Keystone: A Group Key Management Service*

Chung Kei Wong
HRL Laboratories, LLC
3011 Malibu Canyon Rd
Malibu, CA 90265

Simon S. Lam
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712

Abstract

A major problem area in securing group communications is group key management. In this paper, we present the design and architecture of a scalable group key management system called *Keystone*. Keystone uses a novel *key graph* technique for scalable group key management. In Keystone, the authentication of client identity can be offloaded to one or more *registrars* to improve performance. For efficient and reliable key updates, Keystone uses UDP/IP multicast delivery with forward error correction (FEC) to reduce message loss, and provides an efficient *re-synchronization* mechanism for clients to reliably update their keys in case of actual message loss. A prototype of Keystone has been implemented and its performance results are reported.

1 Introduction

Many emerging network applications (such as teleconference and information dissemination services) are based upon a group communications model. As a result, securing group communications becomes a critical networking issue. Recently, Internet Research Task Force (IRTF) has formed Secure Multicast Research Group (SMuG) [4] to investigate the problem of securing group communications. One major problem area is group key management which is concerned with the secure distribution and refreshment of keying material.

A group key management system establishes and maintains *group keys* for groups of clients. A group key may be an encryption key, a signing key, a security association in IPsec, etc. In this paper, we describe the design and architecture of a scalable group key management system called *Keystone* which uses a novel *key graph* technique [8] for scalable group key management.

In Section 2, we illustrate the key graph technique by a simple example. In Section 3, we describe the architecture and operations of Keystone. In Section 4, we discuss how to efficiently and reliably update keys in Keystone. Conclusions are in Section 5.

2 Key Graph

Assume there is a trusted and secure *key server*[1] responsible for group access control and key management, and the key server uses *key graphs* [8] for group key management. A *key graph* is a directed acyclic graph with two types of nodes, *u-nodes* representing members and *k-nodes* representing keys. A member u is given key k if and only if there is a directed path from u-node u to k-node k in the key graph.

Consider a group of nine members. Assume the key server uses the *key tree* (a special key graph) in Figure 1(a) for the group. In this group, member u_9 is given three keys k_9, k_{789}, and k_{1-9}. Key k_9 is called the *individual key* of u_9 because it is shared with the key server only. Key k_{1-9} is the *group key* which is shared with every member, and k_{789} is an auxiliary key shared with u_7 and u_8.

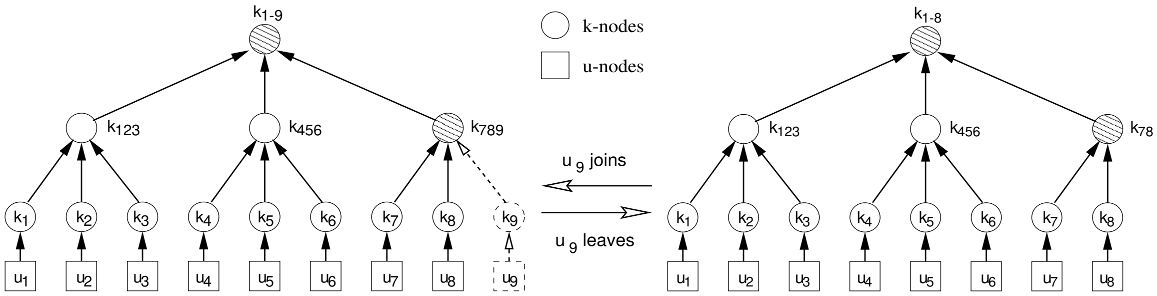
Assume the key server changes the group key after each join/leave. For u_9 to leave, the key server changes the group key k_{1-9} to k_{1-8} and the auxiliary key k_{789} to k_{78}. To distribute the new keys to the remaining members using a *group-oriented* rekeying strategy [8], the key server constructs the following *rekey message* and multicasts it to all members:

$$\{k_{1-8}\}_{k_{123}}, \{k_{1-8}\}_{k_{456}}, \{k_{1-8}\}_{k_{78}}, \{k_{78}\}_{k_7}, \{k_{78}\}_{k_8}$$

where $\{k'\}_k$ denotes key k' encrypted with key k. Only the remaining members can update their keys by decrypting appropriate keys in the rekey message.

*This research was performed at the University of Texas at Austin as part of the doctoral dissertation of Chung Kei Wong. Research sponsored in part by NSA INFOSEC University Research Program grant no. MDA904-98-C-A901 and National Science Foundation grant no. ANI-9977267. In *Proceedings International Conference on Telecommunications*, Acapulco, Mexico, May 2000.

[1] A key server may be distributed and/or replicated to enhance reliability and performance.



(a) before u_9 leaves (after u_9 joins) (b) after u_9 leaves (before u_9 joins)

Figure 1: Key trees before and after a leave (join).

Similarly, for u_9 to join the group of eight members in Figure 1(b), the key server constructs the following rekey message and multicasts it to all members:

$$\{k_{1-9}\}_{k_{1-8}},\ \{k_{789}\}_{k_{78}}$$

The key graph technique is scalable to large dynamic groups because the key server's average computation and communication costs per join/leave increase linearly with the *logarithm* of the group size.

3 Keystone Architecture

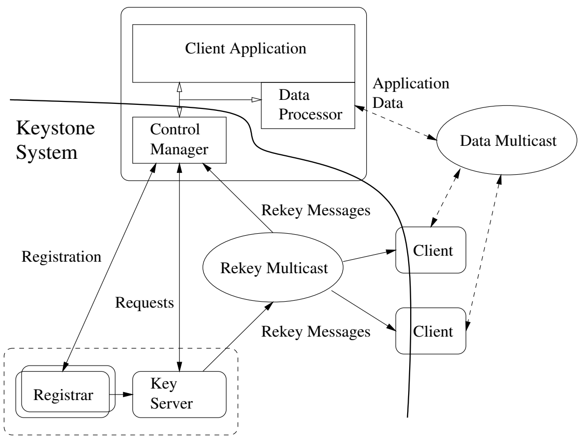


Figure 2: Keystone architecture.

For a key server to exercise access control, i.e., to grant or deny a request, the identities of clients must be authenticated. The key server may authenticate each client using an authentication protocol, such as SSL 3.0 [2], and then generate and share the client's individual key which is used to protect future communications between them. This process of associating a client's identity with its individual key is called *client registration*.

Client registration using an authentication protocol, such as SSL 3.0, is computationally expensive,

	Pentium II Linux server		Ultra 1 Solaris server	
	512-bit	1024-bit	512-bit	1024-bit
512-bit client	7.8	26	36	140
1024-bit client	10	29	42	146

Table 1: Server processing time (ms) for one SSL 3.0 connection using SSLeay.

and a key server becomes a bottleneck when the client registration rate is high, e.g., during initial group setup. Table 1 shows the server processing time to authenticate a client using SSL 3.0 from SSLeay (which is a publicly available implementation of SSL 3.0). Both the client and server use certificates (512-bit or 1024-bit RSA) to mutually authenticate each other, and the certificates are signed directly by a Certificate Authority CA using a 1024-bit RSA key. The SSL connection is established using RSA key exchange, triple DES (CBC mode) encryption, and SHA digest algorithm.

To solve this problem, Keystone uses one or more *registrars* to offload client registration from a key server.[2] Machines running registrars can be added or removed dynamically. Moreover, different registrars may use different authentication services to authenticate different sets of clients at the same time.

Figure 2 shows a typical configuration of Keystone. There are many *client control managers* (one for each client), one or more *registrars*, and only one *key server*. The control manager of a client is responsible for client control functions, e.g., sending requests and processing rekey messages. Each client also has a *data processor* which is not a part of Keystone. A data processor uses group keys from its control manager to perform application data functions, e.g., encryption, decryption, signing, and verification.[3] A registrar

[2] A key server and a registrar may be combined together into one physical entity (a key server with embedded registrar).

[3] In the balance of this paper, we use "client" to mean "client control manager" unless otherwise stated.

authenticates the identities of clients and distributes an authenticated client's individual key to the client and the key server. The key server processes requests from clients, changes keys, and distributes new keys to clients using rekey messages. Currently, Keystone can deliver rekey messages to clients using either unicast or multicast (see Section 4).

In the following, the operations of Keystone are described in detail. To be more concrete in the descriptions, we assume the use of SSL 3.0 for two entities to mutually authenticate and establish a secure communication channel. However, other authentication protocols can be used.

3.1 Registrar setup

After a key server S has been initialized, one or more registrars are setup to handle client registration. Each registrar R makes a TCP connection to the key server S, and then they mutually authenticate and establish a secure communication channel over the TCP channel using SSL. The key server generates a secret key k_R and sends the secret key and a client list to the registrar through the secure channel. After that, the secure channel is terminated. but the TCP channel is kept connected. Figure 3 shows this registrar setup process. We use the notation $x \Leftrightarrow y$ to represent the mutual authentication and establishment of a secure channel between x and y, and the notation $x \Rightarrow y : z$ to denote the sending of message z from x to y through a secure channel.

(1)	$R \Leftrightarrow S$	:	using SSL
(2)	$S \Rightarrow R$	:	registrar key k_R, client list

Figure 3: Registrar setup.

The secret key k_R is called *registrar key* and is used to encrypt/decrypt future communications between the registrar and the key server (through the TCP channel). The client list contains the identities and ID numbers of clients but does not contain access control information.

3.2 Client registration

(1)	$C \Leftrightarrow R$	:	using SSL
(2)	$R \Rightarrow C$	:	ID_C, k_C
(3)	$R \rightarrow S$	:	$\{ ID_C, k_C \}_{k_R}$

Figure 4: Client registration.

For a new client C to register with a registrar R, the client makes a TCP connection to the registrar, and then they mutually authenticate and establish a secure communication channel over the TCP connection using SSL. The registrar generates the client's individual key k_C and sends the client its individual key and ID number through the secure channel. Then, the registrar encrypts the client's individual key and ID number with registrar key k_R, and sends them to the key server. After that, the secure channel and TCP connection between the client and the registrar are terminated. Figure 4 shows this client registration process. We use the notation $x \rightarrow y : z$ to denote the sending of message z from x to y through a reliable channel.

3.3 Request and reply

After having registered, a client C may send requests to the key server S. The client makes a new TCP connection to the key server if there is no existing TCP connection between them. The client encrypts the request with its individual key, and sends the encrypted request to the key server through the TCP channel.

A request may contain operations to more than one group (if there are multiple groups). There are three types of operations: *join*, *leave*, and *re-synchronize*. A request may have both join and leave operations or only re-synchronize operations. A client uses re-synchronize operations to get the keys of currently joined groups, e.g., after losing some rekey messages (see Section 4.2).

(1)	$C \rightarrow S$	:	$\{ \text{request} \}_{k_C}$
(2)	$S \rightarrow C$	:	$\{ \text{ack} \}_{k_C}$, $\{ \text{ind. rekey} \}_{k_C}$

Figure 5: Request and reply.

After decrypting and processing the request, the key server encrypts a reply with the client's individual key and sends the encrypted reply to the client through the TCP channel. The reply consists of an acknowledgment and an *individual rekey* message. The acknowledgment contains the result (granted or denied) of each operation requested and other information. The individual rekey message consists of the keys to be added, deleted, or updated by the requesting client. See Figure 5.[4] After that, the TCP channel is terminated if the key server does not use the TCP channel to send rekey messages to the client (see Section 4); otherwise, the TCP channel is kept connected for the key server to send rekey messages to

[4]The communications between the client and the key server are also protected from modification and replay attack by MAC and sequence number.

	Number of original units, s									
	1	2	3	4	5	6	7	8	9	10
$\alpha = 0.0$	0.100	0.190	0.271	0.344	0.410	0.469	0.522	0.570	0.613	0.651
$\alpha = 0.5$	0.010	0.028	0.009	0.016	0.005	0.008	0.003	0.004	0.001	0.002
$\alpha = 1.0$	0.010	0.004	0.001	0.000	0.000	0.000	0.000	0.000	0.000	0.000

(a) $p = 0.1$

$\alpha = 0.0$	0.200	0.360	0.488	0.590	0.672	0.738	0.790	0.832	0.866	0.893
$\alpha = 0.5$	0.040	0.104	0.058	0.099	0.056	0.086	0.050	0.073	0.044	0.061
$\alpha = 1.0$	0.040	0.027	0.017	0.010	0.006	0.004	0.002	0.001	0.001	0.001

(b) $p = 0.2$

Table 2: Message loss probability $P_m(s, r, p)$ where $r = \lceil \alpha \times s \rceil$.

	Number of original units, s														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
For "rs1", $r =$	2	3	4	4	4	5	5	5	6	6	6	7	7	7	7
For "rs2", $r =$	4	5	6	6	7	8	8	9	9	10	10	11	11	12	12

Table 3: Number of repair units r for repair schemes "rs1" and "rs2".

the client. Note that the key server also constructs and sends rekey messages to other clients.

4 Key Updates

After changing keys in a key graph, a key server distributes new keys to clients using rekey messages. For reliable key updates, a key server may use a reliable delivery mechanism, e.g., TCP or a reliable multicast protocol [1, 3, 6], to send rekey messages to clients. Keystone can use TCP to deliver rekey messages by keeping a TCP connection to each current member. This approach is clearly not scalable to large groups. On the other hand, efficient reliable multicast protocols are not available for large scale use.

To provide efficient and reliable key updates, Keystone uses UDP over IP multicast for efficient rekey message delivery and forward error correction (FEC) technique for message loss reduction. Keystone also provides an efficient *re-synchronization* mechanism for clients to reliably update their keys in case of actual rekey message loss.

4.1 FEC + UDP/IP multicast

A key server can efficiently send a rekey message to a large number of clients using UDP over IP multicast. However, UDP is unreliable and there may be packet losses. Moreover, if a rekey message is larger than the maximum transmission unit (MTU), IP fragmentation occurs and the message is broken into several IP packets. This increases the message loss probability since a receiver needs to get all IP packets in order to reconstruct the message.

To reduce message loss probability, Keystone uses *forward error correction* (FEC) technique [5]. Each rekey message is partitioned into one or more 512-octet units.[5] Then, a number of *repair units* (also called *parity units*) are computed from the original units. Each unit is sent using UDP over IP multicast. The number of repair units r is a function of the number of original units s, and the function is called a *repair scheme*. Since a receiver can re-construct the rekey message from any s of the $s + r$ original and repair units, the rekey message is less likely to be lost.

Assume UDP packet losses are independent and with probability p for a receiver (which is about 10%-20% for UDP packets over MBone [9]). Then, the message loss probability $P_m(s, r, p)$ for the receiver is given by the following equation.

$$P_m(s, r, p) = \sum_{i=r+1}^{s+r} \binom{s+r}{i} p^i (1-p)^{s+r-i}$$

Table 2 shows $P_m(s, r, p)$ for the repair scheme $r = \lceil \alpha \times s \rceil$ with three different α values, 0.0, 0.5, and 1.0.[6] If no repair packet is used (i.e., $\alpha = 0.0$), a larger rekey message is partitioned into more units, and the message loss probability increases. The probability $P_m(s, r, p)$ decreases when α increases, i.e., more repair units are used.

We can bound the message loss probability by computing the number of repair units needed from the $P_m(s, r, p)$ equation. We use the notations "rs1" and "rs2" to denote the repair schemes to make $P_m(s, r, p) \leq 0.001$ for $p = 0.1$ and $p = 0.2$, respectively. See Table 3.

Table 4 shows the average server processing time per join/leave request for different repair schemes on a Ultra 1 machine. Our implementation uses the

[5] Although most path MTUs are as large as 1500 octets, an IP host machine is only required to receive at least a 576-octet IP datagram.

[6] In Table 2, a probability less than 0.0005 is denoted by "0.000" (which is not probability zero).

software erasure coder by Rizzo [7] to generate repair units. We measured the average server processing time for group-oriented rekeying using DES-CBC encryption, MD5 message digest, and 512-bit RSA digital signature. The rekey message signing time is about 46.6 ms. We observe that for group size upto 8192, the use of FEC increases the processing time by at most 1.1 ms which is about 2% of the total processing time. In other words, the use of FEC substantially decreases the message loss probability with only a slight increase in the processing time.

	Group size					
	256	512	1024	2048	4096	8192
$\alpha = 0.0$	50.7	51.0	51.0	51.2	51.3	51.4
$\alpha = 0.5$	50.9	51.3	51.4	51.5	51.6	51.7
$\alpha = 1.0$	50.8	51.4	51.4	51.5	51.7	52.0
"rs1"	50.9	51.5	51.6	51.8	51.9	52.0
"rs2"	51.2	51.9	52.1	52.3	52.4	52.5

Table 4: Average server processing time (ms) per join/leave request.

4.2 Re-synchronization

Since FEC does not provide 100% reliability, rekey messages may still be lost. One straight-forward way to deal with lost rekey messages is for a client to request the lost rekey messages to be re-transmitted. This approach is inefficient especially when the number of lost rekey messages is large, e.g., in a long loss burst that lasts for several seconds or minutes [9]. First, a key server (or a repair entity) needs to store many rekey messages. Second, most of these recovered rekey messages contain old auxiliary keys that are no longer needed by the client.

Instead of using message retransmission, Keystone provides an efficient *re-synchronization* mechanism for clients to reliably update their keys in case of message loss. When a client detected rekey message losses, it sends a *re-synchronize* request to the key server. After receiving a re-synchronize request, the key server encrypts the current group and auxiliary keys (and possibly some previous group keys if needed) for each requested group with the client's individual key, and sends the encrypted keys back to the client. Since no expensive digital signature is needed, the server processing time for a re-synchronize request is only 2.4 ms to 2.5 ms on a Ultra 1 for a group with size from 256 to 8192 using DES-CBC encryption.

5 Conclusions

We have designed and implemented a group key management system called Keystone which consists of three components: *client control manager*, *registrar*, and *key server*. The expensive client registration process can be offloaded from a key server to one or more registrars. Therefore, the client registration rate can be increased by using more machines to run registrars. Moreover, different registrars can be used to provide different authentication services for different sets of clients (at the same time).

For a key server to distribute new keys to a large group of clients, UDP over IP multicast is efficient but not reliable (with a loss probability around 10% to 20% over MBone). To decrease the message loss probability, Keystone uses forward error correction (FEC). The use of FEC substantially reduces (by orders of magnitude) the loss probability of rekey messages with only a slightly increase in the server processing time. Moreover, Keystone provides an efficient re-synchronization mechanism for clients to reliably update their keys in case of actual message loss.

References

[1] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang. A Reliable Multicast Framework for Light-Weight Sessions and Application Level Framing. In *Proceedings of ACM SIGCOMM '95*, 1995.

[2] Alan O. Freier, Philip Karlton, and Paul C. Kocher. The SSL Protocol Version 3.0. Work in progress, Netscape Communications, November 1996.

[3] Hugh W. Holbrook, Sandeep K. Singhal, and David R. Cheriton. Log-Based Receiver-Reliable Multicast for Distributed Interactive Simulation. In *Proceedings of ACM SIGCOMM '95*, 1995.

[4] Internet Research Task Force (IRTF). The Secure Multicast Research Group (SMuG). http://www.ipmulticast.com/community/smug/.

[5] Jörg Nonnenmacher, Ernst Biersack, and Don Towsley. Parity-Based Loss Recovery for Reliable Multicast Transmission. In *Proceedings of ACM SIGCOMM '97*, 1997.

[6] Sanjoy Paul, Krishan K. Sabnani, John C.H. Lin, and Supratik Bhattacharyya. Reliable Multicast Transport Protocol (RMTP). *IEEE Journal on Selected Areas in Communications*, 15(3), April 1997.

[7] Luigi Rizzo. Effective Erasure Codes for Reliable Computer Communication Protocols. *Computer Communication Review*, April 1997.

[8] Chung Kei Wong, Mohamed Gouda, and Simon S. Lam. Secure Group Communications Using Key Graphs. In *Proceedings of ACM SIGCOMM '98,* Vancouver, B.C., September 1998.

[9] Maya Yajnik, Jim Kurose, and Don Towsley. Packet Loss Correlation in the MBone Multicast Network. In *IEEE Global Internet Conference*, 1996.