Ensemble Security

Ohad Rodeh, Kenneth P. Birman, Mark Hayden, Zhen Xiao, Danny Dolev,
{orodeh,dolev}@cs.huji.ac.il, hayden@pa.dec.com, {ken,xiao}@cs.cornell.edu *

September 8, 1998

Abstract

Ensemble is a Group Communication System built at Cornell and the Hebrew Universities. It allows processes to create *process groups* in which scalable reliable fifo-ordered multicast and point-to-point communication are supported. The system also supports other communication properties, such as multicast causal and total ordering, flow control, etc..

This paper describes the security protocols and infrastructure of Ensemble. Applications using Ensemble with the extensions described here benefit from strong security properties. Assuming trusted authenticated members may not be corrupted, all communication is secured from tampering by outsiders.

Our work extends previous work performed in the Horus system (Ensemble's predecessor) adding support for multiple partitions, efficient rekeying, and application defined security policies. Unlike Horus, which used its own security infrastructure with non-standard key distribution and timing services, Ensemble's security mechanism is based on a standard and very widely used security infrastructure: the PGP authentication engine.

*The authors were supported in part by DARPA ITO under ARPA/ONR contract N0014-96-1-10014 and ARPA/RADC contract F30602-95-1-0047, in part by grants from Microsoft and AT&T, and by the Israeli Ministry of Science grant number 032-7892.

1 Introduction

Group Communication Systems (GCSs) are used today in industry where reliability and high-availability are required. Group Communication is a subject of ongoing research and many GCSs have been built [6, 30, 31, 19, 4, 26, 27], some of them commercial products [14, 17]. As the Internet comes into mainstream use, network security becomes a major concern. Group communication, running over the Internet protocols, needs to be efficiently protected against malicious network attacks. This paper describes the security architecture of Ensemble [26], our group communication system.

A GCS introduces the *process group* abstraction. A process group coherently binds together many processes into one entity. Within the context of a group, reliable per-source fifo ordered messaging is supported. Processes may dynamically join and leave a group. Groups may dynamically partition into many components due to network failures/partitions. The GCS is responsible for simplifying these complex scenarios, overcoming the asynchronous nature of the network and keeping the group abstraction consistent. Processes are provided with membership notifications specifying the list of currently alive and connected group members. A notification is provided whenever network connectivity changes or when processes join/leave the group. The *Virtual Synchrony Model* [15, 5] specifies the relationship between message delivery and membership notification.

Ensemble has been developed at Cornell and the Hebrew Universities. It is written in a dialect of the ML programming language [29] in order to facilitate system verification. The design methodology behind Ensemble stresses modularity and flexibility. Thus, Ensemble is divided into many *layers*, each implementing a simple protocol. Stacking together these layers, much like one uses lego blocks, the user may customize the system to suit its needs.

Underlying our work are two fundamental presumptions: First, we have access to a standard off-the-shelf authentication mechanism; Second, the application itself can perform authorization. To secure group messages from tampering and eavesdropping, they are all signed and encrypted. While it is possible to use public key cryptography for this task, we find this approach unacceptably expensive. Since all group members are mutually trusted, we share a symmetric encryption and signature key among them. This key is used to seal all group messages, making the seal/unseal operation very fast (roughly 1000 times faster). Using such a key raises two challenges:

A rekeying mechanism: allowing secure replacement of the current group key once it is deemed insecure, or if there is danger that it was leaked to the adversary. This is challenging since switching to a new key should be done without using the old, possibly compromised key for dissemination. Naturally, one could use public keys for this task yet this leads to high latency.

If one assumes the simple primary partition model, where only a single component of the group may function, then a simple solution is available. One may install a centralized key-server whose responsibility is to disseminate, revoke, and refresh group keys. Only group members in contact with the server may function. Supporting multiple partitions is more difficult since one cannot rely on any centralized service.

Secure key agreement in a group: a protocol that creates secure agreement among group members on a mutual key. Such a protocol should not restrict the Ensemble stack, i.e., all legal layer combinations should still be possible, it should be unobtrusive, and support multiple partitions.

Such a protocol should efficiently handle the case where two group components merge after a network partition where components use different keys. A simple approach (taken for example in [7]) is to add members from the smaller group to the larger one. However, this is inefficient since members are added one at a time, incurs cost proportional to the number of added members

The contributions of this paper are as follows:

- We support security properties for multiple partitions. Earlier work either does not address the issue of group partition or only supports security semantics for the primary partition [27].
- Support for dynamic application defined authorization polices.
- Our work is incorporated in a freely available general purpose group communication system. Hence, in addition to the traditional properties provides by a GCS, applications using our system can benefit from strong security guarantees.
- Ensemble is the first freely available secure group communication system.

We focus on benign failures and assume that authenticated members will not be corrupted. Byzantine fault tolerant systems have been built by other researchers [23, 16], but suffer from poor performance since they use costly protocols and make extensive use of public key cryptography. We believe that our failure model is sufficient for the needs of most practical applications. As demonstrated in the performance section, our system has good performance and scalability.

Group communication has inherently limited scalability. For example, Transis [30] scales to 30 members and Ensemble scales to 100 members. Since a group should be resilient to all scenarios of network partitions, each group component should be completely autonomous. Therefore, our architecture does not rely on any centralized servers or services.

Throughout this paper we use the term authentication or signature refering both to public key signature and to keyed-MD5[1] signatures.

This paper is structured as follows: Section 2 describes the model we use to describe the system and the model of attack. Section 3 describes Ensemble routers and the secure router we have added. Section 4 describes our key agreement protocol — Exchange, section 5 sketches a proof of its correctness. Section 6 describes the Rekey protocol and its optimization. Section 7 describes the Encrypt layer, Section 8 describes system performance, and Section 9 lists related work. Section 10 gives our conclusions and section 11 contains acknowledgements. The appendix contains some protocol details that were removed from the main body of the protocols for exposition clarity.

2 Model

Consider a universe that consists of a finite group $\mathcal{U}$ of n processes. Processes communicate with each other by passing messages through a network of channels. The system is asynchronous: clock drifts are unbounded and messages may be arbitrarily delayed or lost in the network. We do not consider Byzantine failures.

[1] MD5 where the IV (Initial Vector) is fed by a secret key.

Processes may be *partitioned* from each other. A partition occurs when $\mathcal{U}$ is split into a set $\{P_1, \ldots, P_k\}$ of disjoint subsets. Each process in P_i can communicate only with processes in P_i. The subgroups P_i are sometimes called *network components*, or simply Ncomponents. We shall consider *dynamic partitions*, when network components may dynamically merge and split. W.L.O.G., when process $p \in P_1$ crashes we say that the network has partitioned into Ncomponents $\{p\}, P_1 \setminus \{p\}, P_2, \ldots, P_k$. When p recovers, its Ncomponent ($\{p\}$) merges with the rest of $\mathcal{U}$. This allows us to restrict our terminology to merge and partition yet handle the case of process crash. Processes may send point-to-point or multicast messages within $\mathcal{U}$.

A GCS creates process groups in which reliable fifo-ordered multicast and point-to-point messaging is supported. Processes may dynamically join and leave a group. Groups may dynamically partition into many *components* due to network failures/partitions; when network partitions are healed group components remerge through the GCS protocols. Information about groups is provided to group members in the form of *view* notifications. For a particular process p a view contains the list of processes currently alive and connected to p. When a membership change occurs due to a partition or a group merge, the GCS goes through a (short) phase of reconfiguration. It then delivers a new view to the applications reflecting the (new) set of connected members.

In what follows p, q, and s denote Ensemble processes and V_1, V_2 denote views.

The Virtual Synchrony (VS) model describes the relative ordering of messages and view notifications, in Ensemble we use the *Strong Virtual Synchrony* (SVS) [8] variant of VS. If member p is in view V_1 then a membership change occurs and p receives view V_2 then we say the p passes from V_1 to V_2. We say that if p passes from view V_1 to V_2 then all messages received in the interim were received in V_1. Briefly, a system following the SVS model guarantees two properties:

Self Delivery: If process p in view V sends message m then it receives m in view V.

Agreement: If p passes from view V_1 to V_2 and $q \in V_1 \cap V_2$ then p and q receive the same set of messages in V_1.

For example, these properties ensure "atomic failure". If process q in view V sends messages $M = \{m_1, \ldots, m_i\}$ and subsequentally fails then the rest of the processes in V receive the same subset of M in V (before the view change $V \setminus \{q\}$).

Apart from these properties, in Ensemble, each view has a unique leader known to all view members.

We assume the existence of an authentication service available to all group members. Such a service allows members to authenticate each other as well as create private and authentic messages. We currently use PGP to this effect. In general, any form of authentication method can be used. We assume that members' private keys are never cracked. All members publish their public keys through PGP, this information is managed by PGP [20, 13] in the form of a database. We assume this database is never corrupted and when member p retrieves member q's public key then it is up-to-date and correct.

Each member decides on its own trust policy, if p trusts q then we mark this by $p \xrightarrow{t} q$. Ensemble creates group components according to the symmetric transitive closure of this relationship, marked by $p \overset{t}{\equiv} q$, and named *st-trust*. st-trust is created as follows:

Symmetry: If $p \xrightarrow{t} q$ and $q \xrightarrow{t} p$ then $p \overset{t}{\equiv} q$

Transitivity: If $p \stackrel{t}{\equiv} q$ and $q \stackrel{t}{\equiv} s$ then $p \stackrel{t}{\equiv} s$

If $p \stackrel{t}{\equiv} q$ then Ensemble allows p and q to be in the same component. All members inside a group are trusted to act according to protocols. Members may dynamically change their authorization policies and ask Ensemble to reform their group component accordingly. Ensemble will exclude untrusted members and allow trusted members in.

The adversary has access to all untrusted (potentially dishonest) machines and may corrupt or eavesdrop on any packet traveling through the network. Our goal is to protect messages sent between trusted members of $\mathcal{U}$. We do not provide protection against denial of service or traffic analysis attacks. Rather, we restrict ourselves to the authenticity and secrecy of message content . We work with an existing operating system and assume its security and correctness. An OS vulnerability would cause a breach in Ensemble security.

Throughout this paper we discuss a single group G and its possible components. While describing the Exchange and Rekey protocols we assume that all machines able to authenticate themselves are trusted. Later we refine this model with an application trust policy. Thus, the trust relationship may be dynamic and include several non-intersecting domains of mutually trusting machines. We call these domains *st-domains* since they include machines that all belong to a single equivalence class of st-trust.

2.1 Ensemble

Ensemble is a GCS supporting process groups as above. Apart from supporting reliable fifo-ordered multicast and point-to-point communication it may optionally support many other protocols and communication properties such as: multicast total order, multicast flow control, protocol switching on the fly, several forms of failure detection, and more (see [26] for more details).

Ensemble is a user-level library linked to an application. It is divided into many *layers*, each implementing a simple protocol. Applications may customize the Ensemble library and use the set of layers they require. The set of chosen layers is composed into an Ensemble *stack*. All members in a group must have the same stack to communicate.

Ensemble keeps *view-state* information. This information is replicated at all group members and includes data such as: current protocol stack in use, group member names and addresses, the number of members, the group key, etc.. In order to change any of this information, a new view has to be installed. So, if the group key needs to be changed. the group will be prompted for a view change. During the process the leader will broadcast the new view-state, that includes the new group key, and all members will use the new group key in the upcoming view.

Ensemble divides messages into two classes. There are intra-group or regular messages sent between members of a view. They are usually application-generated messages, though some messages may be generated as part of the Ensemble protocols on behalf of the application. In addition, there are inter-component messages or so-called "gossip" messages. These are messages generated by Ensemble for communication between separate components of an Ensemble group. A gossip message is multicasted to $\mathcal{U}$, to anyone who can hear. Normally, communication is not possible between group components due to network partitions. Gossip messages are used to merge components together and they arrive at their destination when network partitions and link failures are repaired. Receipt of a gossip message when partitioned triggers the merge protocol. Protocols that use gossip messages typically make very few assumptions about them: they may be lost, reordered, or be received multiple times.

The regular and secure Ensemble stacks are depicted in Figure 1. The Top and Bottom layer cap the stack from both sides. The membership layers compute the current set of live and connected machines, the Appl_top layer interfaces with the application and provides reliable send and receive capabilities for point-to-point and multicast messages. The RFifo layers provide reliable per-source fifo messaging. The Exchange and Rekey layers are related to the membership layers since the group key is a part of the view information. The Encrypt layer encrypts all user messages hence it is below the Appl_top layer.

Regular	Secure
Top	
	Exchange
	Rekey
Gmp	
Top_appl	
	Encrypt
RFifo	
Bottom	
Routers	

Table 1: The Ensemble stack. On the left is the default stack that includes an application interface, the membership algorithm and a reliable-fifo module. The secure stack, to the left, includes all the regular layers and also the Exchange, Rekey, and Encrypt layers.

2.2 Policies

The user may specify a security policy for an application. The policy specifies for each address[2] whether it is trusted or not. Each application maintains its own policy, it is up to Ensemble to enforce it and to allow only mutually trusted members into the same subgroup. A policy allows an application to specify the members that it trusts and exclude untrusted members from its subgroup.

2.3 Cryptographic Infrastructure

Our design supports the use of a variety of authentication and encryption mechanisms. Currently, the system uses PGP for authentication. Messages are signed using MD5 [21], and encrypted using RC4 [9]. Because these three functionalities are carried out independently any combination of supported authentication, signature, and encryption systems can be used. A future goal is to allow multiple systems to be supported concurrently. Under such a system, processes would be able to compare the systems they have support for and select any system that both have support for.

[2]An Ensemble address is comprised of a set of identifiers, for example an IP address and a PGP principal name. Generally, an address includes an identifier for each communication medium the endpoint is using {UDP,TCP,MPI,ATM,..}.

3 The authentication router

We first describe the simplest part of the security architecture, the authentication router module. Ensemble routers reside at the bottom of the system and each protocol stack uses a specific route. They are called routers because they determine how messages should be sent to their destination process and when messages are received they determine the protocol stack to deliver the messages to.

In Ensemble, the router is the module responsible for getting messages from member p to some set of members $\{q_1 \ldots q_k\}$. Routers use transport-level protocols such as MPI, UDP, TCP, and IP-multicast to send and receive messages. An Ensemble application may use several stacks, all stacks may share a single router. Hence, routers need to decide through which transport to send a message, and when it is received — which protocol stack to deliver it to.

We call this module the router because, like an IP router, it sends and receives messages deciding through which "interfaces" to send messages and where to "forward" them.

We have modified the normal router to create a signed router that is used when the application requests a secure protocol stack. A signed router adds a keyed-MD5 signature to each sent message and verifies the signature of each incoming message before handing it to the protocol stack.

Ensemble signs all outgoing messages using the group key. Regular messages may be verified by other group members since they all share the group key. Gossip messages may be problematic since, initially, different components do not share the same group key. Hence, they are protected using public key cryptography.

When message m arrives at the signed router, belonging to group component A, the router attempts to verify m using the group key. There may be several cases:

m is a regular message:

1. Correctly signed: Pass up the stack. Message m was sent by a group member in A.
2. Incorrectly signed: Drop. Message m may come from a different group component that shares no key with A. It may also be a message sent by the adversary (that does not know the key).

m is a gossip message:

1. Correctly signed: Pass up the stack. Message m is of gossip type, it was sent by a member of a different component that shares the same group key.
2. Incorrectly signed: Mark as *insecure* and pass up the stack. This is a message from a different component B that is signed with B's group key. We ignore the keyed-MD5 signature, since we cannot verify it. Possibly, the inner message is signed using public key cryptography. The Exchange layer will attempt to verify it, if successful, it will process m's contents. Exchange is the only layer that examines such messages, other protocol layers that use gossip messages ignore *insecure* gossip messages.

The signed router uses the HMAC [12] standard to compute message signatures. A cryptographically secure one-way hash function, MD5, is used to hash the message content. MD5 is keyed with the current group key such that the adversary will not be able to forge messages. The router at the sender calculates the keyed hash of M — $H(M)$. Then it sends $H(M)$ concatenated to the clear-text message M. On receipt, $H(M)$ is recalculated from M with the receiver's key and compared with the received signature. If there is a match — the message has been verified.

To summarize, the authentication router attempts to authenticate all messages. Regular unauthenticated messages are dropped, gossip unauthenticated messages are still delivered but marked *insecure*.

4 Exchange layer

In the event of a network failure, a process group may become partitioned into several disjoint components, communication among which is impossible. Ensemble automatically elects a leader for each group component. Later, such a partitioned group may need to merge if communication is restored. Ensemble detects the former situation as the failures of one or more group members (the system does not distinguish communication failures to operational processes from process crashes). The system uses gossip messages to discover opportunities to merge a group.

More specifically, it is the responsibility of the Heal protocol to discover partitioned group components. It is active at each group component leader. Each leader gossips an *IamAlive* message periodically that includes its name and address. When a leader hears a remote leader from the same group, it initiates the merge sequence[3].

Group components cannot communicate with each other unless they possess the same key: only insecure gossip messages are allowed to pass through by the router. The Exchange layer uses these messages to create secure agreement on a mutual group key. The idea is that one of the components securely switches its key to that used by the other component. The Heal layer will activate the merge sequence after both components have the same key. The Exchange layer is active at each component leader acting as a filter of gossip messages. All outbound/inbound gossip messages pass through it. The layer functions via the creation and recognition of two types of gossip message headers. These are, for process p whose principal name[4] is R_p, and whose view key is key_p:

Id: Contains R_p and a nonce[5]. This header is cheap to create.

Ticket: Contains data to be sent securely to some process q. This header is created by encrypting the data with q's public key and signing it with p's private key. Only q will be able to verify that the message came from p and retrieve the protected data. This header is expensive to generate, and is usually long (currently about 1/2KBytes).

The following event handlers are applied to gossip messages by process q:

- Onto each gossip message, add an Id header.
- Upon receiving an Id$(R_p, nonce_p)$, if it is *insecure*, p is trusted, and $R_q < R_p$[6], then create a Ticket to p and gossip it. The ticket data contains key_q and $nonce_p$ to prove message freshness.

[3]This is outside the scope the scope of this paper, for more information see [25]
[4]This is the name, by which the user is known to the authentication service.
[5]A one time random string used to prove message freshness.
[6]Any type of comparison function may be used here.

- Upon receiving a Ticket from p intended for q, where p is trusted, authenticate it and check the freshness of the nonce. Decrypt it to get key_p. If $key_p == key_q$, then ignore it (we have the same key), otherwise $new\_key := key_p$. Prompt the component to go through a view change, with new_key as the group key. The group key is part of the view-state, when the view change is complete new_key will be installed at all the group's routers.

When a group component leader q receives a gossip message from a remote component leader p, it checks whether it has a lower id. If so it securely sends to p its key key_q. Remote leader p authenticates q, decrypts key_q and switches its component key to key_q . From now on, all gossip messages (correctly signed by key_q) will go through and the components will merge using the membership mechanism.

When a process fails within a component, the (new) leader initiates a view change. The group key is not switched: since we assume honesty of failed members, they will not use knowledge of the current key in a malicious manner.

4.1 Example execution

Figure 1 shows an example execution of the Exchange protocol. Initially, two components, A and B, are executing. They begin by using different keys, key_A and key_B, so initially only gossip messages marked *insecure* are delivered to the coordinators. No other communication occurs between the components. The authentication sequence only involves the coordinators, so in the following steps we refer to the coordinators by the name of the component they are in.

1. Both coordinators, A and B, regularly broadcast gossip messages announcing their presence in the system. The gossip messages contain both Id headers from the Exchange protocol and $Heal$ headers from the Heal protocol (the Heal protocol heals group partitions). Only the Exchange layer will examine gossip messages marked as *insecure*.

2. Coordinator A receives an $Id(B, nonce_B)$ gossip message from B marked as *insecure*. A sends a $Ticket(key_A, nonce_B)$ gossip message to B.

3. B receives a $Ticket(key_A, nonce_B)$ gossip message from A, authenticates and verifies the freshness of the nonce. If the check is fine B also gets key_A. B then prompts its group to proceed through an empty view change[7]. When installing the new view, B installs key_A in the group. Now both components are using the same key.

4. After B has installed key_A, when A and B broadcast gossip messages, they are accepted by the receiving coordinator and not marked as *insecure*. Now the Heal layer will examine the gossip messages, and this results in the two components merging into a single component using key_A.

We use nonces in this protocol so as not to rely on local clocks being in synchrony. In order to use local time as a nonce, one needs to use a secure time service which is not currently a standard Internet service.

[7]We say that the view change is empty because there is no group membership changes. Only the key is switched.

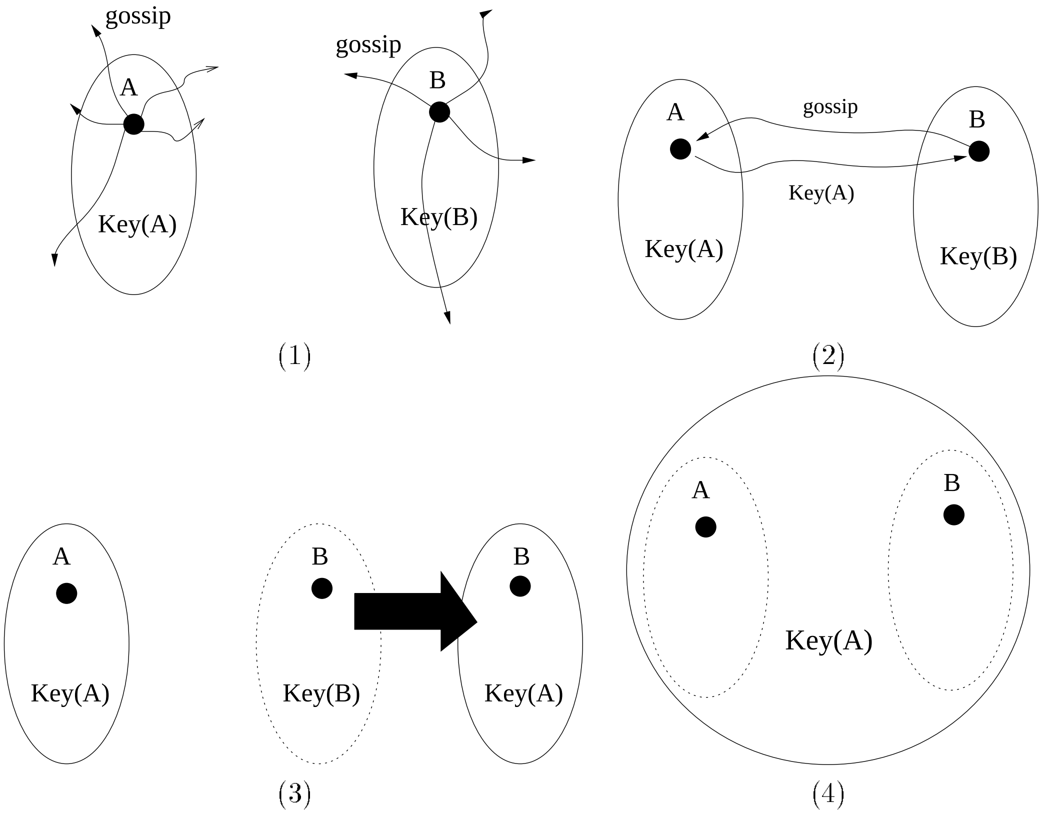

Figure 1: An example execution of Exchange protocol. (1) Leader A and B send out gossip messages. (2) Leader A hears B's gossip message and sends its key to B. (3) B's component switches to $Key(A)$. (4) The components merge since they both use the same key.

5 Validity

Here we informally prove that the Exchange layer achieves secure key agreement. We show that the protocol has the following two properties:

Safety: Only st-trusted members learn the group key.

Progress: Assuming the network remains stable for a sufficiently long period then all members will eventually agree on the same group key.

We begin with a short discussion of the properties of the st-trust relation.

5.1 St-trust

st-trust is an equivalence relation separating the universe of processes into disjoint sets which we call *st-domains* . For the purpose of this discussion we assume the network is stable and no processes crash. In such conditions G's components merge and form *subgroups* according to the trust relationship. We show here that subgroups are set-wise contained in st-domains. That is, if $p \stackrel{t}{\not\equiv} q$ then p and q will never be in the same subgroup.

If $p \stackrel{t}{\equiv} q$ then there is a path $p, m_1, m_2, \ldots, m_k, q$ between p and q such that for each link x, y, $x \stackrel{t}{\rightarrow} y$ and $y \stackrel{t}{\rightarrow} x$. We call such a path an *st-path.*

Examine the possible set of members in any view V_i of which member p is a member. $V_1 = \{p\}$, and the following views $V_2, V_3, \ldots$ are those that p is a member of subsequentally. We inductively prove the for all i, that $q \notin V_i$. For V_1 this is trivial. Inductively assume that $q \notin V_n$. We show that $q \notin V_{n+1}$:

1. Either V_{i+1} is created by a partition or a merge.
2. In the partition case $V_{i+1} \subseteq V_i$ hence $q \notin V_{i+1}$.
3. In the merge case there is a component W such that $V_{i+1} = W \cup V_i$. Denote the leader of V_i by v and the leader of W w. Components W and V_i merge, hence v and w trust each other. There is an st-path from p to each member in V_i. In particular, there is an st-path from p to v. Extend this path to w and from there it is possible to reach all members of W. Hence, there are st-paths from p to all members of V_i and W. We deduce that q cannot be in V_{i+1}.

This shows that subgroups are subsets of st-domains. Are subgroups real subsets of st-domains, or are they equal to them?

If a set of members S all trust each other then S is an st-domain. Clearly, any subgroup containing a single member of S will include the rest. Hence, in this case we have a subgroup equal to an st-domain.

As an example to the other side of the coin assume:

- $U = \{p, q, s\}$
- $p \stackrel{t}{\equiv} q \stackrel{t}{\equiv} s$, but p and s do not trust each other.
- $\{p, q, s\}$ is the st-domain
- U's subgroups are $\{p\}$ and $\{s, q\}$
- The leaders are p and s respectively

The two subgroups will not merge since p and s do not trust each other. Hence, subgroups may be real subsets of st-domains.

5.2 Proof

For the purpose of this proof we assume application authorization policies are static. Group G may be split into several st-domains, we examine a single st-domain H.

Define a *safe* group key to be one known only to members of H. We inductively prove that group keys are safe. A key may serve as a group key through several views. We denote, for key k the views it goes through as $V_{k_1}, \ldots, V_{k_n}$. View $n+1$ is derived from view n either via a merge or a partition. We first show that Exchange is safe at all of its stages. The protocol has three stages any of which may fail:

- An IamAlive($R_p, nonce_p$) message may not reach its destination.
- A ticket($R_q, nonce_p$) may not reach its destination, or reach an untrusted member.
- The view installation phase at p may fail.

We must show that none of these occurrences breaches safety.

Proof:

- If an IamAlive($R_p, nonce_p$) message does not reach other component leaders then p's component will not merge. While this may be a progress problem, it does not breach safety.
- A Ticket($key_q, nonce_p$) may not reach its destination, or reach an untrusted member. If the ticket gets lost then no information is revealed. The ticket can be opened by p alone, thus its capture by a dishonest member does not breach safety.
- If the view installation phase at p fails then only some of p's component members learn key_q. All of p's component is st-trusted by q's component then only st-trusted members learn key_q.

We now show safety. Induction is performed on all group keys at once, according to the number of views they pass.

Base case: Examine any group key k in its first view, V_{k_1}. The only way to create a new key is to start a new Ensemble process p. Process p creates k as its group key, and only p knows it. Thus, k is secure.

Induction assumption: All group keys are safe up to the n-th view.

Induction step: Examine a specific group key k in view $V_{k_{n+1}}$. $V_{k_{n+1}}$ is created from V_{k_n} either by a merge or a partition. If it is a partition then since previously trusted members are still trusted, and since k was not passed to new members during the partition then it is still secure. In the merge case, assuming components V_{k_n} and $V_{k'_m}$ merge, and $m \leq n$ then:

- The exchange protocol reveals k to $V_{k'_m}$'s leader v(only).
- v uses k', which is secure by induction, to disseminate k to to $V_{k'_m}$'s members.

Therefore k is revealed to all the members of $V_{k'_m}$. An Exchange can occur between the leaders of V_{k_n} and $V_{k'_m}$ only if they trust each other. Hence, the two components belong to the same st-domain.

To show the protocol makes progress assume that no network faults occur for a sufficiently long period. Assume further that in st-domain H all members are mutually trusted. If H is split into two components A and B then eventually A's IamAlive messages will reach B, causing B to send A its key and the components to merge. Hence, eventually, all of H's components will merge.

6 The Rekey protocol

Occasionally we need to switch the group key. There may be several reasons for doing this:

Lifetime expiration: Symmetric encryption keys have a bounded lifetime in which they are secure. After such time a dedicated adversary will be able to crack them. Currently, Ensemble uses RC4 [22] as the default encryption mechanism, this is a relatively weak encryption scheme with a 40bit key. Today, there are much stronger algorithms such as DES [9], triple DES [9], and IDEA [28] that employ longer keys (56bits, 112bits, and 128bits respectively). A 128bit key should be safe for hundreds of years even using top of the line machines to try cracking it. Using a weak encryption key allows Ensemble to be exported while maintaining a reasonable level of security. It takes 64 MIPS-years to break an RC4 key, which we believe is more than what a casual eavesdropper is willing to pay. Furthermore, RC4 is faster than stronger encryption algorithms. This allows efficient communication.

Dynamic Trust Policy: Members may dynamically change their trust policy. Thus, old members may not be authorized to listen to current group conversations. This requires the ability to switch the group key, preventing old members from eavesdropping.

OS vulnerability: We work with an existing OS that is not perfect and may be penetrated by a persistent and knowledgeable intruder. The intruder may penetrate old group members and discover the group key. In order not to rely on old members erasing their key, we wish to switch it.

The Rekey protocol provides a way to securely and synchronously switch the communication key used by a group. Unlike the Exchange protocol where we use the old key to disseminate the new key, here the new key is unrelated to the old key and we do not rely on the old key for its dissemination. Thus, possession of the old group key does not allow discovering the new key or eavesdropping on current group conversations. We authenticate members using their public keys, which we assume are never broken.

The Rekey protocol works as follows:

1. The leader chooses a new random key, unrelated to the old group key.
2. The leader signs, encrypts and sends the new key to each member of the group using public key cryptography.
3. A group member, upon receipt of the new key, sends an acknowledgment to the leader.
4. When the leader receives acknowledgments from all group members it starts a view change. The new key will be installed in the router in the upcoming view.

If some members do not acknowledge the receipt of the new key, they may have crashed or partitioned away. A new view excluding the faulty members takes place, and the old key stays unchanged. The user is notified that rekeying has failed and may ask to rekey again. The second invocation is likely to succeed since the faulty members have been removed.

6.1 Authorization polices

When a Rekey is invoked the leader checks that all current members are trusted. If not, it removes the untrusted members from the view, installing a new (smaller) view. The Exchange protocol will prevent untrusted members from rejoining the group. Rekey requires the correct participation of untrusted members. It cannot exclude a Byzantine member.

The application may dynamically change its security policy. This entails the revocation of the old group key to prevent old untrusted members from eavesdropping and altering current group conversation. Thus, the Rekey protocol needs to be invoked by the application whenever a policy change is performed.

6.2 Optimizing the Protocol

The protocol as described above is fairly slow because public key operations are very costly in terms of CPU time and memory (see Table 2). Let us examine the latency of a rekey operation considering only public key operations in a group of 64 members. Assume a seal/unseal operation takes $0.25sec$. The leader performs 63 seal operations while each member performs another unseal operation prior to acknowledging the key. Thus the latency is: $\sim (63 + 1) * 0.25sec = 16sec$

Our protocol aims to be efficient and scalable. Hence we added the following optimizations:

- We spawn a process to perform the encryption/decryption in the background. This removes the expensive public key operations from the critical path so that the protocol stack can keep running as usual. A similar optimization was used in Horus [24].

- To increase the scalability of the protocol we use a tree structure to disseminate the new key as shown in Figure 2. The leader sends the new key to its children, who in turn pass it down the tree (step (a) and (b)). Each member sends an acknowledgment to its ancestor after it has collected acknowledgments from all its children (step (c) and (d)). When the leader receives acknowledgments from all children it multicasts a *RekeyDone* messages to the group and starts a new view (step (e)).

  Using the tree structure, the latency of a rekey operation can be improved substantially. Suppose a binary tree is used. The cost for each level of the tree, except the first, is: leader performs two seal operations, children perform an unseal operation each. This amounts to 3 public key operations. The latency now becomes: $(\log(64) - 1) * 3 * 0.25sec = 3.75sec$.

The latency using a tree-structure as analyzed above is still high. To improve it, we used *secure channels*. A secure channel between members p and q is created as follows:

1. p generates a random symmetric key k_{pq}, encrypts it with q's public key and sign with p's private key.

2. p sends the sealed key to q.

3. q acknowledges the receipt of k_{pq}.

Henceforth any message sent on the secure channel is encrypted and signed using k_{pq}. A secure channel allows sending private information between two peers. In contrast, a group key allows multicasting private information to the whole group.

We added a Secchan layer to Ensemble that handles a cache of secure channels. Whenever private information needs to be passed from p to q, a secure channel between them is created if one does not already exist. The operation to create a secure channel is expensive: it takes two public key operations for encryption/decryption which is around 1/2 second (see Table 2). On the other hand, the next private message between p and q will be encrypted with symmetric key k_{pq}, a much quicker operation lasting a couple of microseconds.

Using this cache, a typical group rekey operation will run much quicker. Assuming a static view V, the first Rekey invocation will run several seconds since new channels must be set up [8]. The next invocation will run much faster since secure channels have already been set up. **[Begin]** If we examine the 64 member case, and focus on the actual rekeying performed (without the following view installation), then the latency is:

- Ensmeble achieves latency of around 2ms between members of a LAN. The $a$verage latency of a reliable multicast is the same.
- Sending the information down/up the tree costs: 2ms * 2 * (tree depth) $= 5ms * (log_2(64+1)-1) = 4*6 = 24ms$.
- Plus the latency of a reliable multicast — $2ms$

All in all, the latency is less then 30ms, much less then the latency of an all PGP implementation. **[End]**

Channels are refreshed periodically to avoid exposure to cracking. Whenever a rekey operation is invoked we discard any channels that violate the authorization policy.

CPU	P120	PP200	P2/300
Seal operation (seconds)	0.38	0.30	0.22

Table 2: Seal performance on different CPUs, we tested on a Pentium 120Mhz (P120), a PentiumPro 200Mhz (PP200), and a Pentium2 300Mhz (P2/300). We sealed a 32 byte message.

7 The Encrypt Layer

Ensemble optionally supports user message privacy. The Encrypt layer encrypts/decrypts all user messages with the group key. User messages are reliably delivered in fifo (sender) order allowing use of chained encryption[9]. Ensemble messages are not encrypted but signed only. They do not contain any secret user information and their encryption will only degrade performance. Currently we use the group key for both authentication and encryption. Since MD5 keys are 16 bytes long, we use only the first 5 bytes for the RC4 key. To improve performance, upon a view change we create all security related data structures and use them henceforth.

[8] Note that we do not need set up secure channels between each pair of members in the group. Only those pairs corresponding to the edges in the dissemination tree need be set up.

[9] Modern encryption ciphers separate a message into fixed sized blocks. One can encrypt each block separately, or, using *chained encryption*, use early blocks to help encrypt the current block.

8 Performance

The division of work between the Exchange/Rekey layers and the Encrypt/Routers is efficient in computation. During normal run time we use the symmetric key which is fast and uses little memory. On the relatively rare occasion of a merge or a requested rekey we use public keys/authentication tickets. These tickets require more computation and memory, one typically uses 1024bit RSA keys or waits for RPC style calls to an authentication server.

Measurements were taken on PentiumPro 200Mhz machines running the MOSIX operating system [2] connected with 2.5Gbit/sec Myrinet. Current OS and communication stack do not achieve maximal hardware performance. The single exception is the rekey measurement where we added several Pentium 120Mhz machines running the BSDI operating system. These are connected through a hub and through a proxy to the PentiumPro's.

In Figure 3 we depict the latency of an Ensemble stack. The numbers are given for a send/recv operation: a message arrives at the stack and then is handed to the application which sends an immediate response. The latency is measured from message arrival to message departure. The X-axis measures message size in bytes and the Y-axis time in seconds. As we can see, latency is constant for all message sizes with a regular stack. This is because the stack does not process message content at all. Basic latency increases for Authentication and Privacy stacks since they have not been as aggressively optimized as the regular stack. They also initialize encryption and authentication contexts and allocate and add 16bytes of signature space to each message. For such stacks latency also increases as a function of message size since MD5 hashing and RC4 encryption pass over message content. The theoretic processing times for an x-byte message are:

- Authentication: $2 * x/21.45 \sim 0.093x$
- Privacy: $2 * (x/21.45 + x/7.52) \sim 0.358x$

Disregarding the initial costs of encryption and hashing, these linear lines asymptotically approach the measured latencies .

	P120	PP200	P2/300
MD5	9.92	21.45	32.5
RC4	2.03	7.52	12.45

Table 3: MD5 and RC4 performance on different CPUs. Performance is measured by the number of bytes processed in a microsecond.

We also tested achievable throughput with Ensemble, see Table 4. We ran an Ensemble application on two machines, one is chosen as leader and it sends as many 1000byte messages to the other member as possible. The maximal achievable throughput using a regular stack is $3330 Kbyte/sec$. As we add authentication, throughout drops to $2850 Kbyte/sec$. When we add encryption throughput drops to $1600 Kbyte/sec$. The bottleneck is the CPU. The third column in the table shows the amount of CPU time per second used for encryption and verification.

Next, we measured the latency of the rekey operation. We created a group of static size and performed 100 rekey operations in succession. Time was measured from the initiation of a rekey to the installation of the new view. Figure 4 depicts times for groups of size 2 to 10 where each process is on a different machine.

Stack	Kbyte/sec	CPU time/sec
Regular	3330	0.000
Auth	2850	0.132
Auth+Privacy	1600	0.286

Table 4: Ensemble throughput with different stacks. As authentication and encryption are added to the stack — performance drops due to heavy CPU load.

9 Related Work

Ensemble is descended from an earlier system named Horus, itself descended from the Isis system [14]. Early work on group communication security was performed in Horus [18, 24]. Our work extends the Horus security architecture incorporating into Ensemble the insights gained from its implementation. We added support for multiple partitions (not just primary partition), group rekey upon demand, application defined security policies, and plugged an off-the-shelf authentication engine (PGP) into the system.

A group communication system designed for Computer Supported Collaborative Work (CSCW) applications has been built in the university of London [3]. In the context of CSCW, objects and files are typically shared between applications. As such, different applications are allowed to perform different operations on shared objects. To enforce these restrictions the most trusted member of the group is chosen as leader. Any message a member wishes to multicast is forwarded to the leader. The leader filters all such messages: discards the malicious ones, enforces the shared objects security policy, and multicasts all legal messages. This work however is yet in its preliminary stage, at the time of this writing it does not provide for leader failure. Furthermore, this work is oriented towards CSCW application alone.

Rampart [23] is a group communication system built in AT&T that is resistant to Byzantine attacks. Up to a third of the members in a Rampart group may behave in Byzantine manner yet the group would still provide reliable multicast facilities. A system providing similar guarantees has been built in the university of Santa-Barbara in California [16]. Byzantine security is rather costly however, and it is difficult to develop applications resistant to such faults. We chose not to support such a fault model in Ensemble.

Other works in the IP multicast security area include [1, 10, 11]. These works describe the management of a session key for (very) large groups such that the infrastructure required is scalable and efficient. Recent work [7] has dealt with the efficient rekeying of large multicast groups. IP multicast is concerned mainly with one-to-many multicast, where a single application multicasts to many clients whose membership is dynamic and not necessarily known. Ensemble is concerned mainly with many-to-many multicasts where any member may multicast to the group and where membership is known. In IPsec, trusted centralized servers may be used to disseminate group keys; in Ensemble, possessing a completely distributed architecture, this is not possible.

10 Conclusions

We have extended the Horus security architecture to support multiple partitions (not just primary partition), added group rekeying upon demand, added security policies and connected our system to an off-the-shelf authentication engine (PGP). Our software is freely available as part of the Ensemble project. We believe that ours is the first freely available secure group communication system.

11 Acknowledgements

We would like to thank Tal Anker for improvements to the optimized Rekey protocol, Yaron Minsky for helping develop the Exchange protocol and for insightful comments, and Idit Keidar for helpful reviews.

References

[1] A. Ballardie. Scalable multicast key distribution. Technical Report 1949, IETF, May 1996.

[2] A. Barak and O. La'adan. The mosix multicomputer operating system for high performance cluster computing. *Journal of Future Generation Computer Systems*, 13(4-5):361–372, March 1998.

[3] A. Rowley and J. Dollimore. Secure group communication for groupware applications. In *European Research Seminar on Advances in Distributed Systems*, march 1997.

[4] B. Whetten, T. Montgomery, and S. Kaplan. A high perfomance totally ordered multicast protocol. In K. P. Birman, F. Mattern, and A. Schipper, editors, *Theory and Practice in Distributed Systems: International Workshop*, pages 33–57. Springer, 1995. Lecture Notes in Computer Science 938.

[5] K. Birman and R. Van Renesse. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1994.

[6] C. Malloth and A. Schiper. View Synchronous Communication in Large Scale Networks. In *Proc 2nd Open Workshop of the ESPRIT projet BROADCAST (#6360)*, July 1995.

[7] C.K. Wong, M. Gouda, and S.S. Lam. Secure group communication using key graphs. In *ACM SIGGCOM.* ACM, September 1998.

[8] R. Friedman and R. Van Renesse. Strong and Weak Virtual Synchrony in Horus. Technical Report TR95-1537, Cornell University, 1995.

[9] U. Goverment. Data encryption standard. Technical Report 46, National Bureau of Standards, Federal, 1977.

[10] H. Harney and C. Muckenhirn. Group key management protocol architecture. RFC 2094, IETF, 1997.

[11] H. Harney and C. Muckenhirn. Group key management protocol specification. RFC 2093, IETF, 1997.

[12] H. Krawczyk, M. Bellare, and R. Canetti. Hmac: Keyed-hashing for message authentication. RFC 2104, IETF, Febuary 1997.

[13] J. Callas, L. Donnerhacke, M. Bellare, H. Finney, and R. Thayer. *OpenPGP Message Format*, July 1998. Internet Engineering Task Force, An Open Specification for Pretty Good Privacy (openpgp) Working Group.

[14] K. Birman , R. Cooper, T.A. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck, and M. Wood. *The ISIS System Manual.* Dept of Computer Science, Cornell University, Sep 90.

[15] K.P. Birman and T.A. Joseph. Exploiting virtual synchrony in distributed systems. TR TR87-811, Department of Conmputer Science, University of Cornell, 1987.

[16] K.P. Kihlstrom, L.E. Moser, and P.M. Melliar-Smith. The securering protocols for securing group communication. In *Hawaii International Conference on System Sciences*, volume 3(31), pages 317–326, january 1998.

[17] S. Maffeis. ibus – the java intranet software bus. http://www.softwired.ch/ibus.htm.

[18] M.K. Reiter, K.P. Birman, and L. Gong. Integrating security in a group oriented distributed system. TR TR92-1269, Department of Conmputer Science, University of Cornell, February 1992.

[19] O. Babaoglu, R. Davoli, and A. Montresor. Partitionalbe Group Membership: Specification and Algorithms. TR UBLCS97-1, Department of Conmputer Science, University of Bologna, January 1997.

[20] P. Zimmermann. Pretty good privacy. http://www.pgpi.com.

[21] R. Rivest. The md5 message digest algorithm. RFC 1321, SRI Network Information Center, 1992.

[22] R. Thayer and K. Kaukonen. A stream cipher encryption algorithm. Internet draft, IETF, 1997.

[23] M. Reiter. Secure agreement protocols: Reliable and atomic group multicast in rampart. In *ACM Conference on Computer and Communication Security*, number 2, pages 68–80, November 1994.

[24] M. Reiter, K.P., Birman, and R. Renesse. A security architecture for fault-tolerant systems. *ACM Transactions on Computer Systems*, 4(12), November 1994.

[25] O. Rodeh. The Design and Implementation of Lansis/E. Master's thesis, Hebrew University, May 1997.

[26] R.V. Renesse, K. P. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building adaptive systems using ensemble. Technical report, Cornell University, 1997.

[27] R.V. Renesse, K.P. Birman, and S. Maffeis. Horus, a flexible group communication system. *Communications of the ACM*, April 1996.

[28] X. Lai, J.L. Massey, and S. Murphy. Markov ciphers and differential cryptanalysis. In *Advances in Cryptology – EUROCRYPT*, 1991.

[29] X. Leroy. The Objective Caml system release 1.07, 1997. http://pauillac.inria.fr/ocaml.

[30] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A Communication Sub-System for High Availability. In *FTCS conference*, number 22, July 1992.

[31] Y. Amir, L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, and P. Ciarfella. Fast Message Ordering and Membership using a Logical Token-Passing Ring. In *13th International Conference on Distributed Computing Systems (ICDCS)*, pages 551–560, May 1993.

12 Appendix

12.1 Running Rekey and Exchange concurrently

The Rekey and Exchange protocols may be invoked concurrently if a merge occurs after the user requested to rekey the group. The leader on the one hand runs the Rekey protocol and disseminates a new random key k_{rnd}, on the other hand it (w.l.o.g.) receives a new key k_{new} from a remote component. Which key should the leader install?

We solve the problem by observing which protocol terminates first. If the Rekey dissemination finishes first, then k_{rnd} will be the new key. If Exchange terminates first then k_{new} is installed and k_{rnd} is discarded.

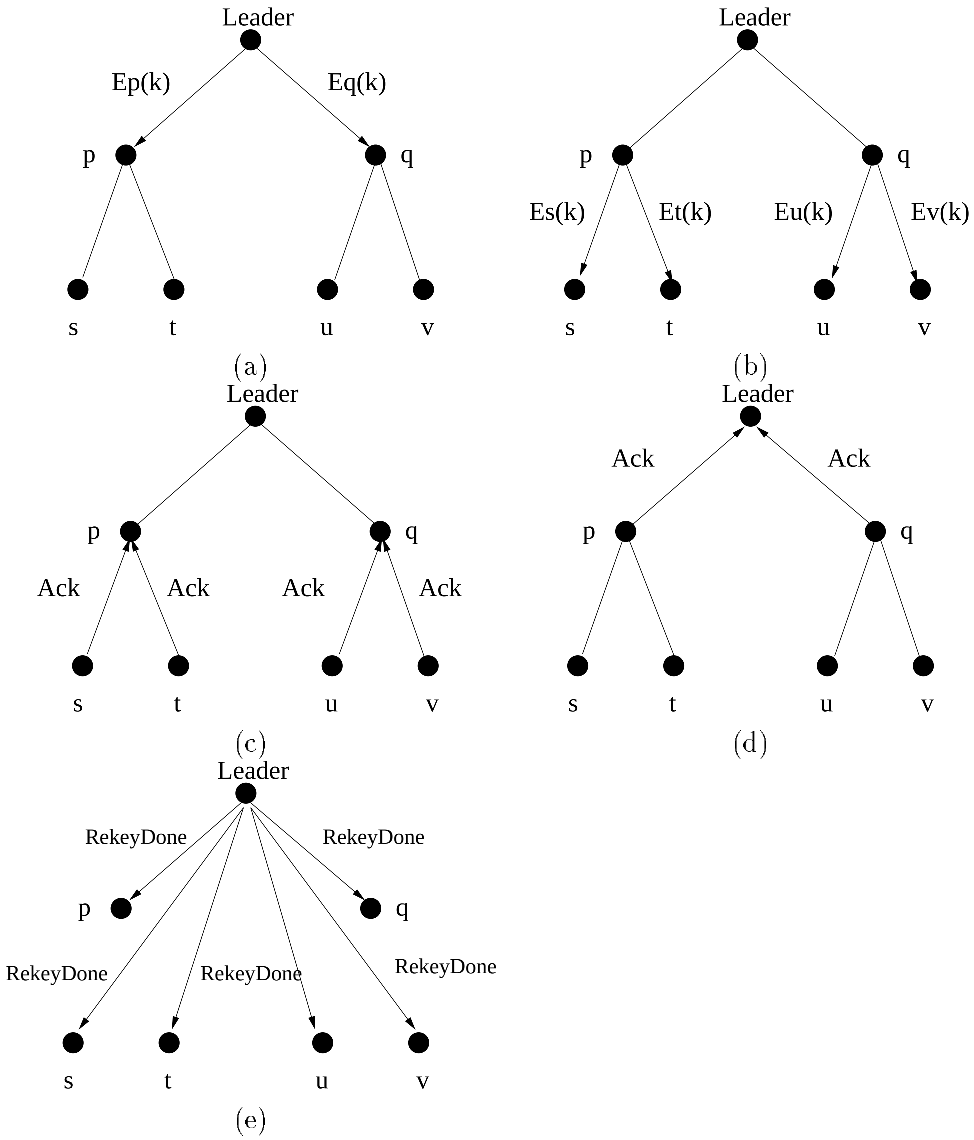

Figure 2: Tree-structured Rekey protocol. (a) Leader sends the new group key, k, down the tree. $E_p(k), E_q(k)$ are sealed electronic envelopes containing k for members p and q, respectively. (b) Upon receipt, p and q in turn pass k down their subtrees. (c) & (d) Acknowledgments climb up the tree. (e) Once the leader receives acknowledgments from all its children, it multicasts a *RekeyDone* message to the group.

.

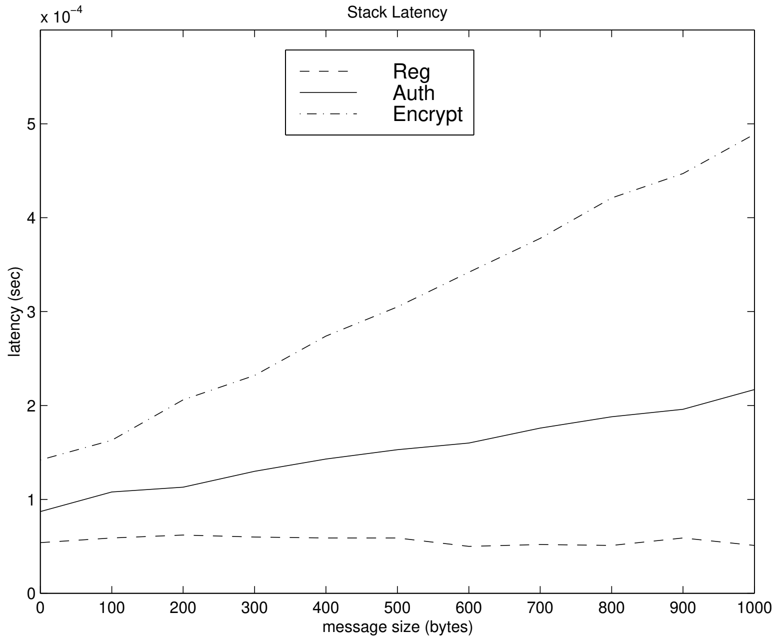

Figure 3: Latency of a send/recv operation using a regular stack, authenticated stack and a private authenticated stack.

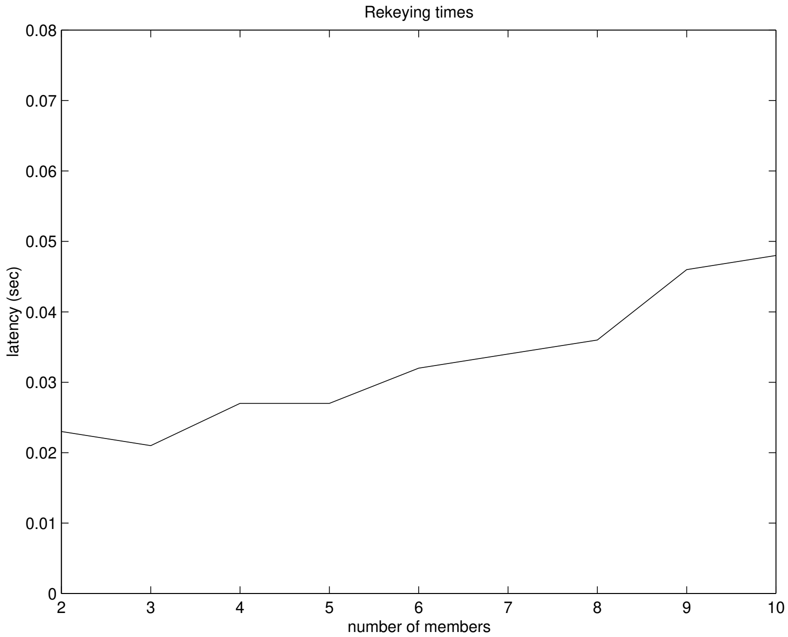

Figure 4: Latency of a rekey operation