eKTR: an Efficient Key Management Scheme in Wireless Data Broadcast Services

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

Pennsylvania State University, University Park, PA 16802

E-mail: qgu@ist.psu.edu, pliu@ist.psu.edu, wlee@cse.psu.edu, chu@ist.psu.edu

Abstract

Wireless broadcast is an effective approach to disseminate data to a number of users. To provide secure access to the broadcast data, 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 in the broadcast system. In this paper, we propose an efficient key management scheme, eKTR, to address this issue. eKTR lets multiple programs share a single key tree so that the users subscribing these programs can hold less keys. In eKTR, we also propose an approach to further reduce rekey cost by identifying the minimum set of keys that must be changed to ensure broadcast security. Our simulations show that eKTR can save about 45% of communication overhead in the broadcast channel and about 50% of decryption cost for each user, compared with the traditional logical key hierarchy based approach.

1. Introduction

Subscribe-publish based wireless data broadcast services have been available as commercial products for many years. In particular, the recent announcement of the MSN Direct Service has further highlighted the industrial interest in and feasibility of utilizing broadcast for wireless data services. Previous studies on wireless data broadcast services have mainly focused on performance issues such as reducing data access latency and conserving battery power of the mobile devices. Unfortunately, the critical security requirements of this type of broadcast services have not yet been addressed.

In the wireless broadcast environment, any user can monitor the broadcast channel and log the broadcast data. If the data is not encrypted, the content is open to the public. Symmetric-key-based encryption is a natural choice for ensuring secure data dissemination. The broadcast data items can be encrypted such that only the users who own the valid keys can decrypt them. Thus, the decryption keys can be used as an effective means for access control in wireless data broadcast services. Nevertheless, a critical issue remains, i.e. *how can we manage the keys when a user joins/leaves/changes the service without compromising security and interrupting operations of other users?*

There are a great number of existing studies on key management in the literature. However, they are not the most efficient solution for users regarding various key management operations under complex subscription options in our system. Hence, we propose a new key management scheme, namely *enhanced key tree reuse* (**eKTR**), based on the finding that users who subscribe multiple programs can be captured by a shared key tree. Moreover, we observe that when a user adds more programs, there is no need to change the data encryption keys for the programs he currently subscribes. This observation results in an important contribution of eKTR where we present an approach to reduce the rekey cost to the minimum without compromising security. This result, adopted as part of our scheme, can also be employed in traditional key management schemes (e.g. *logical key hierarchy* (**LKH**) [1, 2]) to improve their performance.

Sun *et al.* [3] independently proposed a scheme which has a key tree structure similar to ours. However, our research differs from their work in many aspects. First, the subject application and focus of the researches are different. To the best of our knowledge, this is the first paper addressing the security issue in the subscribe-publish based wireless data broadcast services. The distribution (and associated cost) of keys via wireless broadcast to users is unique. Second, [3] mainly focuses on reducing the redundancy of key trees; while eKTR minimizes the rekey cost based on two ideas: the new key structure and the unique approach to decide whether a key needs to be changed. Third, we use extensive simulations to examine the impacts of these two ideas on the performance of eKTR. The experimental results show that these two ideas in eKTR have different dominant impacts on the performance, while both of them can achieve better performance than the representative key management scheme (i.e. LKH).

The rest of the paper is structured as follows. In Section 2, we describe the architecture of the broadcast system, present related works on group key management, and provide the rationale of designing a new key management scheme in our system. In Section 3, the detail of the KTR scheme is presented. In Section 4, we present

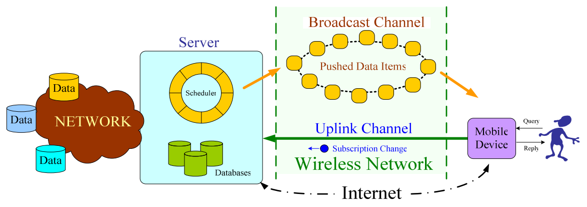


Figure 1: A Wireless Data Subscribe-Publish System

eKTR as the enhancement to minimize the rekey cost by identifying the condition whether a key needs to be changed in order to ensure broadcast security. In Section 5, we present the results of simulations to illustrate the performance improvements in KTR and eKTR. Finally, we conclude in Section 6.

2. Background and Problem Statements

2.1. Architecture of Broadcast Services

A wireless subscribe-publish broadcast system consists of three parts: (1) the broadcast server; (2) the mobile devices; and (3) the communication mechanism. Figure 1 shows a high level view of the system. The server is responsible for scheduling the broadcast of data items and new keys. A mobile device continues monitoring the broadcast data to receive the information of interest to its user. The specialty of this broadcast system is that the mobile device, instead of the server, needs to process user's *queries* and filter out unwanted data packets from a lot of information being broadcast.

The communication mechanism includes wireless *broadcast channels* and (optional) *uplink channels*. Broadcast channel is the main mechanism for data and key dissemination in our system. A broadcast data item is not necessarily designated to one user only. This allows an arbitrary number of users to receive the data packets at the same time and addresses the scalability problem. The uplink channels, which have limited bandwidth, are reserved for occasional uses by the users to change subscription dynamically or request lost or missed keys.

The *subscribe-publish* model of our broadcast system connects data sources and users together. In this paper, data items are grouped into *programs* and a user specifies which programs he would like to access. The set of programs the user subscribes is called the user's *subscription*. Users can subscribe via Internet (off-line) or via uplink channels (on-line) to specify the programs they are interested in receiving (monitoring). Each program has one key to encrypt the data items. The key is issued to the user who is authorized to decrypt and receive the data items. If a user subscribes multiple programs, it needs keys for all of these programs.

2.2. Related Works on Key Management

Secure key management for wireless broadcast is closely related to secure group key management in networking. Mittra [4] proposed to partition a group into multiple subgroups and organize them in a hierarchy, in which clients are the leaves and group security agents are the non-leaf nodes. A more popular approach, i.e. LKH, is proposed in [1, 2]. In LKH, a key tree is applied 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* of a user that is only shared between the system and the user. Other keys in the tree, namely *key distribution keys* (**KDK**s), are used to encrypt new DEKs and KDKs. A user in the tree only knows the keys along the path from the leaf of the user to the root of the key tree. When a user adds or quits a program, the system changes corresponding DEKs and KDKs, and other users use their known keys to decrypt new keys (see examples in [1]). The operation to distribute new keys is called *rekey*. In our system, data and rekey messages are broadcast in the same broadcast channel to the users.

Many variations of LKH have been proposed to further study and improve the LKH approach. [5] proposes a combination of key tree and Diffie-Hellman key exchange to provide a simple and fault-tolerate key agreement for collaborative groups. [6] reduces the number of rekey messages, while [7] reduces message loss and thus lost keys. Balanced and unbalanced key trees are discussed in [1] and [8]. Periodic group re-keying is studied in [9, 10] 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, 10]. Moreover, the performance of LKH is also thoroughly studied [10, 11].

LKH is an efficient and secure key management for the broadcast service with one program, since each group member only needs to hold $O(log(n))$ keys assigned along the path from itself to the root, and the size of a broadcast rekey message for each new DEK is also $O(log(n))$. LKH and most of its variations can definitely be applied in our broadcast system, however they are not the best solution. In the broadcast system, there are multiple programs and complex subscriptions. Typical LKH-based schemes do not exploit the overlap nature of subscriptions in the broadcast system. If each program has an individual key tree, it would be costly for users who subscribe multiple programs. We propose eKTR to let multiple programs share the same key tree. The most similar idea is presented in [3]; however, eKTR can uniquely identify the condition whether a key needs to be changed so as to minimize the rekey cost.

There are some other candidate key management schemes in the literature. In the area of broadcast cryptograph, OR protocol [12] yields maximal resilience against arbitrary coalitions of non-privileged users. However, the size (entropy) of its broadcast key message is at least $O(n)$ [13], where n is the number of privileged

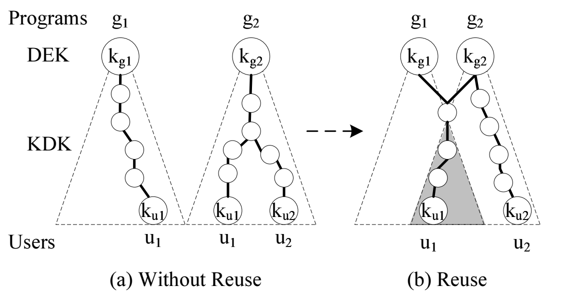


Figure 2: Key Tree Reuse

users. Zero-message scheme[14, 15] does not require the broadcast server to disseminate any message in order to generate a common key. But it only has resilience against coalitions of k non-privileged users, and requires every user to store $O(klog(k)log(n))$ keys. Hence, these approaches are not suitable in our system. 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 does not support join, which is crucial in our system. In self-healing schemes [7, 17, 18], group members can also recover the session key by combining information from any key distribution broadcast preceding and following a lost packet. Since this paper is not focused on recovery of lost keys, these schemes will be incorporated in our system in future works.

2.3. Problem Statements

For security, data packets are encrypted before being broadcast. In this paper, we present a *scalable*, *efficient* and *secure* key management scheme to broadcast keys to hundreds and thousands of users. To address the scalability, LKH is mostly used in the literature. Hence, an intuitive solution is to use a key tree for each program as shown in Figure 2(a).

However, Figure 2(a) is not an efficient solution. As depicted, if user u_1 subscribes two programs g_1 and g_2 simultaneously, he needs to manage two sets of keys. As a result, directly applying LKH may be costly. Hence, by exploiting the *overlap* nature among subscription groups, *key tree reuse* (**KTR**) scheme is proposed to reduce such cost in key management as shown in Figure 2(b).

The idea of key tree reuse is to allow multiple programs to share a sub-key-tree (the gray triangle in Figure 2(b)). Its advantage is clear: with key tree reuse, each user in $g_1 \cap g_2$ only needs to manage one set of keys to handle 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 the traditional LKH approach. Our approach is also very general in that it does not require two programs that share a sub-key-tree to have any semantic relation.

The security requirements considered in group key management include group key secrecy, forward secrecy, backward secrecy and key independence [5]. However, the security requirement of backward secrecy considered in this paper is different from that in the literature. In our system, if a new user knows old keys but cannot decrypt previous broadcast data items, it is still secure. For this difference, **past confidentiality**, instead of backward secrecy, is used for security analysis in this paper, i.e. a member added at time t does not have access to any key that can be used to decrypt program data before time t.

Of course, the security requirements can be easily solved by using the rekey operations in any LKH-based scheme. However, it is not the most efficient way. Assume u_1 quits g_1 in Figure 2(b) at t. u_1 will move out of the gray triangle to a leaf node in g_2 at t. In LKH, all keys that u_1 will no longer use (to ensure forward secrecy) and all keys that u_1 will use (to ensure backward secrecy) will be changed. However, notice that k_{g_2} actually has no need to be changed, since u_1 is already allowed to receive data in g_2 and past confidentiality is ensured. As analyzed later, it is even possible that some KDKs in g_2's tree do not need to be changed either. Hence, eKTR is proposed to carefully inspect every key in order to reduce the rekey cost to the minimum without compromising broadcast security.

3. Key Tree Reuse

3.1. Key Forest

KTR can be modeled as a *key forest* (see Figure 3), where all keys form a directed and acyclic graph. In the graph, each tree, structured as the LKH, represents a set of users with the same subscription. There are two types of trees: *traditional tree* and *reuse tree*. A traditional tree stands for a set of users who subscribe only one of the programs, and its root is the program's DEK. A reuse tree stands for a set of users who subscribe the same multiple programs, and its root is not a DEK, but a KDK from which these programs share the same tree.

All roots form a *root graph*. If a tree is reused by several programs, its root is connected to the roots of these programs. Because a traditional tree is not reused by any other program, its root does not have any outgoing link. Obviously, the key forest has the following properties: (a) any user only belongs to one tree in the key forest, and his individual key is the leaf node of the tree; (b) any leaf node in a tree has only one path to the root following the upward links; (c) departing from any root and following the directed links in the graph, one always ends at the roots of traditional trees. These features ensure that a user will not subscribe and pay for the same program multiple times.

An example of a key forest is given in Figure 3. The

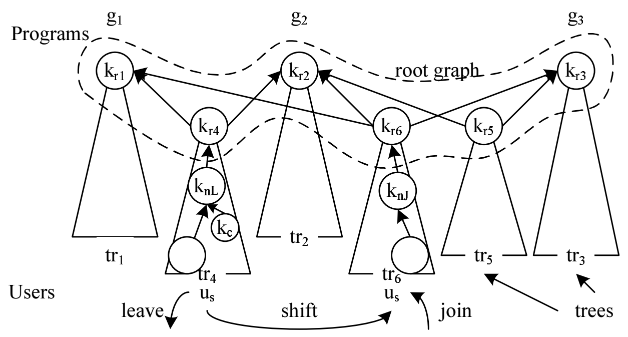


Figure 3: Key Forest

broadcast server provides three programs, g_1, g_2 and g_3. Some users subscribe only one of the programs, while other users may subscribe both g_1 and g_2, or both g_2 and g_3, or all three programs. In this figure, no user subscribes both g_1 and g_3. Hence, there are 6 trees, $tr_1, \ldots, tr_6$, each of which stands for a type of subscription. The root graph in this figure depicts the reuse structure. For example, since tree tr_5 is reused by both g_2 and g_3, its root r_5 is connected to both r_2 and r_3. Finally, the keys, k_{r_1}, k_{r_2} and k_{r_3} are DEKs to encrypt data items, while all other keys are KDKs.

3.2. Rekey Operations

In this paper, rekey consists of three major operations, **join**, **leave** and **shift**. When users subscribe programs, or stop or change current subscription, new keys are distributed in the broadcast channel as the data items. In the key forest, user activities consist of adding the user to a tree (join), removing the user from a tree (leave), or changing the user from one tree to another tree (shift). Hence, these operations are tree oriented instead of program oriented. Table 1 lists the operations and their corresponding user events. Note that the discussion of rekey operations in this paper only considers individual user events. The batch rekey operations for simultaneous user events can be found in [9, 10], and be incorporated in our scheme.

In the key forest, two types of paths are formed before rekey operations. When a user leaves a tree, we form a **leave path**, which consists of keys that the user will no longer use. When a user joins a tree, we form an **enroll path**, 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 LKH, a path always starts from a leaf node of the user and ends at the DEK of his program. Differently, a path in KTR may end at multiple DEKs, if the corresponding tree is reused by multiple programs. For example, in Figure 3, when a user shifts from tr_4 to tr_6, the leave path consists of n_L and r_4, and the enroll path consists of n_J, r_6, r_1, r_2 and r_3. Note that in this example, the user subscribes one more program, but the rekey operation involves two trees. Of course, if LKH is adopted,

Table 1: Rekey Operations

Rekey	User events
Join a tree	A user has not subscribed any program. Then, • He subscribes only one program. • He subscribes multiple programs.
Leave a tree	A user has subscribed several programs. Then, • He quits all current programs.
Shift among trees	A user has subscribed several programs. Then, • He subscribes one more programs. • He subscribes a few more programs. • He quits one of the current programs. • He quits a part of the current programs. • He changes a part of the current programs to another set of programs.

only one tree is involved. Hence, a shift event in KTR may introduce extra rekey cost. However, our simulation shows that this extra cost is minor regarding the overall advantages of KTR.

3.3. Rekey Packets

In rekey operations, we identify all the keys that need to be changed and distributed upon a user event. To broadcast the new keys, the server should first compose rekey packets. The rekey packets are scheduled before the data items that are encrypted with the new keys. The server also needs to provide key indices so that the users can know which key needs to be received. With a simple index approach, a rekey packet is composed as a sequence of key items $[..., \underline{(i,j)\{k'_i\}_{k_j}}, ...]$. Each underlined item in the packet is a pair of key indices and an encrypted key, where k'_i is the new value of k_i, and k_j is the KDK to encrypt k'_i. They are indexed as (i, j).

In this paper, we take the standard LKH approach to encrypt the new key k'_i with k_j. If k'_i is in an enroll path, k_j is 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. The order of the key items is bottom up in the key forest so that a receiver changes the keys from leaves to roots following the paths. Readers can refer to [1, 2] for examples of rekey packets.

4. Enhanced Key Tree Reuse

Although directly applying LKH's rekey operations in KTR can obviously ensure forward secrecy and backward secrecy, it is not the most efficient way. As discussed in Section 2.3, in some situations, a key in the enroll path can be sent to the user without any change so that rekey cost can be reduced to the minimum without compromising past confidentiality. Hence, we provide a generalized analysis and propose eKTR to achieve the minimum rekey cost as follows. Note that, in eKTR, we keep the LKH's rekey operation for keys in the leave path to ensure forward secrecy, but propose another approach to ensure past confidentiality in the enroll path.

Algorithm 1: Update of refresh and renew spots

```
1 If the key is in an enroll path and must be
  changed, a renew spot is added to all its spot
  series;
2 If the key is in a leave path, a renew spot must be
  added to all its spot series;
3 If the key's parent key is in a leave path, a refresh
  spot must be added to its spot series associated
  with the program;
```

4.1. Preliminary of eKTR

The critical problem of eKTR is to identify the minimum set of keys in the enroll path that must be changed to ensure past confidentiality. We name these keys as **critical key**. In another word, eKTR only changes critical keys in the enroll path, while leaving other keys unchanged, and thus the rekey cost can be reduced to the minimum. To formally present the approach to identify the critical keys, we first define the following terms.

Definition 1 *Refreshment, $\delta(k_j, t_\alpha; k_i, t_\beta)$: a rekey message broadcast in the form of $\{k_j(t_\alpha)\}_{k_i(t_\beta)}$ at time t_α, and $t_\beta \leq t_\alpha$.*

In the refreshment, k_j is a key in the leave path, and its value is changed to a new one at t_α when a leave or shift event happens. k_i is the KDK to encrypt the new k_j. k_i is also a child key of k_j, and its value started at t_β, which is prior to t_α, i.e. $t_\beta \leq t_\alpha$.

A refreshment may contain information threatening past confidentiality. For example, assume a user joins the tree of program g_m whose DEK is k_m at t_c, and k_1 is in the enroll path. Assume the user finds the following refreshments from all previous broadcast rekey messages in the time order $t_1 \leq t_2 \leq ... \leq t_m < t_c$:

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

If the server sends k_1 to the user without any change, the user can first decrypt k_2, and then iteratively decrypt k_m's value prior to t_c. Note that backward secrecy is compromised at this stage, but past confidentiality is not yet. We say, past confidentiality at t_m is compromised, if the user can use the old k_m to decrypt the data items that are broadcast before t_c and after t_m but the user is not supposed to get. On the contrary, past confidentiality at t_m is not compromised, if the user either cannot find such a sequence of refreshments to obtain the old k_m or has already legitimately obtained the old k_m and the data items before t_c and after t_m.

Definition 2 *Renew spot of a key: the time point when the value of the key is changed.*

Definition 3 *Refresh spot of a key: the time point when the key is used as the KDK to encrypt its parent key in a refreshment.*

Algorithm 2: Update of revive spots

```
  Function UPDATEREVIVE($k, t, t_c, g_v$);
1 let V be the set of all child keys of $k_0$ ;
2 for $k_i \in V$ do
3     let $t_a$ be the latest renew spot of $k_i$;
4     If $t_a \leq t$ and $\delta(k, t; k_i, t_a)$ exists, add $t_c$ as a
      revive spot to $k_i$'s spot series associated with
      $g_v$ and UPDATEREVIVE($k_i, t_a, t_c, g_v$);
  endfor
```

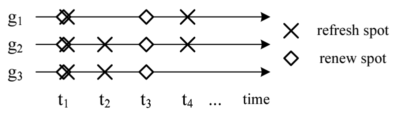
Definition 4 *Revive spot of a key: the time point when (1) the data encryption key of this key's associated program is changed to a new value, and (2) there is a sequence of previous broadcast refreshments that are potentially harmful to past confidentiality.*

The sequences of refresh, renew and refresh spots form *spot series* in the time order. The main feature is that if a key is reused by multiple programs, the key have multiple spot series, each of which is associated with one program (see examples in Section 4.2).

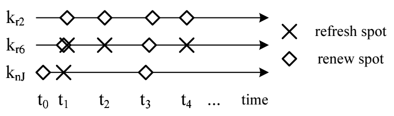
The server updates refresh and renew spots according to Algorithm 1. Each update is triggered by the corresponding user event. After the update of refresh and renew spots, the server updates revive spots according to Algorithm 2. Starting from a renewed DEK, the server iteratively updates revive spots of all keys in the forest. Assume the algorithm is checking a key k, which is renewed at t. It first selects a k_i from all k's child keys. Then, it checks if k_i has such a refreshment $\delta(k, t; k_i, t_a)$. Past confidentiality at t_c will not be threatened, if k_i is renewed after t, i.e. $t_a > t$, or no $\delta(k, t; k_i, t_a)$ exists. Otherwise, i.e. $t_a \leq t$ and $\delta(k, t; k_i, t_a)$ exists, the revive spot of t_c is added to k_i. The algorithm then continues to update k_i's child keys.

4.2. Examples of Spots

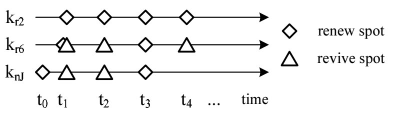
In this part, we demonstrate the spot series from two different dimensions. First, a key has multiple spot series associated with its programs. Figure 4(a) depicts the spot series of k_{r_6} in the key forest of Figure 3. Because k_{r_6} is shared by three programs (i.e. g_1, g_2 and g_3), it has three spot series, and each series is represented by a line in Figure 4(a). Assume, 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



(a) Spot series of key k_{r_6}



(b) Spot series regarding program g_2



(c) Revive spots regarding program g_2

Figure 4: Spot series

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 4(b), 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}.

In Figure 4(c), we draw the revive spot series of the three keys associated with g_2 based on Figure 4(b). 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 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.

4.3. Solution of eKTR

Definition 5 *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. Similarly, a key have multiple ages if it is shared by multiple programs, and each age is associated with one program.*

According to the 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 6 *Age of a subscription: the time interval between the current time to the latest beginning time the user is in a program. Similarly, if a user subscribes multiple programs, he has one subscription age for each program.*

According to the 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.

In the following, we give a generic method to identify critical keys in the enroll path and reduce the rekey cost. Assume 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 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.*

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 5, k's value has never been changed since $t - ka_i$ and was revived at $t - ka_i$. According to Definition 4, 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 6, 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.

Proof of the necessary condition: The necessary condition is equivalent to 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

Algorithm 3: Process of eKTR in the server

```
1 If a join or shift event happens, find the enroll
  path, which has the minimum number of critical
  keys according to TCK, and change all the
  critical keys in the enroll path;
2 If a leave or shift event happens, find the leave
  path, and change all keys in the leave path;
3 Compose and broadcast corresponding rekey
  messages;
4 Update renew and refresh spots according to
  Algorithm 1;
5 Update revive spots and ages according to
  Algorithm 2 and Definition 5;
```

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 6, 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, 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 5, 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 $(ka_i, ua_i)_{k,u}$ satisfies $ka_i \leq ua_i$ at current time t, and k is sent to u without any changed.

Therefore, to ensure past confidentiality, k in an enroll path needs to be changed, if and only if at least one pair of $(ka_i, ua_i)_{k,u}$ satisfies $ka_i > ua_i$ at current time t. Based on the theorem of critical key (**TCK**), eKTR works as in Algorithm 3 upon a user event.

4.4. Examples of critical keys

Corollary 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 the user joins the tree, his subscription ages for all the programs sharing this tree are 0. Hence, if the age

Table 2: Key management schemes

Schemes	*key tree reuse*	*critical key*
eKTR	Y	Y
KTR	Y	N
eLKH	N	Y
LKH	N	N

of a key in the enroll for this program is greater than 0, the key is older than the user's subscription. According to Theorem 1, the key needs to be changed before being distributed to the user.

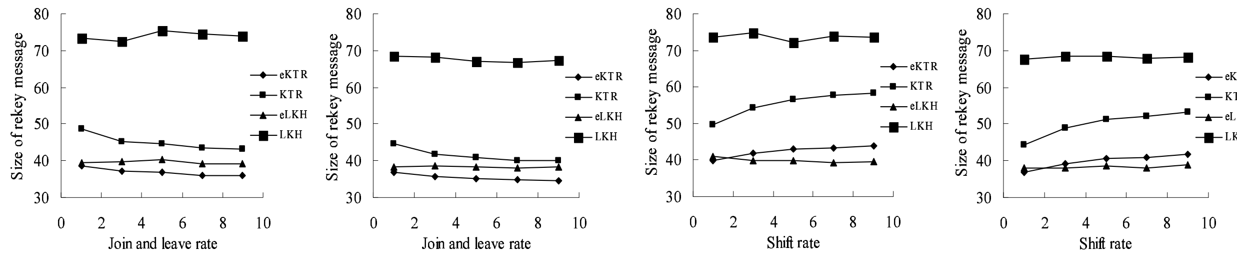
For example, consider a user event that a user u_2 joins tree tr_4 at time t_2 in Figure 3. Assume that before t_2, another user u_1 shifts from tree tr_4 to tr_6 at t_1, and no other event happens between t_1 and t_2. At t_1, assume k_{n_L} and k_{r_4} are in the leave path. At t_2, assume k_{n_L} and k_{r_4} are in the enroll path. Now, we decide which key is a critical key at t_2 according to Corollary 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, 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 5, 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, since 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 example also indicates that even the traditional LKH approach is used in our system, it is not necessary to change all keys in the enroll path when a user subscribes the broadcast data services.

5. Performance Evaluation

Key tree reuse and *critical key* are the two important ideas we developed to improve the performance of key management in wireless broadcast systems. In this section, we conduct a simulation-based performance evaluation to examine their impacts. Table 2 lists four schemes representing different solutions which may or may not adopt these two ideas. The names of the schemes are self-explained. If key tree reuse is adopted, key management is based on the key forest as illustrated in Figure 3; 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). Note that the well-known LKH is used as a base line (i.e. neither key tree reuse



(a) Case I (b) Case II (c) Case III (d) Case IV

Figure 5: Average rekey message size per event

Table 3: 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

nor critical key is adopted). We expect eKTR to be the best scheme, since it adopts both ideas.

5.1. Simulation settings

We assume that the server provides 5 programs. In our experiments, The key forest consists of 31 trees when key tree reuse is adopted. Among them, 26 trees are reused by multiple programs. Each tree represents a unique option of subscriptions. We also assume that there are 10000 users (on average) subscribing the services. All three user events (i.e. join, shift and leave) are modeled as independent Poisson processes. The rates (frequencies) of join and leave are the same in order to control the total number of users remaining at a constant level (i.e. around 10000). We vary the shift rate and the join/leave rate 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 key tree reuse. Such an event is mapped into several user events in schemes that do not adopt key tree reuse. 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).

Two important performance metrics, *average rekey message size per event* and *average number of decryption per event per user* are employed in this evaluation. The former, measured as the number of encryptions $\{*\}_k$ in the rekey message, is used to represent the communication cost. The latter measures the computation cost and power consumption on a mobile device, since the device needs to decrypt new keys from rekey messages.

Four test cases are generated for the evaluation based on major subscriptions and major events (summarized in Table 3). In Case I and Case III, 80% of the users subscribe multiple programs and the other 20% users only subscribe one of the programs; while in Case II and Case IV, 20% of the users subscribe multiple programs and the other 80% subscribe 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 rate at 1.

5.2. Average Rekey Message Size Per Event

We first evaluate performance of the key management schemes in terms of average rekey message size, by fixing the shift rate to 1 and varying the join/leave rate (x-axis) from 1 to 9 as shown in Figure 5(a) and (b). The schemes adopting key tree reuse and/or critical keys significantly outperform the LKH. eKTR is obviously the best solution, while eLKH performs better than KTR. This can be explained as follows. For the schemes that do not adopt key tree reuse, a user needs to join or leave multiple trees when he subscribes or unsubscribes these programs. However, for the schemes that do adopt key tree reuse, the user only needs to join or leave one tree for multiple programs. Thus, the reuse schemes significantly reduce the rekey message size when join/leave is the primary event. Nevertheless, while key tree reuse reduces the cost of rekey messages, the major improvement comes from the adoption of critical keys. Based on our experimental results, by adopting only key tree reuse, KTR reduces the rekey message size to around 60% to 67% of LKH. By adopting critical keys alone, nevertheless, eLKH reduces the rekey message size to around 55% of LKH. This result also validates our claim that many keys in the enroll path do not need to be changed[1].

Next, we evaluate performance of the key management schemes by fixing the join/leave rate to 1 and varying the shift rate (x-axis) from 1 to 9 as shown in Figure 5(c) and (d). In this set of experiments, eLKH turns to be the best solution, although eKTR is trailing closely[2]. As discussed in Section 3.2, this is because extra overhead of shift is introduced in the reuse scheme when a user quits or adds some but not all of his subscribed programs. Hence, eLKH is the scheme of choice when shift is the major event. Similarly, by comparing CASE

1 As a matter of fact, over all the experiments, only around 22% keys in the enroll path need to be changed.

2 eKTR actually performs better when the shift rate is small.

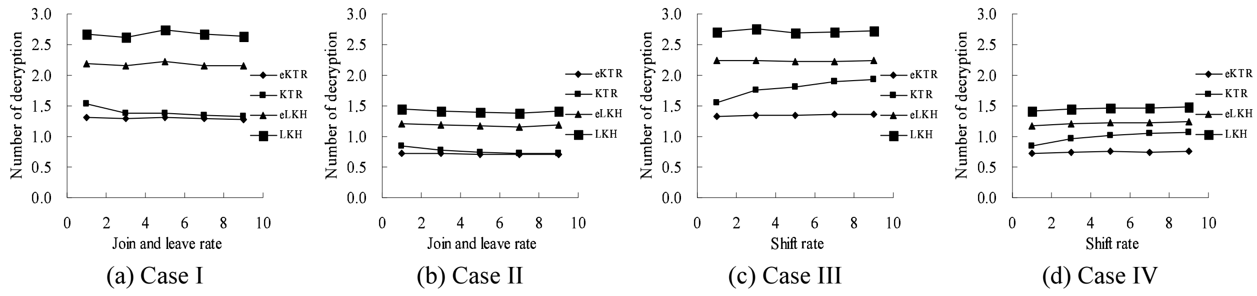


Figure 6: Average number of decryption per event per user

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 5 also shows that eKTR and KTR are more sensitive to the type of major user events than eLKH and LKH. The average rekey message size in eKTR and KTR grows as the shift rate increases and drops as the join/leave rate increases. On the contrary, the average rekey message size in eLKH and LKH remains almost flat.

5.3. Average Number of Decryption Per Event Per User

Power consumption and computation cost are two primary concerns of mobile users. We use the average number of decryption 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 6 shows that the reuse schemes (i.e. eKTR and KTR) is better than eLKH and LKH. In all cases, eKTR is always the best solution! The number of decryption in eKTR is around 48% of that in LKH, while the number of decryption in eLKH is only around 81% of that in LKH. The number of decryption in KTR is between the curves of eKTR and eLKH, and drops as the join/leave rate increases and increases as the shift rate increases. This result clearly validates our intuition that the idea of key tree reuse can effectively reduce the number of keys a user needs to hold for his subscriptions.

Obviously, the adoption of critical keys also has a major impact on reducing user's decryption cost (although not as significant as the key tree reuse). As shown, eKTR results in less decryption than KTR does, and so does eLKH in comparison with LKH. When critical keys is adopted, not only less keys in the enroll path need to be changed, but also less users are affected by the activities of other users in the same tree. This is particularly evident when we compare KTR and eKTR in Case III and Case IV (see Figure 6 (c)-(d)). If critical key is not adopted (i.e. KTR), all users in a tree need to perform some extra decryption when a user shifts to the tree. However, when critical key is adopted (i.e. eKTR), it is possible that no key in a tree needs to be changed when a user shifts to the tree (in order to unsubscribe some of his programs). In this situation, no user in the tree needs to decrypt any key.

Figure 6 also shows that the user subscription pattern has a great impact on the average number of decryption. The average number of decryption 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 more programs, it is more likely that he will be affected by other user activities.

5.4. Summary

Note that KTR only adopts the key tree reuse, while eLHK only adopts the critical keys. Thus, the comparison of KTR and eLKH brings some very good insights regarding to the impacts of our two proposed ideas on the performance metrics. Our experiments in Section 5.2 shows that critical key is a more important factor in reducing the rekey message size than the key tree reuse (see Figure 5). On the other side, key tree reuse is a dominant factor in reducing the number of decryption (see Figure 6).

In summary, eKTR combines the advantages of both ideas of key tree reuse and critical key, and thus is the most efficient scheme for key management in secure wireless broadcast systems. It has a light communication overhead (i.e. its average rekey message size per event is the least or close to the least among all schemes). Meanwhile, it 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 among all solutions).

5.5. Other Concerns

Wireless communication is inherently unreliable. Moreover, users may move out of service areas or intentionally shut down their mobile devices to save power. To simplify our discussion and analysis, we make assumptions that the broadcast channel is reliable and users will not miss keys. Nevertheless, many researches on helping users to recover their keys [10, 17, 16, 19] have been done, which can also be added into our schemes. Furthermore, in our broadcast model, a user can occasionally use the uplink channel (or via wirelines) to request the lost keys. To recover lost keys, overhead may be introduced into the system. However, this overhead is more relevant to the frequency a user loses keys rather than the security of eKTR, and hence is out of scope of our research.

We also notice that the definition of refreshment is semantically related to the form of rekey message when a user leaves a tree. In the situation where a rekey method [9, 10, 17] other than the LKH is used, the definition of refreshment needs to be changed. Note that a refreshment should always help a user to figure out keys in his upper levels. Hence, the refreshments in different rekey methods are equivalent regarding to whether they contain information that potentially compromises past confidentiality.

Finally, if the broadcast server uses critical key to improve the performance of rekey operations, it will incur computation cost on the server side. The worst case is that the server needs to check all keys in order to find the best enroll path with the minimum number of critical keys. The computation complexity in the worst case is thus $O(n)$, where n is the number of users. In our simulations where the broadcast system has 10000 users, the server (a computer with 2.6GHz CPU and 760MB RAM) takes less than $14ms$ for each event with the eKTR scheme.

6. Conclusion

In this paper, we investigate the issues of key management in support of *secure* wireless data broadcast services. To the best knowledge of the authors, this is the first research conducted in the field of subscribe-publish based wireless data broadcast services. We propose eKTR as a scalable, efficient and secure key management approach in the broadcast system. We use the key forest model to formalize and analyze the key structure in our broadcast system. eKTR let multiple programs share a single tree so that the users subscribing these programs can hold less keys. In addition, we propose an 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 eKTR. Our simulation shows that eKTR 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. Built upon the important result obtained thus far, we plan to extend our work to combine the self-healing schemes to help users recover lost keys and adopt the batch schemes to allow simultaneous user events.

References

[1] C. K. Wong, M. Gouda, and S. S. Lam, "Secure group communications using key graphs," in *ACM SIGCOMM*, 1998, pp. 68–79.

[2] D. Wallner, E. Harder, and R. Agee, "Key management for multicast: Issues and architectures," *IETF RFC 2627*, 1999.

[3] Y. Sun and K. R. Liu, "Scalable hierarchical access control in secure group communications," in *IEEE Infocom*, 2004.

[4] S. Mittra, "Iolus: a framework for scalable secure multicasting," in *ACM SIGCOMM*, vol. 277-288, 1997.

[5] Y. Kim, A. Perrig, and G. Tsudik, "Simple and fault-tolerant key agreement for dynamic collaborative groups," in *ACM CCS*, 2000, pp. 235–244.

[6] 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.

[7] C. K. Wong and S. S. Lam, "Keystone: A group key management service," in *International Conference on Telecommunications*, 2000.

[8] M. Moyer, J. Rao, and P. Rohatgi, "Maintaining balanced key trees for secure multicast," *draft-irtf-smug-key-tree-balance-00.txt*, 1999.

[9] 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.

[10] 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.

[11] J. Snoeyink, S. Suri, and G. Varghese, "A lower bound for multicast key distribution," in *IEEE Infocom*, vol. 1, 2001, pp. 422–431.

[12] M. Luby and J. Staddon, "Combinatorial bounds for broadcast encryption," in *Advances in Cryptology, Eurocrypt*, 1998, pp. 512–526.

[13] M. Just, E. Kranakis, D. Krizanc, and P. v. Oorschot, "On key distribution via true broadcasting," in *ACM CCS*, 1994, pp. 81–88.

[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] 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.

[18] 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.

[19] D. Liu, P. Ning, and K. Sun, "Efficient self-healing group key distribution with revocation capability," in *ACM CCS*, 2003, pp. 231–240.