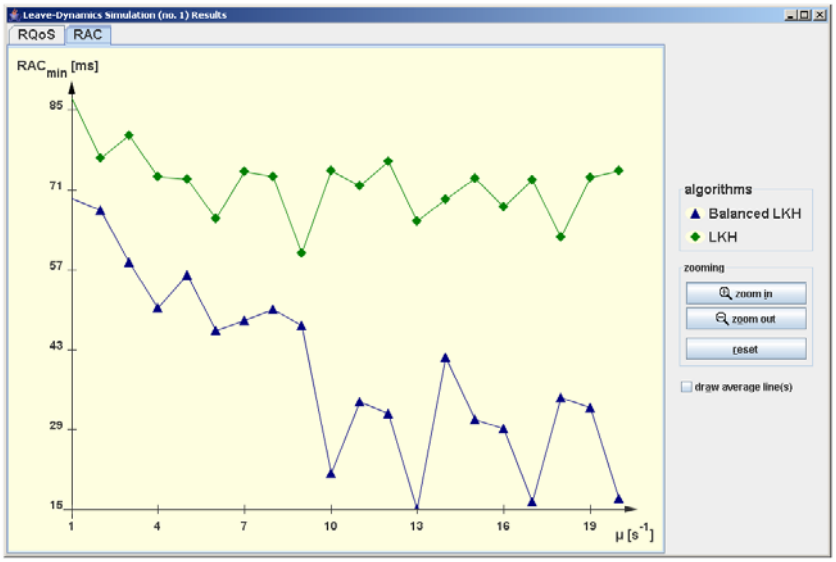


Figure 3.11. RAC in rebalanced vs. non-rebalanced LKH

4 Reconfigurable Architectures

4.1 Overview

This chapter presents an overview of reconfigurable architectures. Section 4.2 illustrates the current trends in chip design and the role of field programmable gate arrays. In Section 4.3 a brief overview of the FPGA architecture and configuration technologies is provided. Section 4.4 outlines both the hardware and the HW/SW design processes for FPGAs. Section 4.5 concludes this chapter with a summary of the hardware platforms and the design tools, which were utilized in the scope of this work.

4.2 Introduction

The advanced progress in semiconductor technology, coined by Moore's Law [Mo65], allows an ever-increasing integration scale and, thus, enables ever-newer and more sophisticated applications which can be realized on one chip. To follow this trend, innovative design and manufacturing strategies are required, which cope with the increasing complexity and allow short time-to-market (TTM) as one of the most important economic factors. The TTM of an integrated circuit consists of two components: the Time-to-Design (TTD) and the Time-to-Production (TTP). The design process of an integrated circuit begins with the problem specification and goes through several steps for design entry, synthesis, place & route, and different simulation and verification processes. At the end, the chip layout is provided, e.g., in form of data for the fabrication of lithographic masks. The production part TTP includes the time consumed by all the steps in the semiconductor manufacturing process beginning with the mask creation and ending with the chip packaging and test. While optimizing the TTM in the last decade was mostly driven by decreasing the TTD through optimizing the computer-aided specification and design tools [Mu00] and partially by reducing the TTP through semi-custom design strategies, the current decade is characterised by a unique and rather radical trend. Based on enhancing hardware with configurability properties, the whole TTP and the major part of TTD for layout design are canceled. Taking, additionally, the advances in the CAD for configurable design into account, TTM values are reached which are widely under those required for the competitive ASIC technology. This trend seems to conform to the prognosis made by Makimoto in 1986 about the progress in the semiconductor technology. Makimoto observed this development since the middle of last century and discovered a 10-year regularity regarding the standardization and customization in the IC market. This result was formulated in the form of a wave known as *Makimoto's Wave* [Ma00], which is

depicted in **Figure 4.1**. An illustration of this wave and an overview of future trends in chip design can be found in [Ma00] and [So06]. In spite of various kinds of programmed ICs the *Field Programmable Gate Arrays* (FPGA) represent the most known and utilized class of configurable architectures.

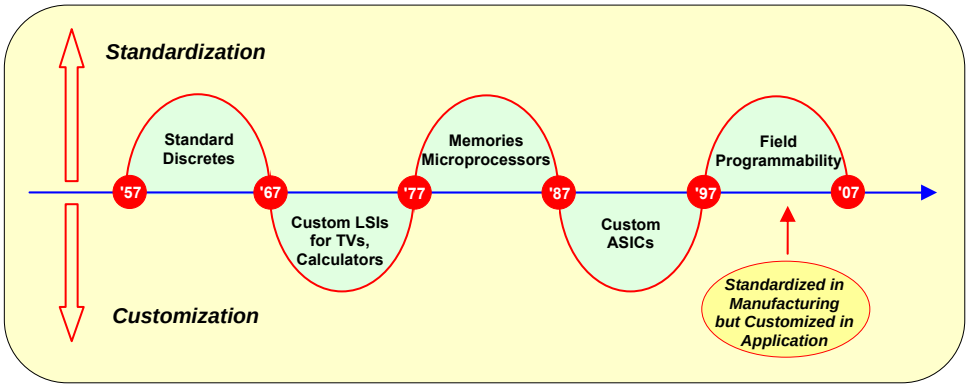


Figure 4.1. Makimoto's Wave

However, short time-to-market is not the only reason why designers currently start 81 percent of their projects using FPGA, see e.g. [Bu06] and [Dp06]. Other advantages of using these architectures include the lower development costs, the lower non-recurring engineering costs, the lower design risks, the providing of correctable and expendable designs, and the enabling of rapid prototyping [Bi06].

4.3 Field Programmable Gate Arrays

The concept Field Programmable Gate Array (FPGA) reflects two essential aspects of this IC technology. These relate to the electrical post-production configuration of these chips, on the one hand, and to the regular organization of the different components of these architectures, on the other.

Figure 4.2 illustrates a generic architecture of a fine-granular FPGA, which includes an array of configurable logical cells, input/output blocks, and routing resources. Modern FPGAs feature much more sophisticated architectures and include pre-manufactured coarse-granular components such as processors, high-performance multipliers, and dedicated memory blocks. Configurable logical cells provide both combinatorial function generators and registers to realize sequential circuits. The combinatorial function generators are either multiplexer based such as many products from Actel [Ac07], for instance, or look-up table based which are provided by Xilinx [Xi07] and other FPGA vendors. Multiplexer based FPGAs rely on the Shannon's expansion theorem which allows the implementation of any Boolean function using 2:1 multiplexers [Jo97]. Look-up table based FPGAs, in contrast, rely on the fact that memories are able to realize Boolean functions if the input variables and the function are connected to the address bus and the

output data bus of the memory, respectively. Depending on the saved data, different functions can be implemented.

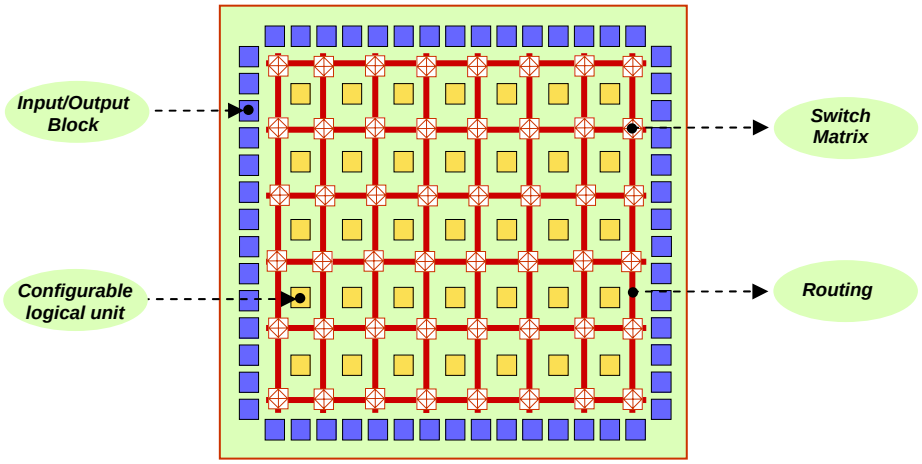


Figure 4.2. FPGA generic architecture

Regarding the configuration technologies three main FPGA classes can be identified: the SRAM-based, the EEPROM-based, and the Antifuse-based FPGAs. **Table 4.1** summarizes the advantages and disadvantage of these different technologies [Wa98]:

Table 4.1. Comparison between different FPGA configuration technologies

	Antifuse-based FPGA	SRAM-base FPGA	EEPROM-based FPGA
Advantages	✓ Good copy protection ✓ No configuration memory ✓ No reloading of configuration after start-up ✓ Small size and efficient	✓ No programming device ✓ Reconfigurability ✓ In-system programming	✓ Good copy protection ✓ No configuration memory ✓ No reloading of configuration after start-up
Disadvantages	✓ Programming device needed ✓ No reconfigurability ✓ No In-system programming	✓ Problematic copy protection ✓ Configuration memory needed ✓ Reloading of configuration after start-up	✓ Programming device needed ✓ Complex reconfigurability ✓ Conditional In-system programming

4.4 FPGA Design Process

Figure 4.3 depicts a typical hardware design process for FPGA. After an in-depth specification of the system requirements and constraints the design is entered as a functional model using a hardware description language such as VHDL [Ie93] or Verilog [Ie01]. Afterwards, a functional simulation is performed to validate the created model. In the case of functional correctness, a RTL-synthesis is applied to the model, which provides with the aid of vendor libraries a structural description of the design denoted as netlist. At this stage a pre-layout simulation can be performed to validate system timing. However, the

timing results provided by this simulation are approximated since the netlist has not yet been mapped to the target hardware and, therefore, no accurate information on routing delays is available. These delays can count for more than 75% of the total delay in modern FPGAs [Ku04]. In a next step the netlist components are placed onto the target architecture and connected using the FPGA routing resources. First at this stage an accurate timing simulation is possible. Provided that the timing results satisfy the system requirements, the bit stream can now be generated and written to the FPGA. Electronic Design Automation (EDA) such as synthesis and place & route relies on executing computer heuristics to find optimal solutions for several NP-hard problems [Ge05]. A typical example for these problems relates to finding the shortest route between two placed gates.

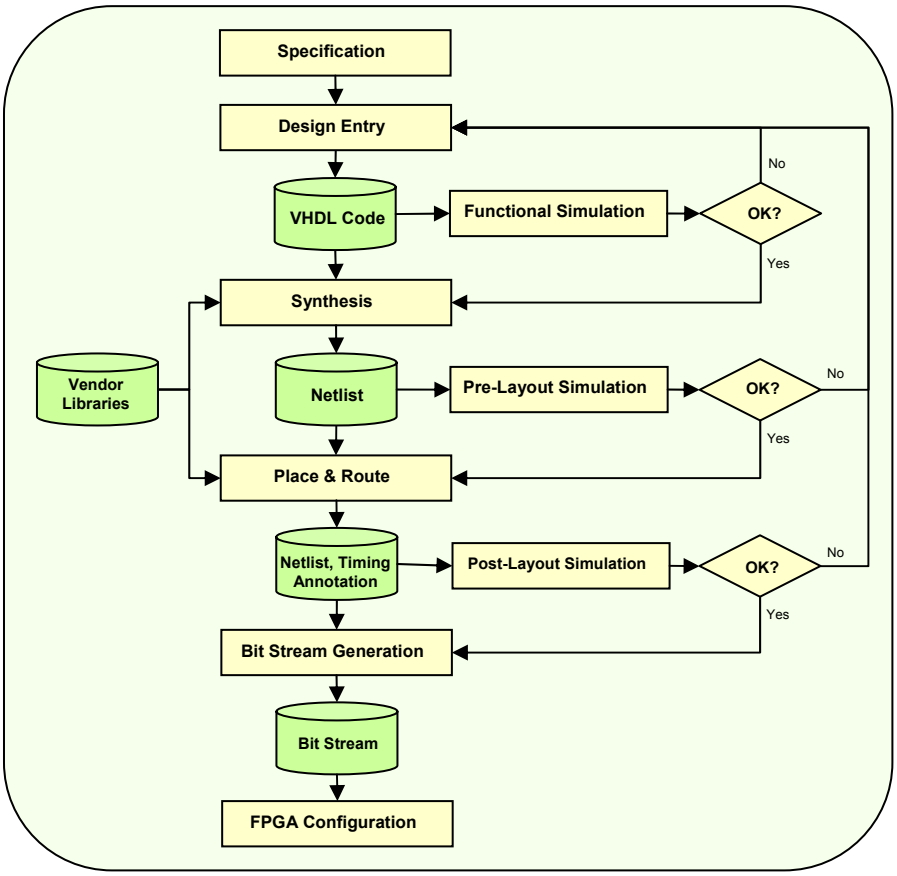


Figure 4.3. FPGA hardware design process

However, FPGA platforms do not only support hardware design. With the aid of several hardwired processor cores or using IP software processors, complete hardware/software solutions are facilitated. IP stands for *Intellectual Property* which can be seen as a product of the "*design for reuse"* paradigm, which wins an increasing recognition nowadays [Ke99], [Dr06].

For a HW/SW co-design, most FPGA vendors provide specific tools, which support separate hardware and software design processes to produce both a hardware and a software bit stream. In a last step both bit streams are merged and written to the FPGA. A verification of the applied HW/SW partition can first be performed at this late stage in the design process, which is highly inefficient. This design approach, which is mainly enforced by using the intellectual property, must be extended by the methods of high-level synthesis [Ga92], [Ga94]. Starting from high-level specification languages such as System-C [Gr02], an exploration of the design space can be applied to decide on a HW/SW partition with optimal resource allocation, and task scheduling and binding [Kl06]. **Figure 4.4** presents a simplified HW/SW design process, which starts with an executable specification of the system behavior followed by HW/SW partitioning phase. After a first cost estimation in this phase, the architecture components are allocated. Based on the system specification, the different tasks are then scheduled and bound to the different hardware and software resources. This step is evaluated by means of an objective function, which supplies an overall estimation of the selected partition based on predefined metrics such as performance and resource usage costs. If the values estimated by the cost function do not meet system requirements, another partition is searched and evaluated. Otherwise, the implementation phase is started, which is composed of both hardware and software design synthesis steps. In the end, two bit streams for the hardware and the software parts are generated and written to the FPGA.

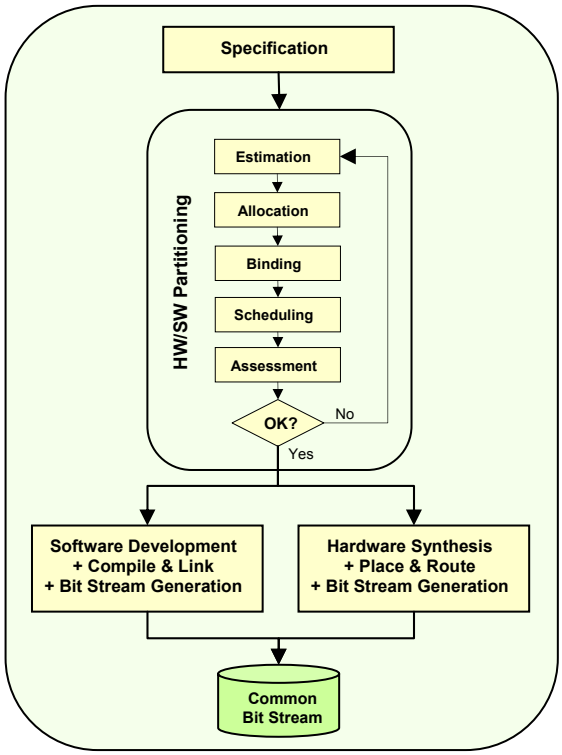


Figure 4.4. A possible FPGA hardware/software co-design process

4.5 Deployed Hardware Platforms

4.5.1 Virtex-II Pro

Xilinx Inc. launched the Virtex-II Pro family at the early 2002 with ten products of different size [Xi02]. **Table 4.2** summarizes some features of two members of this family which are used for the design of the rekeying processors described in next sections.

Table 4.2. Virtex-II Pro family members employed in the rekeying processors

	Configurable Logical Blocks	Block Select RAMs 18 Kb	PowerPC Processors	18*18 bit Multipliers	RocketIO Transceivers
XC2VP20	2320	88	2	88	8
XC2VP30	3424	136	2	136	8

A configurable logical block (CLB) is composed of four slices, where a slice mainly includes two look-up tables (LUT) as combinatorial function generators, two registers for sequential logics, large multiplexers, and fast carry look-ahead chain.

The block SelectRAM (BRAM) resources support synchronous single and dual port modes and can operate in several configurations ranging from 16K x 1 bit to 512 x 36 bit. The actual data size of a BRAM equals 16 Kbit, as each ninth bit is reserved for parity check.

Virtex-II Pro embeds up to 4 hardwired processor cores from the type IBM PowePC 405 (PPC405) [Xi05]. This 32-bit processor is characterised by a Harvard architecture and includes the functional blocks illustrated in **Figure 4.5**. With a clock frequency of about 300 MHz, instructions are executed in a five stage pipeline. Both the instruction and the data cache arrays are 16 KB with two-way set association, where a way includes 256 lines of 32 bytes each. The data cache unit supports both write-back and write-through modes. The memory management unit (MMU) executes several tasks including the translation of the 4 GB effective address space into physical addresses. MMU can be enabled or disabled according to system requirements. To improve the performance of virtual address translation the PPC405 includes dedicated hardware translation look-aside buffers (TLB), which contain parts of the page table.

PPC405 is compatible with the CoreConnect bus architecture [Ib99], which contains among other things two busses to connect the processor block with other system components. These are the Processor Local Bus (PLB) and the On-Chip Peripheral Bus (OPB). The PLB bus consists of 32-bit address, 64-bit data read, 64-bit data write, and 64-bit instruction buses and can operate at clock frequencies up to 100 MHz. The OPB bus is a simpler bus, which is used to connect slower peripheral cores such as a serial URAT interface. OPB bus is connected to PLB bus through a bus bridge which is specified in the CoreConnect architecture.

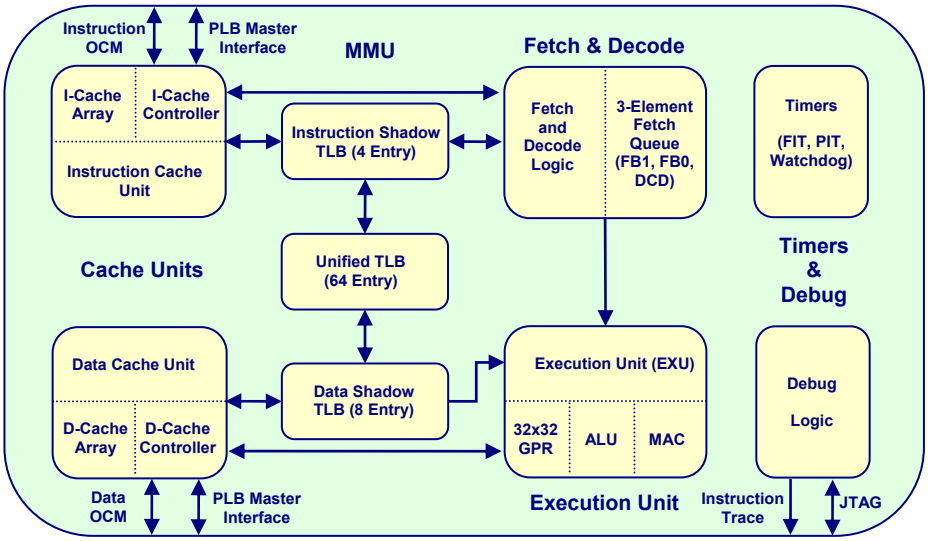


Figure 4.5. Embedded PPC405 Core block diagram

Furthermore, PPC405 has a dedicated memory interface, denoted as On-Chip Memory bus (OCM). The data side OCM bus (DOCM) has 32-bit data read bus, 32-bit data write bus, and 22-bit address bus. The instruction side OCM bus (IOCM) has 64-bit read only bus and 21-bit address bus. In contrast to PLB, memories connected to OCM bus can not be cached. The decision on OCM or PLB to connect system memory is generally difficult and strongly depends on the application. In general, different design alternatives are tested before deciding on the appropriate memory interface [Xi04]. Chapter 7 details this point for the design of the High-Flexibility Rekeying Processor.

4.5.2 Hardware Cards

In this work three different hardware cards were used, which are equipped with a Virtex-II Pro FPGA XC2VP20 or XC2VP30.

4.5.2.1 ADM-XRC-II Pro and ADM-XPL

Both cards are of PCI Mezzanine type and provided by Alpha Data Inc. [Al07]. ADM-XPL is equipped with XC2VP30. ADM-XRC-II Pro supports XC2VP20 and is specified as follows, see **Figure 4.6**.

1. Physically conformant to IEEE P1386 Common Mezzanine Card standard,
2. High performance PCI and asynchronous local bus,
3. Local bus speeds of up to 80MHz,
4. One bank of 256k or 512k x 64 pipelined ZBT SSRAM,
5. One bank of 64MB DDR SDRAM,

6. Two flash devices of 16MB each for bridge and target devices,
7. User clock programmable between 5MHz and 200MHz,
8. User front panel adapter with up to 146 free IO signals,
9. Support of 3.3V PCI or PCIX at 64 bits,
10. On board 125MHz LVPECL oscillator,
11. 4 x RocketIO Multi-Gigabit Transceiver connections.

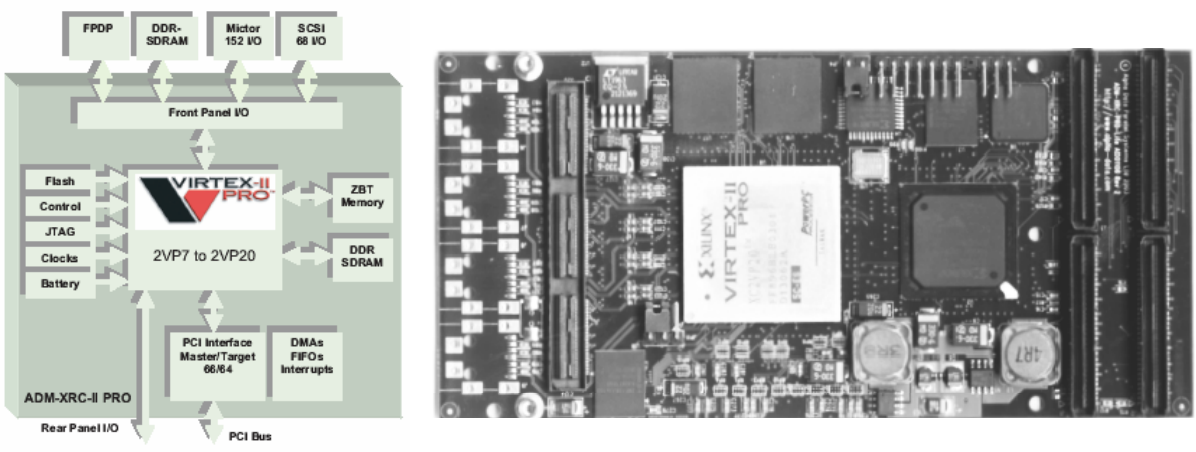


Figure 4.6. ADM-XRC-II Pro [Al07]

4.5.2.2 XUP

This borad is delivered by Digilent Inc. [Di07] and supported by Xilinx within the Xilinx University Program. It features, among other things, the following properties. See **Figure 4.7**.

1. Support of DDR SDRAM DIMM up to 2 Gbytes
2. 10/100 Ethernet port
3. USB2 port
4. Compact Flash card slot
5. XSGA video port
6. Audio Codec

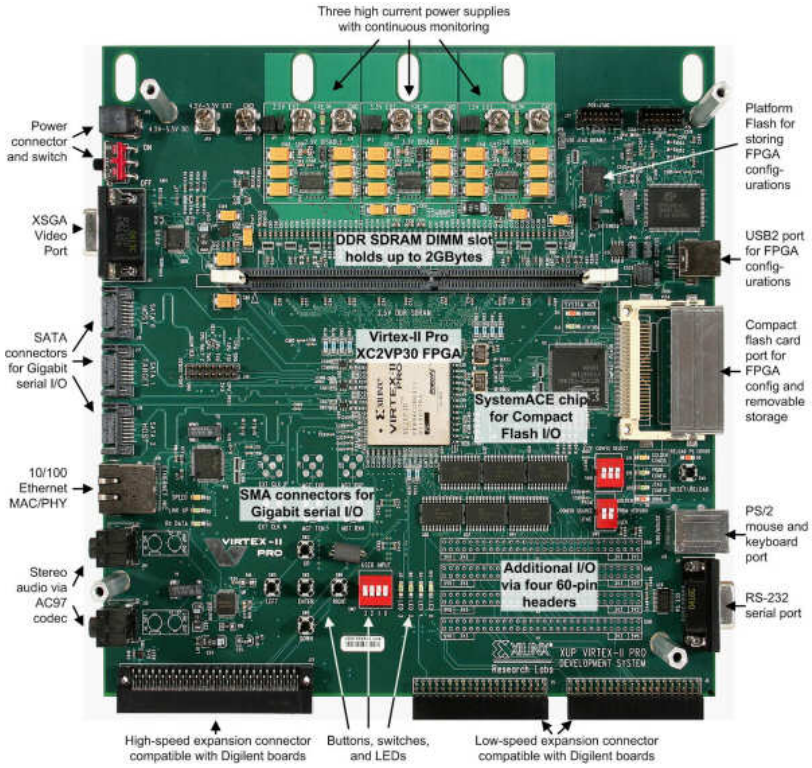


Figure 4.7. XUP Card [Di07]

Table 4.3 depicts the tools and programs, which were employed in the design of the rekeying processors.

Table 4.3. CAD tools employed for the rekeying processor design

Usage	Tool
VHDL functional simulation	ActiveHDL [Ad07]
VHDL RTL synthesis	Synplify Pro [Sy07]+ XST [Xi07]
Place & Route and bit stream generation	ISE [Xi07]
Software development	GNU [Gn07]
HW/SW co-design	EDK [Xi07], hCDM [Kl06]
Measurement, data preparation, and display	Mathematica [Wo07]

5 New Architectures for Group Rekeying

5.1 Overview

Because of several similarities between the hardware and HW/SW rekeying architectures presented in next chapters, this chapter presents a general introduction to these architectures and describes their common features. Section 5.2 depicts the deployment scope of the new solutions and how they distinguish themselves from others. Section 5.3 points out the rekeying security requirements. All rekeying processors are specified by a generic architecture consisting of four main components. This generic architecture is presented in Section 5.4. The different components and several associated aspects are then illustrated in the last four sections

5.2 Introduction

As mentioned in Chapter 1, research work on secure multicast began in the late nineties and crystallized by establishing the Secure Multicast Group (SMuG) in 1999 and the working group Multicast Security (MSEC) in 2000 at the IETF, see [Sm98] and [Ms00]. To overcome the complexity of secure multicast, these groups defined a reference framework, which classifies the different subjects in secure multicast into three problem areas. In addition, each problem area is associated with one or more functional blocks to facilitate a modular design and standardization process. For the problem area of key management, a functional block denoted as *Key Server* was suggested [Ha03]. This block takes the responsibility for the group rekeying task.

The introduction of new rekeying architectures in this work is oriented towards this reference framework. Each of the hardware and hardware/software solutions proposed in the next chapters can be regarded as a realization variant of the key server defined by the multicast working groups. The proposed architectures include:

1. The *Real-Time Rekeying Processor* (RTRP), see [Sh04].
2. The *Batch Rekeying Processor* (BRP), see [Sh05].
3. The *High-Flexibility Rekeying Processor* (HiFlexRP), see [Sh07b] and [Sh07c].

As a generic name for all these architectures the term *Rekeying Processor* (RP) will be used in this chapter, as long as no differentiation is needed. Furthermore, it is assumed that the

rekeying processor operates in the server environment of a group owner, which provides secure multicast content using a dedicated Data Server (DS), e.g. a video server in the case of Pay-TV multicast, see **Figure 5.1**. In addition, a dedicated Registration and Authentication Server (RAS) is employed to register members, to provide identity keys, and to manage rekeying with the aid of the rekeying processor as follows. The RAS sends rekeying requests to the RP in form of instructions, e.g. "join a member". The RP executes these instructions and writes rekeying messages back to the RAS, which sends them to the group members. Alternatively, the RP can send rekeying messages directly to the members, if it supports a networking interface. Most rekeying requests result in a new group key, which is provided by the RP to the data server DS, which uses this key to encrypt data. See **Figure 1.4** for a comparison and **Table 1.4** for the definition of some important terms such as identity key and rekeying message.

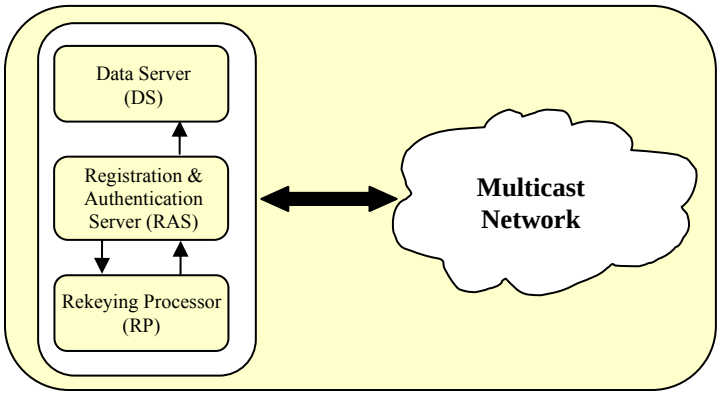


Figure 5.1. Rekeying processor in a server environment

Related work on rekeying optimization relies on minimizing the number of rekeying submessages, i.e. the number of cryptographic operations needed to build these messages. The rekeying processor presented in this work, however, optimizes rekeying performance mainly on the cryptography and the platform layers. In addition, several improvements on the rekeying layer are proposed. This aspect is illustrated in **Figure 5.2**, which represents group rekeying as a three-layer abstraction model. The RP performance optimization on the rekeying layer includes, for instance, the pipelined batch rekeying and the event-driven batch rekeying illustrated in Chapter 2. Another example relates to the key tree management: the presented architectures cause the key tree to grow up from the root side and avoid leaf splitting in the case of full trees. This results in major performance improvement.

Rekeying optimization on the lower layers relies on dedicating a hardware platform for the rekeying task and optimizing the cryptographic operations on this platform. Some cryptographic algorithms, such as the Advanced Encryption Standard AES, feature inherent parallelism and can operate on hardware highly efficiently. Other cryptographic algorithms, like the elliptic curve algorithms, are not designed with parallelism properties. An efficient realization on a hardware platform can be based on segmentation and reassembling of data before and after processing, respectively [Er02], [La06].

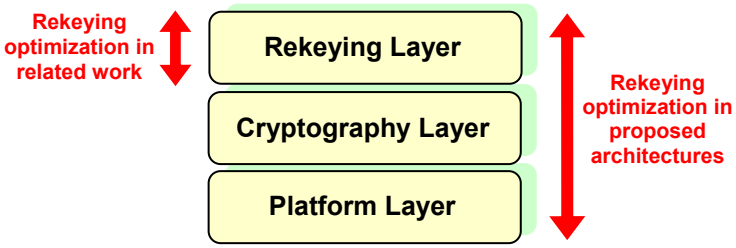


Figure 5.2. Rekeying processor operation layers

5.3 Rekeying Security Requirements

Group Rekeying is a mechanism for access control to ensure data confidentiality. Rekeying data themselves must be protected against manipulation. In total, group rekeying is associated with three security requirements:

1. ***Access control:*** This requirement represents the original demand for backward and forward access control to prevent new members from decrypting old data and leaving members from eavesdropping on future communication, respectively. Together with data encryption, access control ensures the confidentiality of delivered data.
2. ***Group authentication:*** Rekeying data are largely crucial and must be protected against manipulation. Group authentication is the security requirement, which ensures that rekeying data originate from a group member and not from outside. The exact identity of the data sender, however, can not be determined by this authentication mode. The reason is that group authentication employs methods, which are based on Message Authentication Code (MAC). MAC relies on encrypting the rekeying data with a key shared by all members. Group authentication is used in multicast groups where members trust each other, e.g. in collaborative working groups, which use secure multicast for video conferencing.
3. ***Data source authentication:*** Groups with lower trust between members need to exactly identify the sender of rekeying messages. Furthermore, in highly secure systems rekeying messages must be provided with a non-repudiation property to enable a legal authority to verify the data source in the case of denying. MAC-based approaches are not able to fulfil any of these two security requirements for group communication, because the authentication key is shared between all members. One way to realize source authentication for rekeying data relies on using digital signatures, which can be applied to the hash value of a rekeying message.

Section 5.6 presents the security modules deployed in the rekeying processors to realize these security requirements. **Figure 5.3** depicts the three security levels and the supporting architectures. While the Real-Time Rekeying Processor and the Batch Rekeying Processor only fulfil the access control requirement because of the historical development of this work, the High-Flexibility Rekeying Processor satisfies all the security requirements. Higher security, however, comes at the expense of performance, as can be seen in this figure.

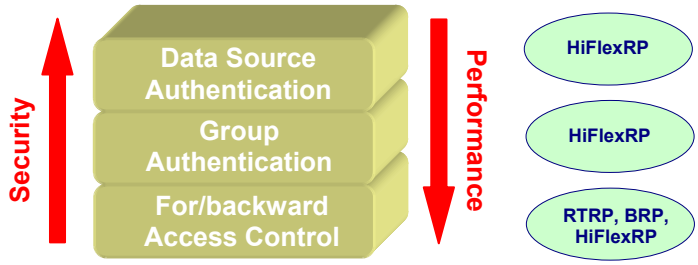


Figure 5.3. Security levels of rekeying architectures

5.4 General Architecture

The proposed rekeying processors are characterized by the general architecture depicted in **Figure 5.4.** The input and output units construct the interface of the rekeying processors to the registration and authentication server according to **Figure 5.1**. Generally, the security unit includes cryptographic primitives for key generation, encryption, secure hashing, message authentication code, and digital signature. The security primitives are implemented as hardware modules in the case of the RTRP and the BRP. The HiFlexRP, however, realizes the security functions partially in software according to the selected HW/SW partitioning [Sh07c]. LKH rekeying is a data-intensive task, where data elements are represented by the tree keys. To reduce data transfer between the RAS and the RP, all keys are generated, saved, and managed on the hardware. In addition to the performance gain, this approach enhances system security due to the hardware storage of these secure data. Except for the group key, which is needed by the data server to encrypt useful data, all other keys are transferred from the RP to the RAS only in an encrypted form. The key tree unit includes both the key storage and the necessary functions to mange the key tree. Two tree management modes are used: the static and the dynamic tree management. The static tree management is specific to the rekeying processors and uses hardware features for this purpose. In contrast, the dynamic tree management is similar to known software solutions. Therefore, this mode is only supported by the HiFlexRP.

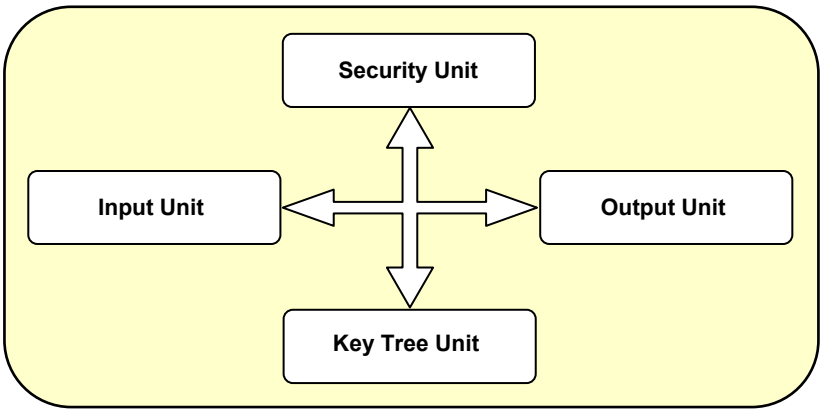


Figure 5.4. General architecture for the rekeying processors

5.5 Key Tree Management

In this section some issues of the key tree management are illustrated. The description is limited to the static tree management, which is used in all rekeying processors. As the dynamic tree management is only specific to some design alternatives of the HiFlexRP, this tree management mode will be first described in Chapter 7.

5.5.1 Key Memory Architecture

All proposed architectures in this work perform group rekeying based on the LKH algorithm with binary trees. The LKH was described in Section 1.4. In principle, trees represent dynamic data structures, which expand and contract according to the current amount of data to be represented. Adapting the tree size to available data reduces the demand for physical memory, which may be shared with other applications. Using a hardware platform for the rekeying task, however, allows allocating some amount of memory dedicated to the key tree. By this means a tree structure can be associated with the statically allocated memory. This association follows the following rules, which are illustrated schematically for a group of 8 members in **Figure 5.5.**

1. The size of allocated memory M_{size} depends on the key length K_{length} and on the maximal group size N_{max}. For binary trees, N_{max} is always assumed to be a power of two, thus:

$$M_{size} = (2N_{max} - 1) \cdot K_{length} \qquad (5.1)$$

2. The lower half of the memory space is dedicated to member identity keys belonging to the first tree level.
3. The upper half is reserved to store the help-keys and the group key, level by level.

Level	Address	Key
0	0000	k_0
	0001	k_1
	0010	k_2
	0011	k_3
	0100	k_4
	0101	k_5
	0110	k_6
	0111	k_7
1	1000	k_{0-1}
	1001	k_{2-3}
	1010	k_{4-5}
	1011	k_{6-7}
2	1100	k_{0-3}
	1101	k_{4-7}
3	1110	k_{0-7}

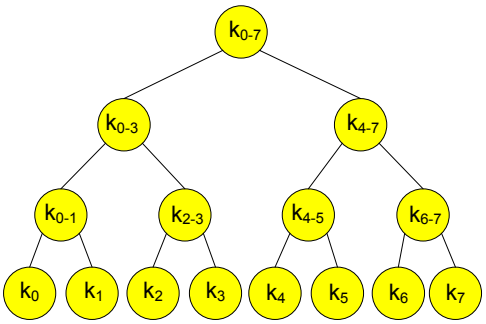


Figure 5.5. Memory architecture for key trees

Dynamic tree management using software techniques relies on operations to insert or remove a tree node or leaf. Therefore, in this management mode the key tree has as many nodes and leaves as the number of keys used currently. In contrast, the insertion and remove of nodes and leaves are not defined in the static tree management since a memory place is allocated for all nodes and leaves all the time. However, in general case not all nodes and leaves contain valid keys. To address this point, additional specification is required to indicate whether a tree node or leaf is in operation or not in the static tree management. This specification is given as follows. Refer to **Example 5.1** and **Figure 5.6** below for an illustration of the following terms.

Definition 5.1:

A *valid key* is an identity key, a help-key, or a group key, which is used by at least one group member.

Definition 5.2:

An *active leaf* is a tree leaf which contains a valid identity key.

Definition 5.3:

A *suspended leaf* is a tree leaf which does not contain a valid identity key.

Definition 5.4:

An *active node* is a tree node which contains a valid key.

Definition 5.5:

A *suspended node* is a tree node, which does not contain a valid help-key and there are no members whose paths to the root pass this node.

Definition 5.6:

A *right suspended node* is a node, which does not contain a valid help-key, however, there is at least one member whose path to the root passes this node from the *left*.

Definition 5.7:

A *left suspended node* is a node, which does not contain a valid help-key, however, there is at least one member, whose path to the root passes this node from the *right*.

Note 5.1:

In the schematic representation of a tree in this work, active nodes and leaves appear grey, all other nodes and leaves are drawn transparent. This notation applies to the case of static tree management only.

Example 5.1:

Figure 5.6 shows an example for the specification of tree nodes and leaves in static tree management. In this example three suspended leaves, one suspended node representing $k_{0\text{-}1}$ and two left suspended nodes representing $k_{0\text{-}3}$ and $k_{6\text{-}7}$ are currently available.

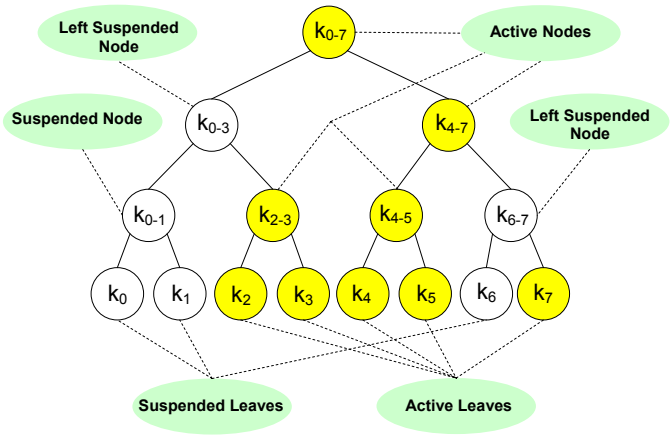


Figure 5.6. Node/leaf type example

5.5.2 Key State Memory

Specifying a tree node as active, suspended, right or left suspended is essential for tree traversing and for the decision on the appropriate operations to be performed on the corresponding key. In the rekeying processors a dedicated memory denoted as *Key State Memory* (KSM) is employed to save information on the state of all tree nodes. The word width of this memory is 2 bit as a node can have one of four modes according to the previous section. **Table 5.1** illustrates the coding of these words which are denoted as *LR words*.

Table 5.1. LR word code for a tree node

LR word	State of corresponding node
00	Suspended
01	Left suspended
10	Right suspended
11	Active

The depth of the key state memory equals the number of the help-keys represented by the key memory. For efficient traversing the address space of the key state memory corresponds to that of the help-keys in the key memory after neglecting the most significant bit. **Table 5.2** shows the key state memory for the tree presented in **Figure 5.6.**

Note 5.2:

Tree leaves can be active or suspended. For a rekeying algorithm, this information can be extracted from the LR word of the corresponding father. Due to this property the state of a key leaf does not need to be saved.

Table 5.2. Key state memory of the tree in **Figure 5.6**

Level	Address	Help-key	LR
1	000	k_{0-1}	00
	001	k_{2-3}	11
	010	k_{4-5}	11
	011	k_{6-7}	01
2	100	k_{0-3}	01
	101	k_{4-7}	11
3	110	k_{0-7}	11

Note 5.3: KSM Implementation

The rekeying processors are implemented on hardware platforms with FPGAs and DDR-SDRAM memories, among other things. To support large groups the key memory is mapped to the DDR-SDRAM. In contrast, because of the long access times of this DDR-SDRAM the KSM is implemented using the Block RAMs (BRAMs) of the FPGA. By this means, LR words can be proved earlier and the decision on the next operation can be made faster. However, because of their limited number and size, BRAMs set a constraint on the maximal group size, as will be seen in the next chapters.

5.5.3 Tree Traversing

In the presented static tree management, traversing relies on the physical tree structure, which allows an efficient node visiting based on simple logical or arithmetic operations on memory addresses. As will be seen in Section 5.5.4, each key in the tree is identified by its physical memory address. This means that for a tree traversing a starting key address is always required to find the relatives of this key in the tree. **Table 5.3** illustrates how to find the father, the sons and the brother of a node identified by a memory address A. See **Figure 5.5** for some examples.

Table 5.3. Estimation a relative node of a node with address A

Relative node	Address of the relative node
Father	Right shift of A with 1 insertion from left
Left Son	Left shift of A with 0 insertion from right
Right Son	Left shift of A with 1 insertion from right
Brother	A + 1

Note5.4:

The traversing strategies apply both to the key memory and to the key state memory. As the processing of a key relies on its LR word, the KSM memory is traversed to decide on the corresponding key state. If this key need to be updated, for example, then the address of the corresponding help-key is estimated from the address of the LR word by extending it with logical '1' from the left side. The key k_{0-3}, for example, is saved at the address 1100, its LR word at the address 100, see **Figure 5.5** and **Table 5.2**.

For the functionality of the rekeying processors, three traversing modes are introduced:

Definition 5.8: Path Traversing

Path Traversing is a visiting of all nodes locating on a tree path from a leaf to the root. The next node in this traversing is called a father.

In the RTRP and the HiFlexRP this traversing mode is used to determine the keys to be updated. In the BRP path traversing is performed in the marking step of batch rekeying to mark the keys to be processed later.

Definition 5.9: Reverse Path Traversing

Reverse Path Traversing is a traversing from the root or a node to the next active node or leaf. The next node/leaf in this traversing is called a right or a left son.

This traversing mode is used only by the RTRP and the HiFlexRP. While path traversing determines a key $k_{x\text{-}y}$ to be updated, reverse path traversing is performed to find the keys, with which the new $k_{x\text{-}y}$ must be encrypted to build the rekeying submessages. Reverse path traversing may result in a help-key or in an identity key.

Example 5.2:

Assume that member m_3 must be removed from the group presented in the left tree of **Figure 5.7**. A path traversing started at k_3 shows that only the group key $k_{0\text{-}7}$ must be updated, as $k_{2\text{-}3}$ will not be used any more. To find the keys, with which $k_{0\text{-}7}^{new}$ must be updated, two reverse path traversing processes are executed beginning with $k_{0\text{-}7}$. The inverse traversing results in the keys k_2 and $k_{4\text{-}7}$, see the right tree. Thus the following rekeying submessages are constructed: $RSM_1 = E_{k_2}(k_{0\text{-}7}^{new})$ and $RSM_2 = E_{k_{4\text{-}7}}(k_{0\text{-}7}^{new})$.

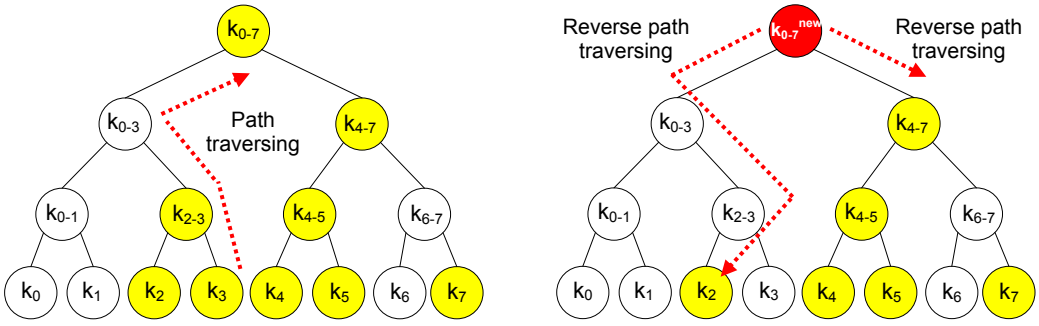


Figure 5.7. Tree traversing example

Definition 5.10: Level Traversing

Level Traversing is a visiting of all nodes belonging to some tree level from left to right. The next node in this traversing is called a brother. This traversing mode is used in batch rekeying during processing to find out the marked keys. Level traversing ensures that the help-keys of some level are only processed after updating the keys of lower levels. This traversing mode will be detailed in the scope of the BRP in next chapter.

5.5.4 Rekeying Submessage Identification

Rekeying messages are sent per multicast. As a result, each member gets all rekeying submessages. In the last example the rekeying submessages RSM_1 and RSM_2 are received by all members. However, member m_2, for instance, is only interested in RMS_1. For this member to only decrypt this rekeying submessage, a kind of identification for these messages must be provided. The proposed architectures use an identification mechanism for rekeying submessages, which relies on identifying all tree keys as follows.

Definition 5.11

A Key Identity (KEYID) is defined as the address of this key in the physical memory.

Definition 5.12

A Member Identity (MEMID) is defined as the address of the identity key of this member in the physical memory.

Note that the MEMID of a member corresponds to the KEYID of the identity key of that member.

Definition 5.13

A Rekeying Submessage Identity (RSMID) is a pair (*x, y*), where *x* represents the KEYID of the encrypted key and *y* refers to the KEYID of the encrypting key.

The rekeying message identification mechanism can now be summarized as follows:

1. During registration, each member is supplied with the KEYIDs of all keys on the path from the corresponding leaf of that member to the root, even if some help-keys on this path are not active.
2. During rekeying, each rekeying submessage is associated with the corresponding RSMID.
3. Getting a rekeying submessage, a group member extracts x from the corresponding RSMID and decrypts the message only if *x* belongs to the KEYIDs saved by this member.

Example 5.3:

Referring to **Example 5.2**, the rekeying submessages RSM_1 and RSM_2 are delivered with the RSMIDs depicted in **Table 5.3**.

Table 5.3. Rekeying submessage identity

RSM	$E_{k_2}(k_{0\text{-}7}^{new})$	$E_{k_{4\text{-}7}}(k_{0\text{-}7}^{new})$
RSMID	(1110,0010)	(1110,1101)

5.6 Hardware Security Modules

As mentioned in Section 5.3 the Real-Time Rekeying Processor and the Batch Rekeying Processor fulfil the security requirement of access control. The High Flexibility Rekeying Processor ensures, in addition, group authentication or data source authentication, depending on the system requirements. **Table 5.4** summarizes the used cryptographic primitives for the different security levels.

Table 5.4. Cryptographic primitives used in the rekeying processors

Security level	Utilization in	Cryptographic method	Cryptographic primitive
Back- and forward access control	RTRP, BRP, HiFlexRP	Encryption	AES-128
		Key generation	AES-based PRNG, ANSI X9.17
Group authentication	HiFlexRP	Message Authentication Code (MAC)	AES-based MAC
Data source authentication	HiFlexRP	Secure hash function	AES-based Meyer hash function
		Digital signature	ECDSA

Regardless of the digital signature, all other cryptographic operations are based on the Advanced Encryption Standard (AES) with a key length of 128 bits. As will be seen in the next sections, a key generation using the Pseudo Random Number Generator (PRNG) specified in ANSI X9.17 is based on two encryptions. Meyer hash function executes an encryption to each block of data to be hashed. The same applies to block cipher-based MAC. This design strategy has the following advantages depending on the underlying executing platform:

Platforms with constrained resource usage:

All the listed block-cipher based primitives for key generation, MAC and hash value determination rely on encryption as a central operation and some XOR operations which are inexpensive. In resource-limited platforms this enables a considerable hardware saving, if only one encryption primitive is implemented and shared by all rekeying cryptographic tasks, excluding digital signature.

Platforms with unconstrained resource usage:

Using a dedicated encryption module to each cryptographic primitive is largely convenient for the rekeying task, if the resource usage is not constrained. This results from the fact that in LKH-based rekeying the number of generated keys equals the half of the number of rekeying submessages. In addition, the processing of a rekeying submessage in the hash or MAC function costs each one encryption. Consequently, using AES-based key generation,

encryption, hash function and MAC for LKH-based rekeying results in the same number of AES executions for each of these operations which enables an efficient rekeying pipelining.

To illustrate this point the case of rekeying with data source authentication is considered. An LKH-based rekeying can be considered as an iterative update of some tree keys, followed by a calculation of a digital signature. In this respect, a key update operation includes:

1. a generation of a new key,
2. two encryptions of the new key with both its sons, and
3. two hash operations on the two resulting rekeying submessages.

Example 5.4:

Referring to **Figure 5.7**, two key updates are required if member m_5 has to be disjoined. For this purpose two key generations are performed with two encryptions each. To build the rekeying submessages four encryptions are also needed. Each rekeying submessage is then entered to the hash module and operates as a key for one encryption, see Section 5.6.3. Accordingly the hash function executes four encryptions, too. **Figure 5.8** illustrates schematically the schedule of the different tasks for this example, where the labels are interpreted as follows.

G: Generating a key

E: Encrypting of a key with one of its sons

H: Hashing of a rekeying submessage

S: Signing the hash value of the rekeying message

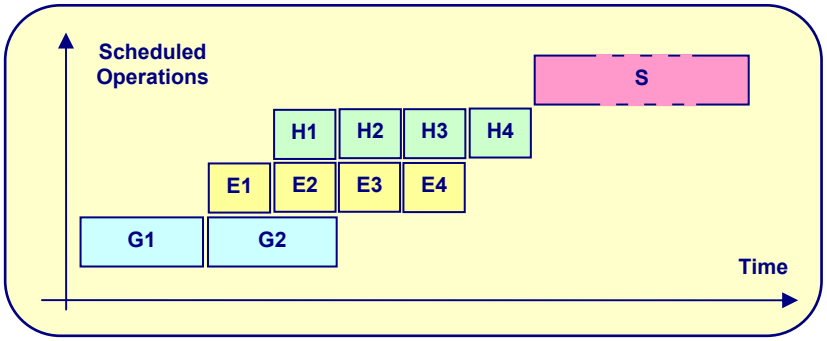


Figure 5.8. Pipelined key update

Note 5.3:

The previous investigation does not consider the rekeying submessage identities. These data are critical and their protection against manipulation is as essential as for rekeying submessages themselves. A RSMID, however, is shorter than a rekeying submessage. To hash these data two RSMIDs are concatenated and entered into the hash module as one data block, see **Table 5.6** for the widths of the different RP data words. This can be done

between the hashing stages of the rekeying submessages or at the end of this directly before the digital signing. The last alternative preserves the pipeline structure.

5.6.1 Encryption Module

All the proposed rekeying processors use the Advanced Encryption Standard (AES) with a key length of 128-bit [Da02], [Ni01]. AES is a block cipher which processes 128-bit data blocks. Each data block is organized as a matrix of 4x4 bytes and processed in nine identical iterations and a last slightly different one. Except for this, each iteration, denoted as round, includes four processing steps of byte substitution, row shift, column mixing, and adding a round key. In the last round the step of column mixing is omitted. Round keys are expanded keys which, are derived from the original key by applying a key schedule process.

Figure 5.9 provides a general view of the encryption module. To perform an encryption the data and the key are first written on the inputs DATA_IN and KEY, respectively, followed by activating the control signal START. The module performs encryption and sets the encrypted data on the output DATA_OUT and signalizes this by setting the signal RDY. During encryption the module does not react to other requests. The inputs DATA_IN and KEY and the output DATA_OUT have all a length of 128 bit.

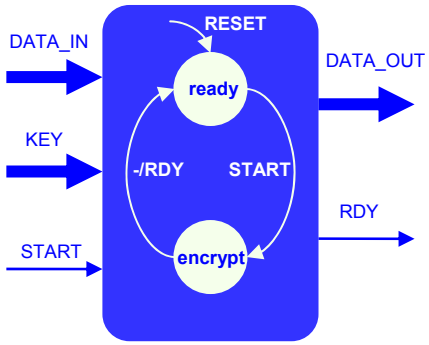


Figure 5.9. Encryption module overview

In the course of this work several realizations of the AES module have been designed with different resource usage and performance figures. In general, the AES module is implemented using an iterative looping architecture with sub-pipelining. Iterative looping means that only one round is implemented in hardware. To perform an encryption a data block is entered into this round 10 times. The purpose of sub-pipelining is to shorten the critical path of the round, which enables higher clock frequencies. One design alternative of the AES relies on two pipelining stages, as depicted in **Figure 5.10.** Note that the missing of the pipeline register behind the Block RAM of Virtex-II Pro is justified by the synchronous functionality of these memories. The module Key Schedule operates synchronously with the encryption path to provide a round key for each iteration on the fly. In total, an encryption using this architecture lasts about 25 clock cycles. For other hardware realization possibilities of the AES refer to [Ma03], [Zh04], and [Go05].

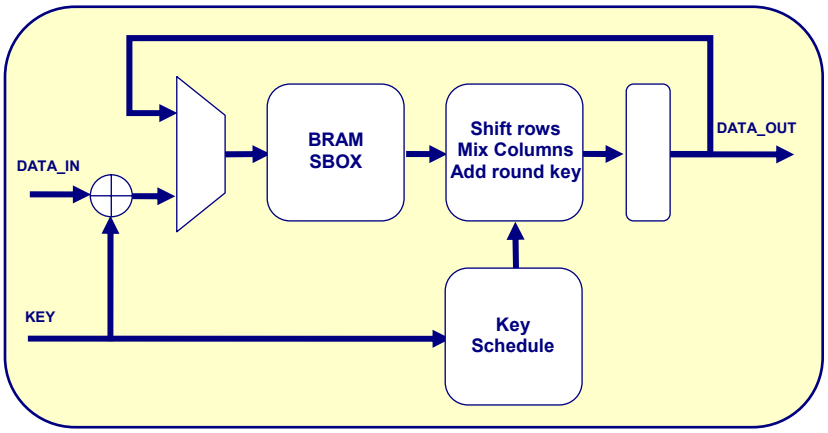


Figure 5.10. AES architecture in HiFlex RP

Block ciphers used for rekeying solutions in related work, e.g. [Wo00], run in complex modes such as the Cipher Block Chaining (CBC), which operates as follows. Before its encryption, a data block is mixed with the encryption result of the previous block to prevent that the same data block results in the same cipher block.
In contrast, the rekeying processors deploy the AES in the simple Electronic Code Book Mode (ECB), i.e. data blocks (tree keys, actually) are encrypted independent of each other. This decision on the operation mode for AES in the rekeying processors is justified as follows:

1. The security problems of ECB do not appear in rekeying encryption. This is because the encrypted data are keys, which are randomly generated and not plain text with known patterns allowing for cryptological analysis.
2. Using other encryption modes causes that each member has to decrypt all rekeying submessages, even if not all these submessages are interesting for that member, see Section 5.5.4

5.6.2 Key Generator

ANSI X9.17 is a key management standard for financial institutions published by the American National Standard Institute [Ni85]. Among other things this standard specifies a key generator based on symmetric-key encryption such as 3DES. However, other block ciphers are allowed according to the generator specification. Therefore, the rekeying processors use AES-128 for this task. For the functionality of this generator three data are needed, which are denoted as the *generator initialization data* in the scope of this work. These data are the key generator K_{gen}, the timestamp D and the initial seed S_0. Because of using AES-128 all these data words have a length of 128 bit. The generation process relies on an initial encryption of the timestamp D with the key K_{gen} and then two encryptions to generate a key and to determine the next seed as depicted in the following steps, where $S_i = S_0$ for the first generated key:

1. $I = E_{Kgen}(D)$

2. $k_i^{new} = E_{Kgen}\ (I\ xor\ S_i)$
3. $S_{i+1} = E_{Kgen}\ (I\ xor\ k_i^{new})$

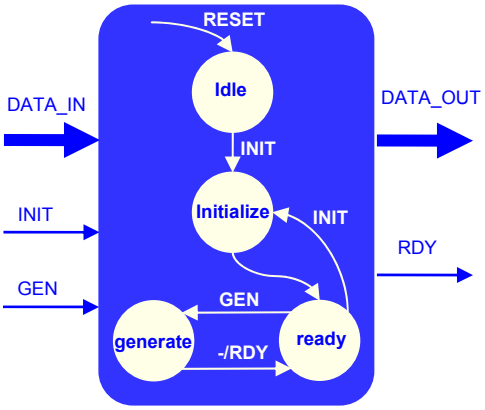


Figure 5.11. Key generator Overview

Figure 5.11 presents an overview of the realized key generator. After the system start-up the generator waits for the initialization data. These parameters are sent to the generator on the 128-bit input DATA_IN in the order: D, K_{gen}, and S_0, where the availability of each parameter on this input is signalized by a logical '1' on the control input INIT. Upon reading these values the generator executes the initial encryption to determine I and switches to the state "ready". From this state a new key can be requested, which is generated in the state "generate". A new initialization can only be performed from the state "ready". A new key is provided on the output DATA_OUT and announced by setting the signal RDY.

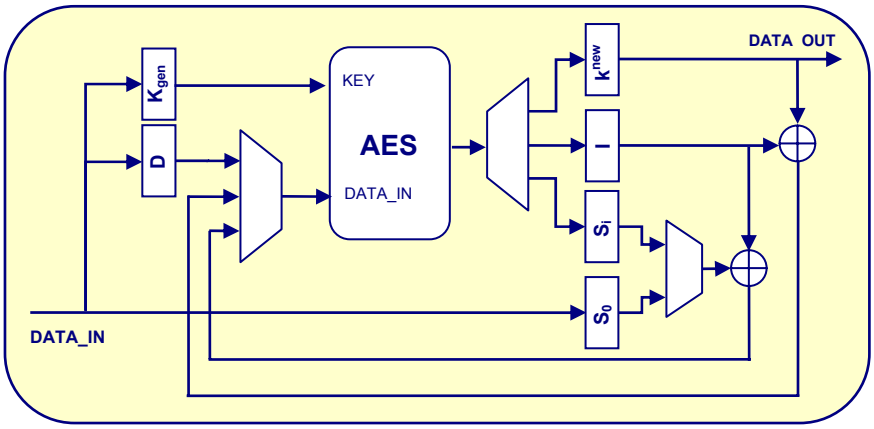


Figure 5.12. Key generator data path

Figure 5.12 represents the data path of the key generator. As mentioned before, the key generator uses one encryption core, which builds the most expensive operation in the generation algorithm. Neglecting the encryption cost to calculate the internal word I, which

is determined only once, it can be seen that a key generation costs approximately two encryptions. This estimation assumes full loading of the generator. Otherwise the determination of the next seed S_i can occur after delivering the new key, which corresponds to generation costs of about one encryption.

5.6.3 Hash Module

Rekeying submessages are hashed using the Meyer hash function [Ma85]. If it is employed alone, secure hashing ensures data integrity. For the purpose of source authentication of rekeying messages the hash value is digitally signed, see Section 5.6.5. Originally, Meyer hash function was defined with DES as a compression function. However, this scheme can be expanded to other block ciphers [Sc96]. Thus, in the scope of this work AES-128 operates as the compression function. A rekeying submessage (RSM) is entered as a data block to be hashed and appears as a key for the internal encryption function. As data, the encryption module takes the hash value of the previous RSM. The encryption result is then xored with the old hash value to deliver the new one. For the first rekeying message RSM a random value H_0 is required. This value has a length of 128 bits and is entered as an initial word. For each further rekeying submessage RSM_i the following relation is applied:

$$H_i = E_{RSM_i}(H_{i-1}) \text{ xor } H_{i-1}$$

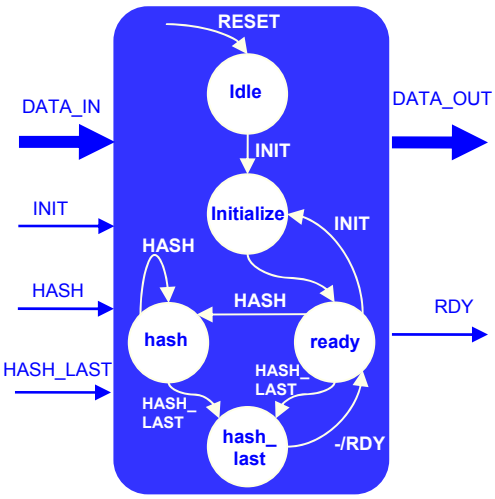


Figure 5.13. Hash module overview

Figure 5.13 provides an overview of the used hash module in the rekeying processors. After the system start-up the initial value H_0 is written to the hash module through the input DATA_IN by setting the control signal INIT. The hash module needs information on the last rekeying submessage to provide the final hash value and to return to the ready state. This is realized by the signals HASH and HASH_LAST. **Figure 5.14** represents the data path of the hash module. As can be seen from this algorithm, hashing a rekeying submessage costs nearly one encryption.

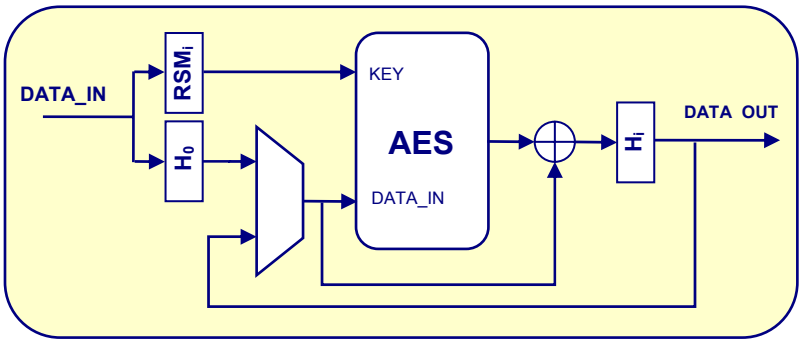


Figure 5.14. Hash module data path

5.6.4 MAC Module

For group authentication the MAC function specified in ISO 9797 with AES-128 instead of DES is used. For this function an authentication key K_{MAC} is required. The first rekeying submessage is encrypted with K_{MAC} directly. Each following rekeying submessage is first xored with the last encryption result and then encrypted with K_{MAC}. The final message authentication code is the encryption result of the last rekeying submessage. The interface and the functionality of the MAC module are largely similar to those of the hash module, therefore, only the data path is presented here for brevity, see **Figure 5.15**.

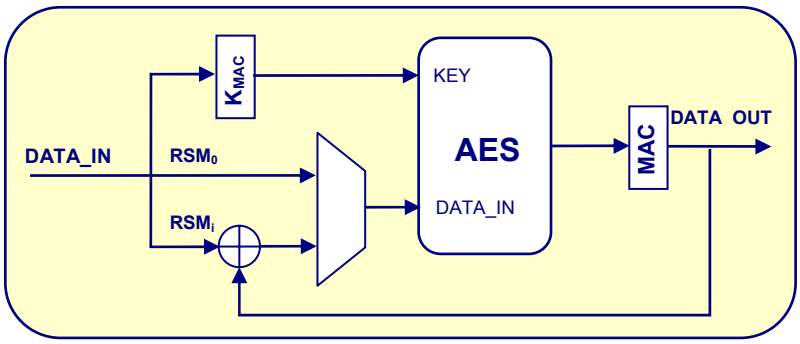


Figure 5.15. MAC module data path

5.6.5 Digital Signature Module

The HiFlexRP uses the Elliptic Curve Digital Signature Algorithm (ECDSA) to sign the hash value of the rekeying message [Ie00]. ECDSA was first proposed by Vanstone in 1992 [Va92]. In 1998 this algorithm was accepted as an ISO standard (14888-3), in 1999 as an ANSI standard (ANSI X9.62), and in 2000 as an IEEE standard (P1363).

To set up the ECDSA the registration and authentication server decides first on a finite field *GF(p)* and on an elliptic curve over this field *EC(GF(p))* of the form:

$$y^2 = x^3 + ax + b$$

where, $4a^3 + 27b^2 \neq 0$

A base point G of the order n is then selected to define the cyclic subgroup. For cryptographic purposes the cyclic subgroup must be large enough. In the best case the cofactor h should equals one where,

$$h = |E|/n$$

In summary, the EC domain parameters are given as a 6-tuple *(p, a, b, G, n, h)*. These parameters must be agreed with all group members.

In a next step the RAS selects a private key d as a random integer in the interval [1, n-1] and calculates the public key Q as a scalar multiplication:

$$Q = dG$$

All the domain parameters and the private key are initially written to the rekeying processor, which performs the ECDSA on the hash value h of the rekeying message. The ECDSA proceeds in the four steps illustrated in **Algorithm 5.1** to estimate the pair *(r,s),* which represents the digital signature.

Algorithm 5.1 ECDSA

Input: h
Output: (r, s)

1. Select a random integer k from the interval [1, n-1]
2. Calculate $P(x_1, y_1) = kG$
3. Calculate $r = x_1 (mod\ n)$, **if** $r = 0$, **go to** 1
4. Calculate $s = k^{-1}(h + dr)(mod\ n)$, **if** $s = 0$, **go to** 1

For its realization the ECDSA is divided into several tasks with different granularities. The most expensive task is the scalar multiplication in Step 2 of the algorithm. This task is executed on a dedicated hardware component, denoted by ECMULT. Another expensive task is the modular inversion in Step 4. Similarly a hardware module, FFINV, is used to implement this operation. Chapter 7 details the design of the ECDSA.

5.7 Input/Output Units

Rekeying processors communicate with the registration and authentication server RAS over the input and output units, see **Figure 5.1** and **Figure 5.4**. In this section the input and output data formats are illustrated without considering the underlying communication protocol between the RP and the RAS. For this purpose, the input and output units are represented as First-In-First-Out storages, denoted as Instruction FIFO and Output FIFO, respectively. The rekeying instructions are written into the Instruction FIFO by the RAS and read from it for execution by the rekeying processor. Rekeying messages are written

into the output FIFO by the RP and read from it by the RAS. The word width of both FIFOs equals 32 bit.

5.7.1 Instruction Set and Input Format

The rekeying processors support a total of ten instructions, where six thereof are present in all architectures. **Table 5.5** summarizes these instructions and the relating parameters and return values.

Table 5.5. Instruction set

Instruction	Utilization	Parameters	Return
InitGen	All RPs	K_{gen}, D, S_0	None
InitHash	HiFlexRP	H_0	None
InitMAC	HiFlexRP	K_{MAC}	None
InitECDSA	HiFlexRP	*p, a, b, G, n, d*	None
InitSysParam	BRP	JBD_{max}, DBD_{max}	None
Join	All RPs	MEMID, k_d	Rekeying Message, k_g
Disjoin	All RPs	MEMID	Rekeying Message, k_g
Resynchronize	All RPs	MEMID	Rekeying Message, k_g
UpdateKg	All RPs	None	Rekeying Message, k_g
DeliverKg	All RPs	None	k_g

Referring to Section 5.6, the first four instructions *InitGen*, *InitHash*, *InitMAC*, and *InitECDSA* are required to set up the security modules by initializing the key generator, the hash module, the MAC module, and the digital signature module, respectively. In normal operation mode one or some of these instructions can be executed separately, e.g. to update the generator key K_{gen} for security purposes.

In Chapter 2 an algorithm was introduced to manage the quality of service and access control in batch rekeying. The Batch Rekeying Processor presented in next chapter implements this algorithm. The instruction *InitSysParam* is defined for this processor to provide the maximal allowable join and disjoin batch delays JBD_{max} and DBD_{max} as system parameters.

The instructions *Join* and *Disjoin* are used to perform rekeying after joining or removing a member, respectively. In the join case the member identity is provided to the rekeying processor as a parameter. As mentioned previously, the generation of identity keys is a task of the registration and authentication server.

LKH is a state-full rekeying algorithm, which means that a member must remain on-line to have the up-to-date keys. Going off-line, the member can lose some of his keys and, thus, need to be resynchronized. The instruction *Resynchronize* is intended for this purpose. The

execution of this operation is simpler than that of the instructions *Join* and *Disjoin*, as no new keys must be generated. In addition the required keys are only encrypted with the identity key of the member to be resynchronized.

The instruction *UpdateKg* is executed to enhance security in the case that a multicast group remains steady for a long period. Only the group key is changed, encrypted with its old value and sent to all the group members.

Executing each of the instructions *Join*, *Disjoin*, *Resynchronize* and *UpdateKg* results in a rekeying message for the group members and in a new group key, which is provided to the data server for data encryption. Besides, the rekeying processors can deliver the group key separately by executing the instruction *DeliverKg*.

The instruction size varies between 32 bits, e.g. for *UpdateKg,* and 800 bits for *InitECDSA.* The operation code is represented by the six most-significance bits of each instruction. The member identity MEMID used in some instructions is represented in the remaining 26 bits of the first instruction word, which allows for group sizes up to 67 million members. As an example, **Figure 5.16** illustrates the structure of the instruction *Join* and how this instruction is written into the Instruction FIFO:

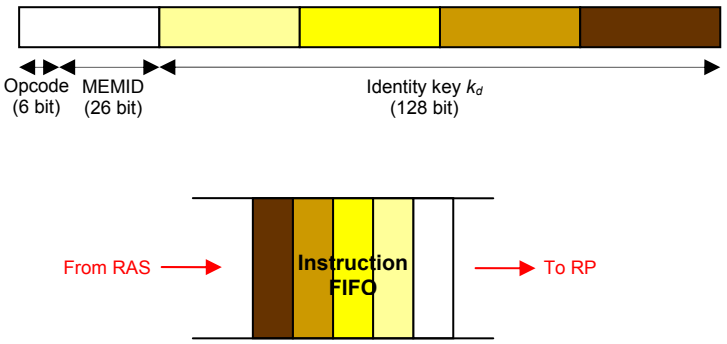


Figure 5.16. Join instruction structure

5.7.2 Rekeying Message Format

The rekeying processors write the rekeying messages into the Output FIFO. **Table 5.6** summarizes the widths of the different data words used in the rekeying processors.

Example 5.5

Referring to ***Example 5.4***, a rekeying processor supporting data source authentication produces a total of 1184 bits to remove the member m_5. This can be detailed as follows.

32 bits (for an identifier which indicates the number of the rekeying submessages included in the rekeying message) + 4*128 bits (four RSMs) + 4*64 bits (four RSMIDs) + 256 bits (digital signature) + 128 bits (for the unencrypted group key) = 1184 bits.

Table 5.6. Widths of the different data words in the RPs

Word	Width (bit)
Tree keys	128
Rekeying submessage	128
Hash value	128
MAC	128
Digital signature	256
Rekeying sub-message identity	64

These data are written into the Output FIFO in the order depicted in **Figure 5.17** from left to right. In addition **Figure 5.17** illustrates the output format for the case of group authentication, for comparison. In this case the digital signature is replaced by a MAC.

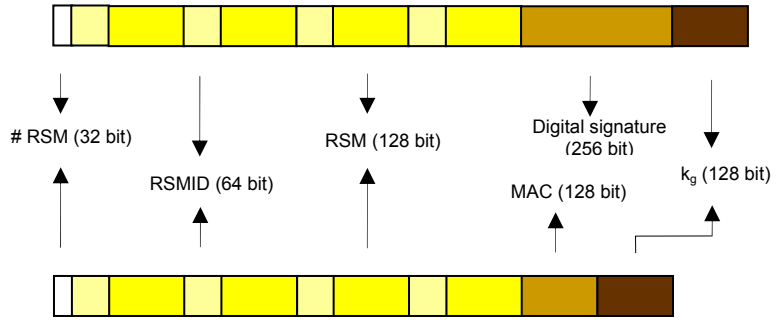


Figure 5.17. Output format

6 Real-Time and Batch Rekeying Processors

6.1 Overview

In this chapter the real-time and the batch rekeying processors are presented. Section 6.1 demonstrates first the general architecture of the RTRP. Then the specific instruction set and the rekeying algorithms of this processor are illustrated. Lastly, implementation results in terms of resource usage and performance features are presented. Section 6.2 provides a similar description of the BRP. This chapter is largely based on the concepts introduced in the Chapter 5 and assumes an understanding of these concepts.

6.2 Real-Time Rekeying Processor (RTRP)

The RTRP performs rekeying requests as soon as they arrive, provided that no other requests are in processing.

6.2.1 Architecture

The RTRP is characterized by the architecture depicted in **Figure 6.1**. This processor receives rekeying requests from the registration and authentication server (RAS), which writes these requests into the *Instruction FIFO*. Fetching, decoding and executing these requests are all tasks managed by the processor controller. As mentioned in the previous chapter, the RTRP fulfills the security requirement for access control in group rekeying. Therefore, the security module of this processor includes two functions for key generation and encryption. As a resource-saving version, the RTRP integrates both these functions into one module, denoted as *GenEnc*, which relies on one AES core. To reduce conflicts caused by this source sharing the GenEnc unit generates keys in advance and saves them into a special FIFO, as long as no rekeying encryptions are required. This FIFO, denoted as *Key FIFO*, allows the storage of 512 keys of the length 128 bits and uses 4 Block RAMs for this purpose. The unit *Key Tree Memory* includes the following two components:

1. The actual *Key Memory* (KM) to save tree keys. This memory is realized using an off-chip SDRAM and an on-chip memory controller.
2. The *Key State Memory* (KSM) used for the static tree management. This memory is realized using on-chip Block RAMs.

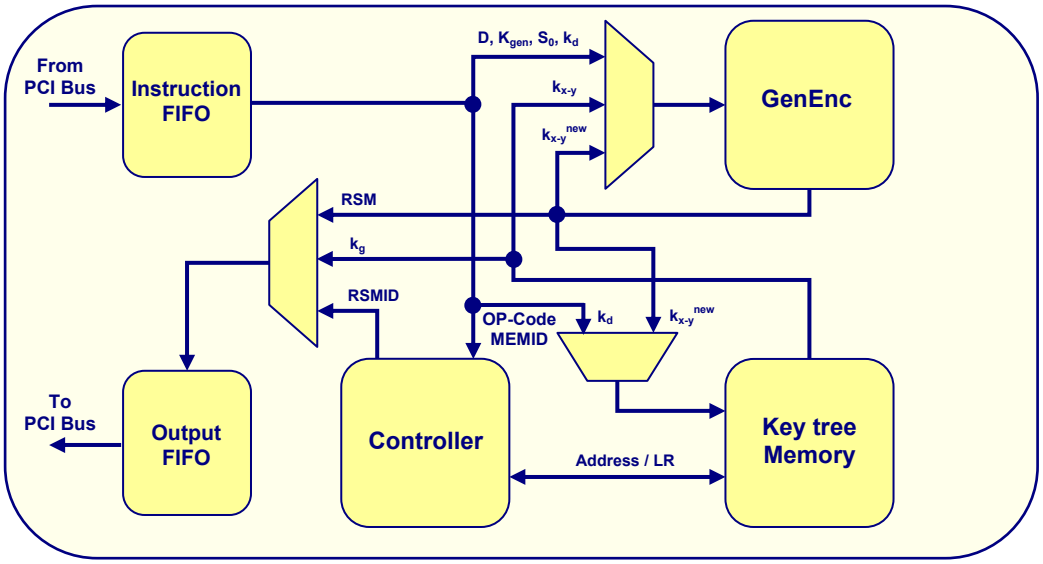


Figure 6.1. RTRP Architecture

The *Controller* is responsible for executing the rekeying algorithms. Because of the complexity of this task, the controller is characterised by a hierarchical architecture of three levels, as illustrated in **Figure 6.2**.

1. On the first level, the *main controller* accepts instructions, decodes them, and activates one of the controllers on the second level.
2. The second level, referred to as the *instruction level*, includes six sub-controllers to support the instruction set of this processor. Note that the RTRP only supports a subset of the instructions, which were explained in the last chapter, see **Table 5.5**. Specifying a sub-controller for each instruction provides an efficient way to modify or add one or more instructions without affecting the basic architecture of the controller. In addition, the modularity on this level enables a pipelining in the instruction execution.
3. The interface with the data path is mainly realized by sub-controllers belonging to the third level, which is called the *hardware level*. Three functional tasks are integrated into this level to process the key addresses, the rekeying submessage identities (RSMID), and the key state words (LR words).

6.2.2 Instruction Set and Rekeying Algorithms

Because of its historical development, the RTRP assumes that each member is supplied with a number of keys, which corresponds to the maximal tree height, regardless of the current group size. Therefore, the concept of suspended keys, introduced in Section 5.5.1 does not apply to this processor. Accordingly, the RTRP relies on a coding mode of LR words, which deviates from that given in **Table 5.1**. In this respect, a LR value of 00 indicates an unused key. A value of 10, 01, or 11 refers to a key, which is used from left, right, or from both sides, respectively.

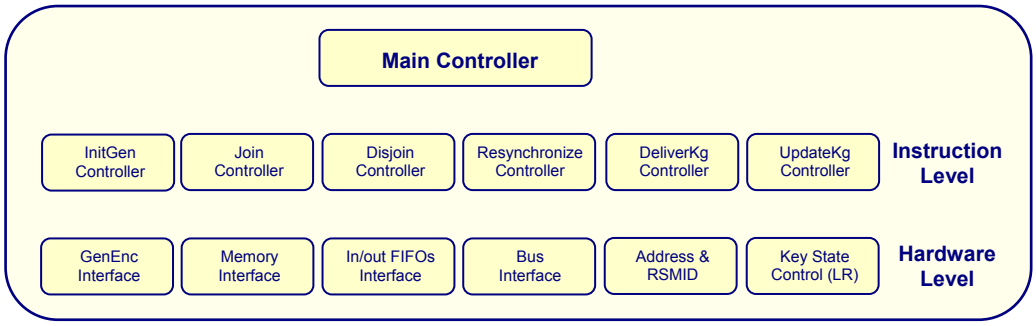


Figure 6.2. RTRP controller hierarchy

The RTRP executes a total of six instructions as seen in **Figure 6.2**. In the following, the execution of the instruction *Join* is presented in some detail. The instructions *Disjoin* and *Resynchronize* are then illustrated briefly. In contrast, other instructions are straightforward and independent of the processor type. For a description of these instructions it is referred to Section 5.7.1 in the last chapter.

6.2.2.1 Join Instruction

The instruction *Join* is associated with two parameters: the member identity MEMID and the identity key k_d of the new member. **Algorithm 6.1** illustrates the processing of this request after its decoding by the main controller. First, the identity key k_d is saved at the memory address MEMID. Afterwards, the join controller starts a path traversing, which is an iterative process to find all keys to be updated. This traversing is based on bit operations on the addresses of these keys as described in the previous chapter. In the RTRP, a new help-key may be encrypted once or twice depending on the corresponding LR value. As can be seen from **Algorithm 6.1**, only those keys, which have an LR value of 11, are encrypted with both sons. Otherwise one encryption is required. To find the keys to be encrypted with, a reverse path traversing is performed. In the case of the RTRP, this traversing always results in the direct sons because of the missing of suspended keys, as mentioned before.

6.2.2.2 Disjoin Instruction

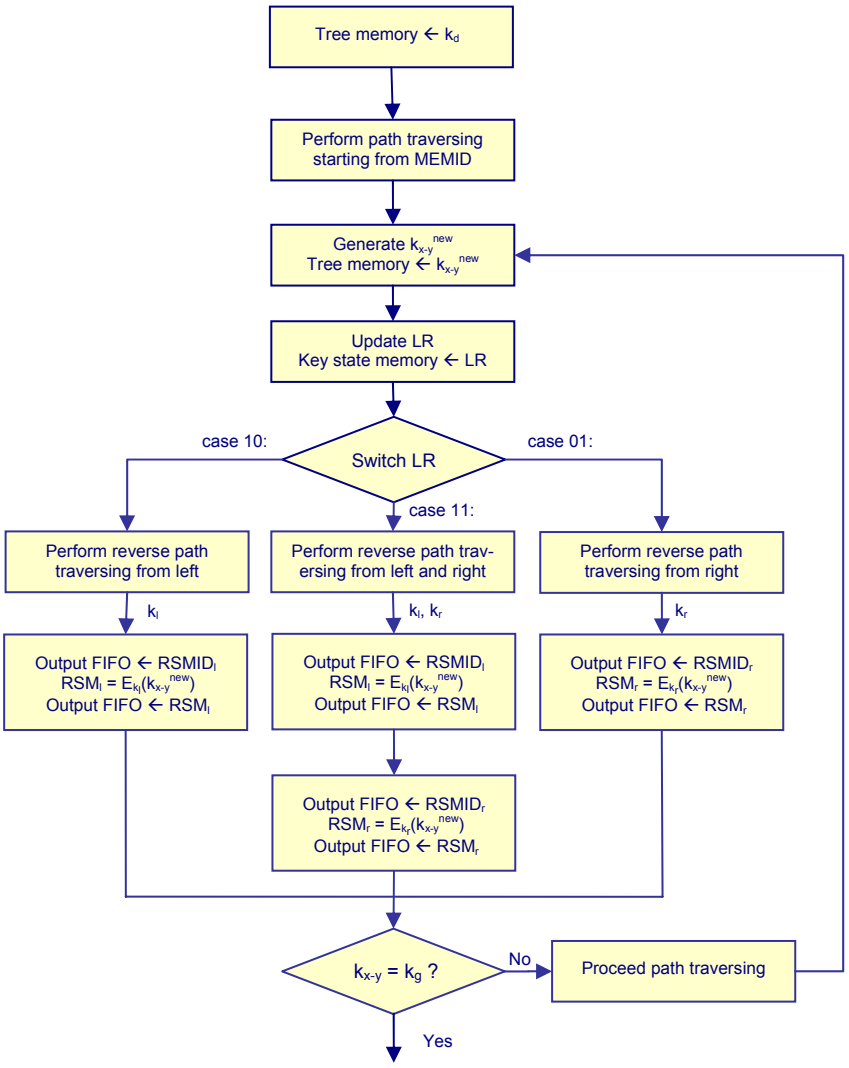
The instruction *Disjoin* includes only a member identity MEMID as a parameter. The execution of this instruction is similar to the join case. However, the LR check and update is performed before key generation, as a disjoin operation can result in an LR value, which equals 00. In this case the corresponding key does not need to be updated or encrypted.

6.2.2.3 Resynchronize Instruction

To resynchronize a member only its MEMID is required. The execution of this instruction is simple, as no key generation is necessary. The keys on the path from the member leaf to the root are only encrypted with the member identity key, which is already known.

Algorithm 6.1 RTRP Member Join

Input: MEMID, k_d
Output: Rekeying Message (RM)



6.2.3 Implementation and Results

A prototype of the RTRP was realized on the PCI card ADM-XRC-II Pro, which was described in Chapter 4. The FPGA includes all the RTRP components except for the key memory, which is realized using the DDR SDRAM as another component of the ADM-XRC-II Pro card.

6.2.3.1 Resource Usage and Maximal group size

Table 6.1 outlines the resource usage for the individual RTRP components on the FPGA. These values, except for SDRAM size, are obtained by a technology-dependent synthesis of the design using Synplify Pro 7.3.3 from Synplicity, Inc. [Sy07].

Table 6.1. Resource usage in RTRP

Component	Area usage			Notes
	CLBs %	# BRAMs	SDRAM	
GenEnc	3	14	--	10 BRAMs for AES + 4 for key FIFO of the generator
Key tree memory	5	--	16 MB	CLBs for SDRAM controller
Key state memory	--	66	--	For LR word storage
Controller	7	--	--	Excluding SDRAM controller
In/Out FIFOs & PCI interface	5	8	--	

The available resources directly affect the supportable group size by the rekeying processor. The external SDRAM of the card ADM-XRC has a capacity of 64 MB which allows the storage of the following number of 128-bit keys:

$$64*1,024*1,024*8/128 = 4,1943,04 \text{ keys}$$

This number corresponds to group size of more than 2 million users according to (5.1). However this estimation is optimistic because it does not consider the memory size needed for the 2-bit LR words. For the storage of this data, the block RAMs of the FPGA are used because of their high performance, which facilitates early reading of LR words and, consequently, a timely decision on the next step. However, the limited number of available BRAMs restricts the group size. This point can be explained as follows. 2VP20 contains a total of 88 BRAMs, where 22 blocks thereof are required for the GenEnc and the FIFOs according to **Table 6.1**. Thus, a rest of 66 BRAMs can be used to save LR words. A BRAM has an effective capacity of 16 Kb. By exploiting all these remaining BRAMs for the Key State Memory the following number of LR words can be saved:

$$66*16Kbit / 2 = 540.672 \text{ LR words}$$

Recall that LR words are only defined for help-keys and that a complete binary key tree contains as many help-keys as the user number minus one. In addition, the user number is always a power of two in complete binary trees. Therefore, the actual maximal group size, which can be supported by the RTRP, is equal to the largest power-of-2, which is smaller than 540.672, i.e.

$$N_{max} = 524.288 = \text{members}.$$

6.2.3.2 RTRP Performance

Table 6.2 summarizes the basic performance figures of the proposed RTRP expressed in terms of clock cycles needed for a particular operation. The value *19* in this table corresponds to the number of tree levels occupied by help-keys for the maximal group size N_{max}. The term *19 Gen.* represents the number of clock cycles needed for the generation of 19 new keys. *38 AES* denotes the number of clock cycles, which are required to perform 38 AES encryptions. The term *X* indicates the number of cycles consumed by different control tasks such as the instruction fetch and decode, and the processing of LR words. As mentioned previously, the processor controller features a modular architecture, which enables a concurrent execution of different tasks such as encryption and memory access operations. Due to this modularity, *X* is small compared to the cycle numbers needed by encryption and key generation and can remain out of consideration in most cases. Differently, *X* in *best-case disjoin* (last row in **Table 6.2**) represents the total time needed in this special case, since neither key generations nor encryptions are needed.

Table 6.2. Performance figures of the RTRP

Performance feature	# Clock cycles
AES encryption	25
Key generation	55
Worst-case join	19 Gen. + 38 AES + *X*
Best-case join	19 AES + *X*
Worst-case disjoin	19 Gen. + 37 AES + *X*
Best-case disjoin	201

The *worst case join/disjoin* occurs when all help keys from the join/disjoin point to the root have to be updated and encrypted twice and the key FIFO of the key generator is empty. The *best case join* occurs when all help-keys from the join point to the root are not used by any current member and the key generator has sufficient keys in its FIFO storage. The *best case disjoin* occurs when all help-keys from the disjoin point to the root will not be used by any remaining group member.

The RTRP was implemented by exploiting the placement and routing tools ISE 6.1 from Xilinx. A clock frequency of 133 MHz was chosen to control the RTRP. This value corresponds to the maximal clock frequency of the external SDRAM.

In order to compare the RTRP performance to other solutions, a software model was built for the rekeying processor with AES encryption, ANSI X9.17 key generation, and LKH rekeying, which is improved by a semi-LR mechanism. The software model was executed on the following two machines:

SW1: AMD Duron 750 MHz, 64 KB (cache), 256 MB (RAM)

SW2: Intel XEON 1.8 GHz, 512 KB (cache), 1GB (RAM).

Table 6.3 depicts a performance comparison between the software and the RTRP solutions.

Table 6.3. RTRP performance vs. software solution

Operation	SW1 (ms)	SW2 (ms)	RTRP (ms)
Worst-case join	1.13	0.6	0.015
Best-case join	0.278	0.144	0.004
Worst-case disjoin	1.115	0.616	0.015
Best-case disjoin	0.836	0.453	0.002

The following example illustrates the advantage of the RTRP solution. Consider the Pay-TV scenario from Chapter 1 and assume that the video provider advertises its service by providing short join times. (Short disjoin times are beneficial for the video provider to keep forward access control, but normally not stated in the advertisement). Join time consists of several slices including the time needed for the calculation of the rekeying submessages. Assume that this time slice equals 1 sec. Under this condition and assuming a worst-case join, the software solution SW1 will be able to join 1s/1.13ms = 885 members under adherence to the offered property in the advertisement. SW2 results in serving 1666 members, and the RTRP features the support of 66,666 members. Obviously, the RTRP can serve considerably more user requests without agreement violation.

6.3 Batch Rekeying Processor (BRP)

The BRP collects a number of rekeying requests within a rekeying interval, marks the keys to be updated, and process these keys together. For detailed description of batch rekeying refer to Chapter 2.

6.3.1 Architecture

The BRP is mainly composed of five units as depicted in **Figure 6.3**. Batch rekeying is a two-step process. In the first step rekeying requests are collected and keys, which need to be updated, are marked. In the next step the marked keys are updated and encrypted to build rekeying submessages. The key marking is a task achieved by the Preprocessing Unit of the BRP. Key update and encryption is performed in the Processing Unit. In [Li01] the concept of subtree is used to denote all marked keys in one rekeying interval. For the BRP this concept is used to refer to the data generated by the Preprocessing Unit to provide information on keys to be processed, the kind of processing, and the order of processing. The first two information are provided by assigning a Marking Flag (MF) and a Left-Right Word (LR) to each help-key, respectively. The processing order is appointed with the aid of the Processing FIFO (PF). The interaction between the Preprocessing, the Subtree and the

Processing Units will be illustrated in Section 6.3.2. In the sequel, some functional units specific to this BRP are described briefly.

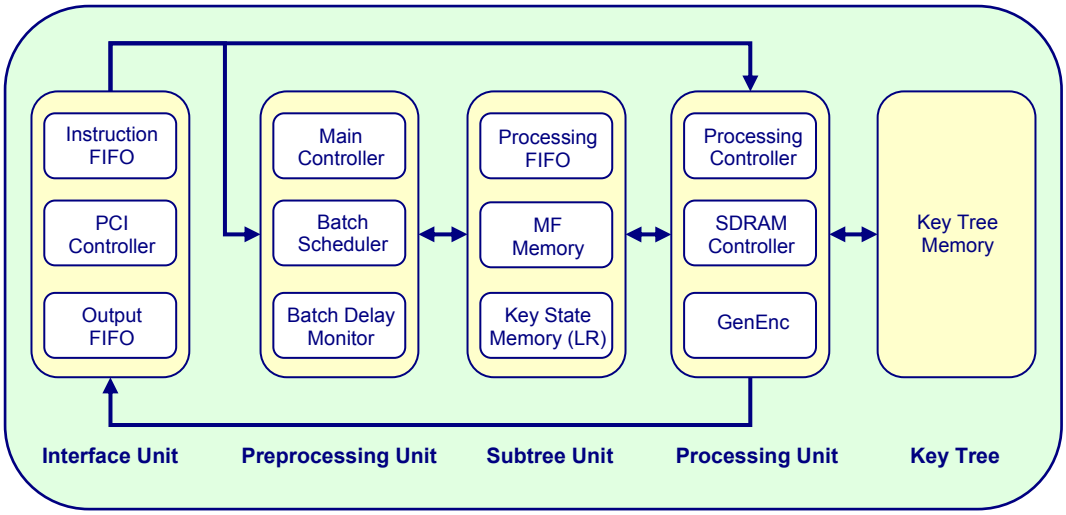


Figure 6.3. BRP Architecture

6.3.1.1 Main Controller

This subunit controls the whole BRP by fetching instructions from the Instruction FIFO, decoding them, and by activating the Batch Scheduler to execute the marking algorithm. In addition, the Main Controller starts the Processing Unit to process an already prepared batch. The Main Controller interacts with the Batch Delay Monitor to control the rekeying interval according the event-driven rekeying algorithm presented in Chapter 2.

6.3.1.2 Batch Scheduler

This module performs the two important tasks of key marking and estimating the batch processing time BPT, see Chapter 2. For the first task the Batch Scheduler receives a rekeying request with some member identity MEMID and marks the help-keys needing to be processed according to the marking algorithm detailed in Section 6.3.2. To estimate BPT the Batch Scheduler gets during marking information on the number of needed key generations and encryptions. Accordingly, it updates the batch processing time BPT for each new request and delivers it to the Batch Delay Monitor.

6.3.1.3 Batch Delay Monitor

This module supports the event-driven rekeying algorithm. For this purpose it measures the actual rekeying interval and compares it with the values T_{max1} and T_{max2}, see inequality (2.6) in Chapter 2. When exceeding one of these values, this unit interrupts the Main Controller to stop marking and start a new rekeying interval. **Figure 6.4** illustrates the architecture of

the Batch Delay Monitor. For this realization the inequality (2.6) is expanded to two inequalities using the formula (2.7) and (2.8) as follows:

$$T < JBD_{max} - BPT_{i+1} + t_{aJ}^{min} \quad (6.1)$$

$$T < DBD_{max} - BPT_{i+1} + t_{aD}^{min} \quad (6.2)$$

Recall that t_{aJ}^{min} and t_{aD}^{min} represent the appearance times of the first join and disjoin requests in a rekeying interval, respectively. The batch processing time BPT_{i+1} is provided by the Batch Scheduler. JBD_{max} and DBD_{max} represent the maximal allowable join and disjoin batch delays, respectively. These system parameters are written to the BRP by executing the instruction *InitSysParam* as depicted in Chapter 5. Rearranging (6.1) and (6.2) results in

$$T - t_{aJ}^{min} < JBD_{max} - BPT_{i+1} \quad (6.3)$$

$$T - t_{aD}^{min} < DBD_{max} - BPT_{i+1} \quad (6.4)$$

Obviously, the left side of inequality (6.3) can be realized by a timer, which is started at the appearance time of the first join request. This timer is denoted as Join Timer in **Figure 6.4**. Similarly a Disjoin Timer is employed to implement the left side of (6.4).

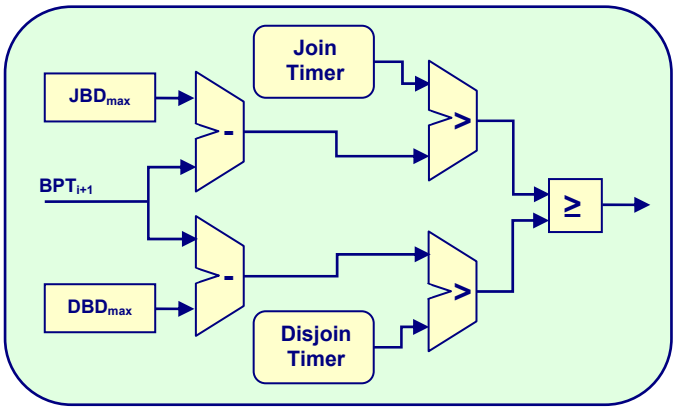


Figure 6.4. Batch Delay Monitor

6.3.1.4 MF Memory

This memory saves a 1-bit marking flag for each help-key. The MF is set by the Preprocessing Unit if the corresponding key needs to be processed. The Processing Unit resets this flag after processing the related key.

6.3.1.5 LR Memory

This unit saves the 2-bit LR words for the help-keys as specified in Section 5.5.2. During marking, an LR word is checked and possibly updated by the Batch Scheduler. Depending

on the LR value of some key, the Processing Unit decides on the kind of processing it must undergo.

6.3.1.6 Processing FIFO (PF)

The Processing FIFO enables an efficient level traversing, which was defined in the previous chapter, see Definition 5.10. During marking the Batch Scheduler pushes the KEYIDs of all level-1 help-keys, which need to be processed, into this FIFO. The Processing Unit pulls these KEYIDs and pushes the KEYIDs of the next-level help-keys needing to be processed in a successive way until all keys have been processed. This mechanism ensures that the processing of a help-key of some level can only take place, after the keys of lower levels have already been handled. Recall that the identity of a key in the rekeying processor corresponds to its physical address, as depicted in Chapter 5, see Definition 5.11.

6.3.1.7 Processing Controller

This module pulls the KEYIDs of keys to be processed from the Processing FIFO, resets their MF-flags, and orders the GenEnc module to build rekeying submessages according to the corresponding LR values.

6.3.1.8 GenEnc

This unit performs both the encryption – to set-up rekeying submessages – and the key generation based on a shared AES-128 core. A dedicated Key FIFO saves pre-generated keys as long as no rekeying encryption is needed.

6.3.2 Instruction Set and Rekeying Algorithms

The BRP supports all instructions known in the RTRP. In addition, the instruction *InitSysParam* is specific to the BRP to initialize the Batch Delay Monitor with the system parameters JBD_{max} and DBD_{max}. In contrast to the RTRP, which performs a rekeying algorithm for each instruction of the type *Join*, *Disjoin*, and *Resynchronize*, the BRP performs two algorithms for a batch of these instructions. These are the *marking algorithm* and the *processing algorithm*. Note that the BRP treats the instruction *Resynchronize* as a *Join* instruction with the difference that no identity key is saved.

In the following, the batch rekeying algorithms will be illustrated assuming a sequential proceeding of the marking and processing tasks, i.e. without pipelining. First, the marking and processing algorithms are presented. An example illustrates then the proceeding of these algorithms. A special problem of applying pipelined batch rekeying is treated in Section 6.3.3.

6.3.2.1 Marking Algorithm

The marking algorithm is performed by the Preprocessing Unit. While starting and ending the marking process is a task of the Main Controller and the Batch Delay Monitor, the actual marking algorithm is executed by the Batch Scheduler. **Algorithm 6.2** depicts the

proceeding of marking for one rekeying instruction such as Join, Disjoin or Resynchronize. The description in **Algorithm 6.2** is highly abstract. More details will be provided in Example 6.1. Note that the marking algorithm only pushes identities of help-keys from the first level into the Processing FIFO. The processing algorithm, later on, pulls these KEYIDs and pushes the KEYIDs of their fathers. By this means it ensured that a key is only encrypted with keys of lower levels, which have already been updated.

Algorithm 6.2 Marking for one instruction

Input: MEMID, instruction type
Output: Updated subtree data

1. Perform path traversing starting with MEMID
2. Push the KEYID of the help-key of level-1 into the Processing FIFO, if this was not yet done by other marking steps in the same interval.
3. Set the marking flags of all help-keys on the path.
4. Update the LR words according to the instruction type. *-- Table 5.1*
5. **return**

6.3.2.2 Processing Algorithm

The processing algorithm is performed by an interaction of the Processing Controller and the GenEnc module in the Processing Unit. This algorithm is started once in each rekeying interval and stopped when all marked keys for this interval have been processed. The processing algorithm is an iterative task, which consists of repeated execution of level traversing. The processing algorithm gets information on the level-1 help-keys, which need to be visited, from the marking algorithm in terms of KEYIDs saved in the Processing FIFO (PF). During processing the corresponding keys, the processing algorithm prepares the level-2 help-keys, which need processing, by writing their KEYIDs into the Processing FIFO. This procedure is repeated for all tree levels. **Algorithm 6.3** illustrates the processing task, in brief.

Algorithm 6.3 Processing

Input: Subtree data

Output: Rekeying message

1. **repeat**
2. Pull a KEYID from the PF.
3. Reset the marking flags of the corresponding key and its father.
4. Push the KEYID of the father into the PF, if this was not yet done.
5. Update/Encrypt the corresponding help-key according to its LR word.
6. **until** PF is empty
7. **return**

Two main remarks can be made to this pseudo code:

1. In this description, the level traversing including the change from level to level is hidden. By means of the Processing FIFO, a part of the processing algorithm, which is responsible for level traversing, is realized using the topological structure of this hardware storage. Consequently, a largely efficient level traversing is provided, as not all keys belonging to some level must be visited, but only those, which were pushed into the Processing FIFO.
2. The if-condition in Step 4 is based on verifying the marking flag of the father. Therefore this flag is reset a priori in Step 3.

Example 6.1: *Marking and Processing Algorithms*

Consider the left key tree in **Figure 6.5** and assume that five rekeying requests appear in the current rekeying interval in the following order: Join m_1, Disjoin m_3, Disjoin m_0, Join m_5, and Disjoin m_2. The processing of these requests results in the right tree in the same figure.

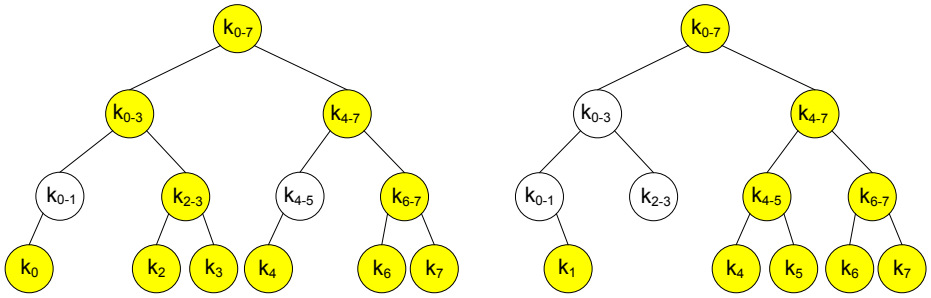


Figure 6.5. Batch rekeying example

Marking

In this example five marking steps are performed, one for each rekeying request. For an illustration of the effect of these preprocessing **Tables 6.4, 6.5** and **6.6** show the development of the marking flags, the LR words, and the PF content in the course of marking, respectively. The following points provide an explanation of some items in these tables.

1. The tables should be considered column-wise from left to right. This order corresponds to the progress of the marking steps.
2. At the start of marking, the marking flags of all help-keys are zero, the Processing FIFO is empty, and the LR words correspond to the left tree in **Figure 6.5**.
3. In **Table 6.5**, LR words, which are updated in some marking step, appear with grey background in the corresponding column.
4. During marking, the LR words of the help-keys $k_{0\text{-}1}$ and $k_{2\text{-}3}$ are updated twice. Nevertheless, the KEYIDs of these keys are pushed into the PF only once. This is realized by checking the marking flag: A KEYID is pushed into the PF only if the corresponding MF is zero.

Table 6.4. MF Memory during marking

Help Key	MF					
	Current Value	**Join m_1**	**Disjoin m_3**	**Disjoin m_0**	**Join m_5**	**Disjoin m_2**
k_{0-1}	*0*	*1*	*1*	*1*	*1*	*1*
k_{2-3}	*0*	*0*	*1*	*1*	*1*	*1*
k_{4-5}	*0*	*0*	*0*	*0*	*1*	*1*
k_{6-7}	*0*	*0*	*0*	*0*	*0*	*0*
k_{0-3}	*0*	*1*	*1*	*1*	*1*	*1*
k_{4-7}	*0*	*0*	*0*	*0*	*1*	*1*
k_{0-7}	*0*	*1*	*1*	*1*	*1*	*1*

Table 6.5. LR Memory during marking

Help Key	LR					
	Current Value	**Join m_1**	**Disjoin m_3**	**Disjoin m_0**	**Join m_5**	**Disjoin m_2**
k_{0-1}	*10*	*11*	*11*	*01*	*01*	*01*
k_{2-3}	*11*	*11*	*10*	*10*	*10*	*00*
k_{4-5}	*10*	*10*	*10*	*10*	*11*	*11*
k_{6-7}	*11*	*11*	*11*	*11*	*11*	*11*
k_{0-3}	*11*	*11*	*11*	*11*	*11*	*10*
k_{4-7}	*11*	*11*	*11*	*11*	*11*	*11*
k_{0-7}	*11*	*11*	*11*	*11*	*11*	*11*

Table 6.6. PF during marking

PF					
Current Entries	**Join m_1**	**Disjoin m_3**	**Disjoin m_0**	**Join m_5**	**Disjoin m_2**
Empty				*KEYID (k_{4-5})*	*KEYID (k_{4-5})*
		KEYID (k_{2-3})	*KEYID (k_{2-3})*	*KEYID (k_{2-3})*	*KEYID (k_{2-3})*
	KEYID (k_{0-1})	*KEYID (k_{0-1})*	*KEYID (k_{0-1})*	*KEYID (k_{0-1})*	*KEYID (k_{0-1})*

5. The final LR value of k_{2-3} equals 00, this means that k_{2-3} will become suspended and does not need to be updated or encrypted. Nevertheless, its KEYID remains in the Processing FIFO to avoid expensive data rearrangement in this FIFO. The processing algorithm, later on, early detects this situation and ignores this key.
6. The KEYID of a help-key represents the address of this help-key in the key memory, the address of the corresponding LR word in the LR Memory, and the address of the corresponding marking flag in the MF Memory.

Processing

The Processing Unit receives the final sub-tree data, which appear in the right-most columns of **Table 6.4**, **6.5**, and **6.6**. The last column of **Table 6.4** unveils six marked keys. Three of these keys – more accurately, their KEYIDs – are already kept in the Processing FIFO. The other three are pushed into the FIFO during processing. Recall that using PF, on the one hand, releases the Processing Unit from looking for marked keys. On the other hand, PF keeps the order of processing, as keys are written into the PF level by level. This enforces the processing of keys of lower levels before those of higher levels.

In this example, the Processing Unit initially pulls the first KEYID entry from the PF which corresponds to $k_{0\text{-}1}$. The LR value is then checked. Since LR($k_{0\text{-}1}$) = 01, i.e. the key is left suspended, this key is not to be updated or encrypted. The KEYID of the father is determined and pushed into the processing FIFO. In addition, both the marking flags of $k_{0\text{-}1}$ and its father are reset. This early resetting of MF($k_{0\text{-}3}$) helps to avoid a second pushing of KEYID($k_{0\text{-}3}$) into the PF during the processing of $k_{2\text{-}3}$. **Tables 6.7** illustrates the development of the PF contents during processing. In this table, KEYIDs belonging to the same level have the same shade of gray.

Table 6.7. PF content during processing

PF after marking	**PF content during the processing of:**					
	$\mathbf{k}_{0\text{-}1}$	$\mathbf{K}_{2\text{-}3}$	$\mathbf{k}_{6\text{-}7}$	$\mathbf{k}_{0\text{-}3}$	$\mathbf{k}_{4\text{-}7}$	$\mathbf{k}_{0\text{-}7}$
KEYID ($k_{4\text{-}5}$)	*KEYID* ($k_{0\text{-}3}$)					Empty
KEYID ($k_{2\text{-}3}$)	*KEYID* ($k_{4\text{-}5}$)	*KEYID* ($k_{0\text{-}3}$)	*KEYID* ($k_{4\text{-}7}$)	*KEYID* ($k_{0\text{-}7}$)		
KEYID ($k_{0\text{-}1}$)	*KEYID* ($k_{2\text{-}3}$)	*KEYID* ($k_{4\text{-}5}$)	*KEYID* ($k_{0\text{-}3}$)	*KEYID* ($k_{4\text{-}7}$)	*KEYID* ($k_{0\text{-}7}$)	

6.3.3 Pipelined Batch Rekeying

Pipelined batch rekeying is characterized by a simultaneous execution of the marking and the processing algorithms, see Chapter 2. Accordingly, a simultaneous access to the Subtree Unit by both the Preprocessing Unit and the Processing Unit must be enabled. From the last section, however, it is obvious that such access may cause a data incoherency problem. **Table 6.8** illustrates the access modes to the data of the Subtree Unit. The first row, for instance, indicates that the MF Memory is accessed for write and read operations by both the Preprocessing Unit and the Processing Unit.

Table 6.8. Access modes on Subtree Unit

Memory	**Access mode**	
	Preprocessing Unit	**Processing Unit**
MF Memory	Read, Write	Read, Write
LR Memory	Read, Write	Read
Processing FIFO	Push	Push, Pop

A first solution of this incoherence problem can be achieved by dividing rekeying intervals into even and odd intervals and using two subtree units:

1. an ***Even Subtree Unit***, which is used by the Preprocessing Unit in even intervals and by the Processing Unit in odd intervals, and
2. an ***Odd Subtree Unit***, which is used by the Preprocessing Unit in odd intervals and by the Processing Unit in even intervals.

While this solution is sufficient to tackle the incoherency of the MF and PF data, it does not solve this problem for the LR words. This can be illustrated as follows. The MF and PF data relate to a set of marked keys in some rekeying interval. Therefore, these data return to their reset values at the end of processing. In contrast, LR words represent the state of all help-keys including the marked ones and are not allowed to be written by the Processing Unit. Assume that the LR word of a help-key $k_{x\text{-}y}$ is initially *00*. In an even rekeying interval this value is updated to the value $LR(k_{x\text{-}y}) = 10$. This new value is written into the Even Subtree Unit. Assume, furthermore, that this key is addressed again in the next interval, which is odd. Preprocessing Unit tries to update the $LR(k_{x\text{-}y})$ based its value in the Odd Subtree Unit which is still *00*, not *01*. In summary, doubling the LR Memory does not solve the incoherency problem. Therefore, the BRP uses a third LR memory which is read and written in both even and odd rekeying intervals to keep up-to-date LR values.

6.3.4 Implementation and Results

A prototype of the BRP was realized on the PCI card ADM-XPL equipped with the FPGA 2VP30, which was described in Chapter 4. The FPGA implements all of the BRP components except for the key memory, which is realized using the DDR SDRAM as another component of the ADM-XPL card.

6.3.4.1 Resource Usage and Maximal group size

Table 6.9 outlines the area usage for the individual BRP components on the FPGA. These values, except for the SDRAM size, are obtained by a technology-dependent synthesis of the design using the program Synplify Pro 7.3.3 from Synplicity, Inc. [Sy07].

The available resources directly affect the supportable group size. For each help-key a total of 8 auxiliary bits are needed for saving the MF and LR data, according to the previous analysis of pipelined batch rekeying in the last section. For the BRP prototype, 64 BRAMs are used to store these data. Accordingly, the following number of help-keys can be managed by the BRP: 64 * 16 Kbit /8 = 131,072. This corresponds to a group size of N_{max} = 131,072 members and a level number of 17. The maximal batch size is limited by the maximal number of KEYIDs, which can be pushed into the Processing FIFO during marking. In the current BRP prototype the PF has a capacity of 512, which allows for a maximal batch size of 1024 requests.

6.3.4.2 BRP Performance

Though the BRP uses the same encryption and key generation modules as the RTRP, a performance comparison between these processors is not proper. This is attributed to the fact that the RTRP and the BRP support largely different operations modes.

Table 6.9. Resource usage in BRP

Component		Area usage		
		CLBs % ~	**# BRAMs**	**SDRAM**
Interface Unit	Instruction FIFO	0	4	--
	PCI Controller	3	0	--
	Output FIFO	0	4	--
Preprocessing Unit	Main Controller	0.26	0	--
	Batch Scheduler	0.69	0	--
	Batch Delay Monitor	0.86	0	--
Subtree Unit	Processing FIFO	0	2	--
	LR & MF Memories	0	64	--
Processing Unit	Processing Controller	1.12	0	--
	SDRAM Controller	4	0	--
	GenEnc	3	14	--
Key Tree	SDRAM	--	--	4 MB

While the RTRP performs rekeying requests immediately, the BRP allows some waiting time under some system constraints. Therefore, using absolute timing figures to compare these processors is not meaningful. Furthermore, a reliable comparing of the BRP performance with a software solution for batch rekeying is impossible because of the difficulty of realizing pipelined batch rekeying using software, among other reasons.

Nevertheless, the BRP can be operated in a semi-immediate rekeying mode, if it is initialized with zero values for both the maximal join and disjoin batch delays JBD_{max} and DBD_{max}. Recall that this initialization can be performed by executing the instruction InitSysParam. In this case the marking break condition will always be fulfilled for the first join or disjoin request. In this operation mode, the BRP was compared with the software solution, which was described in Section 6.2.3.2. **Table 6.10** provides an overview of the measurement results of the worst-case join and worst-case disjoin operations.

Table 6.10. BRP performance vs. SW solution

Operation	SW1 (ms)	SW2 (ms)	BRP (ms)
Worst-case join	1.01 ms	0.537 ms	~ 0.016
Worst-case disjoin	0.998 ms	0.551 ms	~ 0.016

7 High-Flexibility Rekeying Processor

7.1 Overview

This chapter introduces the HW/SW rekeying solution denoted as High-Flexibility Rekeying Processor. Section 7.2 motivates the new solution. The generic architecture of the HiFlexRP is presented in Section 7.3. The execution of the rekeying algorithms on this architecture is then demonstrated in Section 7.4. Section 7.5 illustrates the co-design process followed to partition the task onto hardware and software resources and to schedule the rekeying subtasks. Section 7.6 concludes the chapter with a comparison to related work.

7.2 Introduction

The hardware-only architectures proposed in Chapter 6 provide high rekeying performance in comparison to software solutions. However, they lack flexibility. Because of their hard-wiring, the Real-Time and Batch Rekeying Processors offer low adaptability to various system requirements and group conditions. The new architecture proposed in this chapter combines the hardware performance with the software flexibility to provide highly efficient and, at the same time, high-flexible multicast group rekeying. The flexibility feature of the HiFlexRP relates to several aspects. First, for a rekeying system it is desired to add, remove, extend, or exchange a cryptographic primitive without affecting the overall system functionality. The HiFlexRP supports this feature by means of its modular architecture. Second, a multicast group may be specified by some amount of trustability. For a highly trustable group, rekeying messages only need to be authenticated against non-members. This kind of authentication is denoted as *group authentication* and can be supported by using a simple message authentication code (MAC). In contrast, *data source authentication* must be enabled, if group members do not trust each other and, therefore, need to exactly verify the sender of rekeying messages. A means to achieve this tight authentication mode relies on digital signing of the rekeying message. For the sake of high flexibility the HiFlexRP supports both these authentication modes. Third, in group situations with high dynamics it is recommended to reduce rekeying costs by utilizing batch processing, as long as certain requirements of security and quality of service are fulfilled. The HiFlexRP allows a straightforward switching between real-time and batch rekeying. The functionality of batch rekeying mainly differs from real-time rekeying in the tree management, not in the underlying cryptographic operations. Since the HiFlexRP performs the tree management task as software functions, a switching between these rekeying modes can be supported without effect on the processor architecture.

7.3 HiFlexRP Architecture

A key question in the design of complex systems concerns the relationship between the algorithm to be performed and the architecture as an execution environment. This relationship largely affects the cost, the performance, and the flexibility of the resulting system. The main aspect in this algorithm-architecture relationship relates to the cause-effect direction between algorithm and architecture in the design process. Naturally, an algorithm represents the starting point, i.e. the cause, which leads to the design of an architecture, i.e. the effect, as an executing platform. However, a strict following of this direction enforces a *top-down* design process which can not deliver optimal results without including architectural features such as the execution times of some tasks on some resources. Since these features are not available, because the architecture does not yet exist, they are included in the design process just in an estimated form. Another problem with this design strategy relates to the fact that the resulting architecture is dedicated to a specific algorithm and, thus, obstructs future adapting. The opposite direction in the cause-effect relationship between algorithm and architecture appears in various forms. The most advanced case relies on using prefabricated general-purpose processors and corresponding tools for compiling and linking. In this case, the designer totally abstracts from the underlying architecture and operates on the algorithm level only. This *top-only* design strategy represents the most straightforward method and is largely employed for software development. General-purpose processors, however, only refer a sequential processing of tasks, which is not suitable for computationally intensive algorithms with inherent or required parallelism. To support this kind of algorithms purpose-specific architectures must be developed. The most utilized design strategy in practice is specified by a *bottom-up* nature. After a coarse system partitioning, the design team works on the system hardware and software components separately. The complete system is built up when all components are available. A verification of the applied HW/SW partition, for instance, can first be performed at this late stage in the design process, which is highly inefficient.

For the design of the HiFlexRP an approach is utilized, which is a mix of both the top-down and the bottom-up design strategies. On the bottom level, which is encouraged by the reuse of several available components such as the AES and the key generator units, a generic architecture is developed. This architecture supports all computation-intensive operations in hardware. On the top level, the complete functionality is developed in software which can run on the embedded PowerPC processor. In contrast to known top-down approaches, the final HW/SW partition and scheduling are determined depending on actual timing features, which are provided by means of on-chip measurements of the execution times of different tasks on different resources. The partition process will be detailed in Section 7.5.

Figure 7.1 depicts the generic architecture of the HiFlexRP, which differs from previous architectures in Chapter 6 in the following points:

1. The control-intensive tasks such as tree management are assigned to the PowerPC processor. Note that the data and instruction memories can be connected to the processor either using the On-Chip Memory bus (DOCM and IOCM), or using the Processor Local Bus (PLB), as depicted in Chapter 4. The effect of bus selection will be investigated in Section 7.5.

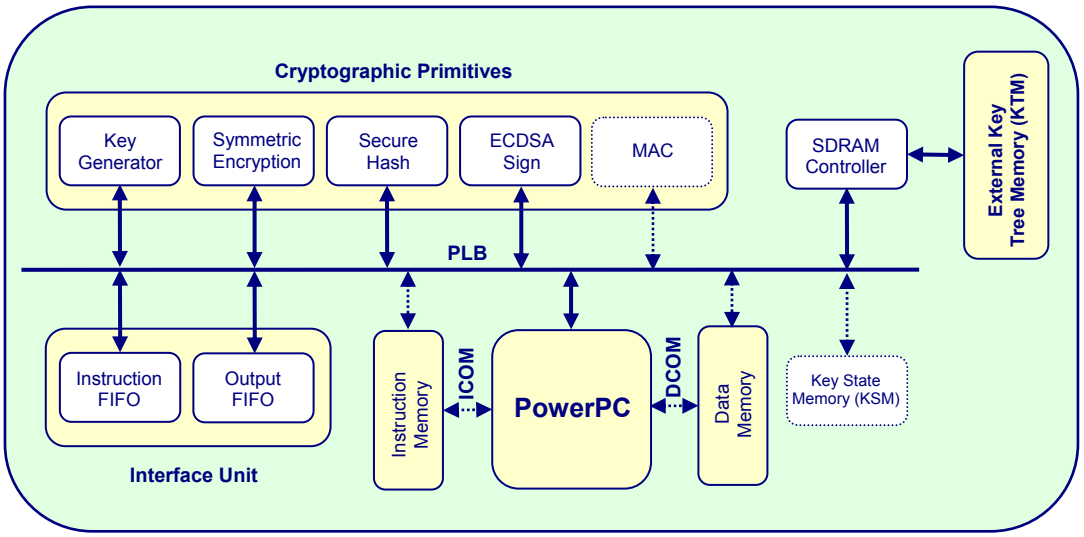


Figure 7.1. General architecture of HiFlexRP

2. The HiFlexRP supports both static and dynamic management of the key tree. The static tree management depends on using a Key State Memory (KSM), as was illustrated in Chapter 5. Dynamic tree management utilizes methods, which are similar to those known in the field of software data structures. The static tree management was investigated thoroughly in the last two chapters. Therefore, the description of the HiFlexRP in this chapter will be limited to the dynamic tree management. Nevertheless, a comparison between these two modes regarding resource usage and rekeying performance will be provided in Section 7.4 and Section 7.5, respectively.
3. In addition to the forward and backward access control, the new architecture supports both group authentication using the message authentication code module (MAC), and data source authentication using the modules Secure Hash and ECDSA Sign. All considerations in this chapter, however, relate to the complex form of authentication, i.e. to the data source authentication. The simple group authentication is omitted, for brevity.

Based on this architecture, the HiFlexRP functionality can be described using a 4-layer model as depicted in **Figure 7.2**. This presentation is generic in the sense that some elements of the different layers may be absent depending on the design alternatives. For example, in the case of dynamic tree management the Key State Memory and, thus, the corresponding KSM driver are omitted. On the top layer, rekeying instructions are fetched from the Instruction FIFO. After their decoding, the necessary functions for key tree management and secure data handling are called. The rekeying results are then prepared and written to the Output FIFO. The intensive operations for tree management and security are embedded to a dedicated layer to enable modularity. The driver layer supports the HW/SW interface and includes functions to initialize the different hardware modules, to

load data, to initiate computation, and to fetch processing results. The various hardware modules settle on the lowest layer.

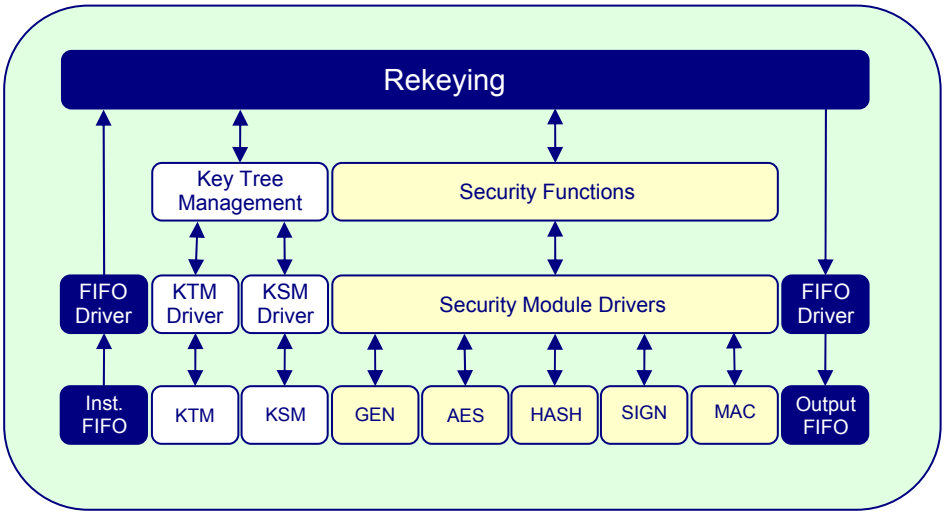


Figure 7.2. Functional layers in HiFlexRP

7.4 Rekeying Algorithms

The concept of rekeying algorithm, in this chapter, refers to the processing steps performed by the PowerPC processor to execute one of the instructions Join, Disjoin and Resynchronize, which were outlined in Chapter 5, see **Table 5.5**. The execution of the instruction Join is the most expensive operation and will be employed, therefore, to illustrate the HiFlexRP functionality, representatively. Before detailing the join algorithm, the next section depicts the data structure used for dynamic tree management.

7.4.1 Tree Data Structure

Key trees differ from search trees in several points, which affect their management mode. The most significant difference relates to the fact that key trees are unordered, i.e. there is no relation between data saved in the different nodes of the tree. Recall that data stored in LKH trees represent keys, which are generated randomly. Therefore, the access to some node or leaf in the tree can not rely on the key data. This matter raises some special management issues of the key tree during joining, disjoining, or resynchronizing a member:

1. Finding a position to add a leaf in the join case is independent of the data to be stored at this leaf, i.e. the identity key of the member. Therefore, additional information must be provided to indicate the addition position. In the case of the HiFlexRP, each node saves information on both the shallowest left leaf and the shallowest right leaf in the subtree with the corresponding node as a root. By Using this information, a rekeying algorithm decides on the join point of a new leaf by looking for the shallowest leaf in the tree

starting at the root. In addition, the HiFlexRP employs a tree management strategy, which expands the tree from the root side to join a member in the case of full trees. By this means, the new group key must be encrypted only with the old group key and with the identity key of the new member. Not only the relating join request profits from this strategy, which is denoted as *bottom-up tree growing* in this work, but also following join and disjoin requests. An in-depth investigation of the advantage of this tree management strategy was provided in a student thesis under the author's supervision [Ab05]. This strategy, however, demands a means to check the tree for fullness. For this purpose, each node contains additional information on both the deepest left leaf and deepest right leaf in the subtree with the corresponding node as a root. Accordingly, each node of the key tree is specified as depicted in **Figure 7.3**. Note that tree leaves are special nodes with a zero value for DSLL, DDLL, DSRL and DDRL. In addition the pointers LS and RS are initialized with NULL all the time. In contrast to static tree management, a tree leaf is not associated with a member identity MEMID. Otherwise, complex search operations must be performed to find tree leaves in the case of member disjoin or resynchronization. To avoid this, another solution is proposed as illustrated in the next point.

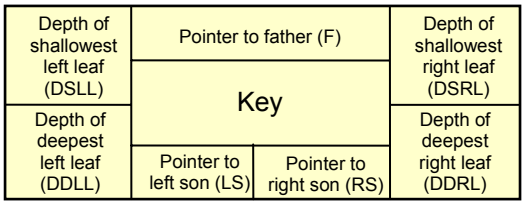


Figure 7.3. Key node structure in dynamic tree management

2. In the case of disjoin or resynchronize requests, the HiFlexRP receives the related member identity MEMID. To update keys, the corresponding member leaf must be found first. For this purpose, an array of pointers to tree leaves is deployed, which is indexed by the member identity MEMID. This array is denoted as Member Leaf Map Array (MLMA) in this work. Thus, a join request causes, besides setting up a new leaf, the creation of a pointer to this leaf, which is inserted into the MLMA at the array position indexed by the MEMID of the new member. By this means, an efficient access to the leaf is possible, if the related member has to be disjoined or resynchronized in future. To avoid data rearranging, the length of MLMA is assumed to be equal to the maximal group size. **Figure 7.4** illustrates a MLMA for a group of 4 members.

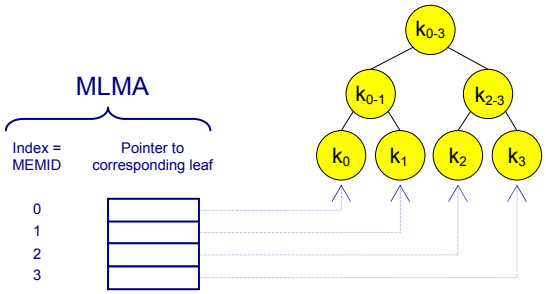


Figure 7.4. Member Leaf Map Array (MLMA)

Obviously, dynamic tree management demands larger memory space to store auxiliary data in comparison to the static tree management, which uses only two bits (LR word) to describe the state of a help-key, whereas for identity keys no auxiliary data are required. This sparing characteristic of static tree management is attributed to using the physical structure of the key memory to keep information on the logical tree topology, as was illustrated in Chapter 5. To support a group of 131,072 members, a total of 32 KB is required to save the needed LR words. In contrast, a memory size of 4.5 MB is demanded to support the same member number using dynamic tree management. This memory size can be estimated as follows. A binary LKH tree with 131,072 members includes 131,072 identity keys and 131,071 help-keys. According to the previous analysis, each key is associated with three pointers and four auxiliary data. Furthermore, a MLMA of the size 131,072 must be created to store pointers to all identity keys. Based on the PowerPC architecture, all the pointers have a length of 32 bits. For the tree topology information such as the DSLL, a character type of the length 8 bits is used. Thus, the following memory size is estimated for the tree auxiliary data:

*131,072 * 32 (for the MLMA) + 262,143 [3*32 (for pointers to father, left and right sons) + 4*8 (for the DSLL, DDLL, DSRL, and DDRL)]= 4.5 MB.*

However, the low memory usage is not the only advantage of static tree management. This mode enables, furthermore, higher rekeying performance compared to dynamic tree management, as will be depicted by the measurement results in Section 7.5.

7.4.2 Join Algorithm

Algorithm 7.1 represents the execution of a join request by the HiFlexRP. This process expects the member identity MEMID and the identity key k_d of the member to be joined, and results in a rekeying message, which is written to the Output FIFO. To tackle the complexity of this algorithm, it is divided into four coarse steps, which will be explained in the next four sections. Consider, first, the following points regarding the pseudo-code of **Algorithm 7.1**:

1. The description of this algorithm assumes normal operation, i.e. the initialization of data structures, e.g. the key tree, and other software and hardware components is proposed to be already done.
2. The algorithm abstracts from the fetch and decode phase of the rekeying instruction, which is performed by the PowerPC processor.
3. The algorithm abstracts from the tree management mode. However, the underlying algorithms perform dynamic tree management. Note that Step 4 is completely independent of the tree management mode.
4. Rekeying messages are digitally signed for the purpose of data source authentication. However, no specific cryptographic primitives are predefined at this stage.
5. This pseudo-code indicates a sequential execution of the different steps of the join algorithm. Performance improvements by means of parallelizing different subtasks will be treated in Section 7.5.

Algorithm 7.1 Join a Member

Input: MEMID, k_d
Output: Rekeying message RM

1.	Add new leaf (MEMID, k_d)	-- *Algorithm 7.1.A*
2.	Update tree topology data (MEMID)	-- *Algorithm 7.1.B*
3.	Update keys on the join path (MEMID) → Hash value *h*	-- *Algorithm 7.1.C*
4.	Sign hash value (*h*)	-- *Algorithm 7.1.D*

7.4.2.1 Add new leaf (MEMID, k_d)

Algorithm 7.1.A represents the processing of Step 1 of **Algorithm 7.1**, which is responsible for creating a new leaf with the corresponding identity key k_d and appending it to the right position in the tree. To allow an efficient access in future, a pointer to this leaf is created and inserted into the member leaf map array (MLMA) at the index MEMID, as illustrated in the last section. This point is presented in the 4th step of **Algorithm 7.1.A** (counting started from the top of the flowchart).

To find an insertion point for the new leaf, the HiFlexRP first checks whether the tree is full, or not. This is realized by verifying the equality of the four parameters DSLL, DDLL, DSRL, and DDRL of the root. A full tree is specified by equal values of all these parameters. In such a case, the leaf insertion is straightforward, as depicted on the right side of the first conditional branching in the flowchart. This case corresponds to the bottom-up tree growing followed by the HiFlexRP, as mentioned in Section 7.4.1. In contrast, for incomplete trees, a tree traversing starting from the root must be performed to find out the shallowest leaf. In both cases, adding a new leaf demands the creation of a new node, which is inserted as a new root in the case of full trees, or at the position of the shallowest leaf, otherwise.

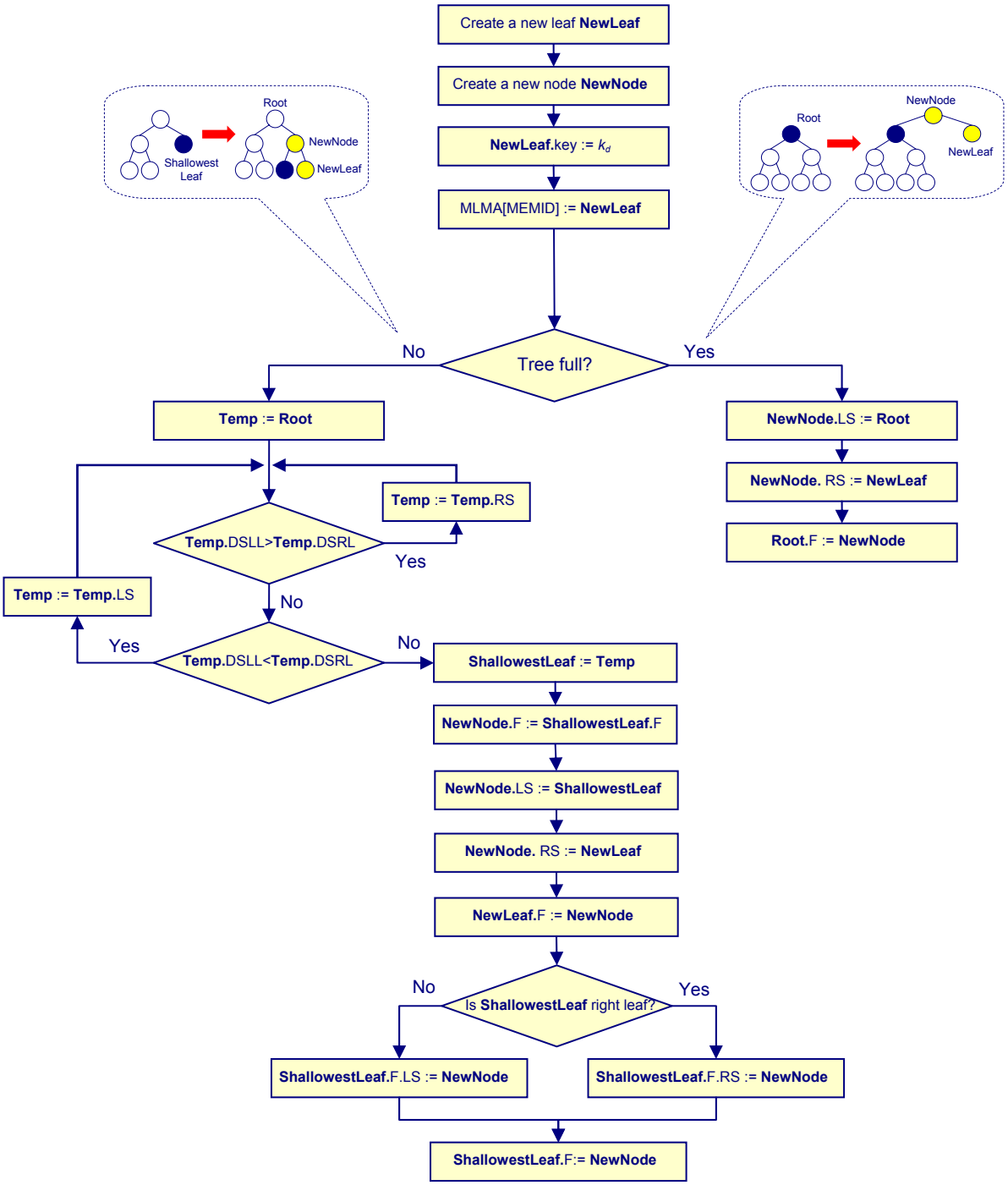
As can be seen from the flowchart, adding a new leaf to the tree is quite complex if compared with the case of static tree management. In this case, this task is trivial, as all tree leaves have predefined positions in the key memory and a member identity MEMID corresponds to the physical address of the corresponding leaf. Thus, adding a new leaf is limited to a memory access to write the identity key k_d at the memory address MEMID.

7.4.2.2 Update tree topology data (MEMID)

After adding a new leaf, the auxiliary data of all nodes on the join path, i.e. DDLL, DDRL, DSLL and DSRL, must be updated, see **Figure 7.3**. This procedure is presented in **Algorithm 7.1.B** for the case of incomplete trees, for clarity. In the case of full trees, only the auxiliary data of the new root must be determined. Note that for one node, either the left data (DDLL and DSLL) or the right data (DDRL and DSRL) must be updated, depending on whether the son of this node on the join path is a left or a right son, respectively. This is realized by using the LeftSonFlag, which is initialized with 0, because a new leaf is always added as a right son. Updating this flag is not detailed in **Algorithm 7.1.B**, for simplicity. This algorithm assumes, furthermore, that the auxiliary data of a leaf are always equal to zero. Example 7.1 provides a brief illustration of this algorithm.

Algorithm 7.1.A Add new leaf

Input: MEMID, k_d



Algorithm 7.1.B Update tree topology data

```
Input: MEMID
1.   node := MLMA[MEMID].F                          -- help-key of level 1
2.   LeftSonFlag := 0;
3.   while (node != NULL) do                        -- traverse to root
4.      switch LeftSonFlag
5.      case 0:
6.         if node.RS.DDLL > node.RS.DDRL then
7.                  node.DDRL := node.RS.DDLL + 1
8.         else     node.DDRL := node.RS.DDRL + 1
9.         if node.RS.DSLL > node.RS.DSRL then
10.                 node.DSRL := node.RS.DSRL + 1
11.        else     node.DSRL := node.RS.DSLL + 1
12.     case 1:
13.        if node.LS.DDLL > node.LS.DDRL then
14.                 node.DDLL := node.LS.DDLL + 1
15.        else     node.DDLL := node.LS.DDRL + 1
16.        if node.LS.DSLL > node.LS.DSRL then
17.                 node.DSLL := node.LS.DSRL + 1
18.        else     node.DSLL := node.LS.DSLL + 1
19.     node := node.F
20.     Update LeftSonFlag
21.  end while
```

Example 7.1

This example illustrates, how the auxiliary data of the node Root are updated after joining a member in the left tree depicted in **Algorithm 7.1.A**. Assuming that NewNode has already updated its data, as depicted in the first row of **Table 7.1**. As NewNode is a right son of Root, case 0 (Step 5 in **Algorithm 7.1.B**) will be selected, i.e. only the right auxiliary data of Root will be updated. Both if-conditions in steps 6 and 9 are false, thus, the statements in steps 8 and 11 are executed, which results in a value of 2 for both DDRL and DSRL of Root, as depicted in **Table 7.1**.

Table 7.1. Updating root auxiliary data for example 7.1

	DDLL	DSLL	DDRL	DSRL
NewNode	1	1	1	1
Root before join	2	2	1	1
Root after join	2	2	2	2

Again, this procedure in dynamic tree management is more complex than the case of static tree management, where only 2 LR bits of each help-key on the join path must be updated.

7.4.2.3 Update keys on the join path (MEMID)

This step of **Algorithm 7.1** deals with the actual rekeying including key generation, encryption, hashing, and writing the rekeying message into the output FIFO. **Algorithm 7.1.C** illustrates the execution of this step by the HiFlexRP. The function *Encrypt* (k_a, k_b) indicates the encryption of the key k_a with the key k_b. The function *Push* takes the responsibility of writing rekeying submessages into the Output FIFO. For simplicity, some details relating to the output message format are neglected.

Algorithm 7.1.C Update keys on the join path

Input: MEMID
Output: Hash value *h* -- *hash value of last rekeying submessage*

1. node := MLMA[MEMID].F -- *help-key of level 1*
2. **while** (node != NULL) **do** -- *traverse to root*
3. Generate new key $k_{x\text{-}y}^{new}$
4. node.key := $k_{x\text{-}y}^{new}$
5. RSM_l := Encrypt($k_{x\text{-}y}^{new}$, node.LS.key)
6. Push RSM_l
7. Hash RSM_l
8. RSM_r := Encrypt($k_{x\text{-}y}^{new}$, node.RS.key)
9. Push RSM_r
10. Hash RSM_r
11. node := node.F
12. **end while**
13. **return** *h*

7.4.2.4 Sign hash value (h)

Signing the hash value represents the most time-consuming step in the join algorithm. The HiFlexRP uses the Elliptic Curve Digital Signature Algorithm (ECDSA), which was described in Chapter 5. For the further analysis in this chapter, the ECDSA approach is given again in **Algorithm 7.1.D**.

Algorithm 7.1.D Sing hash value

Input: *h*
Output: Digital signature (*r*, *s*)

1. Select a random integer *k* from the interval [1, *n*-1]
2. Calculate $P(x_1, y_1) = kG$
3. Calculate $r = x_1 (mod\ n)$, **if** $r = 0$, **go to** 1
4. Calculate $s = k^{-1}(h + dr)(mod\ n)$, **if** $s = 0$, **go to** 1
5. Push (*r*, *s*)

7.5 Design Approach and Performance Features

As introduced in Section 7.2, for the design of the HiFlexRP an approach is employed, which relies on a mix of the bottom-up and the top-down design strategies. On the bottom level, a generic architecture is designed which allows a comprehensive evaluation of different implementation alternatives. On the top level, a software-only solution is produced, which runs on the generic architecture. Depending on a real-time measurement of the execution times of critical software parts, different tasks are migrated to the hardware resources to optimize rekeying performance. After this task migration, a new measurement of the execution times is performed to provide information on further task scheduling and binding. Consequently, the decision on the final design alternative is based on actual timing features of the underlying architecture and not on estimated values, such as in the case of system-level design methods. Nevertheless, as will be seen at the end of this section, the system-level design tool hCDM [Kl06] was used to verify the performed selection of design alternatives, which are illustrated in the next sections. A conformation was proved to large extent.

The design approach in this section will be explained using the example of member join task as given in **Algorithm 7.1**. According to its granularity and data dependencies, this task can be divided into two main subtasks denoted as *Update* and *Sign*:

1. ***Update:*** This subtask gets a rekeying request RREQ, builds the rekeying submessages RSMs, and determines the hash value *h* of these submessages. Thus, the first three steps of **Algorithm 7.1** are combined to the Update subtask.
2. ***Sign:*** This subtask represents Step 4, which signs the hash value *h* resulted from the subtask Update to provide data source authentication.

Figure 7.5 represents the join operation as a task graph. Recall that a RREQ originates from the Instruction FIFO. RSMs and the digital signature (*r*, *s*) are written into the Output FIFO. The following three sections are organized as follows. First, the test environment used to estimate the timing features of the HiFlexRP is presented. Second, several implementation alternatives of the subtask Update are investigated. Third, the subtask Sign is partitioned into fine-granular subtasks to optimize rekeying performance.

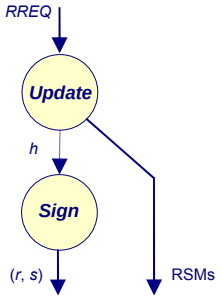


Figure 7.5. Rekeying DFG

7.5.1 HiFlexRP Test Environment

Figure 7.6 illustrates the test environment, which was used to estimate execution time values of different subtasks, as a basis for design decisions. For simplicity, this figure only depicts the components, which directly relate to the test task. The timing values are measured by a special timer, which is connected to the PLB bus. This timer is started and stopped by the test software running on the PowerPC processor. With a word width of 32 bits and a clock frequency of 100 MHz, as a PLB peripheral, time periods up to 4.3 sec can be measured by this component.

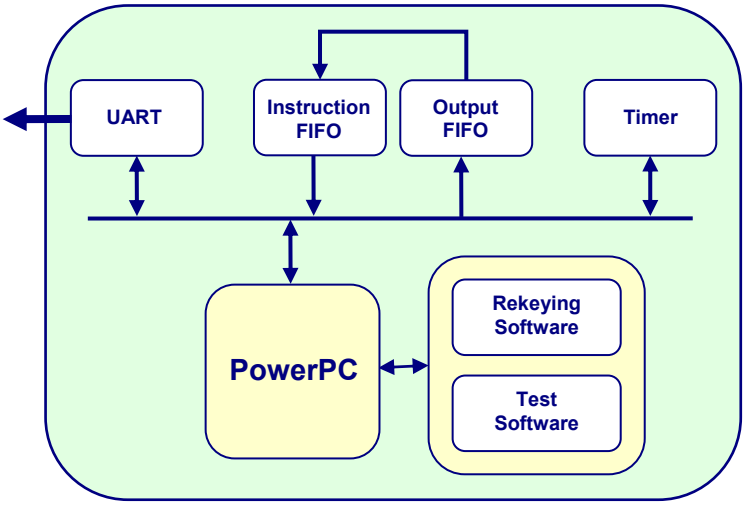


Figure 7.6. Test environment

For the purpose of visualization, the measured timing values are sent to the host over a UART interface. The Instruction and Output FIFOs are included to the test environment to consider the time intervals elapsed by fetching and decoding of new requests and by writing of rekeying data, respectively. As can be seen in **Figure 7.6**, these FIFOs are bypassed to accelerate the measurement: Instead of loading from the host, the rekeying requests are written from the test software into the Output FIFO and from there they are directed into the Instruction FIFO. The write of rekeying instructions is performed before starting the time measurement and, therefore, does not affect timing results. Rekeying submessages are also directed into Instruction FIFO. By this means, the test software can read these data and perform on-chip functional verification, depending on defined test vectors. This verification is performed after stopping the timer to keep reliable timing results.

Algorithm 7.2 illustrates the measurement approach followed to estimate the time elapsed by some rekeying request. First the tree is initialized with a desired member number n_0. Afterwards, a rekeying request according to the format defined in Chapter 5 is written to the Output FIFO, where this request is bypassed to the Instruction FIFO. As following

steps, the timer is triggered, the rekeying is executed, and the timer is stopped immediately at the end of rekeying execution. Next, the rekeying message is fetched from the Instruction FIFO and is verified by comparing with predefined test vectors. In the case of functional correctness, the time values are read from the timer, scaled to the desired time units and sent to display on the host over the UART interface. Otherwise, an error is notified by sending a corresponding message.

Algorithm 7.2. Rekeying performance measurement

Output: Timing values

1. Initialize tree with n_0 members
2. Initialize the Instruction FIFO with the rekeying request.
3. Start timer
4. Initiate rekeying *-- e.g.* ***Algorithm 7.1*** *in join case*
5. Stop timer
6. Verify rekeying result
7. **if** rekeying result is correct **then**
8. Estimate performance
9. **return** timing values
10. **else**
11. **return** "Rekeying error"

Note that the test environment enables also the measurement of subtask execution times such as the subtasks Update, Sign, or even partial subtasks thereof. In such cases, Step 4 of **Algorithm 7.2** is split and the timer starts and stops are inserted at the appropriate points of the data flow graph.

7.5.2 Update Subtask Design Alternatives

As mentioned before, the subtask Update includes all the processing steps starting with the instruction decoding and ending at determining the hash value of the last rekeying submessage. This subtask is both control-intensive, regarding the tree management, and computation-intensive, regarding the cryptographic key generation, encryption, and secure hashing. The realization of the subtask Update is affected by numerous factors, e.g.:

1. The tree management mode: static or dynamic.
2. The realization of cryptographic primitives: software, hardware, or hardware with resource sharing.
3. Memory and caching, BRAM or SDRAM, with or without caching.
4. Bus structure, PLB, OPB or OCM.

Finding the optimal design alternative is a hard problem because of the high interaction among these factors. For a comprehensive analysis, 108 design alternatives for the subtask Update were realized and evaluated. For each solution, the execution times of the worst-case disjoin and worst-case join operations were measured as a function of the group size,

which ranges from 0 to 131,072. The estimation of execution times is based on **Algorithm 7.2**. For an efficient measurement, the test software (see **Figure 7.6**) performs a routine, which initializes the group with 131,072 members stepwise, and interrupts this initialization at the points corresponding to a worst-case join or worst-case disjoin. At these points, the join or disjoin operation is executed and measured, before the initialization is continued. Lastly, the test software prepares the measured timing data for a presentation using Mathematica [Wo07] and sends these data to the host over the UART interface. In the following some design alternatives are discussed representatively, which depicts some interesting issues in the design of the HiFlexRP:

7.5.2.1 Bus selection and caching

As mentioned in Chapter 4, the embedded processor PPC405 features a Harvard architecture with dedicated data and instruction memory interfaces. System memories can be connected either to the Processor Local Bus (PLB) or to the On-Chip Memory bus (OCM). The decision on the appropriate memory bus must take the overall system and the running application into consideration. While the PLB offers 64-bit data busses, compared to 32-bit in the case of OCM, the last is dedicated for memories, i.e. it is not shared by other system components like the PLB. Another decision criterion relates to the cacheability of these memories. Because of dedicating the OCM bus to storage resources, memories connected to this bus do not support caching, in contrast to PLB memories. Therefore, several design alternatives with different memory and bus configurations are evaluated for the HiFlexRP design. Particularly, the worst-case join costs were measured for three systems with OCM memories, with cached PLB memories, and with non-cached PLB memories. All these systems were tested in two cases:

1. Hardware-only realizations of the cryptographic primitives for key generation, encryption, and secure hashing, see **Figure 7.7**.
2. Software-only realizations of these operations, see **Figure 7.8**.

All these design alternatives use static tree management. Other implementations with dynamic tree management, which delivers comparable results, are not reported here for brevity.

Recall that the measured execution times here only relate to the subtask Update, i.e. the time elapsed for determining the digital signature of the hash value is not included. The diagrams presented in **Figure 7.7** and **Figure 7.8** conform the logarithmic relation of rekeying costs to the group size in the LKH algorithm. Note that the *x*-axis in these diagrams has a half-logarithmic scale.

Obviously, the best performance of the subtask Update can be obtained by using PLB instruction and data memories with caching, regardless of the implementation of the security modules and the tree management mode. In contrast, OCM memories are superior to PLB memories, if the last are used without caching. Therefore, the next experiments were performed on systems with cached PLB memories.

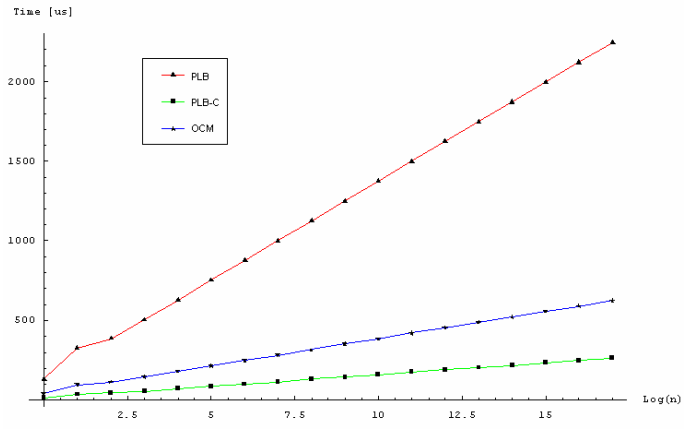


Figure 7.7. Worst-case join cost with hardware security modules

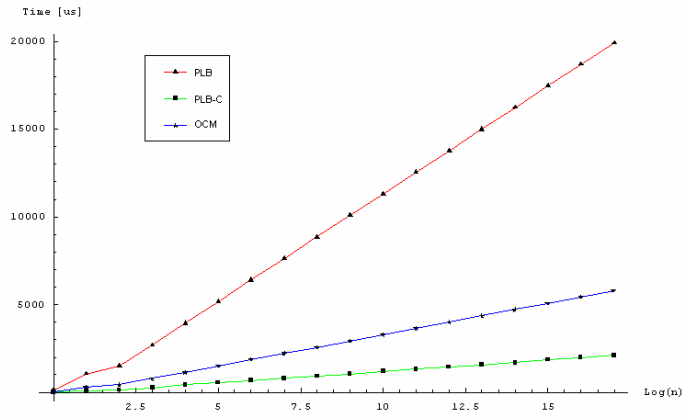


Figure 7.8. Worst-case join costs with software security functions

7.5.2.2 Hardware vs. software security modules

An essential design issue for the HiFlexRP relates to accelerating the time-consuming cryptographic operations. **Figure 7.9** compares the performance of the subtask Update for three realizations of the cryptographic primitives: a software-only, a hardware-only, and a mixed HW/SW realization. The HW/SW alternative, referred to as HS in **Figure 7.9**, is based on a shared AES core for the different security functions. All these system realizations are based on dynamic tree management. Obviously, the hardware-only implementation provides the highest performance compared to the other two design alternatives. Recall that this performance substantially influences the system security and QoS. Furthermore, lower join and disjoin costs enable supporting larger dynamic groups. In spite of its high performance compared to software, the HS alternative restricts the design flexibility, e.g. in the case of using different hardware key generation module.

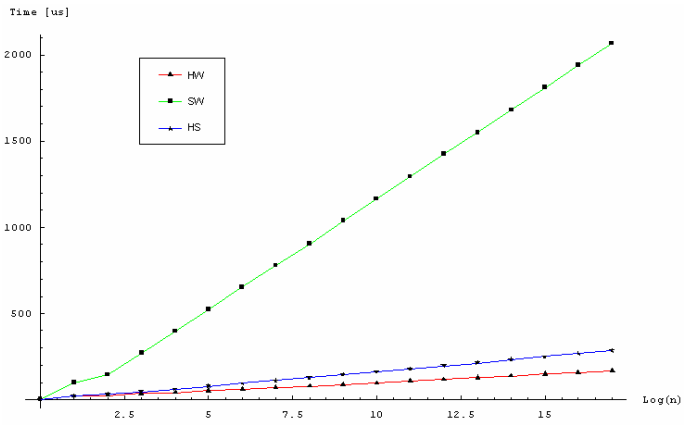


Figure 7.9. Worst-case join costs (HW vs. SW vs. HS)

7.5.2.3 Static vs. dynamic tree management

In Section 7.4.1, a quantitative comparison between static and dynamic tree management modes regarding memory utilization was given. With 4.5 MB needed to store tree auxiliary data in the dynamic case compared to 32 KB in the static case, the superiority of the last is evident. Regarding performance, Section 7.4.2 illustrated the dynamic tree management and qualitatively outlined the differences to the static management mode. To evaluate this pre-estimation, a timing measurement was performed for two design alternatives with pure hardware security modules, see **Figure 7.10**. Obviously, the static tree management is superior to the dynamic tree management in respect of rekeying performance, too.

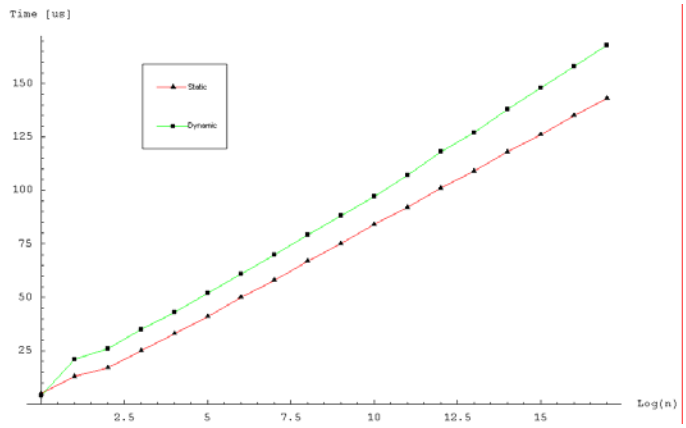


Figure 7.10. Worst-case join costs (static vs. dynamic tree management)

7.5.3 Sign Subtask and HW/SW Partitioning

In this section, the complete join process will be treated with both the subtasks *Update* and *Sign*, see **Figure 7.5**. The following time values were measured for a software-only implementation of these two subtasks.

Update (SW)	Sign (SW)
2,108 µs	52,574 µs

From this table it is obvious that the subtask Sign dominates the rekeying costs. Consequently, hardware acceleration for this task seems to be beneficial. For this purpose, the ECDSA algorithm is analyzed in more detail. When investigating the data dependencies in **Algorithm 7.1.D**, the subtask Sign can be divided into the following four subtasks, which are depicted in **Figure 7.11.**

1. ***PreEcMult:*** This subtask includes the generation and reduction of the random number *k* and represents Step 1 in **Algorithm 7.1.D**.
2. ***EcMult:*** The scalar multiplication – Step 2 in the ECDSA algorithm – is realized by this subtask.
3. ***ModInv:*** This subtask performs the modular inversion of *k* needed in Step 4.
4. ***PostEcMult:*** Steps 3 and 4 in the algorithm - except for the modular inversion - are associated with this subtask.

Note that the modular inversion is the second most expensive operation next to the scalar multiplication. Therefore, a dedicated subtask is assigned to this operation.

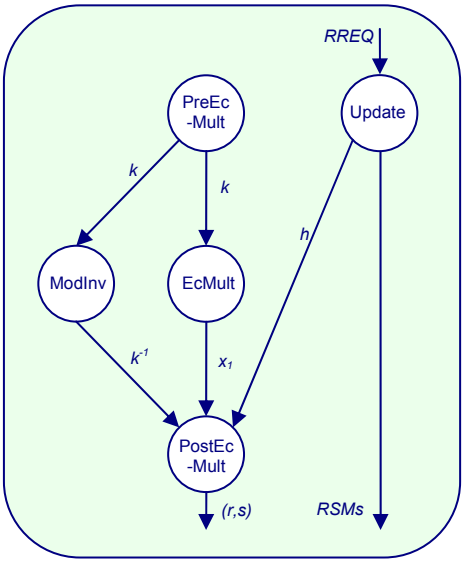


Figure 7.11. Extended rekeying DFG

The extended data flow graph (DFG) in **Figure 7.11** highlights a considerable amount of inherent parallelism in the rekeying task. The Update subtask, for example, may be executed in parallel to the subtasks PreEcMult, EcMult, and/or ModInv. For an accurate decision about a suitable parallelization, timing information on the different subtasks is required. For this purpose, new measurements of the software implementation according to the refined assignments given above were performed, which resulted in the following execution times of these five atomic subtasks.

Update (SW)	PreEcMult (SW)	EcMult (SW)	ModInv (SW)	PostEcMult (SW)
2,108 µs	210 µs	49,517 µs	2,449 µs	399 µs

7.5.3.1 HW/SW-Realization 1 (HW/SW-1)

From the previous table it can be seen that the scalar multiplication is by far the most expensive subtask and, therefore, lends itself to hardware acceleration.

The hardware implementation is based on the architecture proposed in [La06]. This architecture employs a hierarchy of three abstraction levels to manage the complexity of the EC point multiplication. The upper two levels mainly consist of finite state machines, which perform the control tasks necessary for the point multiplication (based on a variant of Lim/Lee exponentiation [Li94]) and the underlying operations for point doubling and addition (according to the algorithms specified in IEEE P1363 [Ie00]). The lowest level realizes the actual computations in the finite field. For the most critical operation, i.e. the modular multiplication, the Montgomery algorithm was employed [Mo85].

The hardware realization does not only improve the performance of EcMult considerably, but also enables the exploitation of the inherent parallelism in the rekeying task shown in **Figure 7.11**. By using this hardware module, a new time measurement was performed for this partitioning variant, which resulted in the following timing values.

Update (SW)	PreEcMult (SW)	EcMult (HW)	ModInv (SW)	PostEcMult (SW)
2,108 µs	210 µs	3,302 µs	2,449 µs	399 µs

Based on these timing values and the DFG presented in **Figure 7.11**, a task scheduling can now be employed as depicted in **Figure 7.12**. To avoid that the software has to wait for the EcMult result, this subtask must be started as soon as possible. This demands the execution of PreEcMult as the first subtask in the task profile in order to supply EcMult with the input value *k*. In contrast, when starting with Update, EcMult would be forced to start at the same time as ModInv at the earliest. This causes the software subtasks to wait for about 853 µs (= 3,302 µs - 2,449 µs). Note that **Figure 7.12** and subsequent schedules are not to scale.

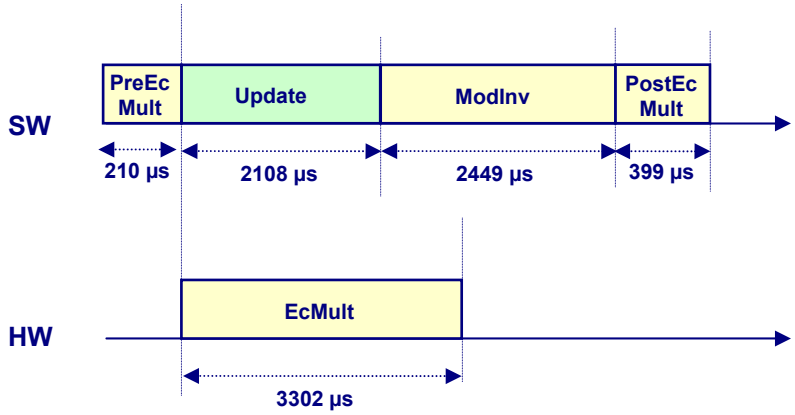


Figure 7.12. Task scheduling for design variant HW/SW-1

The design alternative HW/SW-1 results in a worst-case join time of 5,164 µs. Compared to the software-only solution this corresponds to a performance improvement of about 10.6 times. However, the parallelization possibilities in the rekeying task are not completely exploited yet. ModInv is data-independent of Update and EcMult and can therefore be executed in parallel to them. For this purpose, an additional hardware resource is necessary to execute either ModInv or Update, as depicted in the next design alternatives.

7.5.3.2 HW/SW-Realization 2 (HW/SW-2)

For this design alternative a dedicated hardware module for modular inversion was implemented, which resulted in an inversion time of just 1,091 µs. The relatively small speed-up is attributed to using a different modular inversion scheme in hardware (based the Fermat's little theorem [Me96]) than the one employed in software (based on the extended Euclidean algorithm [Me96]), for reuse reasons. Nevertheless, the overall execution time for the worst-case join operation equals now 3,850 µs, which corresponds to performance improvement of about 14.2 times compared to the software-only implementation. **Figure 7.13** depicts the scheduling for this design alternative.

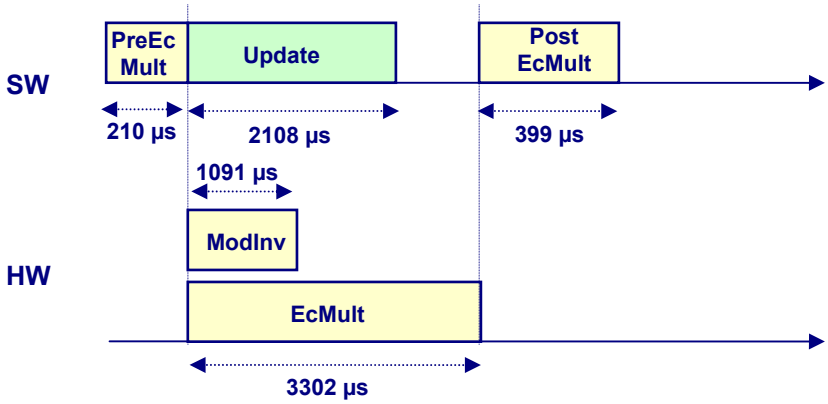


Figure 7.13. Task scheduling for design variant HW/SW-2

7.5.3.3 HW/SW-Realization 3 (HW/SW-3)

This design alternative accelerates the Update task using hardware components for key generation, encryption, and hash function. In this case, the ModInv subtask runs on the PowerPC processor. The scheduling for this design alternative is depicted in **Figure 7.14**. The performance improvement in this case equals the one resulting from the HW/SW-2 variant. Note that the part of the subtask Update regarding key tree management is still executed on software. This does not affect the scheduling because of the free gap in the software resources between the execution of ModInv and PostEcMult.

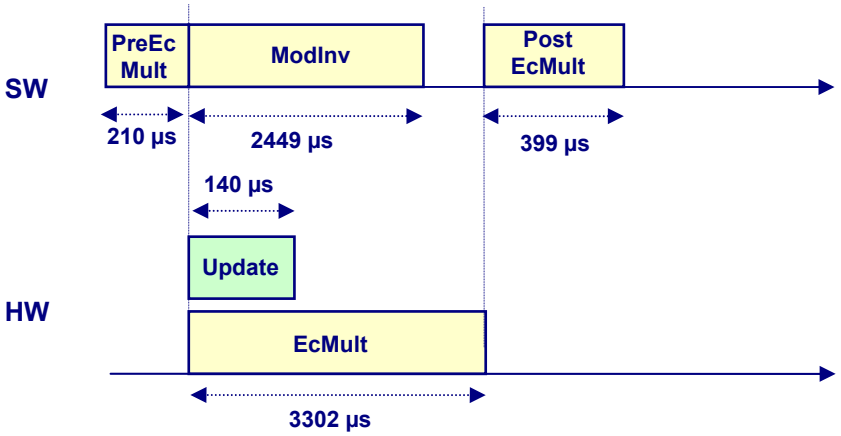


Figure 7.14. Task scheduling for design variant HW/SW-3

Resource usage consideration

So far, the resource usage in the different realizations was not taken into account, just their execution time. From this point of view, variants HW/SW-2 and HW/SW-3 seem to be fully equivalent. However, taking a look at the resource usage of these solutions, as detailed in **Table 7.2**, it can be seen that HW/SW-2 is more efficient in terms of resource usage. Nevertheless, the system designer may prefer the alternative HW/SW-3 in case that the system's flexibility and expandability are of high interest. In this case, the dedicated hardware modules for key generation, symmetric encryption and hash functions can be reused for other cryptographic applications beyond rekeying.

Table 7.2. Resource usage and overall performance

Implementation Variant	Resource Usage on Virtex-II Pro		Worst-case Join Time
	BRAMs	Slices	
SW	50 ≈ 36%	2,347 ≈ 18%	54,679 µs
HW/SW-1	53 ≈ 38%	4,667 ≈ 34%	5,164 µs
HW/SW-2	54 ≈ 39%	5,225 ≈ 38%	3,910 µs
HW-SW-3	83 ≈ 62%	8,420 ≈ 61%	3,910 µs

Remark on automated design approaches

The design task of the HiFlexRP was coined by a small design space because of the availability of already implemented hardware modules, such as AES-core. Therefore, a pure top-down design process was not applicable. Nevertheless, the design space exploration tool hCDM [Kl06] was used afterwards for comparison. This tool expects as input a task graph and some constraints to define which tasks may be executed on which resources. Based on this information the design space is explored and different Pareto-optimal solutions are proposed to the designer, who can decide on one thereof.

For the HiFlexRP case, this tool generated the variants SW-only, HW/SW-1, and HW/SW-2. The variant HW/SW-3, which offers higher flexibility for future expandability, was not generated automatically, since flexibility is not represented as a metric in the objective function of this tool.

7.6 HiFlexRP vs. Related Work

The overall execution time of the worst-case join rekeying was measured to be 3.91ms as shown in **Figure 7.14**. Note that the worst-case disjoin time equals this value, too. This is because of the high similarity of these two operations, on the one hand. The whole Update subtask is executed parallel to other subtasks with longer execution times, on the other. To point out the advantage of the HiFlexRP with regard to performance, **Table 7.3** depicts a comparison with related work, which provides timing information on rekeying costs based on prototype measurements including data source authentication [Wo00], [Am04]. Note that the extremely large timing value given in [Am04] represents the *"total elapsed time from the moment the group membership event happens until the time when the group key agreement finishes and the application is notified about the group change and the new key."*

Table 7.3. Performance comparison

		[Am04]	**[Wo00]**	**HiFlexRP**
Cryptographic primitives	**Encryption**	n/a	DES-CBC	AES-128
	Key generation	n/a	n/a	ANSI X9.17
	Secure hashing	n/a	MD5	AES-based Meyer hash
	Digital signing	RSA-1024	RSA-512	ECDSA-128
Group size		50	8,192	131,072
Execution time		Up to 640 *ms*	12 *ms* – 16.2 *ms*	3.91 *ms*
Measurement conditions		Total elapsed time for disjoin including protocol costs	Average-case disjoin, server processing time only	Worst case disjoin, HiFlexRP processing time only

8 Conclusion

This dissertation presented several novel solutions for the group key management in secure multicast. To keep clarity, the subjects were illustrated with rather high abstraction from details, which are not of fundamental significance for the general understanding. Thus, neither considerations to optimize the simulation process of the rekeying benchmark, nor details on the driver functionality of the HiFlexRP were addressed, for instance. At different points of this document it was referred to special constraints for some performed studies and it was pointed to future work to complete or refine these subjects.

Several items for the further development of this work are scheduled or already being treated. Three points thereof are mentioned in the following.

Extending the rekeying benchmark

The current benchmark prototype only considers the costs of cryptographic operations. Future development will take other cost factors on the server side and the communication overhead into account. In addition, further rekeying algorithms will be included and evaluated to allow more insight into proposed rekeying solutions in related work.

Networking the rekeying processors

The XUP card provides a 10/100 Ethernet port and the embedded design kit (EDK) from Xilinx provides an Ethernet controller as IP core for evaluation purposes. The controller can be run on the FPGA for eight hours free of licence, which is sufficient for prototyping. At the moment it is being tried to connect the rekeying processor to the network over this interface. By this means, the rekeying messages can be sent to group members without involving the registration and authentication server.

Automatic reduction of bus transactions

The design process of the HiFlexRP has raised a highly important and sophisticated problem. The background of this problem and the objective of its solution are presented in the following, in brief.

Neglecting the digital signing, it can be seen that the performance of the Real-Time Rekeying Processor exceeds that of the High-Flexibility Rekeying Processor. For the worst case join, 15 µs are necessary in the RTRP compared to 140 µs in the HiFlexRP. Note, however, that different group sizes are supported by these processors and that hashing costs are only included in the case of HiFlexRP. Nevertheless, the HiFlexRP offers lower

rekeying performance because of the high data transfer between the hardware modules and the software resources over the PLB bus. Based on the followed design strategy, the hardware modules for key generation, encryption, and hashing are realized separately and then connected to the PLB bus. The data dependencies between these modules were not considered in this design approach. When investigating the subtask Update, it can be seen that every generated key must be encrypted and every encrypted key must be hashed. Thus, the subtask Update is specified by a high data dependency between its subtasks. In the current realization, this results in high bus transfer overhead, which deteriorates the overall performance of the Update operation.

An evident solution to this problem relates to reducing the bus transactions over the PLB by replacing them by transfers over internal channels between the different modules. A manual transformation of the current design to a new design with lower bus transaction overhead, however, is largely time-consuming since a considerable part of the software functionality must be migrated to hardware.

An essential research work in the near future will address this transformation by means of computer-aided methods. For this purpose, an approach will be developed, which generates an optimized design alternative, based on a first variant, which can be the result of a bottom-up design process enforced by using IP cores. The desired approach includes three main steps. First, the input design is entered and analyzed. Second, based on the analyzed input an optimized alternative is searched by applying system-level design methods. Thirdly, the selected design alternative is synthesized for implementation on the same platform. An essential advantage of this approach relates to its performance because of the starting with an actual solution und due to the availability of several blocks, which remain unchanged or are just lightly modified. By these means the synthesis process can be accelerated considerably.

Bibliography

[Ab05] Abrooy M., "Dynamische Baumverwaltung für den Rekeying Prozessor: Analyse & Simulation", Studienarbeit, Technische Universität Darmstadt, May 2005.

[Ac07] http://www.actel.com/

[Ad07] http://www.aldec.com/

[Al07] http://www.alpha-data.com/

[Al96] Almeroth K., and Ammar M., "Collecting and Modeling of Join/Leave Behavior of Multicast Group Members in the MBone", in Proc. of High Performance Distributed Computing Focus Workshop, Syracuse, New York, August 1996.

[Am04] Amir Y., Kim Y., Nita-Rotaru C., and Tsudik G., "On the Performance of Group Key Agreement Protocols", ACM Trans. on Information Systems Security, 7(3), 2004, pp. 457-488.

[An00] U.S. Federal Information Processing Standard, ANSI X9.17 Pseudorandom Bit Generator, 2000.

[Ba00] Balenson D., McGrew D.A., and Sherman A., "Key Management for Large Dynamic Groups: One-Way Function Trees and Amortized Initialization", Internet draft, draft-irtf-smug-groupkeymgmt-oft-00.txt, IRTF, Aug. 2000, work in progress.

[Bi00] Rodeh O., Birman K.P., and Dolev D., "Optimized Group Rekey for Group Communication Systems", In Proc. of Network and Distributed System Security Symposium, San Diego, Ca, Feb. 2000.

[Bi06] Bieser C., and Mueller-Glaser, K.-D., " Rapid Prototyping Design Acceleration Using a Novel Merging Methodology for Partial Configuration Streams of Xilinx Virtex-II FPGAs", in Proc. of the 17th IEEE International Workshop on Rapid System Prototyping, p.p. 193-199, Chania, Crete, June 2006.

[Br99] Briscoe B., "MARKS: Zero Side Effect Multicast Key Management Using Arbitrarly Revealed Key Sequences", In Proc. of First Int. Workshop on Networked Group Communication (NGC), Pisa, Italy, November 1999.

[Bu06] Bursky D., "Study: Field-programmable logic rules", A survey conducted by A.G. Edwards & Sons and sponsored by A.G. Edwards, Beacon Technology Partners and

EE Times, http://www.eetimes.com/, June 2006.

[Ca00] Canetti R., Rohatgi P., and Cheng P., “Multicast Data Security Transformations: Requirements, Considerations, and Protocol Design”, draft-irtf-smug-data-trasforms-00.txt, IRTF, June 2000.

[Ca97] Cain B., Deering S., and Thyagarajan A., “Internet Group Management Protocol”, Version 3. Internet Draft, Work in Progress, draft-ietf-idmr-igmp-v3-00.txt, November, 1997.

[Ca99] Canetti R., et al., “Multicast Security: A Taxonomy and Efficient Constructions”, in Proc. IEEE INFOCOM, New York, March 1999.

[Ch02] Chen W., and Dondeti L.R., “Performance Comparison of Stateful and Stateless Group Rekeying Algorithms”, in Proc. of Int. Workshop in Networked Group Communication, 2002.

[Cl07] http://www.cryptopp.com/

[Da01] Davis C.R., “IPSEC Securing VPNs”, McGraw-Hill, 2001.

[Da02] Daemen J., Rijmen V., “The Design of Rijndael”, Springer Verlag, 2002.

[Di00] Dinsmore P.T., et al., “Policy-based Security Management for Large Dynamic Groups: an Overview of the DCCM Project”, in Proc. of the DAPRA Information Survivability Conference and Exposition, Vol. I, Hilton Head, pp. 64-73, January 2000.

[Di06] http://www.digilentinc.com/

[Di76] Diffie W., and Hellman R., “Multiusers Cryptographic Techniques” Proceedings of the AFIPS National Computer Conference, pp. 109-112, June 1976.

[Do00] Dondeti L.R., Mukherjee S., and Samal A., “DISEC: A Distributed Framework for Scalable Secure Many-to-Many Communication”, In Proc. of IEEE Symposium on Computers and Communications, Antibes-Juan les Pins, France, July 2000.

[Dp06] D’Paiva S., “Unified FPGA-ASIC Design Flow Provides Designers Versatility in Meeting Production Goals”, http://www.fpgajournal.com/articles_2006/20061205_magma.htm, Dec. 2006.

[Dr07] Design & Reuse website: http://www.us.design-reuse.com/.

[Ec05] Eclipse, Open Development Platform, Version 3.1.2, http://www.eclipse.org/.

[Ec06] Eckert C., “IT-Sicherheit Konzepte – Verfahren – Protokolle“, Oldenburg Wissenschaftsverlag, 2006.

[El85] ElGamal T., “A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms”, IEEE Trans. on Information Theory, Vol. IT-31, No. 4, pp. 469-472, 1985.

[Er02] Ernst M., Jung M., Madlener F., Huss, S.A., and Bluemel R., “A Reconfigurable System on Chip Implementation of Elliptic Curve Cryptography over GF(2^n)”, In Proc. of Workshop on Cryptographic Hardware and Embedded Systems CHES2002, Redwood City, CA, August 2002.

[Ga92] Gajski, D. D., Wu A., Dutt N., and Lin S., "High-Level Synthesis, Introduction to Chip and System Design", Kluwer Academic Publishers, 1992.

[Ga94] Gajski, D. D., Vahid F., Narayan S., and Gong J., "Specification and Design of Embedded Systems", Prentice hell, 1994.

[Ge05] Gerenz S.H., "Algorithms for VLSI Design Automation", John Wiley & Sons, 2005.

[Gn07] http://www.gnu.org/

[Go03] Goshi J., and Ladner R.E., "Algorithms for Dynamic Multicast Key Distribution Trees", in Proc. of ACM Symp. on Principles of Distributed Computing, 2003, pp. 243-251.

[Go04] Goodrich M.T., Sun J.Z., and Tamassia R., "Efficient Tree-Based Revocation in Groups of Low-State Devices", In Proc. of CRYPTO 2004, LNCS 3152, pp. 511-527, Santa Barbara, California USA, August 2004.

[Go05] Good T. and Benaissa M., "AES on FPGA from the Fastest to the Smallest", CHES 2005, LNCS 3659, pp. 427-440, 2005.

[Go99] Goncalves M., Niles K., "IP Multicasting Concepts and Applications", McGraw-Hill, 1999.

[Gr02] Groetker T., Liao S., Martin G., and Swan S., "System Design with SystemC", Kluwer Academic Publishers.2002.

[Ha01] Halsall F., "Multimedia Communications", Addison-Wesley, 2001.

[Ha03] Hardjono T., and Dondeti L.R., "Multicast and Group Security", Artech House, 2003.

[Ha99] Wallner D., Harder E., and Agee R., "Key Management for Multicast: Issues and Architectures", RFC 2627, IETF, June 1999.

[Ho04] Hoermann W., Leydold J., and Derflinger G., "Automatic Nonuniform Random Variate Generation", Spriger, 2004.

[Ho04] Howarth M.P., Iyengar S., Sun Z., and Cruickshank H., "Dynamics of Key Management in Secure Satellite Multicast", IEEE Journal on Selected Areas in Communications, Vol. 22, No. 2, pp. 308-319, Feb. 2004.

[Ib99] http://www-306.ibm.com/chips/techlib/techlib.nsf/productfamilies/CoreConnect_Bus_Architecture/

[Ie00] IEEE (Institute of Electrical and Electronics Engineers), "Standard Specifications for Public-Key Cryptography", Annex A, 2000. http://grouper.ieee.org/groups/1361/.

[Ie01] IEEE (Institute of Electrical and Electronics Engineers), "IEEE Standard Verilog Hardware Description Language, 1364-2001".

[Ie93] IEEE (Institute of Electrical and Electronics Engineers), "IEEE Standard VHDL Language Reference Manual, 1076-1993".

[Is89] ISO (International Organization for Standardization), "Data Cryptographic Techniques – Data Integrity Mechanism Using a Cryptographic Check Function Employing a Block Cipher algorithm", ISO 9797, 1989.

[Ji02] Jin Q., Jianhua G., Ming J., and Zhang Z., “On Batch Rekeying Based Membership Dynamics Model of Multicast”, In Proc. of IEEE TENCON’02, pp. 145-147, Beijing China, Oct. 2002.

[Jo97] John M., Smith S., “Application-Specific Integrated Circuits”, Addison-Wesley, second printing, August 1997.

[Ke05] Kecher C., “UML 2.0“, Galileo Computing, 2005.

[Ke99] Keating M., Bricaudd P., “Reuse Methodology Manual”, second edition, Kluwer Academic Publishers, 1999.

[Kl06] Klaus S., “System-Level-Entwurfsmethodik eingebetteter Systeme”, Dissertation, Shaker Verlag, 2006.

[Ko87] Koblitz N., “Elleptic Curve Cryptosystems”, Mathematics of Computation, Vol. 48, No. 177, pp. 203-209, 1987.

[Ku04] Kumar M., Mentor Graphics Europe, “Using Physical Synthesis Techniques to Achieve Timing Closure in FPGAs”, in proc. of Sophia Antipolis forum of Microelectronics SAME 2004, Sophia Anipolis France, Oct. 2004.

[La06] Laue R. and Huss S.A., “A Novel Memory Architecture for Elliptic Curve Cryptography with Parallel Modular Multipliers”, In Proc. of IEEE Int. Conf. on Field Programmable Technology, Bangkok Thailand, Dec. 2006.

[Li01] Li X., Yang Y.R., Gouda M., and Lam S.S., “Batch rekeying for Secure Group communications”, in Proc. of Int. WWW Conf., Hong Kong, May 2001.

[Li06] Li J.H., Levy R., Yu M., and Bhattacharjee B., “A Scalable Key Management and Clustering Scheme for Ad-Hoc Network”, in proc. of ACM Inter. Conf. on Scalable Information Systems, Hong Kong, May 2006.

[Li94] Lim C.H., and Lee P.J., “More Flexible Exponentiation with Precomputation”, in Advances in Cryptography – Crypto’94, LNCS 839, pp. 95-107, August 1994.

[Lu05] Lu H., “ A Novel High-Order Tree for Secure Multicast Key Management”, IEEE Trans. on Computers, vol. 54, No. 2, February 2005, p. 214-224.

[Ma00] Makimoto T., “The Rising Wave of Field Programmability”, in proc. FPL 2000, LNCS 1896, pp. 1-6, Villach Austria, August 2000.

[Ma03] Mangard S., Aigner M., and Dominikus S., “A Highly Regular and Scalable AES Hardware Architecture” IEEE Trans. On Computers, Vol 52, No. 4, pp. 483-491, April 2003.

[Ma04] Zhang J., Ma F., Bai Y., and Li M., “Performance Analysis of Batch Rekey Algorithm for Secure Group Communications”, in Proc. PDCAT 2004, LNCS 3320, pp. 829-832, Singapore, 2004.

[Ma85] Matyas S., Meyer C., and Oseas J., “Generating Strong One-Way Functions with Cryptographic Algorithm”, IBM Disclosure Bulletine, 27(10A): 5658-6559, March 1985.

[Me96] Menezes A.J., van Oorschot P.C., and Vanstone S.A., “Handbook of Applied

Cryptography", CRC Press, 1996.

[Mi86] Miller, V., "Use of Elleptic Curves in Cryptography", Advances in Cryptology, CRYPTO'85, LNCS, Vol. 218, pp. 417-426, 1986.

[Mi97] Mittra S., "Iolus: A Framework for Scalable Secure Multicasting", in Proc. of ACM SIGCOMM, pp. 277-288, Cannes, France, September 1997.

[Mi99] Miller C.K., "Multicast Networking and Application", Addison Wesley Longman, 1999.

[Mo04] Moharrum M., Mukkamala R., and Eltoweissy M., "Efficient Secure Multicast with Well-Populated Multicast Key Trees", in proc. of IEEE ICPADS, pp. 214, July 2004.

[Mo65] http://www.intel.com/technology/mooreslaw/index.htm

[Mo85] Montgomery P.L., "Modular Multiplication Without Trial Division", in Mathematics of Computation", Vol. 44, No. 170, pp. 519-521, 1985.

[Mo99] Moyer M.J., Tech G., Rao J.R., and Rohatgi P., "Maintaining Balanced Key Trees for Secure Multicast", Internet draft, June, 1999, http://www.securemulticast.org/draft-irtf-smug-key-tree-balance-00.txt.

[Ms00] http://www.ietf.org/html.charters/msec-charter.html

[Mu00] Mueller-Glaser, K.-D., and Bortolazzi J., "An Approach to Computer-Aided Specification", IEEE Journal of Solid-State Circuits, Vol. 25, No. 2, pp. 335-345, April 1990.

[Ng05] Ng W.H.D., and Sun Z., "Multi-Layers Balanced LKH", in Proc. of IEEE Int. Conf. on Communication ICC, May 2005, pp. 1015-1019.

[Ni01] NIST (National Institute of Standards and Technology), "Advanced Encryption Standard (AES)", Federal Information Processing Standard 197, November 2001.

[Ni81] NIST (National Institute of Standards and Technology), "American National Standard for Data Encryption Algorithm (DEA)", ANSI X3.92, 1981.

[Ni85] NIST (National Institute of Standards and Technology), "American National Standard for Financial Institution Key Management (Wholesale)", American Banker Association, 1985.

[Pe03] Pegueroles, and Rico-Novella F., "Balanced Batch LKH: New Proposal, Implementation and Performance Evaluation", in Proc. IEEE Symp. on Comupters and Communications, 2003, pp. 815.

[Pu98] Pusateri T., "Distance Vector Multicast Routing Protocol Specification". Internet Draft, Work in Progress, draft-ietf-idmr-dvmrp-v3-06.txt, March, 1998.

[Ra01] Rafaeli S., Mathy L., and Hutchison D., "EHBT: An Efficient Protocol for Group Key Management", In Proc. of NGC 2001, LNCS 2233, pp. 159-171, London UK, Nov. 2001.

[Ri78] Rivest R., Shamir A., and Adleman L., "A Method for Obtaining Digital Signatures and Public Key Cryptosystems", Communications of the ACM, Vol. 21, No. 2, pp. 120-126, February 1978.

[Ro00] Rodeh O., Birman K.P., and Dolev D., “Using AVL Trees for Fault Tolerant Group Key Management”, Tech. Rep. 2000-1823, Cornell University, 2000.

[Ro06] Roh J.H., and Lee K.H. “Key Management Scheme for Providing the Confidentiality in Mobile Multicast”, in proc. of IEEE ICACT’06 , Phoenix Park, Korea, Feb. 2006.

[Sc96] Schneier B., “Applied Cryptography”, John Wiley & Sons, Inc., 1996.

[Se04] Zhu S., Setia S., Xu S., and Jajodia S., “GKMPAN: An Efficient Group Rekeying Scheme for Secure Multicast in Ad-Hoc Networks”, in Proc. of IEEE/ACM MobiQuitous, pp. 42-51, Boston, Massachusetts, USA, August 2004.

[Sh03] Sherman A.T., et al., “Key Establishment in Large Dynamic Groups Using One-Way Function Trees”, IEEE Trans. on Software Engineering, Vol. 29, No. 5, pp. 444-458, May 2003.

[Sh04] Shoufan A. and Sorin A. Huss, “A Scalable Rekeying Processor for Multicast Pay-TV on Reconfigurable Platforms”, Workshop on Application Specific Processor, IEEE/ACM Int. Conf. on Hardware/Software Codesign and System Synthesis, Stockholm, Sweden, Sep. 2004. Best paper award.

[Sh05] Shoufan A., Huss S., and Cutleriwala M., “A Novel Batch Rekeying Processor Architecture for Multicast Key Management”, Int. Conf. on High Performance Embedded Architectures & Compilers, LNCS 3793, pp. 169-183, Barcelona, Spain, Nov. 2005.

[Sh06] Shoufan A., and Huss S.A., ”Rekeying Prozessor: Eine skalierbare Lösung für die Schlüsselverwaltung in Gruppenkommunikation”, thema FORSCHUNG, Technische Universität Darmstadt Ausgabe 1/2006, pp.86-89.

[Sh07a] Shoufan A., Laue R., and Huss S.A., “Reliable Performance Evaluation of Rekeying Algorithms in Secure Multicast” to appear in IEEE Int. Symposium on a World of Wireless, Mobile and Multimedia Networks, Helsinki, Finland, June 2007.

[Sh07b] Shoufan A., Laue R., and Huss S.A., “High-Flexibility Rekeying Processor for Key Management in Secure Multicast” to appear in IEEE Int. Symposium on Embedded Computing SEC-07, Niagara Falls, Canada, May 2007.

[Sh07c] Shoufan A., Laue R., and Huss S.A., “Secure Multicast Rekeying: A Case Study for HW/SW-Codesign” 10. Workshop “Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen”, Erlangen Germany, March 2007.

[Sm98] http://www.securemulticast.org/smug-index.htm

[So06] http://www.sony.net/Products/SC-HP/cx_news/vol33/sideview.html

[Sy07] http://www.synplicity.com

[Ta00] Tannenbaum A.S., “Computernetzwerke”, Prentice Hall, 2000.

[Um06] Um H., and Delp E.J., “A Secure Group Key Management Scheme for Wireless Cellular Networks”, In Proc. of IEEE Inter. Conf. on Information Technology: New Generations, Las Vegas, Nevada, USA, April 2006.

[Va92] Vanstone S., „Responses to NIST's Proposal", Communication of the ACM, Vol. 35, pp.50-52, July 1992.

[Wa98] Wannenmacher M., "Das FPGA Kochbuch", International Thomson Publishing GmbH, 1998.

[Wa99] Waldvogel M., Caronni G., Sun D., Weiler N., and Plattner B., "The VersaKey Framework: Versatile Group Key Management", IEEE J. on Selected Areas in Communications, Vol. 17, No. 8, august 1999, pp. 1614-1631.

[Wo00] Wong C.K., Gouda M., Lam S.S., "Secure Group Communications Using Key Graphs", IEEE/ACM Trans. on Networking, Vol. 8, No. 1, pp. 16-30, February 2000.

[Wo07] http://www.wolfram.com

[Xi02] Xilinx, "Virtex-II Pro Platform FPGA Handbook", Oct. 2002.

[Xi04] Lund K., "PLB vs. OCM Comparison Using the Packet Processor Software", Xilinx Application Note XAPP644 (v1.1), Oct. 2004.

[Xi05] Xilinx, "PowerPC 405 Processor Block Reference Guide", V2.1, July 2005.

[Xi07] http://www.xilinx.com

[Ya01] Yang Y.R., et al., „Reliable Group Rekeying: Design and Performance Analysis", in proc. of ACM SIGCOMM, San Diego, CA, August 2001.

[Zh01] Zhang X. B., et al., "Protocol Design for Scalable and Reliable Group Rekeying", in proc. of SPIE Conf. on Scalability and Traffic Control in IP Networks, Denver, August 2001.

[Zh03] Zhu S., et al., "Performance Optimizations for Group Key Management Schemes for Secure Multicast", in proc. of IEEE/ACM Int. conf. on Distributed Computing Systems, 2003.

[Zh04] Zhang Xi. and Pahri K.K., "High-Speed VLSI Architectures for the AES Algorithm", IEEE Trans. on Very Large Scale Integration Systems, Vol. 12, No. 9, pp. 957-967, Sep. 2004.

List of Publications

1. Shoufan A. and Sorin A. Huss, “A Scalable Rekeying Processor for Multicast Pay-TV on Reconfigurable Platforms”, Workshop on Application Specific Processor, IEEE/ACM Int. Conf. on Hardware/Software Codesign and System Synthesis, Stockholm, Sweden, Sep. 2004. “Best paper award”.
2. Shoufan A. and Sorin A. Huss, “Interactive Identification and Correction of Structural Modeling Errors in Conservative VHDL-AMS Models”, IEEJ International Analog VLSI Workshop, Macao, China Nov. Oct. 2004.
3. Shoufan A. and Sorin A. Huss, “Zur Reduktion des strukturellen Gleichungssatzes konservativer VHDL-AMS Modelle”, Analog’05, 8. GMM/ITG-Diskussionssitzung. Entwicklung von Analogschaltungen mit CAE-Methoden, Hannover, März 2005.
4. Shoufan A., Huss S., and Cutleriwala M., “A Novel Batch Rekeying Processor Architecture for Multicast Key Management”, Int. Conf. on High Performance Embedded Architectures & Compilers, LNCS 3793, pp. 169-183, Barcelona, Spain, Nov. 2005.
5. Shoufan A. and Huss S.A., ”Rekeying Prozessor: Eine skalierbare Lösung für die Schlüsselverwaltung in Gruppenkommunikation”, thema FORSCHUNG, Technische Universität Darmstadt Ausgabe 1/2006, pp.86-89.
6. Shoufan A. and Sorin A. Huss, “Construction of a SPICE-similar Simulator for Education”, Analog’05, 8. GMM/ITG-Diskussionssitzung. Entwicklung von Analogschaltungen mit CAE-Methoden, Hannover, Dresden, Germany, Sep. 2006.
7. Shoufan A., Laue R., and Huss S.A., “High-Flexibility Rekeying Processor for Key Management in Secure Multicast” to appear in IEEE Int. Symposium on Embedded Computing SEC-07, Niagara Falls, Canada, May 2007.
8. Shoufan A., Laue R., and Huss S.A., “Secure Multicast Rekeying: A Case Study for HW/SW-Codesign” 10. Workshop “Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen”, Erlangen Germany, March 2007.
9. Shoufan A., Laue R., and Huss S.A., “Reliable Performance Evaluation of Rekeying Algorithms in Secure Multicast” to appear in IEEE Int. Symposium on a World of Wireless, Mobile and Multimedia Networks, Helsinki, Finland, June 2007.

List of Supervised Theses

1. Sven Rettig, “Konzeption, Aufbau und Erprobung eines Projektionsdisplays mit Interner Redundanzarchitektur für sicherheitskritische Anwendungen”, Diplomarbeit, 2003.
2. Abdelouahid Taadou, “Entwurf eines HW-Moduls zur Verschlüsselung und Schlüsselgenerierung in einem Rekeying Prozessor auf Rekonfigurierbarer Plattform” Studienarbeit, 2004.
3. Peter Bungert, “Entwurf und Implementierung eines Controllers für einen Rekeying Prozessor auf einer FPGA Plattform”, Studienarbeit, 2004.
4. Bienvenu Tatsi, “Beschleunigung des Analogsolvers durch Minimierung der Strukturgleichungen von VHDL-AMS Modellen” Studienarbeit, 2004.
5. Zhaoming Dai ”Interaktive Fehlersuche durch Extrahierung der Knoten-, und Maschengleichungen in konservativen VHDL-AMS Modellen”, Diplomarbeit, 2004.
6. Murtuza Cutleriwala, “Batch Rekeying Processor on Reconfigurable Platform”, Master Thesis, 2005.
7. Mojtaba Abrooy, “Dynamische Baumverwaltung für den Rekeying Prozessor, Analyse und Simulation” Studienarbeit, 2005.
8. Marcus Lindner und Joana Otetelisanu “ Effiziente Hardwareimplementierungen des neuen Verschlüsselungsstandards AES“, CAE/CAD-Praktikum, 2006.
9. Tobias Teichner, “HW/SW Entwurf für den Rekeying Prozessor”, Diplomarbeit, 2007.
10. Torsten Hahn, “Digitale Kenngrößenerkennung, Messunsicherheitsbetrachtung und Drahtlose Messwertübertragung von Mehrkomponentenaufnehmern”, Diplomarbeit, 2007.
11. Dominik Litzinger, “Gesteurter Gleichrichter mit Spannungsregler”, Studienarbeit, laufend.