KTR: an Efficient Key Management Scheme for Secure Data Access Control in Wireless Broadcast Services

Qijun Gu, Peng Liu, Wang-Chien Lee, and Chao-Hsien Chu

***Abstract*—Wireless broadcast is an effective approach to disseminate data to a number of users. To provide secure access to data in wireless broadcast services, symmetric key-based encryption is used to ensure that only users who own the valid keys can decrypt the data. Regarding various subscriptions, an efficient key management to distribute and change keys is in great demand for access control in broadcast services. In this paper, we propose an efficient key management scheme (namely KTR) to handle key distribution with regarding to complex subscription options and user activities. KTR has the following advantages. First, it supports all subscription activities in wireless broadcast services. Second, in KTR, a user only needs to hold one set of keys for all subscribed programs, instead of separate sets of keys for each program. Third, KTR identifies the minimum set of keys that must be changed to ensure broadcast security and minimize the rekey cost. Our simulations show that KTR can save about 45% of communication overhead in the broadcast channel and about 50% of decryption cost for each user, compared with logical key hierarchy based approaches.**

***Index Terms*—Wireless broadcast, key management, access control, key hierarchy, secure group communication, key distribution**

I. Introduction

With the ever growing popularity of smart mobile devices along with the rapid advent of wireless technology, there has been an increasing interest in wireless data services among both industrial and academic communities in recent years. Among various approaches, broadcast allows a very efficient usage of the scarce wireless bandwidth, because it allows simultaneous access by an arbitrary number of mobile clients [1]. Wireless data broadcast services have been available as commercial products for many years. In particular, the announcement of the MSN Direct Service (direct.msn.com) has further highlighted the industrial interest in and feasibility of utilizing broadcast for wireless data services.

A wireless data broadcast system consists of three components as depicted in Figure 1: (1) the broadcast server; (2) the mobile devices; and (3) the communication mechanism. The server broadcasts data on air. A user's mobile device receives the broadcast information, and filters the subscribed data according to user's queries and privileges. The specialty of the broadcast system is that (a) the server determines the schedule to broadcast all data on air, and (b) users' mobile devices listen to the broadcast channel but only retrieve data (filter data out) based on users' queries. The communication mechanism includes wireless broadcast channels and (optional) uplink channels. Broadcast channel is the main mechanism for data

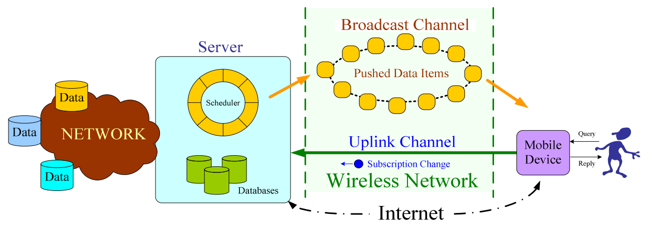


Fig. 1. A wireless data broadcast system

dissemination. Data is broadcast periodically so that users can recover lost or missed data items. The uplink channels, which have limited bandwidth, are reserved for occasional uses to dynamically change subscriptions.

In broadcast services, the basic data unit is **data item**, such as a piece of news or a stock price. Data items are grouped into **programs** and a user specifies which programs he would like to access. Typical programs could be weather, news, stock quotes, etc. For simplicity, we assume that each program covers a set of data items, and programs are exclusively complete. A user may subscribe to one or more programs. The set of subscribed programs is called the user's **subscription**. Users can subscribe via Internet or uplink channels to specify the programs that they are interested in receiving.

Previous studies on wireless data broadcast services have mainly focused on performance issues such as reducing data access latency and conserving battery power of mobile devices. Unfortunately, the critical security requirements of this type of broadcast service have not yet been addressed, i.e. broadcast service providers need to ensure backward and forward secrecy [2], [3] with respect to membership dynamics. In the wireless broadcast environment, any user can monitor the broadcast channel and record the broadcast data. If the data is not encrypted, the content is open to the public and anyone can access the data. In addition, a user may only subscribe to a few programs. If data in other programs are not encrypted, the user can obtain data beyond his subscription privilege. Hence, access control should be enforced via encrypting data in a proper way so that only subscribing users can access the broadcast data, and subscribing users can only access the data to which they subscribe.

Symmetric-key-based encryption is a natural choice for ensuring secure data dissemination and access. The broadcast data can be encrypted so that only those users who own valid keys can decrypt them. Thus, the decryption keys can be used as an effective means for access control in wireless data broadcast services. For example, each program has one unique key to encrypt the data items. The key is issued to the user who is authorized to receive and decrypt the data items. If a user subscribes to multiple programs, it needs an encryption key for each program. Since a user only has keys for his subscription, he cannot decrypt broadcast data and rekey messages designated to other users. At the same time, a data item can be decrypted by an arbitrary number of users who subscribe to it. This allows many users to receive the data at the same time and addresses the scalability problem, or request lost or missed keys.

Nevertheless, when a user subscribes/unsubscribes to a program, the encryption key needs to be changed to ensure that the user can only access the data in his subscription period. Consequently, a critical issue remains, i.e. *how can we efficiently manage keys when a user joins/leaves/changes the service without compromising security and interrupting the operations of other users?* Regarding unique features of broadcast services, we are interested in new key management schemes that can simultaneously provide *security*, *efficiency* and *flexibility*. A broadcast service generally provides many programs; at the same time, users may like to subscribe to an arbitrary set of programs. We envision that a user should be able to flexibly subscribe/unsubscribe to any program of interests and make changes to his subscription at any time. Hence, in addition to security and efficiency, **flexibility** that a user can customize his subscription at anytime is an indispensable feature of key management in broadcast services to support user subscriptions.

Two categories of key management schemes in the literature may be applied in broadcast services: (1) logic key hierarchy (LKH) based techniques [2]–[9] proposed for multicast services ; and (2) broadcast encryption techniques [10]–[16] in current broadcast services (such as satellite TV). We notice that current broadcast encryption techniques, including BISS [17], Digicipher [18], Irdeto [19], Nagravision [20], Viaccess [21], and VideoGuard [22], cannot in fact support flexibility. They normally require users to possess decryption boxes to receive the subscribed programs, and the broadcast services can only provide to users a few packages, each of which includes a fixed set of programs (TV channels). Users cannot select individual programs within a package. If a user wants to change his subscription, the user needs to request another decryption box that can decrypt the subscribed programs. Hence, in this paper, we will focus on adapting more flexible LKH-based techniques.

Nevertheless, directly applying LKH in broadcast services is not the most efficient approach. In broadcast services, a program is equivalent to a multicast group, and users who subscribe to one program form a group. Intuitively, we could manage a separate set of keys for each program, and ask a user to hold m sets of keys for his subscribed m programs. This straightforward approach is inefficient for users subscribing to many programs. If users could use the same set of keys for multiple programs, there would be fewer requirements for users to handle keys. Furthermore, when a user changes subscription, we argue that it is unnecessary to change keys for the programs to which the user is still subscribing, as long as security can be ensured. In this way, rekey cost can be reduced and fewer users will be affected. Therefore, we propose a new key management scheme, namely key tree reuse (**KTR**), based on two important observations: (1) users who subscribe to multiple programs can be captured by a shared key tree, and (2) old keys can be reused to save rekey cost without compromising security. KTR has two components: **shared key tree** and **shared key management**, and its contribution includes the following aspects.

Contributions. First, the proposed scheme takes advantage of a fact in broadcast services: many users subscribe to multiple programs simultaneously. In other words, programs overlap with each other in terms of users. Because existing approaches manage keys by separating programs, they turn to be demanding for the users who subscribe to many programs. Hence, this study contributes to the literature a new scheme (namely KTR) to better support subscriptions of multiple programs by exploiting the overlapping among programs. KTR let multiple programs share the same set of keys for the users who subscribe to these programs. KTR thus inherently enables users to handle fewer keys and reduces the demands of storage and processing power on resource-limited mobile devices.

Second, since multiple programs are allowed to share the same set of keys, a critical issue is how to manage shared keys efficiently and securely. We find that when keys need to be distributed to a user, it is unnecessary to change all of them. In many circumstances, when a user subscribes to new programs or unsubscribes to some programs, a large portion of keys that the user will hold in his new subscription can be reused without compromising security. KTR is a novel approach for determining which keys need to be changed and for finding the minimum number of keys that must be changed. Hence, KTR efficiently handles the rekey of the shared keys and minimizes the rekey costs associated with possible subscriptions. Our simulations show that critical keys can be employed in logical key hierarchy schemes [2], [5] to improve their performance.

The rest of the paper is organized as follows. In Section II, we present related works on group key management. In Section III, the first component of KTR is described that fully utilizes the service structure to reduce the number of keys. In Section IV, the second component of KTR is presented to reduce rekey cost when updating and distributing shared keys. In Section V, we present the results of simulations to illustrate the performance improvements in KTR. Finally, we conclude in Section VI.

II. Related Works on Key Management

A. Logical Key Hierarchy

Secure key management for wireless broadcast is closely related to secure group key management in networking [4]. Logical key hierarchy (LKH) is proposed in [2], [5] that uses a key tree (depicted in Figure 2) for each group of users who subscribe the same program. The root (top node) of the tree is the data encryption key (DEK) of the program. Each leaf (bottom node) in the tree represents an individual key (IDK) of a user that is only shared between the system and the user. Other keys in the tree, namely key distribution keys (KDKs),

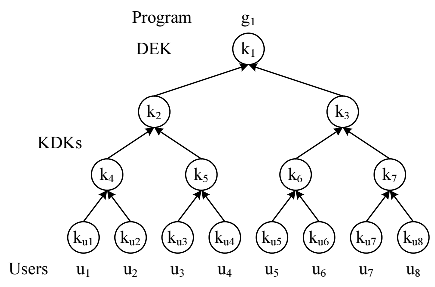


Fig. 2. Logical key hierarchy

are used to encrypt new DEKs and KDKs. A user only knows the keys along the path from the leaf of the user to the root of the key tree.

When a user joins or leaves the group, the server needs to change and broadcast the corresponding new keys, and this operation is called *rekey*, and the broadcast message of new keys is called *rekey message*. In our system, data and rekey messages are broadcast in the same broadcast channel to the users. Assume user u_1 leaves the group (in Figure 2). The server needs to change k_4, k_2 and k_1 so that u_1 will no longer decrypt any data for this group, which is encrypted by $k1$. The rekey message is

$$\{k_4'\}_{k_{u_2}}, \{k_2'\}_{k_4'}, \{k_2'\}_{k_5}, \{k_1'\}_{k_2'}, \{k_1'\}_{k_3}$$

where k_i' is the new key of k_i and $\{k_i'\}_{k_j}$ means k_i' is encrypted by k_j. When u_2 receives this message, u_2 first decrypts $\{k_4'\}_{k_{u_2}}$ based on his individual key k_{u_2} to obtain k_4', then uses k_4' to decrypt $\{k_2'\}_{k_4'}$ and so on to obtain k_2' and k_1'. Similarly, other users can obtain the new keys in their own paths. It is obvious u_1 cannot obtain any new keys from this message, and thus the broadcast data in the future will not be decrypted by u_1.

Now assume u_1 joins the group, and the server needs to change k_4, k_2 and k_1 so that u_1 cannot use the old keys to decrypt old broadcast data. Note that u_1 may have already eavesdropped on some broadcast data before he joined the group. If the server gives u_1 the old keys, u_1 can decrypt the eavesdropped broadcast data. The rekey message is

$$\{k_4'\}_{k_{u_1}}, \{k_2'\}_{k_{u_1}}, \{k_1'\}_{k_{u_1}}, \{k_4'\}_{k_4}, \{k_2'\}_{k_2}, \{k_1'\}_{k_1}$$

The first three components are for u_1 to use his individual key to decrypt the new keys, and the last three are for all existing users to use their old keys to decrypt the new keys. In this way, u_1 will not obtain any old key.

LKH is an efficient and secure key management for multicast services in that each user only needs to hold $O(log_2(n))$ keys for the user's group, and the size of a rekey message is also $O(log_2(n))$, where n is the number of group users. It is also a flexible key management approach that allows a user to join and leave the multicast group at any time. Many variations of LKH have been proposed. Because LKH simply uses independent keys, researchers developed several other approaches [23], [24] that generate new keys by exploiting the relation between child and parent keys or the relation between old and new keys. Our scheme is complementary to these schemes, since our scheme mainly examines whether keys need be changed in stead of how keys are generated.

[6] proposes a combination of key tree and Diffie-Hellman key exchange to provide a simple and fault-tolerant key agreement for collaborative groups. [23] reduces the number of rekey messages, while [9], [25] improve the reliability of rekey management. Balanced and unbalanced key trees are discussed in [5] and [26]. Periodic group re-keying is studied in [7], [8] to reduce the rekey cost for groups with frequent joins and leaves. Issues on how to maintain a key tree and how to efficiently place encrypted keys in multicast rekey packets are studied in [8], [26]. Moreover, the performance of LKH is thoroughly studied [3], [8].

B. *Broadcast Encryption*

There are some other key management schemes in the literature for multicast and broadcast services. [10] used arbitrarily revealed key sequences to do scalable multicast key management without any overhead on joins/leaves. [11] proposed two schemes that insert an index head into packets for decryption. However, both of them require pre-planned subscription, which contradicts the fact that in pervasive computing and air data access a user may change subscriptions at any moment. In addition, [11] only supports a limited combination of programs. [13] proposed a scheme to yield maximal resilience against arbitrary coalitions of non-privileged users. However, the size (entropy) of its broadcast key message is large, at least $O(n)$ [12]. Zero-message scheme [14], [15] does not require the broadcast server to disseminate any message in order to generate a common key. But it is only resilient against coalitions of k non-privileged users, and requires every user to store $O(klog_2(k)log_2(n))$ keys. Naor *et al.* [16] proposed a stateless scheme to facilitate group members to obtain up-to-date session keys even if they miss some previous key distribution messages. Although this scheme is more efficient than LKH in rekey operations, it mainly handles revocation when a user stops subscription. It does not efficiently support joins, which are crucial in our system. Finally, [24], [27] proposed self-healing approaches for group members to recover the session keys by combining information from previous key distribution information.

Compared with LKH-based approaches, key management schemes in broadcast encryption are less flexible regarding possible subscriptions. Conforming to the current practice described in RFC2627 [2], we select binary trees to present our scheme. Note that our scheme does not require binary trees and can be applied in trees of other degrees.

III. Shared Key Structure

Directly applying LKH is not efficient in broadcast services. We use a shared key structure to address the key management. In the following, we describe how a shared key structure is applied and then raise the security and efficiency problems of this scheme. We then present a novel shared key management in Section IV that ensures security and minimizes rekey cost, and also address major issues when applying KTR in a broadcast server.

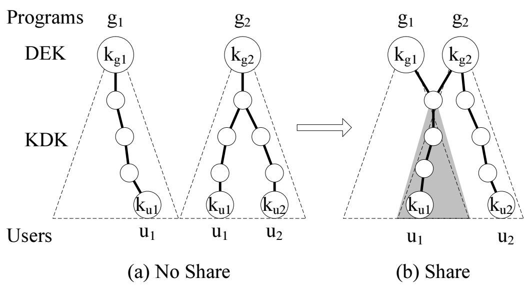


Fig. 3. Shared key tree

A. Key Forest

To address scalability and flexibility in key management, LKH is used as the basis of our scheme. An intuitive solution is to use a key tree for each program as shown in Figure 3(a). However, when user u_1 subscribes to two programs simultaneously, he needs to manage two sets of keys in both trees which is not very efficient (see Figure 3(a)). Hence, shared key tree (**SKT**) is proposed to reduce this cost in key management. As shown in Figure 3(b), we let the two programs share the same sub key tree as represented by the gray triangle. We regroup users so that users subscribing to both programs only need to manage keys in the gray triangle. The advantage of shared key tree is clear: any user subscribing to both g_1 and g_2 only needs to manage one set of keys for both programs. Moreover, when a user joins or leaves a tree shared by multiple programs, the encryption and communication cost for rekey operations can be significantly less than conventional LKH approaches.

In this study, shared keys are modeled as a key forest (see Figure 4), in which all keys form a directed and acyclic graph. The top keys in the forest are the DEKs of the programs. All other keys (KDKs) form trees. Users are placed in trees according to their subscriptions. A tree represents not only a unique subscription, but also a group of users having this subscription. Since a subscription is a set of programs, the root of the subscription's tree is connected to the DEKs of the programs belonging to the subscription. As keys in a tree are shared by the programs, a user only needs to handle the keys in the tree and the DEKs of the connected programs. For example, in Figure 4, since tr_4 represents a subscription of g_1 and g_2, its root k_{r_4} is connected to both k_{g_1} and k_{g_2}. The keys in tr_4 is shared by both g_1 and g_2. A user u_s in tr_4 subscribes g_1 and g_2 and needs to handle keys in the path from his leaf node to the DEKs of the subscribed programs: k_{u_s}, k_{n_L}, k_{r_4}, k_{g_1} and k_{g_2}. Finally, k_{g_1}, k_{g_2} and k_{g_3} are DEKs to encrypt broadcast data, and all other keys are KDKs.

In order to ensure that a user will not pay for subscribed programs multiple times, the key forest obviously should have the following properties, which are guaranteed in any directed and acyclic graph.

Property 3.1: Only one path exists by following the upward links from the root of a tree tr_s to the DEKs of the programs that share tr_s;

Property 3.2: Only one path exists by following the upward links from any leaf node in a tree to the root;

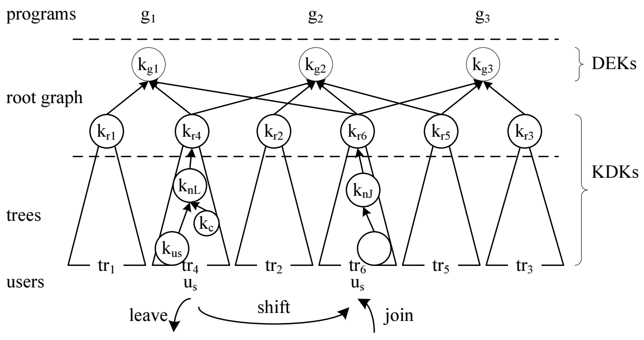


Fig. 4. Key forest

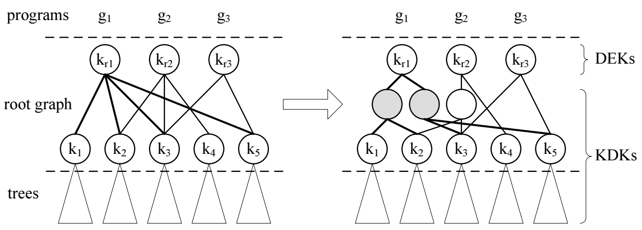


Fig. 5. Multi-layer root graph

Property 3.3: Each user belongs only to one tree in the key forest, and his individual key is the leaf node of the tree.

B. Root Graph

The root graph in Figure 4 depicts how programs share keys. Since m programs could generate $2^m - 1$ different subscriptions, such a two-layer structure in fact brings two major problems in terms of rekey overheads when the number of programs is large.

First, a program may be included in many subscriptions, which means the DEK of the program is connected with many trees. Assume the DEK is connected with n trees. When a user stops subscribing the program, the DEK needs to be updated and distributed to users in n trees. Because the new DEK is encrypted with the roots of the n trees in rekey, $O(n)$ rekey items are generated. Obviously, if n is large, a leave event results in a huge rekey message. For example, in Figure 5(a), 3 programs are included in 5 different subscriptions. Program g_1's DEK k_{g_1} is connected with 4 roots k_{r_1}, k_{r_2} k_{r_3} and k_{r_5}. Hence, when k_{g_1} is updated due to a leave event, 4 rekey items are needed.

To solve this problem, we use a multi-layer structure to connect the DEK with the roots of the shared trees. As in Figure 5(b), k_{g_1} is connected (bold lines) with k_{r_1}, k_{r_2} k_{r_3} and k_{r_5} via two intermediate key nodes (gray circles). Such a multi-layer structure inherently exploits the advantages of LKH. For a leave event, the number of rekey items in the root graph is reduced to $O(log_2(n))$. Note that, for different programs, the number of intermediate nodes and the number of layers may be different, which is obviously determined by the number of trees to which the program is connected. In Figure 5, program 2 is connected with 3 trees, and thus 1 node is needed. No node is needed for program 3, because it is only connected with 2 trees.

Second, a subscription is not a conventional plan that a broadcast service provides, because the subscribed programs of a plan normally cannot be changed by a user. In this paper, users are able to customize the selection of programs in their subscriptions. Thereby, a broadcast service could easily have a large number of different subscriptions. For example, even if a service provides only 30 programs that is a small number in many broadcast services, there could be $2^{30} = 1$ billion different subscriptions, which is much larger than the number of users. Hence, managing keys for all possible subscriptions would overload the server. Now, assume the service has n users and $2^m \gg n$. Although $2^m - 1$ different subscriptions exist, at most n subscriptions are valid, since the number of valid subscriptions cannot be more than the number of users. Hence, this problem can be easily solved by letting the server only manage the valid subscriptions that have at least one user.

Assuming n users are distributed to e trees ($e \leq n$), and each tree is shared by d programs on average ($d \geq 1$), then each tree would have $\frac{n}{e}$ users, and each program would have $\frac{ed}{m}$ trees. As discussed in Section V-A, KTR requires that the broadcast server manage $O((2\frac{n}{e} - 1)e + (\frac{ed}{m} - 1)m) = O(2n - e + ed - m)$ keys. In the worst case where $e = n$, the server needs to manage only $O(n(1 + d) - m)$ keys. In contrast, LKH requires the server to manage $O(2nd-m)$ keys. Obviously, KTR allows the server to manage much fewer keys than conventional LKH-based approaches.

C. Rekey Operations

In this study, we consider user activities of **joining/leaving/shifting** among trees, instead of joining/quitting/changing among programs. Table I lists the mapping between tree-oriented operations and the corresponding program-oriented user events. Consider the example in Figure 4, where a user u_s shifts from tr_4 to tr_6. When u_s was in tr_4, u_s subscribed g_1 and g_2. After he shifts to tr_6, he subscribes g_1, g_2 and g_3. Hence, the shift in fact means the user adds g_3 into his current subscription. Note that the discussion of rekey operations in this study only considers individual user events.

To issue new keys upon a user event, the main task is to identify the keys that need to be changed. We use two types of paths in the key forest to represent the to-be-changed keys. When a user leaves a tree, we say, a **leave path** is formed, which consists of keys that the user will no longer use. When a user joins a tree, we say, an **enroll path** is formed, which consists of keys that the user will use in the future. Similarly, when a user shifts from one tree to another, a leave path and an enroll path are formed. In KTR, a complete path starts from the leaf node and ends at the multiple DEKs of the subscribed programs that share the tree. For example, in Figure 4, when u_s shifts from tr_4 to tr_6, the leave path consists of k_{n_L} and k_{r_4}, and the enroll path consists of k_{n_J}, k_{r_6}, k_{g_1}, k_{g_2} and k_{g_3}. Note that in this example, k_{g_1} and k_{g_2} are the keys that u_s already has and still needs in the future. Hence, k_{g_1} and k_{g_2} are not in the leave path, although u_s leaves tr_4.

To broadcast new keys, the server should first compose rekey packets. In this study, we take the standard LKH approach to encrypt a new key k'_i in a **rekey item** $\{k'_i\}_{k_j}$. If k'_i is in an enroll path, k_j is the old k_i, i.e. $\{k'_i\}_{k_j} \equiv \{k'_i\}_{k_i}$. If k'_i is in a leave path, k_j is a child key of k'_i. Readers can refer to [2], [5] for examples of rekey packets.

TABLE I
REKEY OPERATIONS

Tree	Program oriented events
Join a tree	Assume a user has not subscribed to any program. • He subscribes to one or multiple programs.
Leave a tree	Assume a user has subscribed to several programs. • He unsubscribes to all current programs.
Shift among trees	Assume a user has subscribed to several programs. • He subscribes to one or a few more programs. • He unsubscribes to a part of the current programs. • He changes a part of the current programs.

Although LKH changes all keys in leave and enroll paths, KTR takes different rekey operations for leave path and enroll paths: keys in a leave path are changed as in LKH, while only a few keys in an enroll path are changed as long as security is ensured. In addition, KTR identifies an enroll path with the minimum number of must-be-changed keys to reduce rekey cost without compromising security. The detailed approach is presented in Section IV.

D. Challenges in Shared Key Management

If keys are shared by multiple programs, it is not always cost saving, especially when a user shifts to a new tree where some previously subscribed programs are still subscribed to by the user. Consider the example in Figure 4 where u_s shifts from tr_4 to tr_6. Apparently, g_1 and g_2 are still subscribed to after the shift. In general, two sets of keys need to be changed to ensure security: k_{n_L} and k_{r_4} in the leave path, and k_{n_J}, k_{r_6}, k_{g_1}, k_{g_2} and k_{g_3} in the enroll path.

When keys are not shared (i.e. each program has one individual key tree), fewer keys are involved in the example shift event. As in Figure 4, tr_4 is shared by g_1 and g_2, while tr_6 is shared by g_1, g_2 and g_3. The shift from tr_4 to tr_6 in fact indicates that u_s adds the program g_3 to his current subscription of g_1 and g_2. Hence, if keys are not shared and each program has one individual key tree, only keys in g_3's tree need to be changed. Thus, the shared key scheme has more rekey cost than conventional LKH in this example.

Nevertheless, because u_s in tr_6 is still subscribing to programs g_1 and g_2 as he was in tr_4, we find that keys in the enroll path (e.g. k_{n_J}, k_{r_6}, k_{g_1} and k_{g_2}) might be reused by u_s without compromising security. For example, no matter whether u_s was in tr_4 or shifts to tr_6, u_s always knows k_{g_1} or k_{g_2}. Hence, these two keys can be reused in this shift event. With an in-depth analysis, we find that under certain conditions k_{r_6} and k_{n_J} can be reused as well.

Deciding whether or not a key in an enroll path can be reused depends on whether or not the key can reveal the programs' DEKs that u_s is not supposed to know. Assume that in the example shift event, k_{r_6} has never been changed since t_0, u_s joins tr_4 at t_1 and shifts to tr_6 at t_2, and $t_0 < t_1 < t_2$.

There are at least two situations where k_{r_6} must be changed at t_2. Either k_{r_6} was used to encrypt k_{g_3} as $\{k_{g_3}\}_{k_{r_6}}$ during t_0 and t_2; or k_{r_6} was used to encrypt k_{g_1} or k_{g_2} as $\{k_{g_1}\}_{k_{r_6}}$ or $\{k_{g_2}\}_{k_{r_6}}$ during t_0 and t_1.

In the first situation, if k_{r_6} is reused, u_s can decrypt k_{g_3} from $\{k_{g_3}\}_{k_{r_6}}$, and then decrypt g_3's data that was broadcast before u_s shifts to tr_6 at t_2. Hence, the reused k_{r_6} reveals k_{g_3} before u_s adds g_3 into his subscription. In the second situation, if k_{r_6} is reused, u_s can decrypt k_{g_1} or k_{g_2} from $\{k_{g_1}\}_{k_{r_6}}$ or $\{k_{g_2}\}_{k_{r_6}}$, and then decrypt g_1's or g_2's data that was broadcast before u_s joins tr_4 at t_1. Hence, the reused k_{r_6} reveals k_{g_1} or k_{g_2} before u_s joins the service. In summary, k_{r_6} must be changed in both situations. If k_{r_6} is reused without any change, it can reveal DEKs that u_s should not know. Except these two situations, if k_{r_6} has never been used in the encryptions as discussed above, k_{r_6} can be reused without any change.

A similar but more complicated inspection is required on other keys in tr_6. The principle is to check whether a key in an enroll path can reveal the previous DEK or another program's DEK. The difficulty is that a key may indirectly reveal DEKs, because its parent key may not be a DEK. For example, although k_{n_J}'s parent key is k_{r_6}, k_{n_J} can still indirectly reveal the DEK k_{g_3}. Assume that k_{n_J} was used to encrypt k_{r_6} and then k_{r_6} was used to encrypt k_{g_3}. In this situation, a sequence of rekey items $\{k_{r_6}\}_{k_{n_J}}$,$\{k_{g_3}\}_{k_{r_6}}$ exist in all broadcast messages. If k_{n_J} is reused without any change, u_s can first decrypt k_{r_6} and then k_{g_3}. Thus, k_{n_J} must be changed. If no such sequence of rekey items exist, k_{n_J} can be reused. Hence, the challenge in reusing keys lies in how to find out which sequences of rekey items may compromise security and which keys in such sequences may reveal DEKs.

In the following section, we propose a novel approach in KTR to efficiently address the security issue in reusing keys. Since rekey cost is determined by the number of must-be-changed keys, the cost can be minimized if we can find the minimum number of must-be-changed keys when the user joins or shifts to the tree. We name the must-be-changed keys in an enroll path as **critical keys**. KTR changes all keys in a leave path and only the critical keys in an enroll path, while leaving all the other keys unchanged. In this way, the rekey cost can be minimized.

IV. Shared Key Management

In this section, we first present some important concepts in Section IV-A and IV-B, which are used for identifying critical keys . Then, we present the condition under which a key is critical in Section IV-C and IV-D and the corresponding key management algorithms.

A. Rekey Spots

KTR basically logs how a key was used in rekey messages. We can always find two operations in any rekey message: 1) a key's value is changed or 2) a key is used to encrypt its parent key when the parent key's value is changed. Accordingly, we define two types of spots to log the time points when either operation is committed.

Algorithm 1 Update of refresh and renew spots

```
Assume $k_i$ is used in the rekey messages upon a user event.
1: if $k_i$ is in a leave path then
2:     renew spots must be added to all $k_i$'s spot series;
3: end if
4: if $k_i$ is critical in an enroll path then
5:     renew spots must be added to all $k_i$'s spot series;
6: end if
7: if $k_i$'s parent key $k_j$ is in a leave path then
8:     refresh spots must be added to $k_i$'s spot series that are
       associated with the programs sharing $k_j$;
9: end if
```

Definition 4.1: Renew spot of a key k_i: the time point t when k_i's value is changed. k_i's new value starting from t is denoted as $k_i(t)$.

Definition 4.2: Refresh spot of a key k_i: the time point t when k_i is used to encrypt its parent key k_j's new value in a refreshment $\delta(k_j, t; k_i, t')$.

Definition 4.3: Refreshment, $\delta(k_j, t; k_i, t')$: a rekey message broadcast at t in the form of $\{k_j(t)\}_{k_i(t')}$, and $t' \leq t$.

At a refresh spot t, we say k_i is refreshed when it is used to encrypt its parent key k_i's new value as in the $\delta(k_j, t; k_i, t')$. At a renew spot t, we say k_i is renewed when its valued is changed. k_i is renewed only when it is in a leave path or critical in an enroll path. Accordingly, the rekey message to renew k_i has two possible forms. If k_i is critical in an enroll path, k_i's new value is encrypted with its old value. Hence, the renew message is $\{k_i(t)\}_{k_i(t')}$. If k_i is in a leave path, k_i's new value is encrypted with its child key k_c. Hence, the renew message of k_i is also the refresh message of k_c, i.e. $\{k_i(t)\}_{k_c(t')}$.

When a user changes his subscription, the server needs to change certain keys according to the algorithm presented in Section IV-C and broadcast corresponding rekey messages. Then, refresh and renew spots are logged to the keys that are used in the rekey messages. The sequence of refresh and renew spots thus forms spot series in the time order. If a key is shared by multiple programs, we let the key have multiple spot series, each of which is associated with one program (see examples in Section IV-B).

Algorithm 1 describes the procedure to log refresh and renew spots upon a user event. According to Definition 4.1, renew spots are logged to a key when the key is changed to a new value. Hence, when a key is in a leave path or critical in an enroll path, the key must be changed and renew spots must be logged to that key. Furthermore, according to Definition 4.2, refresh spots must be logged to a key when the key's parent is in a leave path, because the key is used to encrypt its parent in order to renew its parent.

As previously discussed, refreshments may contain information that threatens past confidentiality. For example, assume a user joins a tree shared by program g_m at t_c, and k_1 is in the enroll path. From all previous broadcast rekey messages, *a dangerous rekey sequence* for k_1 is defined as,

Definition 4.4: A sequence of refreshments

Algorithm 2 Update of revive spots

```
1:  let $k$ be $k_v$;
2:  let $t$ be $t_v$;
3:  UPDATEREVIVE($k, t, t_v, g_v$);
4:  function UPDATEREVIVE($k, t, t_v, g_v$)
5:      let $V$ be the set of all child keys of $k$;
6:      for $k_i \in V$ do
7:          find $t_i$ the renew spot of $k_i$ that satisfies $t_i \le t$;
8:          if $\delta(k, t; k_i, t_i)$ exists then
9:              if $t_i$ is the latest renew spot of $k_i$ then
10:                 add $t_v$ to $k_i$'s revive spot series associated with $g_v$
11:             end if
12:             UPDATEREVIVE($k_i, t_i, t_v, g_v$);
13:         end if
14:     end for
15: end function
```

$\delta(k_2, t_2; k_1, t_1), \delta(k_3, t_3; k_2, t_2), \cdots, \delta(k_m, t_m; k_{m-1}, t_{m-1})$ that satisfies

- k_i is k_{i-1}'s parent key,
- $t_1 \le t_2 \le \cdots \le t_m < t_c$,
- $k_1(t_1)$ has never been changed since t_1,
- $k_m(t_m)$ is the value of g_m's DEK.

If the server sends k_1 to the user without any change, the user can decrypt $k_2(t_2)$ and then iteratively decrypt $k_3(t_3)$ to $k_m(t_m)$. Past confidentiality at t_m is thus compromised, if the user uses $k_m(t_m)$ to decrypt the data items that are broadcast during t_m and t_c, which should otherwise be inaccessible to that user. Past confidentiality at t_m is preserved if the user either cannot find such a sequence of refreshments to obtain $k_m(t_m)$ or has already legitimately obtained $k_m(t_m)$ and the data items during t_m and t_c.

Therefore, to identify whether a key is critical in a program, we use a third type of spot to mark the key based on its renew and refresh spots. Similarly, the sequence of revive spots forms revive spot series in the time order. If a key is shared by multiple programs, the key has multiple revive spot series, each of which is associated with one program (see examples in Section IV-B).

Definition 4.5: Revive spot of a key: the time point t when (1) the DEK of this key's associated program has changed or (2) a dangerous rekey sequence exists as in Definition 4.4.

After the update of refresh and renew spots, the server updates revive spots according to Algorithm 2. Assume k_v (the DEK of program g_v) is renewed at t_v. Algorithm 2 updates the revive spots of corresponding keys iteratively, starting from k_v. At the beginning, let $k = k_v$ and $t = t_v$. The algorithm iteratively uses the function UPDATEREVIVE(k, t, t_v, g_v) to update revive spots of related keys. Assuming that the algorithm checks a key k which was renewed at t and was refreshed at sometime later than t, it first selects a k_i from all k's child keys. According to Definition 4.5, if $\delta(k, t; k_i, t_i)$ does not exist, there is no need to log a revive spot to k_i and no need to further check k_i's child keys. Also, if $\delta(k, t; k_i, t_i)$ does exist but t_i is not k_i's latest renew spot, there is still no need to log a revive spot to k_i but the algorithm continues

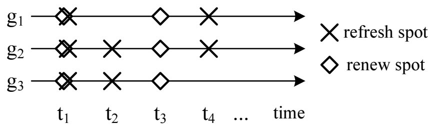


Fig. 6. Spot series of key k_{r_6}

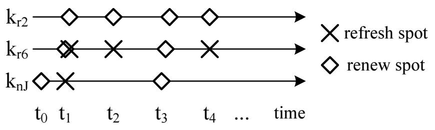


Fig. 7. Spot series regarding program g_2

to check k_i's child keys as they may have revive spots. If $\delta(k, t; k_i, t_i)$ does exist and t_i) is k_i's latest renew spot, a revive spot is logged to k_i and the algorithm continues to check k_i's child keys.

B. Examples of Spots

In this part, we demonstrate the spot series from two different dimensions. First, a key has multiple spot series associated with its programs. Figure 6 depicts the spot series of k_{r_6} in the key forest of Figure 4. Because k_{r_6} is shared by three programs (i.e. g_1, g_2 and g_3), it has three spot series, and each series is represented by a line in Figure 6. In this example, at t_1, a user leaves tr_6. k_{r_6} is first renewed, and all its spot series get a renew spot. Right after it is renewed, k_{r_6} is used in the refreshments $\delta(k_{r_1}, t_1; k_{r_6}, t_1), \delta(k_{r_2}, t_1; k_{r_6}, t_1), \delta(k_{r_3}, t_1; k_{r_6}, t_1)$. Because these refreshments are related to all the programs, refresh spots are added to all k_{r_6}'s spot series. At t_2, a user leaves tr_5. Because only k_{r_2} and k_{r_3} need to be changed, k_{r_6} is used in the refreshments $\delta(k_{r_2}, t_2; k_{r_6}, t_1), \delta(k_{r_3}, t_2; k_{r_6}, t_1)$. Hence, only two refresh spots are added to the series associated with g_2 and g_3. Readers can find that the other spots are for the events where a user joins tr_6 at t_3, and another user shifts from tr_4 to tr_3 at t_4. In brief, renew spots of a key are the same in all of its series, while refresh and revive spots are different regarding their corresponding programs.

The second dimension of spot series is illustrated in Figure 7, where we draw spot series of k_{n_J}, k_{r_6} and k_{r_2} associated with only one program g_2. At t_1, a user leaves tr_6. Assume k_{r_6} and k_{r_2} are in the leave path, but k_{n_J} is not. Hence, the broadcast server composes the refreshments $\delta(k_{r_6}, t_1; k_{n_J}, t_0), \delta(k_{r_2}, t_1; k_{r_6}, t_1)$ to change k_{r_6} and k_{r_2}. At t_2, a user leaves tr_5. The refreshment $\delta(k_{r_2}, t_2; k_{r_6}, t_2)$ is broadcast to change k_{r_2}. At t_3, a user joins tr_6. Assume k_{n_J}, k_{r_6} and k_{r_2} are in the enroll path, these keys are changed. At t_4, a user shifts from tr_4 to tr_3, and $\delta(k_{r_2}, t_4; k_{r_6}, t_4)$ is broadcast to change k_{r_2}. Note that at a refresh spot (for instance, t_1 of $\delta(k_{r_6}, t_1; k_{n_J}, t_0)$, the symmetric key k_{n_J} is refreshed simultaneously as its parent key k_{r_6} is renewed.

In Figure 8, we draw the revive spot series of the three keys associated with g_2 based on Figure 7. At t_2, k_{r_2} is changed and $\delta(k_{r_6}, t_1; k_{n_J}, t_0), \delta(k_{r_2}, t_2; k_{r_6}, t_1)$ can be recorded by any user. Revive spots are added to both k_{r_6} and k_{n_J}, since the

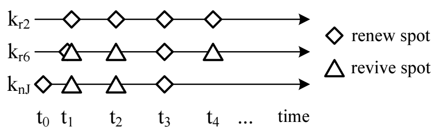


Fig. 8. Revive spots regarding program g_2

refreshments can expose k_{r_2} at t_2 to a new user if either k_{r_6} or k_{n_J} is given to the user without any change. However, at t_4, a revive spot is only added to k_{r_6}, because only $\delta(k_{r_2}, t_4; k_{r_6}, t_4)$ is potentially insecure. No revive spot is added to k_{n_J} at t_4, since it is not used in the rekey operation.

C. Critical Keys

By logging spots to keys, the server can inspect a key's past usage. We introduce the concepts of key age and subscription age to decide whether a key is a critical key.

Definition 4.6: Age of a key: (1) if the key is a DEK, its age is the time interval between the current time to its latest renew spot; (2) if the key is a KDK, its age is the time interval between the current time to the revive spot that is located between the current time and the latest renew spot and is closest to the latest renew spot. Note that a key may have multiple ages if it is shared by multiple programs, and each age is associated with one program.

According to the above definition, the age of a KDK is 0 if and only if there is no revive spot between the current time and the latest renew spot. Otherwise, the age of the key is greater than 0.

Definition 4.7: Age of a subscription: the time interval between the current time to the latest beginning time the user is in a program. Note that if a user subscribes to multiple programs, he has one subscription age for each program.

According to the above definition, the subscription's age is 0 if and only if the user is not in the program. Otherwise, the user is in the program, and his subscription age is greater than 0. If a user stops subscribing a program, the subscription age associated with this program turns to 0. If a user shifts from a tree to another tree while staying in a program, his subscription age with this program continues. Finally, a user can have different subscription ages for different programs.

The traditional LKH approach to protect past confidentiality is to always have the server change keys in the enroll path so that no previous refreshment can be exploited. Although this ensures past confidentiality, this approach is costly. There are some situations where the old refreshments only contain secrets that the user already knows. In these situations, the user can use the old symmetric key to decrypt the old refreshments, thus keeping past confidentiality intact. That being the case, the server can directly distribute the old symmetric key without compromising the past confidentiality.

In the following, we give a generic method to identify critical keys in the enroll path and reduce the rekey cost. Assuming key k is shared by m programs and will be distributed to user u, we can get all k's ages and all u's subscription ages associated with these programs, denoted as $[ka_1, ..., ka_m]_k$ and $[ua_1, ..., ua_m]_u$. Program g_i is thus associated with a pair of ages, denoted as $(ka_i, ua_i)_{k,u}$.

Theorem 4.1: Theorem of Critical Key (**TCK**): k in the enroll path is a **critical key**, i.e. k must be changed before being distributed to u to ensure past confidentiality, if and only if at least one pair of $(ka_i, ua_i)_{k,u}$ satisfies $ka_i > ua_i$ at current time t, i.e. the key is older than the user's subscription regarding program g_i.

a) Proof of the sufficient condition: If k is the DEK of g_i, the proof is obvious. If k's age is older than u's subscription age, there are some data items encrypted by k before the user joins the program. Hence, k needs to be changed so that the user cannot decrypt those data items.

If k is not a DEK, let k_i be the DEK of program g_i, and the latest renew spot of k is t_k. Assume a pair of $(ka_i, ua_i)_{k,u}$ that satisfies $ka_i > ua_i$ at current time t. According to Definition 4.6, k's value has never been changed since $t - ka_i$ and was revived at $t - ka_i$. According to Definition 4.5, u can find such a sequence of refreshments from all previous broadcast rekey messages at $t - ka_i$: $\delta(k_\alpha, t_\alpha; k, t_k), ..., \delta(k_i, t - ka_i; k_\beta, t_\beta)$, where $t_k \leq t_\alpha \leq ... \leq t_\beta \leq t - ka_i$. Hence, if k is sent to u without any change, u can derive k_i at $t - ka_i$ from these refreshments.

According to Definition 4.7, u joined g_i at $t - ua_i$, which means u is only allowed to decrypt data items of g_i broadcast after $t - ua_i$. Because $ka_i > ua_i$, $t - ka_i < t - ua_i$. If k is not changed, u can decrypt data items of g_i broadcast between $t - ka_i$ and $t - ua_i$, and thus past confidentiality at $t - ka_i$ is compromised.

Therefore, if at least one pair of $(ka_i, ua_i)_{k,u}$ satisfies $ka_i > ua_i$ at current time t, k in an enroll path needs to be changed before being distributed to user u to ensure past confidentiality.

b) Proof of the necessary condition: The necessary condition is that if all pairs of $(ka_i, ua_i)_{u,k}$ satisfy $ka_i \leq ua_i$, k does not need to be changed. If k is the DEK of g_i, the proof is obvious. If k's age is younger than user's subscription age, the user has already known all data items encrypted by k. Hence, k does not needs to be changed.

If k is not a DEK, select any program g_i that shares k. Let k_i be the DEK of g_i, and the latest renew spot of k is t_k. We use reduction to absurdity to prove. The opposite of the necessary condition is that past confidentiality for program g_i will be broken if all pairs of $(ka_i, ua_i)_{k,u}$ satisfy $ka_i \leq ua_i$ at current time t, and k is sent to u without any changed.

According to Definition 4.7, u joined g_i at $t - ua_i$ and is allowed to decrypt data items of g_i broadcast after $t - ua_i$. If past confidentiality for program g_i is compromised, u must have derived k_i's value before $t - ua_i$. Because $ka_i \leq ua_i$, $t - ka_i \geq t - ua_i$. u must have derived k_i's value at a time point t' before $t - ka_i$, i.e. $t' < t - ka_i$. Hence, u must have found the refreshments from all previous broadcast rekey messages: $\delta(k_\alpha, t_\alpha; k, t_k), ..., \delta(k_i, t'; k_\beta, t_\beta)$.

According to Definition 4.5, t' is a revive spot of k. If $ka_i = 0$, no revive spot exists after t_k, and thus t' cannot exists. If $ka_i > 0$, t' is a revive spot after k's latest renew spot t_k, and thus $t_k < t' < t - ka_i$. However, according to Definition 4.6, there cannot be any revive spot of k between t_k and $t - ka_i$. Hence, t' cannot exist, and the opposite of the necessary condition is false. Consequently, past confidentiality for any program g_i will not be broken if the pair of $(ka_i, ua_i)_{k,u}$

Algorithm 3 Algorithm of KTR in Broadcast Server

```
1: if a join or shift event happens then
2:     according to TCK, find all critical keys in the tree the user wants to join or shift to;
3:     select the best enroll path that has the minimum number of critical keys;
4:     change all critical keys in the best enroll path, and broadcast corresponding rekey messages;
5: end if
6: if a leave or shift event happens then
7:     change all keys in the leave path, and broadcast corresponding rekey messages;
8: end if
9: update renew, refresh and revive spots according to the latest rekey messages;
```

satisfies $ka_i \leq ua_i$ at current time t, and k is sent to u without any changed.

Theorem 4.1 indicates that changing only critical keys can ensure past confidentiality. Hence, given a key forest, Algorithm 3 is applied to find the best enroll path and minimize the rekey cost. When a join or shift event happens to a tree, the algorithm uses the depth-first tree traversal approach to find all critical keys in the tree. If a path is found to have fewer critical keys than previously visited paths, the algorithm records it as the best enroll path.

D. Examples of Critical Keys

Corollary 4.1: When a user joins a tree, a key in the enroll path is a critical key if and only if one of the key's ages is greater than 0.

Before a user joins the tree, his subscription ages for all of the programs sharing this tree are 0. Hence, if the age of a key in the enroll path for this program is greater than 0, the key is older than the user's subscription. According to Theorem 4.1, the key needs to be changed before being distributed to the user.

Example: consider a user event where a user u_2 joins tree tr_4 at time t_2 in Figure 4. Assume that before t_2, another user u_1 shifts from tree tr_4 to tr_6 at t_1, and no other event happens between t_1 and t_2. At t_1, assume k_{n_L} and k_{r_4} are in the leave path. At t_2, assume k_{n_L} and k_{r_4} are in the enroll path. Now, we determine which keys are critical at t_2 according to Corollary 4.1.

At t_1, because u_1 is still in g_1 and g_2, the broadcast server does not need to change k_{r_1} and k_{r_2}, and only needs to send the refreshments $\delta(k_{n_L}, t_1; k_c, t_{k_c}), \delta(k_{r_4}, t_1; k_{n_L}, t_1)$, where k_c is a child key of k_{n_L} and not known by u_1. According to Definition 4.5, no revive spot is added to k_{n_L} and k_{r_4}, and these two keys are renewed after t_1. Consequently, according to Definition 4.6, at t_2, the ages of both k_{n_L} and k_{r_4} are 0. The server can give k_{n_L} and k_{r_4} to u_2 without any change according to TCK, since u_2 cannot derive k_{r_1} and k_{r_2} before t_2 from the previous refreshments.

In this example, k_{n_L} and k_{r_4} are not critical keys, although they are in the enroll path at t_2. However, k_{r_1} and k_{r_2} are critical keys at t_2 because their ages are greater than 0. The server needs to change k_{r_1} and k_{r_2} before distributing them to u_2. This shows that it is not necessary to change all keys in the enroll path when a user subscribes the broadcast data services as would be required in the traditional LKH approach.

Corollary 4.2: After a user enrolls in a tree, all keys in the enroll path are not older than the user.

According to Theorem 4.1, if a key is older than the user's subscription regarding a program, the key needs to be changed. Hence, at the time when the user enrolls in a tree, the keys, whose ages are older than the user, are renewed and their ages turns to be 0. If the key is not older than the user's subscription regarding any program, the key does not need to be changed. Hence, the key continues to be not older than the user. Therefore, after a user enrolls in a tree, all keys in the enroll path are not older than the user.

Corollary 4.3: When a user shifts from a tree to another tree, the keys overlap both trees do not need to be changed.

Assume the user shifts from tree tr_α to tree tr_β. According to Corollary 4.2, after the user enrolls in tr_α, all keys in the enroll path cannot not be older than the user. Hence, when the user shifts to tr_β, the overlapped keys, which were in the enroll path when user enrolled in tr_α, do not need to be changed according to Theorem 4.1.

Example: in Figure 4, user u shifts from tree tr_4 to tr_6 at t_1. Assume user u is in tr_4 since t_0, i.e. u can decrypt data items of g_1 and g_2 since t_0. k_{r_1} and k_{r_2} are the overlapped keys with tr_4 and tr_6. Because u is still in g_1 and g_2 after he shifts to tr_6, k_{r_1} and k_{r_2} do not need to be changed.

E. Security Analysis

To ensure multicast or broadcast security, group key management should satisfy four security properties [2], [3]: non-group confidentiality, collusion freedom, future confidentiality (forward secrecy), and past confidentiality (backward secrecy). In the following, we discuss how KTR satisfies these properties.

Property 4.1: Non-group confidentiality: passive adversaries should not have access to any group key.

Because keys are encrypted when being broadcast, passive adversaries can not decrypt any key without knowing decryption key. Hence, KTR obviously satisfies Property 4.1.

Property 4.2: Collusion freedom: by sharing group keys, multiple present users can not derive any group key that they are not holding.

When multiple users collude, they may try to share their keys to derived unknown group keys. The sharing can be represented by a subgraph of the paths belonging to the colluding users. However, in KTR, a user does not know any key not on this path. Hence, colluding users do not know any key outside the subgraph that represents the collusion. KTR thus satisfies Property 4.2.

Property 4.3: Future confidentiality (forward secrecy): a leaving user should not have access to any group key after leaving his present group.

According to Algorithm 3, KTR changes all keys in the leave path, because the leaving user holds these keys. Hence, the leaving user will not have the new keys after the user leaves his group. KTR thus satisfies Property 4.3.

Property 4.4: Past confidentiality (backward secrecy): a joining user added at time t should not have access to any keys used to encrypt data before t.

According to Algorithm 3, KTR changes all critical keys in the enroll path when a user joins. Theorem 4.1 basically proves that the joining user can only derive past group keys from critical keys. Hence, changing critical keys and reusing non-critical keys prevent the joining user from obtaining past group keys. KTR thus satisfies Property 4.4.

V. Performance Evaluation

In this section, we analyze and evaluate the performance of KTR at the server side and the client side respectively. We define the following parameters for the analysis.

- The service provides m programs;
- n users subscribes the service;
- n users are distributed to e trees;
- Each tree is shared by d programs on average;
- The leave rate is λ_l: the number of users unsubscribing to the service per unit time;
- The join rate is λ_j: the number of users subscribing to he service per unit time;
- The shift rate is λ_s: the number of users changing subscriptions per unit time;
- The total event rate $\Lambda = \lambda_l + \lambda_j + \lambda_s$

A. Server Side

1) Analysis: Since a server is generally abundant in energy and memory, its computation capacity becomes the main factor that affects the performance of the whole system. If the processing time for each event is large, this would delay user's request. We measured the management cost of a server by two metrics: (a) the total number of keys to be managed, and (b) the number of keys to be inspected and updated per rekey event.

Since a tree has $\frac{n}{e}$ users in average, there are $O(2\frac{n}{e} - 1)$ keys to be handled, including the root. For e trees, the server needs to manage $O(e(2\frac{n}{e} - 1)) = O(2n - e)$ keys. Because a key in any tree needs to keep a separate spot series for each program that shares the tree and each tree is shared by d programs on average, the server needs to keep $O((2n - e)d)$ spot series. In the multi-layer root graph, the server needs to manage DEKs and other non-root keys. Because one program is connected with $O(\frac{ed}{m})$ trees, the server needs to manage $O((\frac{ed}{m} - 1)m) = O(ed - m)$ keys (including DEKs and non-root keys) for m programs. We notice that any one of these keys is connected with only one program. Hence, each key requires only one spot series, and thus the server needs to keep $O(ed - m)$ spot series. In total, the server needs to manage $O(2n-e+ed-m)$ keys with a storage requirement of $O((2n-e)d + ed - m) = O(2nd - m)$. In comparison, LKH requires that the server manages $O((2\frac{ed}{m}\frac{n}{e}-1)m) = O(2nd-m)$ keys.

According to Algorithm 3, KTR needs three steps for each rekey operation: (1) find the leave path and the best enroll path, (2) change keys in the leave path and critical keys in the enroll path, and (3) update renew, refresh and revive spots for affected keys. Compared with LKH, steps (1) and (3) in KTR incur extra cost that includes only comparison and assignment operations. The leave path simply consists of keys that the user will no longer hold, and thus the server can easily find out the leave path. Nevertheless, the server needs to inspect the to-be-joined tree and find out the enroll path with the minimum number of critical keys. Hence, according to Algorithm 3, the computation complexity on the server side is mainly determined by the number of keys to be inspected and the number of updates to be committed.

TABLE II
Computation at the sever side

service	parameters				inspe-	upd-	computa-
size	n	m	e	d	ction	ate	tion (ms)
small	10K	5	31	2.5	655	50K	16 ± 2
large	10K	50	300	12	141	240K	68 ± 16

The number of keys to be inspected is determined by the number of users in the to-be-joined tree and the number of programs that share that tree. Inside the tree, there are $O(2\frac{n}{e})$ keys. The root of the tree is connected to DEKs of the programs that share the tree. As in Figure 5, the path between the root and one DEK has $O(log_2(\frac{ed}{m}))$ keys on average. Hence, there are $O(dlog_2(\frac{ed}{m}))$ keys between the root and the connected DEKs. In total, the server needs to inspect $O(2\frac{n}{e} + dlog_2(\frac{ed}{m}))$ keys in order to find the best enroll path and the critical keys.

The number of updates to be committed is determined by the number of DEKs that must be changed. In the worst case, all DEKs need to be changed and all keys under the DEKs need to be changed. Because each tree is shared by d programs on average, one DEK is connected with $\frac{de}{m}$ trees. Hence, in the worst case, a changed DEK will require the server to update all keys in the trees that are connected with the DEK. Hence, for each DEK, $O(2\frac{de}{m}\frac{n}{e}) = O(2\frac{dn}{m})$ keys need to be updated. In the worst case (i.e. m DEKs need to be changed), the server needs to commit $O(2dn)$ updates.

2) Simulation: We did simulations on a server (a computer with a 2GHz CPU and 2GB RAM running Linux). On average, the server uses tens of milliseconds for one rekey operation in the KTR scheme. Table II shows the simulation results for two services with 10000 users. The first row is a small service that provides only 5 programs and 31 valid trees, and each tree is shared by 2.5 programs on average. The second row is a large service that provides 50 programs and 300 valid trees, and each tree is shared by 12 programs on average. Column "inspection" is the estimate of the number of keys to be inspected in the worst case. Column "update" estimates the number of updates to be committed in the worst case. Column "computation" measures the average computation time for one rekey operation in simulation.

Obviously, the computation time on the server side is mainly determined by the number of updates to be committed in the simulations, as the computation time is almost proportional to the number of updates (via regression analysis). First, since the number of users is fixed in these simulations, when the service has more trees, the average number of users in one tree decreases. Accordingly, the server takes less time to inspect keys in the large service than in the small service. Hence, the server spends the most time on updating spots for affected keys. Since a program is connected with more trees in the large service, a changed DEK will affect more keys. According to the estimation, in the worst case, the server needs to commit updates in the large service $\frac{240000}{50000} = 4.8$ times as much as in the small service. Overall, as measured, the server's computation time in the large service is $\frac{68}{16} = 4.3$ times as much as in the small service.

B. Client Side

1) Analysis: At the client side, three main performance metrics need to be measured: *average rekey message size per event*, *average number of decryption per event per user*, and *maximum number of keys to be stored*, that can well capture the overhead of KTR on resource limited mobile devices in terms of communication, storage, power consumption and computation. Based on these three metrics, we can infer other metrics that are more directly related to the mobile devices. For example, if we decide a particular encryption algorithm, we know the length of a key, the time to compute a key, and the energy consumption to execute the algorithm. Consequently, we can obtain metrics, such as the communication overhead in the rekey messages, etc.

The first metric, *average rekey message size per event*, is measured as the number of encryptions $\{*\}_k$ in the rekey message, represents communication cost and power consumption in a mobile device. In a leave event, the rekey message consists of rekey items for keys in the leave path. Hence, the number of encryptions is $O(2(log_2(\frac{n}{e})+dlog_2(\frac{de}{m})))$. In a join event, the rekey message consists of rekey items for critical keys in the enroll path. Hence, the number of encryptions is $O((1+\rho)(log_2(\frac{n}{e})+dlog_2(\frac{de}{m})))$, where ρ is the ratio of critical keys over all keys in the enroll path. Our simulation shows that ρ is around $22.9 \pm 1.8\%$ in an enroll path. In a shift event, the rekey message consists of rekey items for keys in the leave path and critical keys in the enroll path. Hence, the number of encryptions is $O((3+\rho)(log_2(\frac{n}{e}) + dlog_2(\frac{de}{m})))$. Considering user activities, the average rekey message size would be $O(\frac{1}{\Lambda}(2\lambda_l + (1+\rho)\lambda_j + (3+\rho)\lambda_s)(log_2(\frac{n}{e}) + dlog_2(\frac{de}{m})))$.

The second metric, *average number of decryption per event per user*, measures computation cost and power consumption in a mobile device, since the device needs to decrypt new keys from rekey messages. Define the height of a key as its distance to the bottom of a tree. For example, if a tree has n users, the height of its root key is $log_2 n$. Obviously, if a key's height is h, 2^h users need to decrypt it when it is updated in a rekey event. Hence, in a tree of n users, when a path of keys need to be changed, the total number of decryptions is $O(\sum_{i=1}^{log_2^n} 2^i) \approx O(2n)$. In a leave event, because there are $O(\frac{n}{e}\frac{de}{m})$ users under a to-be-changed DEK, the total number of encryptions for this DEK and its downward keys is $O(2\frac{nd}{m})$. Because d DEKs need to be changed in a leave event, $O(2\frac{nd^2}{m})$ decryptions are committed by affected users. Similarly, in a join event, the total number of decryptions is $O(2\rho\frac{nd^2}{m})$. In a shift event, the number of decryptions is $O(2(1+\rho)\frac{nd^2}{m})$. Considering user activities, the average number of decryptions would be $O(\frac{1}{\Lambda}(2\lambda_l + 2\rho\lambda_j + 2(1+\rho)\lambda_s)\frac{nd^2}{m})$. Hence, on average, each user decrypts $O(\frac{1}{\Lambda}(2\lambda_l + 2\rho\lambda_j + 2(1+\rho)\lambda_s)\frac{d^2}{m})$ rekey items upon receiving a rekey message.

The third metric, *maximum number of keys to be stored*, represents the storage demand which is proportional to the number of keys a user needs to hold. Inside a tree, a user needs to hold keys along the path from its leave node to the root of the tree. Because a tree has $O(\frac{n}{e})$ users, a user holds $O(log_2(\frac{n}{e}))$ keys inside a tree. In the root graph, a user hold keys from the root of the tree to all DEKs of the programs that share the tree. Because a root is connected with d DEKs and each DEK is connected with $O(\frac{ed}{m})$ trees, a user holds $O(dlog_2(\frac{de}{m}))$ keys in the root graph. In total, a user should hold $O(log_2(\frac{n}{e}) + dlog_2(\frac{de}{m}))$ keys. Because a user stays in multiple trees in LKH, the user should hold $O(dlog_2(\frac{n}{e}) + dlog_2(\frac{de}{m}))$ keys if keys are not shared.

2) Simulation: The analysis gives estimates on major performance metrics. We notice that some factors could bring more insightful results. For example, users may not be evenly distributed in trees due to the fact that some programs may be more popular than other programs. Also, users may be more or less likely to stay in current subscriptions than to frequently change their subscriptions. Hence, in the following, we use simulation to examine the impacts of these user behaviors.

In the simulation, we compare KTR with three other representative schemes: SKT, eLKH, LKH, as listed in Table III, to illustrate how *shared key* and *critical key* improve the performance of key management in wireless broadcast services. SKT is the approach in [28] where only shared key tree is applied. eLKH is an approach where only critical key is applied to enhance LKH. Neither shared key tree nor critical key is adopted in LKH. If shared key tree is adopted, key management is based on the key forest as illustrated in Figure 4; otherwise, a key tree is created for each program and a user is assigned to all trees corresponding to the programs he subscribes. If critical key is used, a key in an enroll path is changed if and only if it is a critical key; otherwise, all keys in the enroll path need to be changed (as in the other old schemes). Table III lists four schemes representing different solutions which may or may not adopt these two ideas. The names of the schemes are self-explanatory. If key tree reuse is adopted, key management is based on the key forest as illustrated in Figure 4; otherwise, a key tree is created for each program and a user is assigned to all trees corresponding to the programs he subscribes. If critical key is used, a key in an enroll path is changed if and only if it is a critical key; otherwise, all keys in the enroll path need to be changed (as in other schemes). Note that the well-known LKH is used as a base line (i.e. neither shared key nor critical key is adopted).

a) Settings: We assume that the server provides 50 programs. In our experiments, the key forest consists of 300 trees when shared key tree is adopted. Each tree represents a different option of subscriptions. We also assume that there are 10000 users (on average) subscribing to the services. The root graph in key forest was automatically generated according

TABLE III
KEY MANAGEMENT SCHEMES

Schemes	*shared key*	*critical key*
KTR	Y	Y
SKT	Y	N
eLKH	N	Y
LKH	N	N

TABLE IV
CASES IN KEY MANAGEMENT

Case	Major subscriptions	Major events
Case I	Multiple	Join and leave
Case II	Single	Join and leave
Case III	Multiple	Shift
Case IV	Single	Shift

to subscriptions. Each program had a different depth in root graph depending on the number of subscriptions that include the program. In the worst case where a program is included in all subscriptions, the depth for this program in root graph is 9. Most programs had a depth of 8 or less, because the probability that a program is selected by more than 256 subscriptions is low. At the same time, the tree depth is around 5, since 10000 users are randomly distributed to 300 trees.

In each experiment, we compare two equivalent key graphs for shared key schemes and non-shared key schemes. We first generate a random key forest and assign users to leaf nodes of the key forest. Then, for non-shared key schemes (LKH and eLKH), we assign users to different key trees according to their subscriptions in shared key schemes. Any user event in shared key schemes was also mapped into the equivalent event in non-shared key schemes.

All three user events (i.e. join, shift and leave) are modeled as independent Poisson processes. We let $\lambda_l = \lambda_j$ so the total number of users remains constant (i.e. around 10000). We vary λ_l and λ_s separately in order to observe their impacts on the rekey performance. The result of our performance comparison is obtained by averaging the rekey cost over 3000 random user events. Here, a user event is referred to an event in schemes that adopt shared key tree. Such an event is mapped into several user events in schemes that do not adopt shared key tree. For example, a user joins a tree of multiple programs in KTR is mapped as a sequence of events in LKH (each is corresponding to the user joining a tree of these programs).

Four test cases are generated for evaluation based on major subscriptions and major events (summarized in Table IV). In Case I and Case III, 80% of the users subscribe to multiple programs and the other 20% only subscribe to one of the programs. In Case II and Case IV, 20% of the users subscribe to multiple programs and the other 80% subscribe to only one program. Furthermore, in Case I and Case II, the major events are joins and leaves; while in Case III and Case IV, the major events are shifts. In the simulations, we vary the rates for the major events while keeping the other rates at 1.

b) Average Rekey Message Size Per Event: We first evaluate performance of the key management schemes in terms of average rekey message size, by fixing $\lambda_s = 1$ and varying λ_l (x-axis) from 1 to 9 as shown in Figure 9(a) and (b). KTR and SKT that adopt shared keys significantly outperforms LKH, Because the major user events are join and leave, a user only needs to join or leave one tree when he subscribes or unsubscribes to multiple programs. In contrast, LKH that does not have shared keys requires a user needs to join or leave multiple trees for multiple programs. Furthermore, KTR is better than SKT, because critical key in KTR allows the server to change fewer keys.

Then, we evaluate performance of the key management schemes by fixing $\lambda_l = 1$ and varying λ_s (x-axis) from 1 to 9 as shown in Figure 9(c) and (d). Obviously, when the shift rate grows, the performance of SKT turns worse because of the extra overhead that is introduced in managing shared keys when a user quits or adds some but not all of his subscribed programs. Obviously, the adoption of critical keys incurs the major improvement in terms of rekey message size. KTR and eLKH are the best solutions.

By comparing CASE III and CASE IV (corresponding to the subscriptions of multiple programs and single program, respectively), we found that the extra overhead of shift is higher in CASE III. Figure 9 also shows that KTR and STK are more sensitive to these types of major user events than eLKH and LKH. The average rekey message size in KTR and STK grows as the shift rate increases and shrinks as the join/leave rate increases. On the contrary, the average rekey message size in eLKH and LKH remains almost flat regardless of the rate of user events.

Based on our experimental results, by adopting only shared key tree, SKT reduces the rekey message size to around 60% to 90% of LKH. By adopting critical keys alone, eLKH reduces the rekey message size to around 55% to 65% of LKH. This result also validates our claim that many keys in the enroll path do not need to be changed. In fact, over all the experiments, only around 23% keys in the enroll path need to be changed.

c) Average Number of Decryption Per Event Per User: Power consumption and computation cost are two primary concerns for mobile users. We use the average number of decryptions to measure these costs. Similar to the experiments in the previous section, we vary the rates of major events to observe their impacts on decryption overhead. Figure 10 shows that the schemes adopting critical keys (i.e. KTR and eLKH) are better than SKT and LKH. The number of decryption in eLKH is around 80% of that in LKH, and KTR's is around 80% of SKT's too. In all schemes, the number of decryptions drops as the join/leave rate increases and rises as the shift rate increases.

However, the adoption of shared key has negative impacts on reducing user's decryption cost. In Figure 10, LKH is better than SKT, especially when the shift rate is high. So does eLKH in comparison with KTR. This can be explained as follows. In shared key schemes, when a user shifts from a tree to another, some users will be affected by the changed keys in both the leave and the enroll paths if they subscribe to the programs that the user keeps subscribing to. As previously discussed, shared key schemes introduce extra rekey cost because they change keys in two paths. On the contrary, in non-shared key schemes,

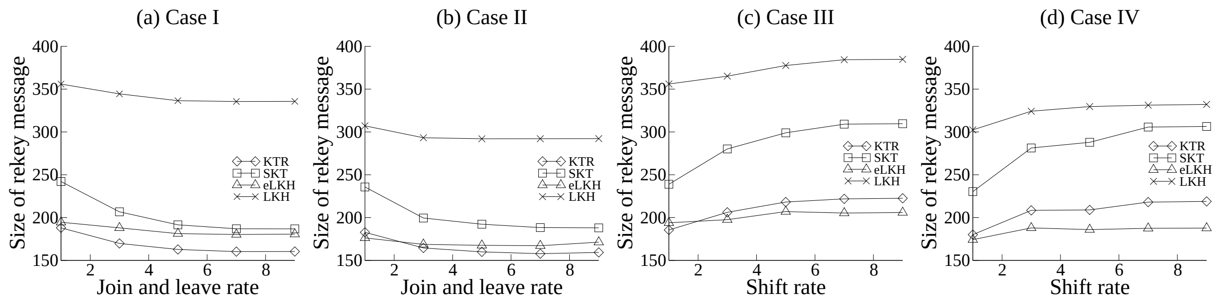


Fig. 9. Average rekey message size per event

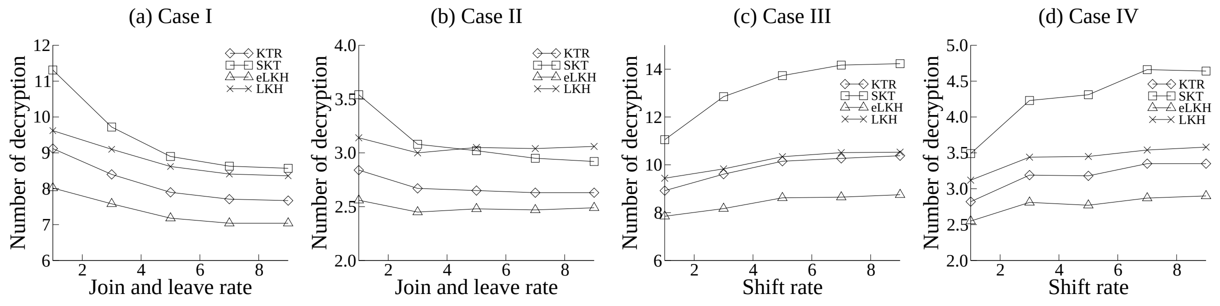


Fig. 10. Average number of decryption per event per user

no key needs to be changed for the programs the user keeps subscribing to. Hence, those users who are affected by keys in two paths in shared key schemes only need to decrypt keys in the leave path in non-shared key schemes. As a consequence, the decryption cost per user is less in non-shared key schemes that that in shared key schemes.

Figure 10 also shows that the user subscription pattern has a great impact on the average number of decryptions. The average number of decryptions in Case II and Case IV (where only 20% of users subscribe multiple programs) is around 55% of that in Case III and Case IV (where 80% of users subscribe multiple programs). Obviously, if a user subscribes to more programs, it is more likely that he will be affected by other user activities.

d) Average Number of Keys Held Per User: Finally, we evaluate the storage demand on mobile devices, which is measured as the average number of keys held by each user. One goal is to save storage by reducing the number of keys each user needs to hold. Since KTR makes programs share keys, KTR saves storage for a user when the user joins a tree shared by more programs. As analyzed before, the structure that programs share trees determines the number of keys that can be saved. However, since users may favor some subscriptions, users may concentrate in some trees. For users subscribing to single programs, KTR has no advantage over LKH.

The average storage demand is also affected by how users are distributed in trees. In Figure 11, we vary the ratio of users who only subscribe single programs and illustrate the average storage demand. Obviously, when most users subscribe to multiple programs (i.e. only 20% users subscribe single programs), KTR can save 45% storage on average for each user compared to LKH which assigns a separate set of keys to each program. When more users (80% users) subscribe

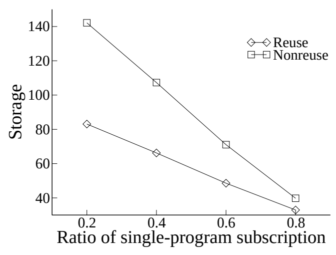


Fig. 11. Average number of keys hold per user

to single programs, the storage reduction in KTR is only 10%.

In summary, KTR combines the advantages of both shared key tree and critical key. Among all schemes, it has a light communication overhead (i.e. its average rekey message size per event is the least or close to the smallest), incurs less computation and power consumption on mobile devices than the other schemes (i.e. its average number of decryption per event per user is the smallest), and requires least storage in mobile devices (i.e. its average number of keys held per user is the smallest). Because a mobile receiver generally only has limited resources, such an overhead saving can greatly benefit the receivers so that they can have a longer working duration and more computation capacity to process broadcast data.

VI. Conclusion

In this work, we investigated the issues of key management in support of *secure* wireless broadcast services. We proposed KTR as a scalable, efficient and secure key management approach in the broadcast system. We used the key forest to exploit the overlapping nature between users and programs in broadcast services. KTR let multiple programs share a single tree so that the users subscribing these programs can hold fewer keys. In addition, we proposed a novel shared key management approach to further reduce rekey cost by identifying the minimum set of keys that must be changed to ensure broadcast security. This approach is also applicable to other LKH-based approaches to reduce the rekey cost as in KTR. Our simulation showed that KTR can save about 45% of communication overhead in the broadcast channel and about 50% of decryption cost for each user, compared with the traditional LKH approach.

Acknowledgment

This work was partially supported by NSF ANI-0335241, NSF CCR-0233324, and Department of Energy Early Career PI Award. Wang-Chien Lee was supported in part by the National Science Foundation under Grant no. IIS-0328881, IIS-0534343 and CNS-0626709.

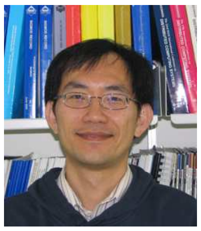
References

[1] J. Xu, D. Lee, Q. Hu, and W.-C. Lee, "Data broadcast," in *Handbook of Wireless Networks and Mobile Computing*, I. Stojmenovic, Ed. New York: John Wiley and Sons, 2002, pp. 243–265.
[2] D. Wallner, E. Harder, and R. Agee, "Key management for multicast: issues and architectures," *IETF RFC 2627*, 1999.
[3] J. Snoeyink, S. Suri, and G. Varghese, "A lower bound for multicast key distribution," in *IEEE Infocom*, vol. 1, 2001, pp. 422–431.
[4] S. Mittra, "Iolus: a framework for scalable secure multicasting," in *ACM SIGCOMM*, vol. 277-288, 1997.
[5] C. K. Wong, M. Gouda, and S. S. Lam, "Secure group communications using key graphs," in *ACM SIGCOMM*, 1998, pp. 68–79.
[6] Y. Kim, A. Perrig, and G. Tsudik, "Simple and fault-tolerant key agreement for dynamic collaborative groups," in *ACM CCS*, 2000, pp. 235–244.
[7] S. Setia, S. Koussih, S. Jajodia, and E. Harder, "Kronos: a scalable group re-keying approach for secure multicast," in *IEEE Symposium on Security and Privacy*, 2000, pp. 215–228.
[8] Y. R. Yang, X. S. Li, X. B. Zhang, and S. S. Lam, "Reliable group rekeying: a performance analysis," in *ACM SIGCOMM*, 2001, pp. 27–38.
[9] M. Onen and R. Molva, "Reliable group rekeying with a customer perspective," in *IEEE GLOBECOM*, vol. 4, 2004, pp. 2072–2076.
[10] B. Briscoe, "Marks: zero side effect multicast key management using arbitrarily revealed key sequences," in *NGC*, 1999, pp. 301–320.
[11] A. Wool, "Key management for encrypted broadcast," *ACM Transactions on Information and System Security*, vol. 3, no. 2, pp. 107–134, 2000.
[12] M. Just, E. Kranakis, D. Krizanc, and P. v. Oorschot, "On key distribution via true broadcasting," in *ACM CCS*, 1994, pp. 81–88.
[13] M. Luby and J. Staddon, "Combinatorial bounds for broadcast encryption," in *Advances in Cryptology, Eurocrypt*, 1998, pp. 512–526.
[14] A. Fiat and M. Naor, "Broadcast encryption," in *Advances in Cryptology, CRYPTO*, 1994, pp. 480–491.
[15] C. Blundo and A. Cresti, "Space requirements for broadcast encryption," in *Advances in Cryptology, Eurocrypt*, 1994, pp. 471–486.
[16] D. Naor, M. Naor, and J. B. Lotspiech, "Revocation and tracing schemes for stateless receivers," in *Advances in Cryptology, CRYPTO*, 2001, pp. 41–62.
[17] *Basic Interoperable Scrambling System*, http://www.ebu.ch/CMSimages/en/tec_doc_t3292_tcm6-10493.pdf, 2002.
[18] "North american mpeg-2 information," http://www.coolstf.com/mpeg/.
[19] "Irdeto access," http://www.irdeto.com/index.html, 2006.
[20] Markus G. Kuhn, "Analysis of the nagravision video scrambling method," University of Cambridge, Tech. Rep., 1998.
[21] "Viaccess," http://www.viaccess.com/en/, 2006.
[22] "Nds videoguard: Security, flexibility, and growth," http://www.ndsworld.com/conditional_access/conditional_access.html, 2006.
[23] R. Canetti, J. Garay, G. Itkis, D. Micciancio, M. Naor, and B. Pinkas, "Multicast security: a taxonomy and some efficient constructions," in *IEEE Infocom*, vol. 2, 1999, pp. 708–716.
[24] A. Perrig, D. Song, and D. Tygar, "Elk, a new protocol for efficient large-group key distribution," in *IEEE Symposium on Security and Privacy*, 2001, pp. 247–262.
[25] C. K. Wong and S. S. Lam, "Keystone: a group key management service," in *International Conference on Telecommunications*, 2000.
[26] M. Moyer, J. Rao, and P. Rohatgi, "Maintaining balanced key trees for secure multicast," *draft-irtf-smug-key-tree-balance-00.txt*, 1999.
[27] J. Staddon, S. Miner, M. Franklin, D. Balfanz, M. Malkin, and D. Dean, "Self-healing key distribution with revocation," in *IEEE Symposium on Security and Privacy*, 2002, pp. 241–257.
[28] Y. Sun and K. R. Liu, "Scalable hierarchical access control in secure group communications," in *IEEE Infocom*, 2004.

Qijun Gu is an assistant professor of Computer Science at Texas State University - San Marcos. He received the Ph.D. degree in Information Sciences and Technology from Pennsylvania State University in 2005, the Master degree and the Bachelor degree from Peking University, China, in 2001 and 1998. His research interests cover various topics on networking, security and telecommunication, including vulnerability in sensor applications, authentication in ad hoc and sensor networks, and security in peer to peer systems, denial of service in wireless networks, key management in broadcast services, worm propagation and containment.

PLACE
PHOTO
HERE

Peng Liu is an associate professor of Information Sciences and Technology at the Pennsylvania State University. He is the Research Director of the Penn State Center for Information Assurance, and Director of the Cyber Security Lab. His research interests are in all areas of computer and network security. Dr. Liu has published a book and over 100 refereed technical papers. His research has been sponsored by DARPA, NSF, AFOSR, DOE, DHS, ARO, NSA, CISCO, HP, Japan JSPS, and Penn State. Dr. Liu is a recipient of the US Dept. of Energy Early CAREER PI Award.



Wang-Chien Lee is an Associate Professor of Computer Science and Engineering at Pennsylvania State University. He received his B.S. from the Information Science Department, National Chiao Tung University, Taiwan, his M.S. from the Computer Science Department, Indiana University, and his Ph.D. from the Computer and Information Science Department, the Ohio State University. He is particularly interested in developing data management techniques (including accessing, indexing, caching, aggregation, dissemination, and query processing) for supporting complex queries in a wide spectrum of networking and mobile environments such as peer-to-peer networks, mobile ad-hoc networks, wireless sensor networks, and wireless broadcast systems. Meanwhile, he has worked on XML, security, information integration/retrieval, and object-oriented databases. His research has been supported by NSF and industry grants. Most of his research result has been published in prestigious journals and conferences in the fields of databases, mobile computing and networking.

PLACE
PHOTO
HERE

Chao-Hsien Chu is a Professor of Information Sciences and Technology and the founding director of the Center for Information Assurance at Penn State. His Ph.D. in Business Administration was from Penn State. Dr. Chu's recent research focuses on information assurance & security, RFID technologies integration, security, and deployment, emerging information/intelligent technologies, and supply chain integration. He has published more than 110 refereed articles in top-ranking journals and in major conference proceedings. Three of his papers received the best paper awards. His research was supported by National Science Foundation, Department of Defense, National Security Agency, HP, Cisco Systems, and others.