SEAL: A Secure Communication Library for Building Dynamic Group Key Agreement Applications ⋆

Patrick P. C. Lee [a] John C. S. Lui [b] David K. Y. Yau [c]

[a] *Department of Computer Science, Columbia University, New York, NY 10027, USA*
[b] *Department of Computer Science & Engineering, The Chinese University of Hong Kong, Hong Kong*
[c] *Department of Computer Sciences, Purdue University, West Lafayette, IN 47907, USA*

Abstract

We present the *SEcure communicAtion Library (SEAL)*, a Linux-based C language application programming interface (API) library that implements secure group key agreement algorithms that allow a communication group to periodically renew a common secret group key for secure and private communication. The group key agreement protocols satisfy several important characteristics: *distributed property* (i.e., no centralized key server is needed), *collaborative property* (i.e., every group member contributes to the group key), and *dynamic property* (i.e., group members can join or leave the group without impairing the efficiency of the group key generation). Using SEAL, we developed a testing tool termed *Gauger* to evaluate the performance of the group key agreement algorithms in both wired and wireless LANs according to different levels of membership dynamics. We show that our implementation achieves robustness when there are group members leaving the communication group in the middle of a rekeying operation. We also developed a secure chat-room application termed *Chatter* to illustrate the usage of SEAL. Our SEAL implementation demonstrates the effectiveness of group key agreement in real network settings.

Key words: Secure group communication, group key agreement implementation.

1. Introduction

Many group-oriented applications require *communication confidentiality*, meaning that the communication data among a group of authorized members are secure and inaccessible to group outsiders. Examples of these applications include secure chat-rooms, business conferencing systems, file sharing tools, programmable router communication, and network games in strategy planning. To offer data privacy, an effective approach is to require all group members to establish a common secret *group key*, which is held only by group members, but not outsiders, for encrypting the transmitted data. In particular, *rekeying*, or renewing the group key, is necessary whenever there is any change in the group membership (e.g., a new member joins the group or an existing member leaves the group) in order to guarantee both *backward confidentiality* (i.e., no joining member can read the previous data) and *forward confidentiality* (i.e., no leaving member can access the future data).

One simple way for a communication group to perform rekeying is to set up a centralized key server that is responsible for renewing the group key and distributing it to all group members. However, relying on a centralized key server introduces the single-point-of-failure problem since if the key server is compromised, the whole key establishment process fails. Moreover, centralized management is not adequate for decentralized network settings such as peer-to-peer or mobile ad hoc networks. Therefore, we should design a *se-*

⋆ This research is supported in part by the RGC Earmarked Grant.
* Corresponding author. Tel.: +852 2609 8407; fax: +852 2609 5024
Email address: cslui@cse.cuhk.edu.hk (John C. S. Lui).
[1] Source can be downloaded from:
http://www.cse.cuhk.edu.hk/∼cslui/ANSRlab/software/SEAL/.

cure group key agreement scheme that allows the group members to agree upon the group key without relying on a centralized key server. The group key agreement scheme should satisfy three properties. The first one is the *distributed property*, which means no centralized key server is required. The second one is the *collaborative property*, which means all group members contribute their own secret piece of information to the generation of the group key. Both distributed and collaborative properties seek to eliminate the single-point-of-failure problem such that when one member is compromised, the group can still continue with its secure communication by excluding the compromised member. The final property is the *dynamic property*, which means the group key agreement scheme retains both accuracy and efficiency even if the group key agreement scheme involves dynamic membership events, such as when a new member joins or an existing member leaves the group.

In (Lee et al., 2002), we proposed three *interval-based distributed rekeying algorithms* (or *interval-based algorithms* for short) called *Rebuild*, *Batch*, and *Queue-batch*, each of which performs rekeying (i.e., renews the group key) on a batch of join and leave requests periodically at regular *rekeying intervals* and satisfies the distributed, collaborative, and dynamic properties. Instead of performing rekeying for each single join or leave event, the interval-based algorithms use the periodic rekeying approach so that the rekeying efficiency is preserved in response to the frequent join and leave events, with a tradeoff of weakening both backward and forward confidentialities. In (Lee et al., 2002), we evaluated the performance of the interval-based algorithms via mathematical modeling and simulation experiments. However, there are still open issues about the actual implementation and performance of the interval-based algorithms in a real network environment.

In this paper, we present the *SEcure communicAtion Library (SEAL)*, an application programming interface (API) library that implements the interval-based algorithms in the C language under Linux and allows a communication group to agree upon a common secret group key for data encryption in real-life group-oriented applications. Through the implementation framework, we analyze the design issues that are critical for implementing the algorithms. Furthermore, we export from SEAL a set of API function calls that can be used to develop secure group-oriented applications.

Using SEAL, we developed a performance testing tool called *Gauger* that simulates a member application participating in a secure group communication. We evaluate the rekeying performance of the interval-based algorithms in a wired LAN where 40 Gaugers continuously join and leave a communication group according to different levels of membership dynamics. We show that the Queue-batch algorithm generally takes less than one second to complete a rekeying operation. Also, we show that our implementation achieves robustness when there are group members leaving the group in the middle of a rekeying operation. Furthermore, we evaluate SEAL using both wired and wireless LAN testbeds so as to provide preliminary insights on how different deployment environments vary the performance of SEAL.

We further used SEAL to implement a secure chat-room application called *Chatter* to illustrate how to deploy SEAL in real-life applications. Also, we identify a number of potential applications that can benefit from SEAL.

The rest of the paper proceeds as follows. In Section 2, we overview the group key agreement algorithms that substantiate our implementation. Section 3 presents the design of the SEAL implementation and address the design issues. Section 4 provides the implementation details. In Section 5, we present a set of SEAL API functions for building secure group-oriented applications. In Section 6, we evaluate the performance of the interval-based algorithms using wired and wireless LAN testbeds. In Section 7, we introduce Chatter, a secure chat-room application built upon SEAL. We also suggest other potential applications where SEAL fits their development needs. Section 8 discusses related work, and Section 9 concludes and suggests future work.

2. Group Key Agreement Algorithms

In this section, we first overview the Tree-based Group Diffie-Hellman (TGDH) protocol (Kim et al., 2004). We then describe the group key agreement algorithms (Lee et al., 2002) that substantiate the group key agreement protocols in our SEAL implementation.

2.1. *Tree-based Group Diffie-Hellman (TGDH)*

In TGDH, each group member holds a binary *key tree*. Each node v is associated with a *secret* (or private) key K_v and a *blinded* (or public) key BK_v. All arithmetic operations are performed in a cyclic group of prime order p with the generator α. Mathematically, the keys K_v and BK_v are related by the formula $BK_v = \alpha^{K_v} \bmod p$.

Each member, denoted by M_i for some integer i, holds the secret keys along its *key path*, which is the sequence of nodes starting from its corresponding leaf

node up to the root node. In addition, each member holds the blinded keys along its *co-path*, which contains the nodes whose siblings belong to its key path, although the member may cache the blinded keys associated with other nodes. Figure 1 depicts a possible key tree composed of six members M_1 to M_6. M_1 holds the secret keys of nodes 7, 3, 1, and 0, and the blinded keys of nodes 8, 4, and 2. Since node 0 lies on the key paths of all members, its associated secret key is chosen as the *group key*.

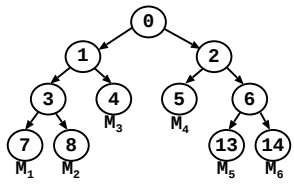


Fig. 1. A possible key tree used in the TGDH algorithm.

Each member has to determine the secret keys along its key path. First, it obtains the secret key at its corresponding leaf node through a secure random number generator. Then it computes the secret key at each non-leaf node, say v, from the keys at the child nodes of v based on the Diffie-Hellman algorithm (Diffie and Hellman, 1976). Mathematically, given a non-leaf node v, its child nodes are represented by $2v + 1$ and $2v + 2$, and its secret key is given by

$$K_v = (BK_{2v+1})^{K_{2v+2}} = (BK_{2v+2})^{K_{2v+1}} = \alpha^{K_{2v+1}K_{2v+2}} \bmod p. \quad (1)$$

To understand how each member computes the group key, we consider again the group shown in Figure 1. Each member M_i first generates its own secret key which is then associated with one of the leaf nodes in the key tree. For example, member M_1 first generates K_7 and requests the blinded key BK_8 from M_2, BK_4 from M_3, and BK_2 from either M_4, M_5, or M_6. Given M_1's secret key K_7 and the blinded key BK_8, M_1 can generate the secret key K_3 according to Equation 1. Given the blinded key BK_4 and the newly generated secret key K_3, M_1 can generate the secret key K_1 based on Equation 1. Given the secret key K_1 and the blinded key BK_2, M_1 can generate the secret key K_0 at the root. Other members can follow a similar approach to compute the group key. From that point on, any communication in the group can be encrypted based on the secret key (or group key) K_0.

When a member joins or leaves the group, all group members carry out a *rekeying* operation, in which they renew the group key to ensure both backward confidentiality (i.e., no joining member can read the previous data) and forward confidentiality (i.e., no leaving member can access the future data). In each rekeying operation, all group members elect a member called *sponsor*, which is responsible for broadcasting the updated blinded keys. By convention, the sponsor is the rightmost member under the subtree rooted at the sibling of the joining or leaving member. We point out that the presence of a sponsor does not violate the collaborative property since the sponsor only facilitates the rekeying operation but does not add extra contribution to the group key.

Figure 2 illustrates the idea of the rekeying operation under both single join and single leave cases. In the single join case, suppose that a new member M_7 joins the group. To keep the key tree balanced, M_7 should be added to the highest leaf node in the key tree. Here, member M_7 can be associated with node 12, which is attached under node 5. The *renewed nodes*, the ones whose keys need to be renewed, are then identified. These include nodes 5, 2 and 0 (the nodes shaded in Figure 2). Also, member M_4 is elected to be the sponsor. It renews K_5, K_2, and K_0 using BK_{12}, BK_6, and BK_1, respectively, and broadcasts the blinded keys BK_5 and BK_2. Members M_1, M_2 and M_3 can compute K_0 upon receiving BK_2, and members M_5 and M_6 can compute K_0 upon receiving BK_5. In this example, we assume that when M_7 joins the group, it broadcasts its individual blinded key (i.e., BK_{12}) and learns the required blinded keys for computing the group key from other members. In the single leave case, suppose that member M_4 wants to leave the group. Nodes 5 and 12 pruned, and node 11 is promoted to be node 5. The renewed nodes are nodes 2 and 0. Also, M_7 is elected to be the sponsor. It renews K_2 and K_0, and broadcasts BK_2. All remaining members can then obtain the group key. It should be noted that although the keys of node 5 (i.e., the individual keys of M_7) remain intact, M_7 may have to broadcast the blinded key BK_5 since node 5, which is initially node 11, does not belong to the co-paths of M_5 and M_6 before the rekeying operation.

Based on the above join and leave events in Figure 2, if we simply change the member association of node 5 from M_4 to M_7, we can *save* one rekeying operation. This intuition motivates the use of interval-based rekeying in which the communication group performs rekeying on a batch of join and leave events periodically. We describe the interval-based rekeying approach in the following subsection.

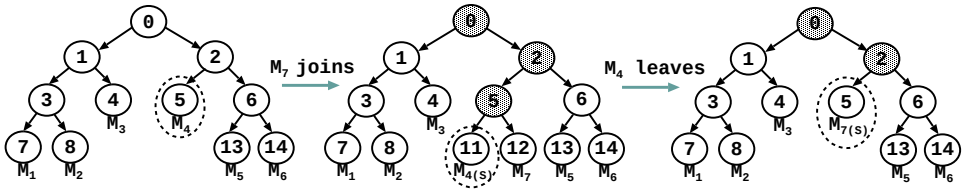


Fig. 2. TGDH rekeying at a single join and a single leave. Note that the shaded nodes are the nodes whose corresponding keys need to be renewed. Also, the notation $M_{i(s)}$ means that member M_i becomes a sponsor.

2.2. *Interval-Based Distributed Rekeying Algorithms*

In this subsection, we overview three interval-based distributed rekeying algorithms (Lee et al., 2002) (or interval-based algorithms for short), namely *Rebuild*, *Batch*, and *Queue-batch*, which are developed based on the TGDH algorithm and perform rekeying on a batch of join and leave events periodically at *regular intervals*. Interval-based rekeying maintains the rekeying frequency regardless of the dynamics of join and leave events, with a tradeoff of weakening both backward and forward confidentialities as a result of delaying the update of the group key. In practice, system administrators can decide the length of the rekeying interval to balance the tradeoff between performance and security.

The interval-based algorithms are developed based on the following assumptions:

– All group members are trusted in the key establishment process. This is justified since the group members participate in the secure group communication.
– The group communication satisfies *view synchrony* (Fekete et al., 1997; Kim et al., 2004) that defines reliable and ordered message delivery under the same membership view. Intuitively, when a member broadcasts a message under a membership view, the message is delivered to same set of members viewed by the sender. Note that this view-synchrony property is essential not only for group key agreement, but also for reliable multipoint-to-multipoint group communication in which every member can be a sender (Kim et al., 2004).
– Rekeying operations of all members are synchronized to be carried out at the beginning of every rekeying interval.
– To obtain the blinded keys of the renewed nodes, the key paths of the sponsors should contain those renewed nodes. Since the interval-based rekeying operations involve nodes lying on more than one key paths, more than one sponsors may be elected. Also, a renewed node may be rekeyed by more than one sponsor. Therefore, we assume that the sponsors can coordinate with one another such that the blinded keys of all the renewed nodes are broadcast only once.
– At the beginning of a rekeying operation, all members know the current key tree structure and the corresponding blinded keys in their own co-path.

In the subsequent sections, we discuss how the above assumptions are addressed in our implementation. In the following, we provide the basic ideas behind the three interval-based algorithms. Readers may refer to (Lee et al., 2002) for the detailed description and simulation analysis of the interval-based algorithms.

– **Rebuild:** The idea of Rebuild is to reconstruct the key tree to a *complete* tree in order to minimize the tree height and hence the rekeying steps by group members in subsequent rekeying intervals. All the non-leaf nodes need to be renewed. Also, we assume that all members become sponsors and they coordinate with each other in broadcasting the renewed blinded keys. Rebuild is suitable for some cases, such as when the membership events are so frequent that we can directly reconstruct the whole key tree for simplicity, or when some members lose the rekeying information and the simplest way of recovery is to rebuild the key tree.
– **Batch:** The main idea of Batch is to add joining members at suitable positions in the key tree. Given a set of joining and leaving members, Batch replaces leaving members with joining members since the keys held by the leaving members have to be renewed anyway. It then adds joining members to the shallowest possible positions in the key tree. Also, it tries to maintain a balanced key tree to prevent members from being located at deep positions and performing extensive rekeying steps.
– **Queue-batch:** Unlike Rebuild and Batch, Queue-batch pre-processes joining members within the idle period of every rekeying interval. This alleviates the processing load during the rekeying operation conducted at the beginning of every rekeying interval and hence speeds up the start of the secure communication with the latest group key. Queue-batch consists of two phases: *Queue-subtree* and *Queue-merge*. In the Queue-subtree phase, which occurs in the midst

of the rekeying interval, joining members are attached to a subtree T', like how TGDH manages the single join event. In the Queue-merge phase, which occurs at the start of the next rekeying interval, the subtree T' is attached to the key tree.

3. Design of SEAL

In this section, we present the design mechanisms of SEAL on how to enable a communication group to perform a rekeying operation (i.e., to renew the group key) in actual implementation.

3.1. *Summary of Design*

We first summarize how SEAL implements a rekeying operation:

(i) When a new member joins a communication group, it connects to a *Spread daemon* (Amir et al., 2004a), a group communication service that guarantees the reliable and ordered message delivery under the same membership view.

(ii) Among the existing group members, the communication group selects a *leader*, a single member that is responsible for synchronizing the rekeying operations carried out by all group members. At regular rekeying intervals, the leader notifies all other group members to start a new rekeying operation via the broadcast of a rekeying message.

(iii) Each group member updates its own key tree based on the agreed-upon interval-based algorithm (i.e., Rebuild, Batch, or Queue-batch) and checks whether it is a sponsor. Any member that becomes the sponsor will broadcast the updated blinded keys.

(iv) Each member carries out the *key confirmation* process (Ateniese et al., 1998) to assure that every other member has actually obtained the same group key. If this key confirmation process succeeds, then the rekeying operation is finished.

(v) If there are some members leaving the communication group in the middle of a rekeying operation (i.e., after the leader initiates the rekeying operation but before the group key is confirmed), the communication group either continues with its existing rekeying operation, or starts a rekeying operation reflecting the departures of those members.

3.2. *Reliable Group Communication*

Our implementation is built upon the *Spread* toolkit (Amir et al., 2004a), which implements the view synchrony property for reliable group communication. When a new member joins a communication group, it connects to a *Spread daemon*, a group communication service which maintains an active TCP connection to all other Spread daemons and keeps track of the current membership status of the communication group. Each Spread daemon can be associated with more than one member, so the group communication model forms a two-level hierarchy consisting of the Spread daemons and the group members. When a member joins or leaves the group, every member will receive the latest membership view from its associated Spread daemon. Also, the Spread daemon associated with the joining or leaving member notifies other daemons to flush the remaining messages to the original membership view and to block the transmission of new messages until all Spread daemons and existing group members install the updated membership view. Similarly, if a Spread daemon fails, the associated members are removed from the membership view by the remaining Spread daemons. Therefore, every existing group member always holds the latest membership view. Also, all messages are originated from the sender and delivered to all members under the same membership view, or equivalently between two consecutive membership events. To ensure the ordered delivery, the Spread daemons append a timestamp to every transmitted message.

Here, we implicitly assume that the Spread daemons always provide trusted membership views. Maintaining an authenticated membership view involves the change of implementation in Spread and is not the focus of this paper. We pose this problem as future work.

The Spread toolkit provides a set of API functions for members to send or receive messages through the Spread daemon under the view synchrony model. We use those API functions as the foundation for our implementation of the interval-based algorithms.

3.3. *Leader*

The *leader* is the single member that is responsible for periodically notifying all group members to start a rekeying operation synchronously at regular rekeying intervals. We select the member that stays in the communication group for the longest time to be the leader. When a group member joins the group as a new participant or is notified that the leader has left the group,

it decides which member should be the current leader based on the agreed-upon membership view provided by its associated Spread daemon (see Section 3.2). Since each member holds the identical membership view, any member that falsely claims to be the leader will be detected by other group members and excluded from the communication group. We point out that the leader is not a centralized key server that generates the group key, so the contributory requirement of our proposed algorithms still holds.

When a member is selected to be the leader of a communication group, it immediately broadcasts a *rekeying message* to the group and repeats the broadcast periodically at regular rekeying intervals. Since new members do not know the rekeying information including the present join and leave events as well as the existing key tree when they join the group, each rekeying message should contain the existing key tree as well as the join and leave requests in the last rekeying interval. The exact construction of the rekeying message will be discussed in Section 4.

It is possible that a newly selected leader does not know the current key tree structure. This occurs when it has just joined the group and has not started any rekeying operation. In this case, the leader should include only an empty tree and the join events in the *first* rekeying message. The leave events, however, are not required as they do not take effect in an empty tree.

3.4. *Sponsor*

Sponsors, as previously stated, refer to the group members that need to broadcast the blinded keys associated with the nodes in a key tree during a rekeying operation. Since each member holds the blinded keys along its own co-path, which is a list of nodes whose siblings belong to its key path, the sponsors have to broadcast the blinded keys of the non-renewed nodes, which are the children of the renewed nodes so that members can compute the secret keys of the renewed nodes. Broadcasting non-renewed blinded keys is essential for the new members which know nothing about the group before they join, as well as for the existing members whose co-path does not include the non-renewed nodes prior to the rekeying operation. (We will later illustrate how it helps the existing members with an example). Therefore, in our implementation, we appoint the sponsors to broadcast the blinded keys of two types of nodes: (1) all renewed nodes and (2) the non-renewed nodes whose parents are renewed nodes.

We first refine the sponsor election criterion as follows: in each rekeying interval, a member becomes a sponsor if it is the rightmost member of the subtree whose root is a non-renewed node but the parent of the root is a renewed node (if the member is the only member in the group, no sponsor is elected). It should be noted that all new members are elected to be sponsors based on this criterion.

After being elected, the sponsors have to decide the exact nodes whose corresponding blinded keys need to be broadcast. We call this decision-making process to be *sponsor coordination*, whose pseudo-code is presented in Figure 3. To understand how it works, consider Figure 4, in which the key tree contains a number of renewed nodes (i.e., nodes 0, 1, 2, and 6). Based on our sponsor election criterion, M_1, M_2, M_5, M_7, and M_8 are elected to be sponsors. According to the algorithm in Figure 3, member M_1 broadcasts BK_3, M_2 broadcasts BK_4 and BK_1, M_5 broadcasts BK_5, M_7 broadcasts BK_{13}, and M_8 broadcasts BK_{14}, BK_6, BK_2, and BK_0. Furthermore, Figure 4 illustrates the need of broadcasting the blinded keys of non-renewed nodes (i.e., nodes 3, 4, 5, 13, and 14) to existing members. For example, M_6 and M_7 do not hold the blinded key of node 14 if it is promoted from one of its child nodes since the node is not originally on their co-paths. Therefore, they have to obtain the blinded key from the sponsors.

The sponsor coordination process satisfies three properties. First, it is *self-computable* because it does not involve any communication between sponsors to determine which blinded keys are to be broadcast. Also, it is *lightweight* because it only requires a sponsor to traverse its key path *once*. Its worst-case complexity is $O(h)$, where h is the depth of the leaf node associated with the sponsor. Lastly, it is *broadcast-efficient* since it broadcasts the keys in a minimum number of rounds. To elaborate the last property, we consider an example in which a renewed node v_p is the root of a subtree and has the left child node v_l and the right child node v_r, whose corresponding sponsors are $M_{l(s)}$ and $M_{r(s)}$, respectively. To decide which sponsor is to be selected to broadcast the blinded key of v_p, we consider two cases. In the first case, if only one child node is renewed – say v_l is renewed but v_r is not – $M_{l(s)}$ can compute the secret key of v_p based on the unchanged blinded key of v_r. This implies that $M_{l(s)}$ can broadcast the blinded keys of v_l and v_p in one round. We therefore select $M_{l(s)}$ to broadcast the blinded key of v_p. In the second case, if both child nodes are renewed, i.e., v_l and v_r are renewed nodes, both sponsors have to wait for the updated blinded keys of v_l and v_r to compute the secret key of v_p. They need two rounds to broadcast the blinded key

Sponsor_Coordination (T) {T is the updated key tree with a set of renewed nodes}
1: $broadcast\_list$ = NULL
2: **if** holds the sponsor's responsibility **then**
3: k_node = leaf node of the member's key path
4: **while** $k\_node \neq T$'s root AND k_node's parent not renewed **do**
5: k_node = k_node's parent
6: insert k_node into $broadcast\_list$
7: **while** $k\_node \neq T$'s root **do**
8: **if** both k_node and k_node's sibling not renewed OR both k_node and k_node's sibling renewed **then**
9: **if** k_node is the right child **then**
10: insert k_node's parent into $broadcast\_list$
11: **else**
12: **break** the **while** loop
13: **end if**
14: **else if** k_node not renewed AND k_node's sibling renewed **then**
15: **break** the **while** loop
16: **else if** k_node renewed AND k_node's sibling not renewed **then**
17: insert k_node's parent into $broadcast\_list$
18: **end if**
19: k_node = k_node's parent
20: **end while**
21: **end while**
22: **end if**
23: return $broadcast\_list$;

Fig. 3. Pseudo-code of the sponsor coordination process.

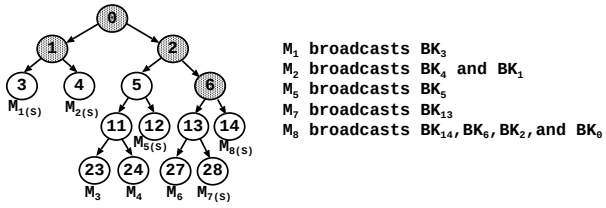


Fig. 4. Example to illustrate the sponsor coordination process in Figure 3.

of v_p. We can therefore select any one sponsor, say the sponsor $M_{r(s)}$ under the right child node, to take this responsibility. Combining the two cases, we can apply similar arguments when v_l is not a renewed node but v_r is and when both v_l and v_r are not renewed nodes.

3.5. *Key Confirmation*

Key confirmation (Ateniese et al., 1998) assures each member that all other members are actually holding the same group key. Providing complete key confirmation requires every member to demonstrate its knowledge of the group key to all other members. One way to achieve this is to ask every member to broadcast the one-way function result of the group key after it is generated. However, this involves many broadcasts and may be infeasible. In another approach given in (Just and Vaudenay, 1996), each member only needs to prove its knowledge of the group key to its neighbors, provided that all members are arranged in a ring. However, such an approach is vulnerable to the collusion attack (Just and Vaudenay, 1996).

In SEAL, we design a weaker but feasible key confirmation approach. We appoint a sponsor based on the sponsor-coordination algorithm to broadcast the blinded group key which lets every member verify if its computed blinded group key is identical to the one it receives. If a member finds that the keys are different, it will report the error. The rekeying operation is finished if every member verifies that both keys are identical.

We do not require any member to explicitly indicate the successful verification of the group key. When a member wants to broadcast a data message, it simply encrypts the message with the latest group key that has been obtained. Any member that has not yet confirmed the correctness of the group key will cache the message until the group key is verified. In general, the duration of a rekeying operation is on the order of seconds (see Section 6), so any cached encrypted message will be decrypted after a short while.

3.6. *Robustness*

We close this section by discussing how SEAL achieves robustness in a rekeying operation. It is possible that a group member leaves the group or encounters system failures during the execution of a rekeying operation. Depending on the type of the leaving member, we consider two cases. First, if the leaving member is neither the leader nor one of the sponsors, or if it is a sponsor but has broadcast all necessary blinded keys for the current rekeying operation, the communication group continues with the existing rekeying operation without being affected and the leave event is reflected in the next rekeying operation. Second, if the leaving member is the leader or a sponsor that has not yet broadcast all required blinded keys, the communication group first selects a new leader if the leaving member is the leader (see Section 3.3). Then the leader broadcasts a rekeying message to start a new rekeying operation which reflects the current leave event. Any renewed nodes whose blinded keys have not yet been broadcast remain renewed in the new rekeying operation. Also, the nodes that are on the key path of the leaving member become the renewed nodes. Given the set of renewed nodes, new sponsors are selected

to broadcast the updated blinded keys according to the sponsor-coordination algorithm (see Section 3.4). This so-called *self-stabilizing* property (Kim et al., 2004) is realized in our implementation.

In addition, within a single rekeying interval, a member may join and then leave the communication group, or it may leave and then join the group. In the former case, the membership events of the member are simply ignored in the next rekeying operation. The latter case, however, is treated as two separate membership events: the leave event of an existing member and the join event of a new member.

4. Implementation Details of SEAL

In this section, we present the implementation details of SEAL. We first describe the preliminary requirements for the library. Then we discuss the system components that constitute the implementation.

4.1. *System Preliminaries*

SEAL is implemented in C under Linux and requires two toolkits: *Spread* (Amir et al., 2004a) (see Section 3.2) and *OpenSSL* (Viega, 2002). Our implementation uses the exported API functions from Spread to send or receive packets according to the view-synchrony group communication model. OpenSSL is a security toolkit offering a cryptography library and a certificate generation tool. We use it to implement cryptographic algorithms as well as to create public-key certificates for the authentication of group members. Both toolkits are the pre-requisites for the SEAL development.

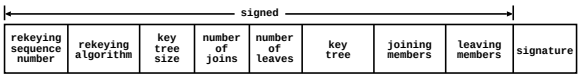
We require that each group member applies digital signatures to every packet that is to be sent to the network. Each group member should first obtain its public-key certificate from a trusted certificate authority (CA). Then the member signs the packets with its long-term (permanent) private key before the packets are sent over the network. In our implementation, we select the X509 certificate standard (ITU-T Recommendation X.509, 1993) and the 1024-bit RSA (Rivest et al., 1978) with SHA-1 (National Institute of Standards and Technology, 1995) signature scheme.

The implementation of SEAL contains several requirements. First, we require that both Spread and OpenSSL are pre-installed in a Linux system. Also, the Diffie-Hellman parameters, which are 1024-bit long in our implementation, should have been initialized and stored in the SEAL source directory before SEAL starts running. In addition, each group member should have a configuration file stating the unique member identifier, the membership details of all possible communication groups, and the connectivity information specifying which Spread daemon is to be connected. Furthermore, each group member should beforehand obtain its long-term private key and the certificates of other group members from a trusted CA. For consistency, a central repository can be set up to provide all necessary information related to the requirements.

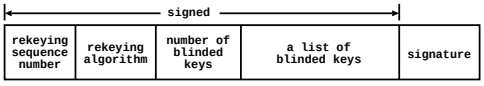
4.2. *System Components*

SEAL is composed of four types of components: (1) *engines*, the entities which hold variables and methods for various functionalities, (2) *queues*, the linked-list structures which store and dispatch packets in first-in-first-out order, (3) *threads*, the execution contexts which handle all protocol operations, and (4) *packets*, the information which is exchanged between group members.

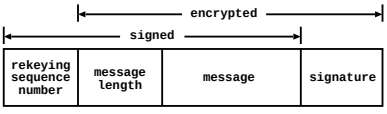
The SEAL packets are classified into two categories: *membership packets* and *regular packets*. Membership packets, including the JOIN, LEAVE, and DISCONNECT packets, are defined in the Spread specification (Amir et al., 2004a). They store the membership information essential for SEAL and the Spread daemons. Regular packets, however, are defined by SEAL. They are used by SEAL for rekeying operations and by underlying group-oriented applications for secure group communication. They include the REKEY, BKEY, and MESSAGE packets. Figure 5 illustrates the formats of the SEAL regular packets.



(a) REKEY packet



(b) BKEY packet



(c) MESSAGE packet

Fig. 5. Formats of the regular packets.

To achieve secure group communication, the member applications that have SEAL installed will exchange SEAL packets with one another. The general operations on the received packets can be summarized

into several steps: (1) The *received_thread* receives packets from the connected Spread daemon and adds them into the *packet_queue*. (2) The *process_thread* retrieves packets from the *packet_queue* if the queue is non-empty. (3) The *process_thread* verifies the signatures attached to the packets. (4) Based on the packet types, the *process_thread* carries out the corresponding operations with the *member_engine*, the *leader_engine*, the *keytree_engine*, the *sesskey_engine*, and the *message_queue*. (5) If the *process_thread* needs to send packets, it creates packets with the *packet_engine*. (6) The *process_thread* signs the packets with the *certkey_engine*. (7) Finally, the *process_thread* sends the packets via the *packet_engine* to the connected Spread daemon and then to the communication group. Figure 6 illustrates the operations on the received packets.

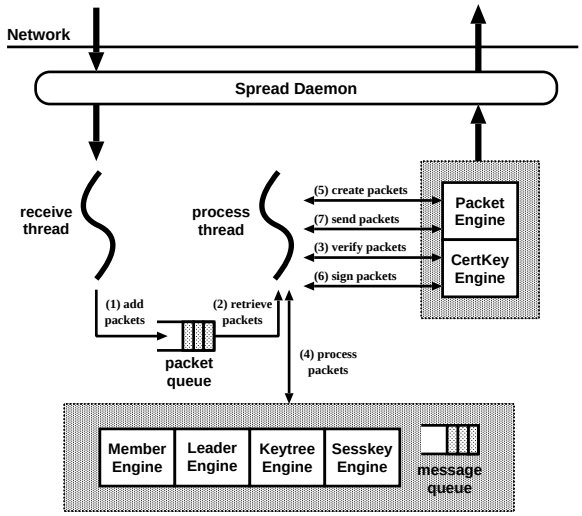


Fig. 6. General operations on the received packets.

Let us take a more detailed look at how the *process_thread* responds to the received packets according to different packet types:

- **Operations on a JOIN/LEAVE/DISCONNECT packet:** The *process_thread* inserts the joining or leaving member into the *member_engine*. Also, depending on the membership events, it performs leader election, and creates the leader-specific components if the member becomes the leader (the operations of the leader-specific components are described later in this subsection). It should be noted that the operations on the LEAVE and DISCONNECT packets are both identical.
- **Operations on a REKEY packet:** The *process_thread* first retrieves the rekeying information including the rekeying sequence number (the identifier of a REKEY packet and hence a rekeying operation), the joining and leaving members, as well as the key tree, from the received REKEY packet. The *process_thread* then starts the rekeying operation, which consists of (1) specifying the leader's identity in the *leader_engine*; (2) synchronizing the joining and leaving members with those in the *member_engine*; (3) updating the key tree in the *keytree_engine* based on the selected interval-based algorithm; and (4) updating the secret and blinded keys of the key path in the *sesskey_engine* and broadcasting any blinded keys if the group member becomes a sponsor.
- **Operations on a BKEY packet:** The *process_thread* obtains the blinded keys from the packet, which is composed of a sequence of blinded keys of some key tree nodes on a key path. If the blinded keys help the group key generation, the *process_thread* computes the secret keys along the key path, which is maintained by the *sesskey_engine*. Besides, if the group member is a sponsor and the *sesskey_engine* contains the new blinded keys to be broadcast, the *process_thread* will broadcast a BKEY packet consisting of the blinded keys.
- **Operations on a MESSAGE packet:** The *process_thread* inserts the MESSAGE packet, which contains the application-level messages, into the *message_queue*. The enqueued packets will later be processed by the application. Here, we derive three independent sub-keys from the latest group key and encrypt/decrypt the message content based on the Triple-DES-CBC standard (Schneier, 1996). In addition, we use the "signature before encryption" approach (Schneier, 1996) as shown in Figure 5.

If a group member is elected to be leader, the *process_thread* will create the leader-specific components, composed of the *rekey_poll_thread*, the *rekey_send_thread*, and the *rekey_queue*. To send a rekeying message to the group, the leader performs the following procedure: (1) The *rekey_poll_thread* periodically issues a rekeying signal (the indicator of performing a rekeying operation) into the *rekey_queue* and notifies the *rekey_send_thread* to send REKEY packets. The rekeying signal also specifies the interval-based algorithm to be performed. (2) When the *rekey_send_thread* is notified, it removes the rekeying signal from the *rekey_queue*. (3) The *rekey_send_thread* gathers the rekeying sequence number from the *leader_engine*, the joining and leaving members from the *member_engine*, and the existing key tree from the *keytree_engine*. (4) The *rekey_send_thread* constructs the REKEY packet based on the gathered details. (5) The *rekey_send_thread* signs the REKEY packet with the *certkey_engine*. (6) Finally, the *rekey_send_thread* sends the packet over

the network.

To ensure that each REKEY packet contains the latest information, the *rekey_send_thread* retrieves the next rekeying signal from the *rekey_queue* and sends its corresponding REKEY packet *only after* all engines are updated in response to the previous rekeying operation. Figure 7 illustrates the leader-specific components and their interactions with other components.

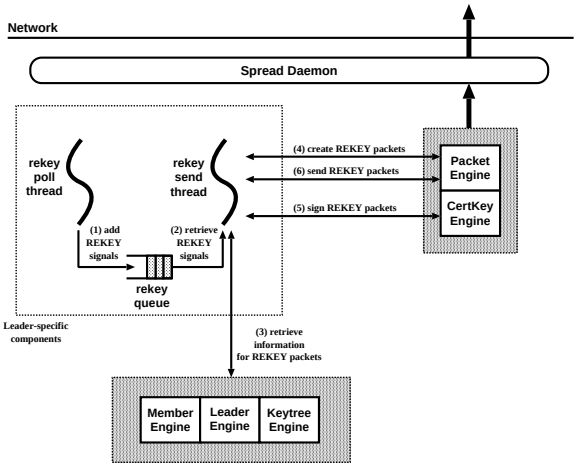


Fig. 7. Leader-specific components.

5. SEAL API

SEAL comprises a number of API functions that allow developers to implement the interval-based algorithms in their secure group-oriented applications. The operations of the API functions rely on a *SEAL session object*, which holds all the components constituting the system architecture of SEAL. Table 1 describes the details of the API functions.

Figure 8 illustrates the flowchart of using the SEAL API functions in a typical secure group-oriented application. The flow is described as follows: (1) the application instantiates a SEAL session object with *SEAL_init()*; (2) it retrieves the long-term private key from a password-protected file with *SEAL_set_passwd()*; (3) it joins a specified communication group with *SEAL_join()*; (4) it implements its application protocols with *SEAL_send()*, *SEAL_recv()*, and *SEAL_read_membership()* in order to send messages, receive messages, and read the current membership status, respectively; (5) it leaves the group with *SEAL_leave()*; and (6) it either joins another or the same group with *SEAL_join()*, or destroys the SEAL session object with *SEAL_destroy()* and ends.

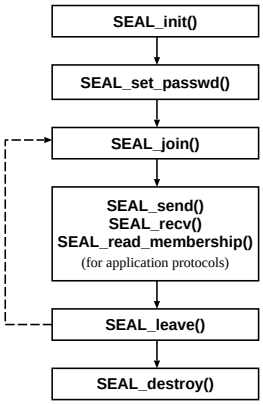


Fig. 8. Flowchart of using the SEAL API functions.

6. Experiments

In this section, we evaluate SEAL under real network settings. Using SEAL, we implemented *Gauger*, a member application that repeatedly joins and leaves a communication group at different times. We then used Gauger to measure the performance of a rekeying operation in Rebuild, Batch, and Queue-batch based on different levels of membership dynamics.

We make two remarks for our evaluation. First, as mentioned in Section 3.6, it is possible that at least one sponsor leaves the group in the middle of the rekeying operation. In this case, a member will receive more than one rekeying message from the leader before the group key is confirmed. Therefore, when we consider the duration of a rekeying operation, it starts from the instant at which a member receives the first rekeying message that indicates the beginning of a rekeying operation to the instant at which the updated group key is confirmed. Second, for Queue-batch, we only consider the Queue-merge phase but not the Queue-subtree phase. The pre-processing steps in the latter do not influence the underlying secure communication, which is protected with the current group key. Thus, by focusing on the Queue-merge phase, our measurements reflect the latency of generating the latest group key starting from the beginning of a rekeying interval.

Our experiment consider the following metrics for a rekeying operation:

– **Rekeying time**: It measures the duration (in seconds) of a rekeying operation.
– **Number of exponentiations**: It measures the computational cost involving the exponentiation operations in the secret key and blinded key computations.
– **Number of broadcast blinded keys**: It measures the communication cost involving the number of blinded

Table 1
Description of the SEAL API functions.

Functions	Synopsis	Return values
SEAL_init()	It creates and initializes an SEAL session object for subsequent SEAL operations.	An initialized session object on success and NULL on failure
SEAL_set_passwd()	It retrieves the long-term private key from a password-protected file.	1 on success and 0 on failure
SEAL_join()	It connects to the Spread daemon and joins the specified communication group. It also initializes all components inside the SEAL session object.	1 on success and 0 on failure
SEAL_send()	It encrypts the application messages with the current group key and sends them to the communication group.	1 on success and 0 on failure
SEAL_recv()	It receives the application messages from the communication group and decrypts them with the current group key.	1 on success and 0 on failure
SEAL_read_membership()	It reports the current group membership status including the existing members, the joining members and the leaving members.	1 on success and 0 on failure
SEAL_leave()	It disables any operations and frees the resources of all components inside the SEAL session object. It then leaves the communication group and disconnects from the Spread daemon.	1 on success and 0 on failure
SEAL_destroy()	It destroys the SEAL session object and free its resources.	1 on success and 0 on failure

keys being broadcast within a rekeying operation.

– **Number of BKEY packets** (see Section 4.2): It measures the number of broadcast packets of blinded keys during a rekeying operation and hence the computational cost of signing or verifying the broadcast packets. Note that each packet may contain more than one blinded key depending on the number of blinded keys that can be computed in one round.

We point out that the measured rekeying time has a higher value and a higher deviation than does the actual rekeying time because of the background processes within the testbed machines. On the other hand, the other three metrics are attributed to the rekeying algorithms only.

6.1. *Wired LAN*

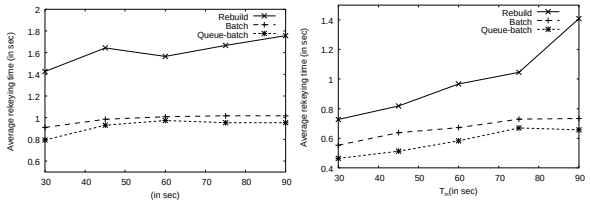
We start our evaluation with a communication group residing in a wired LAN based on the following configurations. We are interested in a single communication group with the group population size of 40 Gaugers. We assign the group members evenly to eight Pentium 4/2.5GHz Linux machines (i.e., each machine has five Gaugers installed). All eight machines are interconnected in a single LAN and are reachable from each other through broadcasts. A Spread daemon with configured parameters is running on each machine. When a Gauger starts execution, it connects to the Spread daemon in the same local machine. In addition, we fix the length of the rekeying interval to be 15 seconds. To observe the effect of the length of the rekeying interval, we vary the frequencies of join and leave events in our experiments. Analysis of varied rekeying intervals can be found in (Setia et al., 2000).

Each Gauger repeatedly joins and leaves the same communication group. The duration for which a Gauger stays inside and outside the group are respectively given by $T_{in} + c$ and $T_{out} + c$, where T_{in} and T_{out} are exponentially distributed, and c denotes a constant period specifying the minimum duration between two membership events for a group member. The reason that we add a constant to the time lengths is to prevent a member from joining or leaving the communication group abruptly so as to guarantee a sufficient amount of time for the resources to be re-allocated between membership events. Throughout our experiments, we set c to be 10 seconds, meaning that a member may join (resp. leave) and then leave (resp. join) the communication group within a single rekeying interval (see Section 3.6). We perform our measurements for two hours for each specified pair of T_{in} and T_{out}.

Experiment 1 (Average analysis of Rebuild, Batch, and Queue-batch with fixed T_{out}'s and varied T_{in}'s): In this experiment, we evaluate the performance metrics of Rebuild, Batch and Queue-batch. We fix T_{out} to be 30 and 90 seconds and vary T_{in}. The reason of fixing T_{out}'s is to control the rate that members join the communication group. The metrics are averaged over the number of existing members in the group at each rekeying interval and then over the number of rekeying intervals.

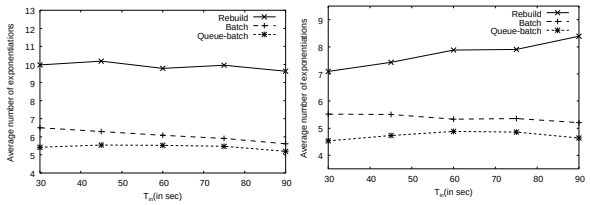
Figures 9(a)-(h) present the results. The metric costs at T_{out} = 30 seconds are larger than those at T_{out} = 90 seconds. With a smaller T_{out}, more members stay in

the group and participate in a rekeying operation. Thus, the size of the key tree is larger and more operations are required to generate a group key. Also, we observe Queue-batch outperforms the other two algorithms in all metrics. For instance, its rekeying time is less than one second on average for all values of T_{in} and T_{out}. This shows the performance gain of Queue-batch by dispersing its rekeying workload throughout the rekeying interval. Although Rebuild has the worst performance as compared to other two algorithms, it provides a simple way to recover the whole key tree (see Section 2.2).



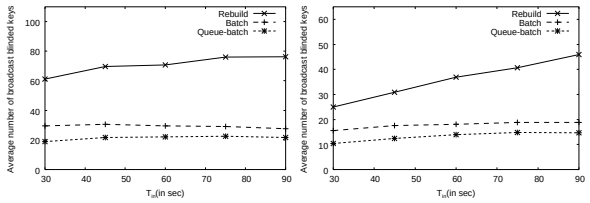
(a) Average rekeying time, $T_{out} = 30s$

(b) Average rekeying time, $T_{out} = 90s$



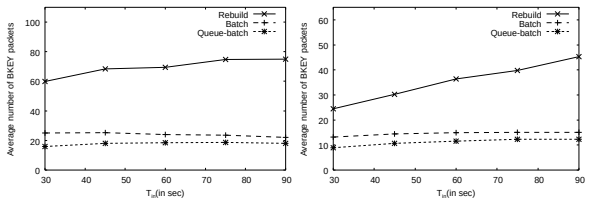
(c) Average no. of exponentiations, $T_{out} = 30s$

(d) Average no. of exponentiations, $T_{out} = 90s$



(e) Average no. of broadcast blinded keys, $T_{out} = 30s$

(f) Average no. of broadcast blinded keys, $T_{out} = 90s$



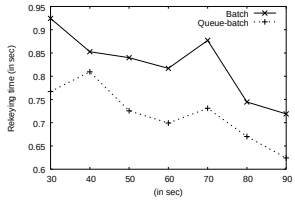
(g) Average no. of BKEY packets, $T_{out} = 30s$

(h) Average no. of BKEY packets, $T_{out} = 90s$

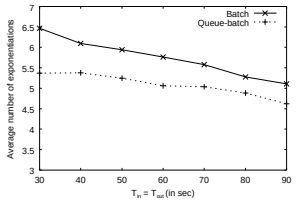
Fig. 9. Experiment 1: Average analysis at different fixed T_{out}'s.

Experiment 2 (Average analysis of Batch and Queue-batch under different levels of membership dynamics): This experiment investigates the performance of Batch and Queue-batch with respect to the frequencies of the join and leave occurrences. We set T_{in} equal to T_{out} and change the pair of T_{in} and T_{out} from 30 seconds to 90 seconds. The smaller the values of T_{in} and T_{out} are, the higher the membership dynamics is. We average the metrics as in Experiment 1.

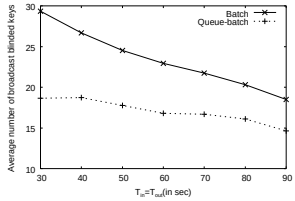
Figures 10(a)-(d) depict the results. As stated at the beginning of this section, the rekeying time has a higher deviation as compared to the other metrics. Nevertheless, for both Batch and Queue-batch, the results of all metrics show a decreasing trend when the membership events occur less frequently (i.e., when T_{in} and T_{out} are larger). Figures 10(b)-(d) show that Queue-batch performs much better than Batch when the group is highly dynamic (i.e., when T_{in} and T_{out} are small). This conforms to the simulation results in (Lee et al., 2002).



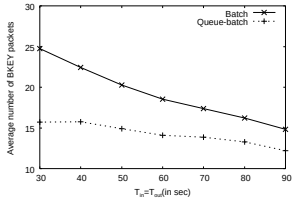
(a) Average rekeying time



(b) Average no. of exponentiations



(c) Average no. of broadcast blinded keys



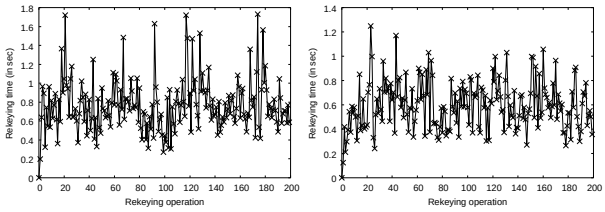
(d) Average no. of BKEY packets

Fig. 10. Experiment 2: Average analysis at different levels of membership dynamics.

Experiment 3 (Analysis of Queue-batch in each rekeying operation): In Section 3.6, we mention how our implementation achieves robustness in response to the case where at least one of the sponsor leaves the group in the middle of a rekeying operation. In this experiment, we evaluate how Queue-batch addresses this robustness issue in each rekeying operation. We consider two cases of membership dynamics where $T_{in} = T_{out}$ = 30 seconds and $T_{in} = T_{out}$ = 90 seconds, corresponding to the high and low membership dynamics, respectively. Here, we focus on the rekeying time of

each operation, which is averaged over the number of members participating in the rekeying operation.

Figures 11(a)-(b) illustrate the rekeying times of the first 200 rekeying operations. The "spikes" shown in the figures indicate that the leader restarts the current rekeying operation in response to the departures of the previous leader or the sponsors that have not yet broadcast the blinded keys for which they are responsible. Figure 11(a) shows more spikes than Figure 11(b), indicating that under higher membership dynamics (i.e., smaller T_{in} and T_{out}), more members leave the group during a rekeying operation, and thus the leader needs to restart rekeying more frequently. In general, our implementation can adapt quickly to the sponsors' departures during rekeying and complete the whole rekeying operation in no more than two seconds. This demonstrates the robustness of our implementation. We point out that this robustness property is achievable in our implementation regardless of the choice of the interval-based algorithms.



(a) $T_{in} = T_{out} = 30s$ (b) $T_{in} = T_{out} = 90s$

Fig. 11. Experiment 3: Analysis of Queue-batch in each rekeying operation.

6.2. *Comparison Between a Wired LAN and a Wireless LAN*

We envision that the hybrid wired/wireless LAN environment would be a common platform for most reliable peer communication applications. Thus, in this subsection, we further evaluate SEAL using both wired and wireless LAN testbeds. Our objective is to study how a wireless LAN, which offers less available communication bandwidth than does a wired LAN, affects the rekeying performance of SEAL.

Figure 12 illustrates the wired and wireless LAN testbeds that are used in our evaluation. In Figure 12(a), we interconnect three Pentium 4/1.8GHz Linux machines with a 100Mbps Fast Ethernet switch and thus form a single wired LAN. Each machine has a Spread daemon installed and is associated with five Gaugers. This wired LAN testbed serves as a baseline for performance comparison. In Figure 12(b), we interconnect the same three machines over a campus-wide wireless LAN, which follows the IEEE 802.11g standard and has a maximum data rate at 54Mbps.

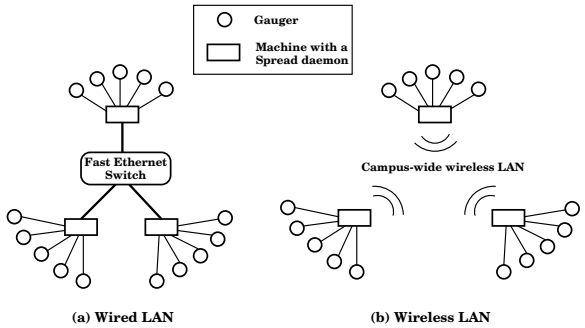


Fig. 12. The wired and wireless LAN testbeds for the evaluation of rekeying performance.

Our measurements are based on the same setting as in Section 6.1, i.e., each of the 15 Gaugers iteratively joins and leaves the same communication group according to the values of T_{in} and T_{out}. Here, we use Queue-batch as the rekeying algorithm. values of T_{in} and T_{out}.

Table 2 presents the metric results for both wired and wireless LAN testbeds when $T_{in} = T_{out}$ = 30, 60, and 90 seconds. Overall, the rekeying time in the wired LAN testbed is about 0.40-0.45 seconds, while that in the wireless LAN testbed is about 0.48-0.53 seconds. As pointed out at the beginning of this section, the measured rekeying time is subject to a high deviation, so we conjecture that the difference between the actual rekeying times in both testbeds could be smaller. In fact, for every pair of T_{in} and T_{out}, we note that both testbeds have very close results in each of the metrics except the rekeying time, with difference no more than 3%. Therefore, the rekeying performance is still preserved in the wireless LAN testbed despite its less available communication bandwidth.

Although the wireless LAN testbed may degrade the performance of the Spread daemons in maintaining view synchrony, we believe that the impact is quite insignificant. Amir et al. (2004a) showed that reliable group communication is scalable to at least 20 Spread daemons with 1000 members in a Fast Ethernet network and that messages can be delivered to all members reliably on the order of milliseconds. Since each of our testbeds only involves three Spread daemons, the overhead incurred by the Spread daemons should be minimal.

This experiment aims to provide a preliminary insight on how the deployment platforms with different communication constraints influence the rekeying per-

Table 2
Metric results for both wired and wireless LAN testbeds, with group size = 15. Note that the percentage difference is computed with respect to the metrics in the wired LAN testbed.

	$T_{in} = T_{out}$ = 30 secs			$T_{in} = T_{out}$ = 60 secs			$T_{in} = T_{out}$ = 90 secs		
	Wired	Wireless	Diff.	Wired	Wireless	Diff.	Wired	Wireless	Diff.
Rekeying time (in seconds)	0.45	0.48	6.7%	0.43	0.53	23.3%	0.40	0.52	30.0%
No. of exponentiations	4.08	4.07	-0.2%	3.81	3.83	0.5%	3.66	3.73	1.9%
No. of broadcast blinded keys	7.16	7.13	-0.4%	6.70	6.67	-0.4%	6.39	6.56	2.7%
No. of BKEY packets	6.24	6.24	0.0%	5.83	5.79	-0.7%	5.59	5.76	3.0%

formance. We emphasize that the efficacy of SEAL also depends on a variety of factors, including the computational resources, the size of a communication group, the membership dynamics, and the choices of cryptographic parameters. Also, connections with very scarce bandwidth (e.g., dialup) can severely impair the performance of SEAL and the Spread daemons. We plan to study these factors in our future work.

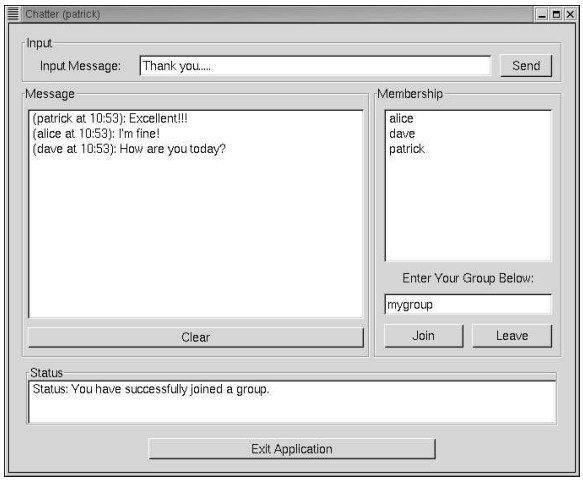
7. Applications

To demonstrate how SEAL can be used in real-life applications, we use the SEAL API functions to build a secure chat-room application called *Chatter*. The Chatter application encrypts the chat-room messages based on the group key (see Section 4 for implementation details). Thus, the group communication is kept confidential against group outsiders.

We implemented Chatter in both graphical and text modes. Its screenshots are illustrated in Figure 13. The graphical mode is built atop the X Window system in Linux. The text mode, on the other hand, is built to provide users with a text-based command console without the need of any graphical-compliant platform. Both interface modes are compatible and can be used together within the same communication group.

In addition to the chat-room applications, SEAL is feasible in a number of potential applications:

– **Audio/video conferencing systems:** Business parties may hold audio/video conferences with their laptop or desktop computers. The conferencing systems usually transfer massive streaming data which may contain confidential business information. They can hence use SEAL to agree upon a secret group key to encrypt the streaming data.
– **File sharing tools:** File sharing is prevalent in peer-to-peer networks. Most shared files usually do not involve sensitive information, but some do. Therefore, if a file sharing application intends to distribute a private file to a group of users, SEAL will help protect the file data.



(a) Graphical mode

(b) Text mode

Fig. 13. Illustration of Chatter.

– **Programmable router communication:** Software programmable routers (Kohler et al., 2000; Yau et al., 2005) provide programmable flexibility over traditional layer-3 routers. One application is to defend against distributed denial-of-service (DDoS) attacks (Yau et al., 2005). In this case, a communication group is formed among routers. Any information exchanged within the communication group should be inaccessible to the attackers in order for the defense

mechanism to succeed. Therefore, SEAL can be used to protect the exchanged information.

– **Network games in strategy planning:** In network games, players may co-operate with each other in deciding the winning strategies over other competitors. This type of interaction involves numerous message exchanges. Thus, the games can use SEAL to encrypt the messages against any cheating attacks, such as eavesdropping and modification of the transmitted game data.

In short, SEAL implements the interval-based algorithms that achieve group key agreement without any centralized key server, and hence is adequate for any secure group-oriented applications in decentralized environments such as peer-to-peer networks or mobile ad hoc networks.

8. Related Work

To achieve secure group communication, Wallner et al. (1997) and Wong et al. (2000) independently proposed the key tree approach that associates keys in a hierarchical tree and rekeys in each join or leave event. Li et al. (2001) and Yang et al. (2001) later applied the periodic rekeying concept in Kronos (Setia et al., 2000) to the key tree setting. All the key-tree-based approaches (Li et al., 2001; Wallner et al., 1997; Wong et al., 2000; Yang et al., 2001) require a centralized key server for key generation.

Burmester and Desmedt (1995); Kim et al. (2001, 2004); Steiner et al. (1998) extend the Diffie-Hellman protocol (Diffie and Hellman, 1976) to group key agreement schemes for secure communications in a peer-to-peer network. Burmester and Desmedt (1995) proposed a computation-efficient protocol at the expense of high communication overhead. Steiner et al. (1998) developed *Cliques*, in which every member introduces its key component into the result generated by its preceding member and passes the new result to its following member. Cliques is efficient in rekeying for leave or partition events, but imposes a high workload on the last member in the chain. Kim et al. (2004) proposed TGDH to arrange keys in a tree structure. Every member holds the keys along its key path and the rekeying workload is distributed to all members. The settings of TGDH are similar to the One-Way Function Tree (OFT) scheme (Sherman and McGrew, 2003) except that TGDH uses Diffie-Hellman instead of one-way functions for the group key generation. Kim et al. (2001) also suggested a variant of TGDH called STR which minimizes the communication overhead by trading off the computational complexity. All the above schemes are decentralized and hence avoid the single-point-of-failure problem in the centralized case, though they involve higher message traffic for distributed communication. Kim et al. (2001, 2004); Steiner et al. (1998) consider rekeying at single join, single leave, merge, or partition events. Our paper considers the more general case with a batch of join and leave events.

Amir et al. (2004b); McDaniel et al. (2001); Wong and Lam (2000) focus on the implementation issues of secure group communication. Antigone protects the group communication data via the specification of security policies. In terms of group key management, *Keystone* (Wong and Lam, 2000) considers the architectural design of the centralized group key distribution protocol based on the key tree setting. It uses UDP over IP and forward error correction (FEC) to provide efficient and reliable delivery of keys. Another work that is closely related to ours is *Secure Spread* (Amir et al., 2004b), which implements a centralized group key distribution protocol and a number of distributed group key agreement protocols including the Burmester-Desmedt model (Burmester and Desmedt, 1995), Cliques (Steiner et al., 1998), TGDH (Kim et al., 2004), and STR (Kim et al., 2001). The project studies the group key management schemes under join, leave, merge, and partition events, and provides a set of function calls suitable for secure application development. In our work, we implemented a programming library based on the interval-based approach and built applications with the library to demonstrate its strengths and effectiveness.

9. Conclusion and Future Directions

This paper presents SEAL, an API library that implements the group key agreement protocols. The library allows a dynamic peer group to undergo secure group communication through the establishment of a common secret group key without any centralized key server. We described the implementation framework for SEAL and addressed several design issues for implementing the group key agreement protocols. We used a set of API functions exported from SEAL to build two group-oriented applications termed Gauger and Chatter. We used Gauger as a performance testing tool to evaluate the group key agreement algorithms in a local area network and to demonstrate the robustness of our implementation during a rekeying operation. We also used Chatter, a secure chat-room application, to illustrate how developers can write secure group-oriented applications readily.

Several enhancements can be made to enrich our current implementation. First, our current implementation assumes that multiple groups independently manage their own key tree. If there exist some members associated with more than one group, then it is possible to bundle multiple key trees together to facilitate the implementation (Jung et al., 2003). Second, we currently assume that Spread daemons provide trusted membership information. It would be interesting to incorporate membership authentication into our future implementation. Finally, we seek to expand the scale of our evaluation and investigate the scalability of our implementation.

References

Amir, Y., Danilov, C., Miskin-Amir, M., Schultz, J., Stanton, J., 2004a. The Spread Toolkit: Architecture and Performance. CNDS-2004-1, Johns Hopkins University.

Amir, Y., Kim, Y., Nita-Rotaru, C., Schultz, J. L., Stanton, J., Tsudik, G., May 2004b. Secure Group Communication Using Robust Contributory Key Agreement. IEEE Transactions on Parallel and Distributed Systems 15 (5), 468–480.

Ateniese, G., Steiner, M., Tsudik, G., November 1998. Authenticated Group Key Agreement and Friends. In: Proc. of 5th ACM Conference on Computer and Communications Security. pp. 17–26.

Burmester, M., Desmedt, Y., 1995. A Secure and Efficient Conference Key Distribution System. Advances in Cryptology – EUROCRYPT '94 950, 275–286.

Diffie, W., Hellman, M., 1976. New Directions in Cryptography. IEEE Transactions on Information Theory IT-22 (6), 644–654.

Fekete, A., Lynch, N., Shvartsman, A., August 1997. Specifying and Using a Partionable Group Communication Service. In: ACM PODC'97. pp. 53–62.

ITU-T Recommendation X.509, November 1993. Information Technology–Open Systems Interconnection–The Directory: Authentication Framework.

Jung, E., Liu, A. X., Gouda, M. G., 2003. Key Bundles and Parcels: Secure Communication in Many Groups. In: Proc. of the 5th International Workshop on Network Group Communication (NGC). LNCS 2816, pp. 119–130.

Just, M., Vaudenay, S., 1996. Authenticated Multi-Party Key Agreement. In: Advances in Cryptology ASIACRYPT '96. LNCS 1163, Springer-Verlag, pp. 36–49.

Kim, Y., Perrig, A., Tsudik, G., November 2001. Communication-Efficient Group Key Agreement. Information Systems Security, Proceedings of the 17th International Information Security Conference IFIP SEC'01.

Kim, Y., Perrig, A., Tsudik, G., Feb 2004. Tree-Based Group Key Agreement. ACM Trans. on Information and System Security 7 (1), 60–96.

Kohler, E., Morris, R., Chen, B., Jannotti, J., Frans-Kaashoek, M., Aug 2000. The Click Modular Router. ACM Transactions on Computer Systems 18 (3), 263–297.

Lee, P. P. C., Lui, J. C. S., Yau, D. K. Y., November 2002. Distributed collaborative key agreement protocols for dynamic peer groups. In: Proc. of the 10th IEEE International Conference on Network Protocols (ICNP). pp. 322–333.

Li, X. S., Yang, Y. R., Gouda, M. G., Lam, S. S., May 2001. Batch Rekeying for Secure Group Communications. In: Proc. of Tenth International World Wide Web Conference (WWW10). Hong Kong, China, pp. 525–534.

McDaniel, P., Prakash, A., Irrer, J., Mittal, S., Thuang, T., June 2001. Flexibly Constructing Secure Groups in Antigone 2.0. In: Proc. of DARPA Information Survivability Conference and Exposition II. pp. 55–67.

National Institute of Standards and Technology, April 1995. The secure hash algorithm (SHA-1). NIST FIPS PUB 180-1, U.S. Department of Commerce.

Rivest, R., Shamir, A., Adleman, L., February 1978. A method for obtaining digital signatures and public key cryptosystems. Communications of the ACM.

Schneier, B., 1996. Applied Cryptography. Wiley.

Setia, S., Koussih, S., Jajodia, S., May 2000. Kronos: A Scalable Group Re-Keying Approach for Secure Multicast. In: Proc. of IEEE Symposium on Security and Privacy 2000. pp. 215–228.

Sherman, A. T., McGrew, D. A., May 2003. Key Establishment in Large Dynamic Groups Using One-Way Function Trees. IEEE Trans. Software Eng. 29 (5), 444–458.

Steiner, M., Tsudik, G., Waidner, M., May 1998. Key Agreement in Dynamic Peer Groups. IEEE Transactions on Parallel and Distributed Systems 11 (8), 769–780.

Viega, J., 2002. Network Security with OpenSSL. O'Reilly.

Wallner, D. M., Harder, E. J., Agee, R. C., July 1997. Key Management for Multicast: Issues and Architectures. Internet draft draft-wallner-key-arch-00.txt, Internet Engineering Task Force.

Wong, C. K., Gouda, M., Lam, S. S., Feb 2000. Secure Group Communications Using Key Graphs. IEEE/ACM Trans. on Networking 8 (1), 16–30.

Wong, C. K., Lam, S. S., May 2000. Keystone: A Group Key Management Service. In: Proc. of International Conference on Telecommunications. Acapulco, Mexico.

Yang, Y. R., Li, X. S., Zhang, X. B., Lam, S. S., August 2001. Reliable Group Rekeying: A Performance Analysis. Proc. of ACM SIGCOMM'01.

Yau, D. K. Y., Lui, J. C. S., Liang, F., Yam, Y., Feb 2005. Defending Against Distributed Denial-of-Service Attacks with Max-min Fair Server-centric Router Throttles. IEEE/ACM Transactions on Networking 13 (1).