The Performance of Group Diffie- Hellman Paradigms

Kieran S. Hagzan  
Rochester Institute of Technology  
Computer Science Department  
ksh4250@cs.rit.edu

Hans- Peter Bischof  
Rochester Institute of Technology  
Computer Science Department  
hpb@cs.rit.edu

Abstract

Decentralized administration, limited resources, and dynamic topologies are characteristics driving the development of innovative solutions to data security problems in mobile ad-hoc networks. The many solutions presented to any particular problem must be comparatively analyzed to determine the best approach in practice. This work describes the methods and experimental results achieved under an implementation and comparison of three Diffie-Hellman based shared-key agreement protocols. Linear, Tree-Based, and Hypercubic Diffie-Hellman were implemented on top of a common broadcast-based I/O framework, and the experimental running-time results recorded. The results indicate that complex but mathematically scalable algorithms are rarely beneficial for small networks of 75 to 100 devices or less, but are crucial when securing larger ad-hoc networks.

1. Introduction

Network protocols often assume the availability of a point-to-point (PTP) two-party communication medium in preference to pure broadcast-based solutions. PTP protocols are well-suited for statically arranged topologies and lend to centralized administration, utilizing algorithms that typically generate one result between exactly two parties. In an ad-hoc environment, only a broadcast medium can be assumed due to the mobile nature of the devices forming the network. Since all I/O over these networks is broadcast, either existing algorithms must be altered to generate one result between an arbitrary number of parties, or completely new solutions must be devised. Data security algorithms fit into this categorization, and special care is taken to preserve scalability for their use mobile and wireless devices.

In 1976, W. Diffie and M.E. Hellman [1] introduced a two-party algorithm designed to generate a shared (private) key over an insecure communication medium. Since the medium is assumed to be insecure, the algorithm is well-suited for broadcast-based as well as PTP networks. Various modifications of the original Diffie-Hellman algorithm exist extending the two-party concept to groups, each utilizing specialized topology arrangements combined with a custom-tailored method of traversal. This topology and traversal combination defines not only the connectivity within the network, but also an upper-bound on the computational complexity of the algorithms themselves. To determine if this theoretical bound is achieved in practice, three Diffie-Hellman variants of differing complexities were implemented, and their running time comparatively analyzed. The following sections describe the three algorithms, their methods of implementation, and the results achieved for each.

2. Background

2.1 Two-Party Diffie- Hellman

The original Diffie-Hellman algorithm relies on the properties of modular exponentiation to generate group keys. The algorithm assumes that “public” values are known to all parties. Let
Two parties, A and B choose random values $X_i$ and $X_a$ respectively from the set of integers \{1,2,3, ... $p$\}. These values $X_i$ and $X_a$ are held secret (private), and the values:

$$Y_i = g^{x_i} \pmod{p}$$

are made public.

**Party A computes:**

$$K_{AB} = Y_a^{x_i} \pmod{p} = (g^{x_a})^{x_i} \pmod{p} = g^{x_a x_i} \pmod{p}$$

**Party B computes:**

$$K_{AB} = Y_b^{x_i} \pmod{p} = (g^{x_b})^{x_i} \pmod{p} = g^{x_b x_i} \pmod{p}$$

To determine a key $K_{AB}$, the only public values known to a cryptanalyst are $Y_i$ and $Y_b$. To determine the value of the key, the values of either $X_i$ or $X_b$ must be determined from $Y_i$ or $Y_b$. Therefore, the security of the generated keys is contingent upon the difficulty of computing the inverse of modular exponentiation (the discrete logarithm) over a finite field (GF($i$)). Many current systems in cryptography rely on the difficulty of computing discrete logarithms; if the computation of discrete logarithms should become computable in polynomial time, these systems will easily be broken.

### 2.2 Linear Group Diffie-Hellman

By the properties of exponentiation, an iterative application of the original Diffie-Hellman algorithm may be used to generate a group shared key over a vector or list topology. Originally presented by Yair Amir, Cristina Nita-Rotaru, Yongdae Kim, and Gene Tsudik [2], the Linear Group Diffie Hellman (LGDH) algorithm proceeds as follows:

Let:

- $p$ be a large prime
- $g$ be a fixed primitive element of GF($p$)
- $N_0, N_1, \ldots, N_I$ represent the topology
- $N_0$ and $N_I$ engage in the typical two-party Diffie-Hellman protocol to compute key $K_{0,1}$. Node $N_i$ then uses the value of $K_{0,1}$ as the value of $X_i$ within a typical two-party exchange with $N_{i+1}$, computing key $K_{0,1,i}$. For instance:

- **Node $N_2$ computes**
  
  $$K_{0,1,2} = Y_1^{x_2} \pmod{p} = (g^{x_0})^{x_2} \pmod{p} = (g^{x_0 x_2})^{x_2} \pmod{p} = g^{x_0 x_1 x_2} \pmod{p}$$

- Generally, to compute a key for a group of $n$ members where $(n > 2)$, each member must compute

  $$K_{0,1,\ldots,n-1} = (g^{x_0 \cdots x_{n-1}}) \pmod{p}$$

Therefore, the total number of two-party Diffie-Hellman exchanges to generate a key for a group of size $n$ is $(n-1)$, each member performing on average $(n-1)/2$ exchanges.

### 2.3 Tree-Based Group Diffie-Hellman

Yongdae Kim, Adrian Perrig, and Gene Tsudik [3] introduced a Diffie-Hellman variant that utilizes the hierarchy of a binary tree to minimize the total number of two-party exchanges when computing a group key. Tree-Based Group Diffie-Hellman (TGDH) itself has been modified to represent a more generic notion of “key-trees” in cryptography.

The general topology arranges the individual members of the group at the leaves of a binary tree, non-leaf (intermediary) nodes are used for key management and do not represent any individual member. The tree is represented as an array, such that parent nodes at index $i$ have children nodes at indices $2i+1$ and $2(i+1)$. The root of the tree is at the topmost level (denoted 0.), and the leaves are at the lowest level $h$. Each node of the tree is represented by $(l,v)$ where $l$ is its level in the tree and $v$ is the index of this node in level $l$.

Let:

$$f(k) = g^v \pmod{p}$$

which is analogous to the Diffie-Hellman protocol. Each level $i$ of the binary tree holds at most $2^i$ nodes, and each node is associated
with the key \( K_{(i, v)} \) and the blinded key

\[
BK_{(i, v)} = f(K_{(i, v)})
\]

To compute \( K_{(i, v)} \) for a particular node \( n \), the knowledge of the key of one child of \( n \) and the blind key of the other child is necessary. The key \( K_{(0, 0)} \) at the root node is the group key shared by all members. In general, any key \( K_{(l, v)} \) is computed recursively as follows:

\[
K_{(l, v)} = (BK_{(l+1, 2v+1)})^{K_{(0, 1)}(mod \, p)}
= (BK_{(l+1, 2v)})^{K_{(0, 1)}(mod \, p)}
= g^{K_{(l+1, 2v)}}(mod \, p)
= f(K_{(l+1, 2v)}, K_{(l+1, 2v+1)})
\]

The total number of two-party Diffie-Hellman exchanges to generate a key for a group of size \( n \) is \((n-1)\), each node performing on average:

\[
\text{ceiling}(\log_2(n))
\]

two-party exchanges.

### 2.4 Hypercubic Group Diffie-Hellman

Hypercubic Group Diffie-Hellman (HGDH) was originally presented by C. Becker and U. Wille [4] as an attempt to reduce the communication complexity of group key distribution. The topology exploits a binary-reflected Gray Code ordering of the nodes to perfectly store the representation of a hypercube within a vector or list. Using this Gray Code ordering of the nodes also provides a guaranteed Hamiltonian cycle on a \( d \) dimensional hypercube. For a hypercube of \( n \) nodes represented as a set \([N_0, N_1, ..., N_n] \):

\[
d = \text{ceiling}(\log_2(n-1))
\]

In this arrangement, each node is connected to \( d \) others, and each node has a unique \( d \) bit address, such that the address of 2 nodes connected by an edge in the \( j^{th} \) dimension differ only in their \( j^{th} \) bit.

Since the hypercube is a binary structure, there are two cases to consider in the key-agreement algorithm. If the hypercube is of “perfect” size (the number of nodes in the hypercube is an exact power of 2) then each node \( N_i \) proceeds through \( d \) rounds, each round \( d_i \) performing an exchange with the node located at:

\[
i(+) \, 2^{l-1}
\]

where \((+)\) denotes the exclusive- or binary operation. If the hypercube is not of “perfect” size, than the nodes in the outermost dimension (the nodes where the most significant bit of their Gray Code ordering is set) perform only the \( d_0^{th} \) exchange to “synchronize” with the perfect innermost dimension. Those nodes of innermost dimension who have no connection to a node in the outer dimension proceed as normal. In this protocol, each node performs at most \( d \), or:

\[
\text{ceiling}(\log_2(n))
\]

exchanges, and a maximum of:

\[
(n \ast \text{ceiling}(\log_2(n))) / 2
\]

exchanges are performed in total to generate a group key of size \( n \).

### 3. Methods

#### 3.1 Hardware

The computing platform used for the testbed included 137 Sun SparcII machines running between 440 and 650mhz, containing between 256 and 512 MB of memory. The nodes are interconnected via a 100Mbps fully switched Ethernet LAN spanning a single subnet. The testbed is a public-domain computing platform, thus result sets were collected during peak and non-peak hours of use, and the individual sets averaged to gain accurate data measurements.

#### 3.2 Network Input/Output Protocol

To simulate the performance of broadcast networks, strict IP addressing was not used for data I/O. Rather outbound packets are destined to the network broadcast address at the IP level regardless of destination. Nodes are identified by a random 128-bit number generated on startup, and the broadcast address is denoted 0. This identifier is included in network “son of data
encapsulation” headers to provide PTP over broadcast communications through filtering upon message reads. This introduces the small possibility \((n / 2^{128})\) for a group of size \(n\) that two nodes choose the same ID, however with a testbed of 137 nodes, the probability of this occurring is reasonably small \((4 \times 10^{-37})\).

A reactor-based I/O pattern was utilized to implement a common framework supporting any Diffie-Hellman based topology and traversal mechanism where the topology can be stored in an array or list. I/O was multiplexed over distinct socket channels by the “select()” system call. Network message headers contain the ID of the source node, the ID of the destination node (or 0 if a true broadcast message,), and a message identifier. The framework reads a broadcast message from the network, filters the message based upon first destination ID and then message type, and messages of unknown type or invalid address are dropped. The framework splits key agreement paradigms into two modules, the topology management module and the key-agreement module.

Messages of a type destined for the topology management module are those messages that affect the structure of the ad-hoc network itself. These include messages that facilitate nodes joining and leaving the group, and the combining and partitioning of whole groups. If a key-agreement is in process, additions and voluntary subtractions from the group are dropped until the key is generated.

Messages of a type destined for the key agreement module are those messages that directly relate to the formation of a group key. Typically, these messages are the broadcasting of public keys and public-key structures, and the initiation and conclusion of Diffie-Hellman protocols. Messages of this type supersede any other messages in processing priority allocated, and thereby halt voluntary changes in the network topology during key agreement.

Devices are assumed to be fault-tolerant, that is no failure detection is performed to maintain network topology when devices leave the network unannounced. The network is also assumed to be reliable, that is no data-reliability protocol has been implemented to deal with dropped packets.

3.3 LGDH Implementation Modifications

Each exchange in the original LGDH algorithm requires two broadcast messages for each of the public keys used in that particular exchange. However mathematically, it is only the public key of the leftmost node in any particular exchange that contributes to the final key, so any broadcasts from the rightmost node can be discarded. This technique is known as the elimination of backward propagation.

All nodes except that at position 0 in the array wait until their left neighbor’s public key is broadcast. Upon receipt of their left neighbor’s key, a node will compute its own shared key and broadcast that value. This process continues until the rightmost node in the storage array has broadcast their public key. This process eliminates the possibility of deadlock within the protocol, since each broadcast literally “pushes” the protocol one node further until the array of nodes is exhausted.

3.4 TGDH Implementation Modifications

The implementation of TGDH assumes that rightmost leaf nodes will wait until their left sibling has broadcast their public key. At this point, right siblings compute the lowest-level blind keys and promote themselves to be their parents, setting the public key of their left sibling in the storage array as well as their own without broadcasting it. The process recursively continues until the rightmost leaf node has assumed responsibility of the root node, at which point it will broadcast the entire tree of public keys it has collected. This final broadcast reduces the number of public-key broadcasts by approximately half.

3.5 HGDH Implementation Modifications

Since the hypercube is mapped by a Gray Code, there are no intermediary nodes and no protocol “waste.” Since a Hamiltonian cycle is mapped over the hypercube, HGDH is intended to be a waste-less protocol, both in terms of computational complexity and
network communication. When a node \( n \) computing a group key in dimension \( d_i \) will initiate the exchange with its partner in dimension \( i \) if the \( i \)th bit representing node \( n \)'s position in the storage array is set. Otherwise, a node will wait until its partner in that dimension has broadcast its public key.

4. Results

Each of the Diffie-Hellman variants implemented was run until such point that all 137 nodes in the hardware testbed were part of one large group. This process was repeated 100 times for each algorithm during different times of day to accommodate for various levels of overall CPU load. The 100 result sets for each algorithm were then averaged together to normalize the results.

4.1 LGDH Results

Since LGDH is an algorithm bounded by linear complexity, a linear regression was performed upon the averaged result set. The indications of that regression are shown in Figure 1. below.

![LGDH Results](image)

The results indicate that a practical upper bound on the running time \( t \) of LGDH for a group of \( n \) members is

\[ t = 0.015002n + 0.15566 \]

seconds.

4.2 TGDH Results

The time complexity of TGDH is bounded by the height of the binary storage tree, which is logarithmic in proportion to the number of leaves or member count of the tree. A logarithmic regression was performed on the averaged result set for the TGDH algorithm, indicated in Figure 2 below.

![TGDH Results](image)

The results indicate that a practical upper bound on the running time \( t \) of TGDH for a group of \( n \) members is

\[ t = 0.25104 \log_2(n) - 0.27517 \]

seconds.

4.3 HGDH Results

The time complexity of HGDH is bounded by the degree of the hypercube, or the number of bits needed to represent the value \((n-1)\) for a group of \( n \) members, which is logarithmic in proportion to \( n \). A logarithmic regression was performed on the averaged result set, indicated in Figure 3.
The results indicate that a practical upper-bound on the running time $t$ of HGDH for a group of $n$ members is 

$$t = 0.31763 \log_2(n) - 0.43708$$

seconds.

4.4 Comparative Results

The combination of these three result sets can be noted in Figure 4.

The combined results in Figure 4. illustrates the following relationship between the three algorithms:

$$t(LGDH) = 0.015002 \ n + 0.15566$$
$$t(TGDH) = 0.25104 \ \log_2(n) - 0.27517$$
$$t(HGDH) = 0.31763 \ \log_2(n) - 0.43708$$

5. Discussion

When the results are comparatively analyzed, the first observation to make is that the algorithms that are logarithmic in complexity performed worse than the linear algorithm for small groups. Oddly, this is the case even though the number of total exchanges for the logarithmic algorithms across the entire group is provably always less than or equal to that of the linear algorithm. Since all three algorithms use the exact same code base to perform network I/O, the only difference between them is literally the topology and key-agreement traversal mechanism combination. Also, the averaging of 100 runs for each algorithm combined with final normalization via regression causes this result set to reflect a good sample of the performance of these algorithms on this particular testbed.

Upon further investigation, the major difference found between the three algorithms is the amount of process-level parallelism that occurs across the network. Considering LGDH, at most two nodes are active in a two-party Diffie-Hellman exchange at any given time. All leaf nodes are initially active for TGDH, but then half of them (the left siblings) become inactive every recursion toward the root. In the HGDH algorithm, all nodes are active at all times until the group key is computable.

Taking this further, when two processes interact, one process may not be ready to interact with the other yet, and may ask the other process to wait for notification. Consider TGDH, where a left child may intend to initiate an exchange with its right sibling, but that sibling is actually an intermediary node whose representative member is active in another exchange. That leftmost child must wait until the right sibling is ready. The more process-level parallelism that occurs, the higher probability that processes will need to wait for each
other due to differences in processor speed, network latencies, and processor availabilities between any two given nodes. LGDH experiences no latency due to parallelism. TGDH suffers from a moderate amount, and all nodes within an instance of HGDH experience some process-level parallelism latency.

6. Conclusions

Since the sizes of mobile ad-hoc networks typically contain less than 100 nodes, the ability to eliminate backward propagation and the total lack of parallelism latency causes plain Linear Group Diffie-Hellman (LGDH) to perform well in practice. For larger networks of this type however, LGDH becomes unusable due to the amount of total network traffic and per-node computation experienced. For larger networks, TGDH is attractive due to the combination of lower parallelism latency and the ability to reduce backward propagations by half. HGDH is a mathematically elegant solution to the problem, but the physical constraints of mobile computing cause the theoretical results differ from those experienced in practice.

Communication and computational complexity are the main considerations when modifying the Diffie-Hellman protocol to suit a particular topology, but latencies from environmental conditions including (but not limited to,) processor availability and parallelism latency should always be considered as well.

7. References


