The Design of a Group Key Agreement API

Giuseppe Ateniese‡, Olivier Chevassut‡, Damian Hasse‡, Yongdae Kim‡, Gene Tsudik‡

August 25, 1999

Abstract

As collaborative applications become more and more popular the need for appropriate security mechanisms becomes apparent. This paper describes a protocol suite and API geared for securing collaborative applications. The API is based on the extensions of Diffie-Hellman key agreement developed in the CLIQUES project. Its core services provide authenticated group key agreement in relatively small and dynamic peer groups.

1 Introduction

Securing group communications is a complex problem posing a number of challenges which range from basic algorithms, systems and communication design, to secure implementation [1]. The two common group security concerns are privacy (assuring that intra-group communication remains secret to non-members) and authentication (assuring that legitimate group members can be identified as such).

The standard approach to supporting group security is based on maintaining a secret quantity known to all group members and no other party. The particulars of generating and distributing this secret quantity are known collectively as group key establishment. When the latter is achieved with one party generating a group secret, the problem is reduced to group key distribution. Whereas, if all group members collectively generate the group secret, the problem is referred to as group key agreement. In both cases, only current group members must have access to the group secret. We say current since group membership can be highly dynamic. Whenever members join or leave a group, there must be means for securely adjusting the group secret.

This paper discusses the design of a group key agreement application programming interface (API). This API, called CLQ_API, is based on the CLIQUES protocol suite for key agreement in dynamic peer groups. (Peer groups are relatively small, non-hierarchical groups typically geared for replication or collaborative applications.) CLIQUES protocols [2, 3, 4], in turn, are the group extensions of the well-known Diffie-Hellman key exchange [5]. CLIQUES provides authenticated contributory key agreement which guarantees key independence, key confirmation, perfect forward secrecy, and resistance to known key attacks.¹

CLQ_API separates cryptography and communication, i.e., the actual group communication is left to the underlying communication subsystem (preferably a reliable group communication system, e.g., SPREAD[7], TOTEM[8] or TRANSIS[9]). Also, network events such as network partitions, failures and other abnormalities are assumed to be taken care by the communication system. This approach allows the crafting of a small, concise and communication-independent API.

The rest of this paper is organized as follows. We begin with the notation and brief descriptions of basic operations in group key agreement. We then describe the CLIQUES protocol suite, which, in turn, consists of the following group operations: join, merge, leave and key refresh. Next, CLQ_API primitives are described in detail along with some examples. Finally, we discuss the efficiency of CLQ_API and conclude with some experimental results obtained with several popular cryptographic packages. (A more detailed description of the API is included in the Appendix.)

¹IBM Research Laboratory, Rüschlikon, Switzerland; gat@zurich.ibm.com
²Lawrence Berkeley National Laboratory, University of California; chevassu@george.lbl.gov
³Computer Networks Division, USC Information Sciences Institute; {hasse, yongdae, gts}@isi.edu Research supported by the Defense Advanced Research Project Agency, Information Technology Office (DARPA-ITO), under contract DABT63-97-C-0031.
⁴For the definition of the above, the reader is referred to [6].
2 Group Key Agreement

2.1 Notation

The following notation is used throughout this paper:

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>number of protocol parties (group members)</td>
</tr>
<tr>
<td>$i, j$</td>
<td>indices of group members</td>
</tr>
<tr>
<td>$M_i$</td>
<td>$i$-th group member; $i \in [1, n]$</td>
</tr>
<tr>
<td>$G$</td>
<td>unique subgroup of $\mathbb{Z}_p^*$ of order $q$ with $p, q$ prime</td>
</tr>
<tr>
<td>$q$</td>
<td>order of the algebraic group</td>
</tr>
<tr>
<td>$g$</td>
<td>exponentiation base; generator in group $G$</td>
</tr>
<tr>
<td>$x_i$</td>
<td>long-term private key of $M_i$</td>
</tr>
<tr>
<td>$N_i$</td>
<td>$M_i$’s session random number $\in \mathbb{Z}_q$</td>
</tr>
<tr>
<td>$S_n$</td>
<td>group shared key among $n$ members</td>
</tr>
<tr>
<td>$K_{ij}$</td>
<td>long-term shared secret between $M_i$ and $M_j$, $i \neq j$</td>
</tr>
<tr>
<td>$</td>
<td>a</td>
</tr>
<tr>
<td>$H$</td>
<td>output size of hash function</td>
</tr>
<tr>
<td>$\text{inv}(a, b)$</td>
<td>multiplicative inverse of $a$ modulo $b$</td>
</tr>
</tbody>
</table>

2.2 Group key agreement operations

A comprehensive group key agreement API must handle adjustments to the group secret stemming from single and multiple member membership changes. This section describes the purpose of each of these operations.

2.2.1 Single member operations:

these include single member additions or deletions. The former occurs when an entity wants to join a group and the latter occurs when one member wants to leave (or is forced to leave) a group. Both operations may be performed by the group controller(s) or by consent of every group member, depending on the local policy.

2.2.2 Multiple member operations:

these also include addition and deletion. The former includes:

- Mass join: multiple disparate new members can be brought into an existing group
- Group fusion: two or more groups may be merged into a single group.

The latter includes:

- Mass leave: multiple members must be removed at the same time.
- Group division: a monolithic group needs to be broken in smaller groups.

2.2.3 Key refresh

has two main purposes:

- Limit the amount of ciphertext generated with the same key
  Since it is easier to perform cryptanalysis with more ciphertext/plaintext pairs, a routine key refresh operation is needed. The lifetime of a key is left up to the application policies.
Recover from the compromise of the current group secret or a member’s contribution. (We note that a compromise of a member’s contribution can result in disclosure of all group secrets contributed to by this member. Therefore, not only the group shared keys, but also the individual key shares must be periodically refreshed.)

3 CLIQUES protocol description

3.1 Overview

As mentioned above, CLIQUES is a protocol suite providing authenticated contributory key agreement for dynamic peer groups.

CLIQUES supports these ones by the following operations:
- Join: a new member is added to the group.
- Merge: one or more members are added to the group.
- Leave: one or more members are removed from (or leave) the group.
- Key Refresh: generates a new group shared key.

Each operation is discussed in the rest of this section.

3.2 Notes

For simplicity, in CLIQUES protocols, the group controller is the last member who has joined the group. This behavior can be easily changed if required, allowing any member to become the controller at any time.

In the following operations, the group secret has the form $S_n = g^{N_1\cdots N_n}$, when $N_i$ is provided by $M_i[2]$. Similarly, the last broadcast message is the set formed by $g^{K_{i,n}N_1\cdots N_n/N_i}$ for all $i$ in $[1, n]$. This set should be kept by each member of the group individually\(^2\). Although the entire group receives this message from the controller, each member uses a different element to compute the new group secret.

Finally, the computation of the long term key, $K_{ij}$, shared between any two members $M_i$ and $M_j$ is assumed to have been done before the its use is required.

3.3 Join

The join operation adds a new member, $M_{n+1}$, to the current group of $n$ members. During this process a new group shared key, $S_{n+1}$, is computed and $M_{n+1}$ becomes the new group controller. Assuming that $M_n$ is the current controller, then the protocol runs as follows:

1. $M_n$ generates a new exponent $N'_n$. Then, $M_n$ produces the following set:

$$M = \{g^{N_1\cdots N'_n/N_i} \mid i \in [1, n-1]\}^3 \cup \{g^{N_1\cdots N_{n-1}}\} \cup \{g^{N_1\cdots N_n}\}$$

Finally, $M$ is sent to $M_{n+1}$.

2. Upon reception of the message, $M_{n+1}$ generates a new exponent $N_{n+1}$ and computes $g^{K_{i,n+1}N_1\cdots N'_nN_{n+1}/N_i}$ for all $i$ in $[1, n]$. Then, this set is broadcasted to the entire group.

3. Upon reception of the broadcast, each $M_i$ computes the shared group key as follows:

$$g^{N_1\cdots N'_nN_{n+1}/N_i} = S_{n+1}. \text{ Similarly, } M_{n+1} \text{ computes the new group shared key } g^{N_1\cdots N'_nN_{n+1}} = S_{n+1} \text{ using the message from step 1.}$$

Steps 1 and 2 require $n$ modular exponentiations. Similarly, step 3 requires 1 exponentiation by each member. Hence, the number of serial\(^4\) exponentiations is $2n + 1$.

\(^2\)This is necessary since in the future any member can become the controller again.

\(^3\)This is achieved by exponentiating $N'_n \ast \text{inv}(K_{i,n}, p) \ast \text{inv}(N_n, q)$ to each element in the last broadcast message.

\(^4\)See section 5 for definition
3.4 Merge

The merge operation is used to add \( k > 0 \) members to the current group of \( n > 1 \) members. Let \( m = n + k \). During this process a new group shared key, \( S_m \), is computed and \( M_m \) becomes the new group controller. Assuming that \( M_n \) is the current controller, then the protocol runs as follows:

1. \( M_n \) generates a new exponent \( N_n' \) and computes \( g^{N_1 \ldots N_{n-1} N_n'} \). Then this message is sent to \( M_{n+1} \).

2. Each member \( M_j, j = n + 1, \ldots, m - 1 \), generates a new exponent \( N_j \) and computes \( g^{N_1 \ldots N_{n-1} N_n \ldots N_j} \). Then this value is sent to \( M_{j+1} \).

3. Upon reception of the message, \( M_m \) broadcasts this value to the entire group.

4. Upon reception of the broadcast, each member \( M_i, i = 1, 2, \ldots, m - 1 \), computes \( g^{N_1 \ldots N_{n-1} N_n \ldots N_m} / N_i \) and sends it to \( M_m \).

5. \( M_m \) generates a new exponent \( N_m \) and produces the following set:

\[
\mathcal{M} = \{g^{K_{im}N_1 \ldots N_{n-1} N_n \ldots N_m / N_i} \mid i \in [1, m-1]\}
\]

Finally, \( \mathcal{M} \) is broadcasted to the entire group.

6. Upon reception of the broadcast, \( M_i, i = 1, 2, \ldots, m - 1 \), computes the group shared key as follows\( (g^{N_1 \ldots N_{n-1} N_n \ldots N_m})^{K_{im}} N_i = g^{N_1 \ldots N_n \ldots N_m} = S_m \). Similarly, \( M_m \) computes the group shared key \( g^{N_1 \ldots N_n \ldots N_m} = S_m \) using the broadcast message in step 3.

If \( k = 1 \), step 2 is not required, and the rest of the protocol runs as above.

Steps 1 and 2 require a total of \( k \) modular exponentiations. Similarly, steps 4 and 6 need one by each member. Finally, step 5 needs \( n + k - 1 \) modular exponentiations. Hence, the number of serial exponentiation is \( n + 2k + 1 \) to add \( k \) members.

3.5 Join vs. Merge

Similar to the merge operation, join can be used to add \( k \) members to a group. This task can be accomplished by repeating the join operation \( k \) times or by extending it to mass join\(^6\). Using these procedures the numbers of modular exponentiations becomes \( 2nk + k^2 \) and \((k+1)(2n+k)/2\). In contrast \( n + 2k + 1 \) exponentiations are required by merge.

On the other hand, the join operation has none reverse broadcasts\(^7\) messages, in contrast to one required by merge. Hence, \( m - 1 \) unicast messages are sent to \( M_m \), in step 4 of merge. Although, this action might lead to congestion at \( M_m \), it might be preferred since each member agrees in the operation.

In conclusion, the merge operation requires significantly less exponentiations than join at the expense of a reverse broadcast. This behavior might be preferred depending on the policy of the application using CLIQUES.

3.6 Leave

The leave operation removes \( k \) members from the current group of \( n \) members. During this process a new group shared key, \( S_{n-k} \), is computed. Similarly, \( M_{n-k} \) might become the new controller, if \( M_n \) the current one leaves the group. For simplicity, suppose that only one member, \( M_d \), leaves the group. The protocol runs as follows:

\(^5\)This is achieved by modular exponentiating the previous group shared key to \( N_n' = inv(N_n, q) \).

\(^6\)Mass join can be performed like the merge operation.

\(^7\)See section 5 for definition.
1. $M_n$ generates a new exponent $N'_n$ and produces the following set:

$$\mathcal{M} = \{g^{K_n N_i/N_i'} | i \in [1, n-1] \text{ and } i \neq d\}^8$$

Finally, $\mathcal{M}$ is broadcasted to the entire group.

2. Upon reception of the message, each $M_i$ computes $g^{N_i/N_i'} = g^{N_i} = S_n$. $M_n$ computes the new group shared key $g^{N_i/N_i'} = S_n$ using the old one.

The member $M_d$ cannot compute the new group shared key, since the group controller does not generate $M_d$’s partial key, $g^{K_{i_n} N_i/N_i'}$. If several members leave the group, the group controller simply does not compute the leaving member’s partial key in step 1. The rest of the protocol runs as above.

If the current controller, $M_n$, leaves the group, then the last member $M_{n-1}$ still in the group performs the above operation. Moreover, since the new controller cannot remove the long term keys from the old one, each member should recalculate its session random number as follows, $N_i = N_i \ast inv(K_{in}, q)$ before performing step 2.

Step 1 requires $n - k$ exponentiations. Similarly, step 2 needs one by each member. Hence, the leave operation requires a total of $n - k + 1$ modular exponentiations.

### 3.7 Key refresh

The key refresh operation updates the current group shared key, $S_n$. The usage of it depends on the policy of the application using CLIQUES. This operation should perform the same procedure as leave does when $k = 0$. Basically, in step 1 of the leave operation, $M_n$ generates the following set

$$\mathcal{M} = \{g^{K_n N_i/N_i'} | i \in [1, n-1]\}$$

for all the members in the group.

### 4 CLQ_API

#### 4.1 Overview

CLQ_API is a light-weight group key management API based on CLIQUES protocol. In conjunction with a reliable and sequenced group communication system, members agree on an authenticated group shared key. The group communication layer(Fig. 1) should provide reliable message delivery functions such as join, leave, partition, and node failures.

![Figure 1: Communication layer and CLQ_API](image)

The properties of CLQ_API are as follows:

1. CLQ_API is small, concise, generic and easy-to-use API. It contains only eight function calls.

2. We assume that the communication layer provides support for all types of network faults. CLQ_API does not deal with partitions and other network abnormalities.

3. CLQ_API is independent from the communication system, i.e., any of the popular ones such as SPREAD, TOTEM, and TRANSIS might be used.

---

8This is achieved by exponentiating each element in the last broadcast message to $N'_n \ast inv(N_n, q)$. 

5
4.2 Required function calls for CLQ-API

In this section, eight function calls are briefly described. Each of them represents one or more steps in a group operation presented in section 3. Details of data structures, definitions and secondary function calls are briefly explained in the appendix.

The following terms will be used throughout this section. explained:

- **Context**: It contains the information about each user required by CLQ-API.
- **Epoch**: This is the message number of a token. This can be used to prevent the replay attack along with time stamp.
- **Time stamp**: This is the time when the token is sent. This can be used to expire a protocol.
- **Token**: It includes epoch, time stamp, and message.

The discussion of function calls follows:

- **clq_proc_join**: It performs step 1 of the join operation. The controller calls this function to hand over information about the current group to a new member, who will eventually become the new controller. The main purposes of this function are:
  - To generate a new session random number for the current controller.
  - To remove long term keys and previous session random number from the partial keys of all users.
  - To add the new session random number to the partial keys of all users.

- **clq_join**: It performs step 2 of the join operation. The new group member calls this function using the token received from the current controller. The main purposes of this function are:
  - To generate a new session random number for the new member.
  - To generate long term keys between the new member and each user.\(^9\)
  - To compute the partial keys for other group members.

- **clq_update_ctx**: It performs the last step of the join, merge, leave, and key refresh operation. Every member calls this function in order to update the group shared key upon reception of the token sent by the new controller.\(^10\) Main purpose of this function is to compute group shared key using the user’s session random number and the token from the controller.

- **clq_update_key**: It performs step 1 and 2 of the merge operation. This function is called by the current controller and by all of the new members but the last one to add their session random number to the group shared key.

  The main purposes of this function are:
  - To generate new session random number\(^11\).
  - To add new session random number to the group shared key.

---

\(^9\)It can be done before calling this function.

\(^10\)In case of leave and refresh operation, the current controller.

\(^11\)If the controller calls this function, then the previous session random number should be removed.
• clq\_factor\_out: It performs step 4 of the merge operation. Every group member calls this function to factor out its own session random number from the group shared key.

• clq\_merge: It performs step 5 of the merge operation. The last new member calls this function to add a new session random number to the received partial keys from the group members.

• clq\_leave: It performs step 1 of the leave operation. Every group member calls this function right after one or more members left the group. The main purpose of this function is to remove the information of the leaving members. If the controller calls this function, then two more steps need to be performed:
  - To generate new session random number.
  - To compute the partial keys for the other group members except for the leaving ones.
If the controller leaves the group, then the last member still in the group should perform.

• clq\_refresh\_key: It performs step 1 of the key refresh operation. The current controller calls this function, when the group shared key needs to be updated. The main purposes of this function are:
  - To generate new session random number.
  - To compute new partial keys by adding this session random number and removing old one.

4.3 Operations of CLQ\_API

Before we explain each of the group operations in the API, we need to define some message types:

- NEW\_MEMBER : It is sent by the controller to a new member, when a new member joins the group.
- KEY\_UPDATE\_MESSAGE : It is sent to every member in order to update the group shared key.
- MERGE\_KEY\_UPDATE : It is sent to every member to update the group shared key in merge operation.
- MERGE\_BROADCAST : It is broadcasted from the last new member when one or more members merge.
- MERGE\_FACTOR\_OUT : It is sent by each group member to the new controller in a merge operation.
- MASS\_JOIN : It is sent to the next new member in a merge operation.

We explain most of the group operations using the function calls in the previous section.

- Join
  
  (1) New member $M_{n+1}$ calls communication protocol join to join a group.
  (2) The current controller calls $clq\_proc\_join$ to generate a token containing NEW\_MEMBER message.
  (3) The token is sent to the new member.
  (4) The new member calls $clq\_join$ to generate a token containing KEY\_UPDATE\_MESSAGE.
  (5) The token is broadcasted to the entire group.
  (6) Every user calls $clq\_update\_ctx$ to compute the new group shared key.
• Merge(one member)

(1) New member $M_{n+1}$ calls communication protocol join to join a group.
(2) The current controller calls \texttt{clq\_update\_key} to generate a token containing \texttt{MASS\_JOIN} message.
(3) The token is sent to the new user.
(4) The new member calls \texttt{clq\_update\_key} to generate a token containing \texttt{MERGE\_BROADCAST} message.
(5) The token is broadcasted to the entire group.
(6) Upon reception of the message, each member except the last one calls \texttt{clq\_factor\_out} to generate a token containing \texttt{MERGE\_FACTOR\_OUT} message.
(7) The token is sent back to the new user.
(8) For each token, the new member calls \texttt{clq\_merge} to generate a token containing \texttt{MERGE\_KEY\_UPDATE} message.
(9) The token is broadcasted to the entire group.
(10) Every user calls \texttt{clq\_update\_ctx} to compute the new group shared key.

• Merge(several members)

(1) Set of members $M_i$ calls communication protocol join to join a group.
(2) The group controller calls \texttt{clq\_update\_key} to generate a token containing \texttt{MASS\_JOIN} message.
(3) The token is sent to the first new member.
(4) Upon reception of the token, the next user calls \texttt{clq\_update\_key} to generate a token containing \texttt{MASS\_JOIN} message.
(5) The token is sent to the next new member.
(6) Upon reception of the token, the last user calls `clq_update_key` to generate a token containing `MERGE_BROADCAST` message.
(7) The token is broadcasted to the entire group.
(8) Upon reception of the message, each member except the last one calls `clq_factor_out` to generate a token containing `MERGE_FACTOR_OUT` message.
(9) The token is sent back to the last new user.
(10) For each output token, the last new member calls `clq_merge` to generate a token containing `MERGE_KEY_UPDATE`.
(11) The token is broadcasted to the entire group.
(12) Every user calls `clq_update_ctx` to compute the new group shared key.

*Leave*

(1) A member $M_d$ calls communication protocol `leave` to leave from a group.
(2) The group controller calls `clq_leave` to generate a token containing `KEY_UPDATE_MESSAGE`.

(3) The token is broadcasted to the entire group.

(4) Upon reception of the token, every user calls `clq_update_ctx` to compute the new group shared key.

- Key refresh

![Diagram of Key refresh operation]

Figure 6: Key refresh operation

(1) The controller calls `clq_refresh_key` to generate a token containing `KEY_UPDATE_MESSAGE`.

(2) The token is broadcasted to the entire group.

(3) Upon reception of the token, every user calls `clq_update_ctx` to compute the new group shared key.

5 Efficiency

5.1 Efficiency of CLIQUES

Communication and computation costs are discussed in this section. We assume the number of current users in the group is $n$.

First, we analyze communication costs illustrated in Table 2. In this section, the following terminologies are used. Round is considered to be the number of messages exchanged among members that need to be done serially. For example, if two members send each a message to another one, this is considered as one round. On the other hand, if a generated message depends on an incoming one, this considers as two rounds. Reverse broadcast occurs when all the group members send a message to one of the participants. Maximum bandwidth is considered to be the size of the largest message sent for the operation.

Table 3 illustrates computation costs. Exponentiation is the most expensive operation as it requires $O(\log^3 p)$ bit operations in $Z_p^*$. Given $a$ and $p$, finding the inverse of $a \in Z_p^*$ requires only $O(\log^2 p)$ bit operations (using the extended Euclidean algorithm). Similarly, the multiplication of $a$ and $b$ modulo $p$ requires $O(\log^2 p)$ bit operations. See [10, 6] for more details. Hence, the speed of each operation mostly depends on the number of the number of serial exponentiations. In other words, to reduce computation overhead, we may use merge instead of join, though the first one requires reverse broadcast. Note that the cost for generating the long term keys is not included in this table.
Table 2: Communication costs

<table>
<thead>
<tr>
<th>Operations</th>
<th>Join</th>
<th>Merge</th>
<th>Leave</th>
<th>Refresh</th>
<th>k-Merge</th>
<th>k-Leave</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of users</td>
<td>n+1</td>
<td>n+1</td>
<td>n-1</td>
<td>n</td>
<td>n+k</td>
<td>n-k</td>
</tr>
<tr>
<td>Rounds</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>k+2</td>
<td>1</td>
</tr>
<tr>
<td>Broadcasts</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Reverse broadcast</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Total messages</td>
<td>2</td>
<td>n+2</td>
<td>1</td>
<td>1</td>
<td>n+2k</td>
<td>1</td>
</tr>
<tr>
<td>Maximum bandwidth</td>
<td>n</td>
<td>n</td>
<td>n-1</td>
<td>n-1</td>
<td>n+k-1</td>
<td>n-k</td>
</tr>
</tbody>
</table>

Table 3: Computation costs

<table>
<thead>
<tr>
<th>Operations</th>
<th>Join</th>
<th>Merge</th>
<th>Leave</th>
<th>Refresh</th>
<th>k-Merge</th>
<th>k-Leave</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of users</td>
<td>n+1</td>
<td>n+1</td>
<td>n-1</td>
<td>n</td>
<td>n+k</td>
<td>n-k</td>
</tr>
<tr>
<td>Serial exponentiation</td>
<td>2n+1</td>
<td>n+3</td>
<td>n-1</td>
<td>n</td>
<td>n+2k+1</td>
<td>n-k</td>
</tr>
<tr>
<td>Total exponentiation</td>
<td>3n+2</td>
<td>3n+2</td>
<td>2n-3</td>
<td>2n-1</td>
<td>3n+4k-2</td>
<td>2n-2k-1</td>
</tr>
</tbody>
</table>

5.2 Comparison of modular exponentiations

As mentioned in the previous section, the modular exponentiation requires most time in performing CLIQUES protocol suite and, therefore, reducing its computation time may enhance the efficiency of our protocol. In this section, we compare modular exponentiation speed in three different cryptographic libraries on three different computers. The target systems have the following features:

Table 4: Target platform

<table>
<thead>
<tr>
<th>Machine</th>
<th>CPU speed</th>
<th>Main memory</th>
<th>OS</th>
<th>Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ultra Sparc I</td>
<td>233 MHz</td>
<td>64 MB</td>
<td>SUN OS 5.5.1</td>
<td>gcc 2.7.2.2</td>
</tr>
<tr>
<td>Pentium I</td>
<td>233 MHz</td>
<td>48 MB</td>
<td>Linux 2.0.36</td>
<td>gcc 2.7.2.3</td>
</tr>
<tr>
<td>Pentium II</td>
<td>450 MHz</td>
<td>256 MB</td>
<td>Linux 2.2.9</td>
<td>egcs 1.1.2</td>
</tr>
</tbody>
</table>

The cryptographic libraries are summarized as follows:

- **RSAREF[11]**
  - This is a cryptographic toolkit for privacy-enhanced mail.
  - We use version 2.0 developed in 1996.

- **Crypto++[12]**
  - This is a free C++ class library of cryptographic schemes published by Wei Dai.
  - Since addition and subtraction are implemented for Pentium assembler, it performs better in Pentium than in other microprocessors.
  - We use version 3.1 developed in May, 1999.

- **OpenSSL[13]**
  - This, the successor of SSLeay[14], is a cryptographic toolkit implementing secure socket layer (SSL v.2/3)[15].
  - This library implements some basic operations by assembler on various platforms.
  - We use version 0.9.3a developed in May, 1999.
We measure the speed of modular exponentiation $y = g^x \pmod{p}$, where $p$ is a random 512 bit prime, $g$ a 512 bit generator for GF($p$) of order $q$ (160 bit), and $x$ a random 160 bit integer. Table 5 shows the comparison. Note that each time is the average value for many tests.

<table>
<thead>
<tr>
<th>Machine</th>
<th>OpenSSL</th>
<th>Crypto++</th>
<th>RSAREF</th>
</tr>
</thead>
<tbody>
<tr>
<td>UltraSparc</td>
<td>16.6 msec</td>
<td>60.2 msec</td>
<td>146.3 msec</td>
</tr>
<tr>
<td>Pentium II</td>
<td>2.5 msec</td>
<td>7.6 msec</td>
<td>22.2 msec</td>
</tr>
<tr>
<td>Pentium I</td>
<td>6.4 msec</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

OpenSSL is faster than any other libraries, which requires only 2.5 msec for one 512-bit modular exponentiation. This measurement implies that CLIQUES protocol suite can be well-suited to dynamic peer group on reasonably fast processors.

6 Conclusion

We have presented and analyzed a new group key agreement protocol suite CLIQUES.

References


Appendix: Data Structures and Definitions of CLQ_API

- **CLQ_CONTEXT**
  - Type: Structure
  - Description: This structure contains the context of a member, $M_i$, in a specific group.
  - Contents
    | Name               | Type        | Contents |
    |--------------------|-------------|----------|
    | member_name        | String      | Name of the user $M_i$ |
    | group_name         | String      | Name of the group      |
    | key_share          | $|q|\text{ bit integer}$ | $M_i$'s session random number, $N_i$ |
    | group_secret       | $|p|\text{ bit integer}$ | Current group shared key |
    | group_secret_hash  | $H\text{ bit integer}$ | Hash of group_secret |
    | group_members_list | CLQ_GML     | List of current group members |
    | first              | Pointer     | Pointer to the first member in the group_members_list |
    | last               | Pointer     | Pointer to the last member in the group_members_list |
    | me                 | Pointer     | Pointer to the member $M_i$ in the group_members_list |
    | params             | CLQ_PARAM   | Diffie-Hellman parameters |
    | key                | CLQ_KEY     | Private and public key of $M_n$ |
    | epoch              | Integer     | Last message number used |

- **CLQ_GML**
  - Type: Double linked list of CLQ_GM data structure
  - Description: This structure is a node of the group_member_list.
  - Contents
    | Name     | Type           | Contents                                |
    |----------|----------------|-----------------------------------------|
    | member   | CLQ_GM         | The current group member                |
    | prev     | Pointer        | Pointer to the previous node in the list|
    | next     | Pointer        | Pointer to the next node in the list    |

- **CLQ_GM**
  - Type: Structure
  - Description: This structure contains information about a specific member.
  - Contents
    | Name               | Type        | Contents |
    |--------------------|-------------|----------|
    | member_name        | String      | Name of the member                      |
    | long_term_key      | $|p|\text{ bit integer}$ | Long term shared key between myself and member_name, i.e. $K_{ij}$ where $i$ is related to the public key of member_name, and $j$ to my private key |
    | last_partial_key   | $|p|\text{ bit integer}$ | Last partial key for me                 |

- **CLQ_TOKEN**
  - Type: Structure
- Description: Communication token used by CLQ_API, see also

### CLQ_TOKEN_INFO
- **Contents**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>t_data</td>
<td>Integer array</td>
<td>Contains the following encoded data in the same order: group_name, message_type, time_stamp, sender_name, epoch, group_members_list (without long_term_key)</td>
</tr>
</tbody>
</table>

- **Contents**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>group_name</td>
<td>String</td>
<td>Name of the group</td>
</tr>
<tr>
<td>message_type</td>
<td>MSG_TYPE</td>
<td>Type of the message</td>
</tr>
<tr>
<td>time_stamp</td>
<td>Integer</td>
<td>Time stamp of the message</td>
</tr>
<tr>
<td>sender_name</td>
<td>String</td>
<td>Name of the sender</td>
</tr>
</tbody>
</table>

- **CLQ_PARAM**
- **Description**: Diffie-Hellman, DH, public parameters, i.e. $p, q$ and $g$
- **Contents**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p$</td>
<td>$p$ bit integer</td>
<td>DH parameter $p$</td>
</tr>
<tr>
<td>$q$</td>
<td>$q$ bit integer</td>
<td>DH parameter $q$</td>
</tr>
<tr>
<td>$g$</td>
<td>$p$ bit integer</td>
<td>DH parameter $g$</td>
</tr>
</tbody>
</table>

- **CLQ_KEY**
- **Description**: Public key and private key of the user
- **Contents**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>priv_key</td>
<td>$q$ bit integer</td>
<td>Private key of the user</td>
</tr>
<tr>
<td>pub_key</td>
<td>$p$ bit integer</td>
<td>Public key of the user</td>
</tr>
</tbody>
</table>

**Appendix**: API Calls of CLQ_API

- **clq_join(ctx, member_name, group_name, input_token, output_token)**
  - **Caller**: New member
  - **Related to**: Join operation
  - **Return value**: 
  - **Parameters**
• **clq_procedure_join**(ctx, member_name, output_token)
  
  - **Caller**: The current controller
  - **Related to**: Join
  - **Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctx</td>
<td>CLQ_CONTEXT</td>
<td>Current group context, ctx will be modified only if the caller is the controller.</td>
</tr>
<tr>
<td>member_name</td>
<td>String</td>
<td>Name of the new member</td>
</tr>
<tr>
<td>output_token</td>
<td>CLQ_TOKEN</td>
<td>This token should be used as the input token for clq_join.</td>
</tr>
</tbody>
</table>

• **clq_update_ctx**(ctx, input_token)
  
  - **Caller**: All the group members
  - **Related to**: Join, Merge, Leave, Refresh, Mass Join, Mass Leave
  - **Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctx</td>
<td>CLQ_CONTEXT</td>
<td>Current context of each member</td>
</tr>
<tr>
<td>input_token</td>
<td>CLQ_TOKEN</td>
<td>Generated by new member or by the current controller, when an update of key is required (i.e. a user join, a user left, or the key has been compromised). It should be the output token of clq_join, clq_merge, clq_leave, or clq_refresh.</td>
</tr>
</tbody>
</table>

• **clq_update_key**(ctx, member_list, input_token, output_token)
  
  - **Caller**: The current controller or new member
  - **Related to**: Merge, Mass Join
  - **Parameters**
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctx</td>
<td>CLQ_CONTEXT</td>
<td>Current group context</td>
</tr>
<tr>
<td>member_list</td>
<td>List of string</td>
<td>List of names of the new members. When a new member calls this function, this list should be null (Since the input_token is valid).</td>
</tr>
<tr>
<td>input_token</td>
<td>CLQ_TOKEN</td>
<td>Output of clq_update_key by the current controller or previous new member. When the controller calls this function, input_token should be null. (Since the member_list is valid).</td>
</tr>
<tr>
<td>output_token</td>
<td>CLQ_TOKEN</td>
<td>Contains refreshed group secret. It will be used as input_token of clq_merge.</td>
</tr>
</tbody>
</table>

- **clq_factor_out(ctx, input_token, output_token)**
  
  - Caller: all group members except the last and second to last one
  - Related to: Merge, Mass Join
  - Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctx</td>
<td>CLQ_CONTEXT</td>
<td>Current member context</td>
</tr>
<tr>
<td>input_token</td>
<td>CLQ_TOKEN</td>
<td>output_token of clq_update_key</td>
</tr>
<tr>
<td>output_token</td>
<td>CLQ_TOKEN</td>
<td>Contains updated last_partial_key by removing each user's key_share. It will be used as input_token of clq_merge.</td>
</tr>
</tbody>
</table>

- **clq_merge(ctx, sender_name, input_token, output_token)**
  
  - Caller: The last new member
  - Related to: Merge, Mass Join
  - Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctx</td>
<td>CLQ_CONTEXT</td>
<td>Current group context. ctx will be modified.</td>
</tr>
<tr>
<td>sender_name</td>
<td>String</td>
<td>Name of the sender of the input_token</td>
</tr>
<tr>
<td>input_token</td>
<td>CLQ_TOKEN</td>
<td>output_token of clq_factor_out</td>
</tr>
<tr>
<td>output_token</td>
<td>CLQ_TOKEN</td>
<td>Contains updated last_partial_key</td>
</tr>
</tbody>
</table>

- **clq_leave(ctx, member_list, output_token)**
  
  - Caller: All the group members
  - Related to: Leave
  - Parameters
### Description

- **clq\_refresh\_key(ctx, output\_token)**
  - **Caller**: Controller
  - **Related to**: Refresh
  - **Parameters**
    | Name | Type         | Description                                                                 |
    |------|--------------|------------------------------------------------------------------------------|
    | ctx  | CLQ\_CONTEXT| Current group context. ctx will be modified.                                 |
    | output\_token | CLQ\_TOKEN | Updated message to be broadcasted to the group                               |

- **clq\_destroy\_ctx(ctx)**
  - **Description**: It frees the space occupied by the current context
  - **Parameters**
    | Name | Type         | Description                                                                 |
    |------|--------------|------------------------------------------------------------------------------|
    | ctx  | CLQ\_CONTEXT| Current group context. ctx will be destroyed.                                |

- **clq\_destroy\_token(token)**
  - **Description**: It frees the space occupied by the input or output\_token
  - **Parameters**
    | Name | Type         | Description                                                                 |
    |------|--------------|------------------------------------------------------------------------------|
    | token | CLQ\_TOKEN  | Input or output\_token                                                      |

- **clq\_first\_user(ctx, member\_name, group\_name)**
  - **Description**: clq\_first\_user is called by the first user who joins a group
  - **Main purposes**
    * Generates key\_share.
    * Generates member context.
  - **Caller**: The first member in a group
  - **Related to**: Join
  - **Parameters**
    | Name        | Type         | Description                                                                 |
    |-------------|--------------|------------------------------------------------------------------------------|
    | ctx         | CLQ\_CONTEXT| Current member context. ctx should be created.                               |
    | member\_name| String       | Name of the first user                                                      |
    | group\_name | String       | Name of the group                                                           |