Scalable and Efficient Key Agreement Protocols for Secure Multicast Communication in MANETs

by Maria Striki, John S. Baras

CSHCN TR 2002-28
(ISR TR 2002-62)

Scalable and Efficient Key Agreement Protocols for Secure Multicast Communication in MANETs

Maria Striki, John S. Baras

Abstract--**In this paper protocols for group key distribution are compared and evaluated from the point of view of Mobile Ad Hoc Networks (MANETs). A MANET is a collection of wireless mobile nodes, communicating among themselves over possibly multihop paths, without the help of any infrastructure such as base stations or access points. So the need to render those networks as autonomous and secure as possible, since no central authorization can be assumed at all times, becomes emergent. Key management is the service that ensures the security of communication among nodes, and the capability of their cooperation as a secure group. It consists of three important services: key generation, user authentication and key distribution. In this work we assume that the participating users have already been authenticated with some mechanism, and we are focused only on studying and comparing protocols for group key establishment in MANETs. We are addressing the issues of Fault-Tolerance and Efficiency for key distribution protocols for group communications in MANETs. Most key distribution protocols that exist today are primarily designed for wire-line networks. These protocols either fail to work as intended or cannot work at all when they are applied to the demanding environment of MANETs. The main reasons for this are: frequent node failures, network partitions and merges, inefficient computational and communication capabilities of certain wireless nodes, network delay, bad quality of signal etc. We determine the framework under which protocols can efficiently work in MANETs, design new protocols or modify existing ones, so that they can be robust, scalable and applicable in this environment. We classify these protocols in two families, contributory and non-contributory. We evaluate them from the point of view of MANETs and compare their performance.**

Index Terms--**Mobile Ad Hoc Network, contributory protocol, Diffie-Hellman Protocol, Octopus Protocol**

I. INTRODUCTION

AS the development of wireless multicast services such as cable TV, secure audio and conferencing, visual broadcasts, military command and control grows, the research on security for wireless multicasting becomes more and more important. Information should be communicated to the appropriate groups of nodes with the utmost security and with respect to the network constraints at the same time. It is essential to develop a secure and robust key management scheme for multicast communications. Key management determines the security, scalability and efficiency of the network. In this work, we study and develop key distribution techniques for wireless environments with unreliable channels, where the topology is changing fast. The nodes of the network may have limited capacity, computational and transmission power (may vary from Satellites, laptops, to PDAs and cell-phones).

Mobile ad-hoc networks are dynamic with no pre-existing infrastructure. Furthermore, in wireless mobile networks high mobility may result in nodes frequently going out of range or running out of battery, leading in temporary links. Collisions, low link quality, distance between nodes and various other factors result in unreliable links or excessive delay in the network. This kind of network is vulnerable to jamming the radio channel, to modifying communication among legitimate participants, to inserting messages and burdening the network traffic. A node may loose its connection to another node(s) because it might either move out of reach or run out of battery resources, or can be compromised. We cannot always assume that a node within the group that has direct connections to all other participant nodes and can perform broadcasts to the whole group exists. A broadcast from a node to its nearby neighbors only seems more viable. Most of the current key distribution protocols are designed for wire-line networks that are free from most of the constraints of MANETs. Furthermore, the computational power of nodes is considered an issue for some wireless mobile nodes due to resources or capacity limitations. Thus, key distribution protocols that are robust enough to survive or tolerate frequent node failures, network partitions and merges, delays in critical messages, ambiguity to determine the state of group members under certain circumstances, extensive computations etc., are needed. In MANETs, we cannot always guarantee the existence of a node with direct connections to all other participants that can broadcast to the whole group. Also, a change in the topology of a group might occur while the group key is being calculated. In some protocols this event may cause enormous overhead, as the operation of calculating the group key must start over. These constraints render a lot of group key distribution protocols inefficient in an environment that requires quick operations and with the lowest possible complexity, to catch up with the rapidly changing topology of the network. We classify the existing protocols in two categories: *contributory* protocols in which all participants take equally part in the key generation and guarantee for their part that the resulting key is fresh, and *non-contributory* ones where group key generation does not require equal participation from all members.

Our objectives in this paper are to study the properties of these two families of protocols, their pros and cons in the view of the MANET environment and evaluate the performance of four protocols representing either family. We selected the One-Way Function Tree (OFT) protocol from the family of non-contributory ones, and 2^d-Octopus protocol with our two modified versions of it, to represent the family of contributory ones. The original OFT assumes a fixed group leader with considerable processing capabilities and therefore may not be fault-tolerant or scalable in the environment of MANETs. On the other hand, it is considered among the most efficient key distribution protocols. By comparing the fault-tolerant Octopus-based protocols to OFT, we gain an insight about the extra overhead required to render key distribution protocols robust, scalable and applicable in MANETs. We derive the cost functions for each protocol in terms of total communication, computation and storage in each node of the group. Furthermore, in both the original 2^d-Octopus protocol and the modified ones we describe how the join and leave procedures occur since they are not described in the original paper.

Some of the most important aspects of *Fault Tolerance* for key distribution protocols we are discussing are: the issue of a single, non-flexible, “omnipotent” group leader that may constitute a single point of failure, the issue of whether protocols can recover from members’ failure during the group key establishment without starting this very costly procedure all over again, and the issue of whether protocols tolerate frequent node failures, group partitions and merges at any time during a session.

Most **non-contributory** protocols are based on a fixed trusted central controller to distribute the key. Finding members within the group able to replace the faulty leader is not enough. The new leader should securely and quickly obtain all the information gathered by the previous leader up to that point which is not an easy task. It would be preferred that the “leader” is selected among group members (*as in contributory protocols*) and have a rather coordinating role, storing the less information possible that can be easily retrieved by any member becoming leader in the future also (*as in TGDH*). Furthermore, in order to reduce group partitions and frequent leader elections, we must take into account the mobility of nodes in the network, the robustness, the computational and processing capabilities of individual nodes. One solution is to dynamically select a node as group leader according to a certain policy that makes sense in a MANET (e.g. select the node that stays connected with the largest number of nodes within its group for the largest amount of time), and to make every such leader operate in a rather restricted area of the network. Therefore, we also require that the procedure of leader election be dynamic and flexible.

In most non-contributory protocols (tree-based), in the event of a node failure, a new group key is computed by updating only a restricted number of keys. The contributions of members for the key establishment are independent and need not follow a strict ordering. In the event of a node failure or delay to respond, the rest of the nodes proceed normally to the key establishment process. In a **contributory** protocol like GDH.2, each member is expected to contribute its portion of the key in a defined slot according to strict ordering. If a node does not respond during the given slot, the whole procedure comes to a standstill as all further actions of members depend on the contribution of the "disappeared" member and we cannot always determine on time if the response of the node is simply delayed or lost, or if the node itself is down or out of reach. Inevitably the key establishment process starts all over again.

However, contributory protocols still acquire some very important properties: they are most appropriate when no previously agreed common secrets among nodes exist, they reflect the totally distributed nature of a group, and their nature is such that no node constitutes a single point of failure. It would be desirable to derive a hybrid protocol that is fault tolerant in MANETs, efficient, and combines the main advantages of the two families of protocols.

We claim that MO and particularly MOT satisfy these requirements. We prove the fault tolerance of Octopus based protocols by analyzing in detail scenarios of failures most likely to occur in MANETs. We discuss the modifications we made to the original 2^d-Octopus. We then show how these modifications that lead to the new protocols MO and MOT, improve the fault-tolerance, the scalability and efficiency of the original 2^d-Octopus in MANETs.

II. Previous Work

Becker and Wille [1] derived lower bounds for contributory key distribution systems from the results of the gossip problem and demonstrated that those bounds are realistic for Diffie-Hellman (DH) based protocols. They used the basic DH distribution extended to groups from the work of Steiner, Tsudik and Waidner [2]. The GDH.2 protocol from this work in particular, acquires some very alluring properties: provides authentication without significant overhead, and minimizes the number of total exchanges. TGDH by Kim, Perrig and Tsudik [13], is a new hybrid, simple and efficient protocol that blends binary key trees with DH key exchange. Becker and Wille [1], introduced the Hypercube protocol as one requiring the minimum number of rounds. In [5], Asokan and Ginzboorg added to the 2^d-cube protocol ways to recover from node failures and password authenticated key exchange. This was very important since most contributory protocols are not designed to recover from node failures. Becker and Wille again, introduced Octopus protocol that required minimum number of messages and they derived the 2^d-Octopus protocol that combined Octopus with Hypercube to a very efficient protocol that worked for an arbitrary number of nodes.

From the non-contributory protocols family, the proposed solutions are based on a simple key distribution center. The simplest is Group Key Management Protocol (GKMP) in which a group manager shares a secret key with each member and uses that key to communicate the secret group key to that member [9]. The Logical Tree Hierarchy method (LKH) [8], creates a hierarchy of keys, each group member is secretly given one of the keys at the bottom of the hierarchy and can decrypt the keys along its path from the leaf to the root. Evolutions of the latter were: Efficient Large-Group Key Distribution (ELK) [12], designed rather for a stationary network and (OFT) [7], that minimized the number of bits broadcast to members after a membership change. The number of keys broadcast to the group in this case, and the computational efforts of the members are logarithmic in the number of members. This very efficient protocol was selected to represent this family of protocols.

III. Security Properties and Constraints in MANETs

The following very important properties have already been proven for the protocols we are discussing.

Forward (Backward) Secrecy: a passive adversary who knows a contiguous subset of group keys cannot discover subsequent (preceding) group keys.
Group Key Secrecy: it is computationally infeasible for a passive adversary to discover any group key.
Key Independence: a passive adversary who knows a proper subset of group keys cannot discover any other group key.

In the rest of this section we discuss the extra security constraints imposed by the MANETs environment. In the work of Kim et al. in [13], blending binary key trees with DH key exchange, results in a secure, simple, fault-tolerant protocol Tree Group Diffie-Hellman (TGDH), without central controller in the group. Any trusted member should be ready to become "sponsor" and assume duties of a central controller for a particular member join/leave operation. If this idea can be extended to MANET environment, then no single point of failure will exist -a weak point in non-contributory protocols.

In non-contributory protocols no external central controller is necessary to exist. A single node in the group that acquires certain computational, storage and communicational capabilities and has less probability to exit the group (voluntary or not) could assume the role of a group leader, in the same sense that there is a leader in contributory protocols like GDH.2. In the latter, a particular member is responsible for the broadcast in round n. So, even in these protocols there exist nodes with some extra responsibilities. The most important constraint for MANETs is that within a group there have to exist one or more nodes that acquire sufficient capabilities to become group leaders and also be robust in the sense that they don’t exit the group as frequently as other members in the group (connectivity loss, run out of battery resources etc). Usually, in a wire-line network we can assume that any node can act as group leader, but in a wireless ad-hoc network this is not always possible due to node limitations. In [13], the authors describe in detail how they handle “cascaded events” (one membership change occurs while another is being handled) which is the most usual case in a MANET. So, if constraints for communication and computational capabilities of nodes are met for non-contributory protocols or even TGDH-like protocols, we could make them robust in MANETs, given that the leader election policy is efficient as well.

We have just described the limitations of each family of protocols, and the framework that makes each one efficient. In a MANET, where no trusted third parties are assumed, if we select a non-contributory protocol for key distribution we have to consider the cost for selecting a node as the group leader and for establishing secure initial keys between this node and each member of the group. Next, we derive the computational and communication costs for OFT, for 2^d–Octopus protocol and its modifications (GDH.2-based Octopus or TGDH-based Octopus), and we estimate their relative performance.

IV. Secure Group Key Agreements and Our Extensions.

We briefly present the **GDH.2**, **OFT**, **TGDH**, and **2^d-Octopus** protocols because they are documented in detail in most of our references and also there is a limitation in pages.

GDH.2

In GDH.2 the up-flow stage is used to collect contributions from all group members. Every member M_i has to compose i intermediate values, each with *i*-1 exponents and one value with *i* exponents, and send them to M_{i+1}. So the formula of the up-flow stage is:
$M_i \rightarrow \{a^{\Pi\{N_K | k\in[1,i] \wedge k\neq j\}} \mid j \in [1,i]\}, a^{N_1 * \ldots * N_i} \rightarrow M_{i+1}$.
Member M_n is the first one to compute the group key. In the second stage M_n broadcasts the intermediate values to all group members. The formula of the broadcast stage is: $M_i \leftarrow \{a^{\Pi\{N_K | k\in[1,n] \wedge k\neq i\}} \mid i \in [1,n]\} \leftarrow M_n$.

Rounds	Messages	Combined message size	Exponentiations per M_i	Total exponentiations
N	*n*	(*n*-1)(*n*/2 +2) -1	(*i*+1), for *i*<*n*, for M_n	(*n*+3)*n*/2 -1

Table 1: cost of GDH.2 parameters and operations

So, in GDH.2 the highest-indexed group member M_n plays a special role by having to broadcast the last round of intermediate values. GDH.2 achieves low number of protocol rounds, and is among the only contributory protocols that provide authenticated group key agreement, support dynamic membership events, and handle partitions and merges.

Evaluation of the Cost for the GDH.2 protocol

The cyclic group G that provides the generator α is of order q and the length of the group key is *K* in bits. Thus, if α^{N_i} mod *q* is *K* bits in length, so is $\alpha^{N_1 \ldots N_n}$ mod *q*.

Initial communication: It requires n rounds (*n*-1 messages from member M_i to member M_{i+1} of length (*i*+1) K bits, one multicast message of *n K* bits from member M_n to the rest *n*-1 members). The total message length in average in bits is: $\sum_{i=1}^{n-1} (i+1)K + nK = (n^2 + 3n - 2)K/2$

Member initial computation: Member M_i does (*i*+1) exponentiations and member M_n does n exponentiations. In average a member does *n*/2 exponentiations, and does one more when it gets the stream message for the group key.

GSC addition computation: Member M_n generates a new exponent and sends an up-flow message to member M_{n+1} that computes the new key and does *n*+1 exponentiations (*n*+1 intermediate values). The simple member gets the broadcast stream and only needs to do one exponentiation. The GSC deletion computation requires *n*-1 exponentiations.

Add/Delete communication: Member M_n sends an up-flow message of (*n*+1) *K* length in bits to member M_{n+1}. Then member M_{n+1} broadcasts to all *n* members its *n*+1 intermediate values. So the total communication in the case of member addition is 2(*n*+1). In the case of member deletion, member M_n only broadcasts a message of length *n*-1 to the rest of the members.

Octopus Protocol

Octopus protocol uses DH key computed in one round as a random input for the subsequent round. It is further assumed that there is a bijection from generator G into the field Z_q from which the participants choose their random secrets. Four parties *A, B, C, D* generate a group key using only four exchanges. First, parties *A* and *B*, then *C* and *D* perform a DH key exchange generating keys α^{ab} and α^{cd}, respectively. Then, *A* and *C* as well as *B* and *D* do a DH key exchange using as secret values the keys generated in the first step. *A*(*B*) sends $\alpha^{\phi(\alpha^{ab})}$ to *C*(*D*) while *C*(*D*) sends $\alpha^{\phi(\alpha^{cd})}$ to *A*(*B*) so that *A* and *C* (*B* and *D*) can generate the joint key $\alpha^{\phi(\alpha^{cd})\phi(\alpha^{ab})}$. Participants P_1, P_2,…, P_n generate a common group key by first dividing themselves into five groups. Four participants P_{n-3}, P_{n-2}, P_{n-1}, P_n take charge of the central control, denoted as *A, B, C, D* respectively. The remaining parties distribute themselves into four groups: $\{P_i \mid i \in I_A\}$, $\{P_i \mid i \in I_B\}$, $\{P_i \mid i \in I_C\}$, $\{P_i \mid i \in I_D\}$, where I_A, I_B, I_C, I_D are pair-wise disjoint, possibly of equal size, and $I_A \cup I_B \cup I_C \cup I_D = \{1,\dots, n-4\}$. Now P_1,…, P_n generate a group key as follows:

1. $\forall\ X \in \{A, B, C, D\}, \forall i \in I_X$, X generates a joint key k_i with P_i via the DH key exchange.
2. The participants *A, B, C, D* do the 4-party key exchange described above using the values: $a=K(I_A)$, $b=K(I_B)$, $c=K(I_C)$, $d=K(I_D)$, where $K(J) := \prod_{i\in J} \phi(k_i)$ for $J \subseteq \{1,\dots, n-4\}$. Thereafter, *A, B, C, D* hold the joint and later group key K= $a^{\phi(a^{K(I_A\cup I_B)})\phi(a^{K(I_C\cup I_D)})}$.
3. The step is described only for A. Parties *B, C, D* act correspondingly. $\forall j \in I_A$, A sends the following two values to P_j: $a^{K(I_B\cup I_A\setminus\{j\})}$ and $a^{\phi(a^{K(I_C\cup I_D)})}$. P_j is able to generate K now; first P_j calculates $(a^{K(I_B\cup I_A\setminus\{j\}})^{\phi(k_j)} = a^{K(I_A\cup I_B)}$ and then K= $a^{\phi(a^{K(I_C\cup I_D)})\phi(a^{K(I_A\cup I_B)})}$.

This protocol requires *n*-4 exchanges to generate the DH keys k_i, 4 exchanges for the key agreement between *A, B, C, D* and finally *n*-4 messages to be sent from *A, B, C, D* to P_1, P_2,…, P_{n-4}. Hence it performs a minimum number of 2*n*-4 exchanges.

protocol	messages	exchanges	simple rounds	Synchr. rounds	broadcasts
Octopus	3*n*-4	2*n*-4	$2\left\lceil {n-2^d}/{2^d} \right\rceil \left\lceil \frac{n-4}{4} \right\rceil +2$	4	-

Table 2: cost of parameters and operations in Octopus protocol

Hypercube Protocol

It minimizes the number of simple rounds. In general 2^d parties can agree upon a key within d simple rounds by performing DH key exchanges on the edges of a d-dimensional cube. For 2^d participants, we identify them on the d-dimensional space $GF(2)^d$ and choose a basis $b_1,\dots, b_d$ of $GF(2)^d$. Now the protocol may be performed in two rounds as follows:

1. In the first round, every participant $v \in GF(2)^d$ generates a random number r_v and performs a DH key exchange with participant $v+b_1$ using the values r_v and r_{v+b1}, respectively.

2. In the *i*-th round, every participant $v \in GF(2)^d$ performs a DH key exchange with participant $v+b_i$, where both parties use the value generated in round *i*-1 as the secret value for the key exchange.

In every round, the participants communicate on a maximum number of parallel edges of the *d*-dimensional cube (round *i*, direction b_i). Thus every party is involved in exactly one DH exchange per round. Furthermore, all parties share a common key at the end of this protocol because the vectors $b_1, \ldots, b_d$, form a basis of the vector space $GF(2)^d$.

2^d-Octopus Protocol

For an arbitrary number of participants (<>2^d), that requires a low number of simple rounds, the idea of the octopus protocol can be adopted again. In the 2^d–Octopus protocol the participants act as in the simple Octopus protocol. However, 2^d instead of four parties are distinguished to take charge of the central control, whereas the remaining n-2^d parties divide into 2^d groups. In other words, in steps 1 and 3 of the Octopus protocol, 2^d participants manage communication with the rest and in step 2 these 2^d parties perform the cube protocol for 2^d participants. In general, we obtain the following complexities for the protocols described above.

Protocol	messages	Exchanges	Simple rounds	Syn. Rounds	broadcast
2^d – cube	N d	n d/2	d	D	-
2^d – octopus	$3(n-2^d) + 2^d$ d	$2(n-2^d) + 2^{d-1}$ d	$2\left\lceil \frac{n-2^d}{2^d} \right\rceil$+d	2+d	-

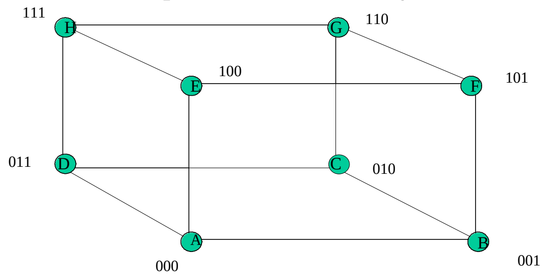
Table 3: Cost of parameters in 2^d-Octopus protocol

The 2^d–Octopus protocol provides a tradeoff between the total number of messages/exchanges needed and the number of rounds. For d=2 (Octopus) the number of exchanges is optimal, whereas the number of simple rounds is comparatively high. On the other hand, if d satisfies $2^{d-1} < n < 2^d$, the number of simple rounds required is very low and the total number of messages is high. The 2^d–Octopus protocols provides a class of key distribution systems without broadcasting which matches the lower bound ($\log_2 n$) for the total number of simple rounds if n is a power of 2. If not, the protocols require about $\log_2 n$+1 simple rounds.

Fault Tolerance Issue for 2^d- Octopus protocol

In [5] the authors claim that this scheme is fault tolerant but they don't analyze all the group disruption cases that this key distribution scheme can come across in a MANET network. They only refer to the case of one or more nodes of a group disappearing from the group (become faulty) during the formulation of the group key. Here, we make an attempt to look at all the possible scenarios and decide if this protocol is or can be made to be fault tolerant.

To this end we are going to use an example to make the study of all cases easier.



Description of Group Key formulation:
Each node is assigned to a vertex in the hypercube and has a unique *d*-bit address drawn from Z_n as in the example. The protocol takes *d* rounds.
Assume a node with address *i*. In the jth round this node performs a two-party DH with the node whose address is $i \oplus 2^{j-1}$. Thus in the jth round neighbors along the jth dimension of the hypercube participate in a two-party DH protocol run. After *d* such rounds, all players will have the same key.
1st Round: {000-001, 010-011, 110-111, 100-101}
2nd Round:{000-010, 001-011, 100-110, 111-101}
3d Round: {000-100, 011-111, 001-101, 110-010}

In the 1st round the DH keys derived are: a^{AB}, a^{CD}, a^{EF}, a^{GH} .
In the 2nd round the DH keys derived are: $a^{a^{AB}a^{CD}}$, $a^{a^{AB}a^{CD}}$, $a^{a^{EF}a^{GH}}$, $a^{a^{EF}a^{GH}}$
In the 3d round the DH keys derived are: $a^{a^{a^{AB}a^{CD}}a^{a^{EF}a^{GH}}}$ for all four pairs.

Possible Scenarios:

1. *Node A {000} becomes faulty immediately before the first round and does not re-appear even after the group key has been derived.* The following occur: Node *B* does not exchange DH keys with any node during the 1st round, it uses a secret of its own, assume *BB*. In the 2nd round, all nodes that need to communicate with *A* in the hypercube result in communicating with its pair-mate of the 1st round, namely *B*. This idea applies for all three rounds.

2. *Two nodes from the same pair become faulty before the first round begins.* Another pair of the hypercube "logically splits" to fill the gap of the disappeared pair: one substitutes the pair that became faulty so that this scenario resembles the previous one. Now however, two nodes instead of one undertake the duties of two pairs of nodes.

3. *A participating node A goes down during the 1st round and does not re-appear during the group key establishment.* As in scenario 1, its pair-mate *B* takes over on behalf of *A* as well, *B* has already computed the common DH key with *A:* a^{AB}. *B* creates a new secret share *BB* and computes a^{BB}. *B* takes over from that point exactly as described in scenario 1 but it communicates both a^{AB} and a^{BB} blinded values to its future pair-mates. The result is that two group keys are computed, based either on the secret share *A* known of course by *A*, or on the secret share *BB* unknown by *A*. If *A* returns to the group before any data transmissions and authenticates itself to the group, it can reconstruct the first group key itself, by getting the appropriate blinded values of keys that are communicated freely in the network. If it comes back after the 2nd round, it can be sent the partial key α^{CD} by any node that knows this value. *A* requires this to reconstruct the partial key $a^{a^{AB}a^{CD}}$. Even if the node re-appears in another part of the network and its direct neighbors are different, the scheme still works. As long as multi-hop communication is supported by the routing protocol, two nodes can be virtual neighbors as well. Assume that node *A* reappears at the end of the 3d round. Then the group key has been already established. Similarly to the previous case, *A* will receive partial information: $a^{a^{CD}}$ and $a^{a^{a^{EF}a^{GH}}}$. On receiving this information *A* adds its own portion, and with the appropriate computations derives the group key. This scenario demands that each node stores the partial keys it computes during all *d*-1 rounds.

Considering however that *d* itself is not a very large number, we see that the nodes need not store too many values. The overhead would be significant if we started over the procedure of key establishment.

If *A* does not re-appear on time, the second group key that excludes *A* is used. Then, if a comes back after the session has started, authenticates itself and still wishes to participate in the group, the group key should be changed again if we want to maintain backward secrecy. However, since the "new" node is a re-appearing node we might want to overlook the backward secrecy rule. In this case, *B* can communicate either the group key or the share *BB* to *A*, encrypted with a^{AB} , the common two-party DH key *A* and *B* had initially calculated. This notion can be *generalized* to prevent the group key establishment from starting all over because of the abrupt disappearance of any node during this process. So, during the first round any of the 2^d nodes, computes two values: one generated by itself only, and one based on the contribution of its pair node as we have already seen. The remaining process, changes only in that more blinded values are communicated now from one pair to another, and more group-keys are finally derived.

4. *Network merge and partition*. If the group gets partitioned due to bad network connectivity then again there is no need to start the computation of the sub-group keys from scratch. The network gets partitioned in two or more groups, each of which can create new subgroup keys, based on previously stored information, limiting the communication and computation overhead. Given the structure of the subgroups, the communication and computation savings can be more (when pairs of one direction in a subgroup were initially pairs in the original group) or less (when all members of one subgroup acquired the same key in the *d*-1 round of the original group. They have to retrieve partial keys stored from the previous round, take scenario 2 into account and perform a hypercube key exchange the same manner as before, using an alternate direction pattern this time).

The case of subgroups merging into the one original group after communication has been restored is less complicated and less costly. It suffices that one member of each subgroup blinds its own subgroup key, sends it to the rest of the subgroups according to a predefined directional pattern, exactly as we do for the hypercube key establishment via DH exchanges. Then, with some additional DH key exchanges-one per member, all the remaining members of all the subgroups can compute the new group key.

Discussion:

The fact that the hypercube protocol requires 2^d participants imposes restrictions to the 2^d-Octopus protocol in terms of addition/eviction cases and group merging/partitions. However, the 2^d-Octopus acquires a lot of beneficial properties. The 2^d hypercube scheme is proven to be robust and fault-tolerant for most cases. In 2^d-Octopus, each member of the hypercube structure is the leader (GSC) of a subgroup of nodes of arbitrary number. Members of a subgroup establish a two-party DH key with the subgroup leader. The GSC uses the partial keys of its members to construct its initial secret share for the hypercube. After the group key is derived, the 2^d GSCs distribute parts of the group key to their sub-group members in a way that a member that does not belong to the sub-group cannot derive the group key. The key distribution protocols that these sub-groups support could be selected with much more freedom however. The original group is distributed now into 2^d subgroups. Each of these subgroups can be deployed in a relatively restricted area of the network and it is easier to handle these groups in a localized manner. Given the topology of the network we have the freedom to assign to each of these subgroups from 0 to as many members the protocol allows. This results in less communication overhead within the sub-group, in less bandwidth consumption, in less traffic in terms of the routing protocol.

Moreover, if a GSC becomes "faulty", it can be replaced by another node from its own subgroup. It is clear now how these properties render the protocol fault tolerant in the cases of addition/deletion, merging/partition.

Non-contributory One-Way Function Tree (OFT) Protocol

The number of keys stored by group members or broadcast to group when the membership changes and the computational efforts of members are logarithmic in the number of members. The protocol uses one-way functions to compute a tree of keys from the leaves to the root. The group manager maintains a binary tree, each node *x* of which is associated with two cryptographic keys, the un-blinded key k_x and the blinded key $k_x' = g(k_x)$, where *g* is a one-way function. The manager communicates securely with subsets of group members after they have obtained their session keys via a symmetric encryption function. Prior to this, it communicates a randomly chosen key to each member, associated with a leaf in the tree. To ensure that this operation is securely performed we assume that these randomly chosen keys are communicated with an asymmetric method (its higher computational complexity provides security) such as the RSA. Interior keys are defined by the rule: $k_x = f\,(g(k_{left(x)}), g(k_{right(x)}))$, where *f* is a mixing function that also adds to the security to the group. The key associated with the root of the tree serves as the group key. Each member knows only the un-blinded node keys on the path from its node to the root, and the blinded node keys of siblings to the nodes of its path to the root.

The member computes the un-blinded keys along its path to the root. If one of the blinded keys changes and the member gets the new value, then it can re-compute the keys on the path and find the new group key. This new value must be communicated to all of the members that store it, namely the descendants of the sibling *s* of *x* that know the un-blinded node key k_s. The manager encrypts $k_x^{'}$ with k_s before broadcast it to the group, providing the new value of the blinded key to the appropriate set of members. When a new member joins the group, an existing leaf node *x* is split, the member associated with *x* is now associated with *left(x)*, and the new member is associated with *right(x)*. Both members are given new keys. The new values of the blinded node keys that have changed are broadcast securely to the appropriate subgroups. In addition, the new member is given its set of blinded node keys, in a unicast transmission.

When the member of node *y* is evicted from the group the member assigned to the sibling of *y* is reassigned to the parent *p* of *y* and given a new leaf key value. The new values of the blinded node keys that have changed are broadcast securely to the appropriate subgroups. A broadcast of *h* keys is required to update all the other members who depended on the blinded node keys that have changed.

OFT	Total Delay	#broadcast bits	# unicast bits	Manager computation	Max member computation	#random bits generated
Initial	3n	2nk+h	0	3n	2h	nK
Add	3h	hK+h	hK	3h	2h	K
Evict	3h	hK+h	0	3h	2h	K

Table 4: Cost of parameters and Operations in OFT protocol

Evaluation of the Cost for the TGDH protocol

The TGDH protocol resembles OFT a lot in general. The basic differences are the following: any member of the tree can act as a central controller depending on its position in the tree, a member knows all blinded keys of the tree at any given time, and in TGDH the merging function is no other than the two node DH key exchange. The secret key *x* of an internal node *s* is the result of the DH key exchange between its offspring left(*s*) with associated secret key *y* and right(*s*) with associated secret key *z*. Then $x = \alpha^{yz}$, and the blinded key of node *s* is α^{x}. We assume that any member at any time can become group leader and broadcast a message to all the members of the group. The mobility of nodes might make it impossible for a simple node to broadcast a message to all members. In such an environment nodes may be close at one time interval and further at another. Therefore, this protocol is weak in MANETs, unless the network is rather restricted so that nodes stay relatively close to each other throughout the entire multicast session so the bandwidth problem is resolved.

The original paper does not define how the tree is originally formulated from a number of nodes wishing all to be part of the same group. We could assume that this is being done gradually, by executing the join operation for each new member that is successively incorporated into the tree. Instead, we take the freedom to propose a more efficient and less costly scheme. At the initial construction of the tree half of the members become sponsors. However, each time we go up a level in the tree towards the root, the number of sponsors is reduced to half because a parental node inherits both sponsors of its children but it needs only one, so the other becomes obsolete. At each level, the sponsors compute the un-blinded and blinded key of their parent, and broadcast the latter to the rest of the members. Each member computes all the keys in its path from the leaf to the root and also gets all the blinded keys of the tree.

Initial Member-Sponsor Computation/Communication: At every step, each node gets the blinded key of its sibling, raises it to the power of its own secret key and blinds the new key by one exponentiation in order to broadcast it. So, a sponsor does two exponentiations and one broadcast at every step. In the 1st level we assume all members are sponsors as well, in the 2nd, half of them remain sponsors etc. In terms of broadcasts we get: *n* in the 1st level, *n*/2 in the 2nd,…, $n/2^h$ in the *h*th. The total number of broadcast messages is approximately 2n, and the total number of exponentiations is 4n. However, we can claim that at maximum each sponsor/member does *log n* = *h* broadcasts and 2*log n* = 2*h* exponentiations.

Initial Member/Sponsor Communication: As we mentioned above, the total number of messages sent is 2*n*.

Initial Member/Sponsor Storage: Since every member stores all blinded keys of the tree, it stores (2^h-1) K= (n-1) K bits for all the blinded keys, and h K for all the un-blinded keys in its path to the root.

Add sponsor computation/communication: The sponsor generates a new member and an intermediate node, gets the blinded new key of the member and raises it to the power of its own secret key. The resulting key becomes the intermediate node's secret key, which the sponsor blinds. Similarly are calculated all updated nodes of the sponsor's path from the leaf to the root. So, the sponsor does 2*h* exponentiations, and sends to all members the updated *h* blinded keys. The new member however must get all blinded keys of the tree (it does not need the blinded keys of the leaves though). So, it gets (*n*/2+*n*/4+…+2) = *n*-2 blinded keys from the sponsor. Thus, the total number of messages the sponsor communicates is *n*-2 in the case of member addition and *h* in the case of member deletion.

Add/Delete member computation: The new member does *h* exponentiations (uses the blinded keys of its co-path) to get the group key. The rest (add/delete case) do from 1 to *h*-1 exponentiations, to compute the group key, since not all blinded keys change for them, and since they have already calculated the secret keys of their co-path. It is as follows: *n*/2 members do 1, *n*/4 do 2, $n/2^h$ do *h* exponentiations. In average each member does 2 exponentiations.

Delete Sponsor Computation/Communication: The sponsor becomes the right-most leaf node of the sub-tree routed at the leaving member's sibling node, which is promoted to replace the leaving member's parent node. The rest is similar to the add sponsor case.

Delete member computation: The members compute the new group key after having received the updated blinded keys that the sponsor broadcasts. As in the case of member addition exactly, every member in average does 2 exponentiations.

Delete communication: The sponsor broadcasts the updated *h* blinded keys to all members of the tree.

V. Design and Description of (MO) and (MOT) Protocols

The n group members are divided into 2^d subgroups of equivalent size. Each sub-group has one leader or else group security controller (GSC), e.g. the sponsor(s) in TGDH. We now modify the initial 2^d–Octopus protocol by replacing the 1st step where partial group keys are derived by the two party DH exchange, with GDH.2 / TGDH partial group key derivation. Subsequently we modify the 2nd and 3rd step of the original 2^d-Octopus protocol to the new hybrid protocols requirements and we analyze their performance. We denote the original 2^d-Octopus protocol as **(O)**, the GDH.2 modified 2^d-Octopus as **(MO)**, and the TGDH modified as **(MOT)** for simplicity. The papers on 2^d–Octopus protocol do not refer to the case of member addition/deletion. We analyze this case as well and derive the cost values for the (O), the (MO), and the (MOT) protocols.

Step1: Each subgroup establishes their sub-group key, or handles member additions/evictions exactly as indicated by GDH.2 and TGDH. We use a pseudo-random number generator C_{rr}, to create the secret shares of the three contributory protocols. This pseudo-random number generator is less complicated than the one we use to create number for the RSA system, C_r. At the end of the protocol evaluation we describe how to derive the complexity of the pseudo-random number generators.

(MO): Message length for 2^d GSCs: $2^{d-1}(\lceil n/2^d \rceil^2 + 3\lceil n/2^d \rceil - 2)K$. Average exponentiations per GSC for $\lceil n/2^d \rceil$ nodes: $\lceil n/2^d \rceil$. Average exponentiations per member: ($\lceil n/2^{d+1} \rceil$+1), according to GDH.2 formulas for the initial phase. In the add phase the GSC does: ($\lceil n/2^d \rceil$+1) exponentiations and the member does one exponentiation.

(O): During the DH key exchange among members and their GSCs, all n-2^d members send one message to their GSCs, and the GSCs broadcast to members 2^d messages in total. This is so because we assume that the GSC sends the same share a^x for members that belong to the same subgroup. Each member j of course creates its own secret share, λ_j, so that the secret key that each member j shares with its GSC is: $a^{x\lambda_j}$. So, we get n messages in total. The GSC does $2\lceil (n-2^d)/2^d \rceil$exponentiations in the initial phase and the member does two exponentiations in the same phase.

(MOT): In the initial phase as we have designed it, the maximum number of messages per sponsor is $log\left\lceil {}^{n}\!/_{2^d} \right\rceil$. For every node that lies in the sponsor's path, one broadcast message is sent (the node's blinded secret key is sent to the rest of members) and two exponentiations are done (one in order to construct the secret key of this node, and one to blind the secret key). In total, there is an exchange of $2\left\lceil {}^{n}\!/_{2^d} \right\rceil$communication messages per subgroup, and 2n for all 2^d subgroups, as we have already seen. Also, in total 4n exponentiations are required for all GSCs at the initial phase. In the case of addition, the sponsor does $2log\left\lceil \frac{n+1}{2^d} \right\rceil$ exponentiations and the member does two. The subgroup size in the addition case is: $\left\lceil \frac{n+1}{2^d} \right\rceil$, so $\left\lceil \frac{n+1}{2^d} \right\rceil$ messages are communicated. In the case of deletion the size of the subgroup is $\left\lceil \frac{n-1}{2^w} \right\rceil$, and $log\left\lceil \frac{n-1}{2^w} \right\rceil$messages are communicated, according to the TGDH formula.

Observation*1*: The group G that provides the generator α is of order q and has length K bits (the length of the group key). Thus, rising α to more exponents does not change the number of bits of the resulting value (exponentiations are modular). The complexity of each exponentiation is denoted as C_E.

Observation*2*: In the example for the simple Octopus protocol we used d=2. However, for d>2, the 3d step should be modified as follows: each of the 2^d participants (GSCs) must communicate d values to its group, in analogy to the two values that are derived in the case of the original Octopus (d=2). For instance, for d=3, we have eight GSCs noted as: *A, B, C, D, E, F,G* and *H*, and we expect the following sequence of operations:
In the 1st round the DH keys derived are: a^{AB}, a^{CD}, a^{EF}, a^{GH} .
In the 2nd round the DH keys derived are: $a^{a^{AB}a^{CD}}$, $a^{a^{AB}a^{CD}}$, $a^{a^{EF}a^{GH}}$,$a^{a^{EF}a^{GH}}$
In the 3d round the DH key derived is: $a^{a^{a^{AB}a^{CD}}a^{a^{EF}a^{GH}}}$ for all four pairs (this is the final group key).

For **(O)** protocol, if we take GSC *A* for instance, it has to communicate to its member *j* the following parts of the group key at the initial phase so that member *j* only securely derives the group key (according to the example for Octopus, d=2): a^{AB/K_j}, $a^{a^{CD}}$, $a^{a^{a^{EF}a^{GH}}}$. Thus, the GSC must communicate d=3 messages to each member of its subgroup. For **(MO)** and **(MOT)** protocols, for any member that belongs to the subgroup of GSC *A* for instance, the three parts of the group key that are sent by *A* are: a^{B}, $a^{a^{CD}}$, $a^{a^{a^{EF}a^{GH}}}$.

At the 3d step for the initial phase, members of all three protocols must execute d exponentiations each, to compute the final group key. This is so because each member gets d parts of the key. It raises the first part to the power of its own secret contribution and gets the first outcome. It then raises the second part to the first outcome, and gets the second outcome etc. It finally raises the dth part of the key to the (d-1)th outcome and gets the group key. It does d exponentiations in total.

For **(MO)** and **(MOT)** protocols, during the addition/eviction events, (d-1) of the d values need not be broadcast anew since they remain unchanged, provided that they are already stored in every member. So, the GSC communicates only one value to the whole sub-group, the value each particular member requires in order to reconstruct the group key, which is the same for every member in the subgroup. The GSC in the group of which a change of membership occurred need

not communicate anything to its members at the 3d step. The subgroup has calculated its updated subgroup key already, and all members of the subgroup can use the remaining (*d*-1) values they are storing to reconstruct the new group key. We illustrate these functions by continuing with the previous example where *d*=3. Assume that a change of membership occurs in the subgroup of GSC *A* and the new subgroup key is now χ . In this case, the members may not need to do all the *d* exponentiations as they did for the initial case. Actually the members that belong to the sub-groups of 2^{d-1} participants (*E*, *F*, *G*, *H* in our example) need to do only one, those of 2^{d-2} participants need to do two,…., those of 2 participants (*C*, *D* in our example) have do *d*-1, and those of the last two participants (*A*, *B* in our example) have to do *d* exponentiations. In average, members of each participant need to do two exponentiations in order to reconstruct the new group key in the case of addition/deletion. In the addition/deletion case where the subgroup key of GSC *A* is substituted we expect the following operations during the 2nd step:
In the 1st round the DH keys derived are: a^{BX} , a^{CD} , a^{EF} , a^{GH} .
In the 2nd round the DH keys derived are: $a^{a^{XB}a^{CD}}$, $a^{a^{XB}a^{CD}}$, $a^{a^{EF}a^{GH}}$, $a^{a^{EF}a^{GH}}$.
In the 3d round the DH key derived is: $a^{a^{a^{XB}a^{CD}}a^{a^{EF}a^{GH}}}$ for all four pairs (this is the new group key).

If we take GSC *A* for instance after the three rounds it should have stored the following values: χ , a^{B} , $a^{a^{CD}}$, $a^{a^{a^{EF}a^{GH}}}$, $a^{a^{a^{XB}a^{CD}}a^{a^{EF}a^{GH}}}$. Observe that the intermediate values stored remain unchanged from the previous time. The members of the subgroup of *A* will use these intermediate values to construct the group key. But the members already acquire these intermediate values as well as χ . Thus, the GSC that witnesses a membership update needs to communicate nothing to its sub-group at the 3d step.

GSC *B* for instance after the three rounds it should have stored the following values: β , a^{X} , $a^{a^{CD}}$, $a^{a^{a^{EF}a^{GH}}}$, $a^{a^{a^{XB}a^{CD}}a^{a^{EF}a^{GH}}}$. Observe that all intermediate values stored remain the same but for one: a^{X} . This is the only value that the GSC needs to communicate to the members of its sub-group after the derivation of the group key. Then, these members will be able to reconstruct the group key themselves by doing *d* exponentiations.

GSC *F* for instance, after the three rounds it should have stored the following values: f, α^{E} , $a^{a^{GH}}$, $a^{a^{a^{XB}a^{CD}}}$ $a^{a^{a^{EF}a^{GH}}}$, $a^{a^{a^{AB}a^{CD}}a^{a^{EF}a^{GH}}}$. Observe that all intermediate values stored remain the same but for one, $a^{a^{a^{XB}a^{CD}}}$. This is the only value that the GSC needs to communicate to the members of its sub-group after the derivation of the group key. Then, these members will be able to reconstruct the group key themselves by doing one exponentiation.

The case of addition/eviction for the 3d step when **(O)** protocol is applied presents the following differences: The subgroup key of each participant *A*, *B* etc is calculated only by the participant and not by the members, from the partial contributions of its members as we have already seen in the original protocol. Thus, using the same example as previously for (MO) and (MOT) protocols we observe the following:

For all participants (GSCs) except for *A* and *B*, one value only, the updated one, is broadcast to the members of their subgroups. The values that are broadcast are blinded values, and their free communication over the network causes no harm for the security of the system. However, both GSCs *A* and *B* need to communicate to their members the value a^{XB} which is not a blinded one. This value is to be known only to participants *A* and *B*. If a node obtains an un-blinded value that is not supposed to know, it can combine it with any of the blinded values that it can get, since they are freely broadcast, and derive more un-blinded keys and perhaps derive the group key. Members of groups of GSCs *X* and *B* need to obtain the un-blinded value a^{XB} however. So, in this case since there exist no subgroup key, we do as follows: for every member *j* with DH key K_j, its GSC communicates the value: a^{XB/K_j}. Clearly, only the particular member that knows K_j can derive the appropriate value to construct the group key. Thus, such a value is communicated individually to every member that belongs either to the subgroup that witnesses a membership update (subgroup of GSC *A* in this example), or to the subgroup of the GSC that is mate to the subgroup with the updated subgroup key, at the first round of the 2nd step (GSC *B* in the example, communicates with GSC A to derive a^{AB}).

GSC Storage:

(MO): The M_n member plays the role of the GSC since it needs to store the whole up-flow message that consists of *n* intermediate values each with *n*-1 exponents and of one value of *n* exponents. So in essence, we can assume that the GSC stores *n*+1 messages of length *K*. Since every GSC performs a DH key exchange with $\lceil {n-2^d}/{2^d} \rceil$ of the total members only, during the 1st step of the 2^d-Octopus protocol, the storage cost for the GSC is $\lceil {n-2^d}/{2^d} \rceil$. For the 2nd step, each participant needs at the worst case to store the value it computes during each of the total *d* rounds, so it stores *d* values. In total, the GSC stores $K(\lceil {n-2^d}/{2^d} \rceil + d)$ bits.

(O): The GSCs store all the information that is stored by (MO) and additionally they could store some of the intermediate products needed to derive the values: AB/K_j, for each member *j*. These products are approximately $\lceil {n-2^d}/{2^d} \rceil$ for each GSC as we are going to analyze in what follows.

(MOT): Since each TGDH tree acquires $\lceil {n-2^d}/{2^d} \rceil$ members, the GSC storage cost according to the TGDH formulas is: $\lceil {n-2^d}/{2^d} \rceil + \log \lceil {n-2^d}/{2^d} \rceil + 1$ at the 1st step and $\lceil {n-2^d}/{2^d} \rceil + \log \lceil {n-2^d}/{2^d} \rceil + (d+1)$ at the 2nd.

Member Storage: A member needs to store the session key and: the private 2-party DH key between the member and its GSC when (O) is applied, the private and the subgroup key when (MO) is applied, the private and the *h* blinded keys for the TGDH tree when (MOT) is applied. Every member needs additionally store the *d* parts that serve in constructing the group key and are sent by its GSC.

Initial communication:

(MO): For the GDH.2 we have: *n* rounds (*n*-1 unicast messages from the M_i member to the M_{i+1} member of length (*i*+1) *K* bits and one multicast message from member M_n to the rest *n*-1 members of length nK bits). The total message length in average is: $\sum_{i=1}^{n-1}(i+1)K + nK = (n^2+3n-2)K/2$

Initially, each of the 2^d participants establishes a DH key with its group of $\lceil (n-2^d)/2^d \rceil$ members. Using the GDH.2 formula we get: 2^{d-1} ($\lceil n/2^d \rceil^2 + 3\lceil n/2^d \rceil - 2)K$ for the total message length for the initial establishments of the group keys for all the 2^d group leaders.

Then, the 2^d GSCs execute the 2^d – Cube protocol via DH exchanges with initial values the keys they have obtained from the operation described above. We have a total of d rounds, in each of which we have a total exchange of 2^d messages (for every simple application of DH we have exchange of two messages, and for every round we have 2^{d-1} pairs and the group key is computed). So, we have a total message length of: $2^d dK$ bits.

At the 3d step, all d values are broadcast to the group as d separate messages. Here, we broadcast the first value to the whole subgroup, eliminating the contribution that corresponds to the subgroup key. This way, only members of the subgroup that know the subgroup key of course, can add the subgroup key contribution to the value they received and derive the appropriate part of the group key. In this case, all de parts are broadcast to the subgroup. So the total number of messages in this case is: $2^d d\ K$. For (MO), we get for the total initial communication: 2^{d-1} ($\lceil n/2^d \rceil^2 + 3\lceil n/2^d \rceil - 2)\ K + 2^d\ d\ K + 2^d d\ K = 2^{d-1}$ ($\lceil n/2^d \rceil^2 + 3\lceil n/2^d \rceil - 2)\ K + 2^{d+1}\ d\ K$

(O): The 2nd step is equivalent. At the 1st step we have (n-2^d) unicast DH messages from the members of the group to their GSCs. The GSCs can broadcast their DH message to the members of their group, thus we have 2^d broadcasts. So we have n messages in total for the 1st step. In the 3d step, we have (n-2^d) unicast and (d-1)2^d broadcast messages for the d values that have to be communicated to the members respectively (the d-1 values are common to the whole subgroup) from the GSCs to their subgroup members. For (O), initial communication is: $(n+2^d d+(n-2^d)+(d-1)2^d)K = (2n+(d-1)2^{d+1})\ K$ bits.

(MOT): Initially at the 1st step, the sponsors of each subgroup broadcast $2\lceil n/2^d \rceil$ keys for each subgroup. The 2nd step is the same as in the previous case so we have $2^d\ d\ K$ communication exchanges. For the 3d step the course of thought is exactly the same as for the (MO) protocol. So the initial communication is: $2^d\ 2\lceil n/2^d \rceil K + 2^d\ d\ K + 2^d\ d\ K = 2nK + 2^{d+1}\ d\ K$

GSC and Member Initial Computation:

(O): Almost all calculations of members are pure exponentiations. Every GSC at the 1st step must do $2\lceil (n-2^d)/2^d \rceil$ exponentiations and must generate $\lceil n/2^d \rceil$ random numbers using the pseudo-random number generator C_{rr}. Furthermore, the GSC after exchanging secret keys with all its sub-group members must multiply them to obtain the secret key it is going to use at the 2nd step. Multiplication of elements that are K bits long each, has complexity of K^2 bits approximately. These are modulo multiplications, thus if we are multiplying $\lceil (n-2^d)/2^d \rceil$ elements, the overall complexity is $\lceil (n-2^d)/2^d \rceil\ K^2$. Every simple member does two exponentiations for this step. At the 2nd step the 2^d GSCs do $2d$ exponentiations since each of them participates in one DH key exchange per round. At the 3d step, every member does d exponentiations as we have already observed. Every GSC computes $\lceil (n-2^d)/2^d \rceil$ products of ($\lceil (n-2^d)/2^d \rceil$)-1 members' secret keys. Each product consists of ($\lceil (n-2^d)/2^d \rceil$-1) factors (the factor that is missing is the secret key associated with the member to which the GSC intends to communicate the result of the particular multiplication). The complexity of these products seems to be (1/2) $\lceil (n-2^d)/2^d \rceil$ ($\lceil (n-2^d)/2^d \rceil$-1) if we compute each product from scratch. But there is a way to substantially reduce this complexity by working as follows: Assume we separate the secret keys of the members into k groups of approximately equal size x. Thus, A=$\lceil (n-2^d)/2^d \rceil$=$k$ x. We will work separately for each such group of x keys. Thus we will find the complexity of the same problem but for a smaller size of group each time, assume P(x), and we will combine all groups together. Generally, the number of multiplications needed in a group of size x, to find all combinations of multiplications of (x-1) and of x elements is: $x(x+1)/2-2$ (This result has been also derived for the same operation in GDH.2 protocol). Assume that we find the complexity for the operation of deriving the appropriate products in each such group. In order to derive an element which is the result of multiplying A-1 factors we have to multiply now k factors. Not two factors are taken out of the same group (there exist k groups of size x). Every such group contains x elements: (x-1) elements are derived by multiplying (x-1) keys, and one is derived by multiplying x keys. For the (x-1) elements of the particular group, say group 1, we have already found the rest of the factors that should be picked from the rest (k-1) groups in order to carry out the final multiplication: we should pick those elements that result from multiplication of x bits. For these (k-1) elements we do (k-2) multiplications, and the product is going to apply to all x-1 elements of group 1. This is exactly the course of thought for all the rest k-1 groups. Thus for all (k-1) such combinations we have to do $(k-1)k/2-2$ multiplications. The complexity of combining all the elements together to produce the appropriate products, once we have calculated the complexity of producing the x elements of each group is: $(k-1)\ k/2+k(x-1)-2$. The overall complexity of the problem is: $P(A) = P(kx) = k(x(x+1)/2-2)+(k-1)\ k/2+k(x-1)-2 = A(x+1)/2 - (2A/x) + (A^2/2x^2) - (A/2x)+A-(A/x)-2$. We want to reduce this complexity as much as possible by defining the appropriate size of x. After calculations we get: $x^3+x=2A$. The complexity (after a little rounding) is: $P(A)= A(A^{1/3}+1.25)$. This is a substantial reduction in complexity. Now $P(\lceil (n-2^d)/2^d \rceil)=(\lceil (n-2^d)/2^d \rceil)^{4/3}+1.25\ (\lceil (n-2^d)/2^d \rceil)$.

After the appropriate products have been derived, the GSC does the following calculation for each of its subgroup members: it raises the blinded key it gets from another GSC during the first round of the 2nd step, to the power of the product related to this member, and communicates the results to the member. It does the same for all its sub-group members, thus it does $\lceil (n-2^d)/2^d \rceil$exponentiations. In total, the amount of computation each GSC does initially is: $\lceil n/2^d \rceil C_{rr}+(2\lceil (n-2^d)/2^d \rceil+2d+\lceil (n-2^d)/2^d \rceil)C_E+(\lceil (n-2^d)/2^d \rceil)^{4/3}+1.25(\lceil (n-2^d)/2^d \rceil)K^2$. The total initial computation on the part of each member is: (d+2) exponentiations.

(MO): The GSC (M_n) does n exponentiations when we apply the (MO) protocol. In the 1st step each one of the 2^d GSCs does $\lceil (n-2^d)/2^d \rceil$ exponentiations. Each member M_i does (i+1) exponentiations of the same complexity each. We can say that in average the exponentiations a member does are (½)$\lceil (n-2^d)/2^d \rceil$. And it does one more exponentiation when it gets the stream message in order to construct the session key K_n. In the 2nd step the 2^d GSCs do $2d$ exponentiations since each of them participates in one DH key exchange per round. In the 3d step every member does d exponentiations as we have noted in the observations. Therefore, a GSC does $\lceil (n-2^d)/2^d \rceil$+2$d$ exponentiations and generates $\lceil n/2^d \rceil$ random numbers (using the pseudo-random number generator C_{rr}) in total. A member does (½)$\lceil (n-2^d)/2^d \rceil$+$d$ exponentiations.

(MOT): At the 1st step, the sponsors of the same subgroup do $2log\left\lceil {n}/{2^d} \right\rceil$exponentiations according to our analysis for TGDH, and also generate $\left\lceil {n}/{2^d} \right\rceil$ random numbers (using the pseudo-random number generator C_{rr}). At the 2nd step, the 2^d GSCs do $2d$ exponentiations since each of them participates in one DH key exchange per round. At the 3d step, every member does d exponentiations as we have noted in the observations.

The papers on the original Octopus protocol do not refer to the cases that we have membership updates (additions/evictions) within the group. We will analyze the case of member addition/deletion and calculate the respective cost values for the (O), the (MO), and the (MOT) protocols.

The key idea here is to initially re-compute the subgroup key only for the subgroup that has accepted the new member to join. So, we run again the protocol executing the 1st step only, for the subgroup mentioned above. Then we run the 2nd and the 3d steps as we are going to describe in what follows. However we can observe here that at the 2nd step not all the calculations need to be done anew. The DH key exchange that does not involve participants whose key value has changed do not need to carry out again the same operation. We observe that at the first round we will modify the keys of two participants, at the second round we will modify the keys of four participants, at the third of eight participants, and at the dth round we will modify the keys of 2^d participants.

Important Security Constraint:

(O): In the case of member addition/eviction we do not acquire the GDH.2 or the TGDH properties to disguise the contributions of the rest of the members of the subgroup to the new/evicted member. If the protocol is used as such, then there is not backward secrecy for the new member. When the GSC sends the d parts of the group key to the new member it essentially sends to it the appropriate parts to construct the old group key. Also, there is no forward secrecy in the case of eviction because the evictee will be able to derive the new key simply by eliminating its own DH key (it may be possible under circumstances) from the previous group key. So, we do the following simple modification: one member preferably of the same subgroup in which the addition/eviction occurs, modifies its secret share and establishes a new DH key with its GSC. We want to have at least another exponent (other than the new member's for the addition case) modified, so that we can disguise the old value from any past and future subgroup members. Thus, the GSC in the case of member join does two DH key exchanges and in the case of eviction does one DH key exchange. Also, the GSC does four exponentiations at the 1st step for the join case and two for the eviction case. Two members do two exponentiations at the 1st step for the join case, and one member does one exponentiation at the same step for the eviction case.

(MO), (MOT): At the 1st step, GDH.2 protocol produces a sub-group key of the form $a^{ab...z}$ = a^N. At the 1st step, TGDH protocol produces a sub-group key of the form $K_{\langle 0,0\rangle} = \alpha^{(\alpha^{r_3\alpha^{r_1 r_2}})(\alpha^{r_4\alpha^{r_5 r_6}})}$. = a^{xy} = a^N. The resulting key for both cases is equivalent to (O) in the case that subgroup contained one member only other than the GSC. A designated member or sponsor from each subgroup that is the first to derive the subgroup key and broadcast it to the rest of members, plays the role of the GSC and provides the subgroup key for the 2nd step. In the end, all 2^d participants broadcast to their subgroups d values similar to those broadcast at the 3rd step when (O) is applied. Each member raises one specific part out of the *d* parts it receives from its GSC, to the power of its subgroup key known from the 1st step. The subgroup key is known to all members that belong to the same subgroup and thus it suffices that all *d* values are broadcast to the members of the same subgroup. So the number of messages communicated in this case is $2^d d$. For the addition/eviction events (d-1) of the *d* values need not be broadcast anew since they remain unchanged, provided that they are already stored in every member. The GSC in the group of which a change of membership occurred need not communicate anything to its members at the 3d step. All the rest of GSCs broadcast only the updated value to their members. Thus, in the case of member eviction/addition the number of messages communicated at the 3d step is: 2^d-1.

GSC Addition Computation:

(MO): In GDH.2 M_n generates a new exponent N_n' and computes a new up-flow message using N_n' not N_n and sends it to the new member M_{n+1}. Now member M_{n+1} becomes the GSC and computes the new key K_{n+1}. And it computes and broadcasts to the other group members the *n* sub-keys. So this member computes n+1 intermediate values, it does n+1 exponentiations. Since in our case the new number of nodes in this subgroup is $\left\lceil n/2^d \right\rceil$+1, we have $\left\lceil n/2^d \right\rceil$+1exponentiations at the 1st step of the algorithm. At the 2nd step of the algorithm the amount of exponentiations each one of the 2^d GSCs does is not the same. We have already explained that the results of some of the DH exchanges remain the same so we don't need to execute the DH operation for every participant in every round. Given that the basis in the *d*-dimensional vector space remains the same, we have a way to determine for every GSC in how many DH exponentiations it is going to participate. The participant that belongs to the subgroup in which an update of membership occurred participates in all the *d* rounds (as well as its mate from the first round of course). In average a GSC participates in $\left\lceil \frac{d+1}{2} \right\rceil$rounds. In every such round, it does two exponentiations. Finally, at the 3d step all the GSCs have computed the group key, they just broadcast to the members of their sub-group the appropriate value indicated at the 3d step of the protocol. As we analyzed earlier, the GSC does no computations at this step. Thus, the GSC does in total: ($\left\lceil n/2^d \right\rceil$+1+2$\left\lceil \frac{d+1}{2} \right\rceil$)$C_E$+$C_{rr}$ computations.

(O): The GSC in the case of join generates two random numbers and does two DH key exchanges (four exponentiations). Then it multiplies the keys of all members together and executes a single exponentiation of this product (complexity of one exponentiation and $\left\lceil n-2^d/2^d \right\rceil K^2$ bits due to the multiplication). This sequence of operations can be also viewed as $\left\lceil n-2^d/2^d \right\rceil$successive exponentiations with the DH key of each member every time. The 2nd step is handled exactly the same way as in (MO): each GSC does 2$\left\lceil \frac{d+1}{2} \right\rceil$exponentiations. At the 3d step only two GSCs do calculations as we have discussed in the observations. These two GSCs do exactly the amount of computations done by any GSC at the 3d step of the initial phase. Each of the two GSCs computes $\left\lceil n-2^d/2^d \right\rceil$ products of ($\left\lceil n-2^d/2^d \right\rceil$)-1 members' secret keys. Each product consists of ($\left\lceil n-2^d/2^d \right\rceil$-1) factors, and the factor that is missing is the secret key of the member to which the GSC intends to communicate the result of the particular multiplication. After the appropriate products have been derived, the GSC does the following calculations for each of its subgroup members: it raises the blinded key that it gets during the first round of the 2nd step to the power of the product related to the particular member, and communicates the results to this member. It does the same for all its subgroup members, thus it does $\left\lceil n-2^d/2^d \right\rceil$exponentiations. In total the amount of computation for each of the two GSCs is: ($\left\lceil {}^{n-2^d}\!/_{2^d} \right\rceil$+4+2$\left\lceil \frac{d+1}{2} \right\rceil$ +$\left\lceil {}^{n-2^d}\!/_{2^d} \right\rceil$)$C_E$ +($\left\lceil {}^{n-2^d}\!/_{2^d} \right\rceil$)$^{4/3}$+1.25 ($\left\lceil {}^{n-2^d}\!/_{2^d} \right\rceil$)$K^2$ +2C_{rr}. The rest of GSCs do (2$\left\lceil {}^{n-2^d}\!/_{2^d} \right\rceil$+2$\left\lceil \frac{d+1}{2} \right\rceil$) exponentiations. Assume that all GSCs store all the products from their previous multiplications, even the intermediate ones. In that case each of the two GSC needs to do as follows: It computes the product of the two updated DH keys. For ($\left\lceil {}^{n-2^d}\!/_{2^d} \right\rceil$-2) members it suffices that their intermediate product of ($\left\lceil {}^{n-2^d}\!/_{2^d} \right\rceil$-3) keys (the product lacks the contributions of the member itself and of the members that are associated with the updated keys) be multiplied with the product of the updated keys. Each of the rest two members is multiplied with one of the updated keys or the other respectively. In this case, each of the two GSCs need only carry out: 1+2+2($\left\lceil {}^{n-2^d}\!/_{2^d} \right\rceil$-2) = 2$\left\lceil {}^{n-2^d}\!/_{2^d} \right\rceil$-1 multiplications. In this case the total amount of calculations these two GSCs do is: (2$\left\lceil {}^{n-2^d}\!/_{2^d} \right\rceil$+4+2$\left\lceil \frac{d+1}{2} \right\rceil$+$\left\lceil {}^{n-2^d}\!/_{2^d} \right\rceil$)$C_E$+2($\left\lceil {}^{n-2^d}\!/_{2^d} \right\rceil$-1) K^2 +2C_{rr}.

(MOT): The sponsor does $2log\left\lceil {}^{n+1}\!/_{2^d} \right\rceil$exponentiations at the 1st step (addition computation case in TGDH) and generates two random numbers. The 2nd step is handled exactly the same way as in (MO), each sponsor (GSC) does 2$\left\lceil \frac{d+1}{2} \right\rceil$exponentiations. The 3d step is similar to the (MO) case, during which the GSC does no further exponentiations. So, the total amount of calculations for the (MOT) protocol is: ($2log\left\lceil {}^{n+1}\!/_{2^d} \right\rceil$+2$\left\lceil \frac{d+1}{2} \right\rceil$) C_E +2C_{rr}.

Member Addition Computation:

(MO): The member gets the broadcast stream and it only needs to do one exponentiation to get the key K_{n+1} in GDH.2 protocol. Thus, members that belong to the subgroup in which membership update occurs are required to do one exponentiation to get the new subgroup key during the 1st step. At the 3d step, as we already know each member of every sub-group must do *d* exponentiations or less to reconstruct the group key, in average two.

(O): At the 1st step two members of the same subgroup only, are required to do two exponentiations to create a two-party DH with the group GSC each. Then, at the 3d step, each member of every subgroup does *d* exponentiations or less to reconstruct the group key, in average two.
(MOT): At the 1st step the members do in average two exponentiations as we have seen in the TGDH formulas. At the 3d step, as we already know each member of every subgroup does *d* exponentiations or less to reconstruct the group key, in average two.

Add Communication:

(MO): For the GDH.2, the M_n member sends to the M_{n+1} member an up-flow message of $(n+1)K$ length. Then M_{n+1} member broadcasts to all n members its n+1 intermediate values, of total length $(n+1)K$. So the total length of the add communication is $2(n+1)K$.
At the 1st step of the protocol, one of the 2^d subgroups only performs the GDH.2. In this group we have $\left\lceil {}^{n-2^d}\!/_{2^d} \right\rceil$members so the length of the add communication is: 2$\left\lceil {}^{n}\!/_{2^d} \right\rceil$K.

At the 2nd step as we have already discussed we have d rounds, in the round i however we have 2^i DH key exchanges instead of 2^d. For every DH exchange we have 2 communication messages sent, one to each node and the common key is computed. So we have $2\ (1+2+\ldots 2^{d-1})$ communication messages. The total number of bits communicated is $2(2^d\text{-}1)K$ bits.
At the 3d step, the GSC needs only communicate (broadcast) one value to the members of its subgroup as we have already discussed. All members receive the value that has changed for them. However, all members of the subgroup in which the new member joined get no value. So, $(2^d\text{-}1)$ $\left\lceil {}^{n-2^d}\!/_{2^d} \right\rceil$members get a single message. The total number of bits transmitted during the 3d step is: $(2^d\text{-}1)K$. Consequently, the total number of bits communicated in the case of a member join for the (MO) protocol is: $2\left\lceil {}^{n}\!/_{2^d} \right\rceil K+2(2^d\text{-}1)K+(2^d\text{-}1)K$.

(O): At the 1st step two DH exchanges take place (four communication messages). The 2nd step is exactly the same way as in (MO). During the 3d step however the communication of messages to the members of two groups (the one in which the membership update occurs and the one whose participant communicates during the 1st round of the 2nd step with the participant of the updated member) is done in unicast. So, $2\left\lceil {}^{n-2^d}\!/_{2^d} \right\rceil$ members get one message and $(2^d\text{-}2)$ groups get a single broadcast message. The total number of bits transmitted during the 3^d step is: $((2^d\text{-}2) +2\left\lceil {}^{n-2^d}\!/_{2^d} \right\rceil)K$. The total number of bits communicated in the case of a member join for the (O) protocol is: $4K+2(2^d\text{-}1)\ K+(2^d\text{-}2)K+2\left\lceil {}^{n-2^d}\!/_{2^d} \right\rceil K$.

(MOT): At the 1st step the new member of the subgroup sends its blinded key to its sibling (sponsor). Then, the sponsor broadcasts the updated h blinded keys to all the members in the tree, and additionally communicates all the n blinded keys of the tree to the new member. So, since the subgroup acquires $\left\lceil {}^{n+1}\!/_{2^d} \right\rceil$ nodes, the total communication bits for this step are: $(1+log\left\lceil {}^{n+1}\!/_{2^d} \right\rceil+\left\lceil {}^{n+1}\!/_{2^d} \right\rceil)K$. The 2nd step is handled the same way as in (MO). The number of bits communicated is $2(2^d\text{-}1)K$. The 3d step is again similar to the 3d step for (MO). So the total communication is: $(1+log\left\lceil {}^{n+1}\!/_{2^d} \right\rceil+\left\lceil {}^{n+1}\!/_{2^d} \right\rceil)K + 2(2^d\text{-}1)K + (2^d\text{-}1)\ K$.

GSC Deletion Computation:

(MO): For the GDH.2 protocol, if member M_p is evicted then M_n, that is the current GSC, computes a new set of n-2 sub-keys. What is missing is the term $\alpha^{N_1*\ldots N_{p-1}*N_{p+1}\ldots N_{n-1}*N_n^{/}}$ $\alpha^{N_1*\ldots N_{p-1}*N_{p+1}\ldots N_{n-1}*N_n^{/}}$ so that M_p cannot compute the new key. The procedure is exactly like the GSC Addition Computation case, with the only difference that now M_n must do n-1 exponentiations. It also generates one random number using the pseudo-random number generator C_{rr}.
During the 1st step, the GSC has to carry out $\left\lceil {}^{n}\!/_{2^d} \right\rceil$exponentiations to compute the updated subgroup key according to the GDH.2 formulas. For the 2nd step as we have already described for the GSC addition computation, the participant that belongs to the subgroup of the new incomer participates in all the d rounds as well as its mate from the first round. In average a GSC participates in $\left\lceil \frac{d+1}{2} \right\rceil$rounds, and does $2\left\lceil \frac{d+1}{2} \right\rceil$exponentiations. For the 3d step, no further exponentiations are required from the GSCs. Thus, the total computation of a GSC for the case of member deletion is: $(\left\lceil {}^{n}\!/_{2^d} \right\rceil+2\left\lceil \frac{d+1}{2} \right\rceil)\ C_E +C_{rr}$.

(O): The GSC for the eviction case does one DH key exchange (two exponentiations) and generates one random number using the pseudo-random number generator C_{rr}. Then it multiplies the keys of all members together and exponents the resulting product (one exponentiation and $\left\lceil \frac{n-2^d}{2^d} \right\rceil K^2$ bits due to the multiplication). This sequence of operations can be also viewed as $\left\lceil \frac{n-2^d}{2^d} \right\rceil$successive exponentiations with the DH key of each member every time. The 2nd step is handled exactly as it is described for the previous cases: each sponsor (GSC) does $2\left\lceil \frac{d+1}{2} \right\rceil$exponentiations. At the 3d step only two GSCs do calculations as we have discussed in the observations. Each of the two GSCs computes $\left\lceil \frac{n-2^d}{2^d} \right\rceil$ products of ($\left\lceil \frac{n-2^d}{2^d} \right\rceil$)-1 members' secret keys. Each product consists of ($\left\lceil \frac{n-2^d}{2^d} \right\rceil$-1) factors, and the factor that is missing is the secret key of the member to which the GSC intends to communicate the result of the particular multiplication. After the appropriate products have been derived, the GSC does the following calculation for each of its subgroup members: it raises the blinded key that it gets during the first round of the 2nd step to the power of the product related to this member, and communicates the results to the member. It does the same for all its subgroup members, thus it does $\left\lceil \frac{n-2^d}{2^d} \right\rceil$exponentiations. In all three steps the total complexity of computations (exponentiations and multiplications) for each of these two GSCs is: ($\left\lceil \frac{n-2^d}{2^d} \right\rceil$+2+2$\left\lceil \frac{d+1}{2} \right\rceil$+$\left\lceil \frac{n-2^d}{2^d} \right\rceil$) C_E+($\left\lceil \frac{n-2^d}{2^d} \right\rceil$)$^{4/3}$+1.25($\left\lceil \frac{n-2^d}{2^d} \right\rceil$)$K^2$. The rest of GSCs do (2$\left\lceil \frac{n-2^d}{2^d} \right\rceil$+2$\left\lceil \frac{d+1}{2} \right\rceil$) exponentiations. Assume that all GSCs store all the products from their previous multiplications, even the intermediate ones. In that case, the following are done: for ($\left\lceil \frac{n-2^d}{2^d} \right\rceil$-2) members it suffices that their intermediate product of ($\left\lceil \frac{n-2^d}{2^d} \right\rceil$-3) keys (the product lacks the contributions of the particular member itself, of the evicted one and of the member associated with the updated key) be multiplied with the updated key. In this case, each of the two GSCs need only carry out: 1+2($\left\lceil \frac{n-2^d}{2^d} \right\rceil$-2) = (2$\left\lceil \frac{n-2^d}{2^d} \right\rceil$-3) multiplications. The total amount of calculations these two GSCs do at all three steps is: (2$\left\lceil \frac{n-2^d}{2^d} \right\rceil$+2+2$\left\lceil \frac{d+1}{2} \right\rceil$+$\left\lceil \frac{n-2^d}{2^d} \right\rceil$)$C_E$ +(2$\left\lceil \frac{n-2^d}{2^d} \right\rceil$-3)$K^2$ +C_{rr} .

(MOT): At the 1st step, exactly as in the Add Sponsor Computation case, the sponsor (GSC) does $2h$ exponentiations, thus $2log\left\lceil \frac{n-1}{2^d} \right\rceil$exponentiations since the size of each group is approximately: $\left\lceil \frac{n-1}{2^d} \right\rceil$. It also generates a random number using the pseudo-random number generator C_{rr}. The 2nd step is handled exactly the same way as in (MO): the number of exponentiations for each GSC is: $2\left\lceil \frac{d+1}{2} \right\rceil$. For the 3d step, no further exponentiations are required from the GSCs. Thus, the total computation of a GSC for the case of member deletion is: ($2log\left\lceil \frac{n-1}{2^d} \right\rceil$+2$\left\lceil \frac{d+1}{2} \right\rceil$)$C_E$ +C_{rr} .

Member Deletion Computation

(MO): For GDH.2, it is exactly the same as for the member Addition Computation case. The member gets the broadcast stream and it only needs to do one exponentiation to get the key K_{n+1} in GDH.2. So, for the 1st step, every member needs to do one exponentiation. Then, at the 3d step the member does d exponentiations at maximum or two in average, and the group key is computed anew.

(O): At the 1st step only one member of the subgroup of the evicted member is required to do two exponentiations. Then, at the 3d step, every member does d exponentiations at maximum or two in average, and the group key is computed anew.

(MOT): At the 1st step the members of the subgroup of the evicted member compute the new subgroup key after having received the updated blinded keys that the sponsor broadcasts. In analogy to the case of member addition, every member in average carries out two exponentiations. At the 3d step, every member does d exponentiations at maximum, or two in average and the group key is computed anew.

Delete Communication:

(MO): In GDH.2 only member M_n broadcasts a message of length n-1 to the rest of the members. So the total overhead for the delete communication is: $(n-1)K$. At the 1st step, the total messages for the delete communication are: $(\lceil \frac{n-2^d}{2^d} \rceil-1)K$. Next, the 2nd and the 3d steps are exactly as in the Add Communication case for (MO). So the total number of bits that have to be communicated is: $(\lceil \frac{n-2^d}{2^d} \rceil-1)K+2(2^d-1)K+(2^d+1)K$.

(O): At the 1st step, we update the contribution of another member of the subgroup in which a membership eviction occurs. This member exchanges new DH key with its GSC (two communication messages). The 2nd step is handled the same way as in the Addition Communication case for (O). The total number of bits communicated is $2(2^d-1)K$ bits. At the 3d step however the communication of messages to the members of two groups (the one in which the membership eviction occurs and the one whose GSC communicates during the 1st round of the 2nd step with the GSC of the evicted member) is done in unicast. So, $2\lceil \frac{n-2^d}{2^d} \rceil$members get one message and (2^d-2) groups get a single broadcast message. The total number of bits transmitted during the 3d step is: $(2^d-2)K+2\lceil \frac{n-2^d}{2^d} \rceil K$. So, the total number of bits to transmit in (MO) is: $2K+2(2^d-1)K+(2^d-2)K+2\lceil \frac{n-2^d}{2^d} \rceil K$ bits.

(MOT): At the 1st step, the sponsor broadcasts the updated h blinded keys to all the members of the tree. So, we have $log\lceil \frac{n-1}{2^d} \rceil$messages broadcast since the size of the subgroup is $\lceil \frac{n-1}{2^d} \rceil$. For the 2nd step we have the same exchange of messages exactly as in the 2nd step of the Addition Communication case. The total number of bits communicated is $2(2^d-1)K$. The 3d step is again similar to the 3d step for (MO). So the total communication is: $log\lceil \frac{n-1}{2^d} \rceil K+2(2^d-1)K+(2^d-1)K$.

Cost	2^d-Octopus (O)	Mod. 2^d- Octopus (GDH.2)- (MO)	Mod.2^d-Octopus (TGDH)-(MOT)
GSC Storage	$K(\lceil \frac{n-2^d}{2^d} \rceil+d)$ / $K(2\lceil \frac{n-2^d}{2^d} \rceil+d)$	$K(\lceil \frac{n-2^d}{2^d} \rceil+d)$	$(\lceil \frac{n}{2^d} \rceil+\log\lceil \frac{n}{2^d} \rceil+d)K$
Member Storage	$(2+d)K$	$(d+1)K$	$(\lceil \frac{n}{2^d} \rceil+\log\lceil \frac{n}{2^d} \rceil+d)K$
Initial GSC Comput.	$(3\lceil \frac{n-2^d}{2^d} \rceil+2d)C_E+(\lceil \frac{n-2^d}{2^d} \rceil)^{4/3}+1.25(\lceil \frac{n-2^d}{2^d} \rceil)K^2+\lceil \frac{n}{2^d} \rceil C_{rr}$	$(\lceil \frac{n}{2^d} \rceil+2d)C_E+\lceil \frac{n}{2^d} \rceil C_{rr}$	$(2\log\lceil \frac{n}{2^d} \rceil+2d)C_E+\lceil \frac{n}{2^d} \rceil C_{rr}$ at max.

Initial Members Comput.	$(d+2)C_E$	$((1/2)\lceil n/2^d \rceil+d)C_E$	$(2\log \lceil n/2^d \rceil+d)C_E$ at max
Initial Comm/tion	$(2n+(d-1)2^{d+1})K$	$(2^{d-1}(\lceil n/2^d \rceil^2+3\lceil n/2^d \rceil-2)+2^{d+1}d)K$	$(2^d 2\lceil n/2^d \rceil+2^{d+1}d)K$
Add GSC Computat.	$(3\lceil (n-2^d)/2^d \rceil+4+2\lceil \frac{d+1}{2} \rceil)C_E+2C_{rr}$ $+(\lceil (n-2^d)/2^d \rceil)^{4/3}+1.25(\lceil (n-2^d)/2^d \rceil)K^2$ $/(3\lceil (n-2^d)/2^d \rceil+2\lceil \frac{d+1}{2} \rceil+4)C_E+2C_{rr}$ $+2(\lceil (n-2^d)/2^d \rceil-1)K^2$, one $(2\lceil (n-2^d)/2^d \rceil+2\lceil \frac{d+1}{2} \rceil)C_E$ rest	$C_E(\lceil (n+1)/2^d \rceil+1+2\lceil \frac{d+1}{2} \rceil)+C_{rr}$, one $C_E(2\lceil \frac{d+1}{2} \rceil)$, rest	$C_E(2\log\lceil (n+1)/2^d \rceil+2\lceil \frac{d+1}{2} \rceil)+2C_{rr}$, one GSC $C_E(2\lceil \frac{d+1}{2} \rceil)$, rest
Add Members Comput.	$4C_E$, two $(2+d)C_E$ max. $2C_E$, the rest dC_E max.	$3C_E$, one subgroup $(1+d)C_E$ max. $2C_E$, rest dC_E max.	$4C_E$, one member $(h+d)C_E$ max $2C_E$, rest dC_E max
Add Comm/tion	$(4+2(2^d-1)+(2^d-2)+2\lceil (n-2^d)/2^d \rceil)K$	$(2\lceil (n+1)/2^d \rceil+2(2^d-1)+(2^d-1))K$	$(\log\lceil (n+1)/2^d \rceil+\lceil (n+1)/2^d \rceil+2(2^d-1)+(2^d-1))K$.
Delete GSC Comput.	$(3\lceil (n-2^d)/2^d \rceil+2+2\lceil \frac{d+1}{2} \rceil)C_E+C_{rr}+$ $(\lceil (n-2^d)/2^d \rceil)^{4/3}+1.25(\lceil (n-2^d)/2^d \rceil)K^2$ / $(3\lceil (n-2^d)/2^d \rceil+2+2\lceil \frac{d+1}{2} \rceil)C_E+C_{rr}+$ $+(2\lceil (n-2^d)/2^d \rceil-3)K^2$, one $(2\lceil (n-2^d)/2^d \rceil+2\lceil \frac{d+1}{2} \rceil)$ rest	$C_E(\lceil (n-1)/2^d \rceil+2\lceil \frac{d+1}{2} \rceil)+C_{rr}$, one $C_E(2\lceil \frac{d+1}{2} \rceil)$, rest	$C_E(2\log\lceil (n-1)/2^d \rceil+2\lceil \frac{d+1}{2} \rceil)+C_{rr}$, one $C_E(2\lceil \frac{d+1}{2} \rceil)$, rest
Del. Members Comput.	$3C_E$, two $(1+d)C_E$ max. $2C_E$, the rest dC_E max.	$3C_E$, one subgroup or $(1+d)C_E$ max. $2C_E$, rest dC_E max.	$4C_E$, one member $(h+d)C_E$ max $2C_E$, rest dC_E max
Delete Comm/tion	$(2+2(2^d-1)+(2^d-2)+2\lceil (n-2^d)/2^d \rceil)K$	$((\lceil (n-1)/2^d \rceil-1)+2(2^d-1)+(2^d+1))K$	$(\log\lceil (n-1)/2^d \rceil+2(2^d-1)+(2^d-1))K$

VI. Security Issues of (MO) and (MOT) Protocols

We argue that the security of the protocols is of the same standard as the security of the initial Octopus. In the 1st step (MO) inherits the properties of GDH.2 protocol, proven to be secure, and (MOT) inherits the properties of TGDH, also proven to be secure. The 2nd step is based on successive secure DH exchanges, and remains basically unchanged from the original version. The 3rd step serves in distributing the group key to all the members according to the principles of DH key exchange, and the principles of GDH.2 and TGDH. Every operation in this step is based on these protocols, and the messages communicated are encrypted with keys that derive from any of these protocols. Thus, our hybrid protocols inherit the security principles of DH in these steps. Security and robustness is ensured at every step and every operation for these protocols.

VII. Comparison

Cost	2^d-Octopus (O)	Mod. 2^d- Octopus (GDH.2)-(MO)	OFT	Mod.2^d-Octopus (TGDH)-(MOT)

GSC Storage	$K(\lceil \frac{n-2^d}{2^d} \rceil + d)$	$K(\lceil \frac{n}{2^d} \rceil + d)$	3n	$(\lceil \frac{n}{2^d} \rceil + \log \lceil \frac{n}{2^d} \rceil + d) K$
Member Storage	(2+d)K	(d+1)K	$2\log \lceil \frac{n}{2^d} \rceil$	$(\lceil \frac{n}{2^d} \rceil + \log \lceil \frac{n}{2^d} \rceil + d) K$

Cost	2^d-Octopus (O)	Mod. 2^d- Octopus (GDH.2) (MO)	OFT	Mod. 2^d - Octopus (TGDH) – (MOT)
Initial GSC Computation	$(3\lceil \frac{n-2^d}{2^d} \rceil + 2d)C_E + (\lceil \frac{n-2^d}{2^d} \rceil)^{4/3} + 1.25(\lceil \frac{n-2^d}{2^d} \rceil)K^2 + \lceil \frac{n}{2^d} \rceil C_{rr}$	$(\lceil \frac{n}{2^d} \rceil + 2d)C_E + \lceil \frac{n}{2^d} \rceil C_{rr}$	$n(C_r + C_{PE}) + n(C_{SE} + 2C_g)$	$(2\log \lceil \frac{n}{2^d} \rceil + 2d)C_E + \lceil \frac{n}{2^d} \rceil C_{rr}$ at max.
Initial Members Computation	$(d+2)C_E$	$((1/2) \lceil \frac{n}{2^d} \rceil + d)C_E$	$C_{PD} + h\,C_{SD} + h\,C_g$	$(2\log \lceil \frac{n}{2^d} \rceil + d)C_E$ at max.
Initial Comm/tion	$(2n + (d-1)2^{d+1})K$	$(2^{d-1}(\lceil \frac{n}{2^d} \rceil^2 + 3\lceil \frac{n}{2^d} \rceil - 2) + 2^{d+1} d)K$	2n K / 3nK	$(2^d 2 \lceil \frac{n}{2^d} \rceil + 2^{d+1} d)K$
Add GSC Computation	$(3\lceil \frac{n-2^d}{2^d} \rceil + 4 + 2\lceil \frac{d+1}{2} \rceil)C_E + 2C_{rr} + (\lceil \frac{n-2^d}{2^d} \rceil)^{4/3} + 1.25(\lceil \frac{n-2^d}{2^d} \rceil)K^2$ / $(3\lceil \frac{n-2^d}{2^d} \rceil + 2\lceil \frac{d+1}{2} \rceil + 4)C_E + 2C_{rr} + 2(\lceil \frac{n-2^d}{2^d} \rceil - 1)K^2$, one $(2\lceil \frac{n-2^d}{2^d} \rceil + 2\lceil \frac{d+1}{2} \rceil)C_E$ rest	$C_E(\lceil \frac{n+1}{2^d} \rceil + 1 + 2\lceil \frac{d+1}{2} \rceil) + C_{rr}$, one $C_E(2\lceil \frac{d+1}{2} \rceil)$, rest	$C_r + C_{PE} + h(C_{SE} + 2C_g)$ / $2C_r + 2C_{PE} + h(C_{SE} + 2C_g)$	$C_E(2\log \lceil \frac{n+1}{2^d} \rceil + 2\lceil \frac{d+1}{2} \rceil) + 2C_{rr}$, one GSC $C_E(2\lceil \frac{d+1}{2} \rceil)$, rest
Add Members Computation	$4C_E$, two $(2+d)C_E$ max. $2C_E$, the rest dC_E max.	$3C_E$, for one subgroup $(1+d)C_E$ max. $2C_E$, rest dC_E max.	$C_{PD} + hC_{SD} + hC_g$	$4C_E$, one $(h+d)C_E$ max $2C_E$, rest dC_E max
Add Comm/tion	$(4 + 2(2^d-1) + (2^d-2) + 2\lceil \frac{n-2^d}{2^d} \rceil)K$	$(2\lceil \frac{n+1}{2^d} \rceil + 2(2^d-1) + (2^d-1))$ K	(2h + 1) K	$(\log \lceil \frac{n+1}{2^d} \rceil + \lceil \frac{n+1}{2^d} \rceil + 2(2^d-1) + (2^d-1))K$.
Delete GSC Computation	$(3\lceil \frac{n-2^d}{2^d} \rceil + 2 + 2\lceil \frac{d+1}{2} \rceil)C_E + C_{rr} + (\lceil \frac{n-2^d}{2^d} \rceil)^{4/3} + 1.25(\lceil \frac{n-2^d}{2^d} \rceil)K^2$ / $(3\lceil \frac{n-2^d}{2^d} \rceil + 2 + 2\lceil \frac{d+1}{2} \rceil) C_E + C_{rr} + (2\lceil \frac{n-2^d}{2^d} \rceil - 5) K^2$, one $(2\lceil \frac{n-2^d}{2^d} \rceil + 2\lceil \frac{d+1}{2} \rceil)$ rest	$C_E(\lceil \frac{n-1}{2^d} \rceil + 2\lceil \frac{d+1}{2} \rceil) + C_{rr}$, one $C_E(2\lceil \frac{d+1}{2} \rceil)$, rest	$C_r + C_{PE} + h(C_{SE} + 2C_g)$	$C_E(2\log \lceil \frac{n-1}{2^d} \rceil + 2\lceil \frac{d+1}{2} \rceil) + C_{rr}$, one $C_E(2\lceil \frac{d+1}{2} \rceil)$, rest
Delete Members Computation	$3C_E$, two $(1+d)C_E$ max. $2C_E$, the rest dC_E max.	$3C_E$, one subgroup or $(1+d)C_E$ max. $2C_E$, rest dC_E max.	$C_{PD} + hC_{SD} + hC_g$	$4C_E$, one $(h+d)C_E$ max $2C_E$, rest dC_E max
Delete Comm/tion	$(2 + 2(2^d-1) + (2^d-2) + 2\lceil \frac{n-2^d}{2^d} \rceil)$ K	$((\lceil \frac{n-1}{2^d} \rceil - 1) + 2(2^d-1) + (2^d+1))$ K	(h+1) K	$(\log \lceil \frac{n-1}{2^d} \rceil + 2(2^d-1) + (2^d-1))K$

Table 4: Complexities of Octopus Key Agreement protocols and of Contributory OFT

Table 4 shows the results of the comparison. In terms of Initial GSC Computation, MOT behaves better than all the other protocols. In terms of GSC Addition/Eviction Computation, (O) demonstrates the worst behavior. MO and mainly MOT, perform much better, particularly when parameter d is relatively small and value ($n/2^d$) is not too large. As for the Initial Communication, the performance of MO is the worst, MOT and (O) however manage to slightly outperform OFT. For the Addition/Eviction Communication (very critical issue for MANETs), (O) is outperformed by both MO and MOT. MOT in particular for the membership update case, achieves the best performance of all the rest of the Octopus-based contributory protocols. This performance places MOT closer to OFT than any other contributory protocol in terms of membership updates cost and this is a major achievement for MOT. Furthermore, MOT outperforms OFT at the overall computation costs for the membership updates for certain combinations of parameters (n,d).

However, unless we divide large groups into subgroups of restricted number of members that also lie topologically close, we cannot claim that our protocol is robust in MANETs. This is the price that we have to pay for achieving robustness and scalability for contributory protocols in environments where the topology is changing fast and nodes have resources limitations. The results for the hybrid contributory protocols we designed for MANETs are not pessimistic at all. In some cases their performance gets close to the performance of one of the most efficient non-contributory protocols that assumes a trusted third party to distribute keys, OFT.

Algorithms for finding large primes:

The Theorem for **Prime Numbers** states that the number of primes less than n, $P(n)$, as n tends to infinity is given by: $\boldsymbol{P(n) = n/\log_e n}$. For simplicity we symbolize $\log_e n=\log n$.
The density of primes is thus $dP(n)/dn = 1/\log n - 1/\log^2 (n)$.
For large n the second term is small and we can omit it. So, for n=512 we get density of primes = (1:512) /log2=1:355
This suggests that primes be found by trial and error. Choose a large (uneven) integer n, test it for primality, if the test fails make change and try again. With 512 -bit integers we would expect success in 355/2= 178 attempts.
Generally the average number of attempts is: $\log_2 n \times (1/2\log 2) = 0.7 \times \log_2 n$

Existing algorithm due to Rabin and Miller (implemented by Knuth) (Algorithm P) for Primality test:

A **Primality Test** provides an efficient probabilistic algorithm for determining if a given number is prime [14], [15]. It is based on the properties of strong pseudoprimes. Given an odd integer n, let $\boldsymbol{n=2^r s+1}$ with s odd. Then choose a random integer a: $\boldsymbol{1 \le a \le (n-1)}$. If $\boldsymbol{a^s = 1 \pmod n}$ or $\boldsymbol{a^{2^j s} = -1 \pmod n}$ for some $\boldsymbol{0 \le j \le r-1}$, then n passes the test. A prime will pass the test for all a.
The test is very fast and requires no more than $\boldsymbol{(1+o(1)) \log_2 n} < 2\times \log_2 n$ modular multiplications. Unfortunately, a number that passes the test is not necessarily prime. Monier and Rabin (1980) have shown that a composite number passes the test for at most 1/4 of the possible bases $\boldsymbol{a}$**.**

Calculation of Key Generation cost: C_r

Overview of key generation

Two random numbers must be generated. For 512-bit key we get two 256-bit random numbers and 2^{254} possible numbers are obtained from 254 random bits.

The whole security of the system depends on the fact that the program must be capable of generating each and every one of these 2^{254} numbers with more-or-less equal probability. Next, from each random number generate a prime number. Starting from the given number, you look at each successive odd number to see if it is prime. A simple sieve is used to eliminate obvious cases, using a table of the first 1028 prime numbers. Anything that survives the sieve is subjected to Fermat's test (if $(x^{p-1} \bmod p) <> 1$, then p is not prime). Next, multiply the two (alleged) primes P and Q to give the modulus N, and from P, Q and N the RSA public and private keys can be derived in the standard manner. One of the most popular secure pseudo-random number generators is the **Blum-Blum-Shub (BBS)** pseudo-random bit generator:

1. Calculate the operations for choosing x relatively prime to n. Pick up an odd number $x<n$. According to the Theorem of Prime Numbers we can find a relative prime number after length(n)/ ($log2\times 2$) attempts (this will be done only the first time we use the BBS pseudo-random number generator).
2. Calculate $x_j = x^2_{j-1}$ (mod n)
3. Take b_j to be the least significant bit of x_j
4. Go to step 2. Derive the new x_j and take the new b_j and repeat as many times as the length of the bits we wish for our key.

A way to speed up this slow operation is: after every multiplication extract the k least significant bits of x_j. As long as $k \leq log_2 log_2 n$, the scheme is cryptographically secure. Assuming that length(x)=length(n) and selecting length(n)=K we find that for step 2 we need multiplication of K bits each time. So we need K^2 operations. We repeat step 2 for (K/k) times in order to derive K random generated bits. k is fixed s.t $k \leq log_2 K$. So, the total number of operations for generation of one key is: $(K/(2\times log2)) + (K/k) \times K^2$.
$\boldsymbol{Cr = (K\times 0.7) \times K^2}$ (this term is used the first time that the BBS pseudo-random generator is initiated, it can even be precalculated, so in fact it does not add to the overall complexity)**+** $\boldsymbol{(K/k) \times K^2 = (K^3\times 0.7) + (1/k) \times K^3}$, where $\boldsymbol{1 \leq k \leq log_2 K}$

Remark: In general this particular algorithm generates a random key with computational complexity O(K^3). In general there are algorithms that generate random keys with computational complexities that vary. In the bibliography we have encountered upon algorithms with the following complexities: O(K^4), O($log_2 K \times K^3$), O($log_2 K \times K^2$), etc. However, we decide to present this particular algorithm because it is among the most popular for generating random keys. Another remark here is that this algorithm just generates random but not prime numbers. In what follows we are going to demonstrate an algorithm that generates random prime numbers.

Estimation of C_g

C_g is the cost for the one-way function that blinds a key. Generally, it is suggested that it can be calculated using the MD5 or the SHA method [10], [11]. We used MD5 to estimate the complexity of C_g.

MD5 takes a message and turns to multiple to 512 bits. If the length of the key is K, $T=\left\lceil \frac{K}{512} \right\rceil$ is the number times we need to multiply 512 bits (sixteen 32-bit words), to construct the padded message that we are going to use in the procedure.

In brief, MD5 makes four passes over each 16-byte chunk of the message. Each pass has a slightly different method of mangling the message digest. The message is processed in 512-bit blocks. Each stage computes a function based on the 512-bit message chunk and the message digest (128-bit quantity) to produce a new intermediate value for the message digest. At the end

of the stage, each word of the mangled message digest is added to its pre-stage value to produce the post-stage value that becomes the pre-stage value for the next stage. The value of the message digest is the result of the output of the final block of the message. Every stage is repeated for all T 512-bits chunks of the message. In every stage a separate step is taken for each of the 16 words of the message. During every such step we have addition of 5 terms (including the previous blocks, the message digest and a constant). In two of the four stages the message digest function has complexity cost of 3×32 bits, and in the rest two stages 2x32 bits. The rest of the terms as well as the output of the digest function is 32 bits. The left rotates imply complexity of 32 bits as well.
Complexity for each chunk (512-bit) of message for all the 16 blocks of 32 bits for all stages:
1st stage:$(4+3)\times32+32$, 2nd stage: $(4+3)\times32+32$, 3d stage: $(4+2)\times32+32$, 4th stage: $(4+2)\times32+32$.
Totally we have: $16\times32\times30 = 15360$.

The complexity for all chunks of the message is approximately: $T\times15360$ bits = $\lceil \frac{K}{512} \rceil$x15360 = $30K$. So we can roughly estimate $C_g = 30K$. The certain conclusion is that C_g is linear to the length of the key and also a cheap and secure operation if we use the MD5 method. So $C_g = 30K$

Calculation of the C_{PE}, C_{PD}:

RSA Algorithm:
Plaintext M is encrypted in blocks
Encryption: $C = M^{e} \bmod n$ **Decryption:** $M = C^{d} \bmod n$
Public key = $<n, e>$ Private key = $<n, d>$
Requirements: There exist $<n, e, d>$ such that $M = M^{ed} \bmod n$ for all $M < n$. It is easy to calculate M^e and C^d for all $M < n$. It is infeasible to find d given n and e

C_{PE}:

We gererate two numbers via the Random Key Generation method. In order to use them for the RSA method we want these numbers p, q to be prime. We have already seen that the complexity for the generation of these two random numbers (not necessarily primes) p and q is: $\mathbf{(1/k)\times(\log_2 p)^3}$, where $\mathbf{1 \le k \le \log_2 p}$ and $\mathbf{(1/m)\times(\log_2 q)^3}$, where $\mathbf{1 \le m \le \log_2 q}$ respectively.
Thus, we need to select two large prime numbers p, q where $p\times q=n$. By method of trial and error we try to see which p, q will pass the *Primality Test*. We have already calculated that we need **length(n)×0.7** trials in average to find number n that fulfills all the requirements. Each different n is candidate to pass the *Primality Test*, which gives complexity: $(1+o(1))\times log_2 n < 2\times log_2 n$ modular multiplications. Each modular multiplication has computational complexity $(\text{length}(n))^2$. So, the total complexity to derive the appropriate prime n given any random number m is: $1.4\times((\text{length}(n))^4)$. However, we need to consider the complexity for deriving this random number m in the first place. We have already calculated that this complexity is: $(1/k)\times(\log_2 m)^3$. Since we need $0.7\times\text{length}(m)$ trials in average, we need to derive a random number $0.7\times\text{length}(m)$ times, thus the overall complexity for deriving the required p and q random numbers are: $(1/k)\times0.7\times(\log_2 p)^4+1.4\times(\log_2 p)^4=((0.7/k)+1.4)\times(\log_2 p)^4$ and $((0.7/k)+1.4)\times(log_2 q)^4$ respectively. We also know that $log_2 n=log_2 p+log_2 q$. Thus, $log_2 p<log_2 n$ and $log_2 q<log_2 n$ and we can use this bound to calculate the overall complexity of generating the random primes p and q. Thus, the overall complexity is: $\mathbf{((1.4/k)+2.8)\times(log_2 n)^4 = ((1.4/k)+2.8)\times K^4}$
We have just calculated the complexity for generating random primes to be used for the RSA key encryption/decryption algorithm.

Proceeding to the next steps, we want: $e\times d = 1 \bmod (p-1)\times(q-1)$. If e is known we can derive d with O(1) calculations.
So we need to calculate the encryption of the session key with length ***K*.**

In many versions of RSA, e is assumed fixed with length K_e. A popular value for e is $2^{16}+1$ or 3. We find that in the worst case we perform $2*K_e$ modular multiplications and $2*K_e$ divisions, and $2*logK_e$ other operations thus the total complexity is approximately $(4K_e+2logK_e)\times K^2$ assuming that length(n)=$\log_2 n$=length(m)=K. Each of these operations has a cost analogous to the length of the message m (in our case the length K of the session key). At every step we multiply K either with itself or with the outcome of a previous multiplication truncated with modulo n so the result is always less than n. Roughly the total number of operations are: $(4\times K_e+2\times log_2K_e)\times(\text{length}(n))^2$.

So the final computation cost for the C_{PE} variable is: $\boldsymbol{C_{PE}=(4\times K_e+2\times \log_2 K_e)\text{ x }K^2}$

Observation: From equation: $e\times d = 1 \bmod (p\text{-}1)\times(q\text{-}1)$ we see that $K_e+K_d\times \geq log_2p+log_2q=K$. Thus, the selection of K_e affects the selection of K_d and vice versa. The most popular value for e is 3. Generally it is preferred d to be very large for the decryption algorithm, since the larger the d the more difficult is for an attacker to break the algorithm. In the case that we select e=3, we select a number d with size $K_d \approx K$. This is why, in the RSA algorithm, the encryption is much faster than the decryption.

C_{PD}:

For the decryption the same idea is adopted. Things however are slightly easier. We already have n, e, d and we only need to carry out the decryption of the message: $m= c^d$ mod n. If we use Shamir's assumption for the unbalanced RSA that $\boldsymbol{c^d \bmod n}$ can be reduced to $\boldsymbol{r^{dl} \bmod p}$ where $\boldsymbol{d1= d \bmod \phi(p)}$ and $\boldsymbol{r = c \bmod p}$**,** we achieve a speedup of length(p). According to Shamir's proposition we have roughly:
$n=p\times q$ but also $lgn=10\times lgn'$ and $lgp=2\times lgp'$, where the same equation used to hold for n', p': $n'=p'\times q'$. From these equations we see that lgn/lgp >5, and this is to say that the use of mod p instead of mod n produces as decryption cost more than 25 times less than the encryption cost:
A similar idea for the decryption used in the Chinese Remainder Theorem: we can speed decryption up to four times by computing: c^d mod p and c^d mod q instead. The Chinese Remainder Theorem then allows us to deduce $c^d \bmod p\times q = c^d \bmod n$. The idea of the Chinese Remainder Theorem is used for the decryption only, since p and q are known only to the member that decrypts the message. In the case of encryption, the member encrypting a message with the public key does not know p and q (unless it has created the public-secret pair).
However, this operation doesn't necessarily make the decryption faster than the encryption. In some asymmetric key systems such as ECC and Braid Method, the decryption speed is faster than the encryption speed. In systems like NTRU and RSA encryption is faster. In RSA this is due to the choice of e that is usually a small number. However, in RSA the encryption speed up achieved by the efficient choice of e for the encryption, is larger than the decryption speed up achieved by the Chinese Remainder Theorem for the decryption. As we mentioned earlier if we select e=3 then we have to select a number d with size $K_d \approx K$.

Thus for the decryption the following is derived: $C_{PD} = ((4/25)\times K_d+(2/25)\times log_2K_d)\times K^2$
Finally: $\boldsymbol{C_{PD} = ((4/25)\times K_d+(2/25)\times log_2K_d)\times K^2}$

Special case: If e=3 then $C_{PD} = ((4/25)\times K+(2/25)\times \log_2 K)\times K^2 = O(K^3)$ whereas $C_{PE}=O(K^2)$.

DES and calculation of C_{SE}, C_{SD}

DES is a ***block cipher***: it operates on plaintext blocks of a given size (64-bits) and returns ciphertext blocks of the same size. Thus DES results in a ***permutation*** among the 2^64 possible arrangements of 64 bits. Each block of 64 bits is divided into two blocks of 32 bits each, a left half block ***L*** and a right half ***R***. DES operates on the 64-bit blocks using *key* sizes of 56- bits. The keys are actually stored as being 64 bits long, but every 8th bit in the key is not used. DES encrypts and decrypts data in 64-bit blocks, using a 64-bit key. It takes a 64-bit block of plaintext as input and outputs a 64-bit block of ciphertext. It has 16 rounds, meaning the main algorithm is repeated 16 times to produce the ciphertext. It has been found that the number of rounds is exponentially proportional to the amount of time required to find a key using a brute-force attack. So as the number of rounds increases, the security of the algorithm increases exponentially

We will calculate the number of computations by following the algorithm step by step:
We have permutation of the initial 64-bit input, then 16 DES rounds for each one of which we generate the per round keys and finally a last permutation of the 64-bit output. These permutations are not random, they have a specific structure so they require a fixed number of operations: $C_{permute}$. To generate the per round keys we do an initial permutation of the 56 useful bits of key, to generate a 56-bit output which it divides into two 28- bit values, called Co, Do. These permutations again require a fixed number of operations: $C_{permute}$
We permute 24 of those bits as the left half and 24 as the right half of the per round key. A completely randomly chosen permutation of k bits would take about $k\times log_2k$ bits. So we have two permutations that take $24\times log_2 24$ bits each. From every bit we produce $log_2 24$ more bits.
In a DES round in encryption the 64 bits are divided into two 32-bit halves called L_n and R_n. The output generates 32- bit quantities L_{n+1} and R_{n+1}. Their concatenation is the 64-bit output of the round. We have to make extra calculations only to produce $R_{n+1.}$ From R_n and per round key of 48-bits we produce an outcome of 32 bits which is XORed with L_n This procedure requires about 8 operations for the 32 bits extension to 48 bits, 64 for the mapping in S boxes, 32 for the last XORing. So, totally we need about $O(2\times C_{permute}+24\times \log_2 24 +16\times 144)$ operations for a DES encryption. It is proven that we need the same amount of operations for the DES decryption.
However we might want a key length or data length larger than 64 bits. We still can use the DES method for larger encrypted messages or larger encryption keys the following way. We will break the whole decryption into chunks of 64 bits and will use DES to encrypt each block. If the key has more than 64 bits we can break the encryption key into chunks as well and perform the encryption of the message. In our case this would be the appropriate thing to do because the session key produced initially from RSA would have length larger than the length of the key produced by DES. So, we need to perform DES encryption and decryption $K/64$ times roughly.

Now we can calculate the computation costs C_{SD} and C_{SE} that are going to be roughly the same.
So: $C_{SE}=C_{SD}= (K/64)\times O(2\times C_{permute}+24\times log_2 24 +16\times 144)$ $C_{SE}=C_{SD} < (K/64) \times 2500$
We see that C_{SE}, C_{SD} have cost linear to the key length so we can attribute to them the final form:
$\boldsymbol{C_{SE}=C_{SD}= C_{DES} \times K}$, where $\mathbf{35 \leq C_{DES} \leq 80}$

Exponentiation

Exponentiation is an operation with cubic complexity: if the size of number that is involved (modulus) is doubled, the number of operations increases by a factor of $2^3 = 8$.
Exponentiation is done by looking at the bit pattern of the exponent as successive powers of 2, and then successively squaring the argument and multiplying with modulus as necessary [14], [15]. At worst this involves $2\times l$ modular multiplications or $4\times l$ multiplications and divisions where l is the length of the exponent. Thus, 16 bit exponentiation involves $64\times(32)^3 = 2^{21} = 2\times 10^6$ operations. The general form is: $4\times l\times(2^\wedge(\log_2 l+1))^3 = 4\times l\times(2\times l)^3=32\times l^4$.

In the case where the exponent and the modulo N have the same size K in bits this complexity could be expressed as: $32\times K^4$

The complexity of this general formula for modular exponentiation can be further refined as we see in what follows:

A 512 bit implementation of RSA in software on most computers will take about one second. But if exponentiation is done under a modulus, things are a lot easier as we see from the following example:

Suppose we want to take 299 to the 153rd power, under a modulus of 355. The first thing we do is to note that 153 equals 128 + 16 + 8 + 1 (binary decomposition), and that therefore,

299^153 = 299^128×299^16×299^8×299^1.

Now, we can compute the term on the far right; it's 299. Under the modulus, it's still 299. But knowing the one on the right, we can compute the one next to it: Doubling the powers by squaring the numbers, and applying the modulus to intermediate results, gives:

299 mod 355 is 299. (299 = 299^1 mod 355)
299^2 = 89401, but 89401 mod 355 is 296. (296 = 299^2 mod 355)
296^2 = 87616, but 87616 mod 355 is 286. (286 = 299^4 mod 355)
286^2 = 81796, but 81796 mod 355 is 146. (146 = 299^8 mod 355)
146^2 = 21316, but 21316 mod 355 is 16. (16 = 299^16 mod 355)
16^2 = 256, and 256 mod 355 is 256. (256 = 299^32 mod 355)
256^2 = 65536, but 65536 mod 355 is 216. (216 = 299^64 mod 355)
216^2 = 46656, but 46656 mod 355 is 151. (151 = 299^128 mod 355)

Thus, 299^153 mod 355 = (151 ×16 ×146 ×299) mod 355 which now looks a whole lot easier. At this point total multiplications/divisions are in the worst case: $\mathbf{4\times \boldsymbol{l}\times \boldsymbol{n}^2}$, where n is the modulo.

Now we proceed by calculating the product and applying the modulus to intermediate results:
151 ×16 = 2416, but 2416 mod 355 is 286.
286 ×146 = 41756, but 41756 mod 355 is 221.
221 ×299 = 66079, but 66079 mod 355 is 49.
Thus, 299^153 mod 355 = 49.
For this part total operations are in the worst case: $2\times log_2 l \times n^2$

And that is how you take a large number to a large power under a large modulus using the most efficient algorithm. The constant application of the modulus operation to intermediate results prevents you from having to deal with any number larger than the square of the modulus in any case.

So, if l is the exponent size and n the modulo size then the total number of operations is: $\mathbf{(4\times \boldsymbol{l}+2\times \boldsymbol{log_2 l})\times \boldsymbol{n}^2}$**.** The general complexity is: $O(l\times n^2)$.

For the case that the exponent and the modulo have the same size we get total complexity of: $O(n^3)$.

In the following Table an approximate estimation of the cost (in bits) of the public encryption/decryption parameters C_{PE} and C_{PD} respectively, of the symmetric encryption/ decryption parameters C_{SE} and C_{SD} respectively, of key generation C_r, of exponentiation C_E and of hashing C_g operations that are used for the estimation of computational costs is presented.

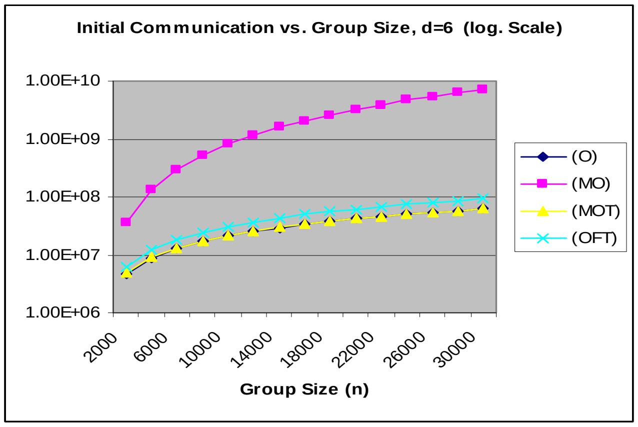
C_{rr}	$(K\times \log 2/2) + (K/k) \times K^2$ pseudo- random number generation

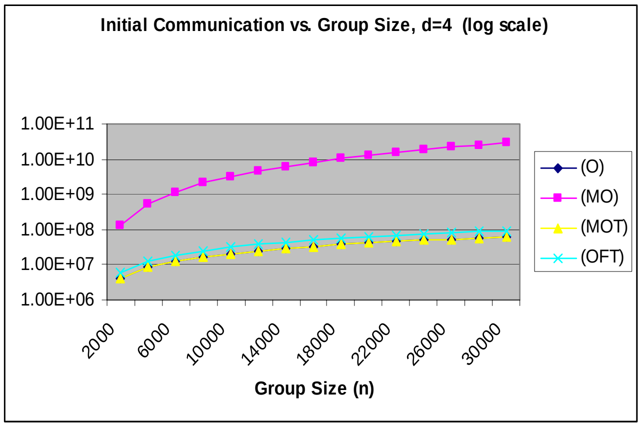
C_r	$((1.4/k)+2.8) \times K^4$ pseudo-random generation for primes (used for RSA keys)
C_{PE}	$(4\times K_e+2\times \log_2 K_e) \times K^2$
C_{PD}	$((4/25)\times K_d+(2/25)\times \log_2 K_d)\times K^2$
C_{SE}	$C_{DES} \times K$
C_{SD}	$C_{DES} \times K$
C_E	$(4\times l+2\times \log_2 l)\times K^2$
C_g	$30\times K$

Table 5: Costs of parameters used in the protocols.

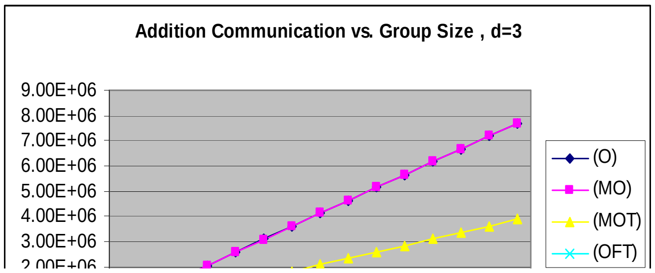
For the asymmetric encryption/decryption we use the RSA method and for the symmetric encryption/decryption we use the DES method:
$C_{SE}=C_{SD}= C_{DES} \times K$, where $35 \le C_{DES} \le 80$, $1 \le k \le \log_2 K$
Most common selections for K_e and K_d are: $K_e =2$, $K_d \approx< K$.
For the exponentiation we assume that l is the size of the exponent and K is the size of modulo n. We select this expression for the complexity, among other variations that exist in the current bibliography.

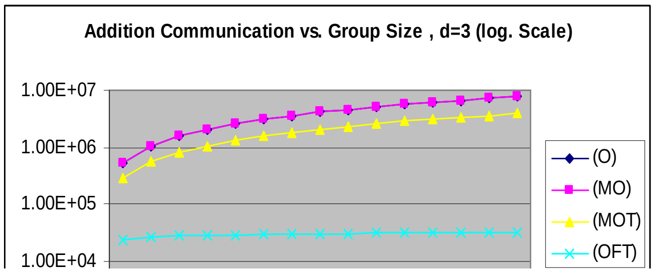
Graphic Results of the Performance Evaluation



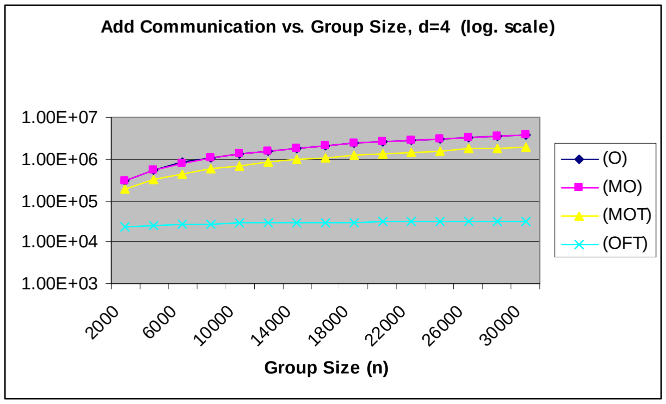


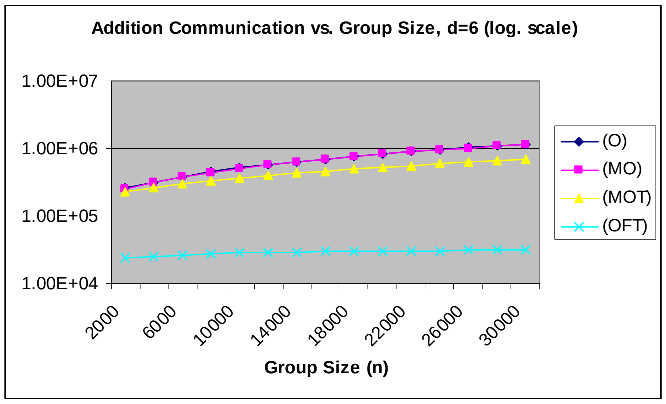
Figures 1, 2: Initial communication for all protocols vs. group size. *d*=6 and *d*=4.



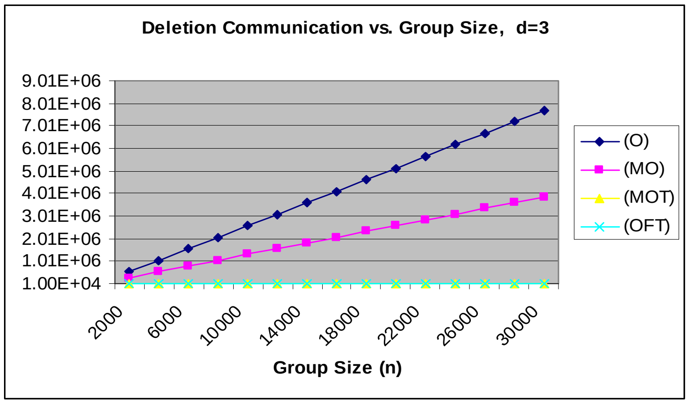


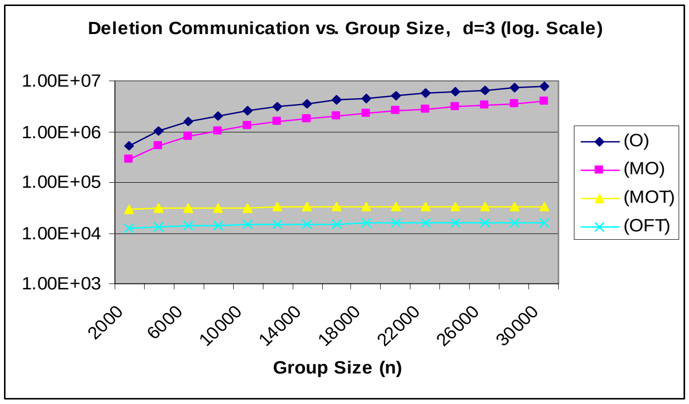
Figures 3, 4: Addition Communication for all protocols vs. group size. *d*=3.



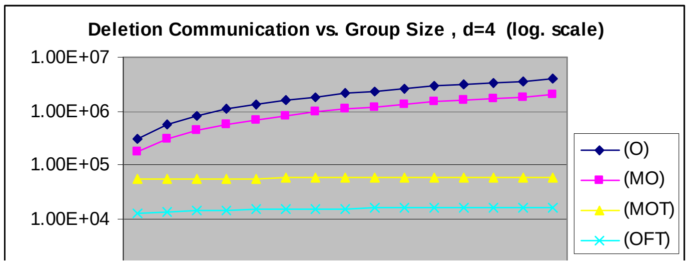


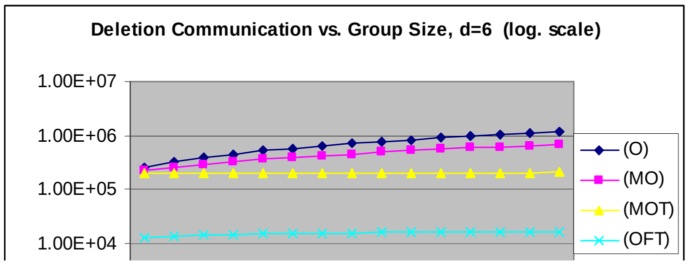
Figures 5, 6: Addition Communication for all protocols vs. group size. *d*=4, *d*=6.



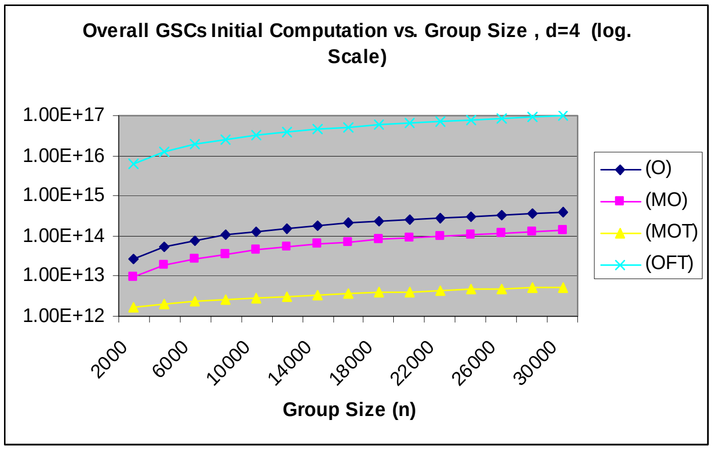


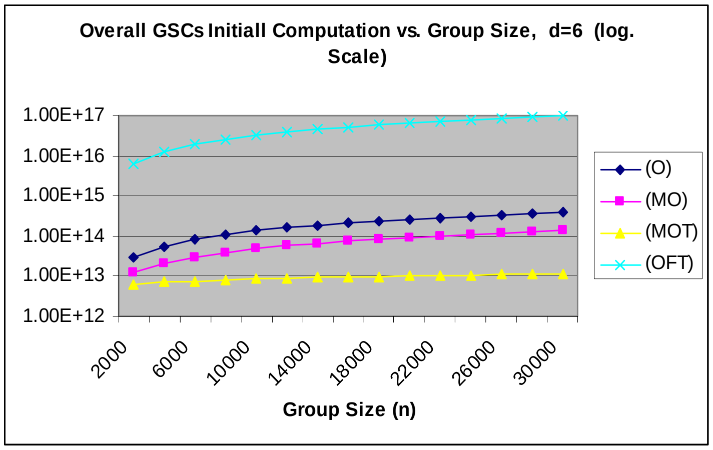
Figures 7, 8: Deletion Communication for all protocols vs. group size. *d*=3



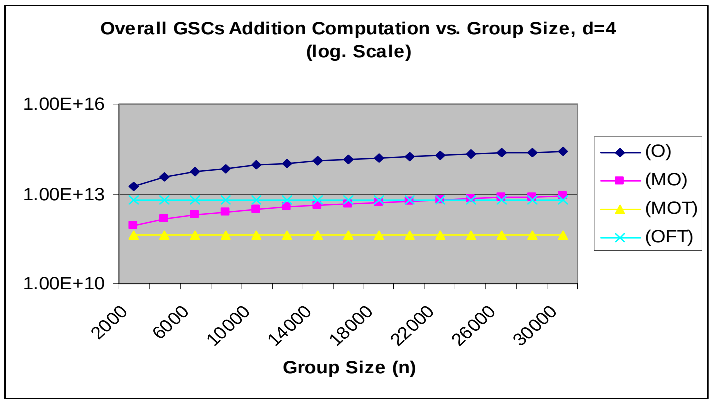


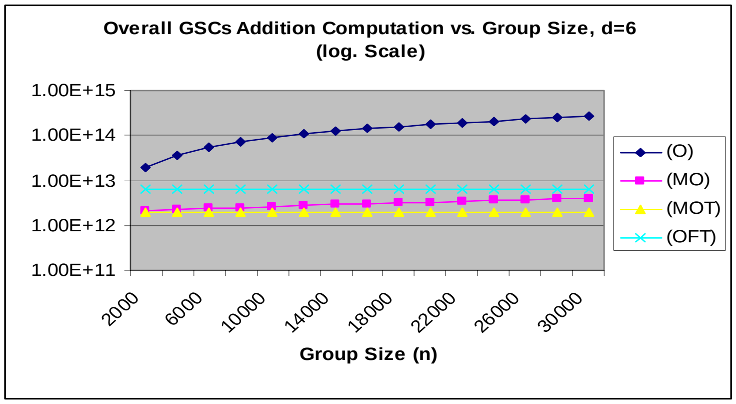
Figures 9,10: Deletion Communication for all protocols vs. group size. *d*=4 and *d*=6.



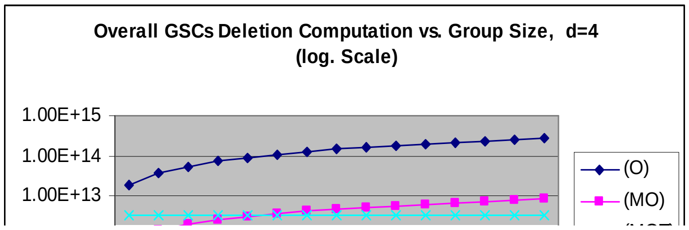


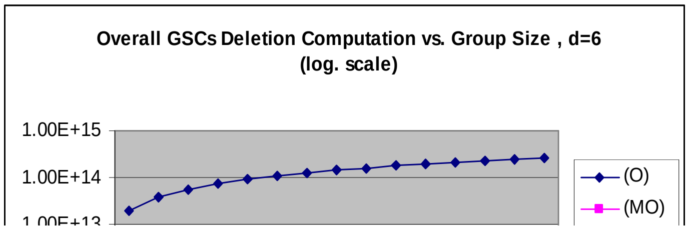
Figures 11, 12: Overall GSCs Initial Computation for all protocols vs. group size. *d*=4, *d*=6.





Figures 13, 14: Overall GSCs Addition Computation for all protocols vs. group size. *d*=4, *d*=6.





Figures 15, 16: Overall GSCs Deletion Computation for all protocols vs. group size. *d*=4, *d*=6.

VIII. SUMMARY AND CONCLUSIONS

The paper discusses the framework and the constraints under which already existing protocols can become scalable and robust in the demanding environment of MANETs. It distinguishes protocols in two families (contributory/non-contributory), discusses their limitations in this environment, and suggests solutions to render protocols scalable and robust. We present two novel hybrid protocols, (MO) and (MOT) based on the original 2^d-Octopus (O). All three of them are described in detail and cost functions in terms of communication and computation are derived for all operations. We developed (MO) and (MOT) as an attempt to make contributory protocols scalable in MANETs. Our objective was to make them efficient as well. By the performance evaluation we conducted, we saw that they outperform the original (O) in most of the cases, and what is more, (MOT) can come close in terms of performance to the very efficient non-contributory (OFT) protocol particularly in the computation costs, under certain combinations of the parameters. The performance of (MOT) concerning the communication overhead in the case of addition/deletion is the one closest to OFT and this is a great achievement for (MOT) in a MANET environment where the topology of nodes changes frequently and fast.

REFERENCES

[1] Klaus Becker, Uta Wille. Communication Complexity of Group Key Distribution,. In Proc. 5th ACM Conference on Computer and Communications Security, pages 1-6, San Francisco, CA USA, November 1998. ACM Press.

[2] Steiner M., Tsudik G., Waidner M., Diffie-Hellman. Key Distribution Extended to Groups. 3rd ACM Conference on Computer and Communication Security, ACM Press, 1996.31-37

[3] Maarit Hietalachti. Key Establishment in Ad-Hoc Networks. Helsinki University of Technology. Laboratory for Theoretical Computer Science.

[4] Adrian Perrig. Efficient Collaborative Key Managements Protocols for Secure Autonomous Group Communication.Carnegie Mellon University.

[5] N.Asokan and Philip Ginzboorg. Key-Agreement in Ad-Hoc Networks. Elsevier Preprint.2000.

[6] Maria Striki, John S. Baras. Efficient Scalable Key Agreement Protocols for Secure Multicast Communication in MANETs.

[7] David McGrew. Alan T. Sherman. Key-Establishment in Large Dynamic Groups Using One-Way Function Trees., May 1998

[8] H. Harney, E.Harder. Logical Tree Hierarchy protocol. Internet Draft, Internet Engineering Task Force, April 1999.

[9] H.Harney, C.Muckenhirn. Group Key Management Protocol (GKMP). Specification/ Architecture, Internet Engineering Task Force. July 1997.
[10] Oded Goldreich, Shafi Goldwasser, Silvio Michali. How to construct random functions. Journal of the ACM, 33(4):792-807, October 1986
[11] Rivest, Ronald L., "The MD5 Message-Digest Algorithm," Request for Comments (RFC) 1321 (1992)
[12] Adrian Perrig, Dawn Song, J.D. Tygar. ELK, a new Protocol for Efficient Large-Group Key Distribution
[13] Yongdae Kim, Adrian Perrig, Gene Tsudik. Simple and Fault Tolerant Key Agreement for Dynamic Collaborative Groups.
[14] Wade Trappe, Lawrence C. Washington. Introduction to Cryptography with Coding Theory. Prentice Hall, 2002.
[15] Network Security: PRIVATE Communication in a PUBLIC World. Author: Charles Kaufman. R. Perlman, M. Speciner.
[16] Secure Data Networking. Michael Purser.