Master Thesis

Group Key Agreement for Ad Hoc Networks

Lijun Liao

Date: 06 July 2005
Supervisor: M.Sc. Mark Manulis

Ruhr-University Bochum, Germany

Chair for Network and Data Security
Prof. Dr. Jörg Schwenk
Homepage: www.nds.rub.de

Erklärung

Hiermit versichere ich, dass ich meine Diplomarbeit selbst verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Bochum Deutschland, 06.07.2005
Ort, Datum

Lijun Liao

Acknowledgements

This diploma thesis was done in Chair for Network and Data Security, Ruhr-University Bochum. I would like to thank M.Sc. Mark Manulis for his encouragement and guidance as my supervisor. I also want to thank him for reading through and correcting my texts.

I wish to thank Prof. Dr. Jörg Schwenk for his advice and supervision. I also wish to thank Dipl.-Ing. Daniel Hamburg of Integrated Information System Group for his help in the implementation with J2ME. For the help in the programming in Java I wish to thank Babak Nasri, Torsten Nitschke and Holger Ebel of AuthentiDate International AG.

Finally I would like to thank my wife Min for all her support, faith and care, and my parents, my sister and brothers for all their financial support.

Abstract

Over the last 30 years the study of group key agreement has stimulated much work. And as a result of the increased popularity of ad hoc networks, some approaches for the group key establishment in such networks are proposed. However, they are either only for static group or the memory, computation and communication costs are unacceptable for ad-hoc networks.

In this thesis some protocol suites from the literature (2^d-cube, 2^d-octopus, Asokan-Ginzboorg, CLIQUES, STR and TGDH) shall be discussed. We have optimized STR and TGDH by reducing the memory, communication and computation costs. The optimized version are denoted by μSTR and μTGDH respectively.

Based on the protocol suites μSTR and μTGDH we present a Tree-based group key agreement Framework for Ad-hoc Networks (TFAN). TFAN is especially suitable for ad-hoc networks with limited bandwidth and devices with limited memory and computation capability.

To simulate the protocols, we have implemented TFAN, μSTR and μTGDH with J2ME CDC. The TFAN API will be described in this thesis.

KEYWORDS: μSTR, μTGDH, TFAN, ad-hoc networks, group key agreement

Contents

1. Ad Hoc Networks

Ad hoc means in Latin „formed for“ or „concerned with one specific purpose“. And what is a wireless ad hoc network? There is still no consistent definition of general ad hoc network properties. And it is not easy to choose a suitable definition. After comparison of a few of definitions we choose that of NIST [83] for our purpose. „A wireless ad hoc network is a collection of autonomous nodes or terminals which communicate with each other by forming a multi-hop radio network and maintaining connectivity in a decentralized manner“.

The principle behind ad hoc networking is multi-hop relaying, which means that the messages are transmitted by the other nodes if the target node is not directly reachable. The absence of any central coordinator and base station makes it difficult to manage the network. Figure 1.1 illustrates an example of an ad hoc network which contains two laptops, two cell phones and two PDAs. The dashed line indicates the wireless link. Since node A cannot reach node D directly, the data from A to D must be transmitted by nodes B and C.

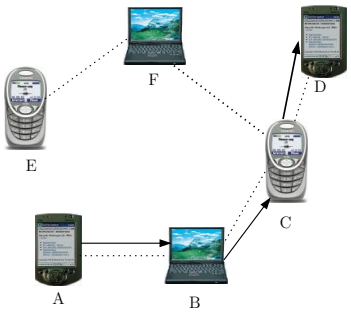


Figure 1.1.: An ad hoc network

The nodes or terminals are such as smart sensors, cell phones, PDAs and laptops. The wireless data transmission technologies such as Bluetooth [118] and 802.11 [44] allow efficient communication and are widely used in military, commerce and private areas. A. Khalili $et.al$. [55] have listed a set of differences between ad hoc networks and infrastructure based networks:

- **No fixed topology:** The network topology in an ad-hoc wireless network is highly dynamic due to the mobility of nodes. They may move in and out of the range of each other. The topology changes if one of those events happens, e.g. the route table and the multicast table must be changed accordingly. This increases the difficulty to management the network.
- **Limited energy:** Mobile devices use generally battery power, which is exhaustible. In order to save the energy, some devices may be in sleepy mode. During this period they are possibly not reachable, or do not process traffic, or change to normal mode with latency. On one hand most wireless devices use spread spectrum communications, which need the receiving and decoding of the signal. These are expensive operations that consume much power. On the other hand some complex computations are also very expensive, for example modular exponentiation, which makes it difficult to implement the public key systems for ad hoc networks.
- **Limited processor:** Most mobile devices have cheap and slow processors, because fast processors cost much more and the size should be as smart as possible to make it easy to take. Hence it takes

much time to operate some complex computations. The most PDAs have currently processors of several hundred MHz.

- **Limited storage capability and other resources:** Because of the size and cost restrictions, the most mobile devices are equipped with limited storage capability. For example, iPAQ hx4700 series of HP have only 192 MB memory. Due to the wireless technologies the network bandwidth is also limited. For example, some PDAs of HP are equipped with WLAN 802.11b, and Bluetooth 1.2.
- **Transient connectivity and availability:** Many nodes may not be reachable at some time so that they can save power.
- **Each node is a router:** The nodes out of the range of a fixed node can not be directly reached by this node. They can only be reached by packet forwarding of other nodes.
- **Shared physical medium:** Unlike wired networks, every device within the range can access the transmission medium.
- **Lack of central management:** Ad hoc networks can be established everywhere and every time. Generally there is no central management available, and we can also not assume that any information is shared.

Due to the lack of fixed infrastructure and limited resources, it will be much more complex to adapt protocols and other technologies from the infrastructure based networks.

Now we have an outline of wireless network. In the following sections we will describe the infrastructure, the routing and multicast routing of ad hoc networks.

1.1. Infrastructure

Since no consistent categories of ad hoc networks are available, we use the mostly used cases, namely mobile ad hoc networks (MANETs) and smart sensor networks.

1.1.1. Mobile ad hoc networks (MANETs)

Along with the development of the next generation of wireless communication systems, there will be a need for the rapid deployment of independent mobile users. Some examples of possible uses include students using laptops to participate in an interactive lecture, business associates sharing information during a meeting, and emergency disaster relief personnel coordinating efforts after a hurricane or earthquake. Such network scenarios cannot rely on centralized and organized connectivity, and can be conceived as applications of Mobile Ad Hoc Networks. A MANET is an autonomous collection of mobile users that communicate over relatively bandwidth constrained wireless links. Because of the mobility of the nodes, the network topology may change rapidly and unpredictably.

1.1.2. Smart sensor networks

A wireless ad hoc sensor network consists of a number of sensors spread across a geographical area. Each sensor has wireless communication capability and some level of intelligence for signal processing and networking of the data. Some examples include a group of soldier establishing communication for tactical communication or a measurement of the air pollution.

There are two ways to classify smart sensor networks, namely whether the nodes are individually addressable, and whether the data in the network is aggregated. The sensor nodes in a superhighway network should be individually addressable, so that one can determine the location of the sensors. Yet the addressability of the sensors which are used to measure the temperature and wind is not so important, because they vary only a little in the same location.

Both, MANETs and Sensor networks can be can be further classified into two broad types: homogeneous and heterogeneous networks. In homogeneous networks all nodes are identical in terms of battery energy and hardware complexity. While in a heterogeneous sensor network, two or more different types of nodes with different battery energy and functionality are used. The motivation is that the more complex hardware and the extra battery energy can be embedded in some nodes, thereby reducing the hardware cost of the rest of the network.

1.2. Routing Protocols for Ad Hoc Wireless Networks

An ad-hoc wireless network consists of a set of mobile nodes that are connected by wireless links. As we have seen, the network topology changes randomly while the nodes move on. Due to the highly dynamic topology and lack of central management, the protocols used in a traditional network to find a path from a source node to a destination node cannot be directly used in wireless ad-hoc networks. So a lot of routing protocols for ad hoc networks have been developed in the recent past.

1.2.1. Requirements

Due to the issues in ad-hoc networks discussed above, a routing protocol for wireless ad hoc networks should satisfy the following requirements:

- Fully distributed
- Adaptive to frequent topology changes
- Route computation and maintenance must involve a minimum number of nodes.
- Minimum of the number of packet collision
- Provide a certain level of quality of service (QoS)
- Use the limited resources such as bandwidth carefully

1.2.2. Classification of routing protocols

We can classify the routing protocols for ad hoc networks according to different criteria. They are routing protocols based on the following criteria:

1. Routing information update mechanism
   These protocols can be driven either by a routing table or on demand. In the first case, every node stores the network information in a routing table, which will be updated periodically. In order to get the path to the destination the node uses an appropriate path-finding algorithm to find the closest path. The typical protocols of this case are DSDV [90], WRP [76], CGSR [21], STAR [38], OLSR [22], FSR [47], HSR [47], and GSR [19].

   In the second case the nodes do not need to maintain the network topology. They get the path when they need it, by using a connection establishment process. The advantage is that nodes do not need to exchange routing information periodically. There are also some protocols which combine both features. The typical protocols of this case are DSR [52], AODV [91], ABR [120], SSA [32], FORP [112], and PLBR [106].

   Some protocols, such as CEDAR [103], ZRP [40], and ZHLS [51], combine both features. We called those protocols hybrid routing protocols.

2. Use of temporal information for routing
   This classification is based on the use of temporal information used for the routing process. Since ad hoc networks are highly dynamic, it is very important to use the temporal information for routing. According to the time of the information, we get two further classifications in this category.

a) Routing protocols using post temporal information which use information about the past status of the links or the status of links at the time of routing to make routing decisions. Such protocols as DSDV [90], WRP [76], STAR [38], AODV [91], FSR [47], HSR [47], and GSR [19] are of this class.
b) Routing protocols using future temporal information, which use information about the expected future status of the wireless links to make routing decisions. Such protocols as FORP [112], RABR [4], and LBR [66] are of this class.

3. Topology information organization
Since the number of nodes in ad hoc networks is generally small, it is possible to use either a flat topology or a hierarchical topology for routing. In the first case, the availability of a globally unique addressing mechanism for nodes in ad hoc wireless networks should be assumed. Such protocols as DSR [52], AODV [91], ABR [120], SSA [32], FORP [112], and PLBR [106] are of this case. Protocols of the second case make use of a logical hierarhy in the network and an associated addressing scheme. CGSR [21], FSR [47] and HSR [47] are of this case.

4. Utilization of specific resources
The protocols of this category can be further classified into two types:

a) **Power-aware routing protocols:** Protocols falling into this class atempt to minimize the baterry power. A typical protocol is PAR [102].
b) **Geographical information assisted routing protocols:** Protocols falling into this class atempt to improve the performance of routing and reduce the control overhead by effictively utilizing the geographical information. A typical protocol is LAR [59].

Several of the protocols addressed above are also described in [74].

1.3. Multicast Routing

Due to the highly dynamic topology, limited bandwidth and other limited resources of ad hoc networks, it is a challenge to adapt existing multicast routing protocols of wired networks, or to develop new protocols for ad hoc networks.

1.3.1. Requirements

A good multicast routing protocol for ad hoc networks should satisfy the following requirements:

- **Robustness:** It should be robust enough to sustain the mobility of the nodes and achieve a high packet delivery ratio.
- **Efficiency:** Multicast efficiency is defined as the ratio of the number of data packets received by the receivers and the total number of packets transmitted in the networks.
- **Control overhead:** In order to manage the nodes in an ad hoc network, it is necessary to exchange the control packets. Due to the limited bandwidth in ad hoc networks, the number of the control packets should be minimal.
- **Quality of service (QoS):** The main parameters for QoS are throughput, delay, delay jitter, and reliability.
- **Independence of the unicast routing protocol:** A multicast routing protocol should be independent of any specific unicast routing protocol.
- **Resource management:** A multicast routing protocol should use minimum power and memory.

1.3.2. Classifications of multicast routing protocols

According to the dependency of the applications we can classify the multicast routing protocols broadly into two types:

1. Application-independent/generic multicast protocols
   The most multicast routing protocols are of this class. They can be further classified along the following dimensions:
   a) Based on topology
      i. **Tree-based:** Only one path exists between a source-receiver pair. Compared to mesh-based protocols, those tree-based are more efficient. Protocols falling into this class are MCEDAR [104], BEMRP [89], MZRP [27], ABAM [121], DDM [50], and WBM [25].
      ii. **Mesh-based:** There may be more than one path between a source-receiver pair, hence protocols of this type are more stable than those tree-based. Protocols falling into this class are AMRoute [65], MAODV [98], and AMRIS [125].
   b) Based on initialization of the multicast session
      i. **Source-initiated:** In protocols of this class, the source initializes the multicast formation. Protocols falling into this class are MZRP [27], ABAM [121], AMRIS [125], ODMRP [63], DCMP [24], and NSMP [62].
      ii. **Receiver-initiated:** In protocols of this class, the receive initializes the multicast formation. Protocols falling into this class are BEMRP [89], DDM [50], WBM [25], PLBM [105], FGMP-RA [20], and NSMP [62].
   c) Based on the topology maitenance mechanism
      i. **Soft state approach:** Protocols falling into this class send control packets periodically to refresh the route. Protocols of this class are MZRP [27], DDM [50], ODMRP [63], DCMP [24], FGMP-RA [20], and NSMP [62].
      ii. **Hard state approach:** Protocols falling into this class send control packets to refresh the route, only when a link breaks. Protocols of this class are BEMRP [89], ABAM [121], WBM [25], PLBM [105], AMRIS [125], and CAMP [37].

2. Application dependent multicast protocols
   Protocols of this class are used for specific application for which they are designed. Only several protocols are developed, such as RBM [15], CBM [129], and LBM [60].

Several of the protocols addressed above are also described in [75].

1.4. Security in Ad Hoc Networks

As discussed at begin of this chapter, due to the issues such as shared physical medium, lack of central management, limited resources, no fixed and highly dynamic topology, ad hoc networks are much more vulnerable to security attacks. Hence it is very necessary to find security solutions, which are much more difficult to develop than in wired networks. As well as in wired networks, the following major security goals should be satisfied. They are confidentiality, integrity, availability, authentication, non-repudiation, which will be further described in Section 3.1.

In the following sections we first address attacks in ad hoc networks, and list several typical special attacks. Then we describe the solutions to ensure ad hoc networks, namely, the solutions to avoid and to minimize the loss resulted by those attacks.

1.4.1. Networks security attacks in ad hoc networks

As well as in wired Networks, we can classify the attacks into two brief categories, namely passive and active attacks. A „passive attack“ attempts to learn or make use of information from the system but does

not affect system resources. A powerful solution to keep the adversary from getting useful information is encrypting the communication. An active attack attempts to alter system resources or affect their operation. Active attacks can be further classified into two types according to the location of attackers, namely internal and external active attacks.

In the following we list at first several attacks in ad hoc networks with brief descriptions, and then discuss the solutions. According to the layer attacked they can be classified into network layer attacks, transport layer attacks, Application layer attacks, and multi-Layer attacks.

1.4.1.1. Network layer attacks

Attacks, which could occur in network layer of the network protocol stack, fall into this class. In following three of those are briefly described.

1. **Wormhole attack:** In this attack, an adversary receives packets at one point in the network, „tunnels“ them to another point in the network, and then replays them into the network from that point [43]. This tunnel between two adversaries are called wormhole. It can be established through a single long range wireless link or a wired link between the two adversaries. Hence it is simple for the adversary to make the tunneled packet arrive sooner than other packets transmitted over a normal multi-hop route.
2. **Black hole attack:** In this attack, a malicious node attempts to suggest false path to the destination. An adversary could prevent the source from finding path to destination, or forward all messages through a certain node [5].
3. **Routing attacks:** In this attack, an adversary attempts to disrupt the operation of the network. The attacks can be further classified into several types, namely routing table overflow attack, routing table poisoning attack, packet replication attack, route cache poisoning, and rushing attack.

   In a routing table overflow attack, an adversary attempts to cause an overflow in routing table by advertises routes to non-existent nodes, while in routing table poisoning attack the adversary sends false routing updates or modifies the actual routing updates to result jam in networks.

Some other attacks in network layer, such as Byzantine attack [10], information disclosure, and resource consumption must also be considered.

1.4.1.2. Transport layer attacks

One of the transport layer attacks is session hijacking. In this type of attack, an adversary obtains the control of a session between two parties. In most cases the authentication process is executed when a session begins, hence an adversary could take the role of one party in the whole session.

1.4.1.3. Application layer attacks

In this type of attack, an adversary analyzes the vulnerability. Dozens of attacks aiming at application layer exist, such as script attack, virus, and worm.

1.4.1.4. Multi-Layer attacks

Attacks, which could occur in any layer of the network protocol stack, fall into this class. In following two of those are briefly described.

1. **Spoofing attack:** Spoofing attacks are also called impersonation attack. The adversary pretends to have the identity of another node in the network, thus receiving messages directed to the node it fakes. One of these attacks is man-in-the-middle attack. In this attack, attackers place their own node between two other nodes communicating with each other and forward the communication.

2. **Denial of service attack:** In this type of attack, the attacker attempts to prevent the authorized users from accessing the services. Due to the disadvantage of ad hoc networks, it is much easier to launch Dos attacks. For example, an adversary could disrupt the on-going transmissions on the wireless channel by employing jamming signals on the physical and MAC layers.

1.4.1.5. Others

Unlike above addressed attacks, in a device tampering attack, devices such as PDA could get stolen or damaged easily. The adversary could then get useful data from the stolen devices and communication on behalf of the owner.

1.4.2. Solution against attacks

Since lot of attacks exist and threat the ad hoc networks, solutions to achieve the security goals must be discovered. Two most important issues are secure routing and cryptography. For cryptography we refer the reader to Section 3.

1.4.2.1. Secure routing

To avoid the above addressed network layer attacks, a secure routing protocol for ad hoc networks should:

- Be able to detect the spiteful nodes in the network and to prevent them from participating in routing process.
- Assure that a correct route could be found, if it exists.
- Guarantee the confidentiality of network topology.
- Be stable against attacks

Some security-aware routing protocols for ad hoc networks have been proposed, such as Security-Aware Ad Hoc Routing Protocol (SAR) [127], Secure Efficient Ad Hoc Distance Vector Routing Protocol (SEAD) [42], and Authenticated Routing for Ad Hoc Networks (ARAN) [100].

2. Mathematical Backgrounds

This chapter discusses a set of mathematical backgrounds required for this thesis. Section 2.1 lists the most notations and definitions that are used. While some mathematical foundations such probability, information, complexity, number, and group theory, are studied in Section 2.2. After that we describe some foundations over elliptic curve in Section 2.3, the problems are discussed in Section 2.4.

2.1. Notations and Definitions

1. $\sum$ denotes the sum of all related elements, e.g. $\sum_{i=1}^{n} a_i$ means the sum $a_1 + a_2 + \cdots + a_n$, and $\sum_{a_i \in S} a_i$ means the sum of all elements belonging to set S.
2. $\log$ denotes the logarithm of base 2, $\ln$ denotes the logarithm of base e.
3. $x := expr$ means that x is defined as $expr$. For example $m^{(n)} := m(m-1)(m-2)\cdots(m-n+1)$.
4. $\{v_1, v_2, \cdots, v_n\}$ denotes the set of elements $v_1, v_2, \cdots, v_n$.
5. $(v_1, v_2, \cdots, v_n)$ denotes the sequence of elements $v_1, v_2, \cdots, v_n$.
6. $\{f(v_1, v_2, \cdots, v_n) \mid \text{pred}(v_1, v_2, \cdots, v_n)\}$ denotes the set of the result of function f of n variables $v_1, v_2, \cdots, v_n$, if they satisfy the predicate $pred$. For example, $\{(p, q, n)|p, q \in \mathbb{N}, p,\ q$ are prime $\}$ denotes the set of all tuples which contain two prime integers and their product.
7. $(f(v_1, v_2, \cdots, v_n) \mid \text{pred}(v_1, v_2, \cdots, v_n))$ denotes the sequence of the result of function f of n variables $v_1, v_2, \cdots, v_n$, if they satisfy the predicate $pred$. For example, $((p, q, n)|p, q \in \mathbb{N}, p, q$ are prime) denotes the infinite sequence of all tuples which contain two prime integers and their product.
8. $\mathbb{Z}$ denotes the set of integers; that is, the set $\{\cdots$, -2, -1, 0, 1, 2, $\cdots\}$.
9. $\mathbb{N}$ denotes the set of natural numbers; that is, the set $\{1, 2, 3, \cdots\}$.
10. $\mathbb{N}_0$ denotes the set of natural numbers and 0; that is, the set $\{0, 1, 2, \cdots\}$.
11. $\mathbb{Q}$ denotes the set of rational numbers; that is, the set $\{\frac{a}{b}|a, b \in \mathbb{Z}, b \neq 0\}$. For example $\frac{5}{7}$, $\frac{111}{997}$ are members of $\mathbb{Q}$
12. $\mathbb{R}$ denotes the set of real numbers.
13. $a \in S$ means that element a is a member of the set S.
14. $A \subseteq B$ means that set A is a subset of set B.
15. $A \subset B$ means that set A is a proper subset of set B, that is, $A \subseteq B$, and $A \neq B$. For example, $\mathbb{N} \subset \mathbb{N}_0 \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R}$.
16. $|A|$ denotes the order of A, namely the number of elements in set A.
17. $A \cup B$ denotes the union of two sets; that is $\{x|x \in A$ or $x \in B\}$.
18. $A \cap B$ denotes the intersection of two sets; that is $\{x|x \in A$ and $x \in B\}$.
19. $A \setminus B$ denotes the difference of two sets; that is $\{x|x \in A$ and $x \notin B\}$.
20. $r \xleftarrow{\mathcal{R}} S$ or $r \in_{\mathcal{R}} S$ means that r is chosen uniformly from set S.
21. $poly(v)$ denotes the class of univariate polynomials with variable v and non-negative coefficients; that is $ploy(v) := \{\sum_{i=0}^{n} a_i v^i | a_i, n \in \mathbb{N}_0\}$.

22. $\epsilon(k)$ denotes a negligible function of parameter k. If $\epsilon(k) <_\infty 1/poly(k) := \epsilon(k) < 1/k^n, \forall n > 0, \exists k_0, \forall k > k_0$, we say it is negligible, else not negligible.
23. A non-negligible function $f(k)$ of parameter k is non-negligible, if $f(k) \geq_\infty 1/poly(k) := f(k) \geq 1/k^n, \exists n > 0, \exists k_0, \forall k > k_0$. Else it is not non-negligible.
24. gcd denotes the greatest common divisor. Given two integers a and b, if and only if
    a) $d \geq 0$, and
    b) $d|a$ and $d|b$, and
    c) $\forall c : c|a$ and $c|b$, follows $c|d$,

    then d is the greatest common divisor of a and b, denoted $d = \gcd(a, b)$.
25. lcm least common multiple. Given two integers a and b, if and only if
    a) $d \geq 0$, and
    b) $a|d$ and $b|d$, and
    c) $\forall c : a|c$ and $b|c$, follows $d|c$,

    then d is the least common multiple of a and b, denoted $d = \text{lcm}(a, b)$.
26. $prime/composite$: Two integers a and b are *relatively prime* if the $\gcd(a, b)$ = 1. For integer $p \geq 2$, p is said to be $prime$ if it has only two positive divisors 1 and itself p, otherwise it is $composite$.
27. $\phi(n)$ denotes Euler phi function of n, it is the number of integers in the interval $[1, n]$ which are relatively prime to positive integer n. It has the following properties:
    a) if n is prime, then $\phi(n) = p - 1$;
    b) if $\gcd(m, n) = 1$, then $\phi(mn) = \phi(m)\phi(n)$;
    c) Given an integer $n \geq 2$, it has a factorization as a product of prime powers.

$$n = p_1^{e_1} p_2^{e_2} \cdots p_l^{e_l} \tag{2.1}$$

    where p_i are pairwise distinct primes, and e_i are non-negative integers, then

$$\phi(n) = n(1 - \frac{1}{p_1})(1 - \frac{1}{p_2}) \cdots (1 - \frac{1}{p_l}) \tag{2.2}$$

28. $a \equiv b \mod n$ means that *a is congruent to b modulo n*, where $n \in \mathbb{N}$, and $n|(a - b)$.
29. $\mathbb{Z}_n$ is the set of integers in interval $[0, n-1]$. Addition, subtraction, multiplication in $\mathbb{Z}_n$ are performed modulo n.
30. $a^{-1} \mod n$ denotes the *multiplicative inverse* of a in $\mathbb{Z}_n$. $x = a^{-1} \mod n$ if $ax \equiv 1 \mod n$. It means that a is invertible if and only if $\gcd(a, n) = 1$.
31. $\mathbb{Z}_n^*$ denotes the *multiplicative group* of $\mathbb{Z}_n$. It is defined as $\mathbb{Z}_n^* := \{a \in \mathbb{Z}_n | \gcd(a, n) = 1\}$. If n is prime, $\mathbb{Z}_n^*$ contains all elements of $\mathbb{Z}_n$ except 0. More formally, $\mathbb{Z}_n^* = \{a | 1 \leq a < n, n\ prime\}$.

    For $n \geq 2$, if $a \in \mathbb{Z}_n^*$, then $a^{\phi(n)} \equiv 1 \mod n$; and if $r \equiv s \mod \phi(n)$, then $a^r \equiv a^s \mod n$. For $\mathbb{Z}_p^*$, where p is a prime, we derive the following properties:
    a) If $a \in \mathbb{Z}_p^*$, then $a^{p-1} \equiv 1 \mod p$, and $a^p \equiv a \mod p$.
    b) If $r \equiv s \mod p - 1$, then $a^r \equiv a^s \mod p$ for all integers a.
32. $Q_n/\overline{Q}_n$: Q_n denotes the *quadratic residue modulo n*. It is defined as $Q_n := \{a | a = x^2 \mod n, x \in \mathbb{Z}_n^*\}$. While $\overline{Q}_n$ denotes the *quadratic non-residue modulo n*. It is defined as $\overline{Q}_n := \mathbb{Z}_n^* - Q_n$.
33. $\left(\frac{a}{n}\right)$ denotes the *Jacobi symbol*, where n is an integer greater than 2.

a) n is prime, p is used here for n.

$$\left(\frac{a}{p}\right) = \begin{cases} 0, & p|a \\ 1, & a \in Q_p \\ -1, & a \in \overline{Q}_p \end{cases} \tag{2.3}$$

b) n is odd with prime factorization of Equation 2.1, then the Jacobi symbol is defined as

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p_1}\right)^{e_1} \left(\frac{a}{p_2}\right)^{e_2} \cdots \left(\frac{a}{p_l}\right)^{e_l}, \tag{2.4}$$

where p_i are pairwise distinct primes and e_i are positive integers.

34. *B-smooth*: Let B be a positive integer. An integer n is said to be *B-smooth*, or *smooth with respect to bound B*, if all its prime factors are $\leq B$.

35. $L_q[\alpha, c]$ is defined as

$$L_q[\alpha, c] = \mathcal{O}(\exp((c + \mathcal{O}(1))(\ln q)^{\alpha}(\ln \ln q)^{1-\alpha})), \tag{2.5}$$

where c is a positive constant, and α is a constant satisfying $0 < \alpha < 1$.

36. $A \leq_P B$ means that *A polytime reduces to B*, where $A,\ B$ are two computational problems.

37. $A \equiv_P B$ means that A and B are *computationally equivalent*, if $A \leq_P B$ and $A \geq_P B$, where A and B are two computational problems.

2.2. Abstract Algebra

This section introduces algebraic groups and fields. The rest of this section gives an overview of the basic properties and arithmetics of polynomials in finite fields.

2.2.1. Groups

A $group$ is a tuple $(G, *)$, where G is a set and $*$ is a binary operation on set G, and fulfills the following three axioms:

1. $*$ is $associative$, which means that $(a * b) * c = a * (b * c),\ \forall\ a, b, c \in G$.
2. An $identity\ element\ 1 \in G$ must exist, which satisfies $a * 1 = 1 * a = a,\ \forall\ a \in G$.
3. $\forall\ a \in G, \exists a^{-1} \in G$, so that $a * a^{-1} = a^{-1} * a = 1$. We call this a^{-1} inverse.

Further if $a * b = b * a$ for all $a, b \in G$, the group G is $abelian$ or $commutative$.

For example, $(\mathbb{Z}_n, +)$, where + is addition modulo n. The identity element is zero, and for each $a \in G$, the inverse a^{-1} is $n - a$. But $(\mathbb{Z}_n, \times)$, where $\times$ is multiplication modulo n, is not a group, since the element zero has no inverse. However, the multiplicative group of $\mathbb{Z}_n$, namely $\mathbb{Z}_n^*$, with the operation modulo n is a group of order $\phi(n)$, with the identity element 1.

In the following we list some important definitions and properties relevant to groups.

1. **Finite:** If $|G|$ is finite, then group $(G, *)$ is finite. The order of the group is the number of elements in set G, namely $|G|$.
2. **Subgroup**: If $H \subseteq G, H \neq \emptyset$, and $(H, *)$ is a group, then group $(H, *)$ is called $subgroup$ of $group(G, *)$. If $H \neq G$, then it is a proper subgroup.

3. **Cyclic**: A group $(G, *)$ is cyclic if there is an element $\alpha \in G$ such that for each $\beta \in G$ there is an integer i with $\beta = \alpha^i$. Such an element is called a $generator$ of G.

   $g \in \mathbb{Z}_n^*$ is a *generator* if $ord(g) = \phi(n)$. A generator in $\mathbb{Z}_n^*$ exists, if and only if $n = 2, 4, p^k$ or $2p^k$, where p is a prime and $k \geq 1$. For $k = 1$ we know that $\mathbb{Z}_p^*$ has a generator. $g \in \mathbb{Z}_n^*$ is a generator of $\mathbb{Z}_n^*$ if and only if $g^{\phi(n)/p_i} \not\equiv 1 \mod n$ for all prime divisors p_i of n.
4. **Order**: For group $(G, *)$, let $a \in G$. The order of a, denoted $ord(a)$, is defined to be the least positive integer t such that $a^t = 1$, if such t exists, otherwise we say that the order is ∞.
5. $\langle a \rangle$: If G is a group and $a \in G$, then the set of all powers of a forms, a cyclic subgroup of $(G, *)$, denoted by $\langle a \rangle$.
6. **Lanrane's theorem**: If $(H, *)$ is a subgroup of a finite group $(G, *)$, then $|H|$ divides $|G|$.

2.2.2. Rings

A ring is a tuple $(R, +, \times)$, where R is a set, + is the addition, $\times$ is the multiplication on R, and satisfies the following four axioms:

1. $(R, +)$ is an $abelian$ group and its identity element is 0.
2. $\times$ is $associative$, that means that $(a \times b) \times c = a \times (b \times c),\ \forall a, b, c \in R$.
3. A multiplicative identity denoted 1 exists, which satisfies $1 \neq 0$, and $1 \times a = a \times 1 = a,\ \forall\, a \in R$.
4. $\times$ is $distributive$ over +, which means, $a \times (b+c) = (a \times b) + (a \times c)$, and $(a+b) \times c = (a \times c) + (b \times c)$.

The ring is communicative if $a \times b = b \times a$.

For example, $\mathbb{Z}$ under the usual addition and mulplication, and $\mathbb{Z}_n$ under the addition and multiplication modulo n are commutative rings.

2.2.3. Fields

A $field$ is a commutative ring in which all non-zero elements have multiplicative inverses. Since in $\mathbb{Z}_p$, where p is a prime integer, each element except 0 has a multiplicative inverse, and $(\mathbb{Z}_p, +, \times)$ is a commutative ring, it is a field. An analogous definition of fields to $order$ of groups and rings is the $characteristic$. If there exists a positive integer t such that $\sum_{i=1}^{t} 1 = 0$, then the least t is said to be the $characteristic$ of a field, otherwise we say the $characteristic$ is 0. The characteristic of the above mentioned ring is p.

An example of field is the set of integers $\mathbb{Z}_n$ under the addition and multiplication modulo n.

2.2.4. Finite fields

A $finite\ field$ is a field $(F, +, \times)$ that contains a finite number of elements. Its $order$ is defined to be the number of the elements of field $(F, +, \times)$. Finite fields should satisfy the following requirements:

1. A finite field $(F, +, \times)$ contains p^m elements for some prime integer p and positive integer m.
2. A unique (up to isomorphism) finite field of order $s = p^m$ exists for every p^m, where p is prime and m is positive integer. This field is denoted $\mathbb{F}_{p^m}$ or $GF(p^m)$.
3. For a finite field $(\mathbb{F}_q, +\times)$, where $q = p^m$ and p prime, its $characteristic$ is p. Furthermore, $(\mathbb{F}_q, +, \times)$ contains a copy of $(\mathbb{F}_p, +, \times)$ as the subfield.
4. *Multiplicative group of* $\mathbb{F}_q$, denoted $\mathbb{F}_q^*$, is a group of all non-zero elements from $\mathbb{F}_q$ under the multiplication. $\mathbb{F}_q^*$ is cyclic with the order $q - 1$.
5. Let $(\mathbb{F}_q, +, \times)$ be a finite field of order $q = p^n$, and $(\mathbb{F}_s, +, \times)$ of order p^m be its subfield, then $m|n$ and $m > 0$. Adversely if the above condition of m and n are fulfilled, there exists a unique subfield of $(\mathbb{F}_q, +, \times)$ of order p^m.

2.2.5. Polynomial rings

Before we specify polynomial rings, we define a $polynomial$. A $polynomial$ of x over a commutative ring $(R, +, \times)$ is:

$$f(x) = a_n x^n + \cdots + a_2 x^2 + a_1 x + a_0,\ a_i \in R \text{ and } n \geq 0. \tag{2.6}$$

a_i is called *coefficient* of x^i. The largest positive integer m such that $a_m \neq 0$ is the degree of the polynomial (denoted $degr(f(x))$), and a_m is called the leading coefficient. If $a_m = 1$, we say that the polynomial is $monic$. If all coefficients a_i with $i \geq 1$ are zero, the polynomial is constant.

If R is a commutative ring, then the polynomial ring is the ring formed by the set of all polynomials in the indeterminate x having coefficients from R, denoted by $R[x]$.

A polynomial ring is a ring which is defined over a polynomial. It is a tuple $(R[x], +, \times)$, where the two operations + and $\times$ are the standard polynomial addition and multiplication with coefficient arithmetic in R. All other issues are the same as in the definition of ring.

The following example of polynomial ring $(\mathbb{Z}_3, +, \times)$ has identity element $f(x) = 0$, and multiplicative identity element $f(x) = 1$. Let $g(x) = x^5 + 2x + 1$, and $h(x) = x^3 + 2x^2 + x$, then

$$\begin{aligned} g(x) + h(x) &= x^5 + x^3 + 2x^2 + 3x + 1 \\ &= x^5 + x^3 + 2x^2 + 1 \end{aligned}$$

$$\begin{aligned} g(x) \times h(x) &= x^8 + 2x^7 + x^6 + 2x^4 + 5x^3 + 4x^2 + x \\ &= x^8 + 2x^7 + x^6 + 2x^4 + 2x^3 + x^2 + x \end{aligned}$$

Let $f(x)$ with positive degree be from $F[x]$, if it cannot be expressed as the product of two polynomials with positive degree from $F[x]$, then $f(x)$ is called *irreducible over* $F[x]$.

What is the operation division for polynomials? Let $g(x), h(x) \in F[x]$, with $h(x) \neq 0$, then the division of polynomial $g(x)$ by $h(x)$ can be written in form

$$g(x) = q(x)h(x) + r(x), \quad q(x),\ r(x) \in F[x],\ degr(r(x)) < degr(h(x)) \tag{2.7}$$

We call $q(x)$ quotient and $r(x)$ remainder. Furthermore we can write

$$g(x) \text{ div } h(x) = q(x) \tag{2.8}$$

$$g(x) \mod h(x) = r(x). \tag{2.9}$$

We take two polynomials from $\mathbb{Z}_2[x]$, $g(x) = x^5 + x + 1$, and $h(x) = x^3 + x + 1$. The division of $g(x)$ by $h(x)$ provides

$$g(x) = (x^2 + 1)h(x) + x^2.$$

So we get $g(x)\ div\ h(x) = x^2 + 1$, and $g(x)\ mod\ h(x) = x^2$.

A few important properties and definitions are listed below.

1. Let $g(x),\ h(x) \in F[x]$, if $g(x) \mod h(x) = 0$, we say that $h(x)$ divides $g(x)$, denoted $h(x)|g(x)$.
2. Let $g(x),\ h(x),\ f(x) \in F[x]$, if $f(x)|(g(x) - h(x))$, we say that $g(x)$ is congruent to $h(x)$ modulo $f(x)$, denoted $g(x) \equiv h(x) \mod f(x)$.
3. Let $f(x) \in F[x]$, $F[x]/f(x)$ denote the set of all polynomials with degree less than that of $f(x)$. The operation are performed modulo $f(x)$. In fact, $F[x]/f(x)$ is a commutative polynomial ring. And if $f(x)$ is irreducible, then $F[x]/f(x)$ is a field.
4. Let $g(x),\ h(x) \in \mathbb{Z}_p[x]$, where p is prime, and not both polynomials are 0. $\gcd(g(x), h(x))$ denotes the greatest common divisor of $g(x)$ and $h(x)$. It is the monic polynomial of greatest degree in $\mathbb{Z}_p[x]$. By definition, $\gcd(0, 0) = 0$.

   For given $g(x),\ h(x) \in \mathbb{Z}_p[x]$, the *Euclidean algorithm for* $\mathbb{Z}_p[x]$ (algorithm 2.218 in [70]) can be used to compute $\gcd(g(x),\ h(x))$, with a running time of $\mathcal{O}(m^2)\ \mathbb{Z}_p - operations$ or $\mathcal{O}(m^2(lgp)^2)$

bit-operations, where m is the greater degree of $g(x)$ and $h(x)$. The *extended Euclidean algorithm for* $\mathbb{Z}_p[x]$ (algorithm 2.221 in [70]) yields $s(x)$ and $t(x) \in \mathbb{Z}_p[x]$, besides the greatest common divisor, such that

$$s(x)g(x) + t(x)h(x) = \gcd(g(x), h(x)). \tag{2.10}$$

Although it provides more result, it has the same runtime as Euclidean algorithm.

5. If $f(x) \in \mathbb{Z}_p[x]$ is not zero, then it can be expressed to be

$$f(x) = af_1(x)^{e_1} f_2(x)^{e_2} \cdots f_l(x)^{e_l}, e_i \in \mathbb{N},\ a \in \mathbb{Z}_p \tag{2.11}$$

and f_i are distinct monic irreducible polynomials in $\mathbb{Z}_p[x]$.

2.2.6. Arithmetic of $\mathbb{F}_{p^m}$

We describe in the following in the arithmetic of a finite field $\mathbb{F}_{p^m}$, where p is prime. If $m = 1$, then $\mathbb{F}_{p^m}$ is just $\mathbb{Z}_p$ and the arithmetic is performed modulo p. For each $m \geq 1$, there exists a monic irreducible polynomial of degree m over $\mathbb{Z}_p$, every finite field has a polynomial basis representation. Some efficient algorithms to find irreducible polynomials over finite fields are introduced in section 4.5.1 of [70]. Hence the elements of the finite field $\mathbb{F}_{p^m}$ can be represented by polynomials in $\mathbb{Z}_p[x]$ of degree $< m$.

$$\mathbb{F}_{p^m} = \{a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \cdots + a_1x + a_0 \mid a_i \in \mathbb{Z}_p\} \tag{2.12}$$

For convenience, the polynomial $a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \cdots + a_1x^1 + a_0$ is represented by the vector $(a_{m-1}a_{m-2} \cdots a_1a_0)$ of length m. Hence

$$\mathbb{F}_{p^m} = \{(a_{m-1}a_{m-2} \cdots a_1a_0) \mid a_i \in \mathbb{Z}_p\} \tag{2.13}$$

Now let $f(x)$ be an irreducible polynomial from $\mathbb{Z}_p[x]$ with degree m, and let $g(x),\ h(x)$ be polynomials from $\mathbb{F}_{p^m}$ with degree less than m, then

1. The $addition$ $g(x) + h(x)$ is a usual addition of polynomials in $\mathbb{Z}_p[x]$.
2. The $product$ $g(x)h(x)$ can be computed as follows, first we multiply $g(x)$ and $h(x)$ as polynomial by the usual method, and then compute polynomial modulo $f(x)$.
3. The $multiplicative\ inverses$ can be computed by the *extended Euclidean algorithm for* $\mathbb{Z}_p[x]$. To compute the multiplicative inverse of $g(x)$, namely $g^{-1}(x)$, we get first $s(x)g(x) + t(x)f(x) = \gcd(g(x), f(x))$. Since $f(x)$ is irreducible, $\gcd(g(x), f(x)) = 1$. Now we get $g^{-1}(x) = s(x)$.

An example of $\mathbb{F}_{2^4}$ is shown in following. Given an irreducible polynomial $f(x) = x^4 + x + 1 \in \mathbb{Z}_2[x]$. Hence $\mathbb{F}_{2^4}$ is the set of polynomials over $\mathbb{F}_2$ of degree less than 4. $\mathbb{F}_{2^4} = \{a_3x^3 + a_2x^2 + a_1x + a_0 \mid a_i \in \{0, 1\}\}$ or $\{(a_3a_2a_1a_0) \mid a_i \in \{0, 1\}\}$.

1. $Addition$: (1101) + (1011) = (0110).
2. $Multiplication$: $(1101)(1011) = (x^3 + x^2 + 1)(x^3 + x + 1)$
   $= x^6 + x^5 + x^4 + x^3 + x^2 + x + 1 \equiv x^2 + x \mod f(x)$.
3. $Multiplicative\ inverse$: The inverse of (0101) is (1011). This can be verified by: $(x^2 + 1)(x^3 + x + 1) = x^5 + x^2 + x + 1 \equiv 1 \mod f(x)$.

2.3. Elliptic Curve Groups

Elliptic curves have been studied extensively for the past 150 years, and Elliptic curve systems as applied to cryptography were first proposed in 1985 independently by Neal Koblitz [61] and Victor Miller [71]. A background understanding of abstract algebra is required, much of which can be found in Section 2.2.

This section is organized as follows: Section 2.3.1 explains elliptic curves over real numbers in order to illustrate the geometrical properties of elliptic curve groups, while sections 2.3.2 and 2.3.3 discuss the elliptic curves groups over the underlying fields of $\mathbb{F}_p$ and $\mathbb{F}_{2^m}$.

2.3.1. Elliptic curve groups over $\mathbb{R}$

An elliptic curve over $\mathbb{R}$ may be defined to be as the set of all points (x, y) which fulfill an elliptic curve equation of the form:

$$y^2 = x^3 + ax + b, \quad x, y, a, b \in \mathbb{R}. \tag{2.14}$$

If $4a^3 + 27b^2 \neq 0$, then the elliptic curve of form 2.14 can be used to form a group. An elliptic curve group over real numbers consists of the points on the corresponding elliptic curve, together with the *point at infinity*, denoted O (definition in Equation 2.18). All points except O can be denoted $(x,\ y)$, where x, y satisfy the Equation 2.14. The *negative of a point* $P(x_P, y_P)$ is defined to be its reflection in the x-axis: $-P = (x_P, -y_P)$. It is to verify, if P is on the curve, then -P must be on the curve. An example of the graph of elliptic curve $y^2 = x^3 - 4x + 0.67$ ($here\ 4a^3 + 27b^2 = 4 \cdot (-4)^3 + 27 \cdot 0.67^2 \neq 0$) is shown in Figure 2.1.

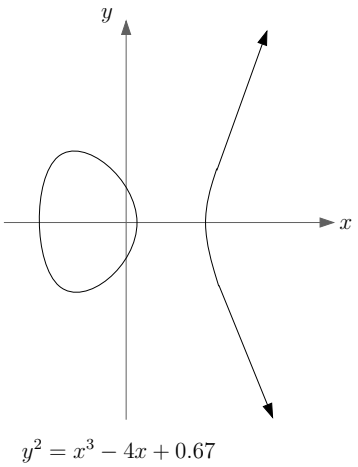


Figure 2.1.: Elliptic curve $y^2 = x^3 - 4x + 0.67$

In the following we discuss the addition of two distinct points, doubling of one point on an elliptic curve. Since the operation can be performed when the points are on the same elliptic curve, for convenience, we omit in the following this condition.

2.3.1.1. Adding two distinct points

The addition of two distinct points $P(x_P,\ y_P)$ and $Q(x_Q,\ y_Q)$ can be further classified into two types.

1. ***-P*** $\neq$ ***Q***: To add P and Q, a line is drawn through both points. This line will intersect the elliptic curve in exactly one more point, denoted -R. The point -R is reflected in the x-axis to the point R. The point R is the addition of P and Q, namely $R = P + Q$.

   Figure 2.2 gives an example of addition over elliptic curve $y^2 = x^3 + 7x$. Given two points $P(-2.35, -1.86)$ and $Q(-0.1, 0.836)$, according to the above described method, we get $R(3.89, -5.62) = P + Q$.

   Beside the above introduced geometric method, the following rules can be applied to compute the addition of two distinct points: $R(x_R,\ y_R) = P(x_P,\ y_P) + Q(x_Q,\ y_Q)$, where $-P \neq Q$.

$$s = \frac{y_P - y_Q}{x_P - x_Q} \tag{2.15}$$

$$x_R = s^2 - x_P - x_Q \tag{2.16}$$

$$y_R = -y_P + s(x_P - x_R) \tag{2.17}$$

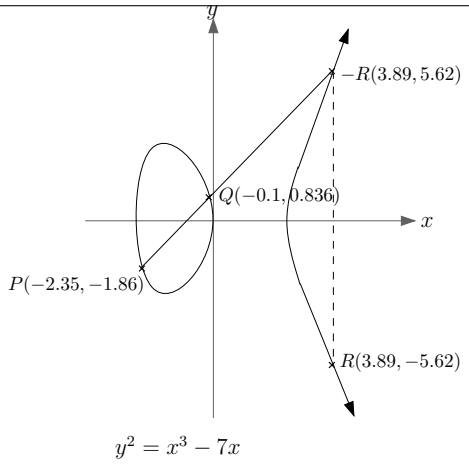


Figure 2.2.: Adding two distinct points P and Q with $-P \neq Q$ on elliptic curve $y^2 = x^3 - 7x$

2. ***-P = Q***: In this case, the line through P and Q is a vertical line, it does not intersect the elliptic curve at a third point. The *point at infinity O* is defined to be the result of addition of P and $-P$.

$$O = P + (-P) \tag{2.18}$$

$P + O = P$ can be derived from Equation 2.18. Hence O is the additive identity of the elliptic curve group. Figure 2.3 illustrates the addition of two points P and $-P$ on elliptic curve $x^3 + 6x + 6$.

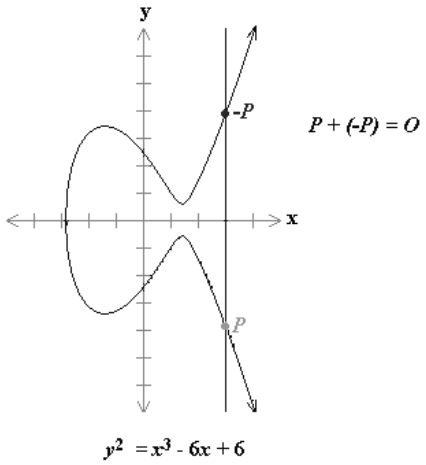


Figure 2.3.: Adding two distinct points P and Q with $-P = Q$ on elliptic curve $y^2 = x^3 - 6x + 6$

2.3.1.2. Doubling a point

The doubling of a point $P(x_P, y_P)$ can be further classified into two types.

1. $-y_P \neq \mathbf{0}$: A tangent line to the curve is drawn at the point P. Since $y_P \neq 0$, this line intersects the elliptic curve at exactly one other point $-R$. $-R$ is reflected in the x-axis to R, which is defined to be the doubling of P: $R = P + P = 2P$. An example of doubling the point $P(2, 2.65)$ on elliptic curve $y^2 = x^3 - 3x + 5$ is shown in Figure 2.4.

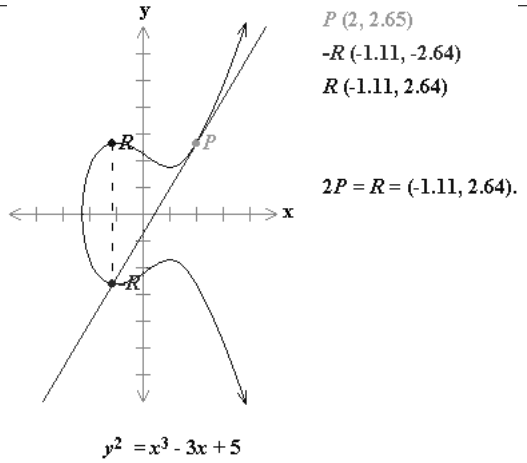


Figure 2.4.: Doubling the point P with $y_{_P} \neq 0$ on elliptic curve $y^2 = x^3 - 3x + 5$

Beside the above introduced geometric method, the following rules can be applied to compute the doubling of a point P: $R(x_{_R},\ y_{_R}) = 2P(x_{_P},\ y_{_P})$, where $y_{_P} \neq 0$.

$$s = \frac{3x_{_P}^2 + a}{2y_{_P}} \tag{2.19}$$

$$x_{_R} = s^2 - 2x_{_P} \tag{2.20}$$

$$y_{_R} = -y_{_P} + s(x_{_P} - x_{_R}) \tag{2.21}$$

2. $-y_{_P}$ = **0**: A tangent line is drawn as in the first case. Since $y_{_P} = 0$, this line intersect does not intersect the elliptic curve at any other point. According to the definition of O in Equation 2.18 and the fact P = $-P$, we get $2P = P + (-P) = O$.

2.3.2. Elliptic curve groups over $\mathbb{F}_p$

Because of round-off errors, computations over the real numbers are slow and imprecise, whereas cryptographic applications require fast and precise arithmetic. Hence elliptic curve groups over the finite fields such as $\mathbb{F}_p$ and $\mathbb{F}_{2^m}$ are used in practice, instead of over $\mathbb{R}$.

An elliptic curve over $\mathbb{F}_p$ may be defined to be as a set of all points (x, y) which fulfill an elliptic curve equation of the form:

$$y^2 = x^3 + ax + b, \quad x, y, a, b \in \mathbb{Z}_{_P},\ p\,prime. \tag{2.22}$$

If $4a^3 + 27b^2 \mod p \neq 0$, the elliptic curve group over $\mathbb{F}_p$ is defined to be set of all points which satisfy Equation 2.22 and the *point at infinity O*. Every point can be negated. Given a point $P(x_{_P},\ y_{_P})$ on the elliptic curve 2.22, the negative point of P is defined as follow:

$$-P = (x_{_P},\ -y_{_P} \mod p) \tag{2.23}$$

There are several major differences between elliptic curve groups over $\mathbb{F}_p$ and over $\mathbb{R}$. Elliptic curve groups over $\mathbb{F}_p$ have a finite number of points, which is a desirable property for cryptographic purposes. Thus the geometry used in elliptic curve groups over $\mathbb{R}$ cannot be used for elliptic curve groups over $\mathbb{F}_p$. However, the algebraic rules for the arithmetic can be adapted for elliptic curves over $\mathbb{F}_p$.

The addition of two distinct points $P(x_{_P},\ y_{_P})$ and $Q(x_{_Q},\ y_{_Q})$ with $-P = Q$, and the doubling of point $P(x_{_P},\ 0)$, result the *point at infinity O*. The algebraic rules for other cases are given below.

1. Addition of two distinct points $P(x_P,\ y_P)$ and $Q(x_Q,\ y_Q)$, where $-P \neq Q$. $R(x_R,\ y_R) = P + Q$.

$$s = \frac{y_P - y_Q}{x_P - x_Q} \mod p \tag{2.24}$$

$$x_R = s^2 - x_P - x_Q \mod p \tag{2.25}$$

$$y_R = -y_P + s(x_P - x_R) \mod p \tag{2.26}$$

2. Doubling of point $P(x_P,\ y_P)$ with $y_P \neq 0$. $R(x_R,\ y_R) = 2P$.

$$s = \frac{3x_P^2 + a}{2y_P} \mod p \tag{2.27}$$

$$x_R = s^2 - 2x_P \mod p \tag{2.28}$$

$$y_R = -y_P + s(x_P - x_R) \mod p \tag{2.29}$$

2.3.3. Elliptic curve groups over $\mathbb{F}_{2^m}$

Elements of the field $\mathbb{F}_{2^m}$ are m-bit strings. The rules for arithmetic in $\mathbb{F}_{2^m}$ can be defined by either polynomial representation or by optimal normal basis representation. Since $\mathbb{F}_{2^m}$ operates on bit strings, computers can perform arithmetic in this field very efficiently.

An elliptic curve with the underlying field $\mathbb{F}_{2^m}$ is formed by choosing the elements a and b within $\mathbb{F}_{2^m}$ (the only condition is that b is not 0). As a result of the field $\mathbb{F}_{2^m}$ having a characteristic 2, the elliptic curve equation is slightly adjusted for binary representation

$$y^2 + xy = x^3 + ax^2 + b, \text{ where } a, b, x, y \in \mathbb{F}_{2^m} \tag{2.30}$$

The elliptic curve includes all points (x, y) which satisfy Equation 2.30. It could be used to form an elliptic curve group over $\mathbb{F}_{2^m}$, which consists of the points on the corresponding elliptic curve, together with the *point at infinity O*.

There are finitely many points on such an elliptic curve, thus there are no round off errors, the same as in elliptic curves groups over $\mathbb{F}_p$. Since the elliptic curve equation is slightly different from Equations 2.14 and 2.22, the rules here applied are slightly different.

Every point can be negated. Given a point $P(x_P,\ y_P)$ on the elliptic curve 2.30, the negative point of P is defined as follows:

$$-P = (x_P,\ x_P + y_P) \tag{2.31}$$

where + is addition in $\mathbb{F}_{2^m}$. As with elliptic curves over $\mathbb{R}$ and $\mathbb{F}_p$, for the point P, $P + (-P) = O$, and if $y_P = 0$, $2P = O$. The rules for other cases are listed below.

1. Addition of two distinct points $P(x_P,\ y_P)$ and $Q(x_Q,\ y_Q)$, where $-P \neq Q$. $R(x_R,\ y_R) = P + Q$.

$$s = \frac{(y_P - y_Q)}{(x_P + x_Q)} \tag{2.32}$$

$$x_R = s^2 + s + x_P + x_Q + a \tag{2.33}$$

$$y_R = s(x_P + x_R) + x_R + y_P \tag{2.34}$$

2. Doubling of point $P(x_P,\ y_P)$ with $y_P \neq 0$. $R(x_R,\ y_R) = 2P$.

$$s = x_P + \frac{y_P}{x_P} \tag{2.35}$$

$$x_R = s^2 + s + a \tag{2.36}$$

$$y_R = x_P^2 + (s + 1)x_R \tag{2.37}$$

2.3.4. Computational Cost

We have discussed the operations addition and doubling in elliptic curves. Now the computational costs of both operations under affine coordinate and projective coordinate are summarized in Tables 2.1 and 2.2.

	Affine coordinate		Projective coordinate
Operation	Inverse	Multiplication	Multiplication
Addition	1	3	16
Doubling (arbitrary a)	1	4	10
Doubling ($a = -3$)	1	4	8

Table 2.1.: Computational cost of addition and doubling over $\mathbb{F}_p$

	Affine coordinate			Projective coordinate	
Operation	Inverse	Multiplication	Square	Multiplication	Square
Addition ($a \neq 0$)	1	2	1	15	5
Addition ($a = 0$)	1	2	1	14	4
Doubling	1	2	1	5	5

Table 2.2.: Computational cost of addition and doubling over $\mathbb{F}_{2^m}$

2.4. Number-Theoretic Reference Problems

If it is difficult or impossible for an adversary to attack a cryptosystem successfully, then we say that the cryptosystem is secure. But what means here *difficult* or $impossible$? To answer those questions we need the classification of security (in Section 2.4.2), while the last sections address some specific computational problems. To understand these above, some definitions of probability, and $\mathcal{P}$ and $\mathcal{P}$ problem are described briefly in Section 2.4.1.

2.4.1. Probability, $\mathcal{P}$ and $\mathcal{NP}$ problem

This section gives first some definitions of probability. After that the Turing machine will be briefly described. It will be used to define the $\mathcal{P}$ and $\mathcal{NP}$ problems, which are discussed in the following.

2.4.1.1. Probability

1. **Experiment:** An experiment is a procedure that returns possible results. We call the separate possible results *simple events*, and the set of all possible results *sample space*. Since discrete sample spaces are sample spaces with only finitely many possible outcomes, we can label all the simple events with $s_1, s_2, \cdots, s_n$, where n is the number of all events. In the following parts if we does not note, we consider only the probability related to discrete sample spaces.
2. **Probability distribution:** A probability distribution P on S is a sequence of numbers $p_1, p_2, \cdots, p_n$, where p_i represents the probability of s_i. They are all non-negative and should fulfill

$$\sum_{i=1}^{n} p_i = 1 \tag{2.38}$$

3. **Event:** An event E is a subset of the sample space S. We denote the probability, which event E occurs, with $P(E)$.

$$P(E) = \sum_{i \in E} P(\{s_i\}) \tag{2.39}$$

where $P(\{s_i\})$ is the probability of event s_i, it can also be simply denoted as $P(s_i)$.

4. **Complementary event:** A complementary of E is defined as the subset of all simple events in sample space S, which do not belong to E. It is denoted as $\overline{E}$. $E \cup \overline{E} = S, E \cap \overline{E} = \emptyset$.

5. **Conditional probability:** Let E_1 and E_2 be two events with $P(E_2) > 0$. We call the probability of event E_1 occurring, given that E_2 has occurred, the conditional probability of E_1 given E_2, denoted $P(E_1|E_2)$, is:

$$P(E_1|E_2) = \frac{P(E_1 \cap E_2)}{P(E_2)} \tag{2.40}$$

According to Bayes' theorem, it can be computed as follows:

$$P(E_1|E_2) = \frac{P(E_1)P(E_2|E_1)}{P(E_2)} \tag{2.41}$$

2.4.1.2. Turing machine

The Turing machine is an abstract machine introduced in 1936 by Alan Turing to give a mathematically precise definition of algorithm or „mechanical procedure“. As such it is still widely used in theoretical computer science, especially in complexity theory and the theory of computation. A Turing machine consists of a tape, a head, a state register, and an action table.

According to the number of used tapes Turing machine is classified into two classes, namely one-tape and k-tape Turing machine. We define now formally Turing machine. A Turing machine is a 7-tuple

$$M = (Q, \Gamma, \sum, s, b, F, \delta),$$

where

- Q is a finite set of states
- Γ is a finite set of the tape alphabet
- $\sum$ is a finite set of the input alphabet ($\Sigma \subseteq \Gamma$)
- $s \in Q$ is the initial state
- b is the blank symbol ($b \in \Gamma \setminus \Sigma$)
- $F \subseteq Q$ is the set of final or accepting states
- $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ is the transition function for one-tape Turing machine, while

$$\delta : Q \times \Gamma^k \rightarrow Q \times (\Gamma \times \{L, R, S\})^k$$

is for k-tape Turing machine, where L is left shift, R is right shift, and S is no shift.

A Turing machine is *deterministic* if the action table has at most one entry for each combination of symbol and state. If it contains multiple entries the machines is *non-deterministic*.

2.4.1.3. Class $\mathcal{P}$ and $\mathcal{NP}$

The complexity class $\mathcal{P}$ is the set of all decision problems that are solvable in polynomial time. More precisely it is defined in [39] as follows.

A language L is recognizable in (deterministic) polynomial time if there exists a deterministic Turing machine M and a polynomial $p(\cdot)$ such that

- on input a string x, machine M stops after at most $p(|x|)$ steps, and
- $M(x) = 1$ if and only if $x \in L$.

The complexity class $\mathcal{NP}$ is the set of all decision problems for which a YES answer can be verified in polynomial time given some extra information. More precisely it is defined in [39] as follows.

A language L is in $\mathcal{P}$ if there exists a Boolean relation $R_L \subseteq \{0,1\}^* \times \{0,1\}^*$ and a polynomial $p(\cdot)$ such that R_L can be recognized in (deterministic) polynomial time, and $x \in L$ if and only if there exists a y such that $|y| \leq p(|x|)$ and $(x, y) \in L$. Such a y is called a witness for membership of $x \in L$.

2.4.2. Computational security

Security can be generally classified into two types.

1. **Information-theoretic** or **unconditional security**: The security is independent of the time and space resources of the adversary.

   Given a cryptosystem $(\mathcal{P}, \mathcal{C}, \mathcal{K},\ Enc,\ Dec)$, where $\mathcal{P}$ is the set of all plaintexts, $\mathcal{C}$ is the set of ciphertexts, $\mathcal{K}$ is the set of all keys, Enc is encryption algorithm, and Dec is the decryption algorithm. We define further E to be the event, that c is the ciphertext of a given plaintext p encrypted with a key k.

   $$E := \{(c, p, k) | Enc_k(p) = c, p \in \mathcal{P}, k \in \mathcal{K}, c \in \mathcal{C}\} \tag{2.42}$$

   The cryptosystem is *perfect secure*, if

   $$P(p|E) = P(p),\ \forall p \in \mathcal{P}, \forall c \in \mathcal{C} \tag{2.43}$$

   That is, the probability of plaintext is independent of the ciphertext. One time pad falls in this type. But it requires that the key is longer than the plaintext. The most symmetric key cryptosystems and all public-key cryptosystems do not fulfill this condition, hence they are not information-theoretic secure.

2. **Computational** or **cryptographic security**: This security usually relies on either bounds of the attacker's resources, i.e., it is secure only against all attackers which do not have more than these resources, or on some number-theoretic reference problems, which are discussed in the remaining sections.

In a cryptographic setting, an adversary is usually assumed to be very powerful. Hence if a computational problem can be solved in polynomial time, at least for a non-negligible fraction of all possible inputs, then we say this problem is *tractable*. As a result all algorithms based on that problem are said to be insecure or not computationally secure.

The most known and studied problems are the *integer factorization, RSA, quadratic residuosity, square root, discrete logarithm, generalized discrete logarithm, Diffie-Hellman, generalized Diffie-Hellman, and subset sum* problems. Some problems can be further classified into types *Computational* and *Decisional*. Since their true computational complexities are unknown up to the present, they are treated to be *intractable*, although nobody can prove this.

In the following a few concrete problems will be introduced.

2.4.3. The integer factorization problem

According to Equation 2.1, an integer n greater than 1 can be formed with $n = p_1^{e_1} p_2^{e_2} \cdots p_l^{e_l}$. The *integer factorization problem*(FACTORING) is defined to find the prime factorization for a given n.

Some factoring algorithms in literature are studied to perform the factoring better. The details of algorithms can be found on pages 90-98 of [70]. To give an overview their complexity is listed in Table 2.3.

Algorithm	Complexity	Remark
trail division (§ 3.2.1 in [70])	$\mathcal{O}(p_2 + \log n)$	p_2 is second largest prime factor of n
Pollard's rho factoring (§ 3.2.2 in [70])	$\mathcal{O}(\sqrt{n})$	$\mathcal{O}(\sqrt{n})$ memory
Pollard's $p-1$ factoring (§ 3.2.3 in [70])	$\mathcal{O}(B \ln n / \ln B)$	$p-1$ is B-smooth
quadratic sieve factoring (§ 3.2.6 in [70])	$L_n[\frac{1}{2}, 1]$	
general number field sieve factoring (§ 3.2.7 in [70])	$L_n[\frac{1}{3}, c]$	$c = (64/9)^{1/3} \approx 1.92$

Table 2.3.: Complexity of factoring algorithms of n

2.4.4. The RSA problem

Given $n > 0$ which is a product of two distinct primes, $c,\ e \in \mathbb{Z}_n \setminus \{0\}$ and $\gcd\{e, \phi(n)\} = 1$, find $m \in \mathbb{Z}_n \setminus \{0\}$ such that $m^e \equiv c \mod n$. Namely, the RSA problem solves the e-th roots modulo n.

The RSA problem polytime reduces to the *integer factorization* problem, denoted *RSAP* $\leq_P$ *FACTORING*. In other words, if the factorization of n is unknown, then there is no efficient procedure known for solving the RSA problem.

2.4.5. The square root problem

Let J_n be the set of all elements of $\mathbb{Z}_n^*$ with Jacobi symbol 1, $J_n := \{a | \left(\frac{a}{n}\right) = 1, a \in \mathbb{Z}_n^*\}$. Since the set of quadratic residues modulo n is subset of J_n, all elements in J_n but not in Q_n are said to be pseudosquares modulo n: $\tilde{Q_n} = J_n - Q_n$.

The *square root problem* is shown below: given an positive odd integer n and $a \in J_n$, decide whether or not a is a quadratic residue modulo n.

If n is a product of two distinct primes, then Q_n and $\tilde{Q_n}$ have the same order. Thus one get a correct guess with the probability $\frac{1}{2}$. But if n is prime, then is is easy to get the correct guess. In fact the *square root problem* polytime reduces to the *FACTORING* problem, denoted *SQROOT* $\leq_P$ *FACTORING*.

2.4.6. The square root modulo n problem

This problem is shown in the following: let n be a composite integer and $a \in Q_n$, find a square root x of $a \mod n$, namely $x^2 \equiv a \mod n$. We discuss in the following only the case that n is a product of two distinct primes, i.e. $n = pq$.

In fact the *FACTORING* problem and the *SQROOT* problem are computationally equivalent, denoted *FACTORING* $\equiv_P$ *SQROOT*. This assumption can be verified. Assumed that a polytime algorithm A can solve *SQROOT* problem, it does the following to solve the *FACTORING* problem.

1. choose $x \overset{\mathcal{R}}{\leftarrow} \mathbb{Z}_n^*$ such that $\gcd(x, n) = 1$,
2. compute $a = x^2 \mod n$,
3. compute y such that $a = y^2 \mod n$,
4. if $y \equiv \pm x \mod n$, repeat from step 1, otherwise $\gcd(x - y, n)$ is a non-trivial factor of n, namely p or q. Since a has four square roots modulo n ($\pm x$ and $\pm z$ with $\pm x \not\equiv z \mod n$), the success probability is $\frac{1}{2}$.

Reversely if there exists a polytime algorithm B that can solve *FACTORING* problem, it is possible to get the prime factors of n. If the prime factors are known, some algorithms known can be used to compute the square root with running time of $\mathcal{O}((\log p)^3)$ bit operations.

The above discussed fact can be generalized to all the cases where n is an arbitrary composite integer. Why n is restricted to the composite n? In the following we will show that it is easy to compute the square root modulo n for a prime n.

If n is prime (p is used in this case for n), then it is easy to find the correct square root of a given integer. In this case there are a few of algorithms to solve this problem. If $p \equiv 3 \mod 4$, $p \equiv 5 \mod 8$, or $p - 1 = 2^s t$, some algorithms (algorithms 3.36, 3.37, 3.39 in [70]) need only $\mathcal{O}((\log p)^3)$ bit operations to compute the square root modulo p. Otherwise computing the square root of p needs in worst case the running time of $\mathcal{O}((\log p)^4)$ bit operations fact 3.35 in [70].

2.4.7. The discrete logarithm problems

The security of many cryptographic techniques depends on the intractability of the discrete logarithm problem. For example the Diffie-Hellman key agreement and its derivatives, ElGamal encryption, ElGamal signature scheme and its variants, DSA signature scheme.

Discrete logarithm is defined as follows: given a finite cyclic group G of order n, a generator g of G, and an element $\beta \in G$, find an integer $x \in \mathbb{Z}_n$ such that $\beta = g^x$. x is called the *discrete logarithm of β to the base g*, denoted $\log_g \beta$.

The problem relevant to discrete logarithm can be further classified into two types.

1. The *discrete logarithm problem* (*DLP*): given a prime p, a generator g of $\mathbb{Z}_p^*$, and an element $\beta \in \mathbb{Z}_p^*$, find an integer x with $0 \leq x \leq p - 2$ such that $g^x \equiv \beta \mod p$.
2. The *generalized discrete logarithm problem* (*GDLP*): given a finite group G of order n, a generator g of G, and an element $\beta \in G$, find an integer x with $0 \leq x \leq n - 1$ such that $g^x = \beta$.

A few algorithms of solving the *DLP* and *GDLP* are studied in the literature. Some of them are described briefly below.

1. **Exhaustive search**: The simplest algorithm for *GDLP* if to compute g^i, $i = 0, 1, 2, \cdots$ until $\beta = g^i$. The complexity takes $\mathcal{O}(n)$ multiplications. Wenn the order n of group G is large it is inefficient.

2. **Baby-step giant-step algorithm**: The *baby-step giant-step algorithm* is a time-memory trade-off of the method of exhaustive search. It requires $\mathcal{O}(\sqrt{n})$ memory and has running time of $\mathcal{O}(\sqrt{n})$ multiplications.

3. **Pollard's rho algorithm for logarithms**: The *Pollard's rho algorithm for logarithms* is applied, if the cyclic group G has a prime order n. It requires the same running time as the *baby-step giant-step algorithm*, but only a negligible amount of storage are required. Thus the Pollard's rho algorithm for logarithms are more suitable in practice.

4. **Pohling-Hellman algorithm**: The *Pohling-Hellman algorithm* computes first the prime factorization of the order n of group G: $n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$, then it takes advantage of the factorization. In fact the it has the running time $\mathcal{O}(\sum_{i=1}^{k} e_i(\log n + \sqrt{p_i})$ multiplications.

5. **Index-calculus algorithm**: The *Index-calculus algorithm* is the most powerful method known to compute discrete algorithms. First it selects a relatively small subset S of G, then a significant fraction of elements of G can be efficiently expressed as products of elements from S. Especially the *Pohling-Hellman algorithm* is appropriate for some group, like $\mathbb{F}_p$ and $\mathbb{F}_{2^m}$. With an optimal choice of the order of S, this algorithm has an running time of $L_q[\frac{1}{2}, c]$ for groups $\mathbb{F}_q$, where $q = p$ or $q = 2^m$, and c is a constant.

Since DSA-Signature and some other widely applied algorithms use the subgroups of $\mathbb{Z}_p^*$, the discrete problem in subgroups of $\mathbb{Z}_p^*$ has special interest. It is described in the following: given two primes p and q such that $q|p-1$. Let G be a cyclic subgroup of $\mathbb{Z}_p^*$ of order q, let g be a generator of G, and let $\beta \in G$, find an integer $x \in \mathbb{Z}_q$ such that $g^x \equiv \beta \mod p$. The Pollard's rho algorithm can solve this problem with complexity $\mathcal{O}(\sqrt{q})$, while the complexity of the index-calculus algorithm (with adaption) is $L_p[\frac{1}{3}, c]$.

To give an overview their complexity is listed in Table 2.4.

Algorithm	Complexity	Remark
exhaustive search	$\mathcal{O}(n)$	
baby-step giant-step algorithm (§ 3.6.2 in [70])	$\mathcal{O}(\sqrt{n})$	$\mathcal{O}(\sqrt{n})$ memory
Pollard's rho algorithm for logarithms (§ 3.6.3 in [70])	$\mathcal{O}(\sqrt{n})$	
Pohling-Hellman algorithm (§ 3.6.4 in [70])	$\mathcal{O}(\sum_{i=1}^{k} e_i(\log n + \sqrt{p_i})$	$n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$
index-calculus algorithm (§ 3.6.5 in [70])	$L_q[\frac{1}{2}, c]$	$q = p$ or $q = 2^m$, c constant
index-calculus algorithm (with adaption) (§ 3.6.5 in [70])	$L_p[\frac{1}{3}, c]$	For *DLP* in subgroups of $\mathbb{Z}_p^*$

Table 2.4.: Complexity of algorithms solving *DLP*/*GDLP*

2.4.8. The Diffie-Hellman problems

The Diffie-Hellman problem is closely related to the discrete logarithm problem (*DLP*). It is significant for public-key cryptography. The security of many cryptography schemes, e.g. Diffie-Hellman key exchange, ElGamal public-key encryption and their derivatives base on the intractability of the Diffie-Hellman problem.

The Diffie-Hellman problem can be further classified into four types.

1. The *computational Diffie-Hellman problem* (*CDHP*): let p be a prime. Given a generator g of $\mathbb{Z}_p^*$, elements $x = g^a$ and $y = g^b$ of $\mathbb{Z}_p^*$, where a and b are unknown, find $g^{ab} \mod p$.

2. The *decision Diffie-Hellman problem* (*DDHP*): let p be a prime. Given a generator g of $\mathbb{Z}_p^*$, elements $x = g^a, y = g^b$, and z of $\mathbb{Z}_p^*$, where a and b are unknown, decide whether or not $z = g^{ab} \mod p$.

3. The *generalized computational Diffie-Hellman problem* (*GCDHP*): Let G be a cyclic group, given a generator g of G, elements $x = g^a$ and $y = g^b$ of G, where a and b are unknown, find g^{ab} in group G.

4. The *generalized decision Diffie-Hellman problem* (*GDDHP*): Let G be a cyclic group, given a generator g of G, elements $x = g^a$,$y = g^b$, and z of G, where a and b are unknown, decide whether or not $z = g^{ab}$ in group G.

If there exists an polytime algorithm A which can solve the discrete logarithm problem in $\mathbb{Z}_p^*$, then given g, p, $g^a \mod p$, $g^b \mod p$, algorithm A computes first a from g^a, and then computes $(g^b)^a = g^{ab}$. Thus *CDHP* polytime reduces to *DLP*, denoted *CDHP* $\leq$ *DLP*. More generally, *GCDHP* $\leq$ *GDLP*.

Apparently the decision problems polytime reduces to the computational problems. Thus *DDHP* $\leq_P$ *CDHP* $\leq_P$ *DLP* and *GDDHP* $\leq_P$ *GCDHP* $\leq_P$ *GDLP*.

2.4.9. Elliptic curve discrete logarithm problem

The elliptic curve cryptosystem(ECC) bases on the intractability of the elliptic curve discrete logarithm problem (*ECDLP*). Since in ECC the elliptic curves over $\mathbb{F}_p$ and $\mathbb{F}_{2^m}$ are mostly applied, the following discussion is sometimes restricted to these both cases.

The elliptic discrete logarithm problem is given in the following: given an irreducible elliptic curve $f(x)$ over $\mathbb{F}_{p^m}$, namely $f(x) \in \mathbb{F}_{p^m}[x]$. Let P and R be two distinct points on $f(x)$, find a positive integer such that $kP = R$.

Recall that the determination of a point kP is referred to as *scalar multiplication* of a point. The *ECDLP* is based upon the intractability of scalar multiplication products. It is widely believed that the elliptic curve discrete logarithm problem is hard to computationally solve when the point P has a large prime order. The known algorithms for solving the *ECDLP* are listed in the following.

1. The Pohlig-Hellman algorithm, it reduces the problem to subgroups of prime order.

2. Shanks' baby-step giant-step method.

3. Pollard's methods, especially the parallel Pollard method of van Oorschot and Wiener.

4. The Menezes-Okamoto-Vanstone (MOV) attack using the Weil pairing.

5. The Frey-Rueck attack using the Tate pairing.

6. The attacks on anomalous elliptic curves, i.e. elliptic curves over $\mathbb{F}_p$ which have p points) due to Semaev, Satoh-Araki and Smart.

7. Weil descent for some special finite fields.

Of the above algorithms, only the anomalous curves attack has polynomial running time. The MOV, Frey-Rueck and Weil descent algorithms are at their fastest subexponential in complexity.

Due to the Pohlig-Hellman algorithm we always restrict to the case where the point P has large prime order. Then the only algorithms which are applicable for all elliptic curves are the methods of Shanks and Pollard, and these methods have exponential complexity.

2.4.10. Elliptic curve Diffie-Hellman problems

Elliptic curves can be applied in many cryptosystems. Such as elliptic curve Diffie-Hellman key exchange (*ECDH*), elliptic curve DSA (ECDSA), and elliptic curve ElGamal (ECElGmal). The security of those elliptic curve cryptosystems bases on the intractability of the elliptic elliptic curve Diffie-Hellman problems, which are closely related to the elliptic curve discrete logarithm problem (*ECDLP*).

Analog to Diffie-Hellman problems, the elliptic curve Diffie-Hellman problems can be further classified into the following two types.

1. The computional elliptic curve discrete Diffie-Hellman problem (*CECDHP*): given an irreducible eliptic curve $f(x)$ over $\mathbb{F}_{p^m}$, and P a point on $f(x)$. Let $X = kP$ and $Y = tP$ be points on $f(x)$, where k and t are unknown positive integers, find the unique third point Z on $f(x)$ such that $Z = ktP$.

2. The decision elliptic curve discrete Diffie-Hellman problem (*DECDHP*) given an irreducible eliptic curve $f(x)$ over $\mathbb{F}_{p^m}$, and P a poit on $f(x)$. Let $X = kP$, $Y = tP$ and Z be points on $f(x)$, where k and t are unknown positive integers, decide whether or not $Z = ktP$.

If there exists a polynomial time algorithm A which solves *ECDLP*, it can be used to solve *CECDHP* in polynomial time. Algorithm A computes first k from kP, and then computes $k(tP) = ktP$. Thus *CECDHP* polytime reduces to *ECDLP*, denoted *CECDHP* $\leq_P$ *ECDLP*. Apparently if an algorithm can solve *CECDHP* in polynomial time, then it can solve *DECDHP* in polynomial time. It computes first $Z' = ktP$, and then compares Z' and Z. Hence *DECDHP* polytime reduces to *CECDHP*. As result, *DECDHP* $\leq_P$ *CECDHP* $\leq_P$ *ECDLP*.

3. Cryptographic Backgrounds

Cryptography is the study of mathematical techniques related to aspects of information security such as confidentiality, data integrity, entity authentication, and data origin authentication. Some 4000 years ago, cryptography was used by Egyptians. Later it was most applied in diplomatic, military and governmental services. Since 1960s as the computers and communications systems booms, cryptography finds its application in private services.

The Data Encryption Standard (DES) is one of the most well-known symmetric cryptosystems. Due to its short key length, it is considered today to be insecure. Hence the Advanced Encryption Standard (AES) was selected as new FIPS standard for symmetric encryption. There are other well-known symmetric schemes, such as IDEA, FEAL, SAFER and RC5.

Diffie and Hellman [28] introduced in 1976 a new concept of public-key cryptosystem which based on the Diffie-Hellman problem. Years later, [61] and [71] introduced a variant of the Diffie-Hellman key exchange, based on the difficulty of the DL problem in the group of points of an elliptic curve (EC) over a finite field. Cryptosystems using EC request much less computation as those cryptosystems without application of EC. Another well-known public cryptosystem based on the difficulty of the RSA problem was introduced in [97]. It is widely used for the digital signature.

The rest of this chapter is organized as follows: Section 3.1 lists some of the most important security goals of cryptography. Then some cryptography mechanisms that are important for this master thesis will be discussed: the symmetric-key encryption with description of AES in Section 3.2, the public-key encryption with description of RSA in Section 3.3, the hash function in Section 3.4, and digital signatures with description of EC-DSA in Section 3.5.

3.1. Security Goals

Cryptography is not the only possibility to provide information security, but the most important one. Some cryptography mechanisms will be addressed in the following sections. Cryptography can be used to achieve the following security properties.

1. **Confidentiality**: It prevents all but those authorized from having the content of the message. There are multiple possibilities to provide confidentiality. They vary from physical to mathematical methods. For example the information to be protected, such as the key, is stored in a room that can only be accessed by the authorized users. Using some encryption algorithms the information can be encrypted so that only the user who has the proper keys and knows the algorithms is able to decrypt it. Encryption can further be roughly classified into two categories: symmetric key encryption (Section 3.2) and public key encryption (Section 3.3).

2. **Integrity**: It prevents unauthorized users from altering data. To assure data integrity, one must be able to detect any unauthorized manipulation of data, such as deletion and insertion. One of the cryptographic approaches to achieve integrity is digital signatures (Section 3.5). Another possibility is given by hash functions (Section 3.4).

3. **Non-repudiation**: It prevents an entity from denying previous actions. In other words, it is a method by which the sender of data is provided with proof of delivery and the recipient is assured of the sender's identity, so that neither can later deny having processed the data. For example, someone who

submits electronic order should not be able to deny it later. One of the cryptographic approaches to achieve non-repudiation is digital signatures (Section 3.5).

4. **Availability**: The service should be available all the time. It must be robust enough to tolerate network failures and must be resistant against Denial-of-Service (DoS) attacks. Since availability is not an issue of this thesis, we refer the reader to the literature for more information.

5. **Authentication**: Authentication is classified into two categories: entity authentication and data origin authentication. In the first case any party entering into a communication session must identify themselves to other participants. In the second case, sent data should be authenticated with respect to its content, time of sending, etc.

Together with key establishment, authentication is the basis of any secure communication. Without some form of authentication, all the other common security properties such as integrity or confidentiality are not more significative.

Authentication bases generally on long-term keys[1] which can be associated with identities. To associate identities with long-term keys, some techniques can be applied. For example a public-key infrastructure (PKI) provides parties with some mechanisms for secure key registration and secure access to long-term keys of prospective peers. The PKI is not an issue of this thesis, because no existing PKIs can be assumed in an ad hoc networks. Hence we refer the reader to literatures such as [82, 45] for more information. Besides the PKI the key transport can also be applied to provides long-term key. Usually there are trusted third-parties known as key distribution centers mediate session keys such as KryptoKnight [72, 48] and Kerberos [68, 69].

Security properties - such as authentication, integrity and confidentiality - are normally only meaningful, if a communication channel is guaranteed during the related session. In order to provider some other security properties, some temporary keys will be applied. Against long-term keys, temporary keys provide the following properties:

1. To minimize the amount of cryptographic material available to cryptanalytic attacks;
2. To minimize the exposure when keys are lost;
3. To make different and unrelated sessions independent;
4. Deriving symmetric session keys from long-terms keys which base on asymmetric cryptography brings a considerable gain in efficiency.

The establishment of such temporary keys, usually called *session keys*, often involves interactive cryptographic protocols. These protocols should ensure that all required security properties, such as authenticity and freshness of the resulting session key, are guaranteed. Such protocols are referred to as key establishment protocols and are the focus of this thesis. Different issues of key establishment will be described in Sections 3.8, 4, and 5.

3.2. Symmetric-Key Encryption

Given an encryption scheme $(\mathcal{P}, \mathcal{C}, \mathcal{K}, \{E_e : e \in \mathcal{K}\}, \{D_d : e \in \mathcal{K}\})$, where $\mathcal{P}$ is the set of all plaintexts, $\mathcal{C}$ is the set of ciphertexts, $\mathcal{K}$ is the set of all keys, E_e is encryption transformation, and D_d is the decryption transformation.

The encryption scheme is defined to be *symmetric* if for all associated encryption/decryption key pairs (e, d), it is computationally „easy“ to determine d knowing only e, and to determine e knowing d. In most practical symmetric-key schemes the keys for encryption and decryption are the same.

[1] Long-term key covers all forms of information which can be associated to identities. For example, it not only includes cryptographic keys such as DES or RSA keys but also encompasses passwords and biometric information.

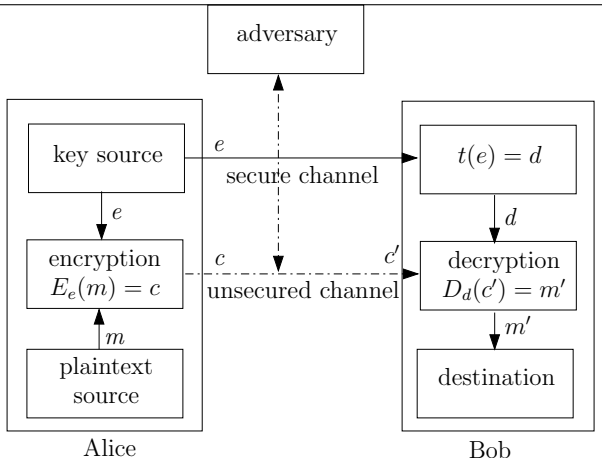


Figure 3.1.: A two-party communication using symmetric-key encryption

Figure 3.1 illustrates a two-party communication using symmetric-key cryptosystem. Alice encrypts the plaintext with the key e and then transports the ciphertext c to Bob via unsecured channel. Since in symmetric-key encryption d can be derived from e easily, e must be transported in a secure channel. Since the ciphertext is transformed via unsecured channel, it could be modified. Thus the ciphertext received by Bob is denoted c'. After receiving the ciphertext c' and encryption key e. Bob derives the decryption key d from e, and then decrypts the ciphertext using d as key. If the ciphertext was modified, Bob will let Alice to send him the ciphertext again.

Symmetric-key schemes can be further classified into more classes, the same as public-key schemes. Two of the mostly used are *block ciphers* and *streaming ciphers*.

3.2.1. Symmetric-key block ciphers

Symmetric-key block ciphers are the most important elements in many cryptographic systems. Individually, they provide confidentiality. As a fundamental building block, their versatility allows construction of pseudorandom number generators, stream ciphers, MACs, and hash functions. They may furthermore serve as a central component in message authentication techniques, data integrity mechanisms, entity authentication protocols, and (symmetric-key)digital signature schemes.

A block cipher is an encryption scheme. The plaintext messages are broken into blocks of a fixed length l, and then each block is encrypted at a time. If l does not divide the message length, the last block will be padded up to l.

Two basic classes of block ciphers, substitution cipher and transposition cipher, will be first introduced, while product ciphers which combines the properties of both above will be studied in the rest of this section.

3.2.1.1. Substitution ciphers

In substitution ciphers, symbols or groups of symbols are replaced by other symbols or groups of symbols. Three important classes of substitution ciphers are *simple substitution ciphers*, *homophonic substitution ciphers*, and *polyalphabetic substitution ciphers*. Simple substitution ciphers will be described in the following, for details about the other two ones please reference to pages 17-18 in [70].

Given a set $\mathcal{A}$ of some symbols. Let $\mathcal{M}$ be the set of all strings of length k over $\mathcal{A}$ and let $\mathcal{K}$ be the set of all permutations on $\mathcal{A}$. E_e is defined in following: For $m = (m_1 m_2 \cdots m_k) \in \mathcal{M}$, $e \in \mathcal{K}$,

$$E_e := (e(m_1)e(m_2)\cdots e(m_k)) = (c_1 c_2 \cdots c_k) = c \tag{3.1}$$

To decrypt the ciphertext c an inverse permutation of e, $d := e^{-1}$, is applied. For a given $c = (c_1 c_2 \cdots c_k)$,

$$D_d := (d(c_1)d(c_2)\cdots d(c_k)) = (m_1 m_2 \cdots m_k) = m. \tag{3.2}$$

3.2.1.2. Transposition ciphers

In this section we describe a *simple transposition cipher*, which permutes the symbols of one block.

Given a symmetric-key block encryption, where each block consists of t subblocks of the same length. Let $\mathcal{M}$ be the set of all messages, and $\mathcal{K}$ be the set of all permutations on the subsets. For each $e \in \mathcal{K}$ and $m = (m_1 m_2 \cdots m_t) \in \mathcal{M}$ the encryption function is defined as follows.

$$E_e(m) = (m_{e(1)} m_{e(2)} \cdots m_{e(t)}) \tag{3.3}$$

The decryption key corresponding to e is the inverse permutation $d = e^{-1}$. The following transposition is used to decrypt $c = (c_1 c_2 \cdots c_t)$.

$$D_d(c) = (c_{d(1)} c_{d(2)} \cdots c_{d(t)}). \tag{3.4}$$

For the same key, each combination of symbols is transported to fixed combination of symbols, which confuses surely the crytoanalyse such as the observation of the distribution of letter frequencies. However it is possible to break some transposition ciphers with anagramming technique.

3.2.1.3. Product ciphers

Simple substitution and transposition ciphers individually do not provide a very high level of security. Yet by combining these transformations it is possible to obtain strong ciphers. The most practical and effective symmetric-key systems are product ciphers, e.g. AES [80], DES [23], FEAL [35], IDEA [67], SAFER (§7.7 in [70]), Triple-DES [23], and RC5 [11]. The essential block cipher AES will be introduced in following. For brevity, the following description omits the exact transformations and permutations which specify the algorithm.

Advanced Encryption Standard (AES), also known as **Rijndael**, was designed by Vincent Rijmen and Joan Daemen, and was adopted by National Institute of Standards and Technology (NIST) as US FIPS PUB 197 [80] in November 2001 after a 5-year standardization process.

Unlike its predecessor DES, Rijndael uses substitution-permutation instead of Feistel function. AES has a fixed block size of 128 bits and a key size of 128, 192 or 256 bits.

AES operates on a 4×4 array of bytes, termed the state. For encryption, each round of AES (except the last round) consists of four stages, while the final round omits the **MixColumns** stage.

1. **SubBytes**: Each byte in the array is updated using an 8-bit S-box. This operation provides the non-linearity in the cipher. The S-box used is derived from the inverse function over $\mathbb{F}_{2^8}$, known to have good non-linearity properties. To avoid attacks based on simple algebraic properties, the S-box is constructed by combining the inverse function with an invertible affine transformation. The S-box is also chosen to avoid any fixed points (and so is a derangement), and also any opposite fixed points. The mechanism is illustrated in Figure 3.2.

2. **ShiftRows**: It operates on the rows of the state; it cyclically shifts the bytes in each row by a certain offset. For AES, the first row is left unchanged. Each byte of the second row is shifted one to the left. Similarly, the third and fourth rows are shifted by offsets of two and three respectively. In this way, each column of the output state of the ShiftRows step is composed of bytes from each column of the input state. The mechanism is illustrated in Figure 3.3.

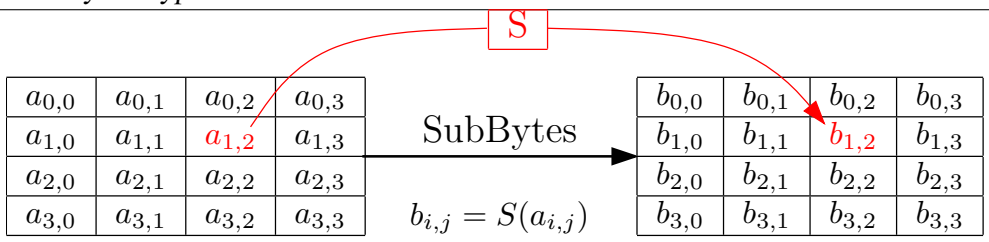


Figure 3.2.: SubBytes function for AES

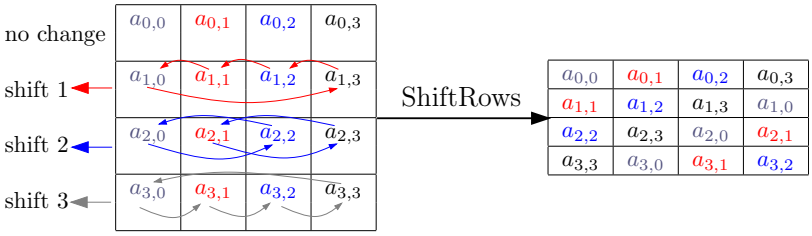


Figure 3.3.: ShiftRows function for AES

3. **MixColumns**: In this step, the four bytes of each column of the state are combined using an invertible linear transformation. Together with ShiftRows, MixColumns provides diffusion in the cipher. Each column is treated as a polynomial over $\mathbb{F}_{2^8}$ and is then multiplied modulo $x^4 + 1$ with a fixed polynomial $c(x)$. The mechanism is illustrated in Figure 3.4.

4. **AddRoundKey**: In this step, the subkey is combined with the state. For each round, a subkey is derived from the main key using the key schedule; each subkey has the same size as the state. The subkey is added by combining each byte of the state with the corresponding byte of the subkey using bitwise XOR. The mechanism is illustrated in figure 3.5.

3.2.2. Streaming ciphers

Due to the fixed relation between some plaintext and ciphertext in the simple substitution cipher and in the simple transposition cipher, both ciphers do not provide enough security against attacks, e.g. cryptoanalysis. Security can be achieved if encryption transformation changes for each symbol of plaintext being encrypted. This provide streaming ciphers.

Let $\mathcal{K}$ be the key space for a set of encryption transformations, and let $\mathcal{A}$ be an alphabet of symbols. Given a simple substitution cipher E_e with block length t where $e \in \mathcal{K}$. Let $(m_1 m_2 m_3 \cdots)$, $m_i \in \mathcal{A}$, be a plaintext string and let $(e_1 e_2 e_3 \cdots)$ be a key stream in $\mathcal{K}$. Then the ciphertext string $(c_1 c_2 c_3 \cdots) = (E_{e_1(m_1)} E_{e_2(m_2)} E_{e_3(m_3)} \cdots)$.

The variation of the key makes it more secure. A secure algorithm should be applied to generate the stream. We call such an algorithm a key stream generator, such as Cipher Block Chaining (CBC), Cipher

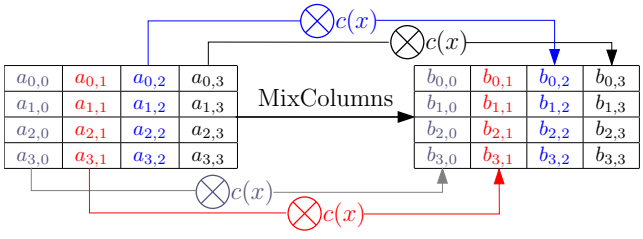


Figure 3.4.: MixColumns function for AES

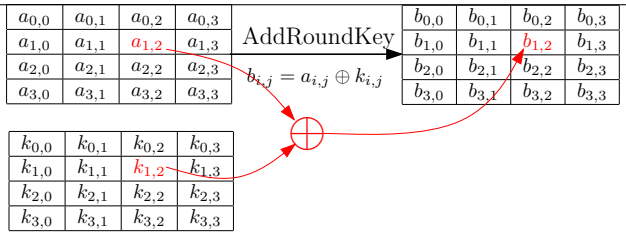


Figure 3.5.: AddRoundKey function for AES

Feedback (CFB), and Output Feedback (OFB). The generator can generate the key at random, or from an initial value called seed, or both.

3.3. Public-Key Encryption

Given an encryption scheme $(\mathcal{P}, \mathcal{C}, \mathcal{K}, \{E_e : e \in \mathcal{K}\}, \{D_d : e \in \mathcal{K}\})$, where $\mathcal{P}$ is the set of all plaintexts, $\mathcal{C}$ is the set of ciphertexts, $\mathcal{K}$ is the set of all keys, E_e is encryption transformation, and D_d is the decryption transformation.

Public-key encryption schemes base on the intractability of solving some computational problems. Consider any pair of associated encryption/decryption transformations (E_e, D_d) and suppose that each pair has the property that knowing E_e it is computationally infeasible, given a random ciphertext $c \in \mathcal{C}$, to find the message $m \in \mathcal{M}$ such that $E_e(m) = c$. This property implies that given an encryption key e it is infeasible to determine the corresponding decryption key d. However e can be derived from d.

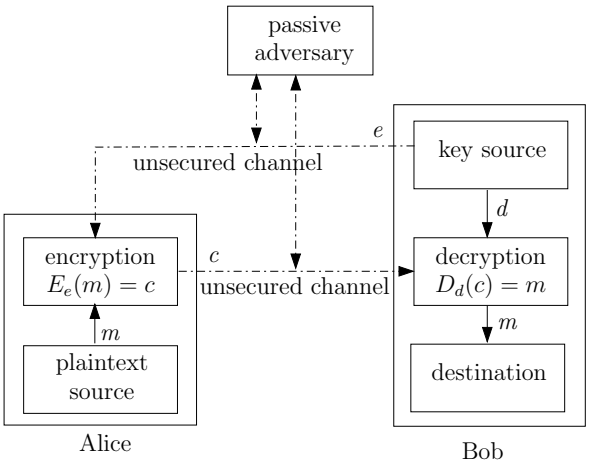


Figure 3.6.: A two-party communication using public-key encryption

Figure 3.6 illustrates a two-party communication using public-key cryptosystem. Bob generates first a keypair (e, d), and then transports only public key e to Alice via any channel, but the secret key d should be kept secure and secret. Alice can subsequently encrypt a message m using the pulic key of Bob, $c = E_e(m)$. The encrypted message could now be sent via any channel to Bob. Bob decrypt subsequently the ciphertext using the corresponding secret key, $m = D_d(c)$.

As described in Figure 3.6, the public key need not to be kept secret (in fact, it may be widely available), only its authentication should be verified. Hence it is much easier to distribute the public key than to distribute the symmetric key.

The main objective of public-key encryption is to provide privacy or confidentiality. Due to the public knowledge of the encryption transformation, public-key encryption alone does not provide data authenti-

cation, source authentication, or data integrity. To achieves such goals some additional techniques, such as message authentication codes (MAC) and digital signatures, must be applied. Public-key encryption schemes are typically much slower than symmetric-key encryption. It is usually applied together with the symmetric-key encryption. The message is usually encrypted with symmetric-key encryption, whose keys are transported using public-key encryption. Some other small data items such as credit card numbers and PINs can be also encrypted by public-key encryption.

Many of public-key encryption schemes are related to computational problems listed in Table 3.1. Some of the problems are described in Section 2.4. The detail of others can be found in the literature. Two of those, RSA and ElGamal encryption schemes will be described below.

public-key encryption scheme	**related computational problem**
RSA	integer factorization problem RSA problem
Rabin	integer factorization problem square roots modulo composite n
ElGamal	discrete logarithm problem Diffie-Hellman problem
generalized ElGamal	generalized discrete logarithm problem generalized Diffie-Hellman problem
McEliece	linear code decoding problem
Merkle-Hellman knapsack	subset sum problem (§ 3.10 in [70])
Chor-Rivest knapsack	subset sum problem (§ 3.10 in [70])
Goldwasser-Micali probabilistic	quadratic residuosity problem (§ 3.4 in [70])
Blum-Goldwasser probabilistic	integer factorization problem Rabin problem (§ 3.9.3 in [70])

Table 3.1.: Some public-key encryption schemes, and the related computational problems

3.3.1. RSA encryption

The RSA encryption scheme was released as a MIT „Technical Memo“ in April 1977, and later published in 1978 by Ron Rivest, Adi Shamir and Len Adleman in [97]. It bases on the intractability of solving the RSA problem, which polytime reduces to the integer factorization problem. In RSA encryption scheme, a key pair (e, d) is first generated using algorithm 3.1.

Algorithm 3.1 (Key generation for RSA public-key encryption)

Input: key bit length l.
Output: keypair (e, d).

1. *Generate two random (and distinct) primes* p *and* q *independently, roughly of* $\frac{l}{2}$ *bits.*
2. *Compute* $n = pq$ *and* $\phi(n) = (p-1)(q-1)$.
3. *Select a random integer* e, $1 < e < \phi$, *such that* $\gcd(e, \phi) = 1$.
4. *Use the extended Euclidean algorithm to compute the unique integer* d, $1 < d < \phi$, *such that* $ed \equiv 1 \mod \phi$.
5. *The public key* (e, n), *and private key* (d, n).

Remark: Because of some attacks, RSA Standard PKCS#1 version 2.1 [99] uses $\lambda(n) = lcm(p-1, q-1)$ instead of $\phi = (p-1)(q-1)$. In order to calculate more efficiently, the Chinese Remainder Theorem (CRT)[2] is commonly used to decrypt the ciphertext. Hence some values related to CRT are computed in the key generation, such as p's CRT exponent $dP \equiv e^{-1} \mod (p-1)$, q's CRT exponent $dQ \equiv e^{-1} \mod (q-1)$, and CRT coefficient $qInv \equiv q^{-1} \mod p$. In this case, the public key is still (e, n), but the private key is extended to be $(d, p, q, dP, dQ, qInv)$.

Assumed entity A will send message m to entity B. First A obtains the public key of B (e, n), and computes the ciphertext $c = m^e \mod n$, subsequently A sends the ciphertext to B. After receipt of ciphertext c, B decrypts c with its corresponding private key. If (e, n) is used, then $m \equiv c^d \mod n$. The scheme is correct since $ed \equiv 1 \mod \phi(n)$, $c^d \equiv m^{ed} \equiv m^{ed \mod \phi(n)} \equiv m \mod n$. Otherwise CRT is used to decrypt c.

One measure of the security of RSA is the length of the modulus n. Theoretical hardware device TWIRL described by Shamir and Tromer in 2003 called into question the security of 1024 bit keys [119]. In 1998, Daniel Bleichenbacher described the first practical adaptive chosen ciphertext attack, against RSA-encrypted messages using the PKCS#1 $v1$ padding scheme [16]. Thus RSA Laboratories has released new versions of PKCS #1 that are not vulnerable to these attacks.

3.3.2. ElGamal encryption

The ElGamal algorithm bases on the intractability of solving discrete logarithms. It was proposed by Taher ElGamal in 1984 [33, 34].

As with all asymmetric key encryption algorithms, ElGamal consists of three components: the key generation algorithm (algorithm 3.2), the encryption algorithm (algorithm 3.3), and the decryption algorithm (algorithm 3.4).

Algorithm 3.2 (Key generation for ElGamal public-key encryption)

Input: key bit length l.
Output: keypair $(pubkey, privkey)$.

1. *Generate a large random prime p of l bits, and a generator g from $\mathbb{Z}_p^*$.*
2. *Select a random integer a in interval [1, p-2], and compute $g^a \mod p$.*
3. *The public key $pubkey = (p, g, g^a)$, and the private key is $privkey = a$.*

Algorithm 3.3 (ElGamal public-key encryption)

Input: public key (p, g, g^a), message $m \in \mathbb{Z}_p$.
Output: ciphertext c

1. *Choose a integer random t from interval $[1, p-2]$.*
2. *Compute $c_1 \equiv g^t \mod p$, and $c_2 \equiv m \cdot (g^a)^t \mod p$.*
3. *Get ciphertext $c = (c_1, c_2)$.*

[2] CRT can be generalized as follows. Given a set of simultaneous congruences

$$x \equiv x_i \mod m_i, \text{ i = 1, 2, ..., r,}$$

where the m_i are pairwise prime. The solution of the set of congruences is

$$x \equiv \sum_{i=1}^{r} x_i y_i \frac{M}{m_i} \mod M,$$

where $M = \Pi_{i=1}^{r} m_i$, and $y_i = \left(\frac{M}{m_i}\right)^{-1} \mod m_i$.

Algorithm 3.4 (ElGamal public-key decryption)

Input: ciphertext $c = (c_1, c_2)$*, private key* a *and public key* (p, g, g^a).
Output: message m.

1. *Compute* $c_1^{p-1-a} \equiv c_1^{-a} \mod p$.
2. *Recover the message* $m = c_1^{-a} \cdot c_2 \equiv g^{-at} \cdot m \cdot (g^a)^t \equiv m \mod p$.

The security of ElGamal bases, in part, on the intractability of solving the discrete logarithm problem (DLP) in finite group. However, the security of ElGamal actually relies on the intractability of solving the Computational Decisional Diffie-Hellman problem (CDHP).

3.4. Hash Functions

Hash function, often called a one-way hash function, is one of the fundamental primitives in modern cryptography.

A *hash function H* is a transformation that takes an input m and returns a fixed-size string, which is called the *hash value h*. We call this transformation $H(.)$, that is, $h = H(m)$. Hash functions with just this property have a variety of general computational uses, but when employed in cryptography, hash functions are usually chosen to have some additional properties.

The input of a cryptographic hash function $H(\cdot)$ can be of any length, while the output must have a fixed length, and $H(\cdot)$ should be relatively easy to compute. $H(\cdot)$ should be further *one-way* and *collision-free*.

A hash function H is said to be one-way if it is hard to invert. That is, given a hash value h, it is computationally infeasible to find some input x such that $H(x) = h$.

A function H is said to be weakly collision-free, if given a message x, it is computationally infeasible to find a message $y \neq x$ such that $H(x) = H(y)$. A strongly collision-free hash function H is one for which it is computationally infeasible to find any two messages x and y such that $H(x) = H(y)$.

For more information and a particularly thorough study of hash functions, see [92].

The hash value represents concisely the longer message or document from which it was computed; this value is called message digest. One can think of a message digest as a *digital fingerprint* of the larger document. Examples of well known hash functions are MD-family (MD2 [53], MD4 [94, 95], and MD5 [96]), RipeMD-family (RIPEMD-128 [30], RIPEMD-160 [30], RIPEMD-256 [30], RIPEMD-320 [30]), and SHA-family (SHA-1 [78, 81], SHA-256 [81], and SHA-384 [81], SHA-512 [81]).

The most cryptographic functions use compression functions to compress the data. A compression function takes a fixed-length input and returns a shorter, fixed-length output. Given a compression function, a hash function can be defined by repeated application of the compression function until the entire message has been processed. In this process, a message of arbitrary length is broken into blocks whose length depends on the compression function, and „padded“ so that the size of the message is a multiple of the block size. For example the block size of SHA-1 and RIPEMD-160 is 512 bits. The blocks are then processed sequentially, taking as input the result of the hash so far and the current message block, with the final output being the hash value for the message. This process is illustrated in Figure 3.7.

Cryptographic hash functions are commonly used for digital signatures and for data integrity. Since hash functions are generally faster than encryption or digital signature algorithms, it is typical to compute the digital signature or integrity check to some document by applying cryptographic processing to the document's hash value, which is smaller compared to the document itself.

Cryptographic functions may be applied for the data integrity. The hash value for an input m will be computed at some point in time. At a subsequent point in time, the integrity of m can be verified by recomputing the hash-value of m and comparing it with the original hash value.

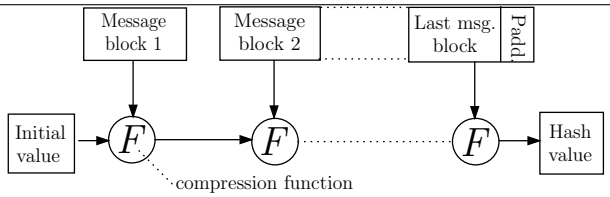


Figure 3.7.: Iterative structure for hash functions

The fourth application of cryptographic hash functions is to derive key information from some secret. In the group key agreement, it will be used to derive keys for authentication, symmetric encryption, from the agreed group key.

3.5. Digital Signatures

Digital signature is a method of authenticating digital information often treated, as analogous to a physical signature on paper. Digital signatures allow an entity to bind its identity to a piece of information.

Let $\mathcal{M}$ be a set of possible messages which can be signed, and $\mathcal{S}$ a set of all possible signatures, and let $\mathcal{V} := \{true,\ false\}$. Two transformations are defined in the following.

1. The signing transformation of entity A: $S_A := M \rightarrow \mathcal{S}$. That means that S_A is a transformation from the message set to the signature set.
2. The verifying transformation of entity A's signatures: $V_A := \mathcal{M} \times \mathcal{S} \rightarrow \mathcal{V}$.

Of the two transformations, S_A must be kept secret and secure, and is used to sign a message, while V_A must be publicly known, and is used to verify the signatures signed for the message by A.

Suppose that entity A wishes to send a message m with its signature to B. A gets the signature through $s = S_A(m)$. It transmits then the message with the signature (m, s) to entity B. B must first obtain the verifying transformation for A's signature, then calculates $V_A(m, s)$. If the result is $true$, then B accepts the signature, otherwise rejects the signature.

A signature is called unforgeable if it is computationally infeasible for any entity other than A to find an $s \in \mathcal{S}$, for any $m \in \mathcal{M}$, such that $V_A(m, s) = true$.

They can be roughly be classified into two types.

1. **Digital signature schemes with appendix**: It requires the message as input to the verification algorithm. Examples of mechanisms providing digital signatures with appendix are the DSA [77], ECDSA [6], ElGamal [34], and Schnorr (§11.5.3 in [70]) signature schemes. Digital signature schemes with appendix are in practice most commonly used.
2. **Digital signature schemes with message recovery**: The signature can be verified without knowing of original message, and the message can be derived from the signature. Examples of mechanisms providing digital signatures with message recovery are RSA [97], Rabin [93], and Nyberg-Rueppel [87] public-key signature schemes. However, they are only suitable for short messages, For example, in an RSA signature scheme, it can only sign a message m which is less than the modulus n.

Any digital signature scheme with message recovery can be turned into a digital signature scheme with appendix by simply hashing the message and then signing the hash value. The message is now required to verify the signature, more exactly, to verify the hash value. In fact, the most in practice applied digital signatures schemes, of both types, hash first the message and then sign the hash value. A typical scenario using hash value instead of the message is illustrated in Figure 3.8. An example, the ECDSA signature scheme, will be described in following to explain the scenario mentioned above.

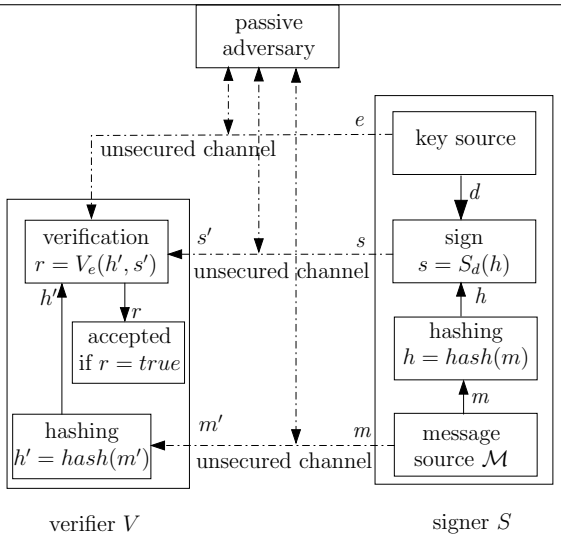


Figure 3.8.: A digital signature scheme using hash value

The Elliptic Curve Digital Signature Algorithm (ECDSA) is a variant of the Digital Signature Algorithm (DSA) which operates on elliptic curve groups. It was proposed as ANSI Standard under the number ANSI-X9.62 [6] (working draft). In ECDSA elliptic curves over $\mathbb{F}_p$ and $\mathbb{F}_{2^m}$ are used. In the following the parameters for both curves, the keypair generation (algorithm 3.5, signature generation (algorithm 3.6) and verification algorithm (algorithm 3.7) will be discussed.

Elliptic curve domain parameters over $\mathbb{F}_p$ or $\mathbb{F}_{2^m}$ shall consist of the following parameters:

1. A field size q which defines the underlying finite field $\mathbb{F}_q$.
   a) $q = p$ for $\mathbb{F}_p$, where p is a prime greater than 3.
   b) $q = 2^m$ for $\mathbb{F}_{2^m}, m \in \mathbb{N}$.
2. (Optional) A bit string SEED of length at least 160 bits.
3. Two field elements a and b in $\mathbb{F}_q$ which define the equation of the following elliptic curve.
   a) $E : y^2 = x^3 + ax + b$ for $\mathbb{F}_p$, and $4a^3 + 27b^2 \not\equiv 0 \mod q$;
   b) $E : y^2 + xy = x^3 + ax^2 + b$ for $\mathbb{F}_{2^m}$, and $b \neq 0$.
4. Two field elements x_G and y_G in $\mathbb{F}_q$ which define a point $G = (x_G, y_G)$ of prime order on E such that $G \neq O$.
5. The prime order n of the point G (it must be the case that $n > 2^{160}$ and $n > 4\sqrt{q}$).
6. (Optional) The cofactor $h = |E(\mathbb{F}_q)|/n$.

For convenience, we define $PARAM$ to be a set of elliptic curve domain parameters: $PARAM := (q, [SEED], a, b, x_G, y_G, n, [h])$. The square brackets mean that the element is optional.

Algorithm 3.5 (Key pair generation of ECDSA)

Input: A valid $PARAM$.
Output: A key pair (Q, d) associated with $PARAM$.

1. *Choose an integer d from interval [1, n-1] at random.*
2. *Compute the point $Q = (x_Q, y_Q) = dG$.*
3. *If $Q = O$ then go to step 1.*

4. *The key pair is (Q, d), where Q is the public key, and d is the private key.*

For convenience, we define Equation 3.5 which converts the element α of a point $R(\alpha, \beta)$ to an integer x (the same for β).

$$x = \begin{cases} \alpha, & \text{for } \mathbb{F}_p; \\ \sum_{i=0}^{m-1} 2^i \alpha_i, & \text{for } \mathbb{F}_{2^m},\ \alpha = (\alpha_{m-1} \cdots \alpha_1 \alpha_0)_2. \end{cases} \tag{3.5}$$

Algorithm 3.6 (Signature generation of ECDSA)

Input: A message m, a valid $PARAM$, a private key d, a hash function H.
Output: A digital signature (r, s), where $1 \leq r, s \leq n-1$.

1. *Compute the hash value h using hash function H : $h = H(m)$.*
2. *Modular computations:*
   - 2.1. *Choose a integer t from interval $[1, n-1]$ at random.*
   - 2.2. *Compute the elliptic curve point $R(x_R, y_R) = tG$.*
3. *Modular Computations:*
   - 3.1. *Convert x_R to an integer x according to Equation 3.5.*
   - 3.2. *If $x = 0$ then go to step 2.1, otherwise $r = x$.*
   - 3.3. *Compute $s = t^{-1}(h + dr) \mod n$.*
   - 3.4. *If $s = 0$ then go to step 2.1.*
4. *Return a digital signature (r, s).*

Algorithm 3.7 (Signature verification of ECDSA)

Input: A message m', a signature (r', s'), a valid $PARAM$, a public key Q, a hash function H.
Output: Accept or reject the signature.

1. *Compute the hash value h' using hash function H : $h' = H(m')$.*
2. *Modular Computations:*
   - 2.1. *If r' or s' is not in interval $[1, n-1]$, then reject the signature.*
   - 2.2. *Compute $c = s'^{-1} \mod n$.*
   - 2.3. *Compute $u_1 = h'c \mod n$ and $u_2 = r'c \mod n$*
3. *Compute the elliptic curve point $R'(x_{R'}, y_{R'}) = u_1 G + u_2 G$.*
4. *Signature Checking:*
   - 4.1. *if $R' = O$, reject the signature.*
   - 4.2. *Convert $x_{R'}$ to an integer x according to Equation 3.5.*
   - 4.3. *Compute $v = x \mod n$.*
   - 4.4. *If $r' = v$, then accept the signature, otherwise reject the signature.*

3.6. Key Size

In [64], Lenstra *et al.* report on the minimum key lengths required for secure symmetric-key cryptosystems, RSA, and discrete logarithm based cryptosystems both over finite fields and over groups of elliptic curves over prime fields.

According to table 1 in [64], to provide adequate protection against the most serious threats, for well-funded commercial enterprises and government intelligence agencies, keys used to protect data today (2006) have to satisfy:

1. Symmetric keys of at least 75 bits. Therefore AES with all key length variations, Triple-DES can be used, and DES should be deprecated.
2. Hash functions of at least 150 bits. So SHA-1[3] and RIPEMD-160 can be used, but MD4 and MD5 not.
3. RSA moduli of at least 1191 bits;
4. Subgroup discrete logarithm systems with subgroups whose order is at least 133 bits with finite fields of at least 1191 bits.
5. Elliptic curve systems over prime fields with primes of at least 141 bits if one is confident that no cryptanalytic progress will take place, at least 148 bits if one prefers to be more careful.

Hence, hash functions which provides not less than 150 bit output, such as SHA-1, RIPEMD-160 can be used, while MD4, MD5, and RIPEMD-128 which provide only 128 bits output must be deprecated.

To protect information adequately for the next 20 years, namely until 2026, in the face of expected advances in computing power, the key length should be updated as follows:

1. Symmetric keys of at least 90 bits. So AES and Triple-DES can still be used.
2. Hash functions of at least 180 bits. So SHA-1 and RIPEMD-160 should no more be applied, but their successors, SHA-256 and RIPEMD-256.
3. RSA moduli of at least 2236 bits;
4. Subgroup discrete logarithm systems with subgroups whose order is at least 160 bits with finite fields of at least 2236 bits.
5. Elliptic curve systems over prime fields with primes of at least 170 bits if one is confident that no cryptanalytic progress will take place, at least 205 bits if one prefers to be more careful.

3.7. Two-Party Key Exchange

Diffie and Hellman [28] developed the original Diffie-Hellman key agreement protocol (DHKE) in 1976. The protocol depends on the Diffie-Hellman problem for its security. It allows two users to exchange a secret key over an insecure medium without any prior secrets.

Due to failure of authentication of the participants, the basic DHKE is vulnerable to a man-in-the-middle attack. In resent years a number of approaches to solve this problem have been researched. An approach to derive a shared secret from some weak shared key such as password, without using Diffie-Hellman key exchange, is present in Section 3.7.2. Another of them, password authenticated Diffie-Hellman key exchange is described in Section 3.7.3.

[3] However, collisions of SHA-1 can be found with complexity less than 2^{69} hash operations [123].

3.7.1. Basic Diffie-Hellmann key exchange

In the original DH protocol, suppose two players A and B want to agree on a shared secret key, they proceed as follows: First, they agree on a prime p and a generator g of the multiplicative group $\mathbb{Z}_p^*$. Then A and B generate randomly private values S_A and S_B from $\mathbb{Z}_p^*$ respectively (more generally, instead of $\mathbb{Z}_p^*$, any cyclic subgroup of order q) can be used along with randomly chosen secrets from $\mathbb{Z}_q$). Then they derive their public values g^{S_A} and g^{S_B}, and exchange them. Finally, A computers $(g^{S_B})^{S_A} \bmod p$, and B computes $(g^{S_A})^{S_B} \bmod p$. Since $(g^{S_B})^{S_A} = (g^{S_A})^{S_B} = g^{S_A S_B}$, A and B now have a shared secret key $S_{AB} = g^{S_A S_B} \bmod p$.

The main drawback of the original DHKE protocol is that it is vulnerable to a man-in-the-middle attack. In this attack, an opponent C intercepts A's public value and sends her own public value to B. When B transmits his public value, C substitutes it with her own and sends it to A. C and A thus agree on one shared key and C and B agree on another shared key. After this exchange, C simply decrypts any messages sent out by A or B, and then reads and possibly modifies them before re-encrypting with the appropriate key and transmitting them to the other party. This vulnerability is present because Diffie-Hellman key exchange does not authenticate the participants. Possible solutions include the use of digital signatures and other protocol variants, such as STS protocol developed by Diffie *et al.* in [29], and password authenticatedDH key exchange presented by Bellovin and Merrit in [14] which will be discussed in Section 3.7.3.

The former protocol allows two parties to authenticate themselves to each other by using digital signatures and public-key certificates. Roughly speaking, the basic idea is as follows. Prior to execution of the protocol, the two parties A and B each obtain a public/private key pair and a certificate for the public key. During the protocol, A computes a signature on certain messages, including the public value $g^a \bmod p$. B proceeds in a similar way. Even though an adversary is still able to intercept messages between A and B, he cannot forge signatures without A's private key and B's private key. Hence, the enhanced protocol is resistant to man-in-the-middle attacks.

In recent years, the original Diffie-Hellman protocol has been understood to be an example of a much more general cryptographic technique, that is the derivation of a shared key from one party's public key and another party's private key. The parties' key pairs may be new generated at each run of the protocol, as in the original Diffie-Hellman protocol. The public keys may be certified, so that the parties can be authenticated.

3.7.2. Generic password authenticated key exchange

In [14], Bellovn and Merrit addressed a protocol called **encrypted key exchange (EKE)**. The involved two parties derive a strong session key from only a weak shared key, in practice usually password, so that it is resistant to dictionary attacks. There are a number of variants of EKE, one of them is described in Section 3.7.3. But the basic form remains unchanged. Let A and B be two parties which share a weak secret P. A has a random key pair (E_A, D_A), where E_A is key for encryption, respectively, D_A is key for decryption. They want to share a strong secret over an insecure channel. The process is illustrated in Protocol 3.1.

Protocol 3.1 (Generic password authenticated key exchange)

1. *A encrypted E_A using P as key and sends the result together with its identity to B.*
   $A \xrightarrow{\quad A, P(E_A) \quad} B$
2. *First, B decrypts the message to get E_A. B chooses then randomly R and encrypts it with E_A. Then the result is encrypted using P as key and sent to A.*
   $A \xleftarrow{\quad P(E_A(R)) \quad} B$
3. *A generates random $challenge_A$ and a secret S_A and encrypts them using R as key. The result is sent to B.*
   $A \xrightarrow{\quad R(challenge_A, S_A) \quad} B$

4. *B does the same as A in step 3. Additionally $h(challenge_{_A})$[4] is added to the message to convince A on the knowledge of R, thus that of P.*

A $\xleftarrow{\qquad R(h(challenge_{_A}), challenge_{_B}, S_B) \qquad}$ B

5. *Now A is sure that B knows P. He should now convince B that he also knows P.*

A $\xrightarrow{\qquad R(challenge_{_B}) \qquad}$ B

At the end of the protocol, both parties are confirmed that the other party knows the shared weak secret P, and share a strong secret $K_{AB} = f(S_A, S_B)$, where f is a public one-way function.

However, this basic protocol of EKE cannot be directly extended to form a contributory group key agreement. Hence it is slightly modified by Asokan and Ginzborg in [8] to solve the above mentioned problem. The idea is to use the secrets S_A and S_B instead of the challenges $challenge_{_A}$ and $challenge_{_B}$ to confirm that the other party does know the shared weak secret. The process is illustrated in Protocol 3.2 (We list only the communication flow, and use same notations as in the basic EKE).

Protocol 3.2 (Modified generic password authenticated key exchange)

1. A $\xrightarrow{\qquad A, P(E_A) \qquad}$ B
2. A $\xleftarrow{\qquad P(E_A(R, S_B)) \qquad}$ B
3. A $\xrightarrow{\qquad R(S_A) \qquad}$ B
4. A $\xrightarrow{\qquad K(S_A, f(S_A, S_B)) \qquad}$ B
5. A $\xleftarrow{\qquad K(S_B, f(S_A, S_B)) \qquad}$ B

The shared strong secret is now $K = f(S_A, S_B)$.

3.7.3. Password authenticated Diffie-Hellman key exchange

We present here one version of the EKE which is based on Diffie-Hellman key exchange for two parties. Let A and B be two parties who want to share a strong secret and have shared a weak secret P before the key exchange. The process is illustrated in Protocol 3.3.

Protocol 3.3 (Password authenticated Diffie-Hellman key exchange)

1. *A generates randomly a secret S_A, computes its public value g^{S_A}, then encrypts it with P, finally A sends its encrypted public value together with its identity to B.*

A $\xrightarrow{\qquad A, P(g^{S_A}) \qquad}$ B

2. *First, B decrypts the message to get g^{S_A}. Secondly B chooses a random secret S_B and a random challenge $challenge_{_B}$. B then computes the session key $K = (g^{S_A})^{S_B} = g^{S_A S_B}$ and its public value g^{S_B}. B encrypts g^{S_B} with P, $challenge_{_B}$ with K. Finally both encrypted data is sent to A.*

A $\xleftarrow{\qquad P(g^{S_B}), K(challenge_{_B}) \qquad}$ B

[4] h is a public one-way function.

3. *A extracts the message to get B's public value and challenge. Using the extracted g^{S_B} it computes the session key $K = (g^{S_B})^{S_A} = g^{S_A S_B}$. Similarly, A chooses $challenge_A$ and sends the encrypted challenges $challenge_A$ and $challenge_B$ to B using K as key.*

$$A \xrightarrow{\quad K(challenge_A, challenge_B) \quad} B$$

4. *B sends encrypted A's challenge back to A to ensure that B knows the session key.*

$$A \xleftarrow{\quad K(challenge_A) \quad} B$$

Now both parties A and B are convinced that the partner does know the shared weak secret, and they share a strong secret $K = g^{S_A S_B}$.

3.7.4. 2-Party elliptic curve Diffie-Hellman

In the elliptic curve Diffie-Hellman (ECDH) key exchange, the two communicating parties A and B agree beforehand to use the same curve parameters and base point G. They each generate their private keys S_A abd S_B, respectively, and the corresponding public keys $P_A = S_A \cdot G$ and $P_B = S_B \cdot G$. Both parties A and B exchange their public keys, and each multiplies its private key with the other party's public key. $S_A \cdot P_B = S_B \cdot P_A = S_A S_B \cdot G$.

3.8. Group Key Management

The primary goal of key group key management is to share a secret among a specified set of participants. Many approaches can be used to achieve this goal. The main approaches are key transport, key pre-distribution, and key agreement. The latter is the focus of this thesis.

The rest of this section is organized as follows: Section 3.8.1 gives an overview of key pre-distribution schemes, while Section 3.8.2 describes briefly key transport schemes. The rest discusses issues in key agreement schemes, especially in context of group key agreement, e.g. cryptographic properties, group events, and criteria to measure group key agreement schemes.

3.8.1. Key pre-distribution schemes

In a key pre-distribution scheme, some secret key information is distributed to all nodes before the communication. If the structure is known, more restrictedly, if the nodes know their neighborhood before deployment, pairwise keys can be established between these nodes (and only these nodes). But for some networks, e.g. ad hoc networks, the network topology is random.

A solution is to store an identical master secret key in all nodes. Any pair of nodes can use this master secret key to securely establish a new pairwise key. But if a single node is compromised, the security of the key establishment is compromised. At the other extreme, one might consider a key pre-distribution scheme in which each sensor stores $N - 1$ keys (where N is the number of nodes in the network), each of which is known to only one other sensor node. If a subset of nodes is compromised, security properties of the group of all other nodes are still guaranteed. Unfortunately, this scheme is impractical for groups with large member number, since it requires extremely much memory. Moreover, this scheme does not easily allow new nodes to be added to a sensor network because the existing nodes will not have the new nodes' keys.

A number of key pre-distribution schemes to solve the above mentioned problem are proposed, some of those can be found in [18, 31, 36, 101]. In scheme in [101], each sensor node receives a random subset of keys from a large key pool before deployment. To agree on a key for communication, two nodes find a common key (if any) within their subsets and use it as shared secret key. Now, the existence of a shared key between a particular pair of nodes is not ensured but is instead probabilistically guaranteed (this probability can be tuned by adjusting the parameters of the scheme).

3.8.2. Key transport schemes

In key transport schemes, the group manager or key server generates secret keys and transports them to the other participants. A prior shared key or PKI should be assumed so that the new generated keys could be transported securely. The member that generates keys bears higher computation costs than others. A number of schemes falling into this class have been designed, e.g., [12, 26, 54, 84, 85, 86, 88]. Some of these have lead to implementations or standards, e.g., the Kerberos[68, 69], the ANSI X9.17 [1], and the KryptoKnight [72] which rely on symmetric key technology, or the ISO-CCITT X.509 Directory Authentication scheme [2], which use public key technology.

The Kerberos scheme may be summarized as follows: Kerberos is based on the Needham-Schroeder protocol [84]. It makes use of a trusted third party, denoted a Key Distribution Center (KDC), which consists of two logically separate parts: an Authentication Server (AS) and a Ticket Granting Server (TGS). Kerberos works on the basis of „tickets“ which serve to prove the identity of users. Kerberos maintains a database of secret keys. Each entity on the network, whether a client or a server, shares a secret key with Kerberos server. Knowledge of this key serves to prove an entity's identity. For example, when a client A needs a key to communicate with a server B, A must obtain the desired key from the KDC prior to communication with B. While in ANSI X9.17 A must contact B first, and let B get the necessary key from the KDC.

The KryptoKnight provides the same set of services as Kerberos. However, the authentication and key distribution protocols used in KryptoKnight are minimal, flexible and scalable. They are suitable for various network settings, such as small devices with limited computing power and low-level networking environments. Unlike Kerberos, KryptoKnight uses a hash function for authentication.

3.8.3. Group key agreement

In key agreement schemes every participant contributes somewhat that will be used to compute common secret key. Schemes falling into this class can be further classified, according to the based technology: symmetric key and asymmetric key cryptography. However, the

Key agreement schemes assume typically some sort of public-key infrastructure which are most not present in ad hoc networks. The solution of this problem in ad hoc networks will be handled in TFAN protocol suite in Section 5.

3.8.3.1. Group communication semantics and support

Without a reliable group communication platform, it is difficult to agree on the group key. Hence a secure group key agreement protocol depends on the underlying group communication semantics for protocol message transport and strong membership semantics.

Two strong group communication systems are mostly used: Extended Virtual Synchrony (EVS) [7, 73] and View Synchrony (VS) [3] group communication system (GCS). Both provide the following services:

1. Group members see the same set of messages between two sequential group membership events;
2. The sender's requested message order is preserved. VS provides a stricter service, while EVS implementations are generally more efficient.

The main difference between these two systems is how they maintain view changes. In an EVS group communication system, view changes occur without clients' involvement, while in a VS system a new view can be installed only with the permission of the client. Due to the intervention of the client, it is impossible to handle truly view changes in „real-life“. What it can do is to isolate a client from these changes and not allow new members or their messages to be presented to the client without its permission.

In a VS group communication system, when an underlying view change occurs, due to *join*, *leave*, *partition* or *merge* (Section 3.8.3.2), group communication system creates a request message indicating that the

current view is outdated and should be replaced with the new view. A member gives the group communication system the permission to install the new view. Till this moment, the member is still able to receive messages from the survived members, but is not allowed to send any messages to the group.

In an EVS group communication system, when a membership change occurs the new view is delivered to the members with the normal stream of messages.

Generally the EVS model is faster and more scalable with respect to view changes, especially for simple group changes, e.g. *join* and *leave*. However, in EVS group communication systems, the member has less control over what is taking place and therefore in general it is harder to program algorithms under EVS than under VS. For example, in EVS group communication systems, until the message is delivered back to the sender, the sender gains no information about which other members might receive its messages. Since the view may change very frequently in EVS group communication systems, the algorithm must consider all combinations of different view changes. However, in VS group communication systems, the member knows exactly which other clients might receive its message upon sending it. Also, view changes only occur as fast as the applications wants them to, although several underlying memberships may be collapsed into one. So they are generally easier to handle and track.

Due to the above mentioned properties of EVS and VS group communication systems, an implementation of any distributed fault-tolerant group key agreement protocol requires VS group communication systems. This is because implementing group key agreement on top of EVS would require the key agreement protocol to incorporate and implement semantics identical to those of VS in order to correctly keep the order of sent.

3.8.3.2. Group membership events

In a dynamic group, the group's view changes over the time, sometimes very frequently. At some time some want to form a group for some purpose. After the forming, some may join in the group and some members may leave the group freely or due to some faults. A powerful group key agreement should consider this dynamic behavior using the underlying group communication system.

In the following we distinguish these events, according to the phase when the events occur, between **Setup operation** and **Dynamic group operation**.

Setup operation Setup operation takes place at the time of group formation. In this phase some agreed protocol is used to form a group, namely, to setup the relationship among all members. There are a number of protocols to identify on the architecture, e.g. [8, 41] for ad hoc networks. In group key agreement schemes depending on key tree, the nodes can be sorted according to some criteria, and form a tree.

As mentioned in Section 3.8.3.1, group key agreement prepares secure group communication, and the protocol overhead in this phase should be minimized as much as possible. Some security properties may depend on the underlying group communication system.

Dynamic group operation Once a group is formed, it is ready to be used by other applications. However, in a dynamic group, the views of the underlying group communication system are dynamic. Hence initial group key agreement is only one part. A comprehensive group key agreement scheme must also be able to handle adjustments to group secrets subsequent to all membership change operations in the underlying group communication system.

All member operations taking place in this phase can be further classified into two types, addition and exclusion of members. Additive events include addition of single and multiple members, while exclusion events include deletion of single and multiple members.

Another most commonly used classification is according to number of members. The operations are distinguished between single and multiple member operations.

All operations are illustrated in Figure 3.9.

Single member operations Simple member operations include the addition and deletion of single member, denoted by **join** and **leave** respectively.

1. **Join**: *Join* occurs when a potential member wishes to join in an existing group for some reason, such as to share document, to join a conference. The member addition is always performed multi-laterally or, at least, mutually. Suppose that A wishes to join the group G. A sends its request to group G, after the arbitration among the current members, G accepts or rejects A's request.
2. **Leave**: *Leave* occurs when a member wishes to leave the group, or is forced to leave the group. In the former case, member deletion is mutual, while in the later case unilateral. There may be various reasons for a member to be forced to leave a group, such as involuntary disconnect or forced exclusion. A group key agreement scheme does not need to bother about reasons. It leaves the underlying group communication system to handle them. However, it must adjust the group key on this change.

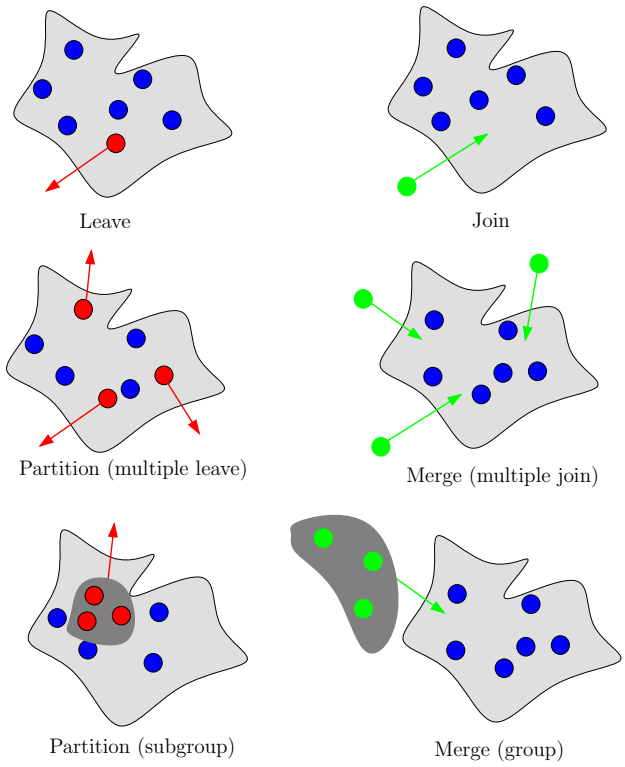


Figure 3.9.: Dynamic group operations

Multiple member operations Multiple member operations are group addition and group exclusion operations, denoted by **group merge** and **group partition**, respectively. The operations of single members are the special cases of multiple member operations.

Group partition can be considered of having the following variants.

1. **Single leave**: It is the special case, the same as **leave** defined above.
2. **Multiple leave**: Multiple members leave the group without forming own subgroup.
3. **Subgroup leave**: All members leave the group and form own subgroup.

We consider multiple leave as group partition, when leaving and remaining members can be organized to two subgroups, respectively. The single leave can be considered to be the leave of a subgroup with only one member.

After **group partition**, member's group view is relative to the subgroup it belongs to. For example, there is a group G consisting of 6 members M_1, M_2, M_3, M_4, M_5, and M_6. Now the group G is split into two smaller groups: group G_1 with members M_1, M_3, M_5, and group G_2 with members M_2, M_4, and M_6. All members in G_1 see M_2, M_4, and M_6 as leaving members, while all members in G_2 see M_1, M_3, and M_5 as leaving members.

A **group partition** can take place for several reasons, summarized in the following [58]:

1. Network failure - occurs when a network event causes dis-connectivity within the group. Consequently, a group is split into fragments some of which are singletons while others (those that maintain mutual connectivity) are sub-groups.
2. Explicit (application-driven) partition - this occurs when the application decides to split the group into multiple components or simply exclude multiple members at once.

Similarly, group merge has the following variants.

1. **Single join**: It is the special case, the same as **join** defined above.
2. **Multiple join**: Multiple potential members wish to join an existing group and, moreover, these potential members do not already form a group of their own.
3. **Group join**: Another existing group wishes to merge into the current group to form a super-group.

It seems that multiple variants of group merge exist. However, we consider multiple join as group join, when the potential members are formed first to a group and then this new group will be merged into the current group. The join can be considered to be the merge of a group with only one member into the current group.

The merge of more than two groups can be considered to be subsequent merging of two groups. Unlike in **group partition**, all members have the same view after **group merge** is handled.

As discussed above, due to some reasons a group is split into some smaller groups. Correspondingly, a group merge can be either voluntary or involuntary [58]:

1. Network fault heal - occurs when a network event causes previously disconnected network partition reconnect. Consequently, all groups (and there might be more than two groups) formed after partition are merged into a single group.
2. Explicit (application-driven) merge - occurs when the application decides to merge multiple pre-existing groups into a single group. (The case of simultaneous multiple-member addition is not covered.)

Due to the properties of ad hoc networks (Section 1), network failures and network fault heals are both common and expected. Hence dealing with group partitions and merges is a crucial component of group key agreement for ad hoc networks.

Group Key Refresh It is desirable to refresh the key periodically, even if the group memberships are not changed. There are two main reasons, one is to limit exposure due to loss of group session keys, the other is to limit the amount of ciphertext available to cryptanalysis for a given group session key. This makes it important for the key refresh protocol not to violate key independence. Additionally, note that the loss of a member's key share can result in the disclosure of all the session keys to which the member has contributed with this share. Therefore, not only session keys, but also the individual key shares must be refreshed periodically.

3.8.3.3. Cryptographic Properties

One of the most critical security properties of group key agreement schemes is key freshness. That's the main purpose of the group operation **key refresh** (Section 3.8.3.2). Key freshness is defined as follows [58]:

A key is fresh if it can be guaranteed to be new, as opposed to possibly an old key being reused through actions of either an adversary or authorized party.

Furthermore, the keys should be independent, which is ensured by both **Forward Secrecy** and **Backward Secrecy**. However, sometimes it is not desirable to provide key independence, but only some weaker form of it, called **Weak Key Independence**, which requires **Weak Forward Secrecy** and **Weak Backward Secrecy**.

One basic security property is that the key can only be known to the involved parties. According to above discussed requirements, Kim *et. al.* define in [58, 56] the following four security properties of a group key agreement.

Definition 3.1 *Assume that a group key is changed m times and the sequence of successive group keys is $\mathcal{K} = (K_0, K_1, \cdots, K_m, \cdots)$.*

1. **Computational Group Key Secrecy**: This is the most basic property. It guarantees that it is computationally infeasible for any passive adversary to discover any group key $K_i \in \mathcal{K}$ for all i.
2. **Decisional Group Key Secrecy**: It ensures that there is no information leakage other that public blinded key informations.
3. **Key Independence** is the strongest property. It guarantees that a passive adversary who knows a proper subset of group keys $\widehat{\mathcal{K}} \subset \mathcal{K}$ cannot discover any other group key from $\bar{\mathcal{K}} = \mathcal{K} - \widehat{\mathcal{K}}$. Key independence can be decomposed into the following two security properties.
   a) **Forward Secrecy**[5]: It guarantees that a passive adversary who knows a contiguous subset of old group keys, e.g. $\{K_i, K_{i+1}, \cdots, K_j\}$, cannot discover any subsequent group key K_k for all i, j, k where $0 \leq i \leq j < k$.
   b) **Backward Secrecy**: It guarantees that a passive adversary who knows a contiguous subset of group keys, e.g. $\{K_j, K_{j+1}, \cdots, K_k\}$, cannot discover any preceding group key K_i for all i, j, k where $i < j \leq k$.

As mentioned in the beginning of this section, some group key agreement schemes use **weak key independence**, which bases on the following two security properties [110]:

1. **Weak Backward Secrecy**: Previously used group keys must not be discovered by new group members.
2. **Weak Forward Secrecy**: New keys must remain out of reach of former group members.

In **Weak Key Independence** the adversary is restricted to be a current or former group member, while **key independence** additionally includes the cases of inadvertently leaked or otherwise compromised group keys.

In certain cases, **(Weak) Key Independence** may be undesirable. For example a new member may be allowed to know the former secret key to get access to the old information. Some group key agreement schemes may allow to specify the security level based on the local security policy.

Secret keys, such as for encryption and for authentication, are derived commonly from the agreed group key. If the deriving function is not one-way, an attacker may get one of the secret keys according to this definition.

[5] Not to be mixed with Perfect Forward Secrecy (PFS). PFS is defined as follows: For a key-establishment protocol, the compromisation of a session key or long-term private key after a given session does not cause the compromisation of any earlier session.

3.8.3.4. Complexity Analysis

Above we have discussed the security properties of group key agreement schemes. Important is also their complexity, namely performance costs. Sometimes trade-off between complexity and security is required, so that the schemes are suitable to particular environments. Three of the most important criteria are computation costs, communication costs, and memory space costs.

Computation costs To achieve exact computational costs is impossible and also impracticable. Different implementations of an identical group key agreement scheme bring different results. Even the same implementation can not guarantee same result in different environments.

However, we can estimate the computation costs by identifying the expensive and time-critical operations. We can ignore the concrete operation time, and only compare the number of such operations. Some computations can be pre-performed before protocol run or computed when the system is idle. Hence only the operations which have to be performed sequentially should be considered. Those operations are denoted by **serial operations** in [56].

Communication costs The communication costs of a group key agreement depends clearly on the topology and properties of the network and the group communication system used. The critical aspects are primarily latency and bandwidth. Additionally the communication costs are implementation-dependent. Hence to achieve fixed communication costs is impracticable. However, we can estimate them by considering the following costs.

1. **Number of rounds**: this affects serial communication delay. As the number of rounds grows, the communication delay and the probability of message loss or corruption are increased.
2. **Total number of messages**: as the number of messages grows, the probability of message loss or corruption, and the delay are increased.
3. **Number of broadcasts and unicasts**: a broadcast operation is much more expensive than a unicast one, since it requires much more acknowledgments with the group communication system. The number of broadcasts should be minimized.
4. **Length of message**: It is critical in networks with limited bandwidth, such as ad hoc networks.

Memory costs In general network environments, devices are usually not limited in their memory space. However in networks with limited memory such as ad hoc networks, memory costs are one of the most crucial criteria. Memory costs are defined as the size of the the information required to perform the group key agreement protocol. For simplicity, we measure memory costs with the number of keys and randoms in the above mentioned information.

4. Analysis of Protocols in Literature

Since the publication of two-party Diffie-Hellman (DH) key exchange in 1976, various solutions have been proposed to extend Diffie-Hellman key exchange to multi-party key agreement, such as [8, 9, 14, 17, 46, 56, 57, 58, 107, 108, 109, 110, 111, 122, 124, 126].

Most notable and best known among those proposals are the protocols by Ingemarson *et al.* [46] (ITW), Burmester and Desmedt [17] (BD), Becker and Wille [13] (2^d-cube and 2^d-octopus), Steiner *et al.* [108, 109, 111] (CLIQUES), and Kim *et al.* [57] (STR), [58] (TGDH).

The last three protocol suites, CLIQUES, STR and TGDH, are studied in this chapter. They are suitable for dynamic peer groups (DPGs), since they provide protocols to setup the group, and to handle dynamic group operations. Additionally, we improve STR and TGDH to save memory costs, message size, and computation costs. All communication channels in these protocol suites are public but authentic. The latter means that all messages are digitally signed by the sender using some sufficiently strong public key signature method such as RSA or DSA. For sake of briefness, we does not consider such authentication in the rest of this chapter for protocol suites CLIQUES, STR, TGDH, improved STR, and improved TGDH.

With respect to ad hoc networks, we introduce Asokan-Ginzboorg protocol, which is developed in [8]. Asokan-Ginzboorg bases on 2^d-cube and 2^d-octopus, and is extended with password authentication and ability to deal with faulty events, so that it is suitable for static ad hoc networks.

The following abbreviations will be used in the tables which summarize the complexity characteristics.

Rounds:	The total number of rounds
Exps/M_i [1]:	The total number of modular modular exponentiations computed by member M_i
All Exps:	Total number of modular exponentiations of all members
Serial [2] Exps:	The serial number of modular exponentiations
Ucasts:	The total number of unicast messages of all members
Bcasts:	The total number of broadcast messages of all members
Ucast Size:	The cumulative unicast message size
Bcast Size:	The cumulative broadcast message size
Msg Size:	The cumulative unicast and broadcast message size

Additionally, for description of protocol suites, the symbols in Figure 4.1 are used for all protocol suites through this and next chapters. Additionally we define *key-path* and *co-path* for STR and TGDH. A *key-path* of member M_i is defined to be the path from M_i to the root, and M_i's *co-path* is the set of siblings of each node in the key-path of member M_i. Figure 4.2 lists some symbols for protocol suites STR and TGDH. Other specific symbols for each protocol suite can be found in the respective section.

The rest of this chapter is organized as follows: Section 4.1 describes Asokan-Ginzboorg and the basic protocols 2^d-cube and 2^d-octopus. CLIQUES is introduced in Section 4.2. STR is studied in Section 4.3, while TGDH in Section 4.4. The comparison of the complexity characteristics of above mentioned protocols is in Section 4.5, according to the criteria described in Section 3.8.3.4.

[1] In protocols suites TGDH and μTGDH M_i is replaced by $\langle l, v \rangle$ which identifies a member's position.

[2] The serial cost assumes parallelization within each protocol round and represents the greatest cost incurred by any participant in a given protocol.

[3] n denotes the number of members in the initial group in operations *join*, *leave* and *partition*, and n means the members in the group with highest key tree in case of merge.

[4] h denotes the height of the initial tree in operations *join*, *leave* and *partition*, and h means the height of highest tree in case of merge.

n	number of protocol participants (group members) [3]
i, j, c, s	indices of group members
M_i	i^{th} group member; $i \in \{1, 2, \cdots, n\}$.
M_*	all group members
G	cyclic algebraic group
$\|G\|$	order of G
g	exponentiation base; generator of G
p, q	large prime number
$r_i, \widehat{r_i}$	secret values from $\mathbb{Z}_{\|G\|}$, randomly generated by M_i
K_G	group key shared among members in a group

Figure 4.1.: Notations

h	height of the tree [4]
$\widehat{h}$	height of the updated tree
T_i	T_i's view of the key tree
$\widehat{T_i}$	M_i's modified tree after membership operation
KEY_i^*	set of keys (and session randoms in STR) in M_i's key-path
BK_i^*	set of M_i^*'s blinded keys (and blinded session randoms in STR)
$KpBK_i^*$	set of blinded keys (and blinded session randoms in STR) in M_i's key-path
$CoBK_i^*$	set of blinded keys (and blinded session randoms in STR) in M_i's co-path

Figure 4.2.: Notations for STR and TGDH

4.1. Asokan-Ginzboorg Protocol Suite

Becker and Wille [13] have proposed the hypercube protocol suite, which shows how multi-party Diffie-Hellman key exchange can be done efficiently. Two of them, 2^d-cube and 2^d-octopus are introduced in Section 4.1.1. Asokan and Ginzborg [8] have extended them by

- adding the password authentication to authenticate the partner,
- suggesting an algorithm to find non-faulty partner to make the protocol suite fault-tolerant.

We call the extended protocol suite **Asokan-Ginzboorg**.

One of the major advantages of the protocols described in this section is the lower memory cost. Each member needs only to know the agreed key, so the *memory cost* is only 1. However, both hypercube and Asokan-Ginzboorg can only handle key agreement for static groups. It is not clear, how they can be extended to deal with the dynamic group operations (Section 3.8.3.2).

Besides the symbols in Figure 4.1, K_{AB} denotes the key shared among members A and B.

4.1.1. 2^d-cube and 2^d-octopus protocol suites

We consider here two protocols in hypecube protocol suite, 2^d**-cube** and 2^d**-octopus**. In 2^d**-cube** protocol the 2^d participants are arranged in a d-dimensional hypercube, denoted *d-cube*. The participants are identified with the vectors of the d-dimensional vector space $\mathbb{F}_{2^d}$.

We take a 2-cube as example, the protocol process is shown in Figure 4.3. There are four participants, A, B, C, and D with 2-bit addresses 00, 01, 10, and 11, respectively. Two rounds are needed to share a secret among them. In the first round, A and B perform a two-party DH key exchange and compute then the secret $K_{AB} = g^{r_A r_B}$. C and D do the same and get the secret $K_{CD} = g^{r_C r_D}$. In next round, A and C perform a

two-party DH key exchange, they use the secret key gained in the previous round, namely K_{AB} and K_{CD}, respectively. Now A and C know the shared secret group key $K = g^{K_{AB}K_{CD}} = g^{(g^{(r_A r_B)} g^{(r_C r_D)})}$. The same do B and D. After two rounds, all four participants have the same key $K_G = g^{(g^{(r_A r_B)} g^{(r_C r_D)})}$.

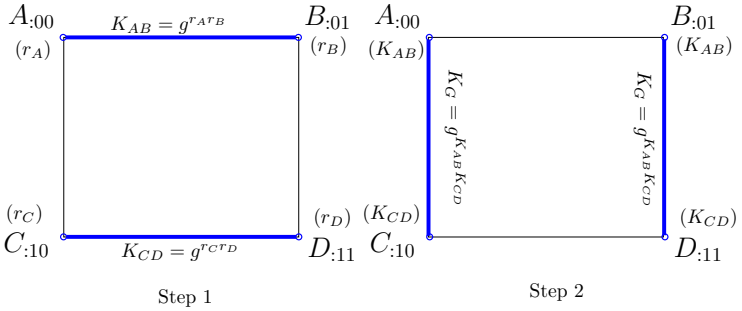


Figure 4.3.: Key exchange's process of d-cube ($d = 2$)

Assume that there are $n = 2^d$ participants. Each participant is identified with a d-bit address with decimal value from 0 to $2^d - 1$. The protocol proceeds in d rounds. In round j, the participant with address A runs a two-party DH key exchange with the node with address $A \oplus 2^{j-1}$. In other words, in round j, two nodes whose addresses are only different at the $(j-1)^{th}$ least significant bit perform the two-party DH key exchange. After j rounds, the nodes who have the identical $n - d$ greatest significant bits of addresses have the same key, and there are 2^{d-j} key pairs. After d rounds all participants have the same key.

The above described protocol works only when the number of participants is a power of 2. For generalized case, a so called 2^d**-octopus** protocol is introduced in the same article of Becker and Wille. All n ($2^d < n < 2^{d+1}$) participants are identified with $d+1$ bits addresses with decimal value from 0 to $n-1$. The first 2^d participants play the role of central controllers. The rest $n - 2^d$ participants are distributed among the central controllers as their wards. A participant with address $A > 2^d$ is the ward of central controller with address $A - 2^d$.

In the first phase, the central controllers perform two-party DH key exchange with their wards. The controllers engage then in a d-cube protocol run using the information gathered in the first phase. In the third and last phase, the key gained in the second phase is distributed to the wards encrypted with the key gained in the first phase as follows. One possibility is using ElGamal Encryption. Assume that the controller and its ward have agreed in the first phase K_{AB} and the public value $PK = g^{K_{AB}}$, and the group key to be distributed is K_G. The controller chooses randomly an element t from G, computes then $C_1 = g^t$ and $C_2 = K_G \cdot (g^{K_{AB}})^t$, and finally sends C_1 and C_2 to the ward. The ward gets then the group key by dividing C_2 by $(C_1)^{K_{AB}}$. More efficient is to use symmetric encryption using the K_{AB} derived symmetric key.

An example is given below. Suppose 7 participants $P_0, P_1, \cdots, P_6$, with 3-addresses 000, 001, $\cdots$, 110, respectively, wish to agree on a secret with hypercube protocol. P_0, P_1, P_2, and P_3 are central controllers. The first three have their wards P_4, P_5, and P_6, respectively. In the first phase, P_0 and P_4, P_1 and P_5, and P_2 and P_6 execute two-party DH key exchange, P_3 do nothing. In the second phase, P_0, P_1, P_2, and P_3 run d-cube protocol to share a secret K. Finally K will be distributed by P_0 to P_4, by P_1 to P_5, and by P_2 to P_6. The process is illustrated in Figure 4.4.

The complexity characteristics of protocols 2^d-cube and 2^d-octopus are summarized in Table 4.1.

4.1.2. Asokan-Ginzboorg protocol suite

In 2^d-cube and 2^d-octopus no authentication of the participants are performed. One solution is to use digital signatures. This requires, in order, the existence of PKI. However, in some networks e.g. ad hoc networks, no

[5]The data here are for the case using ElGamal encryption to distribute the group key by central members to their respective wards. Each central controller needs to compute two modular exponentiations. And each ward one for the decryption. However, by using symmetric encryption, the computation costs of encryption can be neglected.

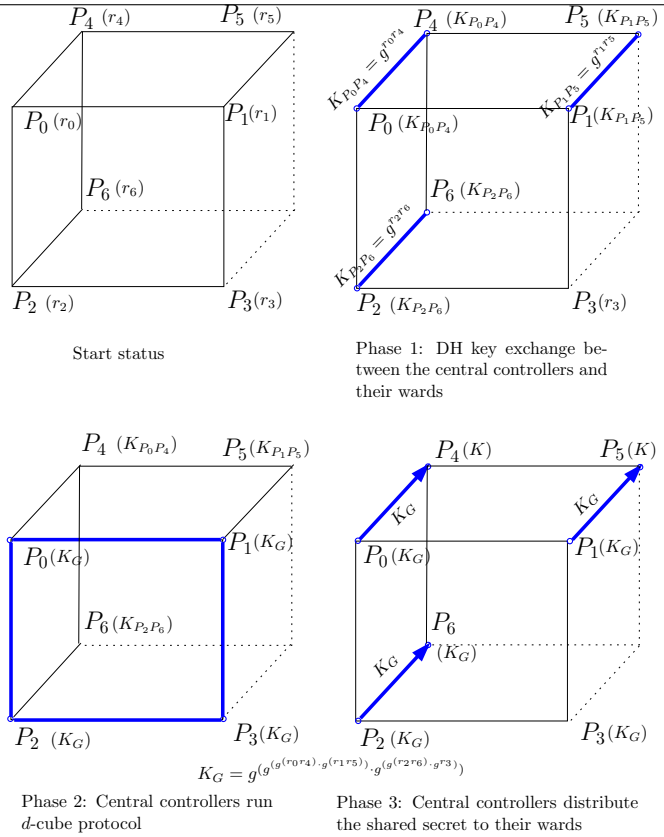


Figure 4.4.: Key exchange's process of 2^d *octopus* ($d = 2, n = 7$)

such infrastructure can be assumed. One solution for group key agreement for such networks was proposed by Asokan and Ginzboorg [8]. The four move two-party Diffie-Hellman key exchange (Section 3.7.3) can be applied instead of each two-party DH key exchange in 2^d-cube and 2^d-octopus.

Due to network faults, a participant may discover during any round that the key exchange with its chosen partner failed. No mechanisms of 2^d-cube and 2^d-octopus can handle such failures. This problem is solved in the extended version, Asokan-Ginzboorg. It works well even if there are such failures. If a participant discovers that its chosen partner in a given round is faulty, it select other potential partners until a non-faulty one is found. This algorithm suggests the „closer" partners before the more distant ones, in term of the euclidean distance between the participants' addresses.

The process finding such a partner is described in algorithm 4.1. For more information please refer to [8].

Algorithm 4.1 (Finding a non-faulty partner in hypercube protocol)

Input: Participant A's address I, round number n.
Output: Success and B if a non-faulty partner B of A at round n exists, else failure.

1. *Let* $mask = 00 \cdots 01$;
2. $mask = mask << (n-1)$;
3. $B = I \oplus mask$;
4. $mask = mask >> 1$;
5. *Call sub-algorithm two_pary_dhke with B and $mask$ as input. Return the result of this call.*

. . . . . .

The sub-algorithm two_pary_dhke to find a partner and perform two-party DH key exchange is described

	2^d-cube	2^d-octopus ($n \neq 2^d$)
Rounds	d	$d+2$
Ucasts	nd	$3(n-2^d)+2^d \cdot d$
Bcasts	0	0
Exps/M_i	$2d-1$	$2d+3$ for central controllers 2 for wards [5]
Ucast size	nd	$3(n-2^d)+2^d \cdot d$
Bcast size	0	0
All Exps	$n(2d-1)$	$2^d \cdot (2d+1)$
Serial Exps	$2d-1$	$2d+3$ [5]
Remark: n:number of members, $d = \lfloor \log_2 n \rfloor$		

Table 4.1.: Complexity of protocol 2^d-*cube* and 2^d-*octopus*

below:

algorithm two_party_dhke(candidate, mask)

1. *if* $mask > 0$*, go to next step; otherwise attempt running the 2-party DH key exchange, return the result (*$success$ *with* $candidate$*, or* $failure$*);*
2. $new\_mask = mask >> 1$*;*
3. $result = two\_party\_dhke(candidate, new\_mask)$*;*
4. *if* $result = success$*, return* $sucess$ *and* $candidate$*; otherwise go to next step;*
5. $new\_candidate = candidate \oplus mask$*;*
6. *return* $two\_party\_dhke(new\_candidate, new\_mask)$.

Unlike the basic hypercube protocols, the number of all sub-rounds will be measured for communication cost. When the number of participants n is not a power of 2, $2^d < n < 2^{d+1}$, the protocol using algorithm 4.1 consumes $d+1$ rounds. The number of sub-rounds can be at most 2^d, when $n = 2^d + 1$. However, by labeling the participants differently the complexity can be significantly reduced. Suppose, there are m faulty or missing participants, the even numbers from 0 to $2(m-1)$ are associated with the faulty or missing ones. The rest of addresses are assigned in sequence to the remaining actual participants. So that each remaining actual participant can find a non-fault partner to process the two-party DH key exchange with maximal two attempts. With this relabeling technique, the worst case complexity is $2(d+1)$ rounds. And the costs for relabeling the participants should also be considered.

In the following, we consider a scenario described in [8] without relabeling the participants. There are n participants, where $2^d < n \leq 2^{d+1}$, and m of those are faulty, where $2^k \leq m < 2^{k+1}$. We assume further that $2^k - 1$ of those faulty participants are in a k-cube $C1$, and the rest faulty ones are in another k-cube $C2$. $C1$ and $C2$ will process in round $k+1$ key exchange.

Then the number of sub-rounds in main round i with $1 \leq i \leq k+1$ is at most 2^{i-1}. In round $k+1$, the number of faulty participants in cube C_1 is $2^k - 1$. Some participant in cube C_2 may try all the faulty players in C1 before finding the only non-faulty participant. Thus the maximum number of sub-round for main round $k+1$ is k. In round i with $1 \leq i < k$, the only non-faulty participant will have to try all possible partners before giving up.

From main round $k+2$ to $d+1$, the maximal sub-rounds is $m+1$. This is because in each of those rounds, there is always one sub-cube with m faulty participants. The same non-faulty participant may select each of the m faulty participants in sequence before being able to complete its round exchange, resulting in $m+1$ rounds.

So at most $\sum_{i=1}^{k+1} 2^{i-1} + (d-k)(m+1) = 2^{k+1} - 1 + (d-k)(m+1)$ rounds are needed. This is also the upper bound, regardless of the positions of the m faulty participants. Correspondingly at most $2^{k+1} - 1 + (d-k)(m+1)$ unicast messages are needed for the non-faulty participants. The $n-m$ participants are non-faulty. So at most $(n-m)(2^{k+1} - 1 + (d-k)(m+1))$ unicast messages will be sent during the key agreement. Every non-faulty participant will process $d+1$ successful key exchange, and at most $2^{k+1} + (d-k)(m+1) - d - 2$ failing key exchanges. Assume that exponentiation will be only computed in a successful DH key exchange, then d exponentiations are computed by each non-faulty participant. In worst case, in each main round, there are at most *max(n - m, number of sub-rounds)* serial exponentiations. The total number of serial exponentiations can be computed as follows:

$$min(n-m, \sum_{i=1}^{k+1} max(n-m, 2^{i-1}) + (d-k) \cdot max(n-m, m+1)).$$

The worst case complexity characteristics are summarized in Table 4.2.

rounds	$2^{k+1} - 1 + (d-k)(m+1)$
UCast MSGs	$(n-m)(2^{k+1} - 1 + (d-k)(m+1))$
BCast MSGs	0
EXP/member	$2d+1$
$\sum$ UCast	$(n-m)(2^{k+1} - 1 + (d-k)(m+1))$
$\sum$ BCast	0
$\sum$ EXP	$(n-m)(2d-1)$
Serial EXPs	min($n-m$, $\sum_{i=1}^{k+1}$max(n-m, 2^{i-1}) + (d-k)·max(n-m, m+1))
Remark: n: number of members, m: number of faulty members, $2^d < n \leq 2^{d+1}$, $2^k \leq m < 2^{k+1}$	

Table 4.2.: Complexity of Asokan-Ginzboorg (worst case)

4.2. Cliques Protocol Suite

In [109, 108, 111] Steiner *et al.* present CLIQUES, a complete family of protocols for key agreement in dynamic peer groups (DPGs). This protocol suite considers initial key agreement, key refresh and membership changes (Section 3.8.3.2). The protocol suite uses Diffie-Hellman key exchange technique.

Since there are various versions of CLIQUES, for sake of clarity, the protocol CLIQUES introduced below is from [111]. We present only the computation process of the group key. For the security proof and other issues please refer to [111].

Assume that we have a group of n members. Each member M_c must know $(g^{\prod(r_m|m\in\{1,\cdots,n\}\setminus\{j\})}|j \in \{1,\cdots,n\} \setminus \{c\})$, $g^{\prod(r_m|m\in\{1,\cdots,n\}\setminus\{c\})}$, and $g^{\prod(r_m|m\in\{1,\cdots,n\})}$. Additionally it must know its own random r_c and the group key. So the *memory costs* of each member are 2 keys and $n+1$ public keys.

4.2.1. IKA protocol

CLIQUES uses two alternative IKA protocols IKA.1/GDH.2 and IKA.2/GDH.3 [111]. Compared to IKA.1, IKA.2 is optimized for large groups or groups having limited resources entities. In IKA.1 the member M_i needs to compute $i+1$ exponentiations. However, in IKA.2, the number of exponentiations are significantly reduced, 4 for M_i with $i \leq n-2$, 2 for M_{n-1}, and n for M_n. Hence we have only interest in IKA.2, for information about IKA.1 please refer to [111]. The process of IKA.2 is illustrated in Protocol 4.1.

Protocol 4.1 (CLIQUES IKA.2)

1. *Round i,* $i \in \{1, \cdots, n-2\}$*:*

M_i $\xrightarrow{\quad g^{\prod(r_m | m \in \{1,\cdots,i\})} \quad}$ M_{i+1}

2. *Round n - 1:*

M_* $\xleftarrow{\quad g^{\prod(r_m | m \in \{1,\cdots,n-1\})} \quad}$ M_{n-1}

3. *Round n:*

M_i $\xrightarrow{\quad g^{\frac{\prod(r_m | m \in \{1,\cdots,n-1\})}{r_i}} \quad}$ M_n

4. *Round n + 1:*

M_* $\xleftarrow{\quad g^{\frac{\prod(r_m | m \in \{1,\cdots,n\})}{r_j}} | j \in \{1, \cdots, n\}) \quad}$ M_n

5. *Every member computes the group key.*

After running IKA.2 protocol, a secret key $K_G = g^{\prod_{i=1}^{n} r_i}$ is shared among the group. Some examples for IKA.2 and the below mentioned protocols are given in chapter 7 in [108]. For the sake of clarity, we refer reader to [108] for such examples. The complexity characteristics of IKA.2 protocol is given in Table 4.3.

Rounds	$n+1$
Ucasts	$2n-3$
Bcasts	2
Ucast size	$2n-3$
Bcast size	$n+1$
EXPs/M_i	3 for $i \in \{1, \cdots, n-2\}$,
	2 for $i = n-1$, n for $i = n$
All Exps	$5n-6$
Serial Exps	$2n+1$

Table 4.3.: Complexity of CLIQUES IKA.2 protocol

4.2.2. Join protocol

To handle a $join$ event, a group controller is needed. It can be any member in the group. The group controller has usually higher performance capability than other members. Let M_c be the group controller, and let M_{n+1} be the potential member. The protocol is illustrated in Protocol 4.2.

In first step, M_c chooses a new random exponent $\widehat{r_c}$. Then r_c is replaced by $\widehat{r_c} r_c$ mod $|G|$ to achieve key independence. The message will be sent by M_c to the new member M_{n+1}. M_{n+1} generates randomly its secret r_{n+1} and embeds r_{n+1} in the message which it then broadcasts to all members in group.

Protocol 4.2 (CLIQUES join)

1. *Round 1: The group controller* M_c
   - *updates its secret to be embedded in the new message,*
   - *unicasts this message to* M_{n+1}.

M_c $\xrightarrow{\quad (g^{\widehat{r_c} \prod(r_m | m \in \{1,\cdots,n\} \setminus \{j\})} | j \in \{1, \cdots, n\}), \; g^{\widehat{r_c} \prod(r_m | m \in \{1,\cdots,n\}} \quad}$ M_{n+1}

2. *Round 2: The new member M_{n+1}*

   - *generates a secret r_{n+1},*
   - *embeds r_{n+1} in a new message,*
   - *broadcasts the message.*

   M_* $\xleftarrow{(g^{\widehat{r_c}} \prod(r_m | m \in \{1,\cdots,n+1\} \setminus \{j\}) | j \in \{1,\cdots,n+1\})}$ M_{n+1}

3. *Every member computes the group key.*

After running join protocol, all members compute the new group key: $K_{G_{new}} = g^{\widehat{r_c} \prod_{i=1}^{n+1} r_i}$. The complexity characteristics of **join** protocol is given in Table 4.4.

Rounds	2
Ucasts	1
Bcasts	1
Ucast size	$n+1$
Bcast size	$n+1$
Exps/M_i	$n+1$ for group controller and new member, 1 for all others
All Exps	$3n+1$
Serial Exps	$2n+1$

Table 4.4.: Complexity of CLIQUES join protocol

4.2.3. Leave protocol

The leave protocol in CLIQUES is relatively simple. It needs only one round. Let M_c be group controller, and M_l be the member which leaves the group. First, M_c updates its secret r_c with $\widehat{r_c} r_c$. M_c constructs then a broadcast message by embedding its new secret. Finally M_c broadcasts the message to the group. The process is illustrated in Protocol 4.3.

Protocol 4.3 (CLIQUES leave)

1. *Round 1: The group controller M_c*

   - *updates its secret,*
   - *embeds the new secret in a new message,*
   - *broadcasts the message.*

   M_c $\xrightarrow{\begin{array}{c}(g^{\widehat{r_c} \prod(r_m | m \in \{1,\cdots,n\} \setminus \{j\})} | j \in \{1,\cdots,n\} \setminus \{c,l\}), \\ g^{\prod(r_m | m \in \{1,\cdots,n\} \setminus \{c\})}\end{array}}$ M_*

2. *Every member computes the group key.*

After running leave protocol, all remaining members compute the new group key: $K_{G_{new}} = g^{\widehat{r_c} \prod_{i=1}^{n} r_i}$. Although the contribution r_l is still factored into the new group key, the left member M_l is unable to compute the new group key, due to the absence of the subkey $g^{\widehat{r_c} \prod(r_m | m \in \{1,\cdots,n\} \setminus \{l\})}$.

The complexity characteristics of leave protocol is given in Table 4.5.

Rounds	1
Ucasts	0
Bcast	1
Ucast size	0
Bcast size	$n-1$
Exps/M_i	$n-1$ for group controller,
	1 for all other remaining members
All Exps	$2n-3$
Serial Exps	$n-1$

Table 4.5.: Complexity of CLIQUES leave protocol

4.2.4. Merge/Multiple join protocol

Strictly said, no merge protocol is defined in CLIQUES. Two solutions to merge two groups are suggested. The first considers a special case of multiple joins, i.e. members of a smaller group are considered as new members joining the larger group. The seconds creates a new super group via a fresh IKA (see Section 4.2.1.

Given a group of n members. k individual new members want to join the group. First, they are labeled from M_{n+1} to M_{n+k}. The rest is illustrated in Protocol 4.4. The first round is identical to the one in join protocol.

Protocol 4.4 (CLIQUES merge/multiple join)

1. *Round 1: The group controller M_c*
   - *updates its secret,*
   - *embeds the secret in a new message,*
   - *unicasts the message to M_{n+1}.*

   M_c $\longrightarrow$ M_{n+1}: $(g^{\widehat{r_c}\prod(r_m|m\in\{1,\cdots,n\}\setminus\{j\})}|j\in\{1,\cdots,n\}),\ g^{\widehat{r_c}\prod(r_m|m\in\{1,\cdots,n\})}$
2. *Round 2 to k: In round i with $2\leq i\leq k$, member M_{n+i-1}*
   - *embeds its secret in a new message,*
   - *unicasts the message to member M_{n+i}*

   M_{n+i-1} $\longrightarrow$ M_{n+i}: $(g^{\widehat{r_c}\prod(r_m|m\in\{1,\cdots,n+i-1\}\setminus\{j\})}|j\in\{1,\cdots,n+i-1\}),\ g^{\widehat{r_c}\prod(r_m|m\in\{1,\cdots,n+i-1\}}$
3. *Round $k+1$: Member M_{n+k}*
   - *embeds its secret in a new message,*
   - *broadcasts the message.*

   M_* $\longleftarrow$ M_{n+k}: $(g^{\widehat{r_c}\prod(r_m|m\in\{1,\cdots,n+k\}\setminus\{j\})}|j\in\{1,\cdots,n+k\})$
4. *Every member computes the group key.*

After running merge/multiple join protocol, all members are able to compute the new group key: $K_{G_{new}}=g^{\widehat{r_c}\prod_{i=1}^{n+k}r_i}$.

The complexity characteristics of **merge/multiple join** protocol is given in Table 4.6.

Rounds	$k+1$
Ucasts	k
Bcasts	1
Ucast size	$\frac{k^2+2nk+k}{2}$
Bcast size	$k+n$
Exps/M_i	$n+2$ for group controller, 1 for $i \in \{1, \cdots, n\} \setminus \{c\}$ $i+1$ for $i \in \{n+1, \cdots, n+k\}$.
All Exps	$\frac{k^2+2nk+3k+4n+2}{2}$
Serial Exps	$\frac{k^2+2nk+k+2n}{2}$
Remark: k: number of new members	

Table 4.6.: Complexity of CLIQUES merge/multiple join protocol

4.2.5. Partition protocol

The partition protocol is similar to the leave protocol. The only difference is, the group controller needs to compute fewer sub-keys. Let L denote the set of all members who leave the group, the process is illustrated in Protocol 4.5.

Protocol 4.5 (CLIQUES partition)

1. *Round 1: The group controller M_c*
   - *updates its secret,*
   - *embeds the secret in a new message,*
   - *broadcasts the message.*

   $M_c \xrightarrow{(g^{\widehat{r_c}} \prod(r_m | m \in \{1,\cdots,n\} \setminus \{j\}) | j \in \{1, \cdots, n\} \setminus L)} M_*$
2. *Every member computes the group key.*

After run of the complete partition protocol, all remaining members are able to compute the new group key $K_{G_{new}} = g^{\widehat{r_c} \prod_{i=1}^{n} r_i}$.

For the case that a group is split into more smaller groups. We can simply consider that all members which belong to other smaller groups leave the group. Each smaller groups proceed independently.

For example, a group with members M_1, M_2, $\cdots$, M_6 is split into groups G_1 with M_1, M_3, and M_5, and G_2 with M_2, M_4, and M_6. In group G_1, M_3 is chosen as group controller, while M_4 is the group controller of group G_2. Supposed the new generated secrets are $\widehat{r_3}$ and $\widehat{r_4}$, respectively. Then the sponsor's broadcast message in group G_1 is

$$(g^{r_2(r_3\widehat{r_3})r_4r_5r_6}, g^{r_1r_2(r_3\widehat{r_3})r_4r_6}, g^{r_1r_2\widehat{r_3}r_4r_5r_6}),$$

and in group G_2 is

$$(g^{r_1r_3(r_4\widehat{r_4})r_5r_6}, g^{r_1r_2r_3(r_4\widehat{r_4})r_5}, g^{r_1r_2r_3\widehat{r_4}r_5r_6}).$$

The new group key for group G_1 is

$$K_{G1_{new}} = g^{r_1r_2(r_3\widehat{r_3})r_4r_5r_6},$$

while the one for group G_2 is

$$K_{G2_{new}} = g^{r_1r_2r_3(r_4\widehat{r_4})r_5r_6}.$$

The complexity characteristics of **partition** protocol is given in Table 4.7.

Rounds	1
Ucasts	0
Bcast	1
Ucast size	0
Bcast size	$n-p$
Exps/M_i	$n-p$ for group controller,
	1 for all other remaining members
All Exps	$2n-2p-1$
Serial Exps	$n-p$
Remark: p: number of leaving members	

Table 4.7.: Complexity of CLIQUES partition protocol

4.2.6. Refresh protocol

The member M_r which is the least recent to have refreshed its key share generates a new share $\widehat{r_r}$ and broadcasts the updated information, as described in Protocol 4.6.

Protocol 4.6 (CLIQUES refresh)

1. *Round 1: The refresher M_r*
   - *updates its secret,*
   - *embeds the secret in a new message,*
   - *broadcasts the message.*

   $M_r \xrightarrow{(g^{\widehat{r_r}} \prod(r_m | m \in \{1,\cdots,n\} \setminus \{j\}) | j \in \{1,\cdots,n\})} M_*$
2. *Every member computes the group key.*

After running the refresh protocol, each member compute the new group key $K_{G_{new}} = g^{\widehat{r_r}} \prod(r_m | m \in \{1,\cdots,n\})$. The complexity characteristics of **refresh** protocol is given in Table 4.8.

Rounds	1
Ucasts	0
Bcasts	1
Ucast size	0
Bcast size	n
Exps/M_i	n for refresher,
	1 for all other members
All Exps	$2n-1$
Serial Exps	n

Table 4.8.: Complexity of CLIQUES refresh protocol

4.3. STR Protocol Suite

STR, suggested by Kim *et al.* [57], bases on a protocol suggested by Steer *et al.* in 1988 [107], and extends this protocol to support dynamic group operations (Section 3.8.3.2). Since STR is optimized against the communication delay, it is particularly efficient for group key agreement in networks where high network delays dominate.

For sake of clarity the blinded keys and blinded session randoms are denoted *blinded values*. Besides the notations in Figure 4.1 and 4.2, the following notations will be used for STR.

br_i	M_i's public blinded session random, i.e. g^{r_i} mod p
k_i	secret key shared among $M_1 \cdots M_i$
bk_i	public blinded k_i, i.e. g^{k_i} mod p.
N_i	Tree node i
IN_i	Internal tree node at level i
LN_i	Leaf node associated with member M_i
BR_i^*	Set of M_i^*'s blinded session randoms

STR uses a key tree to manage the key group. The tree has two types of nodes, namely leaf and internal nodes. Each specific group member M_i is associated with a leaf node LN_i, while an internal node IN_i has two children: the left child IN_{i-1}, and the right child LN_i. Each leaf node LN_i generates randomly a session random r_i which should be kept secretly, and computes the corresponding blinded session random $br_i = g^{r_i}$. An internal node IN_i has a secret key k_i and the corresponding public blinded key $bk_i = g^{k_i}$. The difference is that k_i is not randomly chosen, but the result of a two-party DH key exchange between its two children:

$$k_i = (bk_{i-1})^{r_i} = (br_i)^{k_{i-1}} \equiv g^{k_{i-1}r_i} \mod p, i > 1. \tag{4.1}$$

The internal node with the greatest index is considered to be the root. The secret key of the root is the shared group key. For a group of n members, the root is IN_n, and the group key k_n can be recursively computed using Equation 4.1.

If $n = 4$, its group key is

$$K_G = k_4 = g^{(r_4 g^{(r_3 g^{(r_2 r_1)})})}.$$

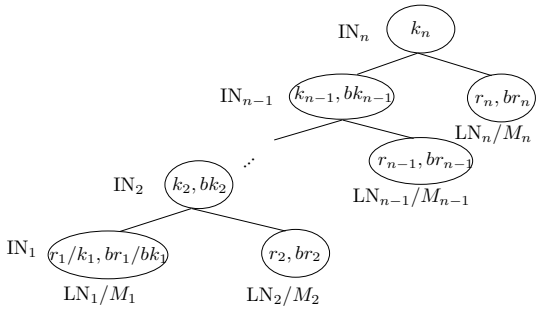


Figure 4.5.: STR tree

Looking at the STR key tree with n members in Figure 4.5, the member $M_c, 1 \leq i \leq n$, must know its own session random, all blinded keys and blinded session randoms, and keys of the path from its parent node to root node. More formally, it must store r_c, br_i, bk_i for $i = 1, \cdots, n$ ($br_1 = bk_1$), and k_i for $i = c, \cdots, n$. So the memory costs of M_c are $n - c + 2$ keys for $c \geq 2$ and n keys for $c = 1$, and $2n - 2$ public keys for all members. The meanings of some symbols are listed in the following:

h	$n - 1$
KEY_3^*	$r_3, k_3, \cdots, k_n$
BK_3^*	$bk_1, \cdots, bk_{n-1}$
BR_3^*	$br_1, \cdots, br_n$
$KpBK_3^*$	$br_3, bk_3, \cdots, bk_{n-1}$
$CoBK_3^*$	$bk_2, br_4, \cdots, br_n$

The rest of this section describes how STR deals with the group operations (Section 3.8.3.2). Some protocols can be further optimized for ad hoc networks, e.g. setup protocol, join protocol, and merge protocol. The optimization can be found in Section 5.2.

4.3.1. Setup protocol

The members who want to form a group can be ordered according to some criteria. The structure of the key tree can be then derived from this order. Suppose each member i, namely leaf node LN_i is equipped with r_i and br_i. The process is illustrated in Protocol 4.7.

Protocol 4.7 (STR setup)

1. *Round 1:*
   Each $M_i, i \in \{1, \cdots, n\}$ broadcasts its blinded session random:
   $M_i \xrightarrow{br_i} M_*$
2. *Round 2:*
   The member M_1
   - *computes recursively $k_i = (br_i)^{k_{i-1}}$ and $bk_i = g^{k_i}$, $\forall i \in \{2, 3, \cdots, n-1\}$ and $k_n = (br_n)^{k_{n-1}}$,*
   - *broadcasts the updated tree with all blinded keys.*
     $M_1 \xrightarrow{\widehat{T_1}(BK_1^*)} M_*$
3. *Each member $M_i, i \geq 2$, computes $k_i = (bk_{i-1})^{r_j}$ and then recursively $k_j = (br_j)^{k_{j-1}}, \forall j \in [i+1, n]$.*

The complexity characteristics of setup protocol are summarized in Table 4.9.

Rounds	2
Ucasts	0
Bcasts	$n+1$
Exps/M_i	$2n-1$ for $i=1$,
	$n-i+1$ for $2 \leq i \leq n$,
Ucast size	0
Bcast size	$2n-1$
All Exps	$\frac{n^2+3n-2}{2}$
Serial Exps	$3n-2$

Table 4.9.: Complexity of STR setup protocol

4.3.2. Join protocol

The current group has n members, the new member is identified with M_{n+1}. The tree will be updated by incrementing $n = n+1$ and adding a new internal node IN_n with two children: the root node IN_{n-1} of the prior tree T_i on the left and the new leaf node LN_n on the right. This node becomes the new root node. Figure 4.6 gives an example of addition of a new member to a group with four members.

For simplicity, we use n in the following to denote the number of group members before operation $join$.

To deal with the join operation, the member M_n is chosen as the sponsor. The new member M_{n+1} broadcasts a join request containing its own blinded key bk_{n+1}. All members M_i with $1 \leq i \leq n$ can compute the new group key. The sponsor unicasts all blinded keys to M_{n+1}. Equipped with this message the new member can also compute the new group key.

However this join protocol does not provide key independence since knowledge of a previous group key can be used to compute the new group key. To remedy the situation, [57] suggests that the sponsor updates

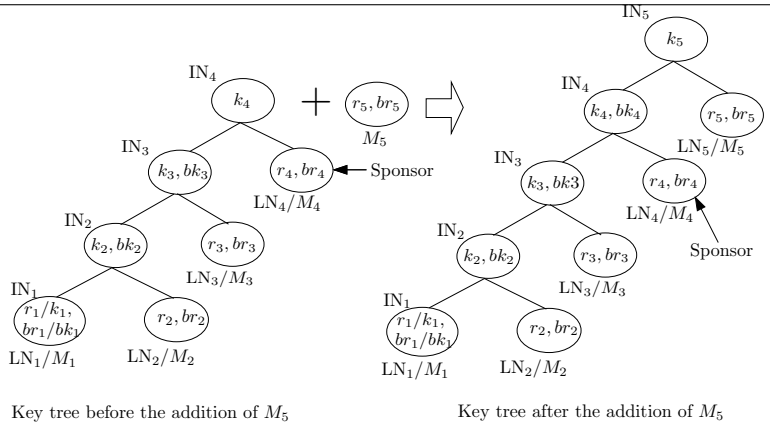


Figure 4.6.: An example of STR join

its session random. The changed information will then be broadcasted to all members. The process is illustrated in Protocol 4.8.

Protocol 4.8 (STR join)

1. *Round 1:*
   The new member broadcasts its own blinded session random br_{n+1}.
   $M_{n+1} \xrightarrow{br_{n+1}} M_*$
   The sponsor $M_s (s = n)$
   - *updates its session random* r_s *and* br_s,
   - *computes new* k_n, bk_n,
   - *broadcasts the updated tree with all blinded keys and blinded session randoms.*
     $M_n \xrightarrow{\widehat{T_n}(BK_n^* \cup BR_n^* \setminus \{br_{n+1}\})} M_*$
2. *Every member* M_i
   - *updates the tree by inserting* M_{n+1},
   - *sets* $n = n + 1$,
   - *computes the keys:*
     - *if* $i = 1, \cdots, n-2$, *computes* $k_j = br_j^{k_{j-1}}$ *for* $j = n-1, n$,
     - *if* $i = n$ *(new member), computes* $k_n = bk_{n-1}^{r_n}$.
     - *if* $i = s$ *(sponsor), computes* $k_n = br_n^{k_{n-1}}$.

The complexity characteristics of join protocol is given in Table 4.10.

4.3.3. Leave protocol

Like in CLIQUES, the leave protocol in STR is relatively simple, only one round is needed. Suppose we have a group of n members and the member M_l with $1 \leq l \leq n$ leaves the group. Again we need a sponsor M_s to update its session random. If $l > 1$, the sponsor is the leaf node directly below the leaving member, i.e. M_{l-1}, otherwise the sponsor is M_2. Since the tree will be updated and renumbered (see below), if M_1 leaves the group, the sponsor is also M_1 after renumbering.

After notification of the leave event from the group communication system, each remaining member updates the key tree by deleting the nodes LN_l and IN_l, and then renumbers the nodes above M_l.

The process is illustrated in Protocol 4.9.

Rounds	2
Ucasts	0
Bcasts	2
Exps/M_i	4 for sponsor, 1 for new member, 2 for all others
Ucast size	0
Bcast size	$2n$
All Exps	$2n+3$
Serial Exps	6

Table 4.10.: Complexity of STR join protocol

Protocol 4.9 (STR leave)

1. *Round 1:*
   Every member
   - *updates the tree by removing the leaving member* M_l,
   - *sets* $n = n - 1$.

   The sponsor M_s *additionally*
   - *updates its session random* r_s *and* br_s,
   - *computes* k_n *and* k_i *and* bk_i, $\forall i \in [max(2, s), n-1]$,
   - *broadcasts the updated tree with all blinded keys and* br_s.

   M_s $\xrightarrow{\quad \widehat{T}_s(BK_s^* \cup \{br_s\}) \quad}$ M_*

2. *Every member* M_i,
   - *if* $i < s$, *computes* $k_j = br_j^{k_{j-1}}$, $\forall j \in [s, n]$.
   - *if* $i > s$, *computes* $k_i = bk_{i-1}^{r_j}$, *and then recursively* $k_j = br_j^{k_{j-1}}$, $\forall j \in [i+1, n]$.

Figure 4.7 gives an example of the exclusion of a member from a group of four members.

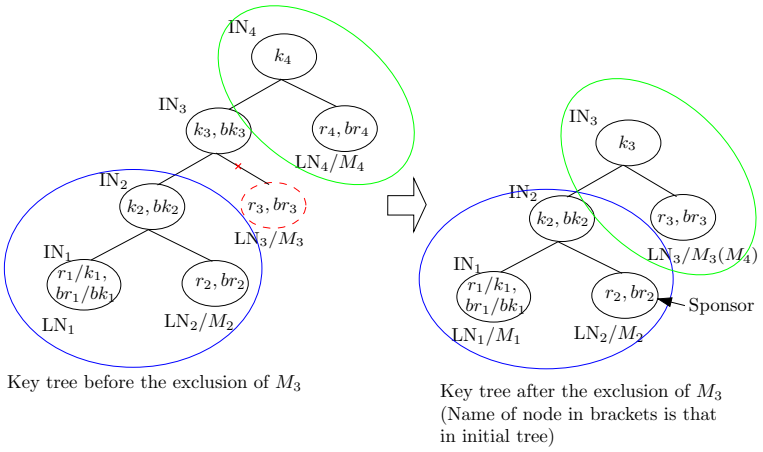


Figure 4.7.: An example of STR leave

The complexity characteristics of leave protocol is given in Table 4.11.

	$s = 1$	$s \geq 2$
Rounds	1	1
Ucasts	0	0
Bcasts	1	1
Exps/M_i	$2n - 4$ for sponsor $n - i$ for member members	$2(n - s)$ for sponsor $n - i$ for M_i with $i > s$ $n - s$ for M_i with $i < s$
Ucast size	0	0
Bcast size	$n - 2$	$n - 1$
All Exps	$\frac{n^2+n-6}{2}$	$\frac{n^2+n-s-s^2}{2}$
Serial Exps	$3(n - 2)$	$3(n - s)$
Remark: LN_s: sponsor in updated tree		

Table 4.11.: Complexity of STR leave protocol

4.3.4. Merge protocol

In CLIQUES, no actual merge protocol is present due to the difficulty to deal with it. The key tree allows relatively simple merge of two groups. The merge protocol covers also the multiple join.

The smaller group is merged onto the larger one, i.e. to place a smaller key tree directly on top of the larger one. If group sizes are equal, we can order them according to some other criteria. A new intermediate node N with two children is created. The root of the larger tree becomes the left child of N, while the lowest-numbered leaf of the smaller trees the right child of N. The root of smaller tree becomes the root of the new tree.

We need a sponsor for each group, The same as in join protocol, the topmost leaf node is selected as sponsor. Both sponsors exchange key their respective key trees containing all blinded keys in the first round, the sponsor of the larger key tree becomes then the new sponsor in round 2. The new sponsor updates first its session random, and then computes all (key, blinded key) pair up to the new root node. It then broadcasts the new tree with all blinded keys and blinded session randoms.

Figure 4.8 gives an example of the merge of two groups, with 3 and 2 members respectively.

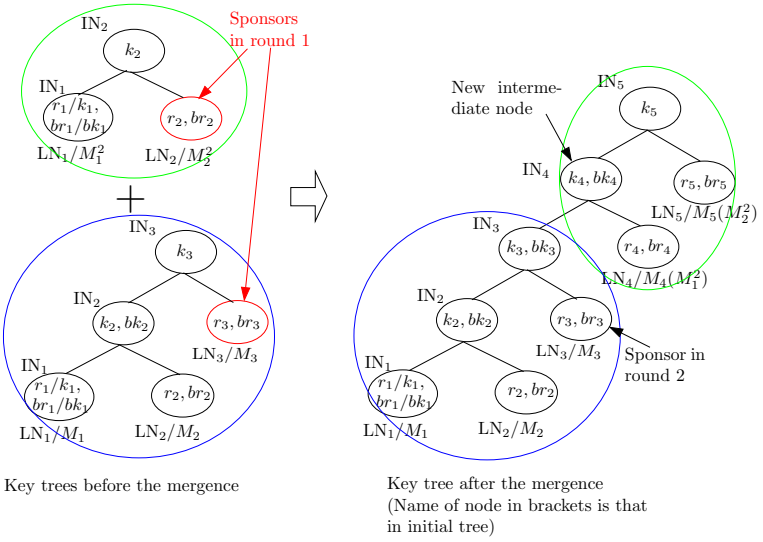


Figure 4.8.: An example of STR merge

Assume we have two trees, the larger one T^1 with n_1 members, and the less one T^2 with n_2 members.

The sponsors of both trees are denoted by M_{s_1} and M_{s_2} respectively. The process to merge two groups is illustrated in Protocol 4.10.

Protocol 4.10 (STR 2-group merge)

1. *Round 1:*
   Both sponsors M_{s_1} and M_{s_2} exchange T^1 and T^2 with all blinded keys and blinded session randoms respectively.
   M_{s_1} $\xrightarrow{\quad T^1(BK^*_{s_1} \cup BR^*_{s_1}) \quad}$ M_{s_2}
   M_{s_2} $\xrightarrow{\quad T^2(BK^*_{s_2} \cup BR^*_{s_2}) \quad}$ M_{s_1}

2. *Round 2: The sponsor M_s (formerly M_{s_1})*
   - *updates its session random,*
   - *computes (k_i, bk_i), $i = n_1 - 1, \cdots, n_1 + n_2 - 1$, and $k_{n_1+n_2}$,*
   - *broadcasts the updated tree with all blinded keys.*
     M_s $\xrightarrow{\quad \widehat{T}_s(BK^*_s \cup BR^*_s) \quad}$ M_*

3. *Every member M_i,*
   - *if $i < s$, computes $k_j = br_j^{k_{j-1}}$, $\forall j \in [s, n]$,*
   - *if $i > s$, computes $k_i = (bk_{i-1})^{r_i}$, and then recursively $k_j = br_j^{k_{j-1}}$, $\forall j \in [i+1, n]$.*

The complexity characteristics of the 2-group merge protocol are summarized in Table 4.12.

Rounds	2
Ucasts	2
Bcasts	1
Exps/M_i	$2k + 2$ for M_n
	$k + 1$ for $M_i, i < n$
	$n - i + 1$ for others M_i
Ucast size	$2(n + k - 2)$
Bcast size	$2(n + k - 1)$
All Exps	$\frac{k(2n+k+3)}{2} + n + 1$
Serial Exps	$3k + 3$
Remark: k: number of members in smaller group	

Table 4.12.: Complexity of STR 2-group merge protocol

The merge protocol can be generalized to merge more than two groups. Suppose we have k groups. All k groups are ordered from the largest to the smallest groups T^1 to T^k with n_1 to n_k members, respectively. The topmost leaf node of each key tree T^i is chosen as the sponsor, denoted by M_{s_i}.

Two of the variants are discussed in following. One is the **consequent 2-group merge**. In this method the first two groups perform a 2-group merge and as result is a updated tree $\widehat{T_M}$. Group of $\widehat{T_M}$ performs then a 2-group merge with the third group, and so on, until all groups are merged. The major advantage of this method is that it can call 2-group merge as sub-protocol. However, new group is formed first after $2(k-1)$ rounds.

Another is **m-group merge**. Its process is described in Protocol 4.11. M_{s_1} becomes in round 2 the new sponsor and is denoted M_s. Unlike 2-group merge, the trees are not unicasted between sponsors, but

broadcasted. The reason is that $m(m-1)$ messages are needed to exchange key tree among m groups using unicast.

Protocol 4.11 (STR m-group merge)

1. *Round 1:*
   Every sponsor M_{s_i} broadcasts its key tree with all blinded keys. For i with $1 \leq i \leq m$:
   $M_{s_i} \xrightarrow{T^i_{s_i}(BK_{s_i} \cup BR_{s_i})} M_*$

2. *Round 2: Every member*

   - *updates the tree by merging all trees,*
   - *sets $n = \sum_{i=1}^{m} n_i$.*

   The sponsor M_s (formerly M_{s_1}) additionally

   - *updates its session random,*
   - *computes k_n and (k_i, bk_i), $\forall i \in [n_1, n-1]$,*
   - *broadcasts the updated tree with all blinded keys.*
     $M_s \xrightarrow{\widehat{T}_s(BK^*_s)} M_*$

3. *Every member M_i,*

   - *if $i < s$, computes $k_j = br_j^{k_{j-1}}$, $\forall j \in [s, n]$,*
   - *if $i > s$, computes $k_i = (bk_{i-1})^{r_i}$, and then recursively $k_j = br_j^{k_{j-1}}$, $\forall j \in [i+1, n]$.*

Rounds	2
Ucasts	0
Bcasts	$m+1$
Exps/M_i	$2k+2$ for M_n
	$k+1$ for $M_i, i < n$
	$n+k+1-i$ for others M_i
Ucast size	0
Bcast size	$4(n+k)-2m-1$
All Exps	$\frac{k(2n+k+3)}{2} + n + 1$
Serial Exps	$3k+3$
Remark: k: number of members in all other groups, m: number of merging groups	

Table 4.13.: Complexity of STR m-group merge protocol

The merge protocol provides backward secrecy since all members are only given blinded keys of the other groups. However, the merge protocol does not provide key independence, since knowledge of a group key of tree T_1 used before merge can be used to compute the group key used after the merge. This problem will be remedied, if the sponsor in the second round update its session random. This solution is considered in the improved merge protocol (Section 5.2.4).

The complexity characteristics of the m-group merge protocol are summarized in Table 4.13.

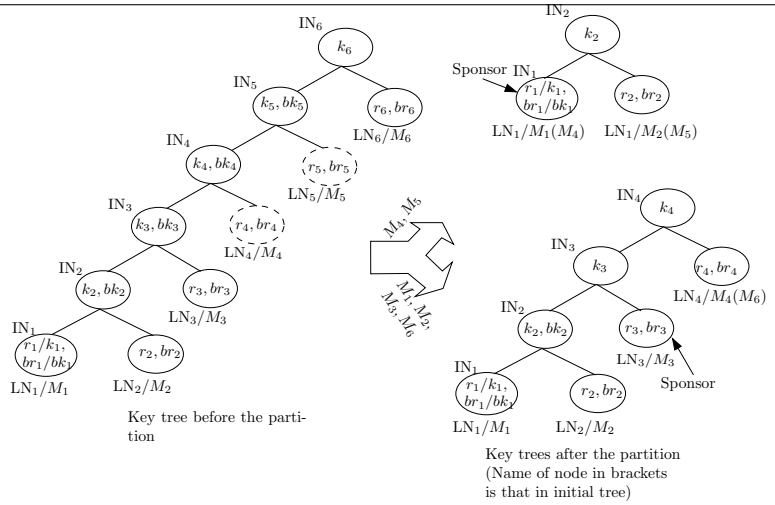


Figure 4.9.: An example of STR partition

4.3.5. Partition proocol

The partition protocol is similar to the leave protocol. The only difference is the choice of the sponsor. We usually choose the surviving leaf node directly below the lowest-numbered leaving member. If no such leaf node exists, in other words, if M_1 leaves the group, we choose the lowest-numbered surviving leaf node as sponsor. An example is given in Figure 4.9.

Suppose we have a group of n members when p of them leave the group. The process is illustrated in Protocol 4.12.

Protocol 4.12 (STR partition)

1. *Round 1:*
   Every member
   - *updates the tree by removing the leaving members,*
   - *sets* $n = n - p$.

   The sponsor M_s *additionally*
   - *updates its session random* r_s *and the blinded one* br_s,
   - *computes new* k_n, *and* (k_i, bk_i), $\forall i \in [max(2, s), n-1]$,
   - *broadcasts the updated tree with all blinded keys and its new blinded session random.*

   M_s $\xrightarrow{\quad \widehat{T}_s(BK_s^* \cup \{br_s\}) \quad}$ M_*

The complexity characteristics of partition protocol are summarized in Table 4.14.

4.3.6. Refresh protocol

No refresh protocol is defined in STR protocol by Kim *et al.* in [57]. With respect to other protocols in STR, we suggest one as follows.

Suppose we have a group of n members, and member M_r, called *refresher*, updates its session random. Then it computes all keys and blinded keys from its parent to root node. After computing it broadcasts the updated tree with all blinded keys and its new blinded session random.

The process is illustrated in Protocol 4.13.

	$s = 1$	$s \geq 2$
Rounds	1	1
Ucasts	0	0
Bcasts	1	1
Exps/M_i	$2m - 2$ for sponsor $m + 1 - i$ for other members	$2(m - s) + 2$ for sponsor $m + 1 - i$ for M_i with $i > s$ $m + 1 - s$ for M_i with $i < s$
Ucast size	0	0
Bcast size	$m - 1$	m
All Exps	$\frac{m^2+3m-4}{2}$	$\frac{m^2+3m-s-s^2+2}{2}$
Serial Exps	$3m - 3$	$3(m - s + 1)$
Remark: p: number of leaving members, $m := n - p$, s: index of sponsor in updated tree		

Table 4.14.: Complexity of STR partition protocol

Protocol 4.13 (STR refresh)

1. *Round 1:*
   The refresher M_r
   - *updates its session random r_r and the blinded one br_r,*
   - *computes k_n, and (k_i, bk_i), $\forall i \in [r, n-1]$,*
   - *broadcasts the updated tree with all blinded keys and new session random.*

   M_r $\xrightarrow{\qquad \widehat{T_r}(BK_s^* \cup \{br_r\}) \qquad}$ M_*

2. *Every member M_i,*
   - *if $i < r$, computes k_j, $\forall j \in [r, n]$,*
   - *if $i > r$, computes k_j, $\forall j \in [i, n]$.*

Like other protocols, we summarize the complexity characteristics in Table 4.15.

	$r = 1$	$r \geq 2$
Rounds	1	1
Ucasts	0	0
Bcasts	1	1
Exps/M_i	$2n - 2$ for sponsor $n - i + 1$ for other members	$2(n - r + 1)$ for sponsor $n - i + 1$ for M_i with $i > r$ $n - r + 1$ for M_i with $i < r$
Ucast size	0	0
Bcast size	$n - 1$	n
All Exps	$\frac{n^2+3n-4}{2}$	$\frac{n^2+3n-r-r^2+2}{2}$
Serial Exps	$3n - 3$	$3(n - r + 1)$
Remark: LN_r: refresher		

Table 4.15.: Complexity of STR refresh protocol

4.4. TGDH Protocol Suite

Kim *et al.* present in [58] the Tree-based Group Diffie-Hellman (TGDH). It uses a balanced binary key tree to manage the keys. The security properties key independence and group key secrecy (see Section 3.8.3.3) are guaranteed in TGDH.

Besides the notations in Figure 4.1, the following notations will be used for TGDH .

$\langle l, v\rangle$	v^{th} node at level l in a tree
$k_{\langle l,v\rangle}$	key of node $\langle l, v\rangle$
$bk_{\langle l,v\rangle}$	blinded key of node $\langle l, v\rangle$
$T_{\langle i,j\rangle}$	a subtree rooted at node $\langle i, j\rangle$

In a TGDH key tree, the node located at level 0, node $\langle 0, 0\rangle$, is the root. The height h of a tree equals the shallowest level. A member is associated with a leaf node. A non-leaf node is called an internal node with two children. The left child of node $\langle l, v\rangle$ is node $\langle l + 1, 2v\rangle$, and the right one is node $\langle l + 1, 2v + 1\rangle$. A node $\langle l, v\rangle$ has the key $K_{\langle l,v\rangle}$ and the corresponding blinded key $bk_{\langle l,v\rangle} = g^{k_{\langle l,v\rangle}}$. The key $k_{\langle l,v\rangle}$ can be recursively computed as follows:

$$\begin{aligned} k_{\langle l,v\rangle} &= (k_{\langle l+1,2v\rangle})^{k_{\langle l+1,2v+1\rangle}} \bmod p \\ &= (k_{\langle l+1,2v+1\rangle})^{k_{\langle l+1,2v\rangle}} \bmod p \\ &= g^{(k_{\langle l+1,2v\rangle} k_{\langle l+1,2v+1\rangle})} \bmod p \end{aligned}$$

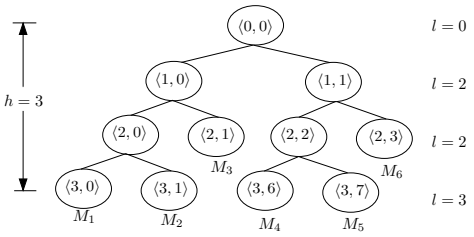


Figure 4.10.: TGDH tree

Figure 4.10 gives an example of a key tree of a group with six members. The group key for this group is

$$k_{\langle 0,0\rangle} = g^{((g^{r_3(g^{r_1 r_2})})(g^{r_6(g^{(r_4 r_5)})}))},$$

where $r_1, \cdots, r_6$ are the keys of members $M_1, \cdots, M_6$ respectively. Considering this example, the meaning of some notations is listed in the following:

h	3
KEY_2^*	$k_{\langle 3,1\rangle}, k_{\langle 2,0\rangle}, k_{\langle 1,0\rangle}, k_{\langle 0,0\rangle}$
BK_2^*	$bk_{\langle 1,0\rangle}, bk_{\langle 1,1\rangle}, bk_{\langle 2,0\rangle}, bk_{\langle 2,1\rangle}, bk_{\langle 2,2\rangle},$
	$bk_{\langle 2,3\rangle}, bk_{\langle 3,0\rangle}, bk_{\langle 3,1\rangle}, bk_{\langle 3,6\rangle}, bk_{\langle 3,7\rangle}$
$KpBK_2^*$	$bk_{\langle 3,1\rangle}, bk_{\langle 2,0\rangle}, bk_{\langle 1,0\rangle}$
$CoBK_2^*$	$bk_{\langle 3,0\rangle}, bk_{\langle 2,1\rangle}, bk_{\langle 1,1\rangle}$

The overhead of the TGDH protocol depends on many factors, e.g. tree height, balance of the key tree, location of insertion points and leaving members. Hence some characteristics can only be estimated for the worst case, and some issues can even not be estimated.

Assume that we have a TGDH tree with n members, there are $n - 1$ internal nodes. Each member must know all blinded keys of all nodes except for the root, and all keys in its key-path. So the *memory costs* for $M_{\langle l,v\rangle}$ are $2n - 2$ public keys and $l + 1$ keys.

The rest of this section gives an overview of the protocols setup, join, leave, merge and partition, and refresh in TGDH.

4.4.1. Setup protocol

[58] does not suggest any setup protocol. Related to other protocols in TGDH, we suggest a setup protocol for TGDH.

Suppose there are n members who wish to form a group (key tree). The shallowest rightmost leaf node is chosen as the sponsor. First, all members are ordered according to some criteria to be $M_1, M_2, \cdots, M_n$. Then the first two members, M_1 and M_2, execute a 2-party DH key exchange. The same do the members M_i and $M_i + 1$, where i is an even integer from $[0, n]$. Now we have $\lceil \frac{n}{2} \rceil$ groups. Each sponsor broadcasts then its tree with all blinded keys. The rest process is similar to be the one in the first round, if we consider a member as a group with only one member. Such process is repeated until all members are in one group. After i-th round the number of groups is reduced to $\lceil \frac{n}{2^i} \rceil$. The setup of the group is finished after $\lceil \log_2^n \rceil$ rounds.

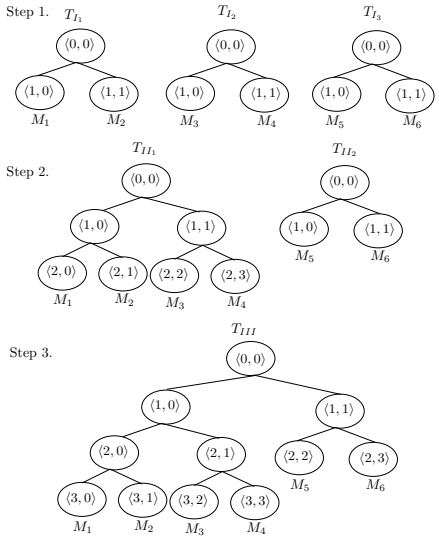


Figure 4.11.: An example of TGDH setup

An example with $n = 6$ is given in Figure 4.11. Assume the order of members is $M_1, \cdots, M_6$. In the first round M_1 and M_2 perform a DH key exchange and forms a new group T_{I_1}. Similarly, groups T_{I_2} of M_3 and M_4, and T_{I_3} of M_5 and M_6 are formed respectively. In the second round, M_2, M_4 broadcasts their trees with respective blinded keys[6]. Then groups T_{I_1} and T_{I_2} form a new group T_{II_1}, T_{I_3} do nothing and is renamed to T_{II_2}. Finally the only two groups T_{II_1} and T_{II_2} form the group T_{III_1} consisting of all members.

The complexity characteristics of setup protocol are summarized in Table 4.16.

4.4.2. Join protocol

Assume that a new member M_{n+1} joins to a group of n members $\{M_1, \cdots, M_n\}$. The new member broadcasts its join-request with its own bkey. After receiving the join-request, each member in current group determines the insertion point. If the tree is fully balanced, that means $n = 2^h$, where h is the height of a tree, the new member joins to the root node. Otherwise, the shallowest rightmost leaf node is the insertion point. The reason of such selection is to keep the key tree as balanced as possible.

The tree must be first updated. As in STR, the sponsor is the shallowest rightmost leaf in the subtree rooted at the insertion node. A new intermediate node is created. Node at the insertion point becomes the left child of the new intermediate node, and the new member becomes the right child.

The process of join protocol in TGDH is illustrated in Protocol 4.14.

[6]Since there are no more groups after T_{I_3}, the sponsor M_6 needs not to broadcast its tree.

Rounds	h
Ucasts	0
Bcasts	$h^2 - h + n$
Exps/$\langle l, v\rangle$	$l + 1 + i$, where i is the least non-negative integer so that $\lfloor \frac{v}{2^i} \rfloor$ is even. If $\langle l, v\rangle$ is the shallowest rightmost leaf node in the finally formed tree, it is $l + i$.
Ucast size	0
Bcast size	$(h-2)2^{h+2} + n + 8$
Serial Exps	$2h - 1$
Remark: $h = \lceil \log_2 n \rceil$	

Table 4.16.: Complexity of TGDH setup protocol

Protocol 4.14 (TGDH join protocol)

1. *Round 1:*
   The new member broadcasts its join-request with its blinded key.
   M_{n+1} $\xrightarrow{\qquad M'_{n+1}sbkey \qquad}$ M_*
2. *Every member*
   - *updates key tree,*
   - *removes all keys and bkeys from sponsor to the root node.*

   The sponsor M_s additionally,
   - *updates its share,*
   - *computes then all keys and bkeys in its key-path,*
   - *broadcasts the updated tree $\widehat{T}_s$ with all bkeys.*

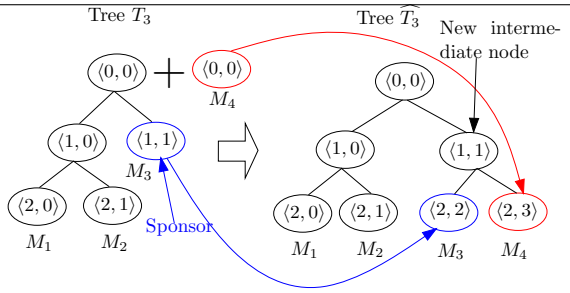
   M_s $\xrightarrow{\qquad \widehat{T}_s(BK^*_s) \qquad}$ M_*
3. *Every member computes the group key using $\widehat{T}_s$.*

Figure 4.12 gives an example of a group with 3 members when a new member M_4 joins into this group. Member M_3 is selected as the sponsor. Each member in the old tree updates the tree by inserting M_4 and removes the keys and blinded keys in M_3's key-path. The sponsor M_3 additionally updates its share and computes $k_{\langle 1,1\rangle}, bk_{\langle 1,1\rangle}$, and $k_{\langle 0,0\rangle}$, and then broadcasts the updated tree with all blinded keys. Equipped with the broadcast message, M_1 and M_2 computes the new $k_{\langle 0,0\rangle}$, and M_4 computes $k_{\langle 1,1\rangle}$ and $k_{\langle 0,0\rangle}$.

The complexity characteristics of the join protocol suggested in [58] is given in Table 4.17.

4.4.3. Leave protocol

The leave protocol is relatively simple. Assume that we have a group of n members and member M_l leaves the group. The rightmost leaf node of M_l's sibling subtree is selected as the sponsor. After notification from GCS about the leave event, each remaining member updates its key tree by deleting M_l. The former sibling of M_d is promoted to replace M_d's parent node. The sponsor must additionally refresh keys in its key-path. The process of leave protocol is illustrated in Protocol 4.15.



1. renames node $\langle 1,1\rangle$ to $\langle 2,2\rangle$.
2. generates a new intermediate node $\langle 1,1\rangle$ and a new member node $\langle 2,3\rangle$.
3. promotes $\langle 1,1\rangle$ as the parent node of $\langle 2,2\rangle$ and $\langle 2,3\rangle$.

Figure 4.12.: An example of TGDH join

Rounds	2
Ucasts	0
Bcasts	2
Exps/M_j	$2l_s$ for sponsor, $l_s + 1 - i$ if $\exists i \in [0, l_s]$ so that $M_j \in T_{\langle l_s - i, v_s/2^i\rangle} \backslash T_{\langle l_s - i + 1, v_s/2^{i-1}\rangle}$
Ucast size	0
Bcast size	$2n + 1$
Serial Exps	$3l_s$
Remark: $\langle l_s, v_s\rangle$: sponsor in updated tree	

Table 4.17.: Complexity of TGDH join protocol

Protocol 4.15 (TGDH leave protocol)

1. *Round 1:*
   Each member
   - *removes the leaving member and relevant parent node,*
   - *removes all keys and bkeys pairs from sponsor to root node.*

   The Sponsor M_s additionally,
   - *updates its share,*
   - *computes then all keys and bkeys in its key-path,*
   - *broadcasts updated tree $\widehat{T}_s$ including all blinded keys.*

   $M_s \qquad \xrightarrow{\qquad \widehat{T}_s(BK_s^*) \qquad} \qquad M_*$

2. *Every member computes the group key using $\widehat{T}_s$.*

Figure 4.13 gives an example of a group of 3 members when M_3 leaves this group. M_5 is selected as the sponsor. The remaining members M_1, M_2, M_4, and M_5 remove the nodes $\langle 2,2\rangle$ and $\langle 1,1\rangle$. M_5 additionally updates its share and computes $k_{\langle 1,1\rangle}$, $bk_{\langle 1,1\rangle}$, $k_{\langle 0,0\rangle}$, and then broadcasts the updated tree with all blinded keys. Equipped with this broadcast message, M_1 and M_2 compute new $k_{\langle 0,0\rangle}$, and M_4 computes $k_{\langle 1,1\rangle}$ and $k_{\langle 0,0\rangle}$.

The complexity characteristics of leave protocol suggested in [58] are summarized in Table 4.18.

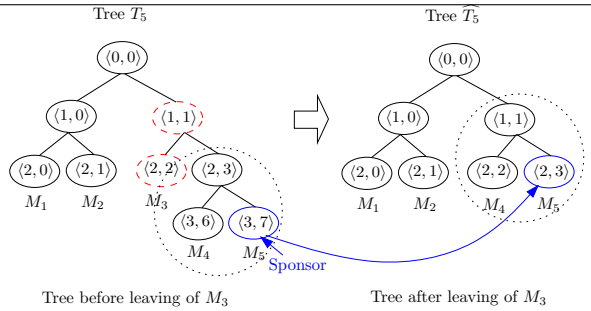


Figure 4.13.: An example of TGDH leave

Rounds	1
Ucasts	0
Bcasts	1
Exps/M_j	$2l_s$ for sponsor, $l_s + 1 - i$ if $\exists i \in [0, l_s]$ so that $M_j \in T_{\langle l_s - i, v_s/2^i\rangle} \backslash T_{\langle l_s - i + 1, v_s/2^{i-1}\rangle}$
Ucast size	0
Bcast size	$2n - 4$
Serial Exps	$3l_s$
Remark: $\langle l_s, v_s\rangle$: sponsor in updated tree	

Table 4.18.: Complexity of TGDH leave protocol

4.4.4. Merge protocol

Compared to CLIQUES, the main virtue of TGDH is that it is much simpler to merge two or more groups. Multiple join can be processed as follows. We assume that m members want to join group G_1. The m individual members form a TGDH group G_2 using setup protocol. Then G_2 merges with G_1.

We consider first the merge of two groups. It can be simply extended to the merge of more than two groups, say $k > 2$, groups by executing the two-group merge $k - 1$ times.

First the two trees are ordered from the highest to lowest, denoted T^1 and T^2. If they are of the same height, they are ordered according to some other criteria. T^2 joins to T^1, and the insertion point is determined. If the two trees are of the same height, we join simply T^2 to the root node of T^1. Otherwise we first try find the rightmost shallowest node where the join would not increase the overall tree height. If no such node exists, the insertion point is the root node.

Assume that we have m trees to be merged. They can be ordered from the highest to the lowest: $T^1, \cdots, T^m$. To perform merge, each tree T^i has its rightmost shallowest leaf node as sponsor M_{s_i}. The process is illustrated in Protocol 4.16.

Protocol 4.16 (TGDH merge protocol)

1. *Round 1: Each sponsor M_{s_i} in tree T^i*
   - *updates its share,*
   - *computes all keys and bkeys in the key-path of T^i (including $bk_{\langle 0,0\rangle}$),*
   - *broadcasts updated tree $\widehat{T^i}$ including only all bkeys. For $i = 1, 2, \cdots, m$*

   M_{s_i} $\xrightarrow{\qquad \widehat{T^i_{s_i}}(BK^*_s \cup \{bk_{\langle 0,0\rangle}\}) \qquad}$ M_*

   Each member
   - *updates the key tree and determinates the new sponsors $M_{s_1}, \cdots, M_{s_t}$,*
   - *removes all keys and bkeys in sponsors' key-paths.*

2. *Round 2 to q (repeat this step until any sponsor computes group key):*
   Each sponsor M_{s_i} with $1 \leq i \leq t$
   - *computes all keys and bkeys pairs in the key-path as far as possible,*
   - *broadcasts updated tree with all blinded keys.*

   M_{s_i} $\xrightarrow{\widehat{T_{s_i}}(BK^*_{s_i})}$ M_*
3. *Every member computes the group key using $\widehat{T_{s_i}}$*

An example of the merge of two groups is given in Figure 4.14. Both sponsors M_5 and M_7 updates first their shares respectively. Then M_5 computes new $k_{\langle 1,1\rangle}$, $bk_{\langle 1,1\rangle}$, $k_{\langle 0,0\rangle}$ and $bk_{\langle 0,0\rangle}$ in tree T_5, and M_7 computes $k_{\langle 0,0\rangle}$ and $bk_{\langle 0,0\rangle}$ in tree T_7. Both sponsors M_5 and M_7 broadcast their updated trees with all bkeys[7]. Each member merges both trees independently, and chooses M_2 as the new sponsor. All members remove then all keys and bkeys in M_2's key-path. M_2 additionally updates its share and computes new $k_{\langle 2,1\rangle}$, $bk_{\langle 2,1\rangle}$, $k_{\langle 1,0\rangle}$, $bk_{\langle 1,0\rangle}$, and $k_{\langle 0,0\rangle}$. M_2 broadcasts then the updated tree with all blinded keys. Equipped with the broadcast message, M_1 computes $k_{\langle 2,0\rangle}$, $k_{\langle 1,0\rangle}$, and $k_{\langle 0,0\rangle}$. M_6 and M_7 compute $k_{\langle 1,0\rangle}$, and $k_{\langle 0,0\rangle}$. All other members M_3, M_4, and M_5 compute only the new group key $k_{\langle 0,0\rangle}$. Since there is only one sponsor for the merge of two groups, M_2 knows all blinded keys in its co-path so that it is able to compute all keys and blinded keys in its key-path.

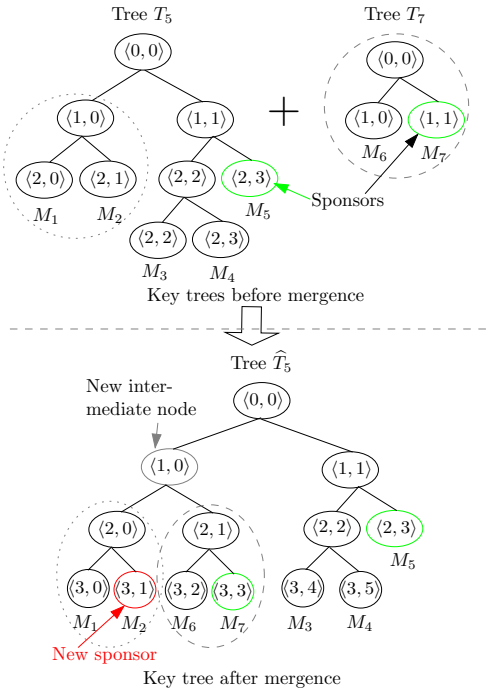


Figure 4.14.: An example of 2-group TGDH merge

The complexity characteristics of merge protocol for the worst case are summarized in Table 4.19.

[7] including the root's bkey

[8] Assume that M_j is in tree T^k before the merge:

- $\mu = \begin{cases} 2l'_{s_k}+1, & M_j \text{ is sponsor } \langle l'_{s_k}, v'_{s_k}\rangle; \\ l'_{s_k}+1-i, & \text{if } \exists i \in [0, l'_{s_k}] \text{ so that } M_j \in T^k_{\langle l'_{s_k}-i, v'_{s_k}/2^i\rangle} \setminus T^k_{\langle l'_{s_k}-i+1, v'_{s_k}/2^{i-1}\rangle}. \end{cases}$
- $\nu = \begin{cases} 2l_{s_1}-1, & M_j \text{ is sponsor } \langle l_{s_1}, v_{s_1}\rangle; \\ l_{s_1}+1-i, & \text{if } \exists i \in [0, l_{s_1}] \text{ so that } M_j \in T_{\langle l_{s_1}-i, v_{s_1}/2^i\rangle} \setminus T_{\langle l_{s_1}-i+1, v_{s_1}/2^{i-1}\rangle}. \end{cases}$

	$m = 2$	$m \geq 3$
Rounds	2	$\lceil \log_2 m \rceil$+1
Ucasts	0	0
Bcasts	3	$2m$
Exps/M_j [8]	$\mu + \nu$	$\mu + \beta$
Ucast size	0	0
Bcast size	$4(n + k - 1)$	$2(m + 1)(n + k - 1) + m(m - 2)$
Serial Exps	3·max(l'_{s_1}, l'_{s_2})+3$\widehat{h} - 2$	3·max($l'_{s_1}, \cdots, l'_{s_m}$)+(3$\widehat{h} - 2$)

Remark: $n(k)$: number of members in the highest tree (all other trees), m: number of merging groups, $\langle l'_{s_k}, v'_{s_k} \rangle$ with $k \in [1, m]$: sponsor of tree T^k before merge, $\langle l_{s_k}, v_{s_k} \rangle$ with $k \in [1, t]$: sponsor in merge.

Table 4.19.: Complexity of TGDH merge protocol (worst case)

4.4.5. Partition protocol

Assume that we have a group of n members and k of them leave the group. In the first round, every remaining member updates its tree by deleting all partitioned members as well as their respective parent nodes. In other words, if all leaf nodes of a subtree leave the group, the root node of this subtree is marked as leaving (namely the whole subtree is marked as leaving) and its leaf nodes are removed from the leaving nodes list. For each leaving node we identify a sponsor using the same criteria as described in Section 4.4.3.

The process of partition protocol is illustrated in Protocol 4.17.

Protocol 4.17 (TGDH partition protocol)

1. *Every member*
   - *updates key tree by deleting all leaving member nodes and their parent nodes,*
   - *removes all keys and bkeys from sponsors to the root node.*

   The shallowest rightmost sponsor additionally updates its share.

2. *Round 1 to q (Repeat this step until any sponsor computes the group key):*
   Each sponsor M_{s_i}
   - *computes all keys and bkeys in the key-path as far as possible,*
   - *broadcasts updated tree including all bkeys.*

   $M_{s_i} \xrightarrow{\widehat{T}_{s_i}(BK^*_{s_i})} M_*$

3. *Every member computes the group key using $\widehat{T}_{s_i}(BK^*_{s_i})$.*

An example of TGDH partition is given in Figure 4.15. A group of seven members $M_1, \cdots, M_7$ is partitioned. From the perspective of M_5, members M_2, M_4, M_6 and M_7 leave the group, so M_5 is in tree $\widehat{T_5}$ after the partition, the same for M_2, M_4, M_6 and M_7. However from the perspective of M_4, the leaving members are M_1, M_3, and M_5. So M_4 is in tree $\widehat{T_4}$ after the partition. The same for M_2, M_6, and M_7.

The complexity characteristics of partition protocol (worst case) are summarized in Table 4.20.

- $\beta = \begin{cases} 2l_{s_k} - 1, & M_j \text{ is sponsor } \langle l_{s_k}, v_{s_k} \rangle \text{ ;} \\ max(\rho_{1_j}, \cdots, \rho_{t_j}), & \text{for other members, where } 1 \leq m \leq t, \text{ and} \end{cases}$

$$\rho_{m_j} = l_{s_m} + 1 - i, if\ \exists i \in [0, l_m]\ so\ that\ M_j \in T_{\langle l_{s_m} - i, v_{s_m}/2^i \rangle} \setminus T_{\langle l_{s_m} - i + 1, v_{s_m}/2^{i-1} \rangle}$$

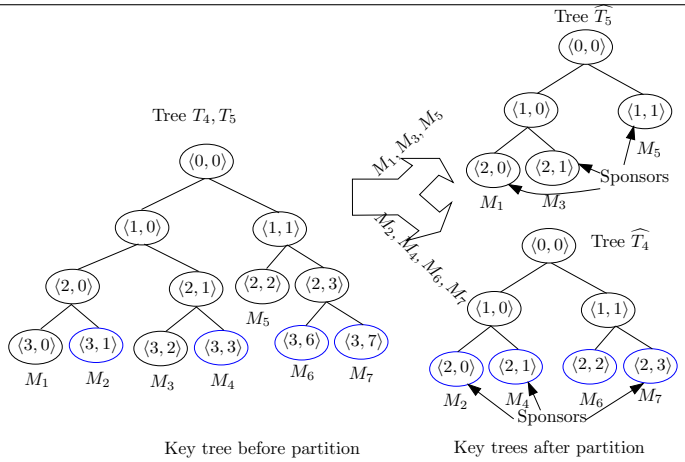


Figure 4.15.: An example of TGDH partition

Rounds	$min(\lceil \log_2 p \rceil + 1, b)$
Ucasts	0
Bcasts	min($2p$,$\lceil \frac{n}{2} \rceil$)
Exps/M_j	$2l_{s_1}$ for sponsor $\langle l_{s_1}, v_{s_1} \rangle$, $2l_{s_m} - 1$ for sponsor $\langle l_{s_m}, v_{s_m} \rangle$, $m \in [2, t]$, max($\rho_{1_j}, \cdots, \rho_{t_j}$) for other members.
Ucast size	0
Bcast size	$2(m-1) \cdot min(2p, \lceil \frac{n}{2} \rceil)$
Serial Exps	$3\widehat{h}$
Remark: m: number of members in updated tree, p: number of leaving members, $\langle l_{s_1}, v_{s_1} \rangle, \cdots \langle l_{s_t}, v_{s_t} \rangle$: sponsors, the first is the rightmost shallowest sponsor.	
Remark: $\rho_{m_j} = l_{s_m} + 1 - i$ if $\exists i \in [0, l_{s_m}]$ so that $M_j \in T_{\langle l_{s_m}-i, v_{s_m}/2^i \rangle} \setminus T_{\langle l_{s_m}-i+1, v_{s_m}/2^{i-1} \rangle}$, for $1 \leq m \leq t$.	

Table 4.20.: Complexity of TGDH partition protocol

4.4.6. Refresh protocol

No refresh protocol is defined in TGDH protocol. With respect to other protocols of TGDH protocol suite, we suggest in the following a refresh protocol. Assume that there is a group with key tree T when a refresher M_r refreshes its share. The process is illustrated in Protocol 4.18.

Protocol 4.18 (TGDH refresh protocol)

1. *Round 1:*
   The refresher M_r
   - *updates its share,*
   - *computes newly all keys and bkeys in its key-path,*
   - *broadcasts the updated tree with all bkeys.*

   $M_r \qquad \xrightarrow{\qquad T_r(BK_r^*) \qquad} \qquad M_*$

2. *Each member except refresher M_i*
   - *removes all keys and bkeys from the refresher M_r to root node,*
   - *updates the blinded keys,*

- *computes the group key.*

The complexity characteristics are summarized in Table 4.21.

Rounds	1
Ucasts	0
Bcasts	1
Exps/M_j	$2l_r$ for refresher, $l_r + 1 - i$ if $\exists i \in [0, l_r]$ so that $M_j \in T_{\langle l_r - i, v_r/2^i \rangle} \backslash T_{\langle l_r - i + 1, v_r/2^{i-1} \rangle}$
Ucast size	0
Bcast size	$2n - 2$
Serial Exps	$3l_r$
Remark: $\langle l_r, v_r \rangle$: refresher	

Table 4.21.: Complexity of TGDH refresh protocol

4.5. Complexity Analysis of Group Key Agreement Protocols

This section analyzes the memory, communication, and computation costs for all protocols described in this chapter. Many symbols are used and are defined throughout this chapter. For simplicity, we list them again in Figure 4.16.

n	number of members in current group, or in the largest group in merge
m	number of merging groups
p	number of leaving groups
k	number of members in all non-largest groups in merge
d	$d = \lceil \log_2 n \rceil$, used in 2^d-cube and 2^d-octopus
h	height of initial TGDH tree
$\widehat{h}$	height of updated TGDH tree
α	sum of height of all TGDH trees except the highest tree T^1 in merge
s	index of sponsor in updated tree in STR
$\langle l_s, v_s \rangle (\langle l_{s_i}, v_{s_i} \rangle)$	sponsor(s) node in updated tree in TGDH

Figure 4.16.: Notation

Among the various costs that we can consider, we focus on the number of stored keys and blinded keys, the number of rounds, the total number of unicast and broadcast messages, the cumulative unicast and broadcast message size, and the serial number of exponentiations. For serial costs operations that are performed by members in parallel are counted as one operation. In this case the highest costs are considered. The total costs are represented by the sum of all participants' costs in a given round (or protocol).

4.5.1. Memory costs

Table 4.22 gives an overview of the memory costs of all protocols. No wonder that the memory costs of protocols 2^d-cube, 2^d-octopus, and Asokan-Ginzboorg are only 1 key. Members in those protocols do not need to store the intermediate keys and their public values so that they are not able to deal with the group changes. So we consider the other three protocols. In CLIQUES each member needs only to store its session

random and the group key, and $n+1$ additional public keys. Members in TGDH and STR have to store all keys in their key-paths and $2n-2$ public keys. In STR the number of keys in the key-path depends on the location of the member, the deeper member needs to store more keys. The average value is $\frac{n+3}{2}$. In TGDH it depends on the level of the member. A member in a deeper level needs to store more keys. Averagely $\lceil log_2^n \rceil + 1$ keys are in the key-path.

As a whole, CLIQUES has the lowest memory cost, whereas STR the highest.

Protocol		keys	public keys
2^d-cube	concretely	1	0
2^d-octopus	concretely	1	0
Asokan-Ginzboorg	concretely	1	0
CLIQUES	concretely	2	$n+1$
STR	per LN_l	$n+2-max(l,2)$	$2n-2$
	averagely	$\frac{n+3}{2}$	$2n-2$
TGDH	per $\langle l, v \rangle$	$l+1$	$2n-2$
	averagely	$\lceil \log_2^n \rceil + 1$	$2n-2$

Table 4.22.: Memory cost of protocols 2^d-cube, 2^d-octopus, Asokan-Ginzboorg, CLIQUES, STR, and TGDH

4.5.2. Communication and computation costs

Table 4.23 gives an overview of communication and computation costs for all protocols except Asokan-Ginzboorg. This is because Asokan-Ginzboorg considers faulty members unlike other protocols, which makes it incomparable with other protocols. Since 2^d-cube and 2^d-octopus provide no protocols for dynamic group operations, only the complexity characteristics of operation *setup* for both are considered. While for all other three protocol suites we analyze additionally the costs of all dynamic group operations (*join*, *leave*, *merge*, *partition*, and *refresh*).

The communication and computation overheads of the TGDH protocol depend on the tree height, the balance of the key tree, the location of the joining tree, and the leaving nodes. In our analysis, we assume the worst case configuration and list the worst-case costs for TGDH. The number of modular exponentiations in STR after a leave event depends on the location of the deepest leaving node (the sponsor). Hence we compute the average cost, i.e. the case when the sponsor is the $\frac{n-p}{2}$-th node in the updated tree. All other protocols, except TGDH and STR, show exact costs. We compare in the following the six operations for CLIQUES, STR and TGDH.

Setup STR is the most communication efficient protocol. Only two rounds are needed to initialize a new group. CLIQUES is the most expensive communication protocol, it requires $n+1$ rounds. TGDH is comparatively efficient, the number of rounds scales logarithmically in the number of members.

We consider now the number of messages and the message size. STR requires only $n+1$ messages, whereas CLIQUES and TGDH require approximately twice as many messages. The cumulative message size of STR is also the lowest, only $2n-1$ blinded keys are contained in the messages. CLIQUES requires approximately 50% more cumulative message size, $3n-2$. TGDH consumes much more bandwidth. Considering a balanced TGDH tree, $h = \lceil \log_2 n \rceil \approx \log_2 n + 1$. The total number of sent blinded keys is $n + (h-2)2^{h+2} + 8 = n + (\log_2 n - 1)2^{\log_2 n + 3} + 8 = 8n \log_2 n - 7n + 8$.

With respect to the serial modular exponentiations, TGDH is the most computation efficient, the number of serial modular exponentiations scales logarithmically in the number of members. CLIQUES and STR require a linear number of serial exponentiations relative to the group size, and STR consumes 50% more serial exponentations.

Protocol Suite	Protocol	Rounds	Ucasts	Bcasts	Ucast size	Bcast size	Serial Exps
2^d-cube	setup	d	nd	0	nd	0	$2d-1$
2^d-octopus	setup	$d+2$	$3(n\text{-}2^d)$ $+2^d{\cdot}d$	0	$3(n\text{-}2^d)$ $+2^d{\cdot}d$	0	$2d+3$
CLIQUES	IKA.2	$n+1$	$2n-3$	2	$2n-3$	$n+1$	$2n+1$
	Join	2	1	1	$n+1$	$n+1$	$2n+1$
	Leave	1	0	1	0	$n-1$	$n-1$
	Merge	$k+1$	k	1	$(k^2{+}2nk$ $+k)/2$	$n+k$	$(k^2{+}2nk{+}$ $k{+}2n)/2$
	Partition	1	0	1	0	$n-p$	$n-p$
	Refresh	1	0	1	0	n	n
STR	setup	2	0	$n+1$	0	$2n-1$	$3n-2$
	Join	2	0	2	0	$2n$	6
	Leave	1	0	1	0	$n-1$	$3n/2$
	2-merge	2	2	1	$2(n{+}k\text{-}2)$	$2(n{+}k\text{-}1)$	$3k+3$
	m-merge	2	0	$m+1$	0	$4(n{+}k)$ $\text{-}2m\text{-}1$	$3k+3$
	Partition	1	0	1	0	$n-p$	$3(n\text{-}$ $p)/2{+}3$
	Refresh	1	0	1	0	n	$3n/2{+}3$
TGDH	setup	h	0	$h^2\text{-}h{+}n$	0	$(h\text{-}2)2^{h+2}$ $+n{+}8$	$2h-1$
	Join	2	0	2	0	$2n+2$	$3h$
	Leave	1	0	1	0	$2n-4$	$3h$
	2-merge	2	0	3	0	$4(n{+}k\text{-}1)$	$3(h{+}\widehat{h})\text{-}2$
	m-merge	$\lceil\log_2 m\rceil{+}1$	0	$2m$	0	$2(m{+}1)(n{+}$ $k){+}m(m\text{-}2)$	$3(h{+}\widehat{h})\text{-}2$
	Partition	$\min(\widehat{h},$ $\lceil\log_2 p\rceil{+}1)$	0	$\min(2p,$ $\lceil\frac{n}{2}\rceil)$	0	$2(n\text{-}p\text{-}1){\cdot}$ $\min(2p,\ \lceil n/2\rceil)$	$3\widehat{h}$
	Refresh	1	0	1	0	$2n-2$	$3h$

Table 4.23.: Communication and computation costs of protocols 2^d-cube, 2^d-octopus, CLIQUES, STR, and TGDH

Join All protocols require 2 rounds and 2 messages. The cumulative size for all protocols is approximately same. While STR and TGDH requires 2 broadcast messages, 1 unicast message and 1 broadcast message are required by CLIQUES. Hence CLIQUES has more communication efficiency than other protocols.

However, CLIQUES is the most expensive in terms of computation, requiring a linear number of exponentiations relative to the group size. TGDH is comparatively efficient, the number of serial modular exponentiations scales logarithmically in the number of members. STR, in turn, requires a constant number of modular exponentiations.

Leave All protocols require 1 rounds and 1 broadcast message. CLIQUES and STR require the same cumulative message size, approximately the group size.

However, TGDH is most computation efficient, the number of serial modular exponentiations scales logarithmically in the number of members. STR and CLIQUES scale linearly in the group size. The number of serial exponentiations of STR is $\frac{3n}{2}$, and of CLIQUES is $2n$.

Merge We first look at the communication costs. Note that the number of communication rounds in CLIQUES scales linearly in the number of the members added to the group, whereas STR is more efficient having a constant number of rounds. Although a merge for TGDH takes multiple rounds, it depends on the number of merging groups which is mostly small.

The message size of CLIQUES, approximately $\frac{k^2+2nk}{2}$, is much larger than in other protocol suites. The merge of STR and TGDH is further divided into 2-group merge (2-merge) and $m-$group merge ($m-$merge). Considering first 2-merge, the number of rounds, of messages, and message size are nearly the same in STR and TGDH. The difference is that two messages in STR are unicast messages. So STR is more communication efficient than TGDH.

Considering m-merge, STR is also more communication efficient than TGDH. Only 2 rounds are required, like in 2-merge, while the number of rounds in TGDH scales linearly in the number of merging groups. With respect to the number of messages and the cumulative message size, STR is also more efficient than TGDH.

So we can conclude that STR is most suitable to handle the merge event with respect to communication costs.

In the following we consider the computation costs. TGDH is most efficient, the number of serial modular exponentiations scales logarithmically in the number of the product of the largest group's size and the number of all members. In STR, it scales in the number of all non-deepest groups. CLIQUES is further the most inefficient protocol to handle the merge event, approximately $\frac{k^2+2nk}{2}$ serial exponentiations are needed.

Partition Table 4.23 shows that the CLIQUES and STR protocols are bandwidth efficient: only one round consisting of one message, and both protocols are of the same message size, which equals the number of remaining group members. While partition is the most expensive operation in TGDH, requiring a number of rounds bounded by at most the minimum of the updated tree's height and $\log_2 p + 1$. The message size in TGDH is also considerable. Hence STR and CLIQUES are most suitable to handle partition event with respect to communication costs.

With respect to the computation costs, TGDH requires a logarithmic number of exponentiations, while CLIQUES and STR scale linearly in the group size. STR requires approximately 50% more computation costs than CLIQUES. Hence TGDH is most suitable with respect to computation costs.

Refresh The costs to handle refresh event are similar to those to handle leave event. Hence we refer the reader to the paragraph analyzing the costs for leave.

4.5.3. Discussion summary

As discussed above, the STR protocol is most suitable in networks where high network delays dominate. However, its computation costs are most expensive. CLIQUES consumes less memory costs compared to STR and TGDH. However, in setup and merge, it takes much more rounds. The computation costs of TGDH in all protocols is much less than those of CLIQUES and STR. The only exception is in protocol join, where STR requires only 6 serial exponentiations. The major disadvantage of TGDH is the great number of required rounds to handle the partition.

5. Protocols for Ad Hoc Networks

In recent years some protocols are suggested for ad hoc networks. Two of those are suggested by Asokan *et al.* in [8] and Hietalahti in [41]. However, most of them are not able, and it is unclear, how they can be extended to handle dynamic group operations. However, such operations are indispensable for ad hoc networks. The topology for key agreement may change voluntarily or involuntarily, e.g. partition when the network failure occurs, or merge when the network fault heals.

In this chapter we present how could STR and TGDH be optimized for ad hoc networks. The optimized versions are denoted *Ad Hoc STR* (*μSTR*) and *Ad Hoc TGDH* (*μTGDH*) respectively. Additionally we propose a *Tree-based group key agreement Framework for Ad-hoc Networks* (*TFAN*) based on *μSTR* and *μTGDH*.

We have implemented all three protocols. The description of TFAN API can be found in Section 6. The classes in μSTR API and μTGDH API are overviewed in Appendix A.3 and A.4 respectively. For details please refer to their javadoc documents.

All protocol suites discussed in this chapter are optimized in the following directions:

1. Using Tree-based Diffie-Hellman over groups of elliptic points (EC-TbDH) key exchange, instead of Tree-based Diffie-Hellman (TbDH), to reduce computation costs. Each node N is associated with the key k_N and the public key $pk_N = g(k_N)$, where $g(\cdot)$ is multiplication with the base point in elliptic curve E over field $\mathbb{F}_p$ or $\mathbb{F}_{2^m}$, i.e. $g(k) = kG$, where $G(x_G, y_G)$ is the base point of elliptic curve E with x-coordinate x_G and y-coordinate y_G. Let N be an internal node with two children, the left N_l and the right N_r. To compute k_N, one needs to know the key associated with one child, and the public key associated with the other. k_N could be computed as follows:

$$\begin{aligned} k_N &= f(k_{N_l} \cdot pk_{N_r}) \\ &= f(pk_{N_l} \cdot k_{N_r}) \\ &= f(k_{N_l} \cdot k_{N_r} \cdot G) \end{aligned}$$

   Where $f(\cdot)$ is a function which transform a point $P(x_P, y_P)$ to a valid secret for next step, e.g. $f(P(x_P, y_P)) = x_P \bmod q$, where q is a prime order of the base point in the elliptic curve E.

   Since EC-TbDH is applied, the most expensive operation is the scalar-point-multiplication (Mul). This operation is cheaper than modular exponentiations in $\mathbb{Z}_p^*$, if properly implemented.

2. Minimizing the number of serial multiplications. Some randoms are generated and their public values are then computed before the member joins to the group. These values are stored in a table, denoted by **random-depot**. If the member needs to update the share, it picks up the first random from random-depot which differs the current share, together with its public value. Additionally the sponsor computes the group key always after the sending of the message in parallel with other members so that one serial multiplication can be saved.

3. Minimizing the message size. After setup of the key tree, for each update (*join*, *leave*, *merge*, *partition*, *and refresh*) only the necessary, instead of all, blinded keys are broadcasted so that the message size can be significantly reduced. We define here the *update-list* of member *M* which stores the nodes whose bkeys should be computed and broadcasted by *M*. The message size is also reduced due to ECC.

4. Minimizing the memory costs by removing redundant information and switching to ECC.

All protocol suites described in this chapter are formed by the following six basic protocols: *setup*, *join*, *leave*, *merge*, *partition* and *refresh*. All protocols share a common framework with the following features:

- Each group member contributes a share to the group key, which is computed as a function of all current group members' shares. Each contribution is equally important.
- Each group key is secret and known only to all current group members.
- As the group grows, one current member, denoted the *sponsor*, changes its share, and the new group key is a function of all current members' shares (including joined members).
- As the group shrinks, leaving members' shares are removed from the new group key, and one remaining member, denoted the *sponsor*, changes its share. The new group key is a function of the sponsor's new share and all other remaining members' shares.
- According to the security policy, a member, denoted *refresher*, changes its share. The sponsor's new share is then factored into the new group key.
- All protocol messages are signed by the sender. All cryptographic strong public key signature schemes can be used. However, to save the computation costs, digital signature schemes using elliptic curve, e.g. ECDSA, are used.

Due to the absence of PKI in ad hoc networks, some other techniques for authentication are suggested for the authentication in the protocol suites described in this chapter.

In heterogeneous ad hoc networks, all protocols described in this chapter can be further optimized. Since the sponsors have to compute not only all keys, but also all blinded keys in their key-paths, it may be undesirable of some nodes. The burden of such sponsors can be alleviated by partaking of other more powerful nodes in the computation. Considering the example in Figure 4.14. The sponsor M_2 updates its share, unicasts its public key to M_1, and let the latter compute the key-pairs in the key-path. M_1 broadcasts then the known bkeys in M_2's key-path. If the sponsor and the other more powerful member are not siblings, the sponsor should first compute the key-pairs that not in the other's key-path. If M_2 let M_4 partake the computation, M_2 must first compute key-pairs of $\langle 2, 0\rangle$.

The remainder of this chapter is organized as follows. Section 5.1 suggests some techniques for authentication. μSTR is discussed in Section 5.2, while μTGDH in Section 5.3, and TFAN in Section 5.4. In Section 5.5 we summarize the complexity characteristics of all protocols discussed in this chapter. The experimental results are summarized in Section 5.6.

5.1. Authentication

Our protocol suite uses public and authentic channel. Everyone, both the member and the adversary, can read the messages. And all messages are signed to guarantee the non-repudiation.

PKI will be used if it exists. However, unlike in traditional networks, no present PKI can be assumed in ad hoc networks. In following we first analyze two techniques suggested in the literature: password-based public key distribution (Section 5.1.1) and PKI for ad hoc networks (Section 5.1.2).

5.1.1. Password-based public key distribution

Every member knows some secret before he joins the group. A typical scenario is that the participants of a meeting wish to form a group. It is realistic to assume that they share some password **P**.

Assume that *n* members wish to form a group. Each member first picks up a key pair for generation/verification of signatures from its random-depot or generates it newly. The public key together with some other information are broadcasted. The detailed process is illustrated in Protocol 5.1. For clarity's sake, private key and public key are denoted sk and pk, respectively.

[1]The symmetric encryption schemes can be applied. And the corresponding key can be derived from P.

[2]It means that (M_i, C_i) is signed with the private key sk_i.

Protocol 5.1 (Password-based public key distribution)

1. *Round 1: Each member M_i, $i = 1, \cdots, n$*
   - *picks up a key pair (sk_i, pk_i),*
   - *encrypts pk_i and its own identity: $C_i = P(M_i, pk_i)$, [1]*
   - *signs (M_i, C_i) with sk_i: $S_i = sign_{sk_i}(M_i, C_i)$, [2]*
   - *broadcasts message (M_i, C_i, S_i) .*

   $M_i \qquad \xrightarrow{\qquad M_i, C_i, S_i \qquad} \qquad M_*$

2. *Each member M_j verifies (M_i, C_i, S_i) for all $i \in \{1, \cdots, n\} \setminus \{j\}$:*
   a) *decrypts C_i and compares the first part with M_i. If they are different, this message is rejected. Otherwise, picks up pk_i and,*
   b) *verifies the signature using pk_i gathered in step a. If correct, it accepts the with M_i associated public key.*

An adversary can surely replace the unencrypted identity. However, without knowing P, it is not able to form an encrypted message whose first part corresponds the identity. Protocol 5.1 is vulnerable to replay attack. An adversary can broadcast a legal message at some later time. To reduce or avoid such attacks, P must be refreshed periodically.

However, Protocol 5.1 is only suitable for static groups, but not for dynamic groups. The reason is that the secret P must be changed after every leaving event ($leave$, $partition$). But how can the new secret P be securely agreed? Since the old members know the same information as the remaining members, they can read any information protected with the old secret.

5.1.2. PKI for ad hoc networks

PKI has been recognized as one of the most successful and important tools for providing security for dynamic networks. However, due to the infrastructure-less nature of ad hoc networks, it is still unclear if the PKI-approaches in traditional networks can be extended to ad hoc networks.

To provide PKI functionality in ad hoc network, Yi *et al.* [128] present an efficient and effective communication protocol which is suitable for heterogeneous ad hoc networks. In this protocol threshold cryptography is applied to distribute the Certificate Authority (CA) functionality over specially selected nodes based on security and physical characteristics of the nodes. Such nodes are called MOCA(MObile Certificate Authority)s.

Assume that there is an ad hoc networks with N members. n of them are selected as MOCAs. They share the key of the unique CA, and k pairwise distinct nodes are needed to construct the full key. Any client requiring a certification service must contact at least k MOCAs. Each contacted MOCA generates a partial signature over the received data and sends it back. At least k such partial signatures must be selected to reconstruct the full signature and successfully receive the certification service. Unlike normal PKI, the certificates are stored in the holder's node. Hence the certificates can not be accessed from certificate depots.

However, the MOCAs have to stay continuously in the group, otherwise shares have to be redistributed, and the communication by issuing the certificate might be too expensive.

Another approach is to use the certificate with short period of validation, e.g. one month. The validity of certificates are not checked over CRL (Certificate Revocation List) or OCSP (Online Certificate Status Protocol), hence no online PKI is required. Advantages of this approach against MOCA is the saving of communication cost and easy management. Furthermore, it is suitable for dynamic groups, unlike the password based authentication described in 5.1. Hence this approach will be applied in our APIs.

Every node must be equipped with the CAs' certificates, and the CAs are set as the trust anchors so that the certificates of members can be verified. We assume that each member has certificates issued by the

trusted CAs. If some nodes wish to form a group, they broadcast their certificates in the first message so that their messages can be verified. To save one broadcast message, the certificate can be sent together with the public key and other informations needed for the group key agreement. To save the memory costs and to compute digital signatures efficiently, ECDSA is used.

In operation *join* the new member broadcasts its certificates. Current group members' certificates are sent by the sponsor to the new member. While in operation *merge* every sponsor broadcasts certificates of all members in its group so that they are known to all members in other groups. Since in operations *leave* and *partition* no new members are added, sending certificates is not required.

5.2. μSTR Protocol Suite

The STR protocol suite described in Section 4.3 has some redundant information. We suggest here an μSTR key tree which contains only the necessary information. In the μSTR tree, to compute the group key, member M_l does not need all blinded keys and blinded session randoms (summarized as bkeys in following), but only those in its co-path. Additionally M_l stores also the keys in its key-path. So the *memory costs* are $n-l+2$ keys and $n-l+1$ public keys for $l \geq 2$, and n keys and $n-1$ public keys for $l = 1$. Compared to the STR tree, $n+l-3$ public keys for $l \geq 2$, and $n-1$ public keys for $l = 1$ have to be saved.

The tree and the criterion to handle the tree to adapt the dynamic events in μSTR are same as in STR, hence they are omitted in this section.

5.2.1. Setup protocol

Compared to Protocol 4.7 of STR, setup protocol in μSTR is optimized as follows: The sponsor computes the root's key first after it broadcasts the message. The broadcast message consists of only a list of all blinded keys in M_1's key-path except br_1/bk_1, but not the whole tree with all bkeys. With this optimization the message size can be halved. Since M_2 knows also all bkeys in its co-path, M_1 and M_2 compute all keys in their key-paths in parallel. M_1 computes additionally the bkeys in its key-path and broadcasts the changed bkeys. Each member except M_1 picks up the bkeys in its co-path from the broadcast message. Equipped with the new bkeys it can compute the keys in its key-path. The detailed process is illustrated in Protocol 5.2.

Protocol 5.2 (μSTR setup)

1. *Round 1: Every member* $M_i, i \in \{1, \cdots, n\}$*:*
   $M_i \quad \xrightarrow{\qquad br_i \qquad} \quad M_*$
2. *Round 2: The member* M_1
   - *computes recursively* k_i *and* bk_i, $\forall i \in [2, n-1]$,
   - *broadcasts all blinded keys* bk_i, $\forall i \in [2, n-1]$.

   $M_1 \quad \xrightarrow{\qquad \{bk_j | 2 \leq j \leq n-1\} \qquad} \quad M_*$

   The member M_2 *computes recursively* k_i, $\forall i \in [2, n]$.
3. *Every member* M_i*,*
   - *if* $i = 1$*, computes* k_n*,*
   - *if* $i \in [3, n]$*, computes recursively* k_j, $\forall j \in [i, n]$.

Looking at the STR key tree with n members in Figure 4.5, M_1 broadcasts the bkeys bk_i, $\forall i \in [2, n-1]$. M_5 picks up the bkey in its co-path, i.e. bk_4. After updating the bkeys all members can compute the group key.

The complexity characteristics of setup protocol are summarized in Table 5.1.

Rounds	2
Bcasts	$n+1$
Muls/M_i	$2n-1$ for $i=1$,
	$n-i+1$ for $2 \leq i \leq n$,
Bcast size	$2n-2$
All Muls	$\frac{n^2+3n-2}{2}$
Serial Muls	$3n-4$

Table 5.1.: Complexity of μSTR setup protocol

5.2.2. Join protocol

As in STR, in the first round, the potential new member broadcasts a join request with the blinded key. The criteria to choose the sponsor and to update the tree is unchanged. After the update of the tree and the computation of key-pairs the sponsor broadcasts the updated tree. Only the bkeys in the sponsor's key-path (always br_{n-1} and bk_{n-1}) are broadcasted, instead of all blinded keys. With this optimization the message size is significantly reduced. Only two bkeys in μSTR are broadcasted, while in STR $2n-1$ bkeys. Every member in the previous group replaces the sponsor's bkey br_{n-1}, the new member picks up the bkey bk_{n-1}. Now all members can compute the new group key. The detailed process is illustrated in Protocol 5.3.

Protocol 5.3 (μSTR join)

1. *Round 1: The new member broadcasts its blinded session random.*
   M_s $\xrightarrow{\qquad br_{n+1} \qquad}$ M_*
2. *Round 2: Each member*
   - *updates the tree by inserting M_{n+1} as described in 4.3.2,*
   - *sets $n = n+1$.*

   The sponsor M_{n-1} additionally,
   - *updates its session random r_{n-1} and blinded session random br_{n-1},*
   - *computes new k_{n-1}, bk_{n-1},*
   - *broadcasts bk_{n-1} and br_{n-1}.*

   M_s $\xrightarrow{\qquad bk_{n-1}, br_{n-1} \qquad}$ M_*
3. *Every member M_i,*
   - *if $i = s$ (sponsor), computes k_n,*
   - *if $i = n$ (new member), picks up bk_{n-1} and computes k_n.*
   - *if $i \in [1, n-2]$, picks up br_{n-1} and computes k_{n-1} and k_n.*

Looking at the example in Figure 4.6, the sponsor M_4 broadcasts the bkeys br_4 and bk_4. The members M_1, M_2, and M_3 replace the old br_4 with the new one. They compute then k_4 and k_5. M_5 picks up bk_4 and computes then k_5.

The complexity characteristics of the join protocol are summarized in Table 5.2.

5.2.3. Leave protocol

Assumed that we have again a group of n members, and the member M_l leaves the group. Each remaining member updates the key tree by removing the leaving member. We consider now the updated tree. If the sponsor is M_1, the blinded keys $bk_i, i \geq 2$ and br_1 will be broadcasted. Otherwise, the blinded keys $bk_i, i \geq s$ and br_s will be broadcasted. The detailed process is illustrated in Protocol 5.4.

Rounds	2
Bcasts	2
Muls/M_i	3 for sponsor, 1 for new member, 2 for other members.
Bcast size	3
All Muls	$2n + 2$
Serial Muls	4

Table 5.2.: Complexity of μSTR join protocol

Protocol 5.4 (μSTR leave)

1. *Round 1: Every member*
   - *updates the tree by removing the leaving member* M_l,
   - *sets* $n = n - 1$.

   The sponsor M_s [2] *additionally*
   - *updates its session random* r_s,
   - *computes* k_i *and* bk_i, $\forall i \in [max(2, s), n - 1]$,
   - *broadcasts* br_s *and* bk_i, $\forall i \in [max(2, s), n - 1]$.

   M_s $\xrightarrow{\{bk_j | max(2,s) \leq j \leq n-1\} \cup \{br_s\}}$ M_*

2. *The sponsor computes* k_n, *and each other member* M_i *picks up the bkey in its key-path,*
   - *if* $i < s$, *computes* k_j, $\forall j \in [s, n]$,
   - *if* $i > s$, *computes* k_j, $\forall j \in [i, n]$.

Considering the example in Figure 4.7. The sponsor M_2 updates r_2 and br_2, computes then k_2 and bk_2. It broadcasts then br_2 and bk_2. M_1 picks up br_2, and M_3 picks up bk_2. Now all remaining members can compute the updated group key.

The complexity characteristics of leave protocol are summarized in Table 5.3.

	$s = 1$	$s \geq 2$
Rounds	1	1
Bcasts	1	1
Muls/M_i	$2n - 5$ for sponsor $n - i$ for other members	$2n - 2s - 1$ for sponsor $n - i$ for M_i with $i > s$ $n - s$ for M_i with $i < s$
Bcast size	$n - 2$	$n - s$
All Muls	$\frac{n^2+n-8}{2}$	$\frac{n^2+n-s-s^2-2}{2}$
Serial Muls	$3n - 8$	$3n - 3s - 2$
Remark: M_s: sponsor in updated tree		

Table 5.3.: Complexity of μSTR leave protocol

[2] $s = l - 1$ if $l > 1$ or $s = 2$ if $l = 1$.

5.2.4. Merge protocol

In the original merge protocol of STR, the sponsor broadcasts trees with all blinded keys and blinded session randoms. The communication cost can be significantly reduced by broadcasting the tree with only the blinded session randoms.

Assume that m trees $T^i, i \in [1, m]$ participate in the merge protocol. The size of tree T^i is denoted n_i with $n_1 \geq n_2 \cdots \geq n_m$. For sake of the clarity, we define n to be n_1, and $k = \sum_{i=2}^{m} n_i$.

First the tree must be broadcasted. In STR it is the work of the topmost member. However, in μSTR, all members know only the blinded session randoms in its co-path, and only M_1 and M_2 know all blinded session randoms. Hence M_1 is chosen as the broadcast sponsor. It broadcasts the tree with all blinded session randoms. After the collection of all merge requests, all members sort the trees as described in Section 4.3.4. The criteria to update the tree and to choose the sponsor is also same as that in Section 4.3.4.

For sake of clarity we define the broadcast sponsor of tree T^i as M_{s_i}, and the sponsor to broadcast the bkeys in the updated tree as M_s. After merging the tree, the sponsor M_s updates its share and computes all keys and bkeys in its key-path, it broadcasts then all bkeys in its key-path. All other members pick up the bkeys in their co-paths from the broadcast message. Since all members have all bkeys in their co-path, they can compute the new group key. In fact the broadcasted blinded session randoms of T^1 are never used. However before the collection of all merge requests, nobody knows which tree is the highest, namely T^1. So the communication costs can not be reduced.

The detailed process of merging m groups is illustrated in Protocol 5.5.

Protocol 5.5 (μSTR m-group merge)

1. *Round 1: Every broadcast sponsor M_{s_i} broadcasts tree T^i with all blinded session randoms. For i with $1 \leq i \leq m$:*
   M_{s_i} $\xrightarrow{\qquad T^i_{s_i}(BR^*_{s_i}) \qquad}$ M_*

2. *Round 2: Every member updates the tree by merging all merging trees.*
   The topmost leaf node of the deepest tree M_n will be selected as sponsor M_s. It additionally
   - *updates its session random r_n and br_n,*
   - *computes k_i and bk_i, $\forall i \in [n, n+k-1]$,*
   - *broadcasts br_n and $bk_i, \forall i \in [n, n+k-1]$.*

   M_n $\xrightarrow{\qquad \{bk_j | n \leq j \leq n+k-1\} \cup \{br_n\} \qquad}$ M_*

3. *The sponsor computes k_{n+k}, each other member M_i:*
   - *if $i < n$, picks up br_n and computes k_j for $n \leq j \leq n+k$,*
   - *if $i > n$, picks up bk_{i-1} and computes recursively k_j for $i \leq j \leq n+k$,*

The example in Figure 5.1 illustrates an example of merge with 3 trees. First the broadcast sponsors M_{s_1}, M_{s_2}, and M_{s_3} broadcast the trees T^1, T^2, and T^3 with all blinded session randoms respectively. Since M_4 is the topmost member in the original tree T^1, it is chosen as the sponsor. After the update of its share and the computation of keys and bkeys in its key-path, M_4 broadcasts $br_4, bk_4, bk_5, bk_6, bk_7$, and bk_8. The members in original tree T^1, M_1, M_2, and M_3, picks up br_4, and $M_l, l > 4$ picks up bk_{l-1}. Now all members have enough bkeys and can compute the group key.

The complexity characteristics of m-group merge protocol are summarized in Table 5.4.

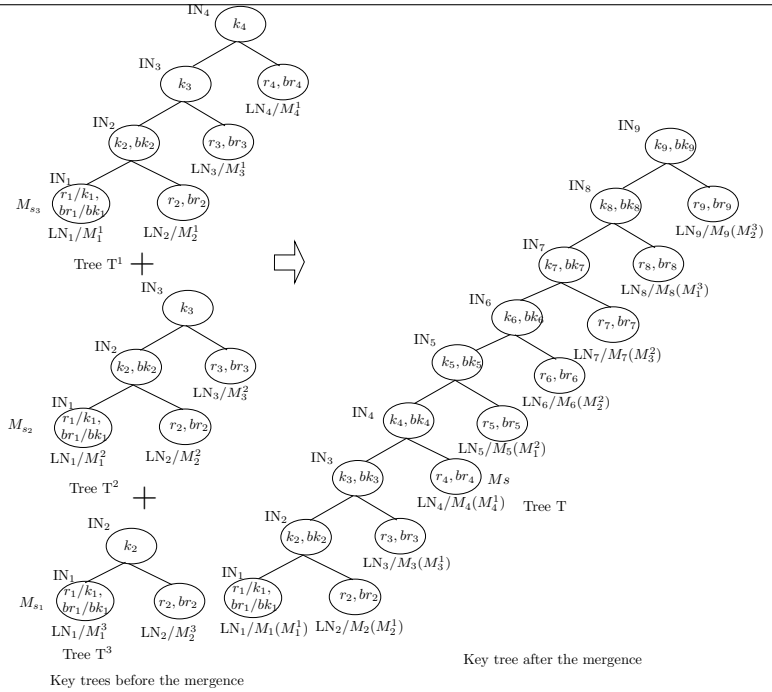


Figure 5.1.: An example of μSTR merge

Rounds	2
Bcasts	$m+1$
Muls/M_i	$2k+1$ for M_n
	$k+1$ for $M_i, i<n$
	$n+k+1-i$ for $M_i, i>n$
Bcast size	$2n+k+1$
All Muls	$\frac{k^2+(2n+3)k+2n}{2}$
Serial Muls	$3k+1$
Remark: $n(k)$: number of members in the group with highest tree(all other groups), m: number of merging groups	

Table 5.4.: Complexity of μSTR m-group merge protocol

5.2.5. Partition protocol

In the partition protocol we remove some blinded keys from the broadcast message. The criteria to update the tree and to choose the sponsor is the same as in Section 4.3.5. After update of the tree, the sponsor updates its session random and blinded session random, it computes then all keys and bkeys in its key-path. It broadcasts then all bkeys in its key-path, instead of the updated tree with all bkeys.

Let M_s be the sponsor, n be the size of initial group, and p be the number of leaving members. The detailed process is illustrated in Protocol 5.6.

Protocol 5.6 (μSTR partition)

1. *Round 1: Every member updates the tree by removing all leaving members.*
   The sponsor M_s additionally,
   - *updates its session random r_s and blinded session random br_s,*
   - *computes newly k_i and bk_i, $\forall i \in [max(2,s), n-p-1]$,*

- *broadcasts* br_s *and* bk_i, $i \in [max(2,s), n-p-1]$.

M_s $\xrightarrow{\{bk_j | max(2,s) \le j \le n-1\} \cup \{br_s\}}$ M_*

2. *The sponsor computes* k_{n-p}, *each other member* M_i,
   - *if* $i < s$, *picks up* br_s *and computes* k_j, $\forall j \in [s, n-p]$,
   - *if* $i > s$, *picks up* bk_{i-1} *and computes* k_j, $\forall j \in [i, n-p]$.

Looking at the example in Figure 4.9, in the resulted tree with four members. The sponsor M_3 updates r_3 and br_3, after the computation it broadcasts br_3 and bk_3. M_1 and M_2 pick up then br_3 from the message, and M_4 picks up bk_3.

The complexity characteristics of partition protocol are summarized in Table 5.5.

	$s = 1$	$s \ge 2$
Rounds	1	1
Bcasts	1	1
Muls/M_i	$2m - 3$ for sponsor	$2(m-s)+1$ for sponsor
	$m - i + 1$ for other members	$m - i + 1$ for M_i with $i > s$
		$m + 1 - s$ for M_i with $i < s$
Bcast size	$m - 1$	$m - s + 1$
All Muls	$\frac{m^2+3m-6}{2}$	$\frac{m^2+3m-s-s^2}{2}$
Serial Muls	$3m - 5$	$3(m-s)+1$
Remark: p: number of leaving members, $m := n - p$, LN_s: sponsor in updated tree		

Table 5.5.: Complexity of μSTR partition protocol

5.2.6. Refresh protocol

In CLIQUES, the member M_r which is the least recent to have refreshed its key share generates a new share $\widehat{r_r}$ and broadcasts the updated information. However, it is difficult to manage the last refresh time of all members, and if the tree structure is changed, no refresh is needed. Hence we define the refresh policy as follows: first a maximal allowed interval with the same group key should be configured. If the tree structure is within this interval unchanged, then the rightmost member (in STR is always M_n) executes the refresh protocol.

The size of the broadcast message can be reduced by removing the unnecessary blinded keys. The detailed process is illustrated in Protocol 5.7.

Protocol 5.7 (μSTR refresh)

1. *Round 1: The refresher* M_r
   - *updates its session random* r_r *and* br_r,
   - *computes (*k_i, bk_i*) for* $r \le i \le n-1$,
   - *broadcasts* $bk_{max(2,r)}, \cdots, bk_{n-1}, br_r$.

M_r $\xrightarrow{bk_{max(2,r)}, \cdots, bk_{n-1}, br_r}$ M_*

2. *The refresher computes* k_n, *and each other member* M_i,
   - *if* $i < r$, *picks up* br_r *and computes* k_j, $\forall j \in [r, n]$,
   - *if* $i > r$, *picks up* bk_{i-1} *and computes* k_j, $\forall j \in [i, n]$.

The complexity characteristics of refresh protocol are given in Table 5.6.

	$r = 1$	$r \geq 2$
Rounds	1	1
Bcasts	1	1
Muls/M_i	$2n - 3$ for refresher	$2(n - r) + 1$ for refresher
	$n - i + 1$ for another members	$n - i + 1$ for M_i with $i > r$
		$n - r + 1$ for M_i with $i < r$
Bcast size	$n - 1$	$n - r + 1$
All Muls	$\frac{n^2+3n-6}{2}$	$\frac{n^2+3n-r-r^2}{2}$
Serial Muls	$3n - 5$	$3(n - r) + 1$
Remark: LN_r: refresher		

Table 5.6.: Complexity of μSTR refresh protocol

5.3. μTGDH Protocol Suite

The TGDH protocol suite described in Section 4.4 has some redundant information. We suggest in this section an μTGDH key tree which contains only the necessary information. In fact a member M_i needs only to know the blinded keys in its *co-path* and its own share and bkey. Looking at the setting in Figure 4.10, only the bkeys of members M_1 and M_3, and internal node $\langle 1, 1\rangle$ are necessary for member M_2 to compute the group key. To save the computational cost all keys in the key-path are also stored in the tree.

One of the major advantages of this key tree is to save memory costs. Assume that we have an μTGDH tree with n members, and its height is h. A member associated with leaf node $\langle l, v\rangle$ stores $l + 1$ keys $k_{\langle l-i,\lfloor v/2^i \rfloor\rangle}$ for $0 \leq i \leq l$, $bk_{\langle l,v\rangle}$, and l blinded keys in its *co-path*. Hence the memory costs is $l + 1$ keys and $l + 1$ public keys. Compared to TGDH, μTGDH requires $2n - 2 - (l + 1) = 2n - l - 3$ public keys less to be saved.

Looking at the example in Figure 4.10, M_2 stores only the keys in its key-path: $k_{\langle 3,1\rangle}$, $k_{\langle 2,0\rangle}$, $k_{\langle 1,0\rangle}$, and $k_{\langle 0,0\rangle}$, and the following bkeys: $bk_{\langle 3,1\rangle}$, $bk_{\langle 3,0\rangle}$, $bk_{\langle 2,1\rangle}$, and $bk_{\langle 1,1\rangle}$.

With the μTGDH tree we introduce the following six optimized protocols.

5.3.1. Setup protocol

One of the improvements of setup protocol in μTGDH is that the sponsor of each sub-tree broadcasts only the respective group key, instead of the whole tree with all blinded keys. It is because that after merge the roots of both sub-trees become the children of the new root. Only the blinded key of the old root of one sub-group is in the *co-path* of all members in another sub-group. With this optimization the cumulative message size can be significantly reduced. The criteria to sort the members and to construct the tree is the same as in TGDH.

Assume that n members wish to form a group, h is $\lceil \log_2^n \rceil$. The detailed process is illustrated in Protocol 5.8.

Protocol 5.8 (μTGDH setup protocol)

1. *Round 1: Every member $M_i, i \in [1, n]$ broadcasts its setup-request with its blinded key*
   M_i $\xrightarrow{\qquad M_i's\ bkey \qquad}$ M_*

2. *Round 2: Every member sorts the members and forms the tree as described in Section 4.4.1. To save the memory cost, it removes then all bkeys except those of itself and of its sibling.*
   Every member $\langle l, v\rangle$ with odd v, identifies the update-list consisting of nodes whose bkeys are to be computed and update by it: $\{\langle l - i, v/2^i\rangle | 1 < i < l, \lfloor v/2^{i-1} \rfloor$ is odd and either $\lfloor v/2^{i-2} \rfloor$ is odd

or $\langle l, v\rangle$ is leaf node)}. Every member computes then the key of its parent, the member with odd v computes additionally the bkey of its parent, broadcasts it, and removes its parent from the update-list.

3. *Round 3 to h: Every member picks up the bkeys in its co-path and computes the keys in its key-path as far as possible. If there are members in the update-list, the member tries to compute the bkeys of those members. The newly computed bkeys are broadcasted and the respective nodes are removed from the upate-list.*

4. *Now every member has all bkeys in its co-path and computes the group key.*

Looking at the example in Figure 4.11. After the tree structure is constructed, the members remove the redundant bkeys. As the result, M_1 and M_2 store the bkeys $bk_{\langle 3,0\rangle}$ and $bk_{\langle 3,1\rangle}$, M_3 and M_4 store the bkeys $bk_{\langle 3,2\rangle}$ and $bk_{\langle 3,3\rangle}$, and M_5 and M_6 store the bkeys $bk_{\langle 2,2\rangle}$ and $bk_{\langle 2,3\rangle}$. Since M_1, M_3, M_5 have even v, they do not need to compute and broadcast any bkeys. The update-list of M_2 is $\{\langle 2, 0\rangle\}$, of M_4 is $\{\langle 2, 1\rangle, \langle 1, 0\rangle\}$, and of M_6 is $\{\langle 1, 1\rangle\}$.

Every member computes then the key of its parent. M_2, M_4, and M_6 compute additionally the bkeys of their parents respectively. The newly computed bkeys $bk_{\langle 2,0\rangle}$, $bk_{\langle 2,1\rangle}$, and $bk_{\langle 1,1\rangle}$ are broadcasted by M_2, M_4, and M_6 respectively, and the corresponding nodes are removed from the update-list. Now only M_4 have nodes in its update-list.

In the third round, M_1 and M_2 pick up $bk_{\langle 2,1\rangle}$ and $bk_{\langle 1,1\rangle}$, M_3 and M_4 pick up $bk_{\langle 2,1\rangle}$ and $bk_{\langle 1,1\rangle}$. M_1, M_2, M_3, and M_4 compute then the key of $\langle 1, 0\rangle$. Since $\langle 1, 0\rangle$ is in M_4's update-list, it computes additionally $bk_{\langle 1,0\rangle}$, broadcasts it and removes $\langle 1, 0\rangle$ from the update-list. Since $bk_{\langle 1,1\rangle}$ is broadcasted in the last round, they can compute the group key.

Equipped with the message from M_4 in the third round, M_5 and M_6 pick up $bk_{\langle 1,0\rangle}$ and can compute the group key.

The complexity characteristics of setup protocol are summarized in Table 5.7.

Rounds	h
Bcasts	$h^2 - h + n$
Muls/$\langle l, v\rangle$	$l + 1 + i$, where i is the least non-negative integer so that $\lfloor \frac{v}{2^i} \rfloor$ is even. If $\langle l, v\rangle$ is the shallowest rightmost leaf node in the finally formed tree, it is $l + i$.
Bcast size	$h^2 - h + n$
Serial Muls	$2h - 1$
Remark: $h = \lceil \log_2 n \rceil$	

Table 5.7.: Complexity of μTGDH setup protocol

5.3.2. Join protocol

The criteria to update the tree and to choose the sponsor is the same as in TGDH. The join protocol is improved by broadcasting the updated tree with only the blinded keys in the sponsor's key-path and in the new member's co-path, instead of all blinded keys. The advantage is to save the communication cost. Surely only the new member need the updated key structure, and the tree can be unicasted to the new member. However, one more message is needed. Hence we encapsule both into one message. It can be further optimized that all other members in initial tree except the sponsor do not need to store the bkey of the new member. The detailed process is illustrated in Protocol 5.9.

Protocol 5.9 (μTGDH join protocol)

1. *Round 1: The new member broadcasts its join-request with its blinded key*
   M_{n+1} $\xrightarrow{bkey\ of\ new\ member}$ M_*
2. *Round 2: Every member*
   - *updates the key tree,*
   - *removes all keys and bkeys in the sponsor's key-path.*

   The sponsor M_s additionally
   - *updates its share,*
   - *computes all keys and bkeys in its keypath,*
   - *broadcasts the updated tree $\widehat{T}_s$ with all bkeys in its key-path and in the new member's co-path.*

   M_s $\xrightarrow{\widehat{T}_s(KpBK_s^* \cup CoBK_{n+1})}$ M_*
3. *Every member M_i picks up bkeys in its co-path from the message, computes then the keys in its key-path.*

Looking at the example in Figure 4.12, the sponsor M_3 broadcasts the updated tree with only the bkeys $bk_{\langle 2,2\rangle}$, $bk_{\langle 1,1\rangle}$, and $bk_{\langle 1,0\rangle}$. M_1 and M_2 pick up then $bk_{\langle 1,1\rangle}$, the new member M_4 reconstructs first the updated tree and picks up the bkeys $bk_{\langle 2,2\rangle}$ and $bk_{\langle 1,0\rangle}$. Now all members can compute the group key. After finishing the join protocol, the bkeys stored by M_1 and M_2 are $bk_{\langle 2,0\rangle}$, $bk_{\langle 2,1\rangle}$, and $bk_{\langle 1,1\rangle}$, by M_3 and M_4 are $bk_{\langle 2,2\rangle}$, $bk_{\langle 2,3\rangle}$, and $bk_{\langle 1,0\rangle}$.

The complexity characteristics of the join protocol are summarized in Table 5.8.

Rounds	2
Ucasts	0
Bcasts	2
Muls/$\langle l, v\rangle$	$2l_s - 1$ for sponsor, $l_s + 1 - i$, $\exists i \in [0, l_s]$ so that $M_j \in T_{\langle l_s - i, v_s/2^i\rangle} \backslash T_{\langle l_s - i + 1, v_s/2^{i-1}\rangle}$
Ucast size	0
Bcast size	$l_s + 1$
Serial Muls	$3l_s - 2$
Remark: $\langle l_s, v_s\rangle$: sponsor in updated tree	

Table 5.8.: Complexity of μTGDH join protocol

5.3.3. Leave protocol

The criteria to update the tree and to choose the sponsor is the same as in TGDH. In leave protocol of TGDH, the updated tree together with all bkeys are broadcasted by the sponsor. However, since all remaining members know the old key and they can form the new tree, no tree are needed to be sent. Furthermore, only the bkeys in the sponsor's key-path are needed. As the result, after the update of the tree and the computation of key-pairs, the sponsor broadcasts only the bkeys in its key-path. The detailed process is illustrated in Protocol 5.10.

Protocol 5.10 (μTGDH leave protocol)

1. *Round 1: Each member*

- *removes the leaving member node and the relevant parent node,*
- *removes all keys and bkeys in the sponsor's key-path.*

The Sponsor M_s additionally,

- *updates its share,*
- *computes then all keys and bkeys in its key-path,*
- *broadcasts all blinded keys in its key-path.*

M_s $\xrightarrow{\qquad KpBK_s^* \qquad}$ M_*

2. *Every member M_i picks up $CoBK_i^* \cap KpBK_s^*$ (in fact only one blinded key), and computes then the keys in its key-path.*

Looking at the example in Figure 4.13, the sponsor M_3 needs only to broadcast $bk_{\langle 1,1\rangle}$. Equipped with $bk_{\langle 1,1\rangle}$ M_1 and M_2 can compute the group key.

The complexity characteristics of leave protocol are summarized in Table 5.9.

Rounds	1
Ucasts	0
Bcasts	1
Muls/$\langle l, v\rangle$	$2l_s - 1$ for sponsor, $l_s + 1 - i$ if $\exists i \in [0, l_s]$ so that $M_j \in T_{\langle l_s - i, v_s/2^i\rangle} \backslash T_{\langle l_s - i + 1, v_s/2^{i-1}\rangle}$
Ucast size	0
Bcast size	l_s
Serial Muls	$3l_s - 2$
Remark: $\langle l_s, v_s\rangle$: sponsor in updated tree	

Table 5.9.: Complexity of μTGDH leave protocol

5.3.4. Merge protocol

The criteria to update the tree and to choose the sponsors is the same as described in Section 4.4.4. In the original merge protocol, the sponsor broadcasts always the updated tree with all blinded keys, which consumes too much bandwidth. It is improved in μTGDH by sending only the newly computed bkeys, and the bkeys of the sponsors are not broadcasted. Therefor the sponsors' bkeys are not removed from the tree, unlike in TGDH.

Another improvement is that each sponsor identifies its update-list. The algorithm to find such nodes is described in Algorithm 5.1. The example can be found in Section 5.3.5. With the application of this algorithm some redundant computation and messages can be reduced.

Algorithm 5.1 (Identifies update-list in μTGDH)

Input: Sponsor M_s ($\langle l_s, v_s\rangle$), other sponsors $otherSponsors$
Output: Update-list of M_s.

```
List identifyUpdateList(Sponsor M_s, Collection otherSponsors){
    List updateList;
    IF(in protocol partition)
        Add Ms to updateList
    END IF
    l = l_s - 1
    v = v_s / 2
```

```
    WHILE(l > 0)
       IF(any members in otherSponsors in T_<l+1,2v+1>)
           IF(Ms is the rightmost sponsor in T_<l+1,2v+1>)
              add <l, v> to updateList
           ELSE
              return updateList
           END IF
       ELSE
           return updateList
       END IF
       l = l - 1
       v = v / 2
    END WHILE
    return updateList
}
```

Assume that m groups wish to merge to a common group. The detailed process of merge is illustrated in Protocol 5.11.

Protocol 5.11 (μTGDH merge protocol)

1. *Round 1: Each sponsor M_{s_i} of tree T^i for $1 \leq i \leq m$:*
   - *updates its share,*
   - *computes all keys and bkeys in its key-path (including $bk_{\langle 0,0 \rangle}$),*
   - *broadcasts updated tree $\widehat{T^i_{s_i}}$ including all bkeys in its key-path.*

   M_{s_i} $\xrightarrow{\widehat{T^i_{s_i}}(KpBK^*_{s_i})}$ M_*

2. *Round 2: Each member in tree T^i removes the keys and bkeys in M_{s_i}'s key-path, and picks up the bkeys in its co-path from the message broadcasted by M_{s_i}.*
   Each member updates the tree and identifies the sponsors. Then all keys and bkeys, from the parents of all sponsors up to the root, are removed.
   Each sponsor identifies its update-list. The sponsor with not empty update-list computes the keys in its key-path as far as possible and broadcasts the newly computed bkeys, then removes the corresponding nodes from the update-list.

   M_{s_t} $\xrightarrow{newly\ computed\ bkeys}$ M_*

3. *Round 3 to q ($q \leq \lceil log_2^m \rceil + 1$, after this round the update-list of the rightmost sponsor becomes empty.):*
   Each member picks up the bkeys in its co-path from the update messages, and computes the missing keys in its key-path as far as possible. The member with not empty update-list try to compute the bkeys of nodes in its update-list. The newly computed bkeys are broadcasted and their corresponding nodes are removed from the update-list.

   M_{s_t} $\xrightarrow{newly\ computed\ bkeys}$ M_*

4. *Each member picks up the bkeys in its co-path from the message broadcasted by the rightmost sponsor and computes then all missing keys up to the group key in its key-path.*

The complexity characteristics of merge protocol are summarized in Table 5.10.

	$m = 2$	$m \geq 3$
Rounds	2	$\lceil \log_2 m \rceil$+1
Bcasts	3	$2m$
Muls/$\langle l, v \rangle$	The same as in Table 4.19 except that $\mu = 2l'_{s_k}$ if $\langle l, v \rangle$ is sponsor $\langle l'_{s_k}, v'_{s_k} \rangle$	
Bcast size	$h + \alpha + \widehat{h} + 2$	$h + \alpha + m\widehat{h}$
Serial Exps	3·max(l'_{s_1}, l'_{s_2})+3$\widehat{h} - 4$	3·max($l'_{s_1}, \cdots, l'_{s_m}$)+(3$\widehat{h} - 4$)
Remark: α: sum of the height of all groups except T^1, $\langle l'_{s_k}, v'_{s_k} \rangle$ with $1 \leq k \leq m$: sponsor of tree T^k before merge, $\langle l_{s_k}, v_{s_k} \rangle$ with $1 \leq k \leq t$: sponsor of operation merge.		

Table 5.10.: Complexity of μTGDH merge protocol (worst case)

5.3.5. Partition protocol

The criteria to update the tree and to choose the sponsor is the same as described in Section 4.4.5. Similar to the merge protocol of μTGDH, the following improvements is applied: Each sponsor identifies first its update-list using Algorithm 5.1. With this improvement each new bkey is only computed and broadcasted one time, while in TGDH the same bkey can be done by multiple sponsors.

Assume that p members leave the group, the height of the updated tree is $\widehat{h}$. The detailed process of partition protocol is illustrated in Protocol 5.12.

Protocol 5.12 (μTGDH partition protocol)

1. *Round 1: Each member*
   - *updates key tree by deleting all leaving member nodes and their parent nodes,*
   - *removes all keys and bkeys from the parent nodes of sponsors nodes up to the root, and the bkey of the rightmost sponsor.*

   Each sponsor identifies then the update-list using Algorithm 5.1.
   Each sponsor computes the keys in its key-path as far as possible and broadcasts the newly computed bkeys, then removes the corresponding nodes from the update-list.
   M_{s_t} $\xrightarrow{\quad the\ sponsor's\ bkey\ and\ newly\ computed\ bkeys \quad}$ M_*

2. *Round 2 to q ($q \leq min(\lceil \log_2 p \rceil + 1, \widehat{h})$, after this round all update-lists become empty.):*
   Each member picks up the bkeys in its co-path from the update messages, and computes the missing keys in its key-path as far as possible. The member with no empty update-list tries to compute the bkeys of the nodes in the list. The newly computed bkeys are broadcasted and their corresponding nodes are removed from the update-list.
   M_{s_t} $\xrightarrow{\quad newly\ computed\ bkeys \quad}$ M_*

3. *Each member picks up the bkeys in its co-path from the message broadcasted by the rightmost sponsor and computes then all missing keys up to the group key in its key-path.*

Looking at the example in Figure 4.15. From the view of members in $\widehat{T_4}$, the sponsors are M_2, M_4, and M_6. The update-list of M_2 consists of the node of M_2. The node of M_4 and $\langle 1, 0 \rangle$ are in M_4's update-list. The nodes in M_7's update-list are $\langle 2, 3 \rangle$ and $\langle 1, 1 \rangle$.

In the first round, since M_2 and M_4 do not know the bkeys of each other, they broadcast the bkeys. M_7 updates its share and computes the key and bkey of $\langle 1, 1 \rangle$, then the bkeys $bk_{\langle 2,3 \rangle}$ and $bk_{\langle 1,1 \rangle}$ are broadcasted. After this round, only M_4's update-list is not empty. Hence it must compute and broadcast $bk_{\langle 1,0 \rangle}$. Now all members have all bkeys in their co-paths respectively, and can compute all keys in their key-paths.

The complexity characteristics of partition protocol (worst case) are summarized in Table 5.11.

Rounds	$min(\lceil \log_2 p \rceil + 1, \widehat{h})$
Ucasts	0
Bcasts	2min(2p,$\lceil \frac{n}{2} \rceil$)
Muls/$\langle l, v \rangle$	The same as in Table 4.20 except that the rightmost sponsor needs one multiplication less.
Ucast size	0
Bcast size	$(\widehat{h}+1)$·min(2p,$\lceil \frac{n}{2} \rceil$)
Serial Muls	$3\widehat{h}-2$
Remark: p: number of leaving members	

Table 5.11.: Complexity of μTGDH partition protocol (worst case)

5.3.6. Refresh protocol

We use the policy as described in 5.2.6. The rightmost member refreshes its share and computes all keys and bkeys in its key-path, and then broadcasts all bkeys in its key-path. Each other member picks up the bkeys in its co-path from the broadcast message and computes then all missing keys in its key-path.

The detailed process of refresh protocol is illustrated in Protocol 5.13.

Protocol 5.13 (μTGDH refresh protocol)

1. *Round 1: The refresher M_r*
   - *updates its key and bkey,*
   - *computes newly all keys and bkeys in its key-path,*
   - *broadcasts all bkeys in its key-path.*

   M_r $\xrightarrow{\qquad KpBK_r^* \qquad}$ M_*

2. *Each member M_i except the refresher:*
   - *removes all keys and bkeys from the refresher M_r to the root,*
   - *picks up $CoBK_i^* \cap KpBK_r^*$ (in fact only one bkey),*
   - *computes all missing keys in its key-path.*

The complexity characteristics are summarized in Table 5.12.

Rounds	1
Ucasts	0
Bcasts	1
Muls/$\langle l, v \rangle$	$2l_r - 1$ for refresher, $l_r + 1 - i$, $\exists i \in [0, l_r]$ so that $M_j \in T_{\langle l_r - i, v_r/2^i \rangle} \backslash T_{\langle l_r - i + 1, v_r/2^{i-1} \rangle}$
Ucast size	0
Bcast size	l_r
Serial Muls	$3l_r - 2$
Remark: $\langle l_r, v_r \rangle$: refresher	

Table 5.12.: Complexity of μTGDH refresh protocol

5.4. TFAN Protocol Suite

In this section we propose a Tree-based group key agreement Framework for Ad-hoc Networks (TFAN) based on the protocol suites μSTR and μTGDH. The structure of TFAN depends on two parameters: the maximal height (m) and the art (art) of main-sub-tree(ms-tree)s (defined below). The parameter art determines the ms-tree's art: S for an μSTR tree or T for an μTGDH tree. Besides the symbols in Figures 4.1 and 4.2 the following symbols are used for TFAN.

h	height of TFAN tree
m	maximal allowed height of ms-trees
MN_i	i^{th} main-node, $0 \leq i \leq h$
MT_i	i^{th} ms-tree, a subtree rooted at the right child of MN_i
$MT^u_{\langle l,v\rangle}$	a subtree rooted at node $\langle l, v\rangle$ in MT_u, used only if $art = T$
SN_i	i^{th} sub-root, the root of MT_i
k_i	key associated with main-node MN_i
bk_i	blinded key associated with main-node MN_i
mk_i	key associated with SN_i
mbk_i	blinded key associated with SN_i
MBK^*_i	set of M_i's blinded keys of sub-roots
$k_{\langle i,\langle l,v\rangle\rangle}$	key of the node $\langle l, v\rangle$ in ms-tree MT_i of art T.
$bk_{\langle i,\langle l,v\rangle\rangle}$	blinded key of the node $\langle l, v\rangle$ in ms-tree MT_i of art T.
$r_{\langle i,l\rangle}$	key of the internal node IN_l in ms-tree MT_i of art S.
$br_{\langle i,l\rangle}$	blinded key of the internal node IN_l in ms-tree MT_i of art S.
$k_{\langle i,l\rangle}$	session random of the leaf node LN_l in ms-tree MT_i of art S.
$bk_{\langle i,l\rangle}$	blinded session random of the leaf node LN_l in ms-tree MT_i of art S.

Figures 5.2 and 5.3 give examples of TFAN trees with μSTR and μTGDH trees as the ms-trees respectively. Unlike in μSTR protocol suite, the internal and leaf nodes of μSTR ms-trees are numbered in inverse direction starting with 0. The root is the topmost node. The nodes $MN_0, \cdots, MN_h$ are called *main-node*s. A main-node MN_i has always two children: the deeper main-node MN_{i+1} as the left child and the root of MT_i as the right child. The only exception is MN_h, it is also the root of MT_h. An ms-tree MT_i is an μTGDH or μSTR tree. It roots at the right child of the main-node MN_i. The root of MT_i is i^{th} sub-root, denoted SN_i. An μTGDH ms-tree has at most 2^m members, and an μSTR ms-tree $m + 1$ members.

The key of a main-node MN_i is denoted k_i, and the blinded key bk_i. Considering first a tree with μTGDH ms-trees, the key associated with node $\langle l, v\rangle$ in ms-tree MT_i is denoted $k_{\langle i,\langle l,v\rangle\rangle}$, and the corresponding blinded key is $bk_{\langle i,\langle l,v\rangle\rangle}$. Considering now a tree with μSTR ms-trees, the key and blinded key associated with internal node IN_i in ms-tree MT_l are denoted $k_{\langle i,l\rangle}$ and $bk_{\langle i,l\rangle}$, respectively. Similarly the session random and blinded session random associated with leaf node LN_l in ms-tree MT_i are denoted $r_{\langle i,l\rangle}$ and $br_{\langle i,l\rangle}$, respectively. For sake of clarity, we define the key and blinded key associated with SN_i to be mk_i and mbk_i respectively.

The path from member M_i to the root is called M_i's *key-path*. And the path from M_i to the root of the ms-tree where M_i locates is denoted M_i's *ms-key-path*. $K_G = k_0$ at the root is the group secret shared by all members. We note that K_G is never applied as a cryptographic key for the purposes of encryption, authentication or integrity. Instead, such special purpose sub-keys are derived from the K_G, such as using a one-way hash function $h(\cdot)$: $k = h(K_G)$.

A TFAN tree can be considered as an μSTR tree whose members are the whole ms-trees. Like in μSTR and μTGDH, to compute the group key the member M_i must know all blinded keys (and blinded session randoms) in its co-path, namely $CoBK^*_i$.

The trees in Figures 5.2 and 5.3. The former uses μSTR trees, and the latter uses μTGDH trees with maximal height $m = 2$ for the ms-trees. The roots in both trees are MN_0. The main-nodes are MN_0,

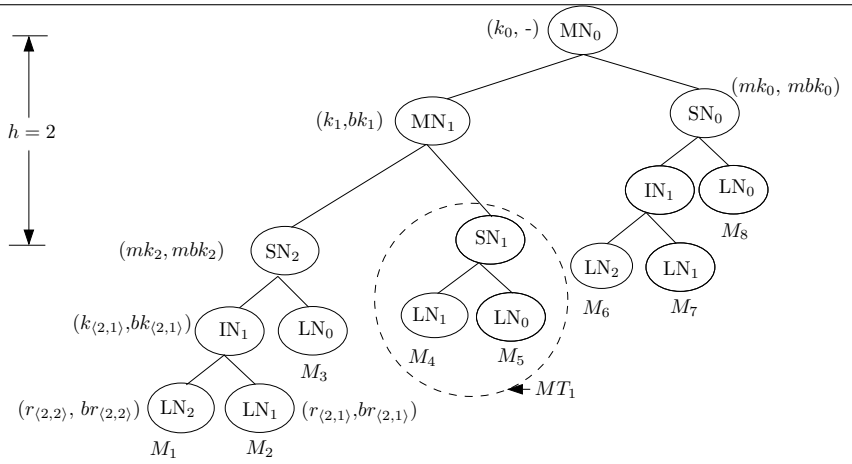


Figure 5.2.: A TFAN tree ($art = S$, $m = 2$)

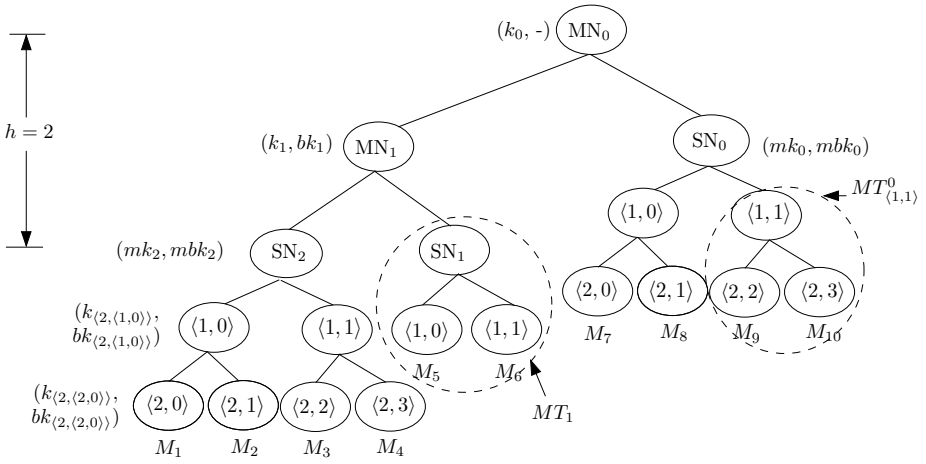


Figure 5.3.: A TFAN tree ($art = T$, $m = 2$)

MN_1, and MN_2. Hence the height is $h = 2$, and there are 3 ms-trees: MT_0, MT_1, and MT_2. The ms-tree MT_1 is illustrated in the figures. Since $m = 2$, at most $m + 1 = 3$ members are allowed in the ms-tree in Figure 5.2, and $2^m = 4$ members in Figure 5.3. Some keys (session randoms) and their blinded values are illustrated in the figures. The examples of some other symbols are listed below:

	Figure 5.2	Figure 5.3
KEY_5^*	$\{r_{\langle 1,0\rangle}, mk_1, k_1, k_0\}$	$\{k_{\langle 1,\langle 1,0\rangle\rangle}, mk_1, k_1, k_0\}$
$KpBK_5^*$	$\{br_{\langle 1,0\rangle}, mbk_1, bk_1\}$	$\{bk_{\langle 1,\langle 1,0\rangle\rangle}, mbk_1, bk_1\}$
$CoBK_5^*$	$\{br_{\langle 1,1\rangle}, bk_2, mbk_0\}$	$\{bk_{\langle 1,\langle 1,1\rangle\rangle}, bk_2, mbk_0\}$
MBK_5^*	$\{mbk_0, mbk_1, mbk_2\}$	$\{mbk_0, mbk_1, mbk_2\}$

Assume that there is a TFAN tree with μSTR ms-trees. Let M_x be the member associated with the leaf node LN_l in ms-tree MT_i. We consider first the general case: M_x is located not in the deepest ms-tree, and is not the greatest numbered leaf node in MT_i. The following blinded session randoms and blinded keys must be known to M_x: $br_{\langle i,j\rangle}$ for $0 \le j \le l$, $bk_{\langle i,l+1\rangle}$, bk_{i+1}, and mbk_j for $0 \le j \le i-1$. And the following keys and session randoms must be also stored by M_x: $r_{\langle i,l\rangle}$, $k_{\langle i,j\rangle}$ for $0 \le j \le l$, and k_j for $0 \le j \le i$. Hence the memory costs for M_x are $i + l + 3$ keys and public keys.

If M_x is the deepest leaf node in the deepest ms-tree, the memory costs for M_x are $h + l + 1$ keys and public keys. Otherwise if M_x is in the deepest ms-tree or the deepest leaf node in its ms-tree, the memory

costs for M_x are $i + l + 2$ keys and public keys.

Now we consider a TFAN tree with μTGDH trees as ms-trees. Again let M_x be the member associated with leaf node $\langle l, v\rangle$ in ms-tree MT_i. Assume that MT_i is not the deepest ms-tree. There are $i + l + 2$ keys in M_x's key-path and $i + l + 1$ blinded keys in M_x's co-path. Additionally M_x stores also its own blinded key. Hence the memory costs for M_x is $i + l + 2$ keys and blinded keys. If M_x is in the deepest ms-tree, the memory costs are $i + l + 1$ keys and blinded keys.

The group key in both examples are computed by M_5 as follows [3]:

- In Figure 5.2:

$$\begin{aligned} K_G = k_0 &= f(k_1 \cdot mbk_0) \\ &= f(f(mk_1 \cdot mbk_2) \cdot mbk_0) \\ &= f(f(f(r_{\langle 1,0\rangle} \cdot bk_{\langle 1,1\rangle}) \cdot mbk_2) \cdot mbk_0) \end{aligned}$$

- In Figure 5.3:

$$\begin{aligned} K_G = k_0 &= f(k_1 \cdot mbk_0) \\ &= f(f(mk_1 \cdot mbk_2) \cdot mbk_0) \\ &= f(f(f(k_{\langle 1,\langle 1,0\rangle\rangle} \cdot bk_{\langle 1,\langle 1,1\rangle\rangle}) \cdot mbk_2) \cdot mbk_0) \end{aligned}$$

The rest of this section describes the six protocols of the framework in details.

5.4.1. Setup

Assume that n members wish to form a group. At first each member broadcasts its blinded key (or blinded session random) with some other information required by the protocol. The nodes can be sorted as follows: $M_1, \cdots, M_n$, according to some criteria, e.g. the nodes with more storage space and more powerful computation ability are sorted before the other nodes. The nodes are then divided into $\lceil \frac{n}{b} \rceil$ blocks with block length $b = m + 1$ if $art = S$, and $b = 2^m$ if $art = TGDH$. All blocks except the last block have each b members, and the last block has the remaining members. The members in a block form a ms-tree. Hence the height of the TFAN tree is $h = \lceil \frac{n}{b} \rceil - 1$.

Remark 1 The members in the same ms-tree perform then the setup protocol of μSTR if $art = S$ or of μTGDH if $art = T$ with the following modification: in μSTR ms-tree the leftmost member computes and broadcasts the bkey of the sub-root, while in μTGDH ms-tree it is done by the rightmost ms-tree.

After the members in MT_h and MT_{h-1} receive bkeys of all sub-roots, they compute all absent keys in their key-paths. The rightmost leaf node in the deepest ms-tree is chosen as the sponsor. It additionally computes and broadcasts bkeys $\{bk_i | i \in [1, h-1]\}$. Each member in ms-tree MT_u with $u \in [0, h-2]$ picks up the bkey bk_{u+1} from the broadcast message. Now each member has all bkeys in its co-path so that it can compute all missing keys up to the group key. The detailed process is illustrated in Protocol 5.14.

Protocol 5.14 (TFAN setup)

1. *Round 1: Each member $M_i, i \in \{1, \cdots, n\}$ broadcasts a setup request with its own bkey:*
   M_i $\xrightarrow{\qquad M_i's\ bkey \qquad}$ M_*
2. *Round 2 to q($q = 2$ if $art = S$, and $q = m + 1$ if $art = T$):*
   In round 2 the members are sorted and divided to several blocks as described in Remark 1. Then the members in the same block execute the setup protocol (μSTR setup protocol for $artS$ and μTGDH setup protocol for $artT$) with the modification described in Remark 1.

[3] Note that all blinded keys and session randoms are elliptic curve points.

3. *Round $q+1$: The sponsor, the rightmost member in the deepest ms-tree, computes then the missing keys and bkeys of the main nodes in its key-path. It broadcasts then the changed bkeys.* [4]

4. *Each member picks up the bkeys in its co-path from the broadcast message, and computes all missing keys in its key-paths.*

We consider first the example with $art = S$ and $m = 2$ in Figure 5.4. All members broadcast first setup requests with their bkeys. After the collection of all setup requests, the members are sorted according to some criteria. Assume that there are 11 members and they are sorted as follows: M_1, M_2, $\cdots$, M_{11}. Since at most 3 members are allowed in an ms-tree, all members are divided as follows: MT_3 (M_1, M_2, M_3), MT_2 (M_4, M_5, M_6), MT_1 (M_7, M_8, M_9), MT_0 (M_{10}, M_{11}). Now the members within an ms-tree execute the μSTR setup protocol with the modification described in Remark 1.

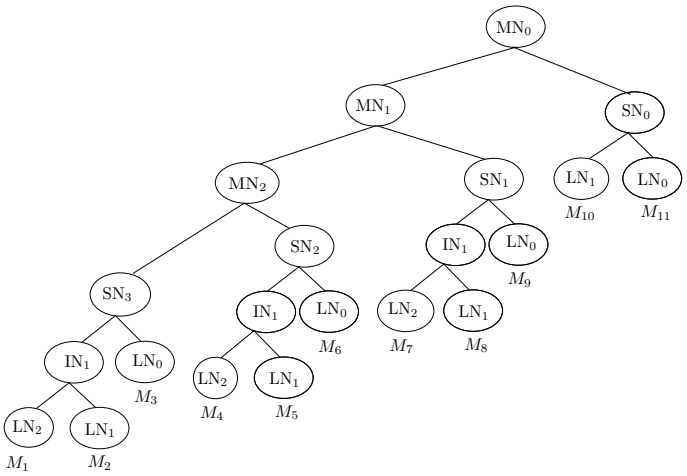


Figure 5.4.: An example of TFAN setup ($art = S$, $m = 2$)

The sponsor for the next round is M_3. Since all members in MT_3 and MT_2 know all bkeys in their co-paths respectively, all members in both ms-trees compute all keys in their key-paths respectively. The sponsor M_3 additionally computes and broadcasts bk_3, bk_2, and bk_1. Equipped with this broadcast message, the members in MT_1 compute k_1 and k_0, and the members in MT_0 compute k_0.

Another example with $art = T$ and $m = 2$ is given in Figure 5.5. After broadcasting blinded keys in the first round, all members in the same ms-tree execute the setup protocol of μTGDH. The rightmost sponsors in each ms-tree: M_4, M_8, M_{12} and M_{14}, compute and broadcast mbk_3, mbk_2, mbk_1 and mbk_0, respectively.

Upon receiving the broadcasted messages, all members in MT_1 and MT_0 computes the keys in their respective ms-key-paths. All members in MT_3 and MT_2 except M_4 compute all keys in their key-paths respectively. M_4 is the sponsor for next round, it computes first (k_2, bk_2) and (k_1, bk_1), broadcasts then bk_2 and bk_1. Finally all members compute all missing keys in their respective key-paths.

The complexity characteristics except the number of multiplications per member are summarized in Table 5.13.

If $art = S$, let M_x be leaf node LN_i in MT_u for $0 \leq u \leq h$, and assume that $h \geq 2$. The numbers of multiplications per members are listed below:

1. If $i = m$: $2m + min(h, u + 1)$;
2. If $i = 0 \wedge u = h$: $2h - 1$;
3. Else: $i + 1 + min(h, u + 1)$.

[4] The members in the deepest two ms-trees know till this round all bkeys in their co-paths, so they can compute the keys of the main nodes in their key-paths in parallel.

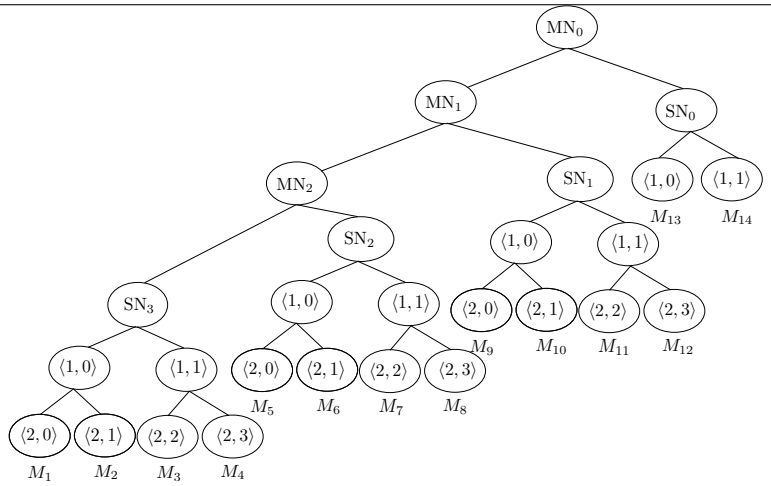


Figure 5.5.: An example of TFAN setup ($art = T$, $m = 2$)

	$art = S$	$art = T$
Rounds	3	$m + 2$
Bcasts	$n + h + 2$	$n + 1 + (m^2 - m + 1)(h + 1)$
Bcast size	$n + mh + m + h - 1$	$n + h - 1 + (m^2 + m + 1)(h + 1)$
Serial Muls	$2m + h - 1 +$ $max(m, 2h - 2)$	$2m + 3h - 3$

Table 5.13.: Complexity of TFAN setup protocol

If $art = T$ and $h \geq 2$, let M_x be leaf node $\langle l, v \rangle$ in MT_u for $0 \leq u \leq h$ ($l = m$ for $1 \leq u \leq h$). The numbers of multiplications per member are listed below:

1. If M_x is $\langle m, 2^m - 1 \rangle$ in MT_h: $2m + 2h - 1$;
2. Else: $l + 1 + i + min(h, u + 1)$, where i is the least non-negative integer so that $\lfloor \frac{v}{2^i} \rfloor$ is even.

5.4.2. Join

Assume that the group has n members $\{M_1, \cdots, M_n\}$, and the new member M_{n+1} wishes to join. M_{n+1} broadcasts a *join* request with its bkey. Each member in current group receives this request and determines the insertion position. It finds first the shallowest not fully filled ms-tree. Assume that this ms-tree is MT_i. The join event is then processed according to the join protocol of μSTR if $art = S$ or of μTGDH if $art = T$ with the extension that the sponsor computes all keys and bkeys in its key-path except k_0.

If all ms-trees are fully filled, a new ms-tree with the new member as its unique member and a new main-node are created. The new main-node becomes the new root of the TFAN tree with the old root as its left child and the root of the new ms-tree as its right child. The shallowest rightmost leaf node in MT_1 (in updated tree) is chosen as the sponsor M_s. The detailed process is illustrated in Protocol 5.15.

Protocol 5.15 (TFAN join)

1. *Round 1: The new member M_{n+1} broadcasts its own bkey.*
   M_{n+1} $\xrightarrow{\quad new\ member's\ bkey \quad}$ M_*
2. *Round 2: Every member*
   - *updates the tree by inserting the new member,*
   - *removes all keys and bkeys in the key-path of the sponsor.*

The sponsor M_s additionally,

- *updates its key and bkey,*
- *computes all keys and bkeys in its key-path except k_0,*
- *broadcasts all bkeys in its key-path.*

M_s $\xrightarrow{\qquad KpBK_s^* \qquad}$ M_*

3. *The sponsor computes k_0, each other member M_i picks bkeys in its co-path and computes the keys in $KEY_i^* \cap KEY_s^*$.*

We consider first the example with μSTR ms-trees in Figure 5.6. Since MT_1 is not fully filled and all ms-trees above it (here only MT_0) are fully filled, M_9 joins in MT_1. M_5 is the sponsor. All members (except M_5 and M_9) remove all keys and bkeys in M_5's key-path. Equipped with the broadcast message from M_5,

- M_1, M_2, and M_3 compute k_1 and k_0,
- M_4 computes $k_{\langle 1,1\rangle}, mk_1, k_1$ and k_0,
- M_9 computes mk_1, k_1 and k_0,
- M_6, M_7, and M_8 compute k_0.

Another example with μTGDH ms-trees is given in Figure 5.7. The new member M_{11} joins into MT_1 and M_6 is the sponsor. The process is similar as the example in Figure 5.6.

The complexity characteristics except the multiplications per member are summarized in Table 5.14.

	$art = S$	$art = T$
Rounds	2	2
Bcasts	2	2
Bcast size	$4 + min(u_s, h-1)$	$l_s + 1 + min(u_s, h-1)$
Serial Muls	$7 + 3min(u_s, h-1)$	$3l_s + 3min(u_s, h-1) + 1$
Remark: MT_{u_s}: the ms-tree containing the sponsor, $\langle l_s, v_s\rangle$: the sponsor's index in MT_{u_s}		

Table 5.14.: Complexity of TFAN join protocol

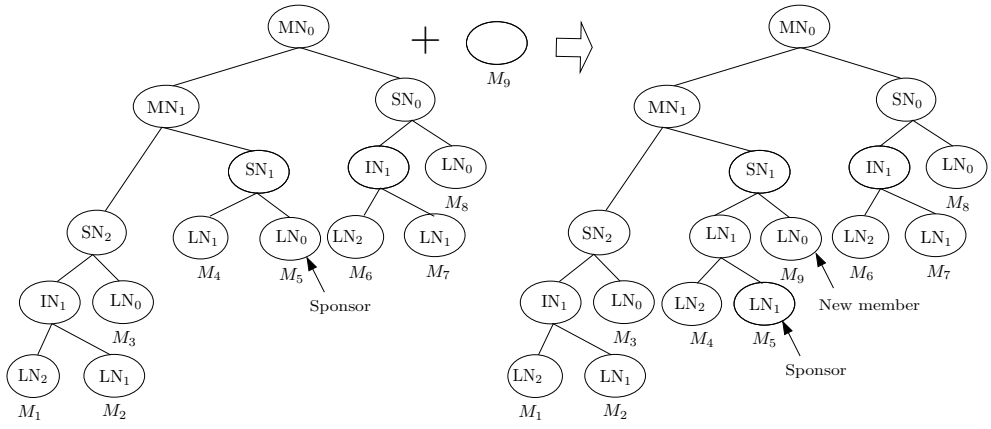


Figure 5.6.: A TFAN join ($art = S, m = 2$)

If $art = S$, let M_x be leaf node LN_i in MT_u for $0 \leq u \leq h$. The numbers of multiplications per member are listed below:

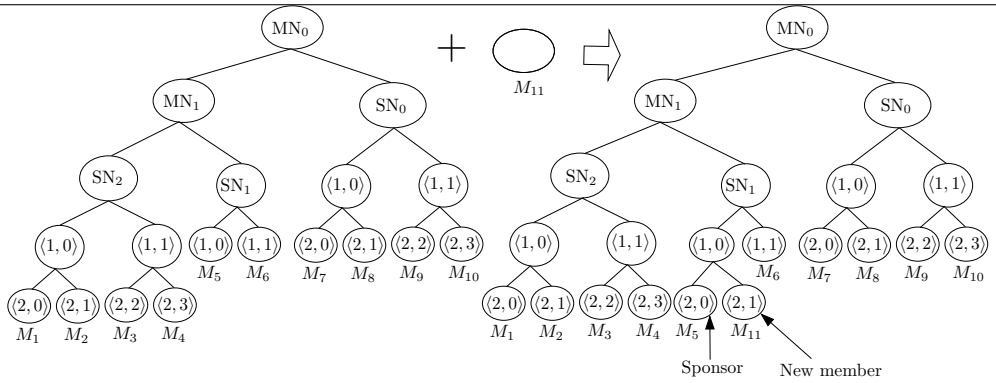


Figure 5.7.: A TFAN join ($art = T$, $m = 2$)

- Sponsor: $2min(u_s, h-1) + 3$;
- New member: $min(u_s, h-1) + 2$;
- Other members in MT_{u_s}: $min(u_s, h-1) + 3$;
- If $u > u_s$: $u_s + 1$;
- If $u < u_s$: $u + 1$.

If $art = T$, let M_x be leaf node $\langle l, v\rangle$ in MT_u for $0 \leq u \leq h$. The numbers of multiplications per member are listed below:

1. Sponsor: $2l_s + 1 + 2min(u_s, h-1)$;
2. Other members in MT_{u_s}: $l_s + 1 - i + min(u_s, h-1)$, $\exists i \in [0, l_s]$ so that $M_j \in MT^{u_s}_{\langle l_s - i, v_s/2^i\rangle} \setminus MT^{u_s}_{\langle l_s - i + 1, v_s/2^{i-1}\rangle}$;
3. If $u > u_s$: $u_s + 1$;
4. If $u < u_s$: $u + 1$.

5.4.3. Leave

Assume that we have a group with n members, and M_l in ms-tree MT_i leaves the group. If M_l is the only member in ms-tree MT_i, the shallowest rightmost leaf node of M_{i+1} if $i \geq 2$, or of M_{h-1} if $i = h$, is chosen as the sponsor. The leave of member in MT_i is processed according to the leave protocol of μSTR if $art = S$ or of μTGDH if $art = T$. The sponsor updates its share and computes all keys and bkeys in its key-path except k_0. The detailed process is illustrated in Protocol 5.16.

Protocol 5.16 (TFAN leave)

1. *Round 1: Each member*
   - *updates the tree by removing the leaving member and respective parent node,*
   - *removes all keys and bkeys in the key-path of the sponsor.*

   The sponsor M_s additionally,
   - *updates its key and bkey,*
   - *computes all keys and bkeys in its key-path except k_0,*
   - *broadcasts all bkeys in its key-path.*

   $M_s \xrightarrow{\quad KpBK_s^* \quad} M_*$

2. *The sponsor computes k_0, each other member M_i picks up bkeys in its co-path and computes the keys in $KEY_i^* \cap KEY_s^*$.*

We consider first the example with μSTR ms-trees in Figure 5.8. M_5 leaves the group. M_4 is the sponsor, since it is directly below M_5. M_4 first updates its new share and computes (mk_2, mbk_2), (k_2, bk_2), and (k_1, bk_1), and then broadcasts $br_{\langle 2,1\rangle}, mbk_2, bk_2$, and bk_1. Equipped with the broadcast message from M_4, M_6 computes mk_2, k_2, k_1 and k_0. All members in MT_u with $u \in \{0, 1, 3\}$ computes k_i for $0 \leq i \leq u$. If M_7, instead M_5, leaves the group, M_6 is chosen as the sponsor, and MT_1 and MN_1 are removed from the tree.

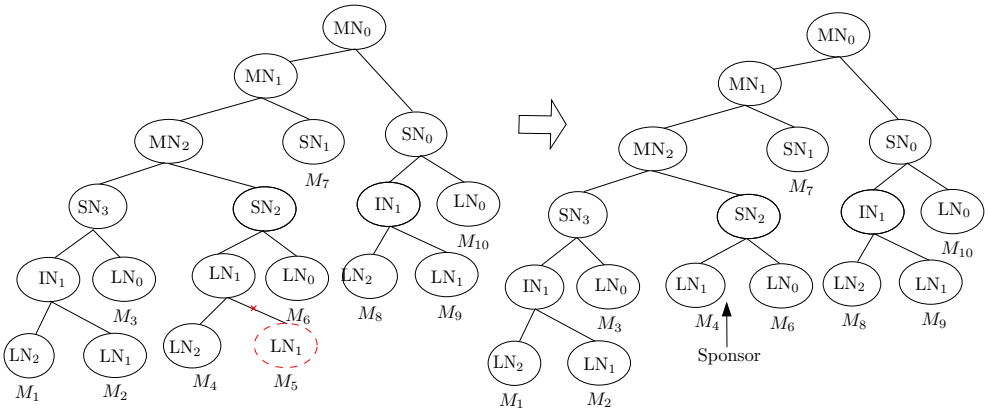


Figure 5.8.: A TFAN leave ($art = S$, $m = 2$)

Another example with μTGDH ms-trees is illustrated in Figure 5.9, where member M_8 leaves the group. According to the leave protocol of μTGDH, M_8 and its parent $\langle 1, 1\rangle$ are removed from the tree, and M_7 is chosen as the sponsor. M_7 first updates its share and then computes (mk_1, mbk_1) and (k_1, bk_1). It broadcasts then mbk_1 and bk_1. Equipped with the message from M_7, members in MT_2 (M_1, M_2, M_3 and M_4) compute then k_1 and k_0, and those in MT_0 (M_9, M_{10}, M_{11}) compute k_0. The members in M_7's ms-tree: M_5 and M_6, compute mk_0, k_1 and k_0.

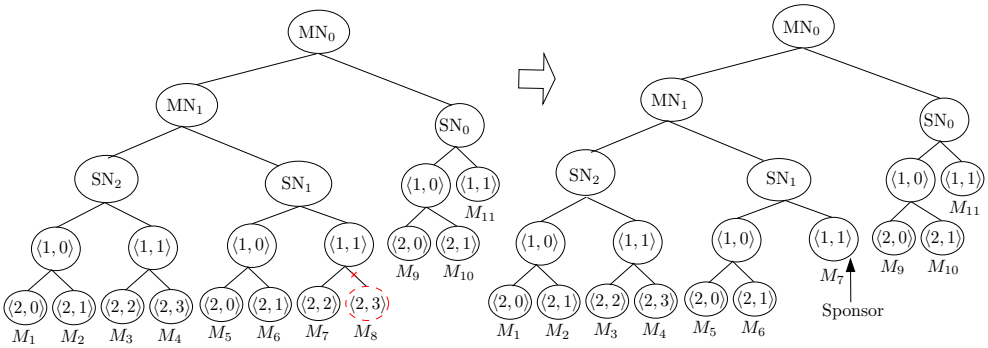


Figure 5.9.: A TFAN leave ($art = T$, $m = 2$)

The complexity characteristics except the number of multiplications per member are summarized in Table 5.14.

If $art = S$, let M_x be leaf node LN_i in MT_u for $0 \leq u \leq h$. The numbers of multiplications per member are listed below:

- Sponsor: $2s + 2min(u_s, h - 1) + 1$;
- Non-sponsor in MT_{u_s}: $min(u_s, h - 1) + s + 2$;

	$art = S$	$art = T$
Rounds	1	1
Bcasts	1	1
Bcast size	$s + min(u_s, h-1) + 3$	$l_s + min(u_s, h-1) + 2$
Serial Muls	$3s + 3min(u_s, h-1) + 4$	$3l_s + 3min(u_s, h-1) + 4$
Remark: MT_{u_s}: the ms-tree containing the sponsor, $LN_s(\langle l_s, v_s\rangle$: the sponsor		

Table 5.15.: Complexity of TFAN leave protocol

- If $u > u_s$: $u_s + 1$;
- Else: $u + 1$.

If $art = T$, let M_x be leaf node $\langle l, v\rangle$ in MT_u for $0 \leq u \leq h$. The numbers of multiplications per member are listed below:

- $2l_s + 2min(u_s, h-1) + 3$ for the sponsor,
- $l_s + 1 - i + min(u_s, h-1)$ if $\exists i \in [0, l_s]$ so that $M_j \in MT^{u_s}_{\langle l_s - i, v_s/2^i\rangle} \setminus MT^{u_s}_{\langle l_s - i + 1, v_s/2^{i-1}\rangle}$ for other members in MT_{u_s},
- $u_s + 1$ if $u > u_s$,
- $u + 1$ if $u < u_s$.

5.4.4. Merge

Assume that m' [5] groups wishes to merge into one group. In the setting of TFAN tree, only the members in the deepest ms-tree know bkeys of all sub-roots. Hence the sponsor to broadcast the tree is from the deepest ms-tree. The trees are sorted from the highest to the lowest tree: $T^1, \cdots, T^{m'}$. If two trees are of the same height, some other criteria must be applied, e.g. lexicographical comparison of the identifiers of the respective sponsors. The broadcast sponsor of tree T^i is denoted M_{s_i}.

A TFAN tree can be considered as an μSTR tree with the whole ms-trees as its leaf nodes. Hence the merge of TFAN can be processed according to the merge protocol of μSTR.

First each broadcast sponsor broadcasts its tree with bkeys of all sub-roots. After collection of all broadcast messages, each member first orders the merging trees, and then merges them. The shallowest rightmost leaf node in the deepest ms-tree T^1 becomes the sponsor M_s for the next round. The keys and bkeys in the sponsor's key-path are removed from the key tree. The sponsor additionally updates its share and computes all keys and bkeys in its key-path. The bkeys in key-path of M_s are then broadcasted. Equipped with this message, each member computes the keys in its key-path and get the group key. The detailed process is illustrated in Protocol 5.17.

Protocol 5.17 (TFAN merge)

1. *Round 1: Each broadcast sponsor M_{s_i} for $1 \leq i \leq m'$ broadcasts its tree with bkeys of all sub-roots.*
   $M_{s_i} \xrightarrow{T^i_{s_i}\{MBK^*_{s_i}\}} M_*$
2. *Round 2: Each member*
   - *updates its tree by merging all trees,*
   - *removes all keys and bkeys in the key-path of the new sponsor.*

   The sponsor M_s additionally
   - *updates its share,*

[5] Since m is used as the parameter of TFAN, m' denotes the number of merging groups instead.

- *computes all keys and bkeys in its key-path except k_0,*
- *broadcasts all bkeys in its key-path.*

M_s $\xrightarrow{\qquad KpBK_s^* \qquad}$ M_*

3. *The sponsor computes the group secret key k_0, each other member M_i picks the bkey in its co-path and computes the keys in $KEY_i^* \cap KEY_s^*$.*

Since merge protocol behaves similar in both cases ($art = S$ and $art = T$), we consider in the following only the example with $art = S$ in Figure 5.10. There are two groups T^1 and T^2 with $h = 3$ and $h = 2$, respectively. Hence T^2 is put on top of T^1. A new main-node MN_1 is created with the root of T^1 as the left child and the root of MT_1 of T^2 as the right child. M_8 updates its share and then computes all keys and bkeys in its key-path except k_0: (mk_2, mbk_2), (k_2, bk_2), and (k_1, bk_1). It then broadcasts mbk_2, bk_2 and bk_1. Equipped with the broadcast message, M_6 and M_7 compute mk_2, k_2, k_1, and k_0. All members in $MT_i, \forall i \in \{0, 1, 3\}$ compute then all k_j for $0 \leq j \leq i$.

The complexity characteristics except the multiplications per member are summarized in Table 5.16.

	$art = S$	$art = T$
Rounds	2	2
Bcasts	$m' + 1$	$m' + 1$
Bcast size	$h + 2\alpha + m' + 2$	$h + 2\alpha + m' + l_s$
Serial Muls	$3\alpha + 4$	$3l_s + \alpha + 1$
Remark: h: height of T^1, α: sum of the height of all groups except T^1, m': number of merging groups		

Table 5.16.: Complexity of TFAN merge protocol

If $art = S$, let M_x be leaf node LN_i in MT_u for $0 \leq u \leq h + \alpha$. The numbers of multiplications per member are listed below:

- Sponsor (in MT_α): $2\alpha + 3$;
- Non-sponsor in MT_α: $\alpha + 2$;
- If $u > u_s$: $\alpha + 1$;
- If $u < u_s$: $u + 1$.

If $art = T$, let M_x be leaf node $\langle l, v \rangle$ in MT_u for $0 \leq u \leq h + \alpha$. The numbers of multiplications per member are listed below:

- $2l_s + 2\alpha + 1$ for the sponsor (in MT_α),
- $\alpha + l_s + 2 - i$ if $\exists i \in [0, l_s]$ so that $M_j \in MT^\alpha_{\langle l_s - i, v_s/2^i \rangle} \setminus MT^\alpha_{\langle l_s - i + 1, v_s/2^{i-1} \rangle}$ for other members in MT_α,
- $\alpha + 1$ if $u > \alpha$,
- $u + 1$ if $u < \alpha$.

5.4.5. Partition

Once again, we have a group of n members, and p of them leave the group, e.g. due to a network failure. First we should identify the members to compute and broadcast the bkeys. Such sponsors who take part in the partitions or leaves within the ms-trees are called ms-sponsors. The member who computes and broadcasts the bkeys of the main-nodes is called sponsor. If all members in an ms-tree leave the group, the ms-tree is removed from the TFAN tree. No ms-sponsors are needed for such events. Otherwise the leaving

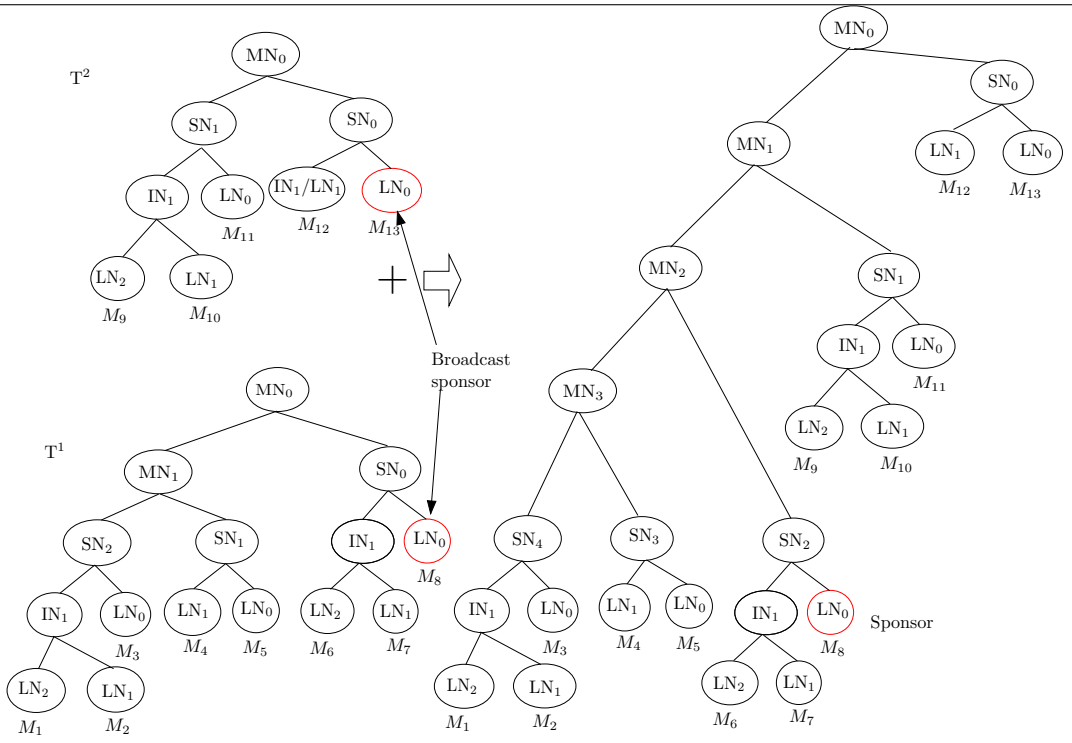


Figure 5.10.: A TFAN merge ($art = S$, $m = 2$)

of members in each ms-tree is processed using the partition protocol [6] of μSTR (if $art = S$) or μTGDH (if $art = T$), the corresponding sponsors in this phase are denoted by ms-sponsors.

The choice of the sponsor is some complex. If all members of the deepest modified ms-tree leave the group, the sponsor is the rightmost member in the ms-tree directly under the removed ms-tree, or in the greatest numbered remaining ms-tree if no ms-tree below it exists. In this case, after the collection of bkeys of all sub-roots, the sponsor updates its share. If there are remaining members in the deepest modified ms-tree then the sponsor is the rightmost ms-sponsor in the deepest modified ms-tree. It updates its share while executing the partition protocol within the ms-tree. Then it computes the missing keys and bkeys of the main-nodes and broadcasts the changed bkeys.

Now each member can pick up the missing bkeys in its co-path from the message and computes then all missing keys in its key-path. The detailed process is illustrated in Protocol 5.18.

Protocol 5.18 (TFAN partition)

1. *Round 1 to q ($q = 1$ for $art = S$, and $q \leq min(\lceil \log_2 p \rceil + 1, \widehat{h})$ for $art = T$): Each member updates the tree by removing the leaving members and respective parents.*
   In the remaining updated ms-trees members perform the leave or partition protocol of μSTR (if $art = S$) or μTGDH (if $art = T$) with the following modifications:
   - *each member removes the keys and bkeys from the sponsor up to the root, and from the parent of other ms-sponsors up to the root.*
   - *only the sponsor updates its share.*
   - *and the rightmost sponsor in each modified ms-tree MT_u computes all keys and bkeys in its key-path and mbk_u, then broadcasts the changed bkeys in MT_u.*
2. *Round $q + 1$: The sponsor M_s,*

[6] If only one member in the ms-tree leaves the group, using the leave protocol instead.

- *updates its share if it is in an unchanged ms-tree,*
- *computes the missing keys and bkeys of the main-nodes in its key-path except the group key,*
- *broadcasts all changed bkeys in its key-path.*

M_s $\xrightarrow{\quad changed\ bkeys\ in\ M'_s key\ path \quad}$ M_*

3. *The sponsor computes the group secret key k_0,*
   Each other member M_i computes the keys in $KEY_i^* \cap KEY_s^*$.

Looking at the example with μSTR ms-trees in Figure 5.11. M_6, M_7, M_8, and M_{10} leave the group. Since there are no members other than M_7 and M_8 in MT_1, MT_1 and MN_1 are removed. M_5 is the ms-sponsor for the leaving of M_6. Another ms-sponsor is M_9. Since M_5 is at the same also the sponsor, it updates its share and computes then mk_1 and mbk_1, and broadcasts $br_{\langle 1,0\rangle}$ and mbk_1. M_8 computes mk_0 and mbk_0, and broadcasts $br_{\langle 0,1\rangle}$ and mbk_0.

In the next round the sponsor M_5 computes k_1 and bk_1, then broadcasts bk_1. Now each member has all bkeys in its co-path and can compute the all missing keys in its key-path.

Another example with μTGDH ms-trees is given in Figure 5.12. Since all members of MT_1 (M_8, M_9) leave the group, it is removed together with its parent MN_1. According to the partition or leave protocol in μTGDH, M_5, M_{10} and M_{13} are the ms-sponsors for the leaving of M_6, M_{11} and M_{13} respectively. M_5 is also the sponsor, hence it updates first its share and computes then all keys and bkeys in its ms-key-path and broadcasts $bk_{\langle 1,\langle 1,0\rangle\rangle}$ and mbk_1. M_{10} and $M13$ broadcast first their bkeys respectively. M_{13} computes then mk_0 and mbk_0, broadcasts then mbk_0. The rest process is similar as in the example above.

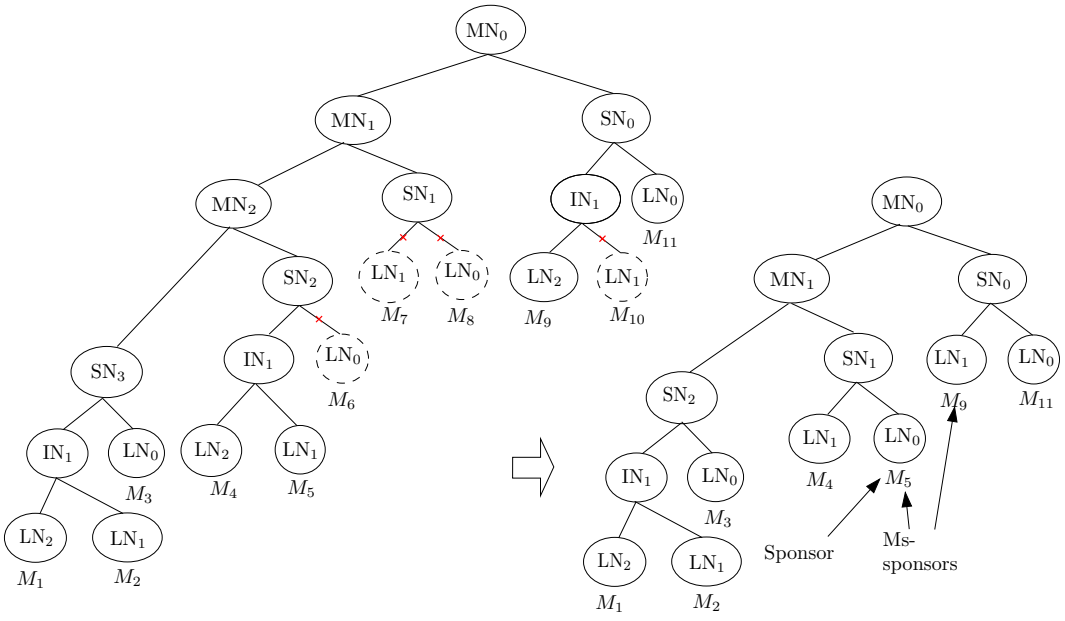


Figure 5.11.: A TFAN partition ($art = S$, $m = 2$)

Since the rounds and number of broadcast messages depend on the positions of leaving members, it is difficult to estimate the concrete values. However, they are bounded by the height of ms-trees and the number of leaving members. Since the cumulative message size in turn depends on the number of messages and the positions of leaving members, it is not evaluated in partition. Similarly the number of multiplications per member is also not evaluated. Onle the number of serial operations is provided. The other complexity characteristics are summarized in Table 5.17. Except the rounds and the number of broadcast messages in case $art = S$, all data are of the upper bound.

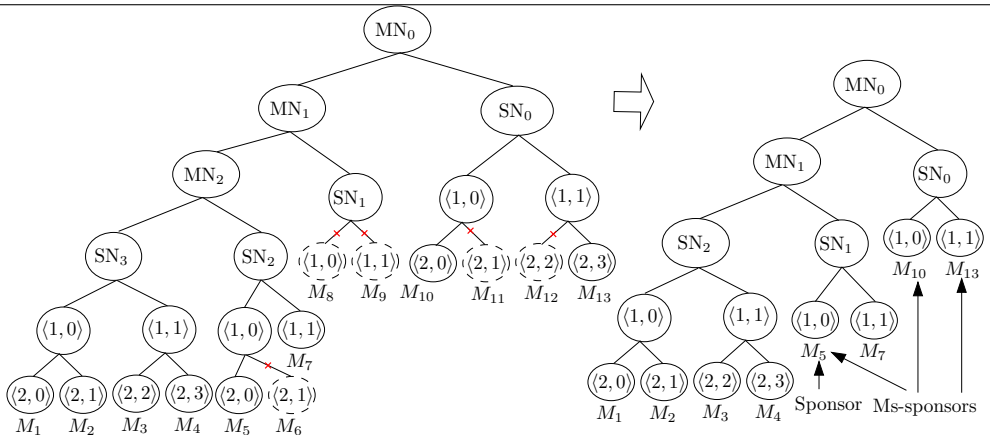


Figure 5.12.: A TFAN partition ($art = T$, $m = 2$)

	$art = S$	$art = T$
Rounds	2	$\frac{m}{2} + 1$
Bcasts	$t + 1$	$\frac{pm}{2} + 1$
Serial Muls	$3m + 3u_s - 1$ [7]	$3m + 3u_s - 2$ [8]
Remark: t: number of ms-sponsors, p: number of leaving members, MT_{u_s}: ms-tree containing the sponsor M_s, $LN_s(\langle l_s, v_s\rangle$: sponsor M_s		

Table 5.17.: Complexity of TFAN partition protocol

5.4.6. Refresh

Again we have a group of n members and of height h. The member M_r, denoted the *refresher*, wishes to update its share. The refresh-policy described in Section 5.2.6 can be also used in TFAN. The refresher changes first its key and bkey, then updates all keys and bkeys in its key-path except k_0, and broadcasts all bkeys in its key-path. It then computes the new group key k_0. After receiving the broadcast message, each other member removes all keys and bkeys in M_r's key-path, picks up the bkey in its co-path from the message. Equipped with all bkeys in the co-path, all members can compute all missing keys in the key-path. The detailed process is illustrated in Protocol 5.19.

Protocol 5.19 (TFAN refresh)

1. *Round 1: The refresher M_r,*
   - *updates its key and bkey,*
   - *computes all keys and bkeys in its key-path except the group key,*
   - *broadcasts all bkeys in its key-path.*

     $M_r \xrightarrow{KpBK_r^*} M_*$
   - *The refresher computes the group secret key k_0,each other member M_i picks up the bkey in its co-path from the message and computes the keys in $KEY_i^* \cap KEY_r^*$.*

The complexity characteristics except the number of multiplications per member are summarized in Table 5.18.

If $art = S$, let M_x be leaf node LN_i in MT_u for $0 \leq u \leq h$. The numbers of multiplications per member are listed below:

[7] This is for the case when the sponsor is one of the ms-sponsors, otherwise it is $3m + 3u_s + 2$.

[8] This is for the case when the sponsor is one of the ms-sponsors, otherwise it is $3m + 3u_s + 3l_s - 2$.

	$art = S$	$art = T$
Rounds	1	1
Bcasts	1	1
Bcast size	$r + min(u_r, h-1) + 3$	$l_r + min(u_r, h-1) + 2$
Serial Muls	$3r + 3min(u_r, h-1) + 4$	$3l_r + 3(min(u_r, h-1) + 4$
Remark: MT_{u_r}: the ms-tree containing the refresher, $LN_r(\langle l_r, v_r \rangle$: the refresher)		

Table 5.18.: Complexity of TFAN refresh protocol

- Refresher: $2r + 2min(u_r, h-1) + 1$;
- Non-refresher in MT_{u_r}: $min(u_r, h-1) + r + 2$;
- If $u > u_r$: $u_r + 1$;
- If $u < u_r$: $u + 1$.

If $art = T$, let M_x be leaf node $\langle l, v \rangle$ in MT_u for $0 \leq u \leq h$. The numbers of multiplications per member are listed below:

- Sponsor: $2l_r + 2min(u_r, h-1) + 3$;
- Non-sponsor in MT_{u_r}: $l_r + 1 - i + min(u_r, h-1)$ if $\exists i \in [0, l_r]$ so that $M_j \in MT^{u_r}_{\langle l_r - i, v_r/2^i \rangle} \setminus MT^{u_r}_{\langle l_r - i + 1, v_r/2^{i-1} \rangle}$;
- If $u > u_r$: $u_r + 1$;
- If $u < u_r$: $u + 1$.

5.5. Complexity Analysis

This section analyzes the memory, communication, and computation costs between STR and μSTR, TGDH and μTGDH, μSTR/μTGDH and TFAN. Since STR and TGDH use the Tree-based Diffie-Hellman (TbDH) exchange in the algebraic group $\mathbb{Z}_p^*$, but μSTR, μTGDH and TFAN use TbDH over groups of elliptic points, denoted EC-TbDH. We assume that all protocols use EC-TbDH in order to achieve a fair comparison.

The notation in Figure 4.16 is used with the following modifications and extensions: m' - number of merging groups, and α - sum of heights of all non-highest TFAN trees. Additionally we use $h_t(\widehat{h_t})$ for the height of the initial (updated) tree in (μ)TGDH to distinguish those in TFAN. The height h is not the same in both cases of TFAN, i.e. in a fully filled TFAN tree, if $art = S$, $h = \lceil \frac{n}{m+1} \rceil - 1$, and if $art = T$, $h = \lceil \frac{n}{2^m} \rceil - 1$.

Like in Section 4.5, we focus on the number of stored keys and bkeys, the number of rounds, the total number of broadcast messages, the cumulative broadcast message size, and the serial number of multiplications.

First we gives an overview of the memory costs of all protocols in Table 5.19. We consider here random TGDH trees and half fully filled TFAN trees, i.e. the number of members in each ms-tree is $\lceil \frac{m+1}{2} \rceil$ for $art = S$ and $\lceil 2^{m-1} \rceil$ for $art = T$. In following we first compare the memory costs between STR and μSTR, TGDH and μTGDH, and between μSTR/μTGDH and TFAN.

Table 5.20 gives an overview of communication and computation costs for STR, TGDH, μSTR, μTGDH, and TFAN. We analyze the costs of all six protocols (*setup*, *join*, *leave*, *merge*, *partition*, and *refresh*) between STR and μSTR, TGDH and μTGDH, and μSTR/μTGDH and TFAN.

Protocol		keys	public keys
STR	per LN_l	$n+2-max(2,l)$	$2n-2$
	average	$\frac{n+3}{2}$	$2n-2$
μSTR	per LN_l	$n+2-max(2,l)$	$n+1-max(2,l)$
	average	$\frac{n+3}{2}$	$\frac{n+1}{2}$
TGDH	per $\langle l,v\rangle$	$l+1$	$2n-2$
	average	$\lceil \log_2 n \rceil + 1$	$2n-2$
μTGDH	per $\langle l,v\rangle$	$l+1$	$l+1$
	average	$\lceil \log_2 n \rceil + 1$	$\lceil \log_2 n \rceil + 1$
TFAN	per LN_l in MT_i	$i+l+3$	$i+l+3$
$(art = S)$	average	$\frac{m+\lceil \frac{n}{m+1} \rceil + 5}{2}$	$\frac{m+\lceil \frac{n}{m+1} \rceil + 5}{2}$
TFAN	per $\langle l,v\rangle$ in MT_i	$i+l+2$	$i+l+2$
$(art = T)$	average	$\frac{2m+\lceil \frac{n}{2^m} \rceil + 3}{2}$	$\frac{2m+\lceil \frac{n}{2^m} \rceil + 3}{2}$

Table 5.19.: Memory costs of protocol suites STR, μSTR, TGDH, μTGDH and TFAN

5.5.1. STR vs. μSTR

5.5.1.1. Memory costs

Required memory space for the keys in both protocol suites (exact and average) is equal. The major advantage of μSTR compared to STR is reduced number of public keys that have to be stored. In STR each member needs to store all public keys (bkeys and blinded session randoms), while in μSTR, only its own blinded session random and the public keys in its co-path must be stored. For the member of leaf node LN_l, the storage space of $n+l-3$ public keys can be saved in μSTR compared to STR.

5.5.1.2. Communication and computation costs

The computation costs in all protocols between STR and μSTR are contiguous. The only differences are that in μSTR the group key is first computed by the sponsor after the broadcasting of the message, and that the new share and its public value are picked up from the random-depot, but not generated and computed in real time (precomputing). Hence in the following we analyze only the communication costs.

Setup Both protocols are contiguous. The only advantage of μSTR is that its message size is one less than that of STR.

Join Both protocols need the same number of rounds and messages. Considering the total message size, STR scales linearly in the number of members, while the total message size in μSTR is constant.

Leave, merge, partition, and refresh Both protocol suites need the same number of rounds and messages. Considering the total message size, both protocol suites scale linearly in the number of the remaining members. However μSTR needs only half message size than STR.

Summary As discussed above, μSTR protocols need much less memory costs than STR. μSTR requires less serial computation time than STR, since random-depot (precomputing) is applied and the sponsor broadcasts its message before it computes the group key. In protocol setup, the message size between STR and μSTR is almost same. In all other protocols, μSTR needs much less message size than STR. Hence as a whole, μSTR is more efficient than STR.

5.5.2. TGDH vs. μTGDH

5.5.2.1. Memory costs

Similarly, required memory space for the keys in both protocol suites (exact and average) is equal. The major advantage of μTGDH compared to TGDH is a smaller number of public keys that must be stored. In TGDH each member must store all public keys (bkeys), while in μTGDH, only its own bkey and the bkeys in its co-path must be stored. For the member $\langle l, v \rangle$ the storage space in μTGDH is $2n - l - 3$ public keys less than in TGDH.

Protocol Suite	Protocol	Rounds	Bcasts	Bcast size	Serial Muls
STR	setup	2	$n+1$	$2n-1$	$3n-2$
	Join	2	2	$2n$	6
	Leave	1	1	$n-1$	$\frac{3n}{2}$
	m'-merge	2	$m'+1$	$4(n+k)$ $-2m'-1$	$3k+3$
	Partition	1	1	$n-p$	$\frac{3(n-p)}{2}+3$
	Refresh	1	1	n	$\frac{3n}{2}+3$
TGDH	setup	h_t	h_t^2-h_t+n	$(h_t$-$2)2^{h_t+2}$ $+n+8$	$2h_t-1$
	Join	2	2	$2n+2$	$3h_t$
	Leave	1	1	$2n-4$	$3h_t$
	m'-merge	$\lceil \log_2 m' \rceil$+1	$2m'$	$2(m'+1)(n+$ $k)+m'(m'$-$2)$	$3(h_t+\widehat{h_t})$-2
	Partition	$\min(\widehat{h_t},$ $\lceil \log_2 p \rceil+1)$	$\min(2p,\lceil \frac{n}{2} \rceil)$	$2(n$-p-$1)\cdot$ $min(2p,\lceil \frac{n}{2} \rceil)$	$3\widehat{h_t}$
	Refresh	1	1	$2n-2$	$3h_t$
μSTR	Setup	2	$n+1$	$2n-2$	$3n-4$
	Join	2	2	3	4
	Leave	1	1	$\frac{n}{2}$	$\frac{3n}{2}-2$
	merge	2	$m'+1$	$2n+k+1$	$3k+1$
	Partition	1	1	$\frac{n-p}{2}+1$	$\frac{3(n-p)}{2}+1$
	Refresh	1	1	$\frac{n}{2}+1$	$\frac{3n}{2}+1$
μTGDH	Setup	h_t	n+h_t^2-h_t	h_t^2-h_t+n	$2h_t-1$
	Join	2	2	h_t+1	$3h_t-2$
	Leave	1	1	h_t	$3h_t-2$
	m'-merge	$\lceil \log_2 m' \rceil$+1	$2m'$	h_t+α+$m'\widehat{h_t}$	$3(h_t+\widehat{h_t})$-4
	Partition	$\min(\widehat{h_t},$ $\lceil \log_2 p \rceil$ +1	2min$(2p,\lceil \frac{n}{2} \rceil)$	$(\widehat{h_t}$+1$)\cdot$ $\min(2p,\lceil \frac{n}{2} \rceil)$	$3\widehat{h_t}-2$
	Refresh	1	1	h_t	$3h_t-2$
TFAN (art=S)	Setup	3	$n+h+2$	n+mh+ m+h-1	2m+h-1+ $max(m$,2h-2)
	Join	2	2	4	7
	Leave	1	1	$\frac{m+h}{2}$	$\frac{3(m+h)}{2}$+4
	Merge	2	$m'+1$	h+2α+m'+2	$3\alpha+4$
	Partition	2	$\widehat{h}+1$	-	$6m$
	Refresh	1	1	$\frac{m+h}{2}+3$	$\frac{3(m+h)}{2}+4$
TFAN ($art=T$)	Setup	$m+2$	n+1+(m^2- m+1)(h+1)	n+h-1+(m^2+ m+1)(h+1)	2m+3h-3
	Join	2	2	$m+1$	$3m+1$
	Leave	1	1	$m+\frac{h}{2}+2$	$3m+3\frac{h}{2}+4$
	Merge	2	$m'+1$	h+2α+m'+m	$3m+\alpha+1$
	Partition	$\frac{m}{2}+1$	$\frac{pm}{2}+1$	-	$6m$
	Refresh	1	1	$m+\frac{h}{2}+2$	$3m+\frac{3h}{2}+4$

Table 5.20.: Communication and computation costs of protocol suites STR, TGDH, μSTR, μTGDH and TFAN

5.5.2.2. Communication and computation costs

The computation costs in all protocols between STR and μTGDH are contiguous. The only differences are that in μTGDH the group key is first computed by the sponsor after the broadcasting of the message, and that the new share and its public value are picked up from the random-depot, but not generated and computed in real time (precomputing). Hence in the following we analyze only the communication costs.

Setup Both protocol suites need the same number of rounds and messages. However the message size required in μTGDH is much less than in TGDH. The reduced message size is $((h_t - 2)2^{h_t+2} + n + 8)$ - $(h_t^2 - h_t + n)$ = $4(h_t - 2)2^{h_t} + 8 + h_t - h_t^2 \approx 4nh_t \approx 4nlog_2^n$.

Join, leave, refresh Both protocol suites need the same number of rounds and messages. Considering the message size, TGDH scales linearly in the number of members, while μTGDH scales logically in the number of members.

Merge Both protocol suites need the same number of rounds and messages. Although the message size consumed in TGDH and μTGDH are complex, it is not difficult to find that TGDH scales linearly in the product of the number of merging groups and the merged group size, while μTGDH scales logarithmically in the product of the number of merging groups and the number of merged group size.

Partition Both protocol suites need the same number of rounds and messages. Considering the message size, TGDH scales linearly in the product of the number of rounds and the number of remaining members, while μTGDH scales logarithmically in the product of the number of rounds and the number of remaining groups.

Summary As discussed above, μTGDH protocols require less memory costs, and less bandwidth due to smaller messages than in TGDH. Both protocol suites need the same number of rounds and messages. μTGDH requires less serial computation time than TGDH, since random-depot (precomputing) is applied and the sponsor broadcasts its message before it computes the group key. Hence as a whole, μTGDH is more efficient than TGDH.

5.5.3. μSTR/μTGDH vs. TFAN

As discussed above, μSTR and μTGDH are better than STR and TGDH respectively, hence we do not consider STR and TGDH in this section.

5.5.3.1. Memory costs

Considering TFAN, it is clear that TFAN with μTGDH ms-trees consumes less storage space than with μSTR ms-trees. If we increase m, the allowed maximal height of ms-trees, the member needs to store more keys and public keys of the ms-tree, however the number of keys and public keys outside the ms-tree is decreased.

It is clear that the required memory space of TFAN with μSTR ms-trees is less than μSTR, and of TFAN with μTGDH ms-trees is more than μTGDH. Hence as a whole, all protocol suites can be sorted from the least to highest memory consumption as follows: μTGDH $<$ TFAN ($art = T$) $<$ TFAN ($art = S$) $<$ μSTR.

5.5.3.2. Communication and computation costs

Setup μSTR and TFAN with μSTR ms-trees are the most communication efficient protocols. Only 2 or 3 rounds are needed to form a new group from all individual group members. TFAN with μTGDH ms-trees is comparatively efficient, $m + 2$ rounds are needed. μTGDH is at most inefficient, scaling logarithmically with the number of group members. According to the number of rounds, they can be sorted as follows: μSTR $<$ TFAN ($art = S$) $<$ TFAN ($art = T$) $<$ μTGDH.

We consider now the number of messages and the total message size. According to number of broadcast messages all protocols can be sorted from the least to the highest as follows: μSTR $<$ TFAN ($art = S$) $<$ TFAN ($art = T$) $<$ μTGDH. And considering the total message size, they are sorted as follows: μTGDH $<$ TFAN ($art = T$) $<$ μSTR $<$ TFAN ($art = S$).

With respect to the serial multiplications, μTGDH is most computation efficient, scaling logarithmically in the number of members. μSTR requires a linear number of serial multiplications relative to the group size. The computation costs of TFAN in both cases depend on m. In general TFAN ($art = T$) is more efficient than TFAN ($art = S$), which in turn more efficient than μSTR. Hence all protocol suites can be sorted with respect to serial multiplications as follows: μTGDH $<$ TFAN ($art = T$) $<$ TFAN ($art = S$) $<$ μSTR.

Considering the communication and computation costs, TFAN ($art = T$) is most suitable to handle the setup event.

Join All protocols require 2 rounds and 2 messages. The total message size of STR is only constant, only 3 bkeys. In μTGDH it scales logarithmically in the number of group members. Since the shallowest ms-tree is usually not fully filled, the new member is joined in this ms-tree. In this case, the message size of TFAN ($art = S$) is 4, and of TFAN ($art = T$) is $m + 1$. Hence according to the required communication costs, the protocol suites can be sorted as follows: μSTR $<$ TFAN ($art = S$) $<$ TFAN ($art = T$) $<$ μTGDH.

Under the assumption that the first ms-tree in TFAN is not fully filled, μTGDH is most expensive in terms of computation, it scales logarithmically in the number of serial multiplications. The number of serial multiplications in TFAN ($art = T$) is $3m + 1$, and in TFAN ($art = T$) is 7. μSTR, in turn, uses only 4 multiplications. Hence according to the required computation costs, all protocol suites can be sorted as follows: μSTR $<$ TFAN ($art = S$) $<$ TFAN ($art = T$) $<$ μTGDH.

Considering the communication and computation costs, μSTR is most suitable to handle the join event.

Leave, refresh All protocols require 1 round and 1 broadcast message. μTGDH has the least cumulative message size, while μSTR the highest. TFAN ($art = S$) consumes more bandwidth than TFAN ($art = T$). In general the cumulative message size required by TFAN is between μSTR and μTGDH. According to the required computation costs, all protocol suites can be sorted as follows: μTGDH $<$ TFAN ($art = T$) $<$ TFAN ($art = S$) $<$ μSTR.

According to the cumulative message size and serial multiplications, all protocol suites can be sorted as follows: μTGDH $<$ TFAN ($art = T$) $<$ TFAN ($art = S$) $<$ μSTR. As a whole μTGDH is most suitable to handle leave and refresh event.

Merge We first look at the communication costs. Note that μTGDH scales logarithmically in the number of merging groups, while all other protocols are more efficient using a constant number of rounds. According to number of rounds, all protocol suites can be sorted as follows: μSTRH = TFAN ($art = T$) = TFAN ($art = S$) $<$ μTGDH.

Furthermore μTGDH consumes double broadcast messages as other protocols. Since the cumulative broadcast message size depends on the art and the maximal allowed height m of ms-trees, it is difficult to give the exact relation between TFAN and other protocols. In general, all protocols can be sorted according to the message size as follows: TFAN ($art = T$) $\approx$ TFAN ($art = S$) $<$ μSTR and μTGDH, while the

relation between μSTR and μTGDH depends on the number of merging groups and the tree structures. With respect to computation costs, all protocols can be sorted as follows: μTGDH $<$ TFAN ($art = T$) $<$ TFAN ($art = S$) $<$ μSTR. Hence TFAN ($art = T$) is most suitable to handle the merge event.

With respect to computation costs, all protocols can be sorted according to the message size as follows: μTGDH $<$ TFAN ($art = T$) $<$ TFAN ($art = S$) $<$ μSTR.

Hence TFAN ($art = T$) is most suitable to handle the merge event.

Partition Table 5.20 shows that the μSTR and TFAN ($art = S$) protocols are most communication efficient, i.e. they require only one or two rounds. While partition is the most expensive operation in μTGDH, requiring a number of rounds bounded by the minimum of either the updated tree's height or $\log_2 p + 1$. The number of rounds in TFAN ($art = T$) is bounded by $\frac{m}{2} + 1$. According to the number of rounds, all protocol suites can be sorted as follows: μSTR $<$ TFAN ($art = S$) $<$ TFAN ($art = T$) $<$ μTGDH.

With respect to the computation costs, μTGDH requires a logarithmic number of multiplications, while μSTR scales linearly in the group size. TFAN is comparatively efficient, and less serial multiplications are needed if $art = T$. According to the computation costs, they can sorted as follows: μTGDH $<$ TFAN ($art = T$) $<$ TFAN ($art = S$) $<$ μSTR.

Considering the communication and computation costs, TFAN is most suitable to handle the partition event.

5.5.4. Discussion summary

As discussed above, the μSTR protocol is most suitable in networks where high network delays dominate. However, its computation costs are most expensive. While μTGDH is most suitable in networks with clients having limited computation and storage space, and with low network delays. However, as described in Chapter 1, the clients in ad hoc networks, in general, have limited processor power, storage capability, and energy. Additionally, the network delays dominate in such networks. Considering these properties, TFAN protocol suite is optimal for the group key agreement in ad hoc networks.

5.6. Experimental Results

To compare the actual performance between μSTR, μTGDH and TFAN protocol suites, we implement all six sub-protocols in TFAN, μSTR and μTGDH, and compare their computation and communication costs in this section. The protocol suites are implemented with J2ME CDC [113], and tested with the virtual machine *CDC HotSpot Implementation* [113] on the laptop with Centrino 1.7 GHz processor and 512 MB memory. We simulate the computation process and measure the total computation delay beginning with the time the event occurs and ending with the time the group key is computed.

5.6.1. Test method

To perform fair comparisons, we consider the following settings:

- We use the EC curve P-192 defined in [79]. The security level corresponds to $p = 2720$ in $\mathbb{Z}_p^*$ and $q = 180$ of its sub-group where DL-problem is hard [64]. We use the BouncyCastle API [117] for J2ME to implement the operations over the elliptic curve. Due to the not optimized implementation the EC-TbDH is much slower than expected. However we can not find another opensource project which provides such operations.
- For a better comparison of the costs we omit the generation and verification of signatures, by simulating of the protocols in application level [9]. The reason is that the application layer implements only

[9] We provide the description of implementation levels further in Section 6.

the TFAN API, the underlying levels, i.e. authentication level, GCS layer, and underlying GCS layer, provide additional functions.

- Random trees for μTGDH are used in the simulation. The TFAN trees are half fully filled, i.e. the number of members in each ms-tree is $\lceil \frac{m+1}{2} \rceil$ for $art = S$ and $\lceil 2^{m-1} \rceil$ for $art = T$, and its μTGDH (if $art = T$) ms-trees are random.

We use the following scenarios to measure the computation delay and communication costs. For setup, join, leave and refresh, the number of current group members is $n = 4k, k \in [1, 16]$. For partition, the group size before partition is $n = 16$, 32, and 64. For merge, the group size after merge is $n = 16$, 32, and 64. For simplicity, we denote TFAN with art and m as TFAN(art, m), e.g. TFAN(T, 2) and TFAN(S, 3).

5.6.2. Setup result

We measure the costs from the time all members have collected all setup requests and begin to form the tree until the time the group key is computed. The TFAN group is only measured if the group size is greater than the most allowed number of members in an ms-tree, in other words, TFAN groups with only one ms-tree are not measured.

Figure 5.13 gives the cost comparisons in computational delay, number of rounds, number of messages, and message size. The x axis denotes group size. The y axis in the top left graph denotes the computational delay in seconds, in the top right graph denotes the number of rounds, in the bottom left graph denotes the number of messages, and in bottom right graph denotes the cumulative message size in kilo bytes. The experiment results shown on Figure 5.13 is identical with the theoretic results in the computation and communication costs.

5.6.3. Join result

We measure the costs for a member to join a group of n members. Figure 5.14 shows the cost comparisons in computational delay, and message size. Since the number of rounds and messages is always 2 in all protocol suites, their comparison is omitted. The x axis denotes group size before join. The y axis in the left graph denotes the computational delay in seconds, in the right graph denotes the cumulative message size in kilo bytes. The computation costs shown on Figure 5.14 are identical with the theoretic results. However the message size scales in the group size in all protocol suites. The reason is that in join protocol the sponsor broadcasts the whole tree structure.

5.6.4. Leave result

We measure the costs for a random member to leave a group of n members. We execute in the simulation the voluntary leave. However, to simulate the involuntary leave, all communication costs relative to the voluntary leave request are not measured. We choose the leaving members randomly from the group and compute the average costs.

Figure 5.15 compares costs for computational delay and message size. Since the number of rounds and messages is always 1 in all protocol suites, their comparison is omitted. The x axis denotes the number of group members before leave. The y axis in the left graph denotes the computational delay in seconds, in the right graph denotes the cumulative message size in kilo bytes. The experiment results shown on Figure 5.15 are identical with the theoretic results in the computation and communication costs.

5.6.5. Merge result

We measure the costs from the time when the broadcast sponsors prepare for the merge to the time when the new group key is agreed. The number of resulting group members is 16, 32, and 64. We assume the maximum number of merging groups is 5. In practice, merge of two groups is most frequent. However, we

allow up to five groups since some group communication systems may allow (require) more than two groups merging.

Figures 5.16, 5.17, and 5.18 show the cost comparisons for the cases with group size after merge: 16, 32, and 64 respectively. In all graphs, the x axis denotes the group size before merge, while the y axis in the top left graph denotes the computational delay in seconds, in the top right graph denotes the number of rounds, in bottom left graph denotes the number of messages, and in bottom right graph denotes the cumulative message size in kilo bytes. The experiment results shown on Figure 5.15 are identical with the theoretic results in the computation and communication costs.

5.6.6. Partition result

We measure the costs from the time the partition event occurs. We execute in the simulation the voluntary partition. However, to simulate the involuntary partition, all communication costs relative to the voluntary partition request are not measured. The leaving members are chosen randomly from the group.

Figures 5.19, 5.20, and 5.21 show the cost comparisons for the the cases with group size before partition: 16, 32, and 64 respectively. In all graphs, the x axis denotes the group size before partition, while the y axis in the top left graph denotes the computational delay in seconds, in the top right graph denotes the number of rounds, in bottom left graph denotes the number of messages, and in bottom right graph denotes the cumulative message size in kilo bytes. The experiment results shown on Figure 5.15 are identical with the theoretic results in the computation and communication costs.

5.6.7. Refresh result

We measure the costs from the time when a member, refresher, begins to update its share to the time the new group key is computed. We choose the refreshers randomly from the group and compute the average costs. Figure 5.22 shows the cost comparisons in computational delay, and message size. Since the numbers of rounds and messages are always 1 in all protocol suites, their comparison is omitted. The x axis denotes the group size. The y axis in the left graph denotes the computational delay in seconds, in the right graph denotes the cumulative message size in kilo bytes.

5.6.8. Conclusions

The above experiment results validates the theoretic ones, in other words, TFAN protocol suite is more suitable for ad hoc networks than μSTR and μTGDH. As shown on the Figures 5.13 to 5.21, the costs for TFAN depends on the art and m. From the experiment results, we suggest the following parameters of TFAN for specified group size n [10]: For $n < 24$ TFAN($T, 2$) and TFAN($S, 7$) are most suitable, and for all other group size less than 100 [11] the TFAN($T, 3$) and TFAN($S, 7$) are most suitable.

[10] Group size before join, leave, and partition, and after merge.

[11] In ad hoc networks the number of clients is usually less than 100.

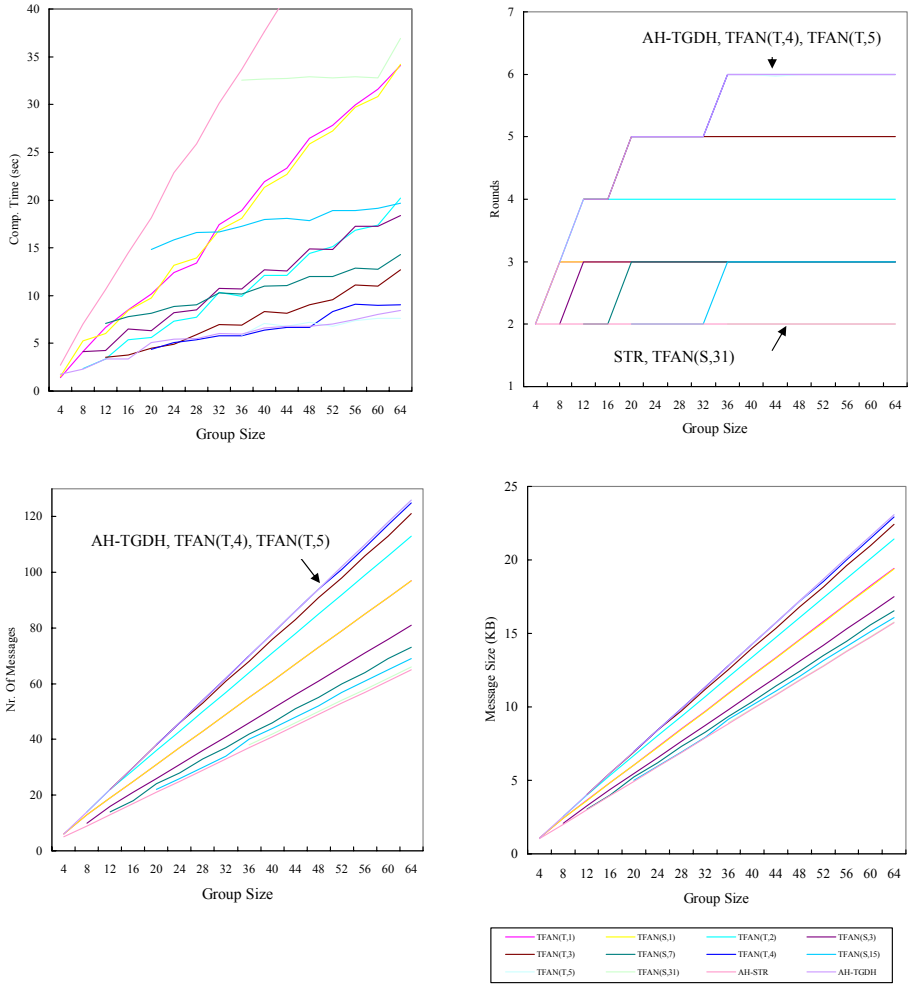


Figure 5.13.: Cost Comparison for Setup: x = group size

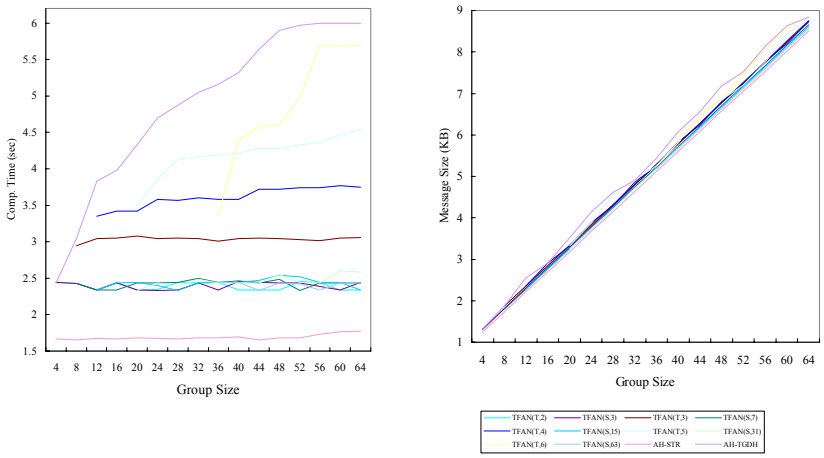


Figure 5.14.: Cost Comparison for Join: x = group size before JOIN

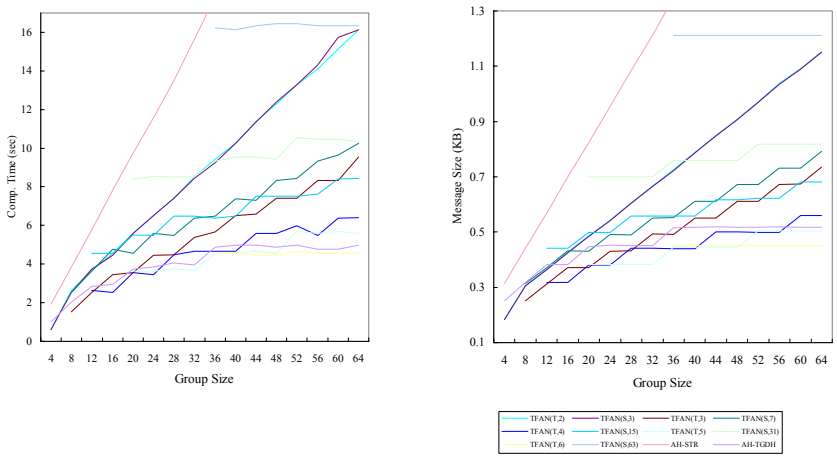


Figure 5.15.: Cost Comparison for Leave: x = group size before LEAVE

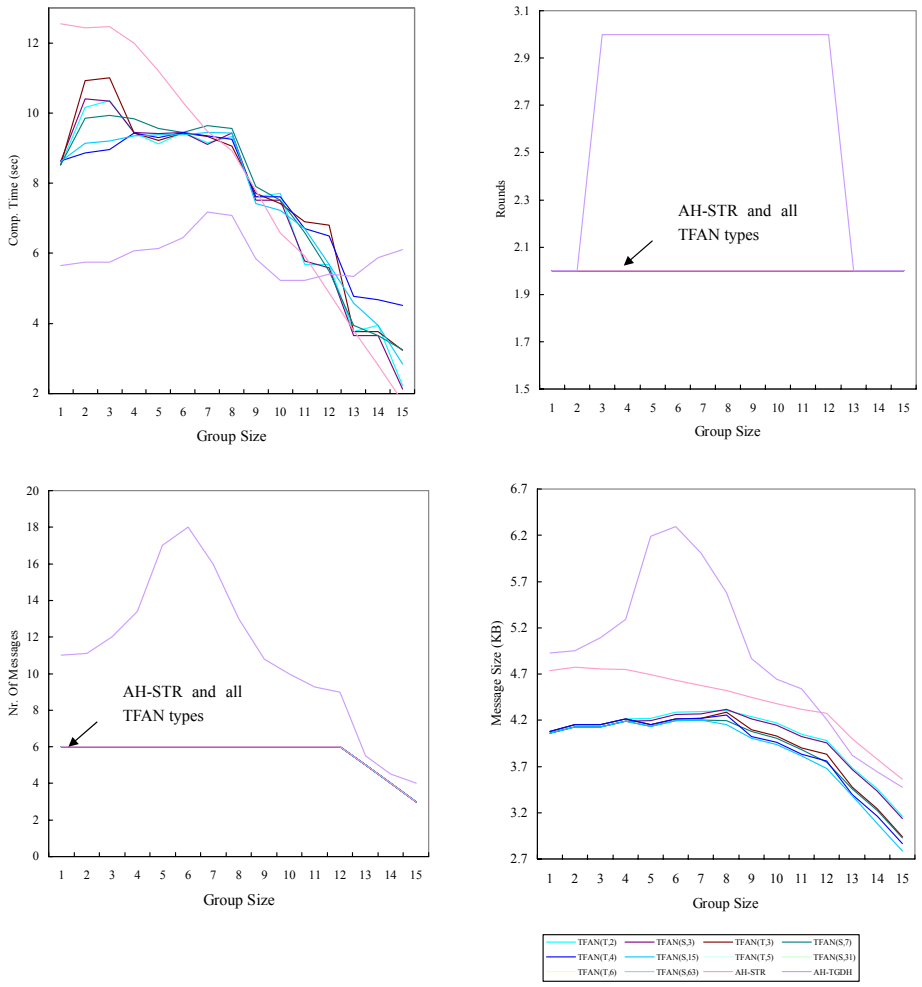


Figure 5.16.: Cost Comparison for Merge (16 users): x = group size before MERGE

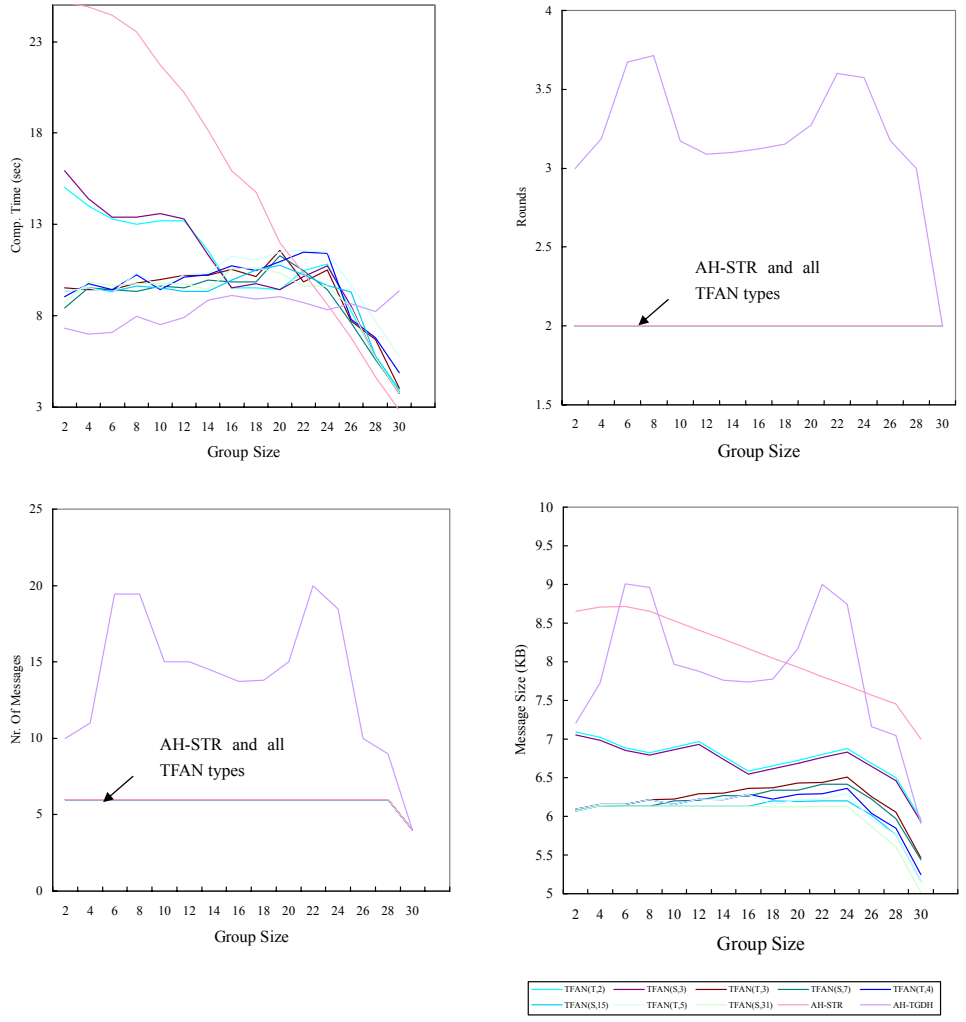


Figure 5.17.: Cost Comparison for Merge (32 users): x = group size before MERGE

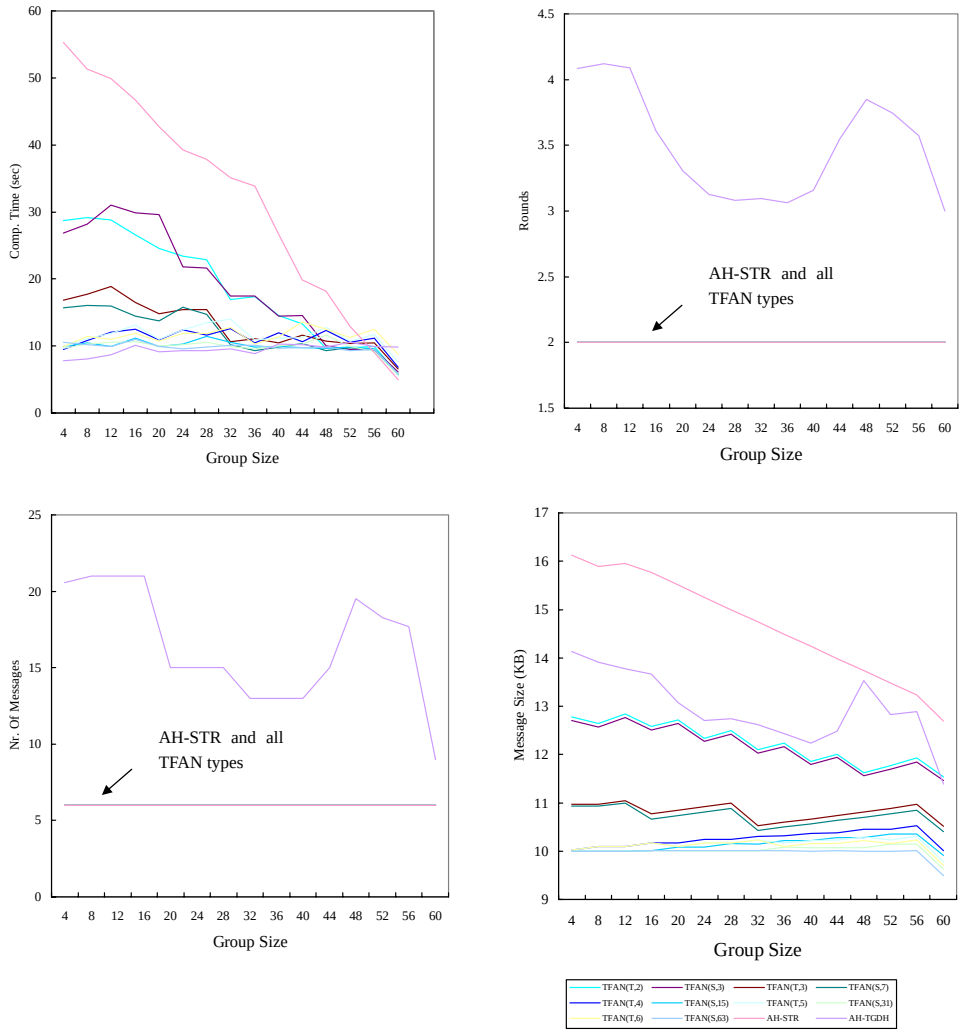


Figure 5.18.: Cost Comparison for Merge (64 users): x = group size before MERGE

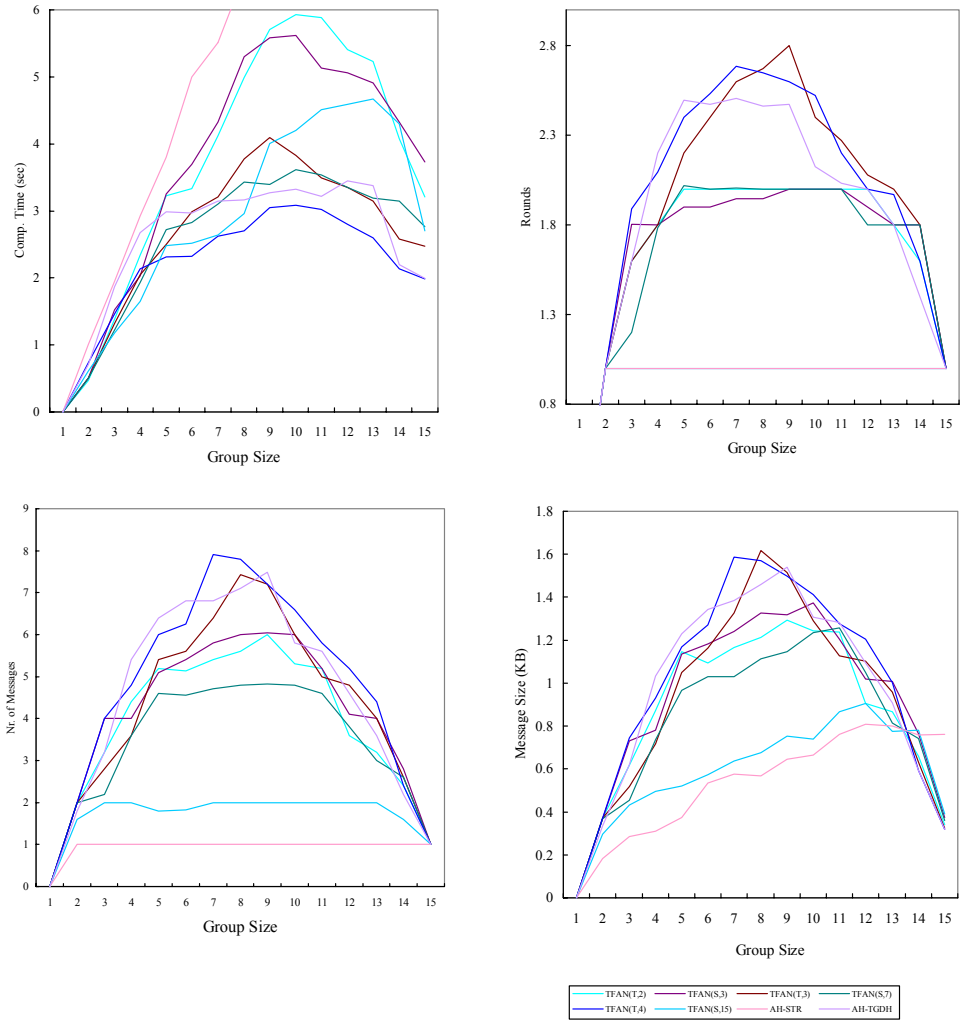


Figure 5.19.: Cost Comparison for Partition (16 users): x = group size before PARTITION

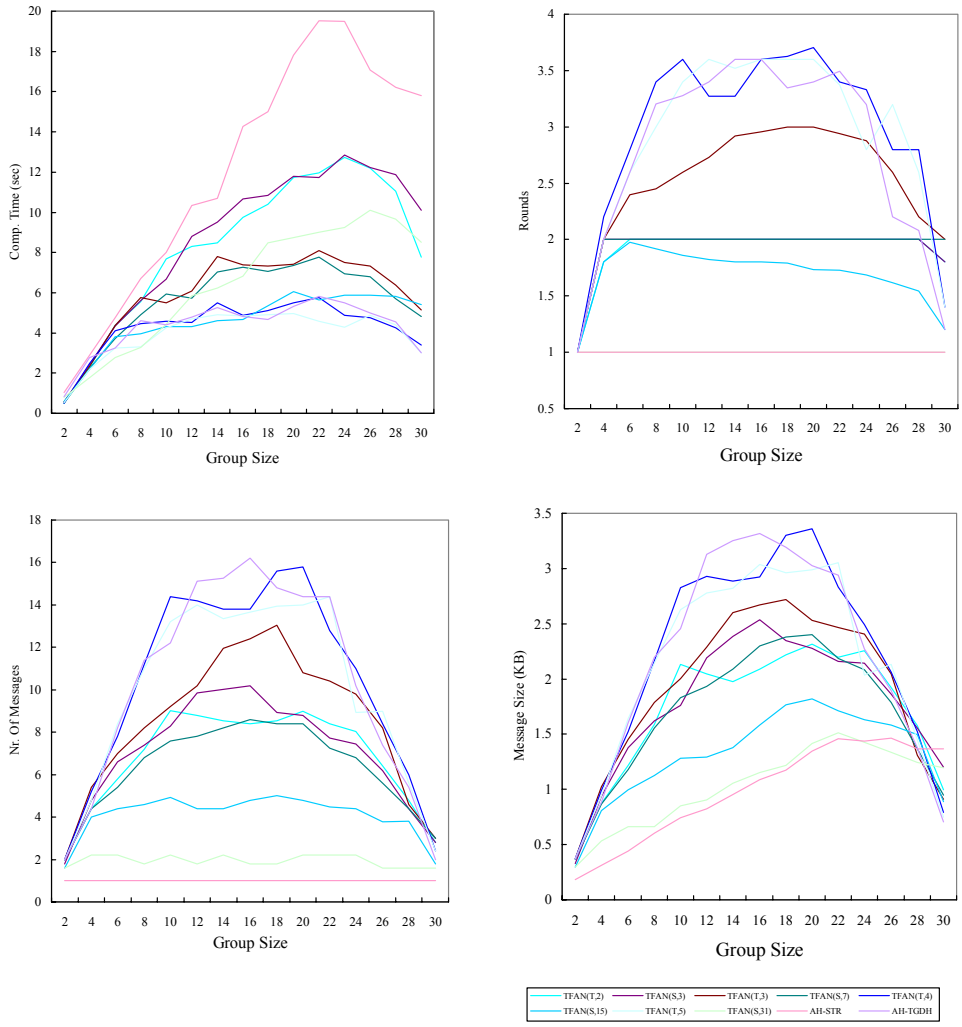
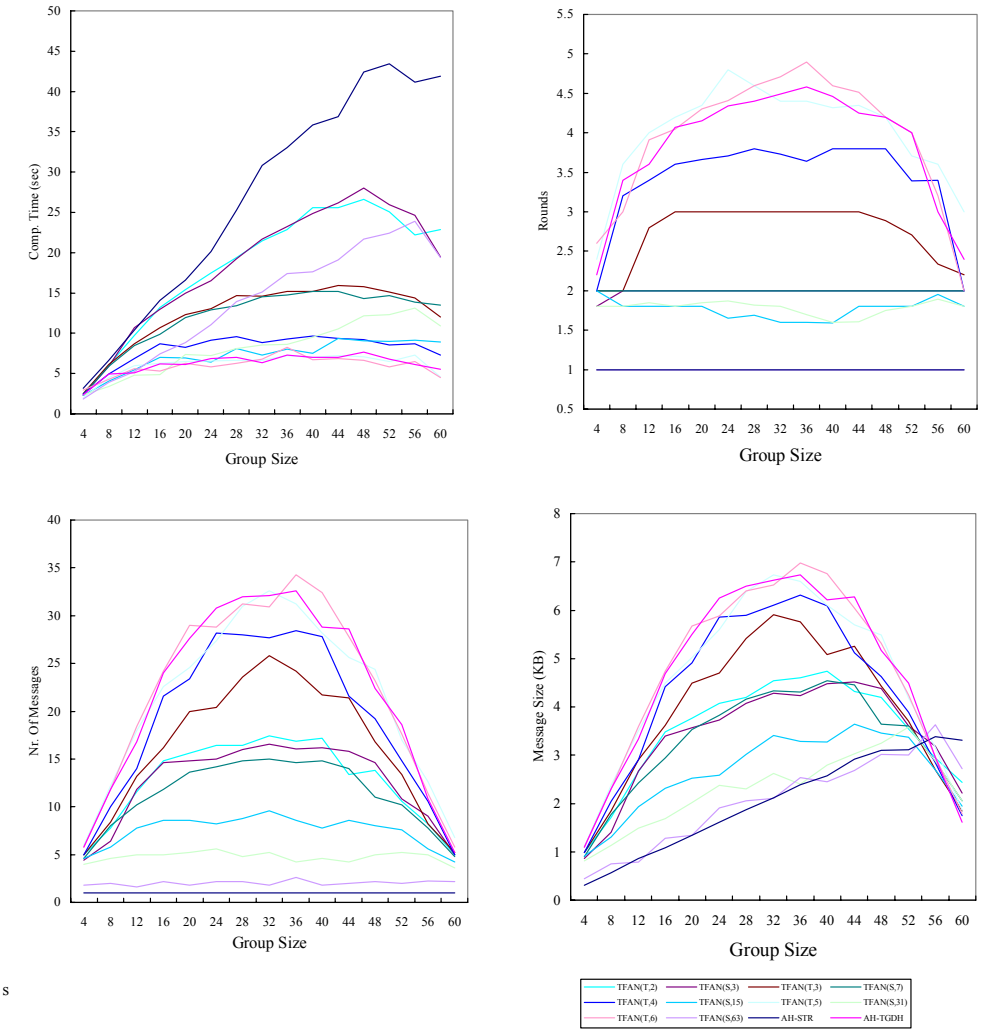


Figure 5.20.: Cost Comparison for Partition (32 users): x = group size before PARTITION



s

Figure 5.21.: Cost Comparison for Partition (64 users): x = group size before PARTITION

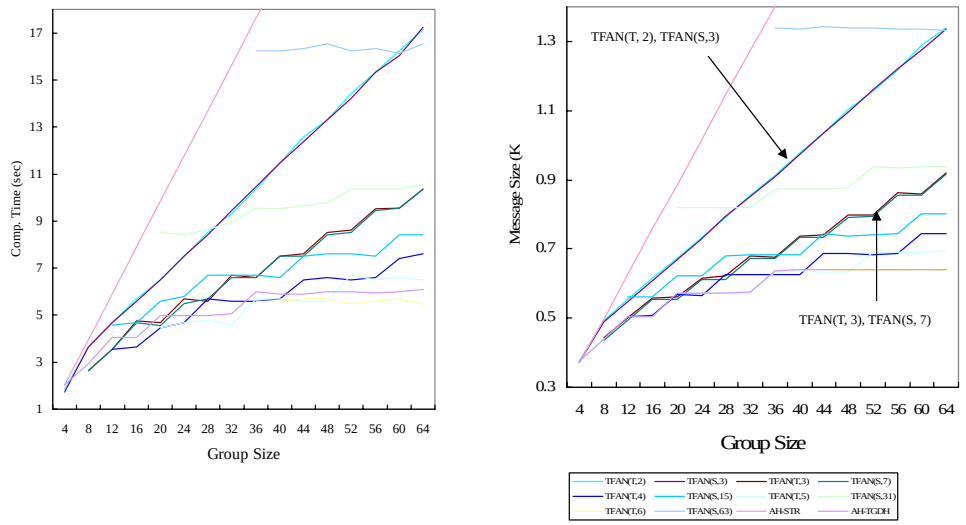


Figure 5.22.: Cost Comparison for Refresh: x = group size

6. TFAN API

In this chapter we describe the TFAN application programming interface (TFAN API). The API implements the TFAN protocol suite in Java. The remaining of this chapter is organized as follows: Section **??** discusses the goals and design principles, and the architecture is overviewed in Section 6.2. In the remaining sections we describe some most important components in TFAN API.

6.1. Goals and Design Principles

The TFAN API is designed to make it simple to agree on a group key with TFAN protocol described in Section 5.4. Since TFAN is designed for ad hoc networks, the TFAN API uses the Connected Device Configuration (CDC) [113] of Java 2 Platform, Micro Edition (J2ME) [115] with Foundation Profile (FP) [114], as the program language. Hence the TFAN API is suitable for all devices supporting at least the Foundation Profile of J2ME CDC, e.g. PDA and laptop.

To use the TFAN API the network must provide a group communication system (GCS). Any GCS can be used if the two given interfaces TfanChannel and ChannelParameter are implemented. Since JGroups [116] are not runnable with J2ME CDC FP, and the JGroups J2ME [49] version does not satisfy our requirements, it was not possible to test the protocol in real ad-hoc networks due to the absence of GCS, therefor simulation on usual PCs was used.

6.2. Architectural Overview

This section describes the TFAN API architecture, defines major classes and interfaces comprising that architecture, and lists major functions that the architecture implements.

6.2.1. TFAN layered architecture

Figure 6.1 shows the TFAN API architectural. The components are layered as shown below:

- The underlying GCS layer implementing the group communication channel, such that it can create groups of processes whose members can send messages to each other. This layer should also be able to report the involuntary events, such as the leaves and partitions. The GCS is response for the following: It receives messages and events, and forwards them to the above layer, and sends the messages generated by the above layer.

  One example of GCS is the JGroups API, it is an open-source toolkit for reliable multicast communication. The main features that are relevant to TFAN API are:

  - Group creation and deletion. Group members can be spread across LANs or WANs,
  - Joining and leaving of groups,
  - Membership detection and notification about joined/left/crashed members,
  - Detection and removal of crashed members,
  - Sending and receiving of member-to-group messages, and

  Any toolkit having the above mentioned features can replace JGroups in the underlying GCS layer.

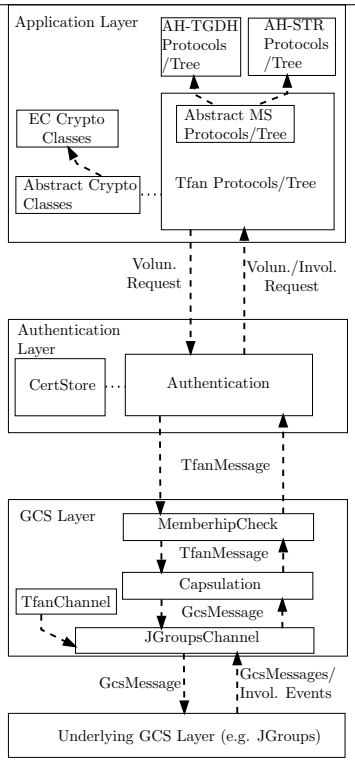


Figure 6.1.: Layer of TFAN API

- The GCS layer declares interfaces and classes intended to handle the messages. It can work in both directions. It receives the messages from the underlying GCS layer, decapsulates them, and then forwards the resulted messages to the above layer: authentication layer. It the reverse direction it encapsulates the messages from the authentication layer and forwards the encapsulated messages to the underlying GCS layer.

  Messages and events are received from the underlying GCS. If an involuntary leave or partition event is received, a respective request is created as a GcsMessage object. The GcsMessage object is then forwarded to the Capsulation object. Unexpected messages are rejected and the accepted messages are then decapsulated to a TfanMessage object.

  The MembershipCheck object checks further whether the message should be accepted or not. Details about the membership check are described in Section 6.4.

- The authentication layer declares the classes to manage the certificates, and to generate or verify signatures. The incoming voluntary request with invalid signature will be rejected. Since involuntary requests are generated by the local underlying protocols, they are always forwarded. Setup requests, join requests, and merge requests from the member of other groups are signed with the embedded sender's certificate. All other voluntary requests, e.g. refresh request and leave request are verified with the stored sender's certificate.

  The outgoing requests, forwarded by the application layer, must be signed so that they can be accepted by other members. For the setup request and join request, the sender's certificate is embedded, and for the merge request, all certificates of the group members are embedded in the message. The requests together with available certificates are then signed and forwarded to the GCS layer.

- The application layer is the core layer in TFAN API. It declares interfaces and classes to handle the agreement of group key of protocol suites TFAN. It covers all protocols in TFAN: setup, join,

leave, merge, partition, and refresh. TFAN protocol suite uses the ms-tree, hence some abstract ms-protocols and ms-trees classes are defined, which must be implemented by any type of ms-protocol suite, e.g. μTGDH tree and μSTR tree. The TFAN protocols and trees access methods of μTGDH and μSTR protocols and trees via the abstract ms classes. The TFAN protocols and tree accesses the cryptographic classes via the abstract cryptographic classes, the advantage is to make it possible to use other classes which implement these abstract ones. In the actual version, the classes for Diffie-Hellman with the elliptic curve cryptosystem are implemented.

6.2.2. TFAN class hierarchy

Figure 6.2 shows most classes and interfaces comprising the TFAN API. The descriptions of some classes

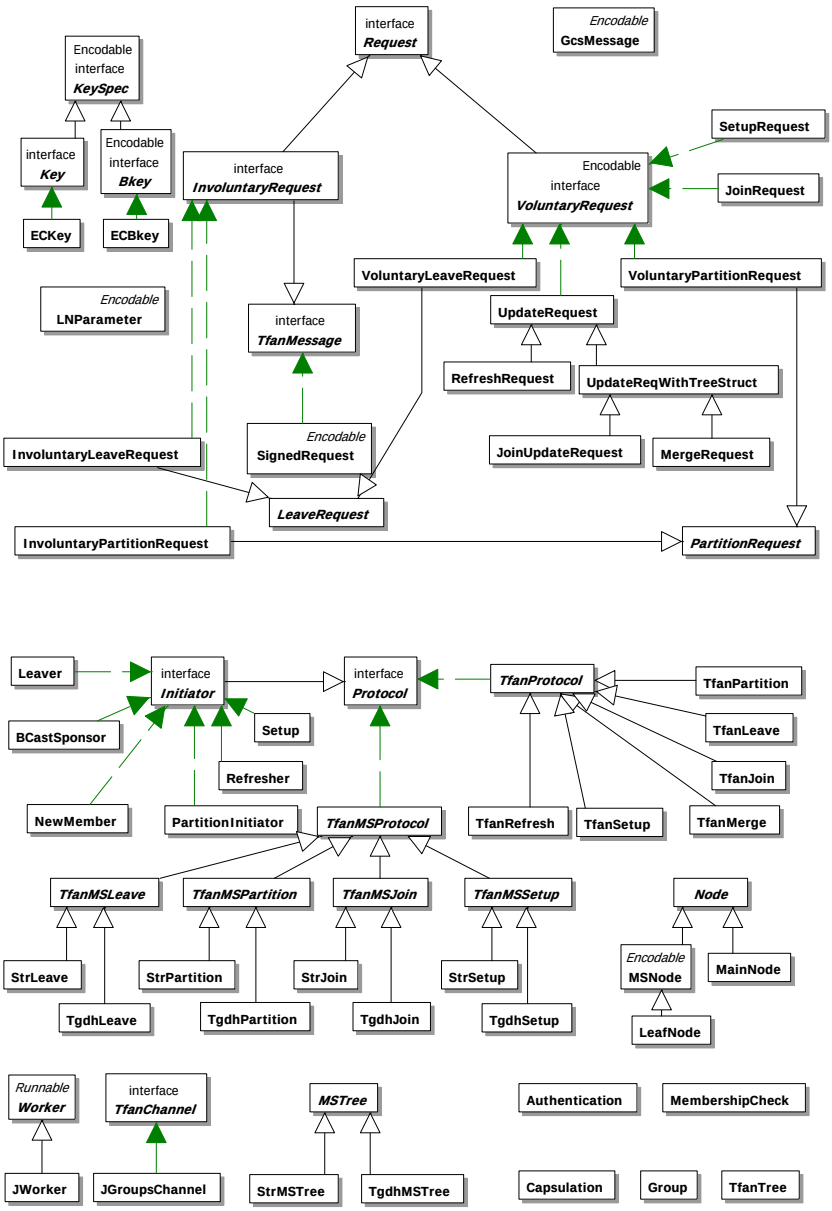


Figure 6.2.: Classes of TFAN API

shown in this diagram will be described in the following sections. All other classes and interfaces can be found in the javadoc documents.

6.2.3. Using the TFAN API

This section defines the syntax and lists the order in which a client application calls some TFAN methods in order to setup a group and to handle the dynamic operations.

We start with two configuration files: group configuration file and client configuration file. Introduction about the configurations can be found in examples in Appendix A.1 and A.2. All members in one group should have the same group configuration, and each member in one group has the unique identity.

1. A TFAN client typically begins a task by obtaining the object of the class implementing the abstract class Worker, e.g. JWorker. Then a thread with the object will be started. This thread runs until the method worker.interrupt() is invoked or the thread is interrupted.

```
Worker worker = new JWorker(groupconfig, memberconfig,
                "keystorepwd", "keypwd");
Thread thread = new Thread(worker);
thread.start();
```

2. If no group exists, the client broadcasts a setup request to the group by invoking `worker.setup()`, then the client invokes automatically the setup protocol with the defined delay, if any other setup requests are received. Otherwise it forms a group with itself as the unique member. The setup of the tree, the update of bkeys, the computation of keys and bkeys, the broadcast of the bkeys, etc., will be done by the program automatically until the group is formed.

3. Assumed that a group with the same group name exists. The client joins into the group. It broadcasts a join request by invoking `worker.join()`. As in setup protocol, all remaining operations will be done by the program automatically.

4. If a client within a TFAN group wishes to leave the group, it just broadcasts a leave request by invoking `worker.leave()`. And if a client wishes to split the group, it chooses first other members which should be in the same sub-group, stores them in `othersInMySubGroup`, an object of `java.util.List`, and invokes `worker.partition(othersInMySubGroup)`.

5. If more than two groups wish to merge into one group, the broadcast sponsor of each group broadcast broadcasts the respective group by invoking the method `worker.merge()`.

6. If some involuntary events happen, such as the crash of some members, all remaining members execute then the leave or partition protocol.

6.3. The classes `Worker` and `Group`

Figure 6.3 shows the UML diagram of both classes. The `Group` class defines a set of attributes: *channelParameter* (`ChannelParameter` object), *gkp* (`GroupKeyParameter` object), *authentication* (`Authentication` object), and *groupname*, to manage the TFAN group. The core attribute is `tree`, a `TfanTree` object. The *groupname* is used by the `Capsulation` object *capsulation*. The description of class `TfanTree` is given in Section 6.6.2, of class `Authentication` in Section 6.5, the others in Section 6.4.

The class `Worker` is an abstract class, the reason is that some functions depend on the used underlying GCS system and the used DH-type. We have implemented a sub-class `JWorker` using the JGroups GCS and the ECDH. `Worker` prepares its sub-classes in two ways to build a worker. The one is with a `WorkerControll` object, a `TfanChannel` object and a `Group` object as parameters. The other with a file of group configuration, a file of client configuration, the passwords to access the owner's private key and certificates stored in an `java.security.KeyStore` object.

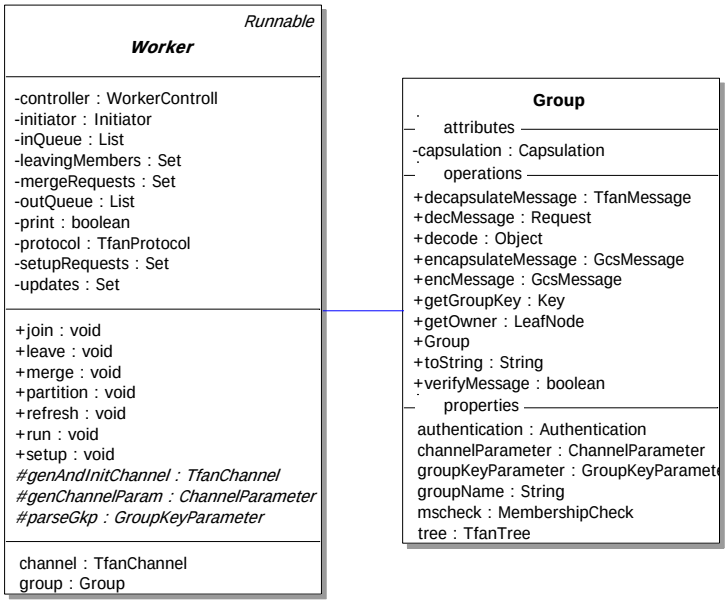


Figure 6.3.: UML diagram of `Worker` and `Group`

`Worker` manages the following queues: *outQueue* stores the messages to be sent, *mergeRequests* stores the merge requests of other groups, *leavingMembers* stores the members having left the group since last partition or leave operation. *inQueue* stores the not handled incoming messages, except for the merge, leave, and partition requests.

As described in Section 6.2.3, the methods *setup*, *join*, *leave*, *merge*, *partition* and *refresh* defined in `Worker` can be used to initiate the respective protocols.

6.4. Components in GCS layer

The most important components in GCS layer are the interface `TfanChannel` and its implementation class `JGroupsChannel`, the classes `Capsulation`, `MembershipCheck` and `GcsMessage`. All components are defined in package *org.tfan.gcs* and its sub package. The UML diagram of them is shown in Figure 6.4.

`GcsMessage` defines two attributes: *groupname* and *tfanMessage*. All requests/messages sent and received by `TfanChannel` are `GcsMessage` objects.

`TfanChannel` declares the abstract methods to initialize the channel, to get the state of the channel, to connect or disconnect/close the channel, to send and receive the channel. Its implementation class, e.g. `JGroupsChannel` should implement all methods, with the respect to the underlying GCS layer. It sends and receives the `GcsMessage` objects. Additionally if some members leave the group involuntarily, the channel generates the respective `InvoluntaryLeaveRequest` or `InvoluntaryPartitionRequest` object, according to some information - such as the view in JGroups, and then encapsulates it in a `GcsMessage` object.

A `Capsulation` object encapsulates an outgoing `SignedRequest` object into a `GcsMessage` by adding the group name. It decapsulates also the incoming `GcsMessage` objects. The message containing `MergeRequest` object is accepted without any check, all other messages are only accepted if they have the same group name as the `Capsulation` object. The accepted GCS messages are then decapsulated to `TfanMessage` objects by removing the group name.

The attribute *mscheck* checks the membership of the sender of the incoming messages. All leave, refresh, partition requests, and update requests (except for merge requests) are only accepted if their senders are

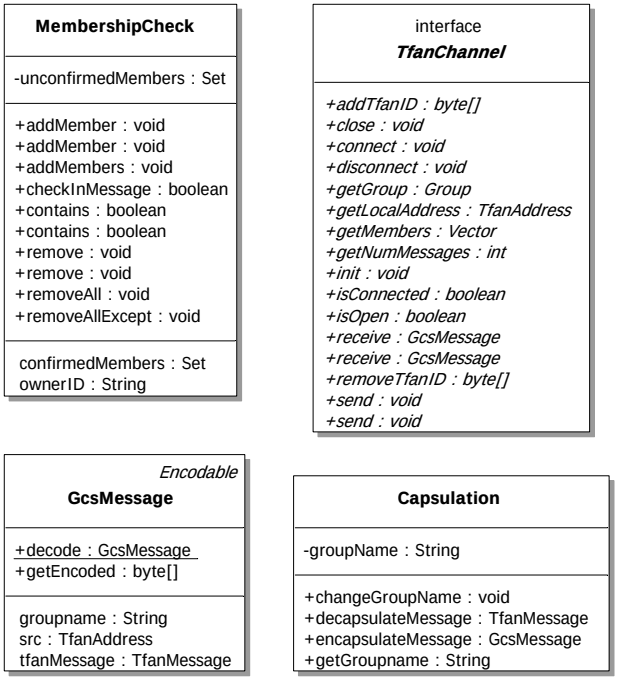


Figure 6.4.: UML diagram of components in GCS layer

members of the receiver's group. It rejects setup requests, if there are at least two members in the group, and rejects join requests, if the sender's identity is used in the group. Merge requests are only accepted if both groups do not have any same member identity. The leaving members will then be removed from the list of members, and the joining members are added to the list.

6.5. Components in authentication layer

The core components in authentication layer are the classes `SignedRequest`, `Authentication`, `CertStore`, and the interface `TfanMessage`, all of them are defined in package *org.tfan.auth*. The UML diagram of them is shown in Figure 6.5.

`TfanMessage` declares no abstract methods. It has the sub-interface `InvoluntaryRequest` and the implementation class `SignedRequest`. The latter defines a set of attributes:

- *mdAlgo* - The message digest algorithm, e.g. „ SHA-1“ and „RipeMD-160“;
- *request* - The voluntary request, such as the object of `JoinRequest` and `VoluntaryLeaveRequest`;
- *encodedCerts* - A list of encoded user certificates. If the request is a join and setup request, the sender's certificate is sent together with the request. And for the merge request, the broadcast sponsor embeds all encoded certificates of the current members in the `SignedRequest`. The voluntary request with embedded certificates are signed before the broadcasting.
- *signature* - The signature built over the voluntary request and the embedded certificates.

`CertStore` is used to manage the certificates. Since the ad hoc devices have in general limited memory, all certificates are stored in the file system instead of memory, and the respective file locations are stored in the maps *subjMapFile* and *issuerSnMapFile*. In both maps, the value of each entry is the certificate file location, and the key is the certificate's subject or the combination of the issuer and the serial number. Invoking the method *getCert(CertID certID)* a certificate returns, or null if no certificate matches the *certID* returns.

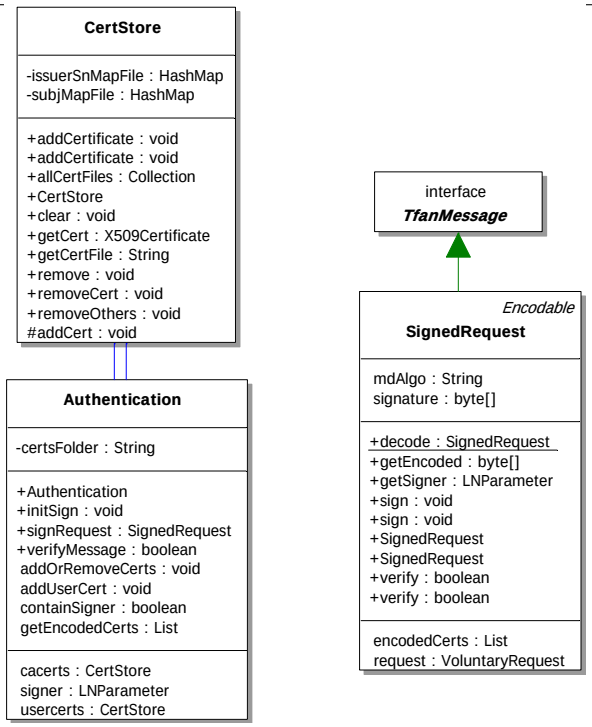


Figure 6.5.: UML diagram of components in authentication layer

The `Authentication` object embeds the certificates into outgoing join, leave and merge requests, and signs all outgoing voluntary requests with the *signer*'s private key. It defines two `CertStore` objects: *cacerts* to manage the trusted CA certificates, and *usercerts* to manage the certificates of the group members. Certificates of leaving members will be removed from *usercerts*. The signatures of all incoming `SignedRequest` objects are verified by the `Authentication` object. Since those with involuntary requests are not signed, they are forwarded without the signature verification. All others with invalid signature are rejected. The new members' certificates are stored in folder *certsFolder* and the locations are managed by *usercerts*, and certificates of the leaving members are removed from *usercerts* and the file system.

The authentication can be replaced by any other authentication mechanism through implementation of required interfaces.

6.6. Components in application layer

This section discusses the most important interfaces and classes in application layer. They are further classified into the following categories:

- *Key components* (Section 6.6.1);
- *Tree components* (Section 6.6.2);
- *Request components* (Section 6.6.3);
- *Protocol components* (Section 6.6.4).

6.6.1. Key components

Interfaces and classes falling into this category are the interfaces `KeySpec`, `Key`, `Bkey`, and `GroupKeyParameter`, and the classes `ECKey`, `ECBkey`, `ECGroupKeyParameter`, and `KeyPair`. They are in

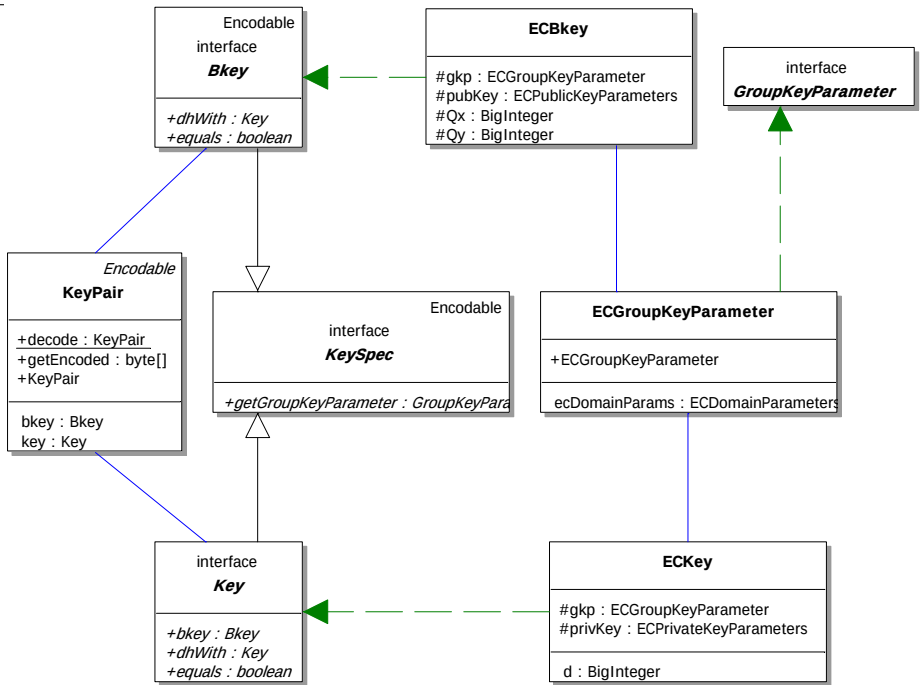


Figure 6.6.: UML diagram of key components

package *org.tfan.crypto* and its sub-package *org.tfan.crypto.ec*. Figure 6.6 shows the UML diagram of the above components.

The interface `KeySpec` defines one abstract method *getGroupKeyParameter* which returns `GroupKeyParameter` object. Its sub-interfaces `Key` and `Bkey` represent the key and bkey defined in TFAN protocol suite. A key pair consisting of a key and its corresponding bkey.

`Key` declares the following methods:

- *dhWith(Bkey)* to execute Diffie-Hellman with a bkey,
- *equals(Key)* to check the equality with another key,
- *bkey()* to derive the corresponding bkey.

Similarly `Bkey` defines the methods *dhWith(Key)* and *equals(Bkey)*.

The remaining components are the implementation classes of the above interfaces. They are implemented for elliptic curve Diffie-Hellman (ECDH).

6.6.2. Tree components

Components falling into this category can be further classified as follows:

- *node components*: the abstract class `Node` and its sub-classes `MainNode`, `MSNode` and `LeafNode`;
- *ms-tree components*: the abstract class `MSTree` and its sub-classes `StrMSTree` and `TgdhMSTree`;
- *tfan tree components*: the class `TfanTree`.

The UML diagram of these components is illustrated in Figure 6.7. For simplicity, all methods and unimportant attributes are omitted. We refer the reader to the javadoc documents for a more detailed information.

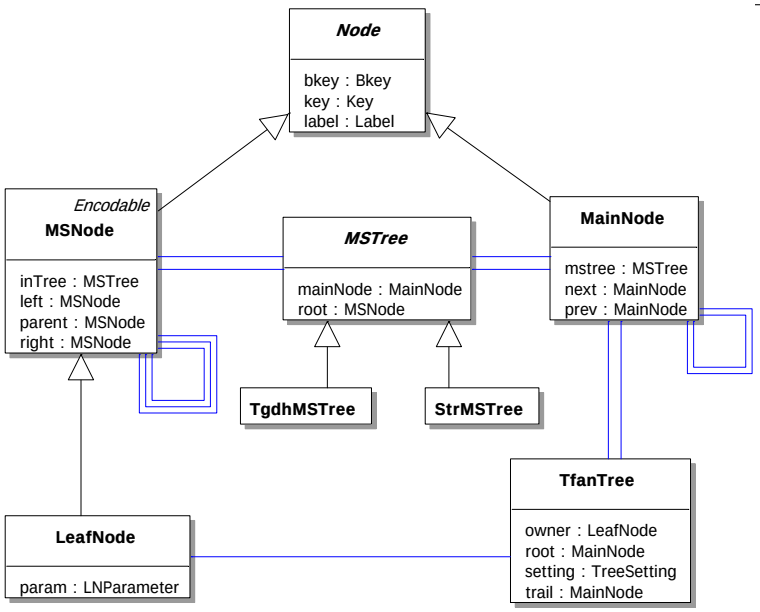


Figure 6.7.: UML diagram of tree components

Each `Node` object has at most a key and a bkey, and must have a label. The type of label depends on the node's type, and the ms-tree's type in case of `MSNode` objects. For example, a main node has the `MNlabel` object, e.g. MN_2, an ms-node in an STR ms-tree has the `StrLabel` object, e.g. IN_8 or LN_8, and the label of an ms-node in a TGDH ms-tree is a `TgdhLabel` object, e.g. $\langle 2, 3 \rangle$.

The class `MSNode` extends the `Node` with some attributes. The attributes *parent*, *left* and *right* define the parent node , left child and right child bides respectively. The attribute *inTree* defines the ms-tree which contains this ms-node. Its sub-class `LeafNode` extends it further with the attribute *param*, an `LNParameter` object. *param* stores the member's information, such as the identity, the IP-address and port, the private key, and the certificate. Since a leaf node has no children, the attributes *left* and *right* must be null.

The class `MainNode` extends the `Node` with the following attributes: *prev* and *next* define the previous and next main node respectively, namely the parent and the left child. And *mstree* defines the ms-tree whose root is the right child of the main node.

Each `MSTree` object has two attributes, *root* defines the root of the ms-tree, and *mainnode* is the parent of *root*. It also defines some abstract methods which should be implemented by the sub-classes, such as `StrMSTree` and `TgdhMSTree`.

The class `TfanTree` defines the attributes and methods to manage the tree used in TFAN protocol suites. The attribute `owner`, a `LeafNode` object, defines the owner of this tree, more strictly, only *owner* has key and the tree is of *owner*'s view. The tree configuration (art and m) is stored in *treesetting*. The attributes *root* and *trail* define the first main node and the last one [1] respectively.

6.6.3. Request components

Requests are used to communicate among the groups, surely they must be encapsulated in other containers. Since there are various requests, the interface `Request` is defined to cover all requests, i.e. all requests must implement `Request`. The UML diagram of the interfaces and classes relative to requests are shown in Figure 6.8.

[1] In real TFAN protocol suite, the last main node and the root of the last ms-tree are represented by the same entity. However, since the main node is a `MainNode` object and the ms-tree's root is an `MSNode` object, it is more convenient to define two nodes: a main node as *trail* and a ms-node as the root of *trail*'s ms-tree. The *trail* is just a placeholder, all accesses to it are forwarded to its ms-tree's root.

According to the generation they can be classified into voluntary requests and involuntary requests, hence they must implement the interfaces `VoluntaryRequest` and `InvoluntaryRequest` respectively, which are sub-interfaces of `Request`. The only two involuntary requests are the involuntary leave request (corr. to the class `InvoluntaryLeaveRequest`) and the involuntary partition request(corr. to the class `InvoluntaryPartitionRequest`). All other requests are voluntary requests.

According to the function the requests can be classified into:

- *Setup request*, corresponding to the class `SetupRequest`. The setup request is applied in the setup protocol. Its unique attribute is *member*, a `LeafNode` object, who wishes to form a new group. Since the request will be broadcasted, no private information should be contained. Hence the attribute *key* and the attribute *signKey* of the attribute *param* should be null.
- *Join request*, corresponding to the class `JoinRequest`. The join request is applied in the join protocol. Its unique attribute is *member*, a `LeafNode` object, who wishes to join into the existing group. Similarly no private information should be contained in the join request.
- *Leave request*, corresponding to the abstract class `LeaveRequest`. The leave request is applied in the leave protocol. It can be voluntary or involuntary. Its unique attribute is *leaver*, an `LNParameter` object without private information, who leaves the group. A voluntary leave request is generated by the leaving member, where as an involuntary leave request is generated by the channel (see Section 6.4).
- *Partition request*, corresponding to the abstract class `PartitionRequest`. The partition request is applied in the partition protocol, and can be voluntary or involuntary. The partition request contains a list of members who leave the group. The voluntary partition request contains additionally the attribute *sender*, which defines the sender of this request. The sender must be one of leaving members. Similar to the involuntary leave request, a involuntary partition request is generated by the channel.
- *Update request*, corresponding to the class `UpdateRequest`. An update request contains the sender and some bkeys. The bkeys are stored in a `java.util.HashMap` object. The class `UpdateRequest` defines methods, which can be invoked to get the bkey of the object that implements the interface `Node`. Sponsors generate the `UpdateRequest` objects to broadcast the changed bkeys. Hence it is used in any protocol that requires sponsors.

  Sometimes the tree structure must also be broadcasted. For this purpose we define the class `UpdateReqWithTreeStruct`, a sub-class of `UpdateRquest`. Since in join protocol, the new member does not know the tree structure, the sponsor generates a `JoinUpdateRequest` object, which is sub-class of `UpdateReqWithTree-Struct`, instead of an `UpdateRequest` request. Another example is the merge request (see below).
- *Merge request*, corresponding to the class `MergeRequest`, a sub-class of `UpdateReqWithTree-Struct`. The merge request is applied in the merge protocol, and can be generated only by the broadcast sponsor. It contains the sender, tree structure and bkeys of all sub-roots. Additionally the attribute *groupname* is defined, which stores the current group name. It is used to choose the group name after the merge operation.
- *Refresh request*, corresponding to the class `RefreshRequest`, a sub-class of `UpdateRequest`. The refresh request is applied in the refresh protocol. Every refresh contains a refresher, and some changed bkeys.

6.6.4. Protocol components

The interfaces and classes falling into this category are the core components of TFAN API. The UML diagram of most of them is shown on Figure 6.9.

According to the function they can be further classified into the following categories:

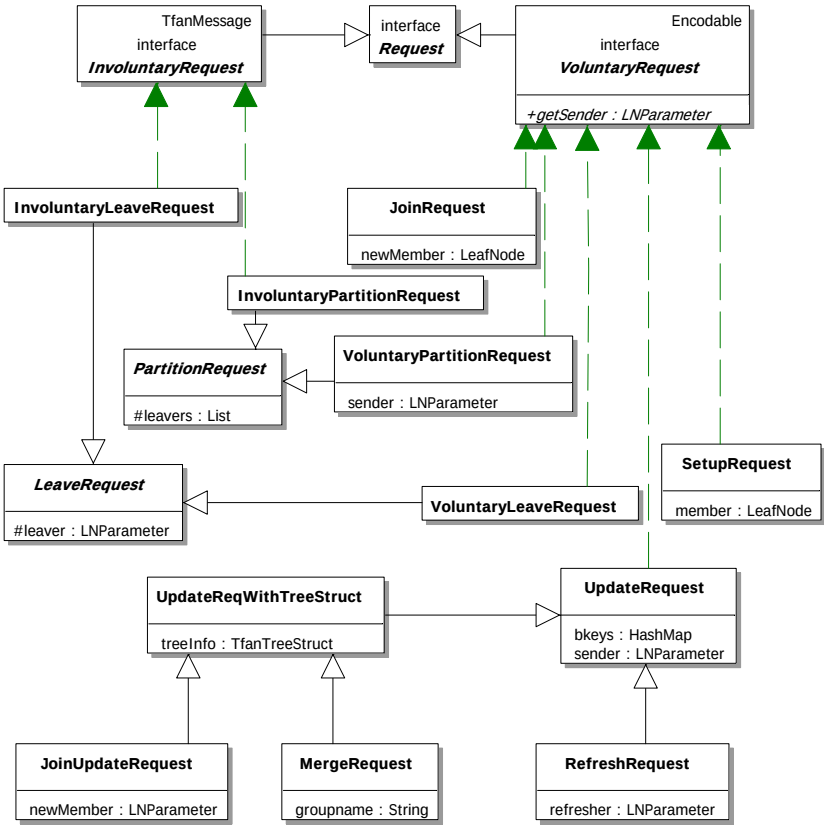


Figure 6.8.: UML diagram of request components

- *The interface* `Initiator` *and its implementation classes*: The interface `Initiator` declares an abstract method *getRequest()* which returns an voluntary request. Its implementation classes define the initiators of all six protocols:
  - *The class* `Setup`: Each member wishes to form a new group should first construct a `Setup` object and then get the setup request with the method *getRequest()*.
  - *The class* `NewMember`: A member wishes to join a group should first construct a `NewMember` object and then get the join request with the method *getRequest()*.
  - *The class* `Leaver`: A member wishes to leave the group should first construct a `Leaver` object and then get the leave request with the method *getRequest()*.
  - *The class* `PartitionInitiator`: To split the group into two sub groups, one of the members will be chosen as the partition initiator. The initiator chooses the members who are in its sub group, constructs a `PartitionInitiator` object, and then get the partition request with the method *getRequest()*.
  - *The class* `MergeRequest`: Only the broadcast sponsor in a group is allowed to send the merge request. It constructs first a `BCastSponsor` object and get the merge request with the method *getRequest()*.
  - *The class* `Refresher`: A member wishes to refresh its share should first construct a `Refresher` object and then get the refresh request with the method *getRequest()*.
- *The abstract class* `TfanMSProtocol` *and its sub classes*: The class `TfanMSProtocol` is abstract, it defines the attributes *mstree* to store the ms-tree relative to the protocols in ms-tree, and *sponsors* to store the sponsors. Since the protocols' processes depend on the used ms-tree, the four sub-classes `TfanMSSetup`, `TfanMSJoin`, `TfanMSPartition` and `TfanMSLeave` are further abstract. They can be used by the TFAN protocols to access their sub-classes. Each class has then

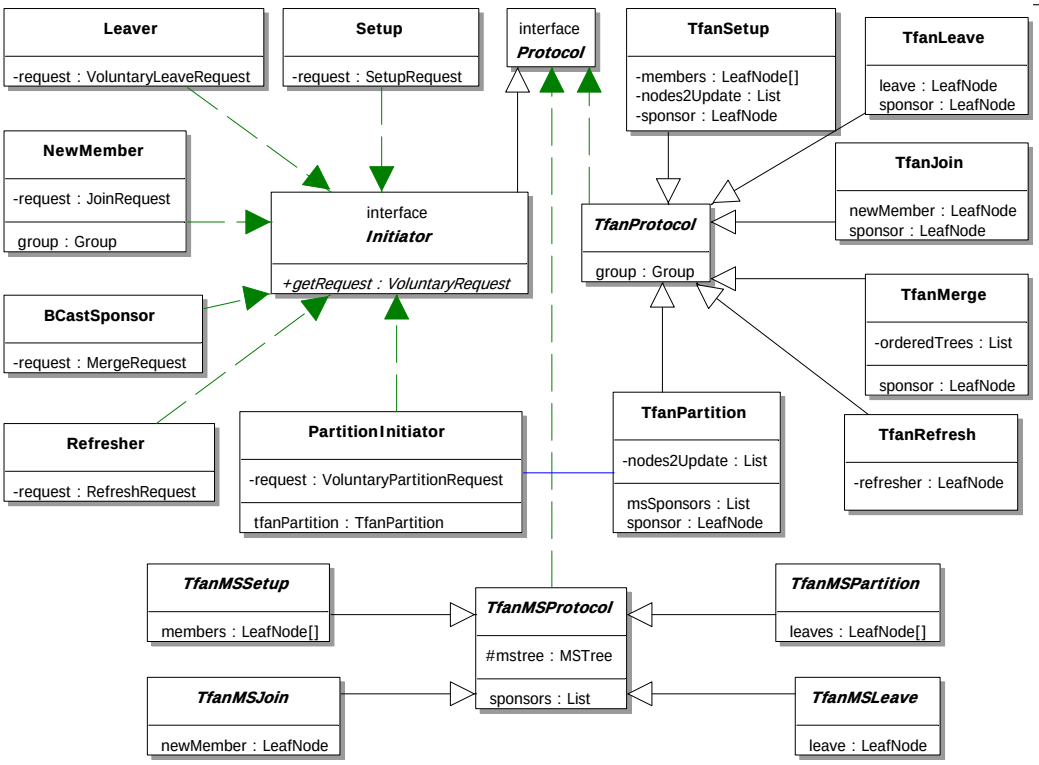


Figure 6.9.: UML diagram of protocol components

two sub-classes: the one for STR ms-tree, the other for TGDH ms-tree, e.g., the sub-classes of `TfanMSSetup` are `StrSetup` and `TgdhSetup`.

- *The abstract class `TfanProtocol` and its sub classes*: The class `TfanProtocol` is the super class of all TFAN protocol classes. The unique attribute of it is *group*. It stores the group which is the target of the protocols. The six TFAN protocol classes are described below.
  - The class `TfanSetup` is used to execute the setup protocol in TFAN protocol suite. The attribute *members* stores all members who wish to form a new group, the sponsor stores the member who should compute and broadcast the bkeys of the main nodes after the collection of bkeys of all sub-roots, and *nodes2update* stores the nodes whose bkeys should be computed and broadcasted by the owner of the group.
  - The class `TfanJoin` is used to execute the join protocol. The attributes *newMember* and *sponsor* store the new member and the sponsor respectively.
  - The class `TfanLeave` is used to execute the leave protocol. The attributes *leave* and *sponsor* store the leaving member and the sponsor respectively.
  - The class `TfanMerge` is used to execute the merge protocol. The unique attribute is *orderedTrees*, i.e. the list of trees whose groups should be merged. The trees in the list are sorted according the criteria described in Section 5.4.4.
  - The class `TfanPartition` is used to execute the partition protocol. The attributes *leavingMaps* stores the leaving members, *msSponsors* stores the ms-sponsors and *sponsor* stores the sponsor. The attribute *nodes2update* is similar to that in `TfanSetup`.
  - The class `TfanRefresh` is used to execute the refresh protocol. The unique attribute is *refresher*.

7. Conclusions

The purpose of this thesis was to find an efficient and secure group key agreement protocol suite suitable especially for ad-hoc networks. The existing protocol suites were overviewed and compared in the light of the memory, communication and computation costs. Considering the memory costs, protocols suites 2^d-cube, 2^d-octopus, and Asokan-Ginzboorg are most suitable, only the group key needs to be stored after the agreement. However, such protocols are only suitable for static groups.

The protocol suites CLIQUES, STR and TGDH are designed for dynamic groups since they include auxiliary protocols to handle dynamic events. The computation and communication costs in merge protocol of CLIQUES is not acceptable for ad hoc networks. While STR is especially suitable for networks where high network delays dominate, TGDH is especially computation protocols suites. However, in both protocol suites each member stores bkeys of all nodes except the root, and the whole tree with all bkeys are broadcasted while updating the bkeys. We have improved both protocols by storing only the node's bkey and bkeys of all nodes in its co-path. We have reduced the size of the broadcast message by including only required bkeys. The tree structure is only broadcasted if it is not known to all members. The improved version of STR is μSTR, of TGDH is μTGDH.

Although the communication and memory costs in μSTR and μTGDH are significantly reduced, the computation costs in μSTR is not acceptable for ad hoc networks with limited computation capability, and the communication costs of protocols setup, partition and merge in μTGDH are not optimized. Hence we have proposed TFAN, a group key agreement suitable for ad-hoc networks. It combines the communication efficiency of μSTR and computation efficiency of μTGDH. The resulting protocol provides an optimal solution for ad-hoc networks.

We have implemented the TFAN API with J2ME CDC. Provided group communication system the TFAN API can run in all systems supporting J2ME CDC, e.g. PDAs and laptops.

As future work, we suggested that a tree in TFAN may have various types of ms-trees, while till now only one type of ms-tree are allowed, either μSTR ms-tree or μTGDH ms-tree, and that each ms-tree has its own maximal height m.

A. Appendix

A.1. An example of TFAN group configuration file

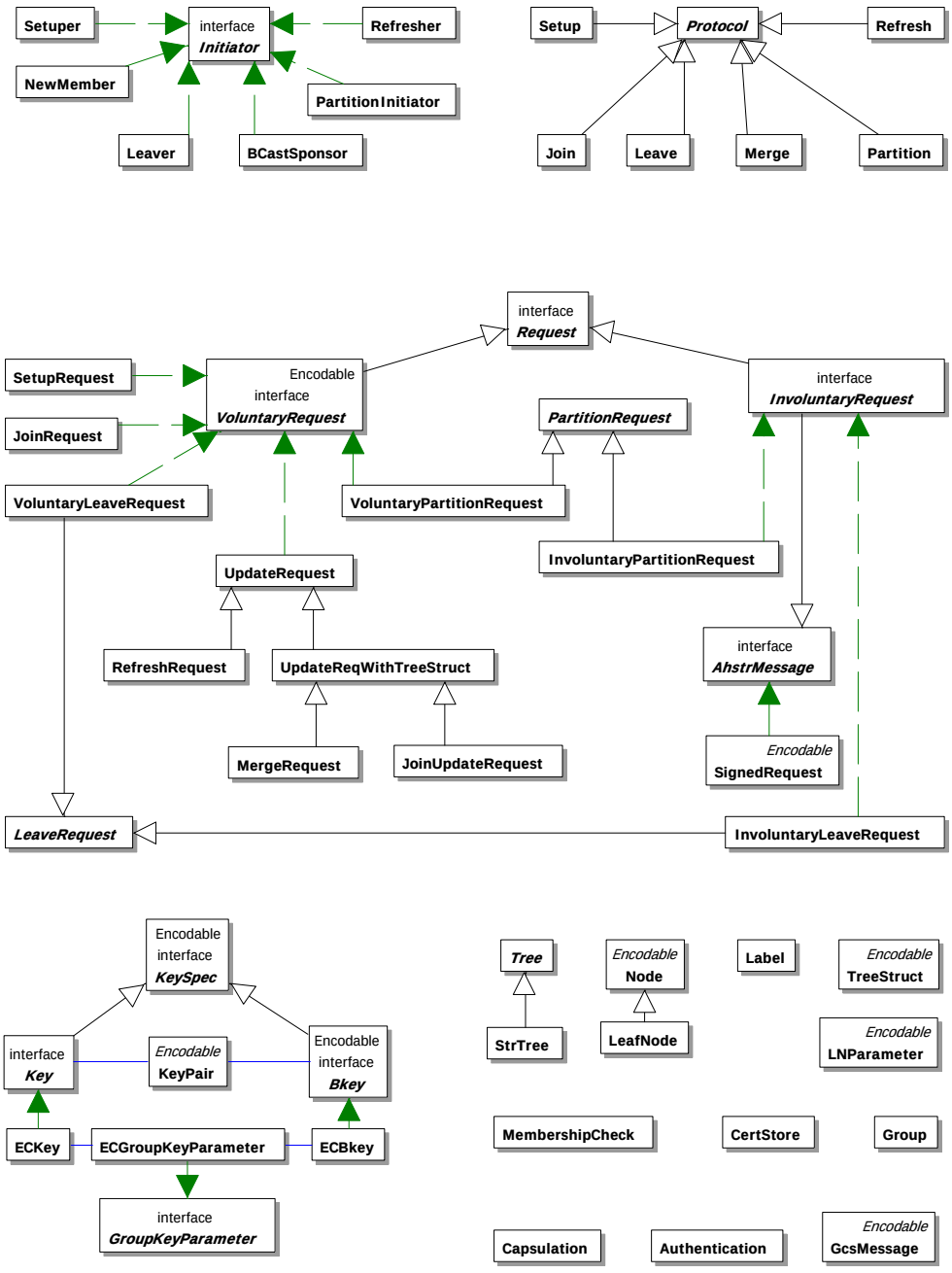
```
# Tree Setting: m is an non negative value, and art is either S
# or T, for invalid art is T insteadly used.
TreeSetting
-m 2
-art T

# Group name
GroupName TFAN-Test Group

# The multicast ip address, from 224.0.0.0 to 239.255.255.255.
GroupAddress   224.0.11.8

#Multicast port, from 1025 to 65525
GroupPort    21108

# Group key parameter.Tfan has defined some consant ec domain
# parameters defined in FIPS-186-2. To use such constant
# parameters, use the following Identifiers: P192, P224,
# P256, P384, P521.
GroupkeyParameter   P192

# Or define the ec domain parameters as follows:
# -key value, radix(default 10)
#GroupkeyParameter
#-a  -3
#-p  6277101735386680763835789423207666416083908700390324961279
#-n  6277101735386680763835789423176059013767194773182842284081
#-s  3045ae6fc8422f64ed579528d38120eae12196d5, 16
#-b  64210519e59c80e70fa7e9ab72243049feb8deecc146b9b1, 16
#-Gx 188da80eb03090f67cbf20eb43a18800f4ff0afd82ff1012, 16
#-Gy 07192b95ffc8da78631011ed6b24cdd573f977a11e794811, 16

# Worker controller settings, the time unit of refresh interval
# is minitus, and for all others is seconds.
WorkerControll
-automerge         true
-autosetup         true
-timeoutmerge      10
-timeoutsetup      1
-timeoutpartition  5
-refreshinterval   10
```

A.2. An example of TFAN member configuration file

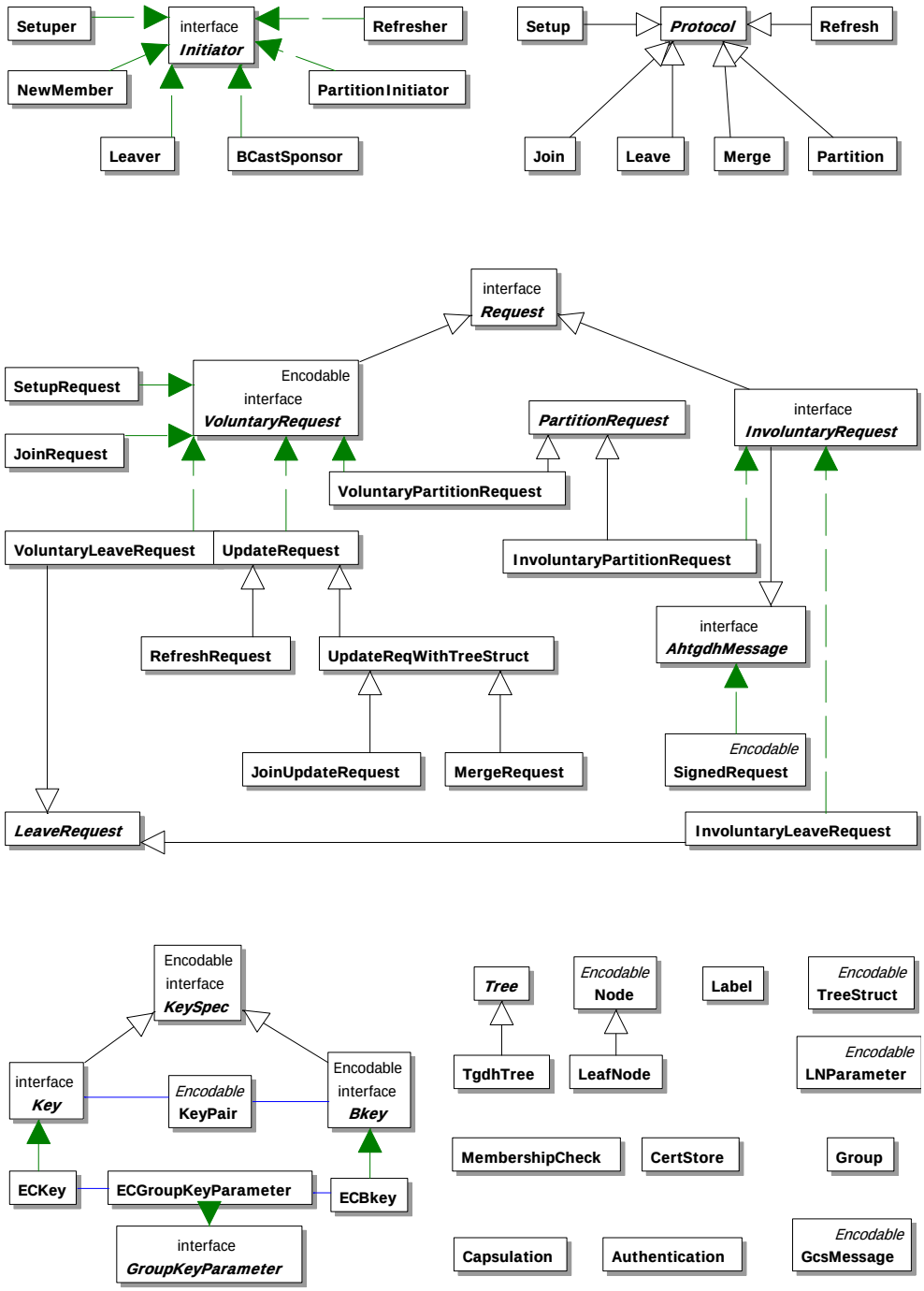
```
# localAddressAsID indicates wether to use the local address as
```

```
# the member id, If the value is true, omit the next line text,
# otherwise, the memberID must be present.
MemberID
-localAddressAsID  false
-MemberID M01

# The keystore that saves the PKCS#8 DER encoded private key
# the passwords to open the keystore and to get the private
# key should be entered manually, if needed.
PrivateKey
-ksfile keystore/keystore-M01.jks
-alias myprivkey

# A list of the file path saving ca-certificates
CACerts
-cacert cacert.pem.der

# The folder depoting the user certificates
CertFolder usercerts
```

A.3. Classes of μSTR API



A.4. Classes of μTGDH API



List of Theorems

List of Algorithms

List of Protocols

List of Figures

List of Tables

Bibliography

[1] ISO 8732: Banking - Key Management (Wholesale). 1988.

[2] OSI Directory - Part 8: Authentication Framework. 1994.

[3] Specifying and Using a Partitionable Group Communication Service. *ACM Trans. Comput. Syst.*, 19(2):171–216, 2001.

[4] S. Agarwal, A. Ahuja, J. P. Singh, and R. Shorey. Route-Lifetime Assessement-Based Routing (RABR) Protocol for Mobile Ad Hoc Networks. *Proceedings of IEEE ICC 2000*, 3:1697–1701, June 2000.

[5] M. Al-Shurman, S.-M. Yoo, and S. Park. Black Hole Attack in Mobile Ad Hoc Networks. In *ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference*, pages 96–97. ACM Press, 2004.

[6] American National Standards Institute. Working Draft ANSI X9.62-1998: Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA). 1998.

[7] Y. Amir. *Replication Using Group Communication over a Partitioned Network*. PhD thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1995.

[8] N. Asokan and P. Ginzboorg. Key Agreement in Ad-hoc Networks. *Communication Systems Laboratory, Nokia Research Center*, February 2001.

[9] G. Ateniese, M. Steiner, and G. Tsudik. Authenticated Group Key Agreement and Friends. In *CCS '98: Proceedings of the 5th ACM conference on Computer and communications security*, pages 17–26. ACM Press, 1998.

[10] B. Awerbuch, D. Holmer, and H. Rubens. An On-Demand Secure Routing Resilient to Byzantine Failures. *Proceedings of the ACM Workshop on Wireless Security 2002*, pages 21–30, September 2002.

[11] R. Baldwin and R. Rivest. RFC2040: The RC5, RC5-CBC, RC5-CBC-Pad, and RC5-CTS Algorithms. October 1996.

[12] R. K. Bauer, T. A. Berson, and R. J. Feiertag. A Key Distribution Protocol Using Event Markers. *ACM Trans. Comput. Syst.*, 1(3):249–255, 1983.

[13] K. Becker and U. Wille. Communication Complexity of Group Key Distribution. In *CCS '98: Proceedings of the 5th ACM conference on Computer and communications security*, pages 1–6. ACM Press, 1998.

[14] S. M. Bellovin and M. Merritt. Encrypted Key Exchange: Password-Based Protocols Secure Against Dictionary Attacks. In *SP '92: Proceedings of the 1992 IEEE Symposium on Security and Privacy*, page 72. IEEE Computer Society, 1992.

[15] L. Biesemeister and G. Hommel. Role-Based Multicast in Highly Mobile But Sparsely Connected Ad Hoc Networks. *Proceedings of ACM MOBIHOC 2000*, pages 43–50, August 2000.

[16] D. Bleichenbacher. Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS#1. *CRYPTO*, pages 1–12, 1998.

[17] M. Burmester and Y. Desmedt. A Secure and Efficient Conference Key Distribution System. *A. D. Santis, editor, Advances in Cryptology - EUROCRY'94, number 950 in Lecture Notes in Computer Science*, pages 275–286, 1995.

[18] H. Chan, A. Perrig, and D. Song. Random Key Predistribution Schemes for Sensor Networks. In *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*, page 197. IEEE Computer Society, 2003.

[19] T. W. Chen and M. Gerla. Global State Routing: A New Routing Scheme for Ad Hoc Wireless Networks. *Proceedings of IEEE ICC 1998*, pages 171–175, June 1998.

[20] C. C. Chiang, M. Gerla, and L. Zhang. Forwarding Group Multicasting Protocol for Multi-Hop, Mobile Wireless Networks. *ACM/Baltzer Journal of Cluster Computing: Special Issue on Mobile Computing*, 1(2):187–196, 1998.

[21] C. C. Chiang, H. K. Wu, W. Liu, and M. Gerla. Routing in Clusted Multi-Hop Mobile Networks with Fading Channel. *Proceedings of IEEE SICON 1997*, pages 197–211, April 1997.

[22] T. H. Clausen, G. Hansen, L. Christensen, and G. Behrmann. The Optimized Link State Routing Protocol, Evaluation Through Experiments and Simulation. *Proceedings of IEEE Symposium on Wireless Personal Mobile Communications 2001*, September 2001.

[23] W. M. Daley and R. G. Kammer. FIPS PUBS 46-3: Data Encryption Standard (DES). *U.S. DEPARTMENT OF COMMERCE/National Institute of Standards and Technology*, October 1999.

[24] S. K. Das, B. S. Manoj, and C. S. R. Murthy. A Dynamic Core-Based Multicast Routing Protocol for Ad Hoc Wireless Networks. *Proceedings of ACM MOBIHOC 2002*, pages 24–35, June 2002.

[25] S. K. Das, B. S. Manoj, and C. S. R. Murthy. Weight-Based Multicast Routing Protocol for Ad Hoc Wireless Networks. *Proceedings of IEEE GLOBECOM 2002*, 1:17–21, November 2002.

[26] D. E. Denning and G. M. Sacco. Timestamps in Key Distribution Protocols. *Commun. ACM*, 24(8):533–536, 1981.

[27] V. Devarapalli, A. A. Selcuk, and D. Sidhu. MZR: A Multicast Protocol for Mobile Ad Hoc Networks. *Internet draft (work in progress), draft-vijar-manet-mzr-01.txt*, July 2001.

[28] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, November 1976.

[29] W. Diffie, P. C. V. Oorschot, and M. J. Wiener. Authentication and Authenticated Key Exchanges. *Des. Codes Cryptography*, 2(2):107–125, 1992.

[30] H. Dobbertin, A. Bosselaers, and B. Preneel. RIPEMD-160: A Strengthened Version of RIPEMD. April 1996.

[31] W. Du, J. Deng, Y. S. Han, and P. K. Varshney. A Pairwise Key Pre-Distribution Scheme for Wireless Sensor Networks. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 42–51. ACM Press, 2003.

[32] R. Dube, C. D. Rais, K. Y. Wang, and T. S. K. Signal Stability-Based Adaptive Routing for Ad Hoc Mobile Networks. *IEEE Personal Communications Magazine*, pages 36–45, February 1997.

[33] T. ElGamal. A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. *CRYPTO*, pages 10–18, 1984.

[34] T. ElGamal. A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. *IEEE Transactions on Information Theory, v. IT-31, n. 4*, pages 469–472, 1985.

[35] FEAL-NX Specifications. *URL: http://info.isl.ntt.co.jp/feal-nx/.*

[36] For Mobile Ad. Distributed Symmetric Key Management.

[37] J. J. Garcia-Luna-Aceves and E. L. Madruga. The Core-Assisted Mesh Protocol. *IEEE Journal on Selected Areas in Communications*, 17(8):1380–1994, August 1999.

[38] J. J. Garcia-Luna-Aceves and M. Spohn. Source-Tree Routing in Wireless Networks. *Proceedings of IEEE ICNP 1999*, pages 273–282, oct 1999.

[39] O. Goldreich. *Foundations of Cryptography - Basic Tools*, pages 12–13. Cambridge University Press, 2001.

[40] Z. J. Haas. The Routing Algorithm for the Reconfigurable Wireless Networks. *Proceedings of ICUPC 1997*, 2:562–566, October 1997.

[41] M. Hietalahti. Efficient Key Agreement for Ad-hoc Networks. Master's thesis, Laborarory for Theoretical Computer Science, Department of Computer Science and Engineering, Helsinki University of Technology, 2001.

[42] Y. Hu, D. B. Johnson, and A. Perrig. SEAD: Secure Efficient Distance Vector Routing for Mobile Ad Hoc Networks. *Proceedings of IEEE WMCSA 2002*, pages 3–13, June 2002.

[43] Y. Hu, A. Perrig, and D. Johnson. Wormhole Detection in Wireless Ad Hoc Networks, June 2002.

[44] IEEE 802.11, The Working Group Setting the Standards for Wireless. *URL: http://grouper.ieee.org/groups/802/11/.*

[45] IETF. Public-Key Infrastructure (X.509) (PKIX). *URL: http://www.ietf.org/html.charters/pkix-charter.html*.

[46] I. Ingemarson, D. T. Tang, and C. K. Wong. A Conference Key Distribution System. *IEEE Transactions on Information Theory*, IT-28(5):714–720, September 1982.

[47] A. Iwata, C. C. Chiang, G. Pei, M. Gerla, and T. W. Chen. Scalable Routing Strategies for Ad Hoc Wireless Networks. *IEEE Jounrnal on Selected Areas in Communications*, 17(8):1369–1379, August 1999.

[48] Janson, Phil, G. Tsudik, and M. Yung. Scalability, Flexibility in Authentication Services: The KryptoKnight Approach. *IEEE INFOCOM'97. Tokyo, Japan*, April 1997.

[49] JGroups J2ME. JGroups J2ME. *URL: http://www.jgroups-me.org/.*

[50] L. Ji and M. S. Corson. Differential Destination Multicast (DDM) Specification. *Internet draft (work in progress), draft-ietf-manet-ddm-00.txt*, July 2000.

[51] M. Joa-Ng and I. T. Lu. A Peer-to-Peer Zone-Based Two-Level Link State Routing for Mobile Networks. *IEEE Journal on Selected Areas in Communications*, 17(8):1415–1425, August 1999.

[52] D. B. Johnson and D. A. Maltz. Dynamic Source Routing in Ad Hoc Wireless Networks. *Mobile Computing*, 353:153–181, 1996.

[53] B. Kaliski. RFC 1319: The MD2 Message-Digest Algorithm. *URL: http://www.ietf.org/rfc/rfc1319.txt*, April 1992.

[54] A. Kehne, J. Schönwälder, and H. Langendörfer. A Nonce-based Protocol for Multiple Authentications. *SIGOPS Oper. Syst. Rev.*, 26(4):84–89, 1992.

[55] A. Khalili and W. A. Arbaugh. Security of Wireless Ad-hoc Networks.

[56] Y. Kim, A. Perrig, and G. Tsudik. Simple and Fault-Tolerant Key Agreement for Dynamic Collaborative Groups. In *CCS '00: Proceedings of the 7th ACM Conference on Computer and Communications Security*, pages 235–244. ACM Press, 2000.

[57] Y. Kim, A. Perrig, and G. Tsudik. Communication-Efficient Group Key Agreement. *Information Systems Security, Proceedings of the 17th International Information Security Conference IFIP SEC'01*, 2001.

[58] Y. Kim, A. Perrig, and G. Tsudik. Tree-based Group Key Agreement. *ACM Trans. Inf. Syst. Secur.*, 7(1):60–96, 2004.

[59] Y. Ko and N. H. Vaidya. Location-Aided Routing (LAR) in Mobile Ad Hoc Networks. *Proceedings of ACM MOBICOM 1998*, pages 66–75, Oct 1998.

[60] Y. B. Ko and N. H. Vaidya. Geocasting in Mobile Ad Hoc Networks: Location-Based Multicast Algorithms. *Proceedings of IEEE WMCSA 1999*, pages 101–110, February 1999.

[61] N. Koblitz. Elliptic Curve Cryptosystems. *Mathematics of Computation*, 48:203–209, 1987.

[62] S. Lee and C. Kim. Neighbor Supporting Ad Hoc Multicast Routing Protocol. *Proceedings of ACM MOBIHOC 2000*, pages 37–50, August 2000.

[63] S. J. Lee, M. Gerla, and C. C. Chiang. On-Demand Multicast Routing Protocol. *Proceedings of IEEE WCNC 1999*, pages 1298–1302, September 1999.

[64] A. K. Lenstra and E. R. Verheul. Selecting Cryptographic Key Sizes. November 1999.

[65] M. Liu, R. Talpade, and A. McAuley. AMRoute: Adhoc Multicast Routing Protocol. *Internet draft (work in progress),draft-talpade-manet-amroute-00.txt*, August 1999.

[66] B. S. Manoj, R. Ananthapadmanabbha, and C. S. R. Murthy. Link Life-Based Routing Protocol for Ad Hoc Networks. *Proceedings of IEEE ICCCN 2001*, pages 573–576, October 2001.

[67] MediaCrypt. IDEA International Data Encryption Algorithm. *URL: http://www.mediacrypt.com/*.

[68] A. Medvinsky and M. Hur. RFC1510: The Kerberos Network Authentication Service (V5). September 1993.

[69] A. Medvinsky and M. Hur. RFC2712: Addition of Kerberos Cipher Suites to Transport Layer Security (TLS). October 1999.

[70] A. J. Menezes, P. C. V. Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, October 1996.

[71] V. Miller. Uses of Elliptic Curves in Cryptography. *In H. C. WIlliams, editor, Advances in Cryptology – CRYPTO '85*, LNCS 218:417–426, 1986.

[72] Molva, Refik, G. Tsudik, E. V. Herreweghen, and S. Zatti. KryptoKnight Authentication and Key Distribution System. *Lecture Notes in Computer Science*, 648:155–174, November 1992.

[73] L. Moser, Y. Amir, P. Melliar-Smith, and D. Agarwal. Extended Virtual Synchrony. *ICDCS'94*, pages 56–65, June 1994.

[74] C. R. Murthy and B. S. Manoj. *Ad Hoc Wireless Networks - Architectures and Protocols*, pages 299–364. Person Education, 2004.

[75] C. R. Murthy and B. S. Manoj. *Ad Hoc Wireless Networks - Architectures and Protocols*, pages 365–450. Person Education, 2004.

[76] S. Murthy and J. J. Garia-Luna-Aceves. An Efficient Routing Protocol for Wireless Networks. *ACM Mobile Networks and Applications Journal, Special Issue on Routing in Mobile Communication Networks*, 1(2):183–197, Oct 1996.

[77] National Institute of Standards and Technology. FIPS PUBS 186: Digital Signature Standard (DSS). May 1994.

[78] National Institute of Standards and Technology. FIPS PUBS 180-1: Secure Hash Standard. April 1995.

[79] National Institute of Standards and Technology. FIPS PUBS 186-2: Digital Signature Standard (DSS). Jan 2000.

[80] National Institute of Standards and Technology. FIPS PUBS 197: Specification for the Advanced Encryption Standard (AES). November 2001.

[81] National Institute of Standards and Technology. FIPS PUBS 180-2: Secure Hash Standard. August 2002.

[82] National Institute of Standards and Technology (NIST). NIST PKI program.

[83] National Institute of Standards and Technology (NIST). Wireless Ad Hoc Network Projects. *URL: http://w3.antd.nist.gov/wahn_home.shtml*.

[84] R. M. Needham and M. D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Commun. ACM*, 21(12):993–999, 1978.

[85] R. M. Needham and M. D. Schroeder. Authentication Revisited. *SIGOPS Oper. Syst. Rev.*, 21(1):7–7, 1987.

[86] B. C. Neuman and S. G. Stubblebine. A Note on the Use of Timestamps as Nonces. *SIGOPS Oper. Syst. Rev.*, 27(2):10–14, 1993.

[87] K. Nyberg and R. Rueppel. A New Signature Scheme Based on the DSA Giving Message Recovery. *1st ACMConference on Compute and Communications Security*, pages 58–61, 1993.

[88] D. Otway and O. Rees. Efficient and Timely Mutual Authentication. *SIGOPS Oper. Syst. Rev.*, 21(1):8–10, 1987.

[89] T. Ozaki, J. Kim, and T. Suda. Bandwidth Efficient Multicast Routing Protocol for Ad hoc Networks. *Proceeding of IEEE ICCCN 1999*, pages 10–17, October 1999.

[90] C. E. Perkins and P. Bhagwat. Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers. *Proceedings of ACM SIGCOMM 1994*, pages 234–244, August 1994.

[91] C. E. Perkins and E. M. Royer. Ad Hoc On-Demand Distance Vector Routing. *Proceedings of IEEE Workshop on Mobile Computing Systems and Applications 1999*, pages 90–100, February 1999.

[92] B. Preneel. *Analysis and Design of Cryptographic Hash Functions*. PhD thesis, Katholieke University Leuven, 1993.

[93] M. O. Rabin. Digitalized Signatures and Public-Key Functions as Intractable as Factorization. Technical report, 1979.

[94] R. Rivest. The MD4 Message Digest Algorithm. *Advances in Cryptology - Crypto '90*, pages 303–311, 1991.

[95] R. Rivest. RFC 1320: The MD4 Message-Digest Algorithm. *Network Working Group*, 1992.

[96] R. Rivest. RFC 1321: The MD5 Message-Digest Algorithm. *Internet Activities Board*, 1992.

[97] R. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[98] E. M. Royer and C. E. Perkins. Multicast Operation of he Ad Hoc On-Demand Distance Vector Routing Protocol. *Proceedings of ACM MOBICOM 1999*, pages 207–218, August 1999.

[99] RSA Laboratories. PKCS#1 v2.1: RSA Cryptography Standard. June 2002.

[100] K. Sanzgiri, B. Dahill, B. N. Levine, C. Shields, and E. M. B. Royer. A Secure Routing Protocol for Ad Hoc Networks. *Proceedings of IEEE ICNP 2002*, pages 78–87, November 2002.

[101] L. SCHENAUER and V. D. GLIGOR. A Key-Management Scheme for Distributed Sensor Networks. pages 41–47, 2002.

[102] S. Singh, M. Woo, and C. S. Raghavendra. Power-Aware Routing in Mobile Ad Hoc Networks. *Proceedings of IEEE WCNC 2000*, pages 1220–1225, September 2000.

[103] P. Sinha, R. Sivakumar, and V. Bharghavan. CEDAR: A Core Extraction Distributed Ad Hoc Routing Algorithm. *IEEE Journal on Selected Areas in COmmunications*, 17(8):1454–1466, August 1999.

[104] P. Sinha, R. Sivakumar, and V. Bharghavan. MCEDAR: Multicast Core Extraction Distributed Ad-Hoc Routing. *Proceedings of IEEE WCNC 1999*, pages 1313–1317, September 1999.

[105] R. S. Sisodia, I. Karthigeyan, B. Manoj, and C. S. R. Murthy. A Preferred Link-Based Multicast Protocol for Wireless Mobile Ad Hoc Networks. *Proceedings of IEEE ICC 2003*, 3:2213–2217, May 2003.

[106] R. S. Sisodia, B. S. Manoj, and C. S. R. Murthy. A Preferred Link-Based Routing Protocol for Ad Hoc Wireless Networks. *Journal of Communications and Networks*, 4(1):14–21, March 2002.

[107] D. G. Steer, L. Strawczynski, W. Diffie, and M. Wiener. A Secure Audio Teleconference System. In *CRYPTO '88: Proceedings on Advances in cryptology*, pages 520–528. Springer-Verlag New York, Inc., 1990.

[108] M. Steiner. *Secure Group Key Agreement*. PhD thesis, Naturwissenschaftlich-Technischen Fakultät I der Universität des Saarlandes, December 2001.

[109] M. Steiner, G. Tsudik, and M. Waidner. CLIQUES: A New Approach to Group Key Agreement. pages 380–387. IEEE Computer Society Press, May 1998.

[110] M. Steiner, G. Tsudik, and M. Waidner. Key Agreement in Dynamic Peer Groups. *IEEE Transactions on Parallel and Distributed Systems*, August 2000.

[111] M. Steiner, G. Tsudik, and M. Waidner. Key Agreement in Dynamic Peer Groups. *IEEE Trans. Parallel Distrib. Syst.*, 11(8):769–780, 2000.

[112] W. Su and M. Gerla. IPv6 Flow Handoff in Ad Hoc Wireless Networks Using Mobility Prediction. *Proceeding of IEEE GLOBECOM 1999*, pages 271–275, December 1999.

[113] SUN Microsystem. Connected Device Configuration (CDC) of J2ME; JSR 36, JSR 218. *URL: http://java.sun.com/products/cdc/index.jsp*.

[114] SUN Microsystem. Foundation Profile (FP) of J2ME CDC; JSR 46. *URL: http://java.sun.com/products/foundation/index.jsp*.

[115] SUN Microsystem. Java 2 Platform, Micro Edition (J2ME). *URL: http://java.sun.com/j2me/index.jsp*.

[116] The JGroups Project. JGroups - A Toolkit for Reliable Multicast Communication. *URL: http://www.jgroups.org/*.

[117] The Legion of the Bouncy Castle. *URL: http://www.bouncycastle.org/*.

[118] The Official BlueTooth ®Webseite. *URL: http://www.bluetooth.com/*.

[119] The Weizmann Institute Relation Locator (TWIRL). The TWIRL integer factorization device. *URL: http://www.wisdom.weizmann.ac.il/ tromer/twirl/*.

[120] C. K. Toh. Associativity-Based ROuting for Ad Hoc Mobile Networks. *Wireless Personal Communications*, 4(2):1–36, March 1997.

[121] C. K. Toh, G. Guichala, and S. Bunchua. ABAM: On-Demand Associativity - Based Multicast Routing for Ad Hoc Mobile Networks. *Proceedings of IEEE VTC 2000*, pages 987–993, September 2000.

[122] W.-G. Tzeng and Z.-J. Tzeng. Round-Efficient Conference-Key Agreement Protocols With Provable Security. *Advances in Cryptology - ASIACRYPT'2000, Lecture Notes in Computer Science, Kyoto, Japan*, December 2000.

[123] X. Wang, Y. L. Yin, and H. Yu. Finding Collisions in the Full SHA1. *CRYPTO 2005*.

[124] C. K. Wong, M. Gouda, and S. S. Lam. Secure Group Communications Using Key Graphs. *IEEE/ACM Trans. Netw.*, 8(1):16–30, 2000.

[125] C. W. Wu, Y. C. Tay, and C. K. Toh. Ad Hoc Multicast Routing Protocol Utilizing Increasing id-numberS (AMRIS) Functional Specification. *Internet draft (work in progress), draft-ietf-manet-amris-spec-00.txt*, November 1998.

[126] W.-H. Yang and S.-P. Shieh. Secure key Agreement for Group Communications. *Int. J. Netw. Manag.*, 11(6):365–374, 2001.

[127] S. Yi and P. Kravets. Security-Aware Ad Hoc Routing for Wireless Networks. *Proceedings of ACM MOBIHOC 2001*, pages 299–302, October 2001.

[128] S. Yi and R. Kravets. Key Management for Heterogeneous Ad Hoc Wireless Networks. 2002 2004.

[129] H. Zhou and S. Singh. Content-Based Multicast (CBM) in Ad Hoc Networks. *Proceedings of ACM MOBIHOC 2000*, pages 51–60, August 2000.