

Chapter 8. Broadcast and Multicast Key Distribution and Authentication

As we introduced so far, the security mechanisms are aimed to a pair of communication entities for providing mutual authentication and key establishment, then for delivering protected communications, such as integrity checking and confidentiality. However, if the communication scenarios are changed, the solutions to the one-to-one (i.e., unicast) communication cannot be directly adopted to one-to-many (1TM) which is referred to as *multicast communication*, in this case, one entity sends a common message to the group of entities under some protocols. The other possible communication scenarios are many-to-one (MT1) or many-to-many (MTM) communications.

In this chapter, we introduce concepts of star based or binary key tree based key management and distribution, and key revocation methods in the first two sections, then we introduce hash chains and Merkle trees for authentication in Sections 3 and 4. It is worth to point it out that the methods introduced here have varieties of applications which are not limited to secure multicast communications.

1 Basic Models for Multicast Key Distribution

We consider a system with n communication entities $\{u_0, u_1, \dots, u_{n-1}\}$ and one key management and distribution (KMD) server. KMD server responds to distribute a common key to each entity. This key is referred to as a *multicast key or group key*, which will be used for protection of multicast communication in the group.

Trust Model:

- a) Public-key based approach: KMD server's public-key is bound with a certain group relation or membership relation for carrying out the group communication, which is known to each entity in the multicast group. For each entity, KMD server is employed as a certification authority to issue a certificate to a pair of public-key and private key for each entity. Those private keys are referred to as *individual keys*.
- b) Symmetric-key approach: There exists a secure link between KMD server and an entity for which they can pre-share a key securely priori to its joining the group.

Pre-conditions: KMD server and entities employ their own security associations which consist of a pseudorandom number generator (PRNG), an encryption and decryption operators $E_k(m)$ and $D_k(m)$ (e.g., AES Rijndael), a key deviation function (KDF), and the other requested crypto schemes.

⁰Copyright ©2008 L. Chen and G. Gong. All rights reserved. May be freely reproduced for educational or personal use.

In this chapter, the keys which we used is deviated from KDF using pre-shared keying materials in a symmetric key case. However, we omit this step in this chapter for simplicity. The reader should keep this in mind, i.e., any keys shared from the pre-shared keying materials cannot be directly used in the protected communications as keys. The keys used for either encryption or authentication must be derived from KDF using the shared keying materials as a *seed*. In the public-key approach, the shared keys from running a mutual authentication and key establishment protocol, introduced in Chapter 3, is deviated from KDF using the shared Diffie-Hellman or RSA or ECC keys (see Appendix F).

Security requirement (PRECEDING/SUCCEEDING SECRECY): A multicast system is said to have *preceding secrecy* if any newly joining member cannot decrypt the previous established protected communication sessions using his keys, and it said to have *succeeding secrecy* if a member who left the system or his individual key is revoked cannot decrypt the protected communication sessions after his leaving or the key revoked.

In other words, a new entity can only decrypt the protected communications after his joining and cannot decrypt any preceding sessions, and a former member or an entity with a revoked key is excluded from the succeeding sessions. Note that in the literature, some times this property was referred to as forward or backward secrecy. However, there is a term called perfect forward secrecy, defined in Chapter 3, which has a totally different meaning. So, we avoid to use them in order not to create some confusion about technical terms.

1.1 Key Sharing Scenarios

There are three scenarios for distribution of keys in a multicast communication network.

Case 1. Single common key shared in the whole multicast network, which is distributed by a KMD server. The KMD server responds to handling new entities joining or entities leaving the network. We assume that adversary cannot launch any attack from insider, i.e., the entities in the network cannot give this key to adversary on purpose. So the attack is from outsiders who try to figure out the shared key. The main disadvantages of this scenario are as follows.

Two Main Disadvantages:

- (i) **Rekeying Process:** Whenever an entity joining or leaving, the KMD server will revoke the group key, and run a rekeying process, which is essentially the same process as the initial phase for the key distribution, in order to provide the preceding/succeeding secrecy.
- (ii) **Robustness:** An adversary who compromises any entity and extracts his multicast key will compromise the security of the whole system. Tamper-proof memory for the shared key is a possible solution for this attack.

Case 2. Each entity pre-shares an individual key with the KMD server. When KMD server sends a message to the multicast group, he uses the individual key for each entity for providing protection

Table 1: Comparisons of Three Scenarios of Key Sharing Status

Key sharing status	$ K(u_i) $	$ K(KMD) $	$ M(x) $	Rekeying needed
Case 1	2	$n + 1$	1	Yes
Case 2	1	n	n for KMD or $n - 1$ for u_i	No
Case 3	n	n	$n - 1$	No

(encryption or source authentication). In this case, since the communication is protected by an individual key for each entity, there is no problems for rekeying, since it only involves the entity leaving or joining the network. However, this approach essentially is of unicast, which does not save any communication cost.

If one of the group members is the sender who sends a message to the rest of the group member, he will establish a protected unicast communication with the KMD server using his individual key shared with the KMD server. Then the KMD server, on behalf of this entity, to decrypts and/or authenticates this message and unicast it to each other group member in the same way as described above.

Case 3. Pairwise key shared in each pair of two entities. In this case, each entity could be in the role of the KMD server. Whenever an entity sends a message to the rest of the group members, this entity takes the KMD server's role in Case 2. Thus, it shares the same concerns as Case 2. Note that in this case the KMD server is only involved in the issuing certificates in the public-key case or pre-load keys into entities in the symmetric-key case, and it is not involved during the multicast communication stage.

In Table 1, we list the communication cost and memory requirements for each scenario using the following notations.

- $K(x)$ denotes the set of keys of a party x . Then $|K(u_i)|$ and $|K(KMD)|$ represent the numbers of keys in entity u_i and KMD server respectively.
- $M(x)$ is the set consisting of ciphertexts or authentication tags that a party x which could be KMD server or any entity in the group sends to the multicast group in a protected manner for delivering one multicast message.

In the above three scenarios, basically, the last two directly apply the unicast protections to the multicast case. Only Case 1 takes the advantage of multicast communication, i.e., makes the communications more efficient. This scenario has wide spectrum applications, such as multimedia transmission, vehicular ad hoc networks, and wireless sensor networks, just list a few. After each

join/leave, KMD server needs to update a shared key with the new group. In other words, KMD server needs to redistribute a new group key to a new multicast group. This constitutes a cost of k encryptions where k is the size of the new group.

1.2 A Naive Protocol

In the following, we describe the process how to share a common key between entities and the KMD server, i.e., the scenario described in Case 1 in detail in order to inspire how to design a better protocol. We consider the parameter set $(n, R, N_{init}, N_{leave}, N_{join})$ where

- n is the number of the group members.
- R denotes the number of keys for each entity, which determines the robustness of the protocol for being resistant against to attacks on compromising keys.
- N_{init} , N_{leave} , and N_{join} are the maximum numbers of messages that KMD will send to entities in the initial key distribution phase and the re-keying phase for a leave or a join, respectively, which measure the communication overhead of these three processes.

Note that in the following protocols, we present the public-key approach. For the symmetric-key case, the step for sharing individual keys between KMD server and entities will be replaced by the process that KMD server pre-loads those keys to the entities which will be present in a multicast group.

PROTOCOL Θ : STAR TOPOLOGY BASE MULTICAST KEY DISTRIBUTION

Initial Phase:

1. KMD server: For each i ,
 - (a) Run with each entity the mutual authentication and key establishment protocol (MAKE) to share a key, say k_i with entity i , which serves as the individual key between the entity i and KMD server.
 - (b) Randomly pick r , and send $c_i = E_{k_i}(r)$ to entity i by unicast.
2. Entity i : Decrypt c_i using the individual key k_i to obtain r , which serves as a multicast key.

Rekeying Phase: KMD server does the following operations.

1. If an entity leaves the network, then the KMD server runs the step (b) in the initial procedure except for the leaving entity.
2. If a new entity joins the network, then the KMD server runs the step (a) with the new entity, and execute the step (b) within the new group.

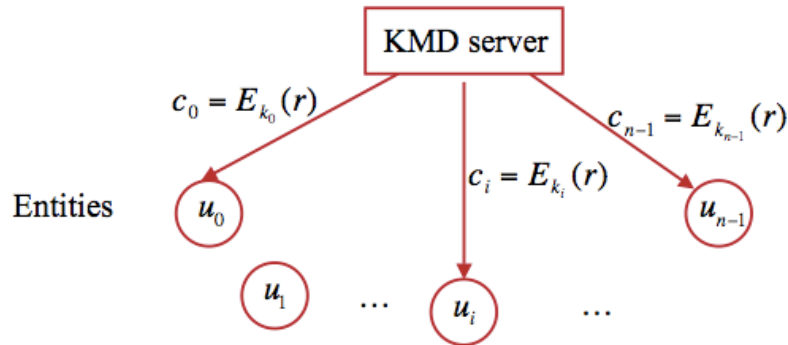


Figure 1: Multicast key distribution: a naive protocol

In this protocol, each entity holds two keys where one is the individual key and the other the group key, i.e., $R = 2$. Once an entity leaves or joins the network, the group key is revoked, and a new group key is delivered to a new group. Thus, the numbers of messages that KMD server sends in the initial key distribution phase and the re-keying phase for a leave or a join, respectively, are equal to $N_{init} = n$, $N_{leave} = n - 1$ and $N_{join} = n + 1$. Thus the parameters of this protocol are $(n, 2, n, n - 1, n + 1)$. As pointed before, this protocol has two disadvantages. One is that the rekeying process is frequently executed. When the number of the entities in the networks is dynamically changed, it brings a large communication overhead. The other is that the attacker only needs to compromise one key, then the system is crashed.

The key structure of the naive protocol can be considered as the network has a star topology, as shown in Figure 1 for which the KMD server is at the center, and all the entities are connected to the center. This also can be considered as a tree where the shared key is placed at the root and the individual keys are placed at leaves. Each entity holds keys along the path from itself to the root.

This is a simplest tree. The question is whether it is possible to yield a trade-off among the number of the keys held by each entity, the communication overhead in the rekeying process and tolerant to key compromising attack if we attach entities to the leaves of a general tree. The answer is a yes. We will explain this approach in the next section. In other words, we will introduce some redundancy to the scheme such that it could reduce the communication overhead for rekeying processes as well as it could provide robustness to compromising key attack.

2 Logic Key Tree Based Multicast Key Distribution

Before we present a logic key tree based key distribution for a multicast group, we need some terminologies from graph theory.

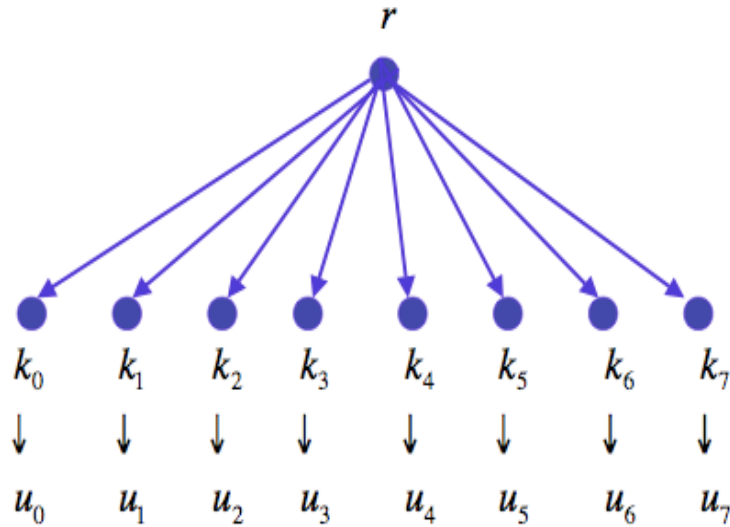


Figure 2: Key Graph of a Group with 8 Entities

2.1 Basic concepts of graph theory

A graph G consists of two types of elements, namely vertices and edges. Every edge has two endpoints in the set of vertices, and is said to *connect* or *join the two endpoints* and we also say that these two vertices are *adjacent*. An edge can thus be defined as a set of two vertices (or an ordered pair, in the case of a directed graph). It is common to represent a graph by a *drawing* where we represent each vertex by a point in the plane and represent edges by line segments or arcs joining two some of the pairs of points. The *degree* of a vertex is the number of the edges that the vertex being connected.

A *binary tree* is a connected acyclic graph such that the degree of each vertex is no more than 3, for example, Figure 3 is a binary tree. A vertex (it is also called a *node* if the tree is a directed tree), is called a *leaf* of the tree if it has only one edge, and a vertex is called the *root* of the tree if it has two edges, but none of the edges is adjacent to a leaf. For example, in Figure 3, vertices k_i are leaves and r is the root.

A *path* from one vertex to another vertex in a binary tree consists of an alternating sequence of vertices and edges so that it starts and ends at vertices. The *length* of a path is the number of edges in the path. The *depth* of a vertex is defined as the length of the path from the vertex to the root. Thus, the root vertex has depth 0. For example, any leaf in Figure 3 has depth 3, and vertex k_{10} has depth 2.

A vertex with depth k is a *parent (vertex)* of an adjacent vertex with depth $k + 1$ and is a *child (vertex)* of an adjacent vertex with depth $k - 1$. *Siblings* are vertices that share the same parent vertex. For example, in Figure 3, k_{12} is the parent vertex of vertices k_4 and k_5 , and it is a child

vertex of the vertex k_{21} on the other hand. The height of a tree is equal to the largest depths. A *subtree* with a vertex as a root consists of all children with depths larger than the depth of the vertex. A *full* binary tree is a tree in which every vertex has zero or two children. A binary tree is a *complete* binary tree if the depths of all its leaves differ by 1.

Note that we abuse the notation, i.e., we use k_i or k_{ij} to represent both the vertex index (i, j) and the data value attached to that vertex. Also, in this chapter, we use an undirected binary tree instead of a directed binary tree, since it is better to represent multicast keying materials. For more results about tree, the reader is referring to [2].

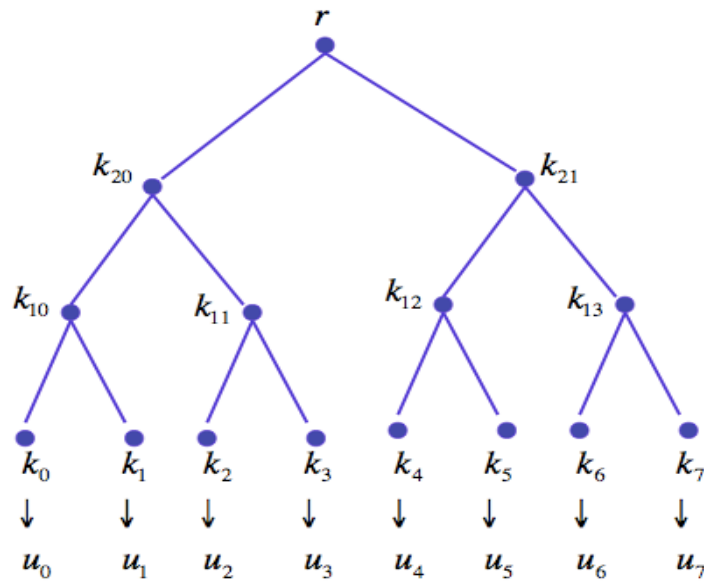


Figure 3: A Logic Key Tree with 8 Entities

2.2 Tree topology based multicast key distribution protocol

This protocol consists of three different protocols, one is for the initial set up for the assignment of the keys to a multicast group, the second and third are to handle the cases when an entity leaves or joins the network, respectively.

PROTOCOL II: TREE TOPOLOGY BASED MULTICAST KEY DISTRIBUTION

A. Initial Phase

PROTOCOL 1 (INITIAL PHASE)

KMD server: Perform the following operations.

1. Generation of a complete tree with height $h = \lceil \log n \rceil$: KMD server generates r and

k_{ij} 's from a PRNG, attaches r to the root, called a *root key or group key*, and k_{ij} to a vertex with index (i, j) for all the rest of parent vertexes in the tree, called them the *parent keys or vertex keys or subgroup keys* depending on the context, the leaves will hold their individual keys, established later, and each entity is attached to the leaf with its individual key. This tree is referred to as a *logic key tree (LKT)*. (Note that we can always form a logic key tree from n entities in which the depths of all the leaves are either h or $h - 1$, i.e., a complete tree. In the following process, we present the case of a leaf with depth h for simplicity.)

2. Establish individual keys and distribution of group key and subgroup keys:
 - (a) For each i , if the individual keys are not preshared, then KMD server and the entity u_i run the MAKE protocol to establish individual key k_i .
 - (b) Then the KMD server finds a path from vertex u_i to the root. Let $K(u_i)$ be the set consisting of all vertex keys along the path, which we may represent it as

$$K(u_i) = \{k_i, k_{1,j_1}, k_{2,j_2}, \dots, k_{h-1,j_{h-1}}, r\}.$$

The KMD server assigns $K(u_i)$ to u_i by sending the following ciphertexts to u_i :

$$C_i = \{E_{k_i}(k_{1,j_1}), E_{k_{1,j_1}}(k_{2,j_2}), \dots, E_{k_{h-1,j_{h-1}}}(r)\}. \quad (1)$$

In other words, each ciphertext is obtained by encrypting a parent key using the corresponding child key in the path.

Each entity: Perform the following operations.

1. Entity u_i uses its individual key k_i , shared with the KMD server, to decrypt the first ciphertext to obtain k_{1,j_1} , then to decrypt the second cipher text to obtain k_{2,j_2} using the key k_{1,j_1} , and so on, until r is obtained.
2. The entity u_i stores his key set $K(u_i)$.

In the following, we use an example to explain the above initial phase.

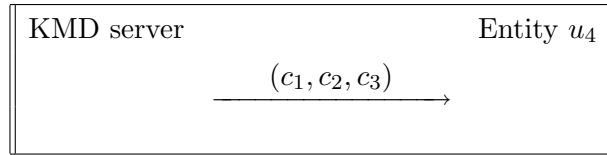
Example 1 Let $n = 8$, and KMD server generates a logic key tree, which is shown in Figure 3. A path from the vertex k_4 to the root r consists of four vertexes: $k_1, k_{1,2}, k_{2,1}$, and r . Thus the key set of entity u_4 is given by

$$K(u_4) = \{k_4, k_{1,2}, k_{2,1}, r\},$$

which is shown in Figure 4 where the vertexes are circled. KMD server prepares the following ciphertexts:

$$\boxed{\begin{array}{l} c_1 = E_{k_4}(k_{1,2}) \\ c_2 = E_{k_{1,2}}(k_{2,1}) \\ c_3 = E_{k_{2,1}}(r) \end{array}}$$

and sends (c_1, c_2, c_3) to the entity u_4 , i.e.,



Upon receiving (c_1, c_2, c_3) , the entity u_4 performs the following successive decryption using his individual key k_4 .

$k_{1,2} = D_{k_4}(c_1)$
$k_{2,1} = D_{k_{1,2}}(c_2)$
$r = D_{k_{2,1}}(c_3)$

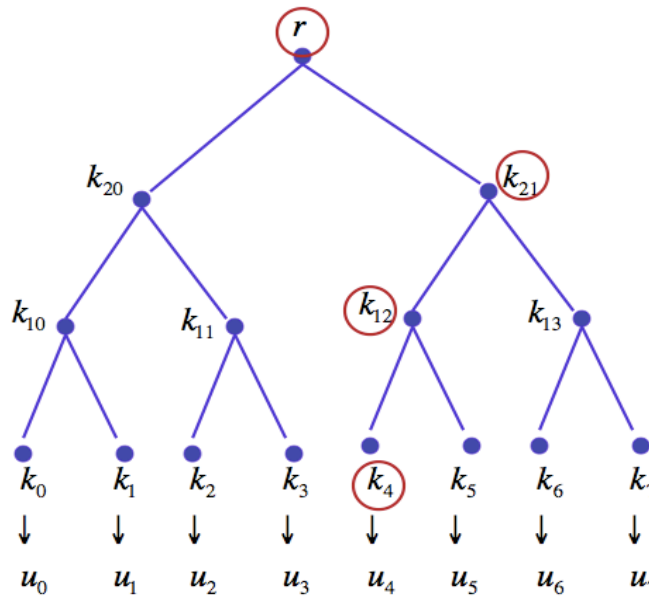


Figure 4: The key set of entity u_4 in LKT

Next we describe the rekeying processes when an entity joins/leaves the network.

B. Rekeying Process for Leave

In the following protocol, we assume that entity u_i leaves the network where the depth of u_i is h . (Usually, it may be less than or equal to h . If the depth of $u_i < h$, the process is similar as follows. We omit it for the case of leave. But we will present it in a general format for the case of join.)

PROTOCOL 2 (REKEYING PROCESS FOR LEAVE)

KMD server: Perform the following operations.

1. Find a path from vertex u_i to the root and retrieve the key set of u_i , $K(u_i)$, as shown below

$$K(u_i) = \{k_i, k_{1,j_1}, k_{2,j_2}, \dots, k_{h-1,j_{h-1}}, r\}.$$

All the keys in $K(u_i)$ need to be revoked including the individual key.

2. Locate the sibling of u_i , say u_t (note that $t = i$ or $t = i + 1$), which holds the same set of the parent keys as that of u_i . In other words, the key set of u_t is given by

$$K(u_t) = \{k_t, k_{1,j_1}, k_{2,j_2}, \dots, k_{h-1,j_{h-1}}, r\}.$$

When u_i leaves, KDM server attaches u_t to its parent vertex $(1, j_1)$, assigns the individual key k_t of u_t to the vertex, and removes the parent key k_{1,j_1} .

3. KMD server generates a set of new keys for updating the parent keys in $K(u_i)$ except for k_{1,j_1} since this vertex becomes a leaf vertex. We may assume that the updated $K(u_t)$ is given by

$$K'(u_t) = \{k_t, k'_{2,j_2}, \dots, k'_{h-1,j_{h-1}}, r'\}.$$

4. *Paired encryption and multicast transmission:* KMD server updates keys $k_{i,j}$ to $k'_{i,j}$ to the entities in the subtrees which possess the original keys $k_{i,j}$ using the keys of the siblings of the vertexes in the path of u_i to the root in the following fashion. KMD server encrypts the new parent key using the vertex's newly updated key and its sibling's key respectively, and multicasts this pair of the ciphertexts to the entities attached in their respective subtrees.

Entities: Decrypt to obtain these vertex keys, and update their key sets.

We use the LKT in Example 1 to illustrate the rekeying process.

Example 2 Assume that entity u_4 leaves the network.

The KMD server's updating process

1. As shown in Figure 4, a path from u_4 to the root gives the key set of u_4 is given as below

$$K(u_4) = \{k_4, k_{1,2}, k_{2,1}, r\}.$$

2. Find the sibling of u_4 , which is u_5 . KDM server attaches u_5 to its parent vertex $(1, 2)$, assigns the individual key k_5 of u_5 to the vertex, and removes the parent key $k_{1,2}$.
3. KMD server generates $\{k'_{2,1}, r'\}$ for updating the parent keys $\{k_{2,1}, r\}$ in $K(u_4)$ except for $k_{1,2}$ since that vertex is changed to a leaf vertex.

4. Paired encryption and multicast subsets:

- (a) Encrypt the new parent key $k'_{2,1}$ using the individual key k_5 of u_5 and its sibling key $k_{1,3}$, i.e., compute $c_1 = E_{k_5}(k'_{2,1})$ and $c_2 = E_{k_{1,3}}(k'_{2,1})$, and multicast c_1 to $\{u_5\}$ and c_2 to $\{u_6, u_7\}$.
- (b) Find the sibling of the vertex $(2, 1)$, which is the vertex with index $(2, 0)$. Their parent is the root. Encrypt their new root key using the newly updated key of the vertex $(2, 1)$ and the key of its sibling $(2, 0)$, i.e., compute $c_3 = E_{k'_{2,1}}(r')$ and $c_4 = E_{k_{2,0}}(r')$, and multicast c_3 to $\{u_5, u_6, u_7\}$ and c_4 to $\{u_0, u_1, u_2, u_3\}$.

The ciphertexts and multicast subsets are shown in the following diagram.

Vertex and its sibling	Paired ciphertext	Multicast set
(1, 2)	$c_1 = E_{k_5}(k'_{2,1})$ →	$\{u_5\}$
(1, 3)	$c_2 = E_{k_{1,3}}(k'_{2,1})$ →	$\{u_6, u_7\}$
(2, 1)	$c_3 = E_{k'_{2,1}}(r')$ →	$\{u_5, u_6, u_7\}$
(2, 0)	$c_4 = E_{k_{2,0}}(r')$ →	$\{u_0, u_1, u_2, u_3\}$

The siblings of the vertexes in the path from u_4 to the root are shown in Figure 5 in framed boxes. The vertex keys, framed by circles and boxes in Figure 5, are the keys involved in the rekeying process when entity u_4 leaves the network.

Decryption at the entities' side: Upon receiving those ciphertexts, the entities are able to update their vertex keys. For example, entities u_6 and u_1 will perform the following respective operations.

Entity u_6 's updating process:

1. The key set of u_6 is given by (see Figure 5)

$$K(u_6) = \{k_6, k_{13}, k_{21}, r\}.$$

2. Confirm that k_{21} and r need to be updated.
3. From the received two messages: c_2 and c_3 , compute

$$\begin{aligned} k'_{2,1} &= D_{k_{1,3}}(c_2) \\ r' &= D_{k'_{2,1}}(c_3) \end{aligned}$$

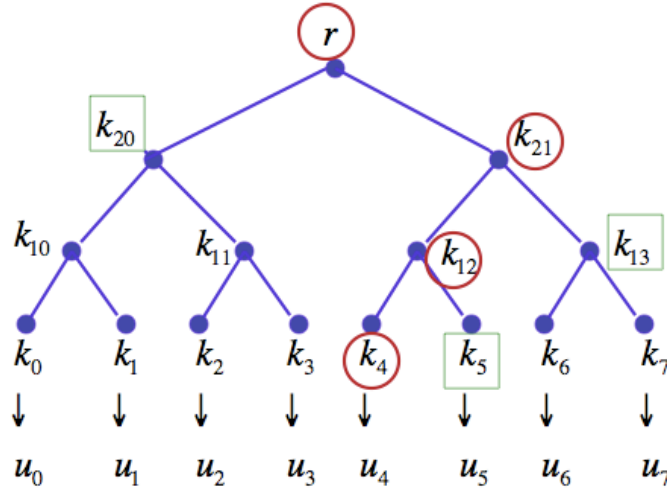


Figure 5: The vertex keys are involved in the rekeying process

The entity u_6 newly updated key set is now changed to:

$$K'(u_6) = \{k_6, k_{13}, k'_{21}, r'\}.$$

Entity u_1 's updating process:

1. The key set of u_1 is given by

$$K(u_1) = \{k_1, k_{10}, k_{20}, r\}.$$

2. So, the entity u_1 only needs to update the root key r .
3. From the received message $c_4 = E_{k_{2,0}}(r')$, u_1 decrypts it using his key k_{20} :

$$r' = D_{k_{2,0}}(c_4).$$

The updated key set now becomes that

$$K'(u_1) = \{k_1, k_{10}, k_{20}, r'\}.$$

After the rekeying process for u_4 leaving, the LKT is changed to a new tree, shown in Figure 6.

□

C. Rekeying Process for a Join

In the following protocol, we denote the entity who joins the network as u_n .

PROTOCOL 3 (REKEYING FOR A JOIN)

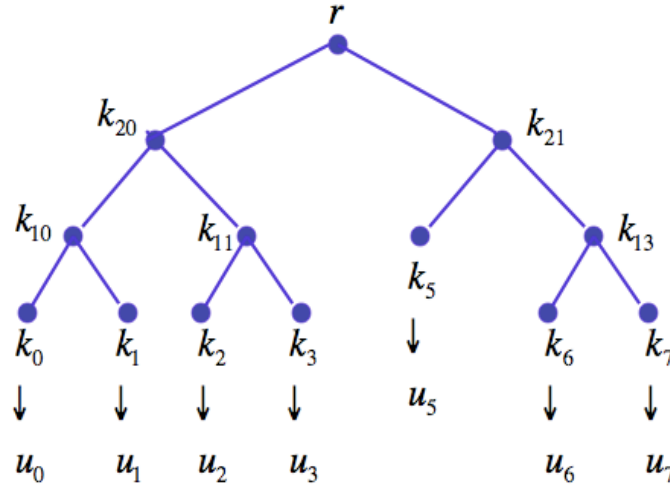


Figure 6: A new LKT after the rekeying process

1. This step is similar to the initial phase in Protocol 1. In the following, we assume that $n < 2^h$. The KMD servers performs the following operations.

(a) Run the MAKE protocol with entity u_n and establish an individual key of u_n , denoted as k_n . Find a leave vertex with the smallest depth, say h' (if there are more than one, pick the first one from left to right order), denoted this leave vertex as u_i . Make this vertex as a parent vertex of the two leaves to which the entities u_i and u_n are attached.

(b) The original key set of u_i is given by

$$K(u_i) = \{k_i, k_{s+1, j_{s+1}}, \dots, k_{h-1, j_{h-1}}, r\}, \text{ where } s = h - h'.$$

(c) Generate a key set for u_n from a PRNG, denoted as

$$K'(u_n) = \{k_n, k'_{s, j_s}, k'_{s+1, j_{s+1}}, \dots, k'_{h-1, j_{h-1}}, r'\}$$

where k'_{s, j_s} is a parent key of u_i and u_n .

(d) Encrypt k'_{s, j_s} using their individual keys of u_i and u_n and send to them, i.e., compute $E_{k_i}(k'_{s, j_s})$ and $E_{k_n}(k'_{s, j_s})$, and send the first ciphertext to u_i and the second ciphertext to u_n . The key set of u_i will be updated to

$$K'(u_i) = \{k_i, k'_{s, j_s}, k'_{s+1, j_{s+1}}, \dots, k'_{h-1, j_{h-1}}, r'\}.$$

2. Execute Step 4 in Protocol 2 in a way that u_i were treated as leaving the network for the following updating

$$\{k_{s+1, j_{s+1}}, \dots, k_{h-1, j_{h-1}}, r\} \xrightarrow{\text{updated to}} \{k'_{s+1, j_{s+1}}, \dots, k'_{h-1, j_{h-1}}, r'\} \quad (2)$$

where u_i is included in any subgroup of which u_n belongs. (Here we virtually assume that $K(u_n) = \{k_n, k'_{s,j_s}, 0 \dots, 0\}$.)

Example 3 Let $n = 5$ and the LKT is given by Figure 7. Here $h = 3$.

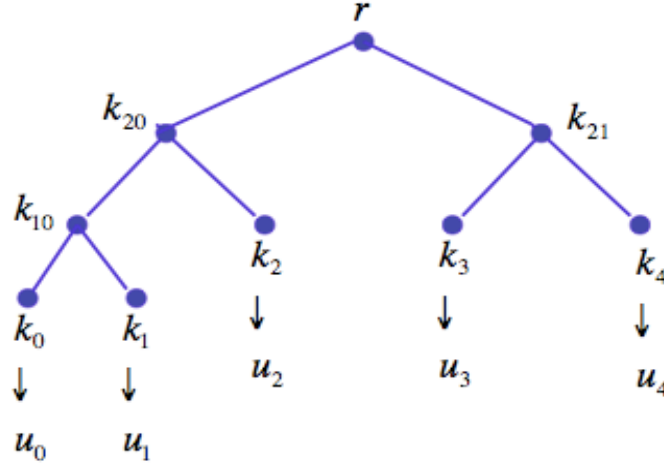


Figure 7: A LKT of 5 entities

1. KMD server performs the following operations.

- (a) A new entity is denoted as u_5 . From the LKT, u_2 is the first leaf with depth 2 from left to right order. Make the vertex where u_2 is attached as a parent vertex with index $(1,1)$, and attach u_2 and u_5 as its children.

- (b) The original key set of u_2 is given by

$$K(u_2) = \{k_2, k_{2,0}, r\}.$$

- (c) The KMD server generates a key set for the new entity u_5 , denoted as

$$K'(u_5) = \{u_5, k'_{1,1}, k'_{2,0}, r'\}$$

where $k'_{1,1}$ is a parent key of u_2 and the new entity u_5 .

- (d) Compute $c_1 = E_{k_2}(k'_{1,1})$ and $c_2 = E_{k_5}(k'_{1,1})$, and send c_1 to u_2 and c_2 to u_5 . The key set of u_2 needs to be updated to

$$K'(u_2) = \{u_2, k'_{1,1}, k'_{2,0}, r'\}.$$

2. The required updates are

$$\{k_{2,0}, r\} \longrightarrow \{k'_{2,0}, r'\}.$$

The KMD server runs Step 4 in Protocol 2 in a way that u_2 were treated as leaving the network for which the key set of u_5 is virtually defined as $K(u_5) = \{u_5, k'_{1,1}, 0, 0\}$ and u_2 is included in any subgroups of which u_5 belongs for the updates. The details of the updates are illustrated in the following diagram.

KMD	Entities
$c_3 = E_{k'_{1,1}}(k'_{2,0})$	$\{u_2, u_5\}$
$c_4 = E_{k_{1,0}}(k'_{2,0})$	$\{u_0, u_1\}$
$c_5 = E_{k'_{2,0}}(r')$	$\{u_0, u_1, u_2, u_5\}$
$c_6 = E_{k_{2,1}}(r')$	$\{u_3, u_4\}$

After the join of u_5 , the LKT becomes a new LKT, as shown in Figure 8.

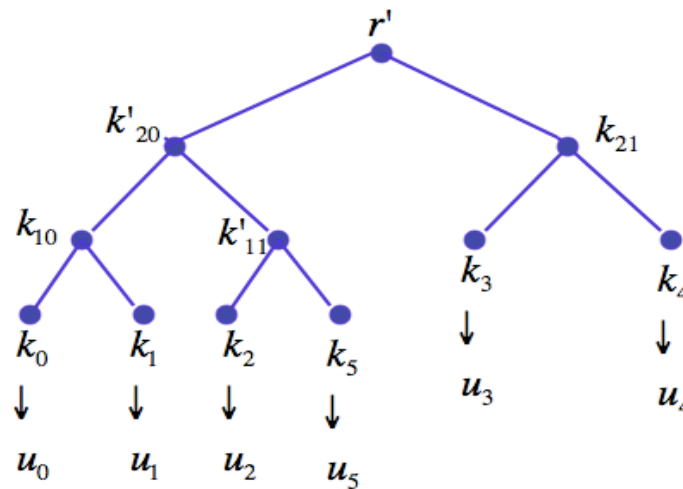


Figure 8: A new LKT after the join of u_5

Remark 1 For the case $n = 2^h$, the rekeying process for a join is similar as we described in Protocol 3 except that for this case the height of the tree is increased from h to $h + 1$. In this case, the leaf to which entity u_0 is attached will become a parent vertex with u_0 and u_n as its children.

Table 2: Comparisons of Protocols Θ and Π

Topology	Protocol	$R = K(u_i) $	N_{ini}	N_{leave}	N_{join}	Robust to compromising keys
Star	Θ	2	n	$n - 1$	$n + 1$	No
LKT	Π	$\leq \lceil \log n \rceil + 1$	$\leq \lceil \log n \rceil n$	$\leq 2\lceil \log n \rceil - 2$	$\leq 2\lceil \log n \rceil$	Yes

2.3 Performance Evaluation

In the following, we determine the parameters of the LKT based key distribution for multicast communications, presented in Protocols 1-3.

Property 1 *The parameter set of Protocol Π is given by $(n, hn, 2h - 2, 2h, h + 1)$ where $h = \lceil \log n \rceil$.*

Proof. From Protocol 1, we know that each entity holds $h + 1$ keys. So, we only need to determine the numbers of keys sent by the KMD server in the three protocols.

In Protocol 1, i.e., the initial phase of the multicast key distribution, an individual key between an entity and the KMD server is derived by executing the MAKE, so we do not include the communication cost from this step. After the individual keys are shared, the KMD prepares either h or $h - 1$ ciphertexts given by (1) for each entity depending on the depth of the leaf and sends them to the entity. So, the total number of ciphertexts is at most $N_{init} = hn$.

From Steps 3 and 4 of Protocol 2, there are at most $h - 1$ parent keys that need to be updated. For each parent key, it is encrypted by two different keys, i.e., two children's keys and multicast to them. Thus, the number of ciphertexts sent by the KMD server is at most $2(h - 1) = 2h - 2$.

For a join in Protocol 3, the encryptions occur in Step 1-(d) and Step 2. The former produces 2 ciphertexts, and the latter, from Protocol 2, the number of ciphertexts produced is less than or equal to $2(h - 1)$. Thus, the total number of the ciphertexts sent by the KMD server for a join is up bounded by $2 + 2(h - 1) = 2h$. \square

Remark 2 The assignment of the parent keys to each entity in Protocol 1, the initial phase, can be changed to use the strategy of Protocol 3. In this way, the number of the ciphertexts will be dramatically reduced from nh to less than or equal to $2n$. In this case, the parameters for Protocol Π becomes $(n, 2n, 2h - 2, 2h, h + 1)$

The comparisons of performance between Protocol Θ , the naive protocol presented in Section 1.2 and Protocol Π are given in Table 2.

Hence the rekeying process of the LKT based Protocol Π is much more efficient than that of the star based Protocol Θ . This is obtained by increasing the complexity of the initial process and the

number of keys held by each entity as cost. In Protocol II, each entity holds h (recall $h = \lceil \log n \rceil$) keys not including the root key. If attacker can manage to hurl some vertex key of an entity in some way, then the attacker has to have the ciphertexts along the path from this vertex to the root at hand in order to crash the root key, because the root key is hidden by h ciphertexts for each entity. Thus Protocol II is robust to the key compromising attack.

3 Hash Chain Based Authentication

In Sections 1 and 2 of this chapter, we have introduced the star or tree based multicast key distribution protocols. The root key shared by the KMD server and all the entities in the group is used for protected communications, i.e., encryption and authentication. However, if the purpose of a multicast group is to let all the other entities can authenticate any message sent by an entity in which confidentiality of the message does not need to be protected, then a protocol for achieving this goal without a multicast key is a more efficient approach. Another scenario is how to authenticate multiple messages broadcasted by a communication party efficiently.

Generally, this can be done by applying a digital signature scheme. In other words, a sender signs the message that he broadcasts or multicasts, and receivers are able to verify the digital signature of the sender. So the transmitted message is authenticated. However, usually, the verification of a digital signature is much more computational cost than the verification of an authentication tag generated by a symmetric key scheme, like a hash function. Precisely, the problem we like to tackle can be formalized as follows.

Problem.

- (a) A server A broadcasts n messages (x_1, x_2, \dots, x_n) at n different time instances. Design a scheme for which each receiver can efficiently authenticate x_i with modest computation cost and memory requirement.
- (b) Given n messages (x_1, x_2, \dots, x_n) generated by one entity or n entities where each entity responds to generate one message, design an algorithm by which a server A or system A can efficiently authenticate a randomly chosen x_i , but the system does not hold a copy of x_i .

In the rest of the section, we assume that entity A 's security association contains a hash function h , an *MAC* (message authentication code), digital signature scheme *Sig*, and related cryptographic schemes. We denote A 's identity as ID_A . We also assume that all these cryptographic primitive algorithms are secure. For example, hash function h is secure if it is collision resistant, i.e., it is computationally infeasible to find x and y such that their hash values are equal, i.e., $h(x) = h(y)$.

3.1 Hash chains

Entity A picks an initial value y , computes

$$k_i = h^{n-i}(y), i = 0, 1, \dots, n-1, \quad (3)$$

and the end point k_0 is either signed by A as $Sig_A(k_0)$ or k_0 is committed by A for which any intended verifier knows this commitment and accepts as an authenticated value. In the following, we use $Sig_A(k_0)$ to explain the approach. The vector

$$(Sig_A(k_0), k_0, k_1, \dots, k_{n-1}) \quad (4)$$

is called a *hash chain* with length n .

Property 2 *If hash function h is a one-way function or with collision resistance, then each key, say k_i , in the hash chain, defined by (3), is authenticated by the end point k_0 in the following fashion.*

1. A verifier checks whether the digital signature of A on k_0 is true. If so, continue the next step. Otherwise, declare a failure.
2. Authenticate k_i by checking the validity of $k_{i-1} = h(k_i)$ for $i = 1, \dots, n - 1$.

Proof. Since k_0 is signed by A and h is a one-way function or with collision resistance, thus the authenticity of k_0 is obtained by verifying the digital signature of A , the authentication of k_1 is established through $k_0 = h(k_1)$ by the authenticity of k_0 and one-wayness or collision resistance of h , k_2 is authenticated through $k_1 = h(k_2)$ by the authenticity of k_1 and h 's property, and so on. \square

Remark 3 If k_0 is not signed by system A , instead, A pre-loads a common key to all the entities who wish to authenticate this chain, then the authentication tag of k_0 is provided by a MAC, which is also implemented by h . In this case, the security of the system is determined by the one-wayness or collision resistance of hash function h , which is independent of the computational power of an attacker.

3.2 Hash Chain Based Message Authentication

We consider Problem (a) which we addressed at the beginning of the section. If an communication entity multicasts or broadcasts n messages, say x_1, \dots, x_n instead of one message, and we assume that there is no preshared key among the sender and multiple receivers. How a receiver can verify the authenticity of these n messages? In general, the sender could sign each message and each receiver should verify the digital signature for each message. Since a digital signature scheme is either a RSA signature, DSS and ECDSA, which all need heavy computational recourse. Using a hash chain is to verify one digital signature and $n - 1$ hash values instead of to verify n digital signatures for authentication n messages without requesting for a preshared multicast key between a sender and multiple receivers. This is presented as follows.

Entity A prepares a hash chain $\{k_i\}$ with length $n + 1$, i.e., we have $k_i = h^{n+1-i}(y), i = 0, 1, \dots, n$, as defined in (3) for authenticating n messages x_1, \dots, x_n .

1. The authentication tag of message x_i is $MAC(k_i, x_i)$ where k_i is served as a key for MAC .

- (a) At time instance t_0 , the entity sends $(k_0, Sig_A(k_0))$. A receiver verifies the digital signature of A on k_0 .
- (b) At time instance $t_1: t_0 < t_1$, the entity A sends $(x_1, MAC(k_1, x_1))$, and at time instance $t_1 + \Delta t$, A releases k_1 . Then a receiver first confirms whether

$$k_0 = h(k_1). \tag{5}$$

If this is true, the receiver verifies the authentication tag $MAC(k_1, x_1)$. If it is successful, then the receiver accepts x_1 and stores k_1 .

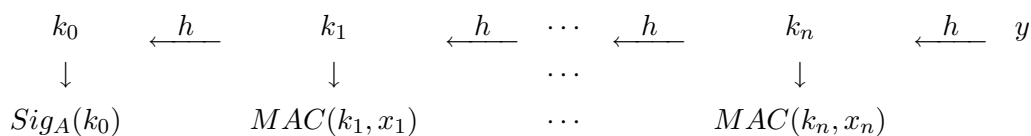
- (c) Assume that the receiver has successfully verified i messages and stored (k_0, k_{i-1}) . At time instances $t_i: t_0 < t_1 < \dots < t_i$, and $t_i + \Delta$ where $t_i > t_{i-1} + \Delta$, the entity A sends $(x_i, MAC(k_i, x_i))$ and releases k_i respectively. After receiving k_i , the receiver first confirms whether

$$k_{i-1} = h(k_i). \tag{6}$$

Since k_{i-1} has been authenticated, the validity of (6) results in an authentication of k_i . Then the receiver verifies the authentication tag $MAC(k_i, x_i)$.

2. Continue this process till finishing the authentication of message x_n . After used up all the values in the hash chain, the entity needs to produce a new hash chain for next n messages.

The process for computing an authentication tag for each message x_i is shown below.



The time differences for sending messages and releasing keys are shown in the following diagram.

t_0	t_1	$t_1 + \Delta$	t_2	$t_2 + \Delta$	\dots	t_n	$t_n + \Delta$
k_0	x_1		x_2		\dots	x_n	
$Sig_A(k_0)$	$MAC(k_1, x_1)$		$MAC(k_2, x_2)$			$MAC(k_n, x_n)$	
		k_1		k_2	\dots		k_n

The delayed authentication is to prevent the man-in-the-middle attack. If the key k_i for authenticating the i th message x_i is released at the same time for sending x_i and $MAC(k_i, x_i)$, the authentication tag of x_i , the attacker could replace the message x_i by some other message, say x'_i and the attacker is able to produce $MAC(k_i, x'_i)$ since he possess key k_i . This scheme is secure under the perfect synchronization between the sender and the receivers.

3.3 Hash Chain Based Access Authentication

In Chapter 4, we have introduced to hide passwords by the access control protocol RADIUS. Here we show how to design a one-time password scheme using hash chains for access authentication. In fact, hash chains are first studied by Lamport [5] for authentication n passwords used in a one-time password fashion. We present this scheme as follows. So, this is a scenario of Problem (b).

In remotely accessed computer systems, a user identifies himself to the system by sending a secret password. There are three possible ways an attacker could learn the user's secret password and then impersonate him when interacting with the system:

- (1) By intercepting the user's communication with the system, e.g., eavesdropping on the line connecting the user's terminal with the system, or observing the execution of the password checking program.
- (2) By gaining access to the information stored inside the system, e.g., reading the system's password file.
- (3) By the user's inadvertent disclosure of his password, e.g., choosing an easily guessed password.

The third possibility cannot be prevented by any password protocol, since two individuals presenting the same password information cannot be distinguished by the system. Currently, the countermeasure to this threat is to use biomedical methods, like, voice recognition, fingerprint, etc.. A countermeasure for the first two attacks is to use a hash chain in the following way.

1. A user generates a hash chain, defined by (3), stores k_0 in the system (here k_0 is not signed, instead it is committed to the system), and uses k_i as his i th password, $i = 1, \dots, n - 1$. In other words, the user will use different password to remote identify himself to the system at different time.
2. At time t_1 , the user uses k_1 as his password, the system verifies whether $k_0 = h(k_1)$. Assume that the user has used the first $i - 1$ passwords and stored (k_0, k_{i-1}) . At time instance t_i : $t_0 < t_1 < \dots < t_i$, he sends k_i as his new password, and the previous used passwords are expired. Upon receiving it, the system verifies whether $k_{i-1} = h(k_i)$, and grant an access if it is successful.

Once all $n - 1$ passwords associated with the initial value y are used up, the user needs to generate a new hash chain, and submit the new end point value again to the system.

Now we explain how the first two attacks fail. This is a one-time password scheme, i.e., each password is only used once. To see that the scheme is secure against eavesdropping and tampering with the communication, suppose that knowing the first $t - 1$ passwords k_1, \dots, k_{t-1} , an attacker is able to find the next password k_t . Since $k_{t-1} = h(k_t)$, let $w = k_{t-1}$ and $z = k_t$, then $w = h(z)$. Thus given w , the attacker is able to compute z which contradicts the one-wayness or collision resistance of h . Since the password sequence is determined in advance, no amount of tampering

with the communication will allow an attacker to impersonate the user. On the other hand, if attacker can gain access to the system, since the stored latest passwords is no long used in the next session, and the hash function is one-way or collision resistance, the attacker cannot get any information about next password. Thus the second attack is defeated by this protocol.

Robustness of the scheme: If the system and the user are out of synchronicity due to some system errors, the user sending k_i and the system using k_t to authenticate it. If $i < t$, then this can be detected by repeatedly applying h to k_t until it reaches a match $k_i = h^s(k_t)$. For example if $t = i + 10$, then

$$h^{10}(k_t) = h^{10}(h^{n-t}(y)) = h^{10}(h^{n-(i+10)}(y)) = h^{n-i}(y) = y = k_i.$$

In this way, k_t can be authenticated and the system can accept the password k_t . If $i > t$, then the system needs to request a later value. In this way, the user can still be granted an access.

4 Merkle Trees for Authentication

In last subsection, we have explained that a special scenario of Problem (b) in Section 3 of this chapter, i.e., one-time password scenario, which can be solved in terms of a hash chain. In this section, we look at a solution to a general scenario of Problem (b), which is provided by application of Merkle trees. Recall the concepts about graph theory. A Merkle tree is a complete binary tree where each parent vertex is attached by a hash value with inputs from two children.

Let B be a set consisting of a group of users, which will be worked in a cooperated way. The n data $\{x_0, x_1, \dots, x_{n-1}\}$ are generated by B , and they will be authenticated by A .

Generation of Merkle Tree:

1. The party B generates a complete tree with height $h = \lceil \log n \rceil$. The n leaves are ordered from left to right, and hash value $h(x_i)$ is attached to vertex leaf i .
2. Each parent vertex is attached with a hash value where an input to h is the concatenation of the hash values from two children vertexes. In other words, let a parent vertex have index (i, j) , denote an attached value as $a_{i,j}$, and the two children of $a_{i,j}$ as v_{left} and v_{right} . Then

$$a_{i,j} = h(v_{left} || v_{right}) \tag{7}$$

where $x || y$ means the concatenation of x and y . (Here we also abuse the notation for a vertex or a value that is attached to the vertex for simplicity.) This tree is referred to as a *Merkle tree*.

3. B submits the root value r to server A by either signing r or by a protected way.

System A authenticates any randomly chosen x_i :

1. Find a path from the leaf x_i to the root. Let $Auth(x_i)$ be the set consisting of the vertex values in the path. We may assume that

$$Auth(x_i) = \{a_i, a_{1,j_1}, \dots, a_{h-1,j_{h-1}}, r\}. \quad (8)$$

Let $Sib(x_i)$ denote the set consisting of all siblings of the vertexes in $Auth(x_i)$ whose elements are given by

$$Sib(x_i) = \{a_t, a_{1,l_1}, \dots, a_{h-1,l_{h-1}}\} \quad (9)$$

where a_t is the sibling of a_i so $t = i - 1$ or $t = i + 1$, a_{1,l_k} is the sibling of a_{1,j_k} so, $l_k = j_k - 1$ or $l_k = j_k + 1$ for $k = 1, \dots, h - 1$.

2. A requests B to submit all the sibling values given by (9). After receiving those values, A iteratively computes the following hash chain:

$$\begin{aligned} a_i &= h(x_i) \\ a_{1,j_1} &= h(a_i || a_t) \\ a_{2,j_2} &= h(a_{1,j_1} || a_{1,l_1}) \\ &\vdots \\ d &= h(a_{h-1,j_{h-1}} || a_{h-1,l_{h-1}}) \end{aligned}$$

Check whether $d = r$. If so, accept message x_i is from the party B .

Note. The difference between a logic key tree and a Merkle tree is that the value attached to a vertex is a random number in a logic key tree which will be serves as a key, and it is a hash value obtained from its two children in a Merkle tree.

Example 4 Let $n = 8$, then $h = 3$. For messages $\{x_0, x_1, \dots, x_7\}$, we have a Merkle tree, as shown in Figure 9. The vertex values in the Merkle tree are given in Table 3.

The group B submits the root value r to the server A . When B sends x_2 to A , A first determines $Auth(x_1)$, the set consisting of the vertex in the path from x_1 to the root, i.e.,

$$Auth(x_1) = \{a_1, a_{1,0}, a_{2,0}, r\}.$$

Then A requests $Sib(x_1)$, the set consisting of all siblings in $Auth(x_1)$:

$$Sib(x_1) = \{a_0, a_{1,1}, a_{2,1}\}.$$

In Figure 9, the vertexes circled are of $Auth(x_1)$ and vertexes boxed are their respective siblings. Upon receiving $Sib(x_1)$, system A performs that following iterative computations

$$a_{1,0} = h(a_0 || a_1) \quad (10)$$

$$a_{2,0} = h(a_{1,0} || a_{1,1}) \quad (11)$$

$$d = h(a_{2,0} || a_{2,1}). \quad (12)$$

Table 3: Hash Values in A Merkle Tree with 8 Leaves

$a_i = h(x_i), i = 0, \dots, 7$
$a_{1,0} = h(a_0 a_1) = h(h(x_0) h(x_1))$
$a_{1,1} = h(a_2 a_3) = h(h(x_2) h(x_3))$
$a_{1,2} = h(a_4 a_5) = h(h(x_4) h(x_5))$
$a_{1,3} = h(a_6 a_7) = h(h(x_6) h(x_7))$
$a_{2,0} = h(a_{1,0} a_{1,1}) = h(h(h(x_0) h(x_1)) h(h(x_2) h(x_3)))$
$a_{2,1} = h(a_{1,2} a_{1,3}) = h(h(h(x_4) h(x_5)) h(h(x_6) h(x_7)))$
$r = h(a_{2,0} a_{2,1})$ $= h(h(h(h(x_0) h(x_1)) h(h(x_2) h(x_3))) h(h(h(x_4) h(x_5)) h(h(x_6) h(x_7))))$

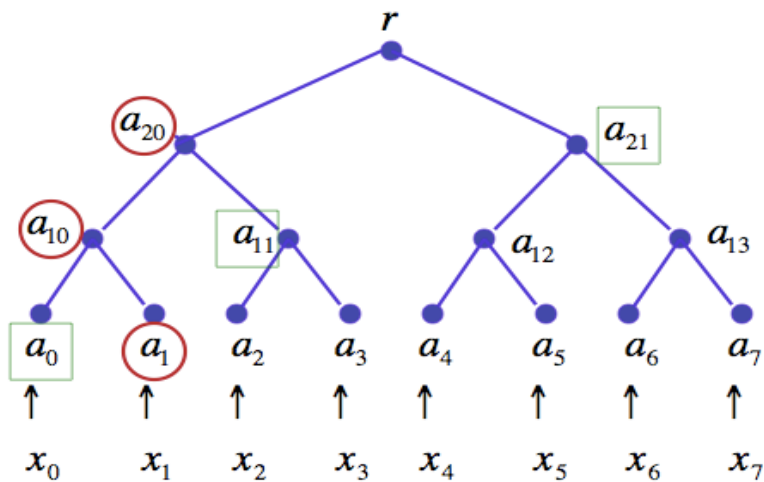


Figure 9: A Merkle Tree of 8 Messages

Then server A verify whether $d = r$. If so, the authenticity of x_1 is confirmed. The validity of the verification is established by the hash chain given by (10)-(12) as follows. If $d = r$, then d is authenticated, because the authenticity of r . From (12), $a_{2,0}$ and $a_{2,1}$ are authenticated by the authentication of d ; from (11), in terms of the authenticity of $a_{2,0}$, $a_{1,0}$ and $a_{1,1}$ are authenticated; from (10), a_0 and a_1 are authenticated by the authenticity of $a_{1,0}$. Since $a_1 = h(x_1)$, thus x_1 is authenticated.

Notes

The concept for multicast key distribution using logic key trees are independently introduced in [9] and [10] in 1999. After that, this approach is investigated or extended to a number of broadcast or multicast encryption scenarios [1], [8] [4]. Hash chains are first proposed by Lamport in 1981 for one-time password authentication [5]. It is used for the delayed authentication by Perrig *et al.* [7] in 2000, which is known as TESLA scheme. There are a variety of variants of TESLA based authentications for wireless systems after that. Merkle tree based authentication appeared in Merkle's Ph. D thesis in 1979 [6], which inspired researchers for investigating the problems on finding authentication paths efficiently [3] for about 3 decades (this is the same problem for finding a path for the rekeying process in logic key tree based multicast key distribution).

Exercises

1. In a multicast communication system, there is one key distribution and management (KDM) server and seven entities in the system at the initial phase of the system. The KDM server employs the logic key tree (LKT) based multicast key distribution scheme. We denote those seven entities as $\{u_0, u_1, \dots, u_6\}$ where $n = 7$.
 - (a) Construct an LKT for this group, and determine the key set for each entity (attach the entities to the leaves in the order from left to right with numbering 0, 1, \dots , 6).
 - (b) In the initial phase, what are the ciphertexts transmitted to u_6 ?
 - (c) Demonstrate the procedure that u_6 retrieves its key set.
2. With the assumption in Exercise 1, we assume that u_0 leaves.
 - (a) Show how the rekeying process in Protocol 2 in Section 2 works.
 - (b) How many ciphertexts are received by u_1 ? Which keys of u_1 and u_6 must be updated respectively? Why?
 - (c) Assume that u_0 leaves at time instance a , and the KDM does not execute the rekey protocol until the time instance b . However, at time $t : a < t < b$, the KDM multicasts a message which is encrypted by the multicast key r to the group. What happens in this case? Comment your result.

- (d) After the rekeying process, show the new LKT (you may give a sketch for this LKT with updated keys).
 - (e) After the rekeying process for u_0 's leaving, there is one new user who will join the system. Determine an LKT resulting from the rekeying process for a join.
3. Design a hash chain with length 101 for use in the hash chain based message authentication for authenticating 100 messages.
- (a) Explain why a key used in the generation of an authentication tag of i th message cannot be transmitted at the same time instance when the message and its authentication tag are transmitted.
 - (b) Can you give some examples of communication networks for which the synchronization between sender and receivers are easily achieved?
4. Design a one-time password scheme for which a preshared password between a system and a user can generate 1000 one-time passwords.
- (a) What is the 900th password used by the user?
 - (b) *Investigation of robustness of the scheme:* Due to some hardware errors, the system and the user are out of synchronization. The user sends the 900th password, i.e., k_{900} , the system uses k_{910} to do verification. Can the system successfully authenticate the user? Justify your answer.
5. Given 9 messages: $\{x_0, x_1, \dots, x_8\}$, using a Merkle tree design a scheme by which a system A can efficiently authenticates a randomly chosen x_i , but the system does not hold a copy of x_i .
- (a) Show a Merkle tree of your design.
 - (b) Determine the authentication path of x_3 in your design.
 - (c) Provide the procedure that x_3 has being authenticated by A .

References

- [1] R. Canitte, J. Garay, G. Itkis, D. Micciancio, M. Naor, and G. Pinas, Multicast security: a taxonomy and some efficient constructions, *Proceedings of the IEEE INFOCOM*, vol. 2, New York, March 1999, pp. 708716.
- [2] R. Diestel. *Graph Theory*, Springer-Verlag, Heidelberg, Graduate Texts in Mathematics, Volume 173, 3rd Edition, 2005.
- [3] M Jakobsson, T. Leighton, S. Micali, and M. Szydlo, Fractal Merkle tree representation and traversal, *CT-RSA 2003, Lecture Notes on Computer Sciences*, vol. 2612, M. Joye (Ed.), Springer-verlag, pp. 314326, 2003.

- [4] S.Q. Jiang and G. Gong, Multi-service oriented broadcast encryption, *Australian Conference on Information Security and Privacy*, H.Wang *et al.* (Eds.), *Lecture Notes on Computer Sciences*, vol. 3108, Springer-verlag, pp. 1-12. 2004.
- [5] L. Lamport, Password authentication with insecure communication, *Communications of the ACM*, Volume 24, Issue 11, pp. 770-772, 1981.
- [6] R. Merkle, *Secrecy, Authentication, and Public Key Systems*, UMI Research Press, 1982. Also appears as a Stanford Ph.D. thesis in 1979.
- [7] A. Perrig, R. Canetti, J.D. Tygar, and Dawn Song, Efficient authentication and signing of multicast streams over lossy channels, *Proceedings of IEEE Symposium on Security and Privacy 2000*, 2000, pp. 56-73.
- [8] D. Micciancio and S. Panjwani, Optimal communication complexity of generic multicast key distribution, *EUROCRYPT 2004*, Cachin and Camenisch (Eds.), *Lecture Notes on Computer Sciences*, vol., 3027, Springer-verlag, pp. 153-170, 2004.
- [9] D. Wallner, E. Harder, and R. Agee, Key management for multicast: issues and architectures, *IETF, RFC 2627*, June 1999.
- [10] C.K. Wong, M. Gouda, and S.S. Lam, Secure group Communication using key graphs, *IEEE/ACM Trans. Networking*, vol. 8, no. 1, pp. 16-30, Feb. 2000.