ORNL/TM-13470

Computer Science and Mathematics Division

Mathematical Sciences Section

GROUP KEY MANAGEMENT

Tom Dunigan and Cathy Cao

Mathematical Sciences Section
Oak Ridge National Laboratory
P.O. Box 2008, Bldg. 6012
Oak Ridge, TN 37831-6367
thd@ornl.gov cao@cs.utk.edu

Date Published: September 1998

Research was supported by the Office of Scientific Computing of the Office of Energy Research, U.S. Department of Energy.

Prepared by the
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831
managed by
Lockheed Martin Energy Research Corp.
for the
U.S. DEPARTMENT OF ENERGY
under Contract No. DE-AC05-96OR22464

Contents

GROUP KEY MANAGEMENT

Tom Dunigan and Cathy Cao

Abstract

This report describes an architecture and implementation for doing group key management over a data communications network. The architecture describes a protocol for establishing a shared encryption key among an authenticated and authorized collection of network entities. Group access requires one or more authorization certificates. The implementation includes a simple public key and certificate infrastructure. Multicast is used for some of the key management messages. An application programming interface multiplexes key management and user application messages. An implementation using the new IP security protocols is postulated. The architecture is compared with other group key management proposals, and the performance and the limitations of the implementation are described.

1. Introduction

The Internet and private intranets are increasingly being used for business, government, and military communication. Information flowing over these networks often needs to be authenticated and perhaps encrypted to protect against modification or disclosure. Information protection is usually provided by each application, if provided at all. Standards and implementations are being developed that would provide information protection for data flowing between pairs of network applications. However, there is a growing body of group software, where several applications or computers collectively communicate, providing shared access to files, whiteboards, video, and audio. These group applications could protect their communications with a massive collection of pairs of encrypting keys, but many group applications utilize multicast protocols. With multicast, only a single copy of a message is transmitted but it can be received by all members of the multicast group. Clearly a single group key is needed for protecting multicast communication. This report looks at techniques for protecting communication among a group of participants. Specifically we define an architecture that

- specifies group communication policy
- defines group membership requirements
- generates and distributes a group key

The architecture is flexible and extensible, allowing one to specify various algorithms for encryption, signing, and hashing. Strong authentication and authorization is required, but centralized key servers and complex public key infrastructures are avoided. The associated protocol is efficient in terms of memory, bandwidth, and number of messages. Key management and group access are managed by members of the group. The architecture provides perfect forward secrecy or optional key escrow.

An implementation of this architecture is also described in this report. The implementation is self-contained, providing the necessary certifying infrastructure. The implementation is portable and interoperable over various computer architectures and UNIX operating systems. The implementation includes a simple application programming interface (API). The implementation is simple to deploy and economical to operate.

The organization of this report is as follows. The next section reviews other research related to key management. Section Three describes our group key management architecture. Section Four describes our current implementation. Section Five compares our architecture and implementation with other group key management schemes and describes how the architecture might be implemented under the developing IP security protocols. Section Six summarizes our findings and suggests further work. This report is is a synopsis of the work reported in [6].

Our research makes only a few assumptions. Our group key management architecture assumes the existence of a public key infrastructure in order to provide strong authentication. As part of the implementation, we have provided a simple public key infrastructure and mechanisms to create the necessary certificates for authenticating group members. Though our long-term goal is to provide scalable group key management, the funding agency was specifically interested in a workable solution for only dozens of members. Thus the current architecture manages only a limited number of members. The architecture does not assume or require the existence of the developing IP security protocols.

2. Background and related work

There are several critical technologies that are required for our group key management architecture. In this chapter we summarize those technologies and review related works on key management.

2.1. Public Keys

The cryptographic tools that are needed to provide secure communication (integrity, privacy, and authenticity) are hash functions, encryption functions, random number generators, and public key functions. Hash, or one-way, functions like MD5 [24] or SHA [25] provide message integrity. Encryption functions (DES, IDEA, Blowfish, RC5 [25]) use a shared secret to encrypt and decrypt messages, providing message privacy. The strength of an encryption algorithm is usually in proportion to the key length. Strong (unpredictable) random numbers are needed for key generation.

Shared-secret cryptographic systems are difficult to manage. One needs a secure channel to establish the shared secret between two parties. For n parties, each party needs to establish shared secrets (keys) with each other, or there needs to be a central, trusted key distribution center (KDC) that can establish a key on behalf of one or more parties, for example, Kerberos [19]. One of our design goals, however, was to avoid a such a central point of failure and potential performance bottleneck. Finally, shared-secret systems do not provide strong authenticity to a third party – if two people share a key, a third party cannot be certain which of the two was the originator of an encrypted message.

Public key algorithms like RSA and DSA [25] simplify key management and provide message authenticity. Each party has a public/private key pair. The public and private keys are mathematically related. The private key must be kept secret, but the public key can be published in a directory or otherwise made publicly available. Alice can encrypt a message for Bob using Bob's public key. Bob can decrypt the message using his private key. Alice can encrypt (or sign) a message with her private key, and others can use her public key to verify that Alice, and only Alice, signed the message. Encryption/decryption (or sign/verify) involve many multiplications of very large (1000-bit) numbers and can be quite slow – a thousand times slower than secret-key encryption algorithms [25]. As a result, when Alice digitally signs a document, she usually signs (encrypts) a hash of the document.

Although distributing public keys is easier than distributing shared secrets, one still needs a mechanism to assure that a public key really belongs to Alice or to Bob. Public keys are often signed by a certifying authority or by a friend, and these signed keys (or certificates) are distributed informally or by a directory service (e.g., X.500 or even the Internet domain name service). If someone manages to discover your private key, then you need a way to invalidate the public key. So one often needs a public key infrastructure to name, certify, distribute, and revoke keys. Most of the key management protocols, including ours, assume that a public key infrastructure is in place. However, our implementation provides its own simple infrastructure similar to that proposed by Ellis [7].

2.2. Diffie-Hellman

Given a public key system, one could use it to encrypt and decrypt messages between two entities, but the computations are slow. Also if the two parties already share a secret, that secret could be used to encrypt messages. However, to protect past encrypted traffic, neither of these approaches is satisfactory. It is desirable to have a new traffic encrypting key for each session. Using a new traffic key not derived from previous keys provides "perfect forward secrecy" [14]. That is, learning the key for one session provides no information on the key for past or future sessions.

An algorithm that provides perfect forward secrecy was described by Diffie and Hellman [8]. Like public key schemes, it is based on the exponentiation of large (1000-bit) numbers. The Diffie-Hellman algorithm is used by many key management protocols and applications. It permits two entities, Alice and Bob, to establish a shared secret. A prime, p, and generator, g, are publicly known. Alice generates a secret random number a and calculates $\alpha = g^a \bmod p$. Bob generates a secret random number b and calculates $\beta = g^b \bmod p$. They exchange α and β, then their shared secret is $\alpha^b = \beta^a = g^{ab} \bmod p$. An eavesdropper hearing the exchange cannot calculate the shared secret without knowing one of the secrets a or b, and cannot easily calculate a or b from the exchange due to the intractability of doing discrete logarithms. The algorithm is vulnerable to a person-in-the-middle attack, but that can be avoided if Alice and Bob sign their exchanges with their public keys.

Diffie-Hellman can be extended to establish a group key by having the group members do exchanges with a designated group controller or by arranging the members into a ring. Ingemarsson [17] describes a conference key distribution system using Diffie-Hellman with the members arranged in a ring. Paoli [23] also describes a ring-based system for conference key distribution. The conference key models often assume a single packet source (e.g., the conference speaker or a pay-tv system) and that the packet source shares a different secret key with each source. Just [18] and Burmester [5] also describe extensions to Diffie-Hellman to support multi-party key establishment.

2.3. IPv6 and secure multicast

Substantial progress has been made in defining and implementing the security options associated with the next version of IP [1]. Security extensions are defined to provide both authenticity and encryption to the current generation IP (IPv4) and to the new version (IPv6). The effort has been directed toward pair-wise communication security and not group or multicast security. Two hosts establish a security association (SA) that describes the algorithms and keys to be used in secure communication. The SA is referenced by destination address in conjunction with a Security Parameter Index (SPI). A number of key management protocols have been proposed, including ISAKMP [9], SKIP [2], and Photuris [28]. These protocols negotiate the SA parameters (algorithms for encryption, hashing, signing, lifetimes, etc.) and establish the secure channel, assisted by a Diffie-Hellman exchange.

We have done some experimenting with Naval Research Lab's implementation of IPv6 along with Cisco's ISAKMP implementation. ISAKMP is in turn based on the Oakley key management protocol [14]. The privileged ISAKMP daemon does SA/key management through the PF_KEY interface [21]. When machine policy or the application requests an SA, the kernel notifies the ISAKMP daemon which then negotiates with the destination daemon, and then both set up the SA structures in the kernel. The key exchange involves some sort of authentication. In the case of the ISAKMP implementation (September, 1996), DSA was used for authentication.

The IP security architecture [1] notes that association management for multicast applications is not trivial. The destination address of a security association can be a multicast address, but some third party or member of the multicast group will need to establish the SA/SPI and somehow communicate that to the rest of the group. Then each member needs to establish the SA within its host's kernel. Sender authentication is also problematic and does not scale well.

Gong [13] describes the basic elements for trusted multicast (membership policies, joining/leaving, encrypted/authenticated messaging). He also describes his Enclave system[12] which provides secure group collaboration over the Internet. Ballardie and Crowcroft [4] discuss the security threats to IP multicast and the need for group access control.

2.4. Secure PVM

Secure PVM [10] was our first effort with group key management. PVM [11] provides a programming library and unprivileged daemons that allow a user to parallelize his application over a collection of networked hosts. We added a group key management protocol to the PVM protocol to establish a shared group key among the networked hosts so they could encrypt PVM messages. Diffie-Hellman was used to establish a shared key between each slave PVM daemon and the master daemon. The master daemon established a group key and distributed it to each slave daemon under their respective Diffie-Hellman keys. There was no authentication of the Diffie-Hellman exchange, but this was considered to be an acceptable risk. The master PVM deamon knows who the group members are going to be – a weak form of an access control list. Other systems with embedded group key management are Enclave [13], ICKDS [20], and RHODOS [23].

2.5. Tree-based key distribution

Ballardie [3] describes a scalable multicast key distribution protocol (SMKD) that utilizes the multicast routers of the Core Based Tree (CBT) multicast protocol [3]. SMKD assumes that the routers and group members have access to a public key infrastructure. In SMKD, a group initiator provides a primary core router with a digitally signed access control list for the group. The list enumerates those hosts and routers that may participate in the secure group communication. The primary core establishes the security association parameters, SPI, group session key (GTEK), and key encrypting key (GKEK) for the group. This group information (ACL, keys, SPI) is distributed by the primary core to secondary cores as join requests are processed. The group information is signed by the primary core router and encrypted with the public key of the secondary router. The group information can be propagated down the routing tree in this fashion. Thus any number of end node routers can process group-join requests from hosts, providing a scalable key distribution. SMKD requires some modifications to the IGMP [1] protocol and, of course, assumes that CBT is deployed (which it is not).

The protocol consists of a JOIN-REQUEST and JOIN-ACK message. A host wishing to join a multicast group uses IGMP to transport a signed token to its nearest CBT router. If the router does not have the group information yet, the request is forwarded up the CBT under the signature of each router. If the routers in the path are authorized, the group information is passed from the primary core back down the tree. The end router then can verify the requesting host is on the ACL and send the group information to the host encrypted with the host's public key.

Recently, Wallner, Harder and Agee [27] proposed a scalable key management infrastructure. The group controller is the root of a key distribution tree where the intermediate nodes are assigned encryption keys. Each group member establishes a pair-wise key with the controller (e.g., using Diffie-Hellman), and each member acquires the keys of the nodes above it in the tree. The memory requirements for the intermediate keys and the bandwidth requirements (number of messages) are analyzed.

2.6. GKMP

Harney and Muckenhirn describe the architecture [15] and specification [16] for a Group Key Management Protocol (GKMP). In the fall of 1996, they also made available a demonstration implementation. Our work adheres closely to the GKMP architecture and its goals, but deviates from its protocol specification. The GKMP architecture assigns group control to one of the group members. A group authority defines and signs a group token that lists the group members (an access control list) and the military security clearance (level and categories) of the group. A public key infrastructure is assumed, but the group authority signs the members' public keys and their clearance. The group controller (or controllers) enforces the group-join policy and generates a group key encrypting key (GKEK) and a group session or traffic key (GTEK). The architecture describes key lifetimes and rekeying requirements as well as compromise recovery. Certificate revocation lists (CRL) are proposed as a means of identifying compromised

[1] IGMP is a management protocol for IP multicast.

group members.

The GKMP specification describes the various messages, state diagrams, and protocol used for group key management, though not the specific algorithms nor detailed message layout. The messages include Diffie-Hellman exchange, group join, download keys, member permission, group token, multicast rekey, and member delete. All group key management messages are signed by the sender and verified by the receiver. Each member performs a Diffie-Hellman exchange with the group controller and an encrypted channel is established between the controller and each member. The Diffie-Hellman exchange provides keying material to the group controller for generating the GKEK and GTEK. GKMP requires seven messages for a new member to join the group (join request (1), Diffie-Hellman exchange (2), group token (1), member permission (1), group keys (1), ack (1)). The group key message is encrypted with the Diffie-Hellman key, and rekey messages are encrypted with the GKEK. Rekey requires another Diffie-Hellman exchange between the controller and first member.

Harney's GKMP demonstration application is a multi-threaded C application (30,000 lines of code/comments) for Sun OS. Key management demonstration is controlled from a menu-driven command program *demsetup* which communicates with *dem-drv* running on one or more member hosts. Member roles (group authority, group controller, group member) are assigned by *demsetup* and then *demsetup* can be instructed to initiate key/permission certification, group creation, group join, and rekey (not multicast). The public key information (DSA) is generated as part of the demonstration. UDP is used for message transport, DSA for message and certificate signatures, SHA for hashing, and DES for encryption.

3. Group Key Management Architecture

The critical components of a group key management architecture are

- a means of identifying group members (authentication)
- a means of defining and controlling access to the group (authorization)
- a protocol to join the group and to distribute and refresh encryption keys
- cryptographic software for secure key-management communication

In addition, the architecture should not require any central authority or key distribution center. The architecture should be self-contained and portable, not requiring any special hardware or special system-level services or privileges. The architecture should provide an application interface that maintains the key management environment while managing the application messages in accordance with the group security requirements. Finally, the architecture should be efficient in its use of network bandwidth.

Our architecture uses public/private keys to verify member identity. A group access policy defines what is required to join the group, and various access certificates are used to control access and enforce the membership policy. A simple protocol using signed messages and multicast is used to implement the key management communication. There are messages to join, rekey, update the key, rejoin, and locate the group. A suite of cryptographic functions provides encryption, hashing, random number generators, and digital signatures.

Any network host can create and manage a group. The entity or host defining a group is designated the **group authority**. The group authority specifies the group name, or group id, the security policy and cryptographic algorithms required, and communication requirements (multicast address, port, time-to-live and management port). This information is conveyed in a group information token (Table 3.1) that is signed by the group authority. The token includes a validity period, and the group authority need not be active when the group is actually convened.

group id
algorithms (cipher,hash,sig)
key length
policy
security clearance
validity period
mcast addr,port,ttl
mgt port
group controller
SPI
access control list

Table 3.1: *Group information.*

The group information token includes the id (network address or name) of the **group controller**. This can be statically assigned by the group authority, or if the group authority is active, a group discovery protocol is supported. With group discovery, the first group member contacting the group authority will be designated as the group controller.

Using the information in the group information token, a member joins the group by communicating with the group controller. The group controller enforces the group policy, validating a member's id, signature, and access certificates. The group controller manages key lifetimes and rekeying as well as compromise recovery.

In the following sections, we will look in more detail at the key management message protocol and the access control policy.

3.1. Message protocol

Group keys are managed by means of a simple message protocol between group controller and group member. Each message consists of a header (see Table 3.2) followed by one or more payloads. A message is transported over a protocol such as UDP or TCP. For UDP, the group member is responsible for message reliability (timeout and retransmission). Each key management message is signed by the sender, assuring authenticity and message integrity. The message header indicates the message type and length. The header is not encrypted though some payloads are encrypted. The message header indicates the keyid of the key used for encryption, and the policy field indicates the encryption algorithm. The header also contains a nonce, or cookie, to discourage message replay, and contains the type of the following payload. Message types include group join, group rekey, group rejoin. All lengths or other integer values are transported in network-byte order. Variable length ASCII strings are transported with a trailing NULL byte (i.e., a C string).

type
policy
keyid
nonce
version
next payload
length

Table 3.2: *Key management message header.*

Each payload in the message has its own header (Table 3.3). The payload header indicates the length of the payload and the type of the next payload, or zero if this is the last payload of the message. Payloads include the group information token, Diffie-Hellman values, the various certificates required for joining, and a signature payload.

next payload
length

Table 3.3: *Payload header.*

A member wishing to join a group first obtains the group information either from a pre-delivered certificate or by sending a group discovery message to the group authority. For group discovery, the first member is designated as the group controller and the group controller's name becomes part of the group information token. Subsequent members contact the group controller with a group-join message (Table 3.4). The group-join message is signed by the sender and includes the sender's Diffie-Hellman value and any certificates required by the group policy.

message header
D-H value
authorization certificates
signature

Table 3.4: *Group-join message.*

The group controller processes the group-join request by validating the signature on the requester's message and confirming the required certificates are valid and meet the group policy. If the requester is authorized, the controller generates his Diffie-Hellman value and calculates the Diffie-Hellman shared secret key. The controller also generates the group keys (GKEK and GTEK) and sends back a group-join reply (Table 3.5) signed with the controller's key. The reply contains the controller's Diffie-Hellman value, and the group keys encrypted with the Diffie-Hellman shared secret. The member receives the reply, verifies the signature, calculates the Diffie-Hellman shared secret key and uses it to decrypt the group key block.

message header
gc D-H value
encrypted key block
signature

Table 3.5: *Group-join reply message.*

The group controller may also receive a message from the group authority indicating the need for a group rejoin because a member has been compromised. The group controller also monitors key lifetimes and multicasts a group rekey message which contains a new GTEK encrypted under the GKEK key. If multicast is not supported, the group controller saves each member's network address and simulates multicast for rekey or rejoin by sending a unicast message to each member. (Secure PVM and the GKMP demonstration implementation use only unicast for key management.) If members notice their keys are out of date (perhaps, having missed a rekey message), they can send a key update request to the group controller asking for the current group key.

The simplest mode of operation for an application is to join the group and then get the GTEK and then disassociate itself from the key management process. If however, the application wishes to remain under the group key management protocol, getting rekey or rejoin messages, then the application must conform to the API and message structure provided by our architecture. Table 3.6 illustrates the application header that is built and recognized by our API. The application header has the same first few fields as the key management message and has the length of the original application message. (The transported message may have increased in size to account for padding for encryption and for a signature.) The header also indicates encoding convention used and the keyid of the encrypting key.

type
flags (enc,sig,hmac)
keyid
length

Table 3.6: *Application message header.*

3.2. Group Authorization

The policy field in the group information token specifies the type of authorization required to join a group. The join request is signed by the requesting member so the group controller can authenticate the request. At a minimum, the group controller must have a public key certificate for the requester signed by the group authority. (Our reference implementation provides certificate signing by the group authority.) In addition the policy may require that the requester be on the group access control list, or have a ticket (invitation), or have a clearance certificate. The policy can specify that some or all of these certificates are required to join a group.

Our certificates are much like SPKI certificates [7] and consist of certificate type, algorithms used for signature, a validity period (start time, expiration time), subject id, authorization field (what the subject is permitted to do), and the signer's (group authority) digital signature over the certificate (Table 3.7). For a key certificate, the authorization field is the public key of the subject. For a ticket certificate, the authorization field is just the group id. For a clearance certificate, the authorization field contains the requester's clearance level and categories. Notice that these certificates are not like a capability (i.e., they are not transferable) since they contain the subject id and must be transported in a message signed by the subject to be valid. The group authority can choose expiration times short enough to preclude the need for certificate revocation procedures. The lifetime of these certificates is assumed to be short, for example, hours or days. The short lifetime permits us to avoid the difficulties in maintaining certificate revocation lists. If a certificate or private key has been compromised, the group authority or group controller can multicast a group-rejoin request to the group, effectively disbanding the current group. The group authority can then issue new certificates or issue a new group token with stronger access requirements (e.g., a new ACL or now requiring a ticket).

type
algorithms (hash,sig)
validity
subject id
authorization
signature

Table 3.7: *Authorization certificate.*

The group information may contain a military type security clearance. This consists of a hierarchical clearance level (top secret, secret, classified, unclassified) and one or more categories. If a clearance is specified by the group policy, then the requester must provide a clearance certificate (signed by the group authority). A member's clearance certificate specifies his clearance level and the categories for which he is cleared for that level. To be admitted to the group, the requester's clearance must "dominate" the group clearance.

Though not directly a part of group key management, the policy field also specifies how application messages should be encapsulated. Application messages may be integrity checked using a keyed-hash [22] or hashed and sender-authenticated with the sender's digital signature and/or optionally encrypted with the group cipher and key (GTEK). If the policy specifies just encryption, the encrypted payload includes a CRC-32 checksum for message integrity.

3.3. Key escrow

The group policy may specify that key escrowing is required. This results in the group controller building an escrow payload and appending it to all group key distribution and key update messages. The escrow payload consists of the key id, GTEK, and GKEK signed by the group controller and encrypted with the public key of the escrowing agent. Like the US Government Clipper technology [25], this permits a sniffer to collect the group traffic and then later decrypt it if one has access to the private key of the escrowing agent. Unlike the hardware Clipper technology, the receiver does not ignore data with faulty or missing escrow payloads. The receiver, in fact, ignores the escrow payload. The escrow payload obviously defeats perfect forward secrecy.

4. Implementation

To refine and validate our architecture, we implemented a group key management library under UNIX. The code is written in C (about 3,000 lines) and utilizes software from *ssh* [26], RSA, and GNU's multiprecision arithmetic package. Appendix A contains additional implementation details and a multicast chat program to demonstrate the simplicity of the key management calls.

4.1. Certificate infrastructure

To provide a fully operational software package, we implemented software to generate and sign the various certificates needed for group access control and to serve as the group authority. We used *ssh-keygen* to generate our RSA public/private key pairs and then used our certifying software to create public key certificates signed by the group authority. Note, our public/private keys are not the ones used by *ssh* and our key management software does not use *ssh*, only some of the *ssh* source and key-generation programs.

The certificates are stored in a *keys* directory, and it is up to the user or a networked file system to distribute the certificates to the group member's hosts. The certificates can use local names and are short-lived. Each member needs his public and private key and any permission certificates and the public key certificate of the group authority. The private key should be encrypted and protected, but the certificates need not be protected since they are immutable as a result of the accompanying digital signature. The group controller (which could be any member) needs the public key certificates for all members. The user interface to the group authority services are still quite primitive, but adequate for demonstration.

4.2. API

An application wishing to support secure group communication can operate in one of two modes. In one mode the application remains in the key management infrastructure. In the other mode, the application joins the group merely to get a shared key and does not participate in rekeys or rejoins.

For the first mode, the following functions are provided:

group_join() gets the group token for the given group. The group token indicates the group controller and key management port and the multicast addressing information. The member submits his credentials and joins the group.

group_open() obtains the file descriptor for the multicast port. This permits the application to operate asynchronously using *select()*.

group_msend() sends the application data on the multicast channel. The function builds the header and encapsulates the message (encryption, signatures) as specified by the group policy.

group_recv() receives application data from the multicast channel. Underneath, the function handles key management messages as well, doing rekeys and rejoins.

Appendix A has a sample multicast chat program that illustrates the use of these functions.

For the second mode the functions are:

group_join() as above, joins the group.

group_getkey() retrieves the GTEK for the group.

group_encap() encapsulates the applications data buffer according to the group policy. It is up to the application to send or whatever the resulting buffer. This function gives the application access to the library's encryption, signing, and hashing services.

group_decap() decodes a block of data according the the group policy.

The group policy specifies how the application messages should be encoded. Our implementation provides for application message integrity, privacy, and authenticity. If message integrity is required (and not privacy), then a cryptographic message digest (e.g., keyed-MD5 hash [22]) is provided. For multicast communication with a shared group key, one cannot be sure of who the sender was unless the message is signed. If sender authenticity is required, then the application message is signed by the sender. (The receiver will need public key certificates for senders in order to verify sender authenticity.) The digital signature implies message integrity as well. The signed message can optionally be encrypted if privacy is required. If only encryption is specified in the policy, a simple checksum (CRC-32) is attached and encrypted with the message to provide message integrity. The group information token specifies the encryption algorithm (DES, 3DES, IDEA, Blowfish, TSS, RCfour), providing tradeoffs in speed and cryptographic strength.

4.3. Performance

The headers, hashes, and signatures increase the size of messages, but only on the order of a hundred bytes or so, and our key management protocols require only two messages per member, so our implementation makes relatively efficient use of network bandwidth. The memory requirements for group key management are reasonable and include memory for the key management library code, storage for public keys and for four encrypting keys (the Diffie-Hellman key, GTEK, GKEK, and the previous GTEK), and possible storage for the access control list.

Many of the operations involved in group key management are very compute intensive. Digital signatures require hashing the message and then encrypting the hash. Hashes are relatively fast, but the hash time is in proportion to the message length. The digital signature and Diffie-Hellman require multi-precision multiplies of 1000-bit numbers. Random number generation for key generation can be slow and take time in proportion to the number of bits of entropy needed. Encryption/decryption is somewhat slower than hashing and also a function of message length, though for the key management function only the group key block (GTEK/GKEK) is encrypted. (Our API provides encryption/decryption for the application as well, [10] provides a more detailed analysis of the effects of hashing and encryption on message passing.)

The actual time for a member join will be a function of the CPU speed of the member and group controller hosts and of the network latency. The Diffie-Hellman calculations can take several seconds or more on a 120 MHz Pentium depending on the speed of the random number generator. A digital signature can add another second. (On a Sun Sparc 2, a member join can take ten seconds. Cao [6] provides additional data on cryptographic performance in group key management.) This time penalty is inflicted only once for each member at group join. A later rekey does not require the Diffie-Hellman exchange, since the rekey is done under the GKEK. If several members attempt to join at the same time from the same group controller, then some members will be delayed. Such congestion can occur after a rejoin message is multicast to the group. Multi-threading the group controller will provide little benefit, since the operations are compute-intensive. For large groups, multiple group controllers are needed (see section 5.3).

5. Critical Analysis

In this section, we identify the various limitations and shortcomings in our architecture. We compare our architecture with Harney's GKMP [15], Ballardie's SMKD [3], and Secure PVM [10]. We also describe how group key management might utilize the developing IP security standards.

5.1. Limitations

None of the group key management architectures and implementations have had the benefit of extensive testing or scrutiny. Though group compromise recovery is supported, it is not clear how group compromise is discovered nor how the group controller or authority is notified. Scalable compromise recovery is unsolved (see section 5.3).

Our implementation would benefit from a nice graphical interface for defining and editing group definitions. It is not clear how one should advertise or discover groups or how the group definition should be integrated with existing multicast group registries (e.g., *sdr*). Our certification infrastructure is simplistic.

5.2. Comparison with GKMP

Table 5.1 compares the group key management features of our architecture and demonstration implementation with GKMP, SMKD, and secure PVM. SMKD does not actually have an implementation, so it is not clear if it is a tractable solution to secure group communication. SMKD uses the public key to encrypt the group key, so perfect forward secrecy is not provided. Secure PVM does not authenticate its Diffie-Hellman exchanges, but that was considered an acceptable risk in view of the way slave daemons join the collaboration.

GKMP's demonstration application does not provide an independent certificate structure – keys and permissions are built and distributed by the demonstration control program. Multicast rekey is not actually implemented, though the architecture describes it. The implementation does not handle timeout/retransmission.

Feature	ours	GKMP	SMKD	sPVM
authentication	Y	Y	Y	
PKI	Y			
ACL	Y	Y	Y	Y
clearance	Y	Y		
ticket	Y			
escrow	Y			
rekey	Y	Y		
delete	Y	Y		
multicast	Y	Y	Y	
SA/SPI	Y		Y	
PFS	Y	Y		Y
implementation	Y	Y		Y
portable	Y			Y
scalable			Y	

Table 5.1: *Comparison of group key management schemes.*

The access policy is an ACL and signed clearance certificates. Both controller-initiated and member-initiated join are supported by GKMP, our architecture provides only member-initiated join. Our member-initiated join requires only two messages compared with GKMP's seven.

Our implementation provides additional modes of authorization for group join, and the group authority may specify the algorithms for encryption, hashing, and signing. Key escrow may be specified, and timers are provided for timeout/retransmission and for rekeying when keys expire. Our rekey does not require another Diffie-Hellman exchange, and multicast is used for rekey and rejoin. The implementation provides several levels of abstraction in the API and compiles and runs on most UNIX systems, even with differing byte orders. Our certificate infrastructure provides a simple and portable means to establish a working group key management system.

5.3. Scalability

Our group key management architecture scales in the sense that there is no central group manager or key distribution center for all groups. Each group can have its own independent group authority and controller. However, if one group has thousands of members, our architecture does not describe how multiple group controllers might be deployed. One could argue that it is not much of a secret if thousands of people share it, so perhaps the need for secure communication for large groups is of minor importance. Our initial design assumptions permitted us to avoid this difficult issue. Our group token (and GKMP's) is distributed to all the group members, so in theory, several members could act as group controller. The difficulty is setting up a topology to distribute the control. One needs the notion of a "near" group controller. For efficiency, the key management topology should closely match the underlying network topology. For reflector services like IRC, CU-SeeMe, or NetMeeting, one could use the hierarchy of reflectors to act as a hierarchy of group controllers. For IP multicast with DVMRP or PIM, one could entrust the MBONE routers with group controller privileges but most applications would not be willing to include such routers into their realm of trust. Wallner and Ballardie both propose scalable, tree-based key distribution. Ballardie's SMKD uses the underlying network topology but requires that the core routers be trusted. With multiple group controllers, synchronizing rekeying and member-delete becomes more difficult. A scalable group key management protocol remains an open research issue.

5.4. Integration with IPsec

Since the new IP security protocols [1] are not widely deployed, our short-term design goal was to develop a group key management architecture at the application level. We have done some early testing of ISAKMP [9] with the new IP protocols and have included an SPI field in our group information token. Ballardie's SMKD presumes the primary core has established a Security Association and holds an SPI along with the GKEK and GTEK, though it does not describe how the SA is created.

Our ISAKMP testing was done with NRL's IPv6 implementation. Assuming that implementation, our group key management architecture could be simplified. Each member would not need to do the Diffie-Hellman negotiation to set up a secure channel with the group controller. Rather, a member would use the underlying IP security protocols to establish a secure channel with the group controller. We would still need to provide a signed group information token, and the member would still need to provide signed credentials for joining the group. The group controller would verify credentials and create a Security Association based on the specifications in the group information token. However, the current IPv6 implementation does not allow an unprivileged process (i.e., our group controller) to set the key and SPI for a multicast group.

It is possible for *root* to manually set up a multicast SA with the *key* program, where INADDR_ANY is used as the source address with the multicast address and key in the SA. To establish the SA under program control we would need a privileged daemon, a *gkmd* or an extension to the ISAKMP daemon, to provide access to the kernel's SA tables. The *gkmd* would need to communicate only with local tasks. A group member would transmit to the local daemon the SA information, GTEK, multicast parameters, and SPI from the group information token. The daemon using the PF_KEY interface would establish the SA in the kernel. The member could then create the socket for the group multicast channel, and the kernel would then handle encrypting and authenticating the multicast group communication.

If the application required sender-authentication, the application would be responsible for appending digital signatures to the payload or setting up a separate SA/SPI for each sender. Group rekey and compromise recovery would still have to be managed by the application and would also require being able to update the keying material in the multicast SA through the daemon. The daemon would need to enforce SA lifetimes and remove expired group SA's from the kernel. Designing a way to create and modify a multicast SA will be the subject of further research.

6. Summary

Our group key management architecture enables a collection of authenticated and authorized network entities to establish a common encryption key. The implementation is portable and provides the necessary components for creating and verifying authorization certificates and public keys. No centralized key management facility is needed, and any number of groups can operate independently. The implementation does not require the developing IP security protocols and does not require multicast, though it can use multicast to speed up some key management functions. A simple API is provided to develop applications that need secure group communication.

Further work remains to support alternative certificate and public key algorithms. It would be beneficial to include a list of alternative certificate authorities in the group token. Additional messages would be required to pass long access control lists. More testing over other architectures is needed, specifically 64-bit. We have source for a version of *ssh* that compiles under Windows NT, so porting the group key management software to NT could be undertaken. The implementation needs more user-friendly interfaces for group creation and certificate management. Finally, there are open research issues on how to deploy multiple group controllers for large groups and how to establish the group SA under the new IP security protocols.

7. References

[1] R. Atkinson. "Security Architecture for the Internet Protocol". *RFC 1825*, August 1995.

[2] Ashar Aziz and Martin Patterson. "Design and Implementation of SKIP". In *INET'95*, June 1995.

[3] A. Ballardie. Scalable Multicast Key Distribtuion. *RFC 1949*, May 1996.

[4] A. Ballardie and J. Crowcroft. Multicast-specific Security Threats and Counter-Measures. *Proceeding of the INternet Society Symposium on Network and Distributed System Security*, February 1995.

[5] M. Burmester and Y. Desmedt. Efficient and Secure Conference Key Distribution. *International Workshop on Security Protocols*, Vol. 1189:119–129, May 1996.

[6] Cathy Cao. Group key management. Technical report, University of Tennessee, 1997. Master's Thesis.

[7] C.Ellison, W. Frantz, and B. Thomas. Simple Public Key Certificate. *IETF draft spki*, March 1997.

[8] Whitfield Diffie and Martin Hellman. "New Directions in Cryptography". *IEEE Transactions on Information Theory*, Vol. IT-22:644–654, November 1976.

[9] Douglas Maughan and Mark Schertler and Mark Schneider and Jeff Turner. "Internet Security Association and Key Management Protocol (ISAKMP)". Internet Draft - Work in Progress, February 1997. URL: ftp://ds.internic.net/internet-drafts/.

[10] T. H. Dunigan and N. Venugopal. Secure PVM. Technical report, Oak Ridge National Laboratory, 1996. ORNL/TM-13203.

[11] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *"PVM - A Users' Guide and Tutorial for Network Parallel Computing"* . The MIT Press, 1994.

[12] L. Gong. Enclaves: Enabling Secure Collaboration over the Internet. In *Proc. 6th USENIX Secuirty Symposium*, pages 149–159, 1996.

[13] L. Gong and N. Shacham. Elements of Trusted Multicasting. In *Proc. of IEEE International Conference on Network Protocols*, pages 23–30. IEEE, 1994.

[14] D. Harkins and D. Carrel. The Resolution of ISAKMP with Oakley . *IETF draft oakley-03*, February 1997.

[15] H. Harney and C. Muckenhirn. Group Key Management Protocol (GKMP) Architecture. *IETF draft gkmp-arch*, June 1996.

[16] H. Harney and C. Muckenhirn. Group Key Management Protocol (GKMP) Specifciation. *IETF draft gkmp-spec*, June 1996.

[17] I. Ingemarsson, D. Tang, and C. Wong. A Conference Key Distribution System. *IEEE Transactions on Information Theory*, Vol. 28-5:714–720, September 1982.

[18] M. Just and S. Vaudenay. Authenticated Multi-Party Key Agreement. In *Advances in Cryptology ASIACRYPT'96*, pages 26–35. Springer-Verlag, 1996.

[19] J. Kohl and B. Neuman. "The Kerberos Network Authentication Service (V5)". *RFC 1510*, October 1993.

[20] K. Koyama and K. Ohta. Identity-based Conference Key Distribution Systems. *CRYPTO '87*, Vol. 293:175–184, 1987.

[21] D. McDonald, B. Phan, and R. Atkinson. A Socket-based Key Management API . *Proceedings INET96*, June 1996.

[22] P. Metzger and W. Simpson. "IP Authentication using Keyed MD5". *RFC 1828*, August 1995.

[23] D. Paoli and A. Goscinski. The Development and Testing of the Ring Based Conference Authentication Service. Technical report, Deakin University, 1993. TR-C93/06.

[24] R. Rivest. "The MD5 Message-Digest Algorithm". *RFC 1321*, April 1992.

[25] Bruce Schneier. *Applied Cryptography*. Wiley, 1996.

[26] Tatu Ylonen. "The Secure Shell (SSH) Remote Login Protocol". Internet Draft - Work in Progress, November 1995. URL: http://www.cs.hut.fi/ssh.

[27] D. Wallner, E. Harder, and R. Agee. Key Management for Multicast: Issues and Architecture. *IETF draft wallner-key*, July 1997.

[28] William Simpson and Phillip Karn. "The Photuris Session Key Management Protocol". Internet Draft - Work in Progress, November 1995. URL: ftp://ds.internic.net/internet-drafts/.

Appendix

A. Implementation notes

Our test implementation uses a number of functions from the *ssh* distribution (version 1.2.16), including the public key file management functions (and the program *ssh-keygen*, RSA glue routines, the RSA reference library, various cipher routines (DES, IDEA, RC4, TSS), and GNU's GMPlib for multi-precision arithmetic. (We do not actually use the *ssh* keys in *.ssh*, rather generate our own key files using *ssh-keygen* and certify them with our *certify* program.) The code has been tested on Sun OS, SGI IRIX, and Linux and with architectures having different byte orders.

The major modules that we wrote to support our group key management are:

gkmp.h contains the main data structures and constants (see below).

gkmp_subs.c contains the functions for building and parsing key management messages. The group controller code is also in this file.

netsubs.c contains the routines for managing IP/UDP transport, including multicast, and timeout/retry.

dh_subs.c contains the code for doing a Diffie-Hellman key exchange.

rsasign.c contains the code for doing an RSA signature and verify.

hmac-md5.c contains the code for doing a keyed MD5.

truerand.c is Matt Blaze's random number generator for UNIX.

ga.c is the group authority code that provides a signed group token and assigns the group controller to the first contacting member.

cert.h defines the data structures for our certificate infrastructure.

cert_subs.c contains the code for creating and verifying the various certificates (key, group info, ticket, permission).

certify.c is a group authority service program to create certificates for ssh public keys and for tickets and permissions.

Most of the key data structures are defined in *gkmp.h*. Its contents follow.

```
/* gkmp.h  */

#define GKMP_PORT 8642
#define GK_MAXBUFF 4096
/* bytes in a signature */
#define GK_SIGLTH 128
#define GK_MD5LTH 16
#define GK_KEY_TIMEOUT 60
#define GK_TIMEOUT 11
#define GK_RETRY 3


/* messages types */
#define GKMP_APP 0
#define GKMP_CLOSE 1
#define GKMP_REQGC  2
#define GKMP_ERROR 3
#define GKMP_JOIN  4
#define GKMP_REPLYGC  5
#define GKMP_REPLYJOIN  6
#define GKMP_REJOIN  7
#define GKMP_KEYUPD  8
/* GC initiated messages */
#define GKMP_REKEY 20
#define GKMP_DOREJOIN 21

/* appl message/policy flags */
#define GKF_ENC 1
#define GKF_SIG 2
#define GKF_HMAC 4

/* payloads */
#define GPL_ERROR 1
#define GPL_GRPINFO 2
#define GPL_SIG 3
#define GPL_CERT 4
#define GPL_DH1 5
#define GPL_DH2 6
#define GPL_OPAQUE 7
#define GPL_KEYS 8
#define GPL_PERMS 9
```

```
#define GPL_TICKET 10
/* note: escrow payload is invisible */

/* error codes */
#define GERR_NOGA 1
#define GERR_NSGRP 2

/* message struct's */
#define COOKIE_LEN 4

struct GK_Hdr {
  unsigned char type;   /* message type */
  unsigned char rsvd1;
  unsigned char keyid;  /* keyid 1-255 */
  unsigned char flags;  /* whether appl is encrypted/signed  from policy */
  unsigned char ga_cookie[COOKIE_LEN];  /* nonce gen'd by ga */
  unsigned char next_payload;
  unsigned char version;
  unsigned short len;    /* net byte order */
};

struct GK_Payload {
  unsigned char next_payload;
  unsigned char reserved;
  unsigned short payload_len;
};

#define APP_HDR_LEN  (sizeof(struct App_Hdr))
struct App_Hdr {
  unsigned char type;   /* message type */
  unsigned char rsvd1;
  unsigned char keyid;  /* keyid 1-255 */
  unsigned char flags;  /* whether appl is encrypted/signed  from policy */
  unsigned long ulth;  /* length of original user data (net byte order) */
};

/* group info */
#define GKMP_KEYLTH 16

enum LEVELS {LVL_U, LVL_C, LVL_S, LVL_TS};
enum HASHES {GK_MD5, GK_SHA, GK_RIPEM, GK_HAVAL};
```

```
enum SIG_ALGS {GK_RSA, GK_DSA, GK_ECC};
/* permission categories -- bits */
#define GK_CAT1 1
#define GK_CAT2 2
#define GK_CAT3 4
#define GK_CAT4 8


/* grp_policy  -- AND bit set implies others that are set are all required */
/*  0 implies, just need key signed by ga */
#define GK_AND 128
#define GK_ACL 64
#define GK_SECLBL 32
#define GK_TICKET 16
#define GK_ESCROW 8
  /* also see message flags */

struct Group_Info {      /* in net byte order */
  char * grp_id;
  unsigned char grp_cipher;
  unsigned char grp_keylth;   /* # of bytes of keying material */
  unsigned char grp_hash;
  unsigned char grp_sig;
  unsigned char grp_seclvl, grp_seccat; /* security label (level, categories) */
  short grp_port;           /* application port */
  unsigned long grp_timbeg, grp_timend;   /* lifetime, UNIX time() seconds */
  unsigned long grp_maddr;     /* 0 or mcast addr in net byte order */
  unsigned long grp_spi;
  char * grp_gc;
  short grp_gcport;        /* key mgt port */
  unsigned char grp_policy;  /* auth required */
  unsigned char grp_ttl;     /* mcast ttl */
  int grp_member_cnt;
  char * * grp_members;      /* ACL */
};
```

A simple chat program that uses multicast and group key management follows. Key management is mostly hidden from the application. Key establishment is handled in *group_join()* and lifetime rekeying are handled by *group_recv()*. The group token specifies the group policy, including the type of encryption and integrity that is performed for the application by *group_msend()* and *group_recv()*.

```
#include <sys/time.h>
#include <sys/types.h>
#include <stdio.h>
#include "gkmp.h"
fd_set rdfds, wrtfds, tmprd, tmpwrt;

main()
{
        int fd,grpid;
        char buff[1024];
        char hname[64];
        int lth,n;

        gethostname(hname,sizeof(hname));
        FD_ZERO(&rdfds);
        FD_ZERO(&wrtfds);
        grpid = group_join("smtalk","thdsun",GKMP_PORT,hname);

        fd = group_open(grpid);
        FD_SET(0, &rdfds);
        FD_SET(fd, &rdfds);
        while(1){
                tmprd =rdfds;
                tmpwrt = wrtfds;
                n = select(FD_SETSIZE, &tmprd, &tmpwrt,NULL,NULL);
                if (n <0){perror("select"); exit(1);}
                if (FD_ISSET (fd, &tmprd)) {
                        group_recv(fd,buff,sizeof(buff),&lth);
                        if (lth) printf("%s\n",buff);
                } else if (FD_ISSET (0,&tmprd)){
                        if (gets(buff) == NULL) break;
                        group_msend(grpid, buff,strlen(buff)+1);
                }
        }
        close(fd);
}
```