RZ 3170 (# 93216) 06/09/99
Computer Science/Mathematics 14 pages

Research Report

The Design of a Group Key Agreement API

G. Ateniese[1], D. Hasse[2], Y. Kim[2] and G. Tsudik[2]*

[1]IBM Research
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

[2]USC Information Sciences Institute
Marina del Rey, CA 90292
U.S.A.

*Research supported by the Defense Advanced Research Project Agency, Information Technology Office (DARPA-ITO), under contract DABT63-97-C-0031.

IBM Research
Almaden · Austin · Beijing · Haifa · T.J. Watson · Tokyo · Zurich

The Design of a Group Key Agreement API

G. Ateniese[1], D. Hasse[2], Y. Kim[2] and G. Tsudik[2]

IBM Research, Zurich Research Laboratory, 8803 Rüschlikon, Switzerland
[2]*USC Information Sciences Institute, Marina del Rey, CA 90292*

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 [2–4], 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_APIseparates 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_APIprimitives are described in detail along with some examples. Finally, we discuss the efficiency of CLQ_APIand conclude with some experimental results obtained with several popular cryptographic packages. (A more detailed description of the API is included in the Appendix.)

2 Group Key Agreement

2.1 Notation

The following notation is used throughout this paper:

Table 1. Notations

n	number of protocol parties (group members)
i, j	indices of group members
M_i	i-th group member; $i \in [1, n]$
G	unique subgroup of $\mathbb{Z}p^*$ of order q with p, q prime
q	order of the algebraic group
g	exponentiation base; generator in group G
x_i	long-term private key of M_i
N_i	M_i's session random number $\in \mathbb{Z}q$
S_n	group shared key among n members
K_{ij}	long-term shared secret between M_i and M_j, $i \neq j$
$\|a\|$	bit length of integer a
H	output size of hash function
$inv(a, b)$	multiplicative inverse of a modulo b

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.

[1] For the definition of the above, the reader is referred to [6].

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.

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.

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 $S_n = g^{N_1 \dots N_n}$, when N_i is provided by M_i[2]. Similarly, the last broadcast message is the set formed by $g^{K_{in} N_1 \dots N_n / N_i}$ for all i in $[1, n-1]$. 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:

[2] This is necessary since in the future any member can become the controller again.

1. M_n generates a new exponent N_n'. Then, M_n produces the following set:

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

Finally, $\mathcal{M}$ is sent to M_{n+1}.
2. Upon reception of the message, M_{n+1} generates a new exponent N_{n+1} and computes $g^{K_{i,n+1} N_1 \ldots N_n' N_{n+1} / N_i}$ 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 $(g^{\frac{N_1 \ldots N_n' N_{n+1}}{N_i} \cdot K_{i,n+1}})^{K_{i,n+1}^{-1} \cdot N_i} = g^{N_1 \ldots N_n' N_{n+1}} = S_{n+1}$. Similarly, M_{n+1} computes the new group shared key $g^{N_1 \ldots N_n' N_{n+1}} = S_{n+1}$ 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 $2n + 1$.

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 $g^{N_1 \ldots N_{n-1} N_n'}$[5]. Then this message is sent to M_{n+1}.
2. Each member M_j, $j = n+1, \ldots, m-2$, generates a new exponent N_j and computes $g^{N_1 \ldots N_n' \ldots N_j}$. Then, this value is sent to M_{j+1}.
3. Upon reception of the message, M_{m-1} generates a new exponent N_{m-1} and computes $g^{N_1 \ldots N_n' \ldots N_{m-1}}$. Then, this value is broadcasted to the entire group.
4. Upon reception of the broadcast, each member M_i, $i = 1, 2, \ldots, m-1$, computes $g^{N_1 \ldots N_n' \ldots N_{m-1} / N_i}$ and sends it to M_m.
5. M_m generates a new exponent N_m and produces the following set:

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

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 $(g^{\frac{N_1 \ldots N_n' \ldots N_m}{N_i} \cdot K_{im}})^{K_{im}^{-1} \cdot N_i} = g^{N_1 \ldots N_n' \ldots N_m} = S_m$. 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 $k = 1$, the current controller M_n starts from step 3 of the above operation after removing the previous exponent N_n. The rest of the protocol runs as above.

Steps 1 through 3 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] This is achieved by exponentiating $N_n' * inv(K_{in}, p) * inv(N_n, q)$ to each element in the last broadcast message.
[4] See section 5 for definition
[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.

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:

1. M_n generates a new exponent N'_n and produces the following set:

$$\mathcal{M} = \{g^{K_{in} N_1 \dots N'_n / N_i} \mid i \in [1, n-1] \text{ and } i \neq d\}$$ [8]

   Finally, $\mathcal{M}$ is broadcasted to the entire group.
2. Upon reception of the message, each M_i computes $(g^{\frac{N_1 \dots N'_n}{N_i} \cdot K_{in}})^{K_{in}^{-1} \cdot N_i} = g^{N_1 \dots N'_n} = S_n$. M_n computes the new group shared key $g^{N_1 \dots N'_n} = S_n$ 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, $g^{K_{dn} N_1 \dots N'_n / N_d}$. 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, $N_i = N_i * inv(K_{in}, q)$ 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

$$\mathcal{M} = \{g^{K_{in} N_1 \dots N'_n / N_i} \mid i \in [1, n-1]\}$$

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 it and 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.

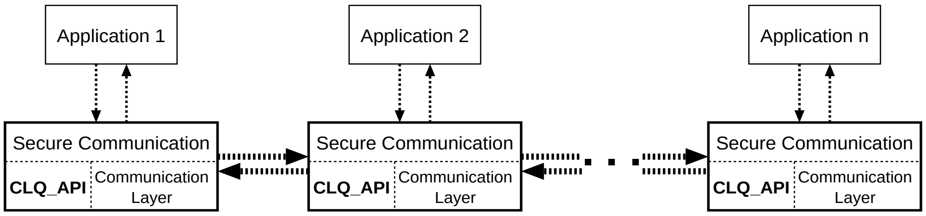

Fig. 1. Communication layer and CLQ_API

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.

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

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 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 generates 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, 2 or 3* of the merge operation. This function is called by the current controller and by some of new members 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.

- clq_factor_out: It performs *step 4* of the merge operation. Every group member calls this function to factor out each user's 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.

[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.

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_BROADCAST : It is broadcasted from the current controller 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 used during the merge[12].

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

- Join

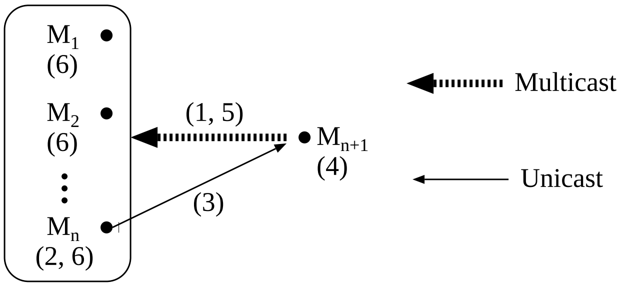

Fig. 2. Join operation

(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

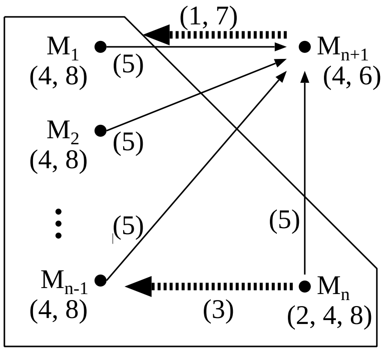

Fig. 3. Merge operation

(1) New member M_{n+1} calls communication protocol join to join a group.
(2) The current controller calls clq_update_key to generate a token containing MERGE_BROADCAST message.
(3) The token is broadcasted to the entire group.
(4) Upon reception of the message, each member calls clq_factor_out to generate a token containing MERGE_FACTOR_OUT message.

[12] See the next section for the details.

(5) The token is sent back to the new user.
(6) For each token, the new member calls clq_merge to generate a token containing KEY_UPDATE_MESSAGE.
(7) The token is broadcasted to the entire group.
(8) Every user calls clq_update_ctx to compute the new group shared key.

– Mass Join

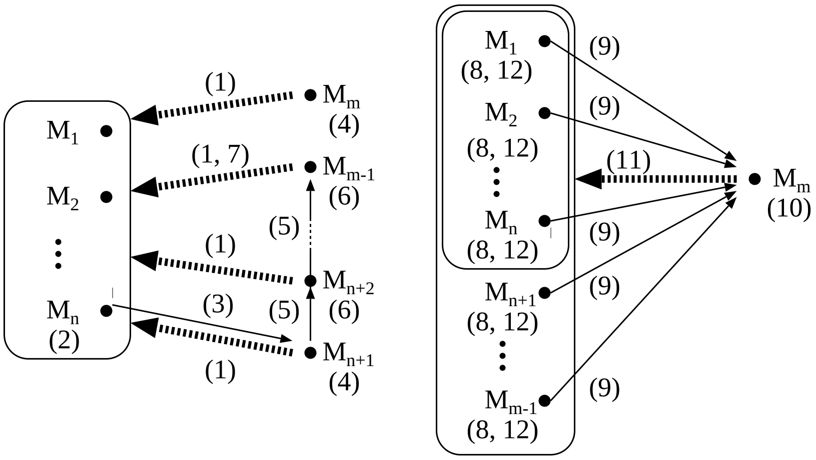

Fig. 4. Mass join operation

(1) Set of members M_i calls communication protocol join to join a group.
(2) The group controller calls clq_update_key to generate a token containing MASS_JOIN message.
(3) The token is sent to the first new member.
(4) Upon reception of the token, the next user calls clq_update_key to generate a token containing MASS_JOIN message.
(5) The token is sent to thee next new member up to the second to last member.
(6) Upon reception of the token, the second to last user calls clq_update_key to generate a token containing MASS_JOIN message.
(7) The token is broadcasted to the entire group.
(8) Upon reception of the message, each member 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 KEY_UPDATE_MESSAGE.
(11) The token is broadcasted to the entire group.
(12) Every user calls clq_update_ctx to compute the new group shared key.

– Leave

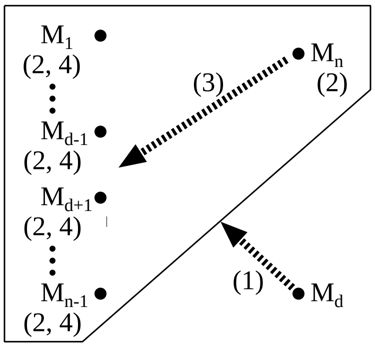

Fig. 5. Leave operation

(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

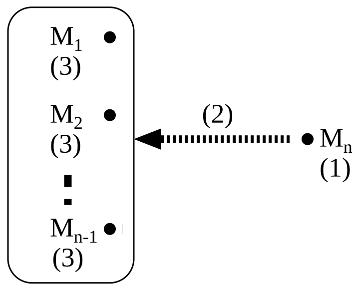

Fig. 6. Key refresh operation

(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 2. Communication costs

Operations	Join	Merge	Leave	Refresh	k-Merge	k-Leave
Number of users after operation	n+1	n+1	n-1	n	n+k	n-k
Rounds	2	3	1	1	k+2	1
Broadcasts	1	2	1	1	2	1
Reverse broadcast	0	1	0	0	1	0
Total messages	2	n+2	1	1	n+2k	1
Maximum bandwidth	n	n	n-1	n-1	n+k-1	n-k

Table 3 illustrates computation costs. Exponentiation is the most expensive operation as it requires $O(log^3 p)$ bit operations in $\mathbb{Z}p^*$. Given a and p, finding the inverse of $a \in \mathbb{Z}p^*$ 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 3. Computation costs

Operations	Join	Merge	Leave	Refresh	k-Merge	k-Leave
Number of users after operation	n+1	n+1	n-1	n	n+k	n-k
Serial exponentiation	2n+1	n+3	n-1	n	n+2k+1	n-k
Total exponentiation	3n+2	3n+2	2n-3	2n-1	3n+4k-2	2n-2k-1

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

Machine	CPU speed	Main memory	OS	Compiler
Ultra Sparc I	233 MHz	64 MB	SUN OS 5.5.1	gcc 2.7.2.2
Pentium I	233 MHz	48 MB	Linux 2.0.36	gcc 2.7.2.3
Pentium II	450 MHz	256 MB	Linux 2.2.9	egcs 1.1.2

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 5. Comparison results

Machine	OpenSSL	Crypto++	RSAREF
UltraSparc	16.6 msec	60.2 msec	146.3 msec
Pentium II	2.5 msec	7.6 msec	22.2 msec
Pentium I	6.4 msec	N/A	N/A

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

1. B. P. Ran Canetti, *A taxonomy of multicast security issues*, April 1999. draft-irtf-smug-taxonomy-00.txt.
2. M. Steiner, G. Tsudik, and M. Waidner, “Diffie-hellman key distribution extended to groups,” in *3nd ACM Conference on Computer and Communications Security*, pp. 31–37, ACM Press, Mar. 1996.
3. M. Steiner, G. Tsudik, and M. Waidner, “CLIQUES: A new approach to group key agreement,” in *IEEE International Conference on Distributed Computing Systems*, May 1998.
4. G. Ateniese, M. Steiner, and G. Tsudik, “Authenticated group key agreement and friends,” in *ACM Symposium on Computer and Communication Security*, November 1998.
5. W. Diffie and M. Hellman, “New directions in cryptography,” *IEEE Transactions on Information Theory*, vol. IT-22, pp. 644–654, Nov. 1976.
6. A. Menezes, P. V. Oorschot, and S. Vanstone, *Handbook of applied cryptography.* CRC Press series on discrete mathematics and its applications, CRC Press, 1996. ISBN 0-8493-8523-7.
7. Y. Amir and J. Stanton, “The spread wide area group communication system,” Tech. Rep. CNDS-98-4, The Johns Hopkins University, avalaible at http://www.cnds.jhu.edu/projects/commedia/spread/docspread.html, 1998.
8. L. Moser, P. Melliar-Smith, D. Agarwal, R. Budhia, and C. Lingley-Papadopoulos, “Totem: A fault-tolerant multicast group communication system,” *Communications of the ACM*, vol. 39, no. 4, 1996.
9. Y. Amir, D. Dolev, S. Kramer, and D. Malki, “A communication sub-system for high availability,” in *22nd Annual International Symposium on Fault-Tolerant Computing (FTCS)*, 1992.
10. N. Koblitz, *A Course in Number Theory and Cryptography.* New York: Springer-Verlag, 1987.
11. RSA Laboratories, “Rsaref2.0(tm): A cryptographic toolkit for privacy-enhanced mail,” Jan. 1996. http://www.rsa.com.
12. W. Dai, “Crypto++,” May 1999. http://www.eskimo.com/~weidai/cryptlib.html.
13. OpenSSL Project team, “Openssl,” May 1999. http://www.openssl.org/.
14. T. Hudson and E. Young, “SSLeay,” Dec. 1998. http://www.ssleay.org/.
15. K. E. Hickman and T. Elgamal, “The SSL protocol,” RFC draft, Netscape Communications Corp., June 1995. Version 3.0, expires 12/95.

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	$\|q\|$ bit integer	M_i's session random number, N_i
group_secret	$\|p\|$ bit integer	Current group shared key
group_secret_hash	H 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	$\|p\|$ 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	$\|p\|$ bit integer	Last partial key for me

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

Name	Type	Contents
length	Integer	Size of t_data
t_data	Integer array	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)

- CLQ_TOKEN_INFO
  - Type : Structure
  - Description : This structure contains information about the token.
  - Contents

Name	Type	Contents
group_name	String	Name of the group
message_type	MSG_TYPE	Type of the message
time_stamp	Integer	Time stamp of the message
sender_name	String	Name of the sender

– CLQ_PARAM
- Type : Structure
- Description : Diffie-Hellman, DH, public parameters, i.e. p, q and g
- Contents

Name	Type	Contents
p	$\|p\|$ bit integer	DH parameter p
q	$\|q\|$ bit integer	DH parameter q
g	$\|p\|$ bit integer	DH parameter g

– CLQ_KEY
- Type : Structure
- Description : Public key and private key of the user
- Contents

Name	Type	Contents
priv_key	$\|q\|$ bit integer	Private key of the user
pub_key	$\|p\|$ bit integer	Public key of the user

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

Name	Type	Description
ctx	CLQ_CONTEXT	Context for the new user, this should be created in this function.
member_name	String	Name of the user calling this function
group_name	String	Name of the group that has been joined. This name has to match with the one that is included in the input_token.
input_token	CLQ_TOKEN	Message received from the controller. It is the output_token generated by clq_proc_join.
output_token	CLQ_TOKEN	New key update message to be broadcasted to the group. It will be used as input_token of clq_update_ctx.

– clq_proc_join(ctx, member_name, output_token)
- Caller : The current controller
- Related to : Join
- Parameters

Name	Type	Description
ctx	CLQ_CONTEXT	Current group context, ctx will be modified only if the caller is the controller.
member_name	String	Name of the new member
output_token	CLQ_TOKEN	This token should be used as the input_token for clq_join.

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

Name	Type	Description
ctx	CLQ_CONTEXT	Current context of each member
input_token	CLQ_TOKEN	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.

- clq_update_key(ctx, member_list, input_token, output_token)
  - Caller : The current controller or new member
  - Related to : Merge, Mass Join
  - Parameters

Name	Type	Description
ctx	CLQ_CONTEXT	Current group context
member_list	List of string	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).
input_token	CLQ_TOKEN	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).
output_token	CLQ_TOKEN	Contains refreshed group_secret. It will be used as input_token of * clq_factor_out by the current group members or * clq_update_key by the next new member.

- 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

Name	Type	Description
ctx	CLQ_CONTEXT	Current member context
input_token	CLQ_TOKEN	output_token of clq_update_key
output_token	CLQ_TOKEN	Contains updated last_partial_key by removing each user's key_share. It will be used as input_token of clq_merge.

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

Name	Type	Description
ctx	CLQ_CONTEXT	Current group context. ctx will be modified.
sender_name	String	Name of the sender of the input_token
input_token	CLQ_TOKEN	output_token of clq_factor_out
output_token	CLQ_TOKEN	Contains updated last_partial_key

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

Name	Type	Description
ctx	CLQ_CONTEXT	Current group context
member_list	List of string	List of members leaving
output_token	CLQ_TOKEN	Updated message to be broadcasted to the group

- 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