The Design of a Group Key Agreement API

Giuseppe Ateniese
Computer Science Department
Johns Hopkins University
ateniese@cs.jhu.edu

Olivier Chevassut
Lawrence Berkeley National Laboratory
University of California
chevassu@george.lbl.gov

Damian Hasse Yongdae Kim Gene Tsudik*
Computer Networks Division
USC Information Sciences Institute
{*hasse,yongdaek,gts*}*@isi.edu*

Abstract

As collaborative applications grow in popularity the need for appropriate security guarantees, services and mechanisms becomes apparent. This paper describes a protocol suite and an 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 (on the order of 100 members) dynamic peer groups.

1. Introduction

Securing group communication is a complex issue. It poses a number of challenges ranging from basic cryptographic algorithms to systems, communication design and secure implementation [4]. The two common group security concerns are: 1) privacy: assuring that intra-group communication remains secret to non-members, and 2) 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 only to all current group members 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 an Application Programming Interface (API) for group key agreement. This API, called CLQ_API , is based on the CLIQUES protocol suite and is geared specifically for dynamic peer groups. (Peer groups are relatively small, non-hierarchical groups typically used for replication or collaborative, many-to-many applications.) CLIQUES protocols [14, 15, 3], in turn, are the group extensions of the well-known Diffie-Hellman key exchange [6]. 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 cryptographic protocols from communication, i.e., the actual group communication is left to the underlying communication subsystem (preferably, a reliable group communication system, e.g., SPREAD[2], TOTEM[11] or TRANSIS[1]). Moreover, network events such as network partitions, failures and other abnormalities are assumed to be taken care of by the same communication system. As described in subsequent sections, this allows the design of a small, concise and communication-independent API.

The rest of this paper is organized as follows. We begin with the notation and the brief description 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

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

[1] For the definition of the above, the reader is referred to [10].

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.)

2. Group Key Agreement

The following notation is used throughout this paper:

n	number of protocol parties (group members)
i, j	indices of group members
p, q	prime integers
M_i	i-th group member; $i \in [1, n]$
G	unique subgroup of Z_p^* of order q
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 $\in Z_q$
S_n	group shared key among n members
K_{ij}	long-term shared secret between M_i and M_j
$\|a\|$	bit length of integer a
H	output size of hash function
$inv(a, b)$	multiplicative inverse of a modulo b

2.1. Group key agreement operations

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

Single-member changes: include single member additions or deletions. The former occurs when a prospective member wants to join a group and the latter – when a 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 all group members, depending on the local policy.

Multiple-member changes: also include addition and deletion. The former can be further broken into:

- Mass join: multiple disparate new members are brought into an existing group
- Group fusion: two or more groups are merged to form a single group.

The latter includes:

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

Key refresh: is not a membership change operation, however, we discuss it here for the sake of completeness. It 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 group key refresh operation is needed. The lifetime of a key is determined by the application-specific policy.
- Recover from the compromise of a 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 Protocols

As mentioned above, CLIQUES is a protocol suite providing authenticated contributory key agreement for dynamic peer groups. The following operations are supported by CLIQUES:

- 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 remainder of this section. The mapping between the group membership changes and the corresponding key agreement (cryptographic) operations is as follows:

Membership Operation	CLIQUES Operation
Forced leave	Leave
Voluntary leave	Leave
Single join	Join
Mass join	Merge
Group fusion	Merge
Group fission	Leave
Mass leave	Leave
Key refresh	Key Refresh

3.1. Group Controller and Group Key

Group Controller. In the current version of CLIQUES, the last member to join the group becomes a *group controller*. Equivalently, the role of a group controller is always played by the newest member. This behavior can be easily changed if required. In fact, any selection criteria can be used to select a group controller as long as it yields a consistent outcome across all group members. (CLIQUES assumes that the underlying communication system provides a timely and consistent membership view to all group members; this property is referred to as membership *view synchrony*.) One obvious alternative to the present criterion is to let the oldest member be the group controller.

We emphasize that a group controller is not in any way a privileged group member. It exists to prevent contention and should be viewed as a burden (or a chore) rather than a privilege. Specifically, the group controller assists new members in joining the group and initiates operations stemming from departing members or the need for periodic re-keying.

Furthermore, it is important to note that *any* group member can, at any time, initiate a group re-key by unilaterally updating its key share and distributing appropriate partial keys to the rest for the group[2]. (See below.)

Group Key. In the following operations, the group key (also referred to as group secret) has the form $S_n = g^{N_1 \cdots N_i \cdots N_n}$, where n is the group size and N_i ($0 < i \leq n$) is provided by the i-th member, M_i[14]. In all CLIQUES protocols, the last broadcast message is always the set formed by:

$$g^{K_{in} \frac{N_1 \cdots N_n}{N_i}} \quad \forall i \in [1, n-1]$$

This set, from the last broadcast, must be retained by each group member. This is necessary since any member can become a controller due to a group partition or a network fault. (In other words, a non-controller can become a group controller if all "younger" members fail or become partitioned out.) Although the entire group receives this message from the current controller, each member uses a different element (key) to compute the new group secret.

Finally, the computation of the pairwise long-term key, K_{ij}, shared between any two members M_i and M_j is assumed to be performed before its use is required in the protocol.

3.2. 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, the protocol runs as follows:

1. M_n generates a new secret and random exponent N'_n and produces the following set:[3]

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

   Next, $\mathcal{M}$ is sent to M_{n+1}.

2. Upon receipt of the message, M_{n+1} generates a new secret exponent N_{n+1} and computes:

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

   This set is broadcast to the entire group (including itself).

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

   $$(g^{\frac{N_1 \cdots N'_n N_{n+1}}{N_i} \cdot K_{i,n+1}})^{K^{-1}_{i,n+1} \cdot N_i} = g^{N_1 \cdots N'_n N_{n+1}} = S_{n+1}$$

Steps 1 and 2 require n modular exponentiations (by M_n and M_{n+1}, respectively) and step 3 requires a single exponentiation by each member. Hence, the protocol requires $(2n+1)$ serial exponentiations.

3.3. Merge

The merge operation adds $k > 0$ members to the current group of $n \geq 1$ 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 M_n is the current controller, the protocol runs as follows:

[2] Though we fully support re-keying mechanism, its usage depends on local policy.

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

1. M_n generates a new exponent N'_n and computes: $g^{N_1 \ldots N_{n-1} N'_n}$ by modular exponentiation of the previous group shared key with $(N'_n * inv(N_n, q))$. Then, this value is sent to M_{n+1}, the first new member.

2. Each new (merging) member M_j, $j = n+1, \ldots, m-1$, generates a new exponent N_j, computes $g^{N_1 \ldots N'_n \ldots N_j}$ and forwards the result to M_{j+1}.

3. Upon receipt of the accumulated value, M_m simply broadcasts it to the entire group.

4. Every group member M_i (old and new) received the broadcast, computes $g^{N_1 \ldots N'_n \ldots N_{m-1}/N_i}$ and sends back to M_m.

5. Having received all responses from the group members, M_m generates a new secret exponent N_m and produces the following set: [4]

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

which it broadcasts to the group.

6. Each member M_i computes the group shared key exactly as in the last step of the Join protocol.

In case of a single-member Merge ($k = 1$), 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 while steps 4 and 6 each involve one exponentiation (in parallel by each member). Finally, step 5 needs $(n + k - 1)$ exponentiations. Thus the total number of serial exponentiation for a k-member Merge is $(n + 2k + 1)$.

3.4. Leave

The leave operation removes one or more members from the group of n members. As a result, a new group shared key S_{n-k}, is computed where k is the number of the departing members. Let the set $\mathcal{K}$ denote the leaving members.

There are two cases to consider: 1) the current controller $M_n \in \mathcal{K}$ and 2) $M_n \notin \mathcal{K}$. In the first case, the controller's role is passed on to the most recent remaining member M_d where $d = n - k$. (See Section 4.1 below.) Assuming that members are indexed in the order of joining the group (i.e., M_n is the most recent member to join) the remaining members must be *renumbered* after a Leave. Now, even if the previous controller is not in $\mathcal{K}$, it will acquire a new index and become M_d due to renumbering.

The protocol runs as follows:

1. The controller M_d generates a new exponent N'_d, produces the following set and broadcasts it to the remaining group.

$$\mathcal{M} = \{g^{K_{i,d} N_1 \ldots N'_d / N_i} \mid M_i \notin \mathcal{K}\}$$

2. Having received $\mathcal{M}$, each M_i computes:

$$S_d = (g^{\frac{N_1 \ldots N'_d}{N_i} \cdot K_{i,d}})^{K_{i,d}^{-1} \cdot N_i} = g^{N_1 \ldots N_d}$$

Members of the departing set $\mathcal{K}$ cannot compute the new key since the group controller only computes partial keys for the remaining group.

Step 1 requires $n - k$ exponentiations whereas step 2 needs a single exponentiation by each member. In total, the leave operation requires $(n - k + 1)$ serial exponentiations.

3.5. Key refresh

The key refresh operation updates the current group shared key, S_n. It is, in fact, a special case of a Leave with $k = 0$. Its usage depends on the application policy.

4. CLQ_API

CLQ_API is a group key management API based on the CLIQUES protocol suite. It is fairly small, containing only eight function calls.

CLQ_API does not perform any communication. All API calls either accept a token, produce one or both. As mentioned before, CLIQUES (and CLQ_API) requires reliably and sequenced communication and membership view synchrony. Any communication system providing these features can be used in conjunction with CLQ_API , e.g., SPREAD, TOTEM, and TRANSIS might be used.

4.1. CLQ_API Calls

This section describes CLQ_API calls. Each call represents one or more steps in one of the group operations presented in Section 3. Details of the data structures, definitions and secondary function calls can be found in the appendix.

The following terms are used throughout this section:

[4] Note that $\mathcal{M}$ is the same as its counterpart in the Join protocol.

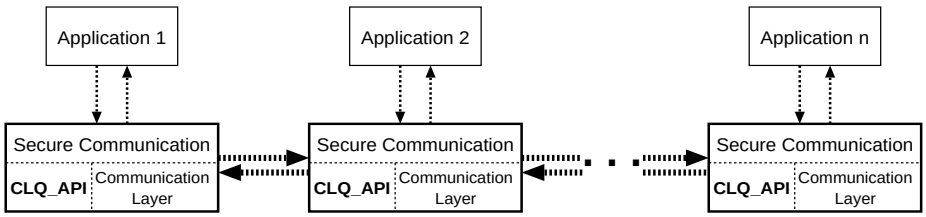

Figure 1. Communication layer and CLQ_API

Context: contains information about each user required by CLQ_API.

Epoch: message sequence number of a token. It is used to keep track of the group secret evolution and to prevent replay attacks.

Timestamp: time when a token is sent. It is used to check the freshness of an incoming token.

Session random: secret key share generated by each member.

Token: basic protocol unit; includes: epoch, timestamp, and a CLIQUES protocol message.

We now describe the CLQ_API calls:

- clq_proc_join: 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 purposes of this function are:
  - generate a new session random for the current controller.
  - remove long term keys and previous session random from the partial keys of all users.
  - add the new session random into the partial keys of all users.
- clq_join: performs *step 2* of the join operation. The new group member calls this function using the token received from the current controller. The purposes of this function are:
  - generate a new session random for the new member.
  - generate long-term keys between the new member and each user (can also be done before calling this function).
  - compute the partial keys for other group members.
- clq_update_ctx: 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 receipt of the token sent by the current controller. The main purpose of this function is to compute group shared key with the caller's session random and the incoming token.
- clq_update_key: performs *step 1 and 2* of the merge operation. This function is called by the current controller and by all (but last) of the new members to add their session randoms to the group shared key.

  The main purposes of this function are:
  - generate new session random.
  - add new session random to the group shared key.
- clq_factor_out: performs *step 4* of the merge operation. Every group member (except the controller) calls this function to factor out its own session random number from the group shared key.
- clq_merge: performs *step 5* of the merge operation. The last new member (new controller) calls this function to add its session random to each of received partial keys from the group members.
- clq_leave: performs *step 1* of the leave operation. Every group member calls this function right immediately after one or more members leave the group. A group member may also become the current controller after calling this function; this can happen if it finds itself to be the newest member within the remaining 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:
  - generate new session random.
  - compute partial keys for other group members (except, of course, the departed ones).

- clq_refresh_key: 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.

4.2. Operations of CLQ_API

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

- NEW_MEMBER : sent by the controller to a new member, when a new member joins the group.
- KEY_UPDATE_MESSAGE : sent to every member in order to update the group shared key.
- MERGE_KEY_UPDATE : sent to every member to update the group shared key in merge operation.
- MERGE_BROADCAST : broadcast from the last new member when one or more members merge.
- MERGE_FACTOR_OUT : sent by each group member to the new controller in a merge operation.
- MASS_JOIN : sent to the next new member in a merge operation.

We now clarify group operations using the function calls from the previous section.

- Join

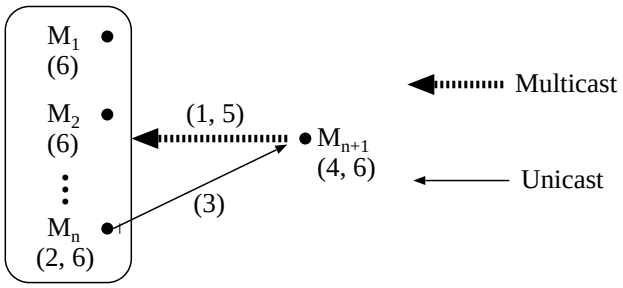

Figure 2. Join operation

(1) New member M_{n+1} calls communication protocol JOIN primitive.

(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 broadcast to the entire group.

(6) Every user calls clq_update_ctx to compute the new group shared key.

- Merge (single member)

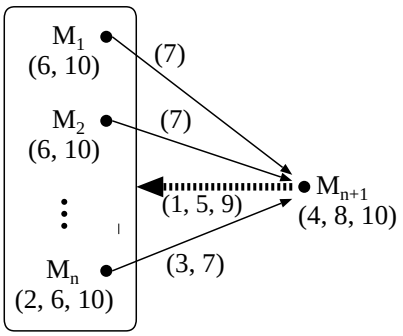

Figure 3. Merge operation

(1) New member M_{n+1} calls communication protocol JOIN primitive.

(2) The current controller calls clq_update_key to generate a token containing MASS_JOIN message.

(3) The token is sent to the new user (who becomes the new controller).

(4) The new controller calls clq_update_key to generate a token containing MERGE_BROADCAST message.

(5) The token is broadcast to the group.

(6) Each member (except the new one) calls clq_factor_out to generate a token containing MERGE_FACTOR_OUT message.

(7) Each member sends the output token back to the new controller.

(8) For each received token, the new controller calls clq_merge to generate a token containing MERGE_KEY_UPDATE message. When the last token is received, MERGE_KEY_UPDATE returns an output token.

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

(10) Every user calls clq_update_ctx to compute the new group shared key.

- Merge (multiple members)

(1) A MERGE event occurs either at an explicit request or as a result of a multiple members calling communication protocol join to join a group.

(2) The current group controller calls clq_update_key to generate a token containing MASS_JOIN message.

(3) The token is sent to the first merging 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 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

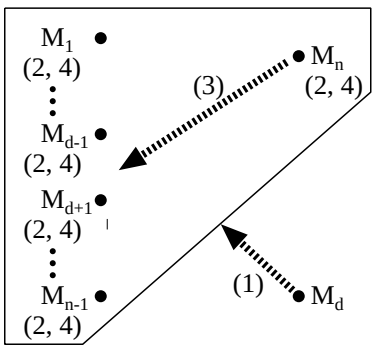

Figure 5. Leave operation

(1) The leaving member M_d calls communication protocol LEAVE primitive. (In case of a member disconnect or network failure the communication system notifies the group of the event.)

(2) All group members call clq_leave; only the group controller obtains the output token containing KEY_UPDATE_MESSAGE.

(3) The token is broadcast 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

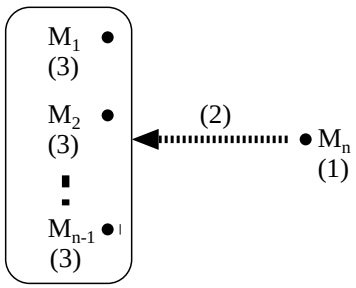

Figure 6. Key refresh operation

(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

The communication overhead is summarized in Table 1.

Table 2 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 [9, 10] for more details. Hence, the speed of each operation depends largely on the number of serial exponentiations. (Note that the cost for generating the long term keys is not included in this table.)

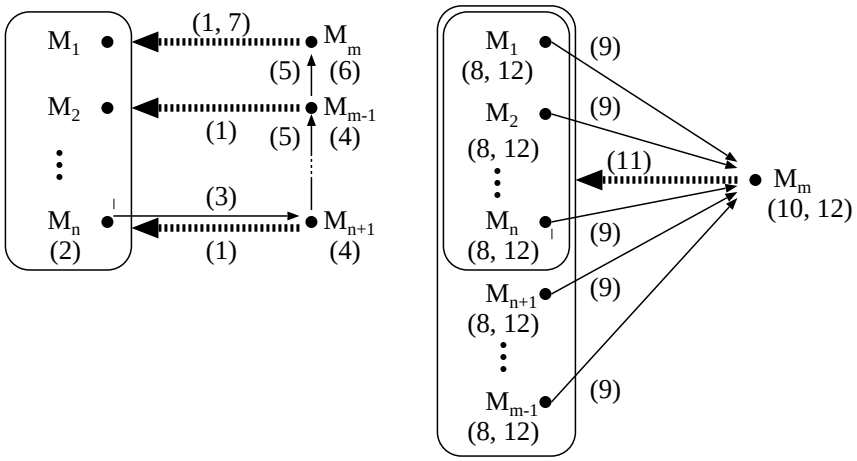

Figure 4. Merge(several members) operation

Table 1. 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 2. 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

5.1. Exponentiation

As mentioned earlier, modular exponentiation is most expensive operation in CLIQUES protocols. In this section, we compare the performance of modular exponentiation operations using three different cryptographic libraries on three different processors(See Table 3).

The cryptographic libraries are summarized as follows:

- RSAREF[13]
  A cryptographic toolkit for privacy-enhanced mail. We used version 2.0 developed in 1996.
- Crypto++[5]
  A public domain C++ class library of cryptographic schemes published by Wei Dai. Note that, since addition and subtraction are implemented for the Pentium assembler, this package performs better on Pentium than in other microprocessors. We used version 3.1 developed in May 1999.
- OpenSSL[12]
  A successor of SSLeay[8], OpenSSL is a cryptographic toolkit implementing Secure Socket Layer(SSL v.2/3)[7]. (Implements some basic operations in assembler on various platforms.) We used version 0.9.3a developed in May 1999.

We measured the performance 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 4 shows the comparison.

At the time of this writing, OpenSSL appears to be the fastest of all public domain cryptographic libraries. On a Pentium II processor it requires only 2.5 *msecs* for each 512-bit modular exponentiation.

6. Conclusion

This paper describes a protocol suite and an API designed specifically for securing dynamic collaborative applications in unreliable networks. The API is based on the extensions of Diffie-Hellman key agreement developed in the CLIQUES project. It provides core security services (most notably, authenticated key agreement) for relatively small and dynamic peer groups.

References

[1] 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.

[2] Y. Amir and J. Stanton. The spread wide area group communication system. Technical Report CNDS-98-4, The Johns Hopkins University, avalaible at www.spread.org/docs/docspread.html, 1998.

[3] G. Ateniese, M. Steiner, and G. Tsudik. Authenticated group key agreement and friends. In *ACM Symposium on Computer and Communication Security*, November 1998.

[4] R. Canetti and B. Pinkas. *A taxonomy of multicast security issues*, April 1999. draft-irtf-smug-taxonomy-00.txt.

[5] W. Dai. Crypto++, May 1999. available at www.eskimo.com/~weidai/cryptlib.html.

[6] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, Nov. 1976.

[7] K. E. Hickman and T. Elgamal. The SSL protocol. RFC draft, Netscape Communications Corp., June 1995. Version 3.0, expires 12/95.

[8] T. Hudson and E. Young. SSLeay, Dec. 1998. available at www.ssleay.org.

[9] N. Koblitz. *A Course in Number Theory and Cryptography*. Springer-Verlag, New York, 1987.

[10] 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.

[11] 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*, 39(4), 1996.

[12] OpenSSL Project team. Openssl, May 1999. available at www.openssl.org.

[13] RSA Laboratories. Rsaref2.0(tm): A cryptographic toolkit for privacy-enhanced mail, Jan. 1996. available at www.rsa.com.

[14] M. Steiner, G. Tsudik, and M. Waidner. Diffie-hellman key distribution extended to groups. In *3nd ACM Conference on Computer and Communications Security*, pages 31–37. ACM Press, Mar. 1996.

[15] 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.

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
    - member_name (String): Name of the user M_i
    - group_name (String): Name of the group

Table 3. 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

Table 4. Comparison results

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

    - 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
    - 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
    - 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 member_name.
- CLQ_TOKEN
  - Type : Structure
  - Description : Communication token used by CLQ_API, see also CLQ_TOKEN_INFO
  - Contents
    - length (Integer): Size of t_data
    - t_data (Integer array): Contains the following encoded data: 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
    - 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
    * 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
    * 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
  - Parameters
    * 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 : Current controller
  - Related to : Join
  - Parameters
    * 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 : Every group member.
  - Related to : Join, Merge, Leave, Key Refresh
  - Parameters
    * 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 : Current or new controller.
  - Related to : Merge
  - Parameters
    * 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 : Every group member (except the last one[5])
  - Related to : Merge
  - Parameters
    * 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.

[5] If the current controller, i.e. the last member, calls clq_factor_out then the output_token will return NULL

- clq_merge(ctx, sender_name, input_token, output_token)
  - Caller : The last new member
  - Related to : Merge, Mass Join
  - Parameters
    * 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 : Every group member
  - Related to : Leave
  - Parameters
    * 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 : Key Refresh
  - Parameters
    * 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 : Frees the space occupied by the current context
  - Parameters
    * ctx (CLQ_CONTEXT): Current group context. ctx will be destroyed.
- clq_destroy_token(token)
  - Description : Frees the space occupied by the token
  - Parameters
    * 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
    * 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