ELK, a New Protocol for Efficient Large-Group Key Distribution*

Adrian Perrig Dawn Song J. D. Tygar
University of California Berkeley
{perrig,dawnsong,tygar}@cs.berkeley.edu

Abstract

Secure media broadcast over the Internet poses unique security challenges. One problem access control to a large number of subscribers in a public broadcast. A common solution is to encrypt the broadcast data and to disclose the decryption key to legitimate receivers only. However, how do we securely and efficiently establish a shared secret among the legitimate receivers? And most importantly, how can we efficiently update the group key securely if receivers join or leave? How can we provide reliability for key update messages in a way that scales up to large groups?

Recent research makes substantial progress to address these challenges. Current schemes feature efficient key update mechanisms assuming that the key updates are communicated reliably to the receivers. In practice, however, the principal impediment to achieve a scalable system is to distribute the key updates reliably to all receivers. We have designed and implemented ELK, a novel key distribution protocol, to address these challenges with the following features:

- *ELK features perfectly reliable, super-efficient member joins.*
- *ELK uses smaller key update messages than previous protocols.*
- *ELK features a mechanism that allows short hint messages to be used for key recovery allowing a tradeoff of communication overhead with member computation.*
- *ELK proposes to append a small amount of key update information to data packets, such that the majority of receivers can recover from lost key update messages.*
- *ELK allows to trade off security with communication overhead.*

*This publication was sponsered in part by the Defense Advanced Research Projects Agency under DARPA contract N6601-99-28913 (under supervision of the Space and Naval Warfare Systems Center San Diego), by the National Science foundation under grant FD99-79852, and by the United States Postal Service under grant USPS 102592-01-Z-0236. Views and conclusions contained in this document are those of the authors and do not necessarily represent the official opinion or policies, either expressed or implied of the US government or any of its agencies, DARPA, NSF, USPS.

1 Introduction

This paper introduces *ELK*, an efficient, scalable, secure method for distributing group keys. ELK has widespread applications, such as access control in streaming multimedia broadcasts.

A common solution for controlling access to the broadcast information is to encrypt the data and to distribute the secret decryption key (group key) only to the legitimate receivers. The general approach is to use a central server for key management. Key management is complicated by dynamic groups, where members may *join* and *leave* at any time. Members should only be able to decrypt the data while they are members of the group, and so the key server needs to update the group key on member join and leave events.

Changing the key for large groups in a scalable, robust, and efficient manner is particularly challenging [14, 34, 37]. A solution must deal with arbitrary packet loss, including lost key update messages. In general, previous approaches have built on reliable multicast (which has high communication overhead in large-scale use) or queries to a central server to request retransmission of keys (which introduces substantial load for a central server.)

In general, a system designer faces a variety of tradeoffs between scalability, security, efficiency, and reliability. Here is the scenario we consider:

- We put a premium on scalability. We are interested in situations where we have widespread video or audio streaming over a network to a large number of receivers.
- We are interested in moderate security. We consider the case when many receivers use commodity hardware, e.g., PCs without tamper-resistant hardware. Note that the absence of tamper-resistant hardware limits the ultimate security of keys [42, 43].

- Since the key server broadcasts key update messages to all the group members, the communication overhead can be prohibitively high for large dynamic groups. Due to the continuous increase in computation power, we design ELK to trade off computation for lower communication overhead. More concretely, we design a member join protocol that does not require any broadcast but requires that the server computes a one-way function on all keys in each time interval. ELK also introduces *hints*, a technique which makes key updates smaller but requires receiver computation.

- We must provide high reliability, but we do not assume that we can build a reliable multicast system.

We have designed and implemented *ELK*, a novel key distribution protocol that addresses these issues. In particular, ELK addresses reliability for key update messages, unlike prior work. ELK also allows a content provider to directly trade-off security and communications efficency.

Here are some features of ELK:

- ELK does not depend on reliable multicast.

- ELK uses smaller key update messages than previous protocols.

- ELK features perfectly reliable, super-efficient member joins.

- ELK addresses reliability by using short hint messages. This improves reliability and allows a trade-off of communication overhead with member computation.

To study the application of ELK, we consider two different types of information goods: low-cost goods and “perishable” information goods. We apply new ways to assess the desired level of security. For low-cost goods we consider the system to be secure as long as the cost of the attacker to break the key is larger than the cost of the information. For perishable information goods we require that the minimum time to break the key surpasses the lifetime of value of the data.

2 Security Requirements for Group Key Distribution

We consider dynamic groups where users can join or leave the group at any time. The main security properties of a group key management system for dynamic groups are:

1. *Group Key Secrecy* – guarantees that it is computationally infeasible for an adversary to discover any group key.

2. *Forward Secrecy* – (not to be confused with Perfect Forward Secrecy or PFS in key establishment protocols) guarantees that a passive adversary who knows a contiguous subset of old group keys cannot discover subsequent group keys. This property ensures that a member cannot learn about the new group keys after it leaves the group.

3. *Backward Secrecy* – guarantees that a passive adversary who knows a subset of group keys cannot discover preceding group keys. This property ensures that when a new member joins the group, he cannot learn about the previous group keys.

These properties are commonly used in secure group communication. Steiner, Tsudik, and Waidner give a more formal definition [31].

3 Notation and Background

We use the following notation.

- To encrypt message M with key K we write $\{M\}_K$.

- To concatenate the messages M_1 and M_2 we write $M_1 \mid M_2$.

- In the following description, we use many different pseudo-random functions (PRFs) with varying input and output lengths. The notion of a PRF family was proposed by Goldreich, Goldwasser, and Micali [12]. Instead of defining one PRF for each purpose, we define a family of PRFs that use key K on input M of length m bits, and output n bits: $\mathsf{PRF}^{\langle m \to n \rangle} : \mathcal{K} \times \{0,1\}^m \to \{0,1\}^n$. We write $\mathsf{PRF}^{\langle m \to n \rangle}_K(M)$.

- The function $\mathsf{LSB}^{\langle n \rangle}(M)$ returns the n least significant bits of M (assuming that M is longer than n bits).

- We use a number of key derivation functions to ensure that the keys and arguments in various places are independent. The following keys are derived from K_i as follows:
  $K_i^\alpha = \mathsf{PRF}^{\langle n \to n \rangle}_{K_i}(1)$, $K_i^\beta = \mathsf{PRF}^{\langle n \to n \rangle}_{K_i}(2)$,
  $K_i^\gamma = \mathsf{PRF}^{\langle n \to n \rangle}_{K_i}(3)$, and $K_i^\delta = \mathsf{PRF}^{\langle n \to n \rangle}_{K_i}(4)$

  (Note that $1, 2, 3$ and 4 are arguments to the PRFs. This key derivation is solely to ensure independence of keys for security reasons. However, to simplify understanding of the protocols we advise ignoring the Greek superscripts on a first reading of this paper.)

4 Review of Previous Key Distribution Schemes

4.1 Setting

In broadcast key distribution, we assume a central key distribution server that can authenticate and authorize individual receivers. The model is that the receiver wishing to decrypt the broadcast content contacts the key server by unicast and requests the decryption key. The key server authenticates the receiver with a standard authentication protocol and sets up a secure channel (offering confidentiality, integrity, authenticity). The server sends key information to the client, which consists of a group key to decrypt the content, as well as a set of member-specific keys for key management purposes.

The broadcast information is encrypted with the group key to achieve confidentiality and access control. To ensure forward and backward secrecy after receivers join or leave the broadcast (defined in section 2), the key server broadcasts encrypted key updates that only the legitimate members can decrypt.

To analyze the overhead of key distribution schemes, we consider the following resources. (Note that the total number of receivers is N).

- **Receiver storage**. Each receiver stores a number of member-specific keys and one group key. Since the number of keys to store is usually small ($O(\log(N))$), receiver storage is not an issue.

- **Key server storage**. The key server stores all the member-specific keys. In current schemes the number of member-specific keys is about $2N$. For groups with millions of receivers, the key server storage can be on the order of multiple megabytes. Researchers have investigated schemes that reduce the server storage overhead [7, 16]. However, in this work we assume that the key server has sufficient storage, because a broadcaster that sends data to millions of paying subscribers should have ample key storage.

- **Receiver and key server computation**. Since the processing speed of workstations continues to increase, computation overhead is not as important.

- **Bandwidth** is the most constrained resource. In particular, key update information broadcast to all receivers needs to be as small as possible. Since we assume abundant storage and computation resources, our goal is to trade off computation or storage to lower communication cost. We put a premium on broadcasts, however we also try to limit unicasts.

4.2 Review of Logical Key Hierarchy (LKH)

To ensure forward and backward secrecy, the group key needs to be updated and distributed whenever a member joins or leaves the group. ELK is based on a key tree, and extends the *logical key hierarchy* (also called LKH [14, 34, 36, 37]) and *one-way function tree* (OFT [2]) approaches to achieve an efficient and secure key distribution system. We include a brief review of LKH below. In LKH, a *key distribution center* (or *key server*) maintains a key tree which will be used for group key updates and distribution. Figure 1 shows a sample key tree. Each node in the tree represents a cryptographic symmetric key. The key distribution center associates each group member with one leaf node of the tree and the following invariant will always hold: Each group member knows all the keys from its leaf node up to the root node, but no other node in the key tree. We call the set of keys that a member knows the *key path*. Since all members know the key at the root node, that key is used as the group key, that we denote with K_G. For illustration, the key path of member M_2 in figure 1 is the nodes associated with the keys $\{K_5, K_2, K_1\}$. When a member joins the group, it receives all the keys on the path from its leaf node up to the root from the key distribution center, sent over a secure channel.[1] When a member leaves the group, all the keys that the member knows, including the group key and its key path, need to be updated. The main reason for using such a key tree is to efficiently update the group key if a member joins or leaves the group.

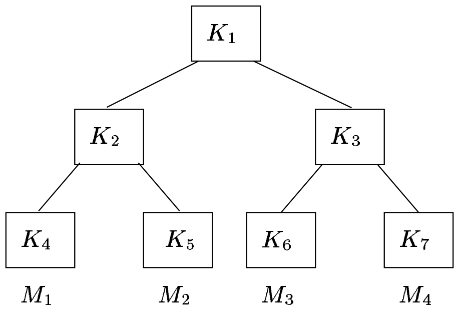

Figure 1. Sample hierarchical key tree

If a member joins the group, the key server authenticates the member and assigns it to a leaf node of the key tree. The key server will then send all the keys on the key path to the member. To preserve backward secrecy all the keys that the new member receives need to be independent from any previous keys (the new member should not be able

[1]The issues of member authentication and secure channel setup are orthogonal to the main thrust of this paper and we assume that secure mechanisms are used.

to decrypt traffic that was sent before it joined the group). Hence, the key server replaces all keys on the new member's keypath with fresh, random keys and sends each of these new keys to the group on a "need to know" basis. We illustrate this protocol with an example. Assume the setting depicted by figure 1 and for simplicity we assume that a new member M_4 joins the group. Assuming that the last leaf of the tree is empty, the key server places the new member M_4 at that leaf, chooses new the keys on M_4's key path and sends K_7', K_3', and K_1' to M_4 over a secure link. To update the key paths of the previous members, the server broadcasts the following key update message to the group: $\{K_3'\}_{K_6}, \{K_1'\}_{K_3'}, \{K_1'\}_{K_2}$. Member M_3 needs to update keys K_3 and K_1 on its key path. Since M_3 knows K_6, it can decrypt the first part of the key update message and recover K_3', and as soon as it knows K_3' it can decrypt the new group key K_1'. Members M_1 and M_2 both know K_2, so they can decrypt the new group key K_1' from the final part of the key update message. Below we show how to improve on this join protocol so no broadcast message is necessary.

The group leave, however, is more difficult to perform efficiently. The challenge is to replace the current group key such that only the legitimate members receive the new key but the leaving member does not. In fact, all keys that the leaving member knows need to be changed to ensure forward secrecy. The keys are replaced sequentially from the leaf up to the root key. This protocol is best explained with an example. We assume the group that figure 1 shows, where member M_3 leaves the group. The key server updates the keys K_1, K_3 and generates the new keys K_1', K_3'. It then broadcasts the message $\{K_3'\}_{K_7}, \{K_1'\}_{K_3'}, \{K_1'\}_{K_2}$. Member M_4 knows K_7 and can hence decrypt and obtain K_3', which allows it to obtain K_1'. Members M_1 and M_2 know K_2 and so they can directly obtain the new group key K_1'. Leaves are efficient because they only require updating $\lceil \log(N) \rceil$ keys, where N is the number of group members and assuming that the key tree is balanced.

5 Reliability for Key Update Messages

When members join or leave a group, the key server updates the group key and broadcasts a key update message to the group. If a group member does not receive the key update message, it will not be able to decrypt the subsequent messages encrypted with the new group key. With the exception of the recently proposed Keystone protocol by Wong and Lam [38], previous systems addressed the problem of lost key updates only marginally. Previous schemes assume to recover from lost key updates through the following mechanisms:

1. A naive approach is to let members request a key update by unicast from the key server. Clearly the naive unicast recovery mechanism does not scale, although it can be used in conjunction with other techniques as a fallback recovery mechanism. In fact, both ELK and the Keystone protocol use unicast as a fallback mechanism. We sketch such a recovery protocol in Appendix A.

2. Another approach is to replicate key update packets (multi-send). Although replication is a powerful method to achieve robustness, it is well known that packet loss in the Internet is correlated [26]. This implies that key update packets sent in close succession risk loss if the first one is lost. A strategy which separates redundant packets would cause the client to wait for replicated key update messages when it receives data that it cannot decrypt.

3. Reliable multicast schemes, such as SRM [11] or STORM [40] may be used to achieve reliable delivery of key updates. These schemes add substantial complexity and might not scale to TV-size audiences. Furthermore, these systems are not designed for robustness in an adversarial environment, and hence opportunities for denial-of-service attacks exist. Similarly, a reliable group communication toolkit, such as those used in small-group key agreement protocols, such as TOTEM [23], or HORUS [33] are prohibitively expensive and would not scale to large groups.

4. Wong and Lam [38] use forward-error correction or error-correcting codes for key update packets [17, 27]. More specifically, the idea is to use a scheme such as Rabin's IDA [28], Reed-Solomon [29] codes, or Digital Fountain codes [5] to split up a key update packet into n packets, and when the receiver gets any m packets it can reconstruct the key update. In such a scheme the receiver needs to receive sufficient packets to reconstruct the desired information. Moreover, since packet loss is correlated, packets that are sent in close succession may all be dropped [26]. Wong and Lam assume statistical independent loss, which does not cover correlated packet loss (e.g. due to temporary congestion) [38].

In this work, we use a combination of new mechanisms to achieve reliable key updates. Our work is motivated by the following observations:

1. Member joins are free in our model if no broadcast is necessary. Our member join protocol does not require any broadcast message. However, if a large number of members join concurrently, the key server may need to distribute a message of length $\lceil \log_2(N) \rceil$ bits to encode the location of the joining node, where N is the number of group members. Since member join events

usually require no broadcast information, no information can get lost.

2. In the case of a key update message after a member leaves, half the members only need a single key in the key update message (assuming our key tree is balanced). Similarly, one quarter of the members only need two keys to update their key path. In general $1/2^i$ of the members only need i keys to update their key path. Another way of viewing this is that sending i keys will help $1 - 2^{-i}$ of the members to update their key path. We call the keys that help more members the *maximum impact keys* (MIK).

3. Most key management protocols separate key update messages and encrypted data packets. The receiver must receive both of them to read encrypted data. In ELK, we add the key update directly to the data packets. Since space in data packets is limited, we can at most add a small amount of key update information. Previous protocols had lengthy key update messages, so they could not use this approach.

ELK features a method to compress key updates by trading off key update message size and receiver computation. The resulting key update is small enough that the sender can piggyback it in data packets. The details are described below.

With these mechanisms in place, the majority of group members can recover from a lost key update if they receive the hints in a data packet. The remaining small fraction needs to contact the key server through unicast.

6 ELK: An Efficient Large-Group Key Distribution Protocol

ELK stands for Efficient Large-Group Key distribution protocol. We describe ELK in this section. We first describe the basic key update mechanism, followed by the join and leave protocols. We then analyze the security of ELK, and discuss ELK's advantages. Later we will show how ELK allows member joins without requiring broadcasts. (Only unicasts to adjacent members of the new member are required. See Section 6.2.1 for details.)

6.1 Basic ELK Mechanisms

In ELK all members are at leaves in the logical key hierarchy. ELK is composed of two basic mechanisms:

- Key update
- Key recovery (through hints)

From now on we assume that the key length is n bits.

6.1.1 Child Contribution for Parent Key Update

As we discuss in section 3, the server updates the keys of the key tree when members leave the group. We use a new key update protocol in ELK, where the left and right child keys contribute to update the parent key. This approach is similar in nature to the OFT protocol [2], but our construction allows small hints that allow legitimate members to reconstruct the key without the key update, as we shall see in the next section.

We consider the case where we want to update a key K that has the two child keys K_L and K_R. The new key K' is derived from K and contributions from both children. The left child key K_L contributes n_1 bits to the new key, which are derived by a pseudo-random function using key K_L and applied to K. We call the left contribution $C_L = \mathsf{PRF}^{\langle n \to n_1 \rangle}_{K_L^\alpha}(K)$ (n_1 bits long). Similarly, C_R is n_2 bits long and is derived from the right child key K_R and K as follows: $C_R = \mathsf{PRF}^{\langle n \to n_2 \rangle}_{K_R^\alpha}(K)$. The right child key K_R contributes n_2 bits, hence C_R is n_2 bits long. We concatenate the two contributions to form a new key of length $n_1 + n_2$: $C_{LR} = C_L|C_R$. To compute K' we compute a pseudo-random function with C_{LR} as the key and the previous key K as the data: $K' = \mathsf{PRF}_{C_{LR}}(K)$. Without loss of generality we assume that $n_1 \leq n_2$. Since the security of this key distribution scheme is at most $O(2^n)$ (because of the n bit key length), we have $n_1 + n_2 \leq n$.

The key update message needs to contain enough information that the members on the left who know K_L can recompute K', as well as the members on the right who know K_R. The left members can derive C_L themselves, but they need C_R. Hence the key update message contains $\{C_R\}_{K_L}$, which is n_2 bits long. Similarly, for the members on the right, the key update contains $\{C_L\}_{K_R}$, which is n_1 bits long.

The details of this key update are listed in the box labeled Procedure 1. This construction allows us to construct compact hint messages, that enable the legitimate receivers to reconstruct the updated key, as we will see below.

6.1.2 Key Recovering Using Hints

Instead of broadcasting the key update message that has length of $n_1 + n_2$ bits, legitimate members who know K and either K_L or K_R can also recover the new key K' from a hint that is smaller than the key update message, by trading off computation for communication. If we assume that a member can perform 2^{n_1} computations, we can construct a smaller key update that we call a *hint*.

Consider first the right-hand members that know K and K_R. They can derive the right contribution C_R that is n_2 bits long. If they would also have a checksum, they could brute force the missing n_1 bits of K' from the left side con-

The left-hand contribution is $C_L = \mathsf{PRF}^{\langle n \to n_1 \rangle}_{K^\alpha_L}(K)$
The right-hand contribution is $C_R = \mathsf{PRF}^{\langle n \to n_2 \rangle}_{K^\alpha_R}(K)$
Hence $C_{LR} = C_L \mid C_R$
The new key becomes $K' = \mathsf{PRF}^{\langle n \to n \rangle}_{C_{LR}}(K)$
Recall from section 3 that $K^\alpha_L = \mathsf{PRF}^{\langle n \to n \rangle}_{K_L}(1)$ and $K^\beta_L = \mathsf{PRF}^{\langle n \to n \rangle}_{K_L}(2)$. The purpose of this key derivation is to make K^α_L and K^β_L independent.
To update the key, the server broadcasts
$\{\mathsf{PRF}^{\langle n \to n_2 \rangle}_{K^\alpha_R}(K)\}_{K^\beta_L}, \{\mathsf{PRF}^{\langle n \to n_1 \rangle}_{K^\alpha_L}(K)\}_{K^\beta_R}$.
This key update message has a length of $n_1 + n_2$ bits. The members who know K_L can derive K^β_L, decrypt $\mathsf{PRF}^{\langle n \to n_2 \rangle}_{K^\alpha_R}(K)$ from the key update, and compute K'. The same applies to the members who know K_R.

Procedure 1: Key update

tribution. The hint message contains the *key verification* $V_{K'}$, which is derived from the new key $V_{K'} = \mathsf{PRF}_{K'}(0)$ and has a length of n_3 bits. The right-hand members compute the following candidate keys. For each possibility for C_L, they compute C'_{LR} and obtain a candidate key $\tilde{K} = \mathsf{PRF}_{C'_{LR}}(K)$. The member verifies the candidate key by checking against the key verification $\mathsf{PRF}_{\tilde{K}}(0)$ to see if it equals $V_{K'}$.

If $n_1 = n_2$, the left-hand members compute the key in the same way as the right-hand members. In the usual case, however, $n_1 < n_2$; and the left-hand members need to obtain $n_2 - n_1$ bits of the right contribution, so that they still only need to brute-force n_1 bits. So the hint message also contains the least $n_2 - n_1$ bits of C_R, encrypted with K_L. With this help, the left-hand members compute the 2^{n_1} possibilities for C'_{LR} and obtain a candidate key $\tilde{K} = \mathsf{PRF}_{C'_{LR}}(K)$. They verify the candidate key by checking the key verification $\mathsf{PRF}_{\tilde{K}}(0)$ to see if it equals $V_{K'}$.

(The above description is simplified, see the box labeled Procedure 2 for the details.)

It is clear that the correct key is output by this procedure. A problem is that the procedure might deliver more than one candidate key, in which case some of them are false positives. If the key verification is also n_1 bits long, we might expect to receive one false positive key next to the correct key. If a member recomputes multiple keys, it will receive an additional key on each level, since a false positive key produces on average just one false positive key, but the correct key will most likely result in the correct key for the next level along with another false positive key. To prevent this, we set $n_3 > n_1$. Generally setting $n_3 = n_1 + 1$ results in "half a false positive key" on average, which works well in practice.

The hint message is composed of the key verification and the partial right contribution:
$V_{K'} = \mathsf{PRF}^{\langle n \to n_3 \rangle}_{K'^\gamma}(0)$, $\{\mathsf{LSB}^{\langle n_2 - n_1 \rangle}(\mathsf{PRF}^{\langle n \to n_2 \rangle}_{K^\alpha_R}(K))\}_{K^\beta_L}$.
Recall that $K'^\gamma = \mathsf{PRF}^{\langle n \to n \rangle}_{K'}(3)$. The key reconstruction is slightly different for the members in the left and right sub tree. Recall that the updated key is
$K' = \mathsf{PRF}^{\langle n \to n \rangle}_{C_{LR}}(K)$ and
$C_{LR} = \mathsf{PRF}^{\langle n \to n_1 \rangle}_{K^\alpha_L}(K) \mid \mathsf{PRF}^{\langle n \to n_2 \rangle}_{K^\alpha_R}(K)$.
The members on the left know K_L, derive K^β_L, and can hence decrypt part of the right key's contribution $\mathsf{LSB}^{\langle n_2 - n_1 \rangle}(\mathsf{PRF}^{\langle n \to n_2 \rangle}_{K^\alpha_R}(K))$. Now they know all but n_1 bits to compute $\tilde{K} = \mathsf{PRF}^{\langle n_1 + n_2 \to n \rangle}_{C'_{LR}}(K)$, with
$C'_{LR} = \mathsf{PRF}^{\langle n \to n_1 \rangle}_{K^\alpha_L}(K) \mid x \mid \mathsf{LSB}^{\langle n_2 - n_1 \rangle}(\mathsf{PRF}^{\langle n \to n_2 \rangle}_{K^\alpha_R}(K))$.
The members can exhaustively try all 2^{n_1} possibilities for x and verify the resulting guess $\tilde{K}$ by using the key verification: $\mathsf{PRF}^{\langle n \to n_3 \rangle}_{\tilde{K}^\gamma}(0) \overset{?}{=} V_{K'}$. If they match, the key is a candidate.
The members on the right know K_R and they can compute n_2 bits of C_{LR}, so they only need to exhaustively try 2^{n_1} combinations for x: $\tilde{K} = \mathsf{PRF}^{\langle n_1 + n_2 \to n \rangle}_{C'_{LR}}(K)$, where $C'_{LR} = x \mid \mathsf{PRF}^{\langle n \to n_2 \rangle}_{K^\alpha_R}(K)$. They also use the same hint information to verify the validity of the key.

Procedure 2: Key recovery from hint

The advantage of the hint is that it is $2n_1 - n_3$ bits shorter than the key update (In general, $n_3 = n_1 + 1$ and hence the hint is $n_1 - 1$ bits shorter than the full key update). In the ideal case $n_1 = n_2$ and then the hint is only half the size of the key update. However, n_2 is determined by the security parameter, which we discuss further in section 7. We discuss the security in Appendix B.

6.2 The ELK Key Distribution Protocol

The previous subsection introduces the mechanisms we use to construct ELK. We now use these mechanisms to describe ELK in detail. We show how the protocol works for member join and leave events. For the following protocol description, we assume that a large number of members are present in the group (we do not discuss the boundary cases when only a few group members are present).

Even though group key distribution protocols allow users to join or leave the group at any time, in general it is considered a good practice to aggregate join and leave requests that occur in one time interval into one group key update [30]. Also as we will see below, the key update message is smaller when the key server aggregates multiple membership events. Therefore we assume that in our system the key server divides the time up into intervals. The key server aggregates all membership events that occur within one time interval into one group key update. The duration of the aggregation interval is application specific and we do not discuss it here.

6.2.1 Member Join Event

In a member join event, the key server assigns the new member to a node in the key tree and the new member receives all the keys on the path from its leaf node to the root. To preserve backward secrecy, all the keys that the new member receives must be independent of previous group keys. Looking at independence from a computational perspective, we require that it be computationally infeasible for the new member to derive previous group keys. In most previous schemes, the updated keys need to be broadcast to the affected group members.

Since the key server broadcasts key update messages to all the group members, the communication overhead can be prohibitively high for large dynamic groups. Due to the continuous increase in computation power, we design ELK to trade off computation for lower communication overhead. To support efficient member join events, we propose a novel approach where no broadcast messages are needed but requires that the server computes a one-way function on all keys in each time interval. In ELK the entire key tree is updated in each time interval using the following procedure. To update a key K_i in the key tree, the new key K'_i is derived by $K'_i = \mathsf{PRF}^{\langle n \to n \rangle}_{K^\delta_i}(K_G)$, where $K^\delta_i = \mathsf{PRF}^{\langle n \to n \rangle}_{K_i}(4)$ and K_G is the current group key. To update the group key we use the derivation $K'_G = \mathsf{PRF}^{\langle n \to n \rangle}_{K^\delta_G}(0)$. Each member can update its key path independently in each time interval. Hence no broadcast messages are needed for a member join event. Since the computation of the PRF is efficient, the computational overhead of the receiver is negligible. The overhead at the server is larger, but still tractable. First, the server can pre-compute the future keys of the tree in a low-priority background process. A second approach is to recompute keys on the fly based on need. The major advantage of this approach is that no broadcast information is necessary when members join.

Even though no broadcast messages are necessary when members join, the key server might still need to send a few unicast messages as some members might be moved to new locations in the key tree as new nodes are added to the key tree. In the rest of this section, we first describe the complete join protocol for single members join and give an example.

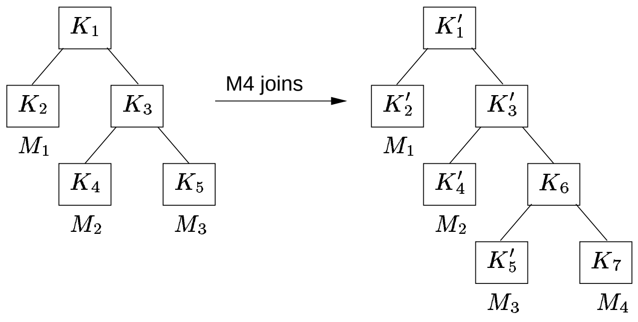

Figure 2. Member join event

Protocol 1. *Single Member Join*

1. *The key server updates all keys K_i in the key tree: $K'_i = \mathsf{PRF}^{\langle n \to n \rangle}_{K^\delta_i}(K_G)$, and the new group key is $K'_G = \mathsf{PRF}^{\langle n \to n \rangle}_{K^\delta_G}(0)$.*

2. *If an empty leaf node is available, the server assigns a new random key to the leaf node and sends it over a secure channel to the new member along with the updated keys on the key path. Thus the join event is done, and no more exchanges or broadcasts are necessary.*

3. *If no leaf node is available, the key server assigns the new member M to a new leaf node N_M of the key tree and assigns it a new random key K_M.*

4. *The key server picks a node N_j of the key tree to insert the new member. Assume that the key at that node is K_j. The server demotes the node N_j and generates a new parent node N_P for the leaf N_M and node N_j. The node N_j becomes the left child, and the node N_M the right child of N_P. The key value of the parent becomes $K_P = \mathsf{PRF}^{\langle n \to n \rangle}_{K^\delta_j}(1)$.*

5. *The key server sends the new member all updated keys on its key path from the leaf node up to the root over a secure channel.*

6. *The key server sends the joining location to the members that are below node N_j, which allows them to independently update their own key paths. In the general case, N_j is a leaf node and the key server unicasts the message to that member. If a larger number of members are present, the server includes N_j in the key update message, which takes at most $\lceil \log_2(N) \rceil$ bits.*

Figure 2 shows an example join event where member M_4 joins the group an the server decides to insert it at the leaf node of M_3. We illustrate the steps of the ELK join protocol on this example. In the first step, the server updates all keys in the key tree. Step 2 does not apply since no empty leaf nodes are available. In step 3 the key server generates the new leaf node and assigns it a new random key K_7. In step 4, the server decides to merge M_4 to the node of K'_5, and generates the parent node K_6 of K'_5 and K_7. The server computes the new key: $K_6 = \mathsf{PRF}^{\langle n \to n \rangle}_{K^{\delta}_5}(1)$. In step 5 the server sends M_4 the message $\{K'_1, K'_3, K_6\}_{K_7}$. In step 6, the server sends the joining location of the new member to M_3 by unicast, which tells M_3 to update its key tree and to compute K_6.

(Multiple member join events work equally well. Several possibilities exist to deal with them. Here is one way. If the members are placed into empty leaf nodes no overhead is generated besides registering new members with the key server. If no empty leaf nodes exist, the key server can first generate a smaller key tree with the new members, and join that tree to one node of the current group key tree. In this case the server only needs to communicate the location of a single node to the members that live below the joining node.)

6.2.2 Member Leave Event

The member leave event is more complicated than the member join event, because all the keys that the leaving member knows need to be replaced with new keys that the leaving member must not be able to compute (forward secrecy, see Section 2). The key server uses the child contribution scheme outlined in Section 6.1.1 to update the keys on the path from the leaf node of the leaving member up to the root. The server broadcasts a key update message containing the updated keys and also attach hint messages to data packets to enable key recovery in case the key update message is lost.

Protocol 2. *Member Leave*

1. *The server deletes the leaf node corresponding to the leaving member, as well as the parent node of the leaf node, and promotes the sibling node.*

2. *All remaining nodes on the key path of the leaving member need to be updated. For each of these keys, the update procedure for key K_i is as follows. The new key is*
$K'_i = \mathsf{PRF}^{\langle n \to n \rangle}_{C_{LR}}(K_i)$, *with*
$C_{LR} = \mathsf{PRF}^{\langle n \to n_1 \rangle}_{K^{\alpha}_{il}}(K_i) \mid \mathsf{PRF}^{\langle n \to n_2 \rangle}_{K^{\alpha}_{ir}}(K_i)$, *where K_{il} and K_{ir} are the left and right child keys, respectively.*

3. *The server broadcasts the key update message of all keys that were updated in the previous step. Hence, for each key K'_i the update message contains* $\{\mathsf{PRF}^{\langle n \to n_1 \rangle}_{K^{\alpha}_{il}}(K_i)\}_{K^{\beta}_{ir}}, \{\mathsf{PRF}^{\langle n \to n_2 \rangle}_{K^{\alpha}_{ir}}(K_i)\}_{K^{\beta}_{il}}$.

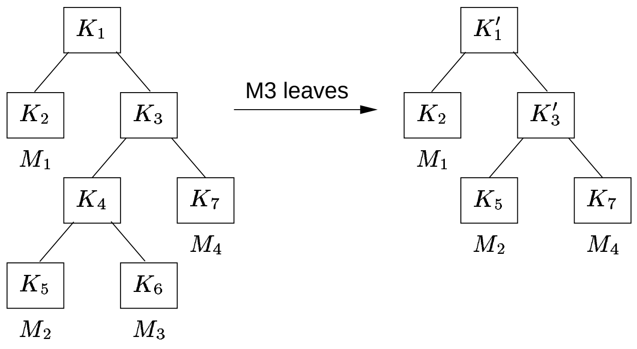

Figure 3. Member leave event

4. *The server attaches the following hint message to data packets. For each new key K'_i the server may send* $\mathsf{PRF}^{\langle n \to n_3 \rangle}_{K'^{\gamma}_i}(0), \{\mathsf{LSB}^{\langle n_2 - n_1 \rangle}(\mathsf{PRF}^{\langle n \to n_2 \rangle}_{K^{\alpha}_{ir}}(K_i))\}_{K^{\beta}_{il}}$.[2]

This is clearer in an example. See Figure 3 and the box labeled Example 1.

6.2.3 Multiple Member Leave Events

Above, we write that there are several good ways to realize multiple member join events. Multiple member leave events have some subtlety though, so we sketch the approach in slightly greater detail.

In case multiple members leave in the same interval, the key server aggregates all the leaving members and creates a joint leave key update message. ELK can aggregate the concurrent member leave events particularly well and provides (in addition to current savings) a 50% savings over OFT [2][3] for keys when both children change. The reason for OFT's inefficiency is that if both child keys change, OFT needs two key updates (one for each child), whereas ELK only needs one. ELK factors fresh contributions from both child keys into the parent key on a leave key update, regardless on whether the children changed. The size of the key update message in ELK is $(a - j)(n_1 + n_2)$, where a is the number of updated keys and j is the number of leaving members. Since the number of updated keys is always less than the sum of all keys in each key path, it is always advantageous to aggregate multiple member leave events. In

[2]It seems that a pre-computation attack is possible here, since an attacker could pre-compute the image $\mathsf{PRF}^{\langle n \to n_3 \rangle}_{K}(0)$ for all the keys K. In this case, when the server publishes $\mathsf{PRF}^{\langle n \to n_3 \rangle}_{K'^{\gamma}_i}(0)$ the adversary can look up the $\approx 2^{n-n_3}$ pre-images that are candidates for the key. In practice, however, n is larger than 64 bits, foiling such an attack. Note that the size of ELK key updates remains the same regardless of the length n of keys.

[3]We discuss OFT in Section 8. Like OFT, we derive keys using a binary tree with members represented at leaves.

Consider the example in Figure 3. Member M_3 leaves the group. In step 1, the key server deletes the nodes that correspond to the keys K_4 and K_6. The server promotes the node of K_5. In step 2, the server updates the keys on the key path of the leaving member K_3 and K_1.
The server computes the new key K_3' as follows:
$K_3' = \mathsf{PRF}_{C_{LR3}}^{\langle n \to n \rangle}(K_3)$, with
$C_{LR3} = \mathsf{PRF}_{K_5^\alpha}^{\langle n \to n_1 \rangle}(K_3) \mid \mathsf{PRF}_{K_7^\alpha}^{\langle n \to n_2 \rangle}(K_3)$.
To update key K_1 the key server computes
$K_1' = \mathsf{PRF}_{C_{LR1}}^{\langle n \to n \rangle}(K_1)$ and
$C_{LR1} = \mathsf{PRF}_{K_2^\alpha}^{\langle n \to n_1 \rangle}(K_1) \mid \mathsf{PRF}_{K_3'^\alpha}^{\langle n \to n_2 \rangle}(K_1)$.
In step 3 the server broadcasts the key update, which contains the following:
$\{\mathsf{PRF}_{K_5^\alpha}^{\langle n \to n_1 \rangle}(K_3)\}_{K_7^\beta}, \{\mathsf{PRF}_{K_7^\alpha}^{\langle n \to n_2 \rangle}(K_3)\}_{K_5^\beta}$ for K_3', and
$\{\mathsf{PRF}_{K_2^\alpha}^{\langle n \to n_1 \rangle}(K_1)\}_{K_3'^\beta}, \{\mathsf{PRF}_{K_3'^\alpha}^{\langle n \to n_2 \rangle}(K_1)\}_{K_2^\beta}$ for K_1'.
Finally, subsequent data packets contain the following hint (step 4):
$\mathsf{PRF}_{K_1'^\gamma}^{\langle n \to n_3 \rangle}(0), \{\mathsf{LSB}^{\langle n_2 - n_1 \rangle}(\mathsf{PRF}_{K_3'^\alpha}^{\langle n \to n_2 \rangle}(K_1))\}_{K_2'^\beta}$,
$\mathsf{PRF}_{K_3'^\gamma}^{\langle n \to n_3 \rangle}(0), \{\mathsf{LSB}^{\langle n_2 - n_1 \rangle}(\mathsf{PRF}_{K_7^\alpha}^{\langle n \to n_2 \rangle}(K_3))\}_{K_5^\beta}$

Example 1: Member leave protocol

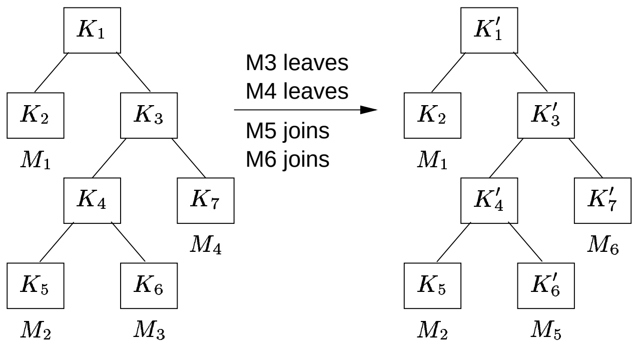

Figure 4. Member leave event

contrast, the size of the OFT key update message is $(a-1)n$ bits.

We illustrate multiple member leave events with an example. Assume the setting of Figure 4. For this example we assume that the leaving member nodes are not collapsed, because the key server replaces the leaving member nodes with new members. If members M_3 and M_4 both leave, and new members M_5 and M_6 take their spot, the keys K_4, K_3, and K_1 need to be updated. If the member leave events are processed sequentially, the update message for M_3 leaving is $3(n_1+n_2)$ bits long, and the message for M_4 is $2(n_1+n_2)$ bits long. If the server aggregates leaves, the message is only $3(n_1+n_2)$ bits long.

6.3 Security Analysis

A sketch of the security analysis appears in Appendix B. There we show the following observations hold:

- With overwhelming probability, a passive adversary needs to perform $\Omega(2^n)$ operations to brute-force an ELK group key.
- With overwhelming probability, an active adversary needs to perform $\Omega(2^n)$ operations to brute-force any ELK group key preceeding the time it joins the group.
- After the active adversary leaves the group, with overwhelming probability it needs to perform $\Omega(2^{n_1+n_2})$ operations to derive the new ELK group key.

We also show that pre-computation does not reduce the effort to brute-force a later ELK group key.

6.4 Advantages

ELK provides advantages over previous solutions for multicast group key distribution protocols. The join protocol uses key server computation to achieve member joins that do not require any broadcast message, hence greatly improving the scalability. Other advantages are the reduced size of group key updates, as well as the further reduced hint messages that allow legitimate members to recover from lost key updates. The hint messages can drastically reduce the number of members that need to contact the key server to recover from lost key update messages.

Because of the small footprint of the hint message, each data packet may carry a hint with it. Hence, if the receiver missed a key update, but receives the corresponding data packet, it will be able to recover the group key from the hint with high probability and decrypt the message.

Since the encrypted data without the decryption key is useless, as well as a key without corresponding data is useless, combining the two seems natural. This linking of the hint with the message, however, is a powerful mechanism that ELK makes possible, due to the small footprint of the hint.

Another innovation of ELK is to distribute partial key tree update information. Key update information provides diminishing returns. Hence the idea is to disseminate a small amount of information that enables the majority of the members to recover from a lost update message. The remaining members (a small fraction) can be dealt with on an individual basis.

7 Applications and Practical Issues

In this section we discuss the choice for the parameters for ELK and arguments for its security. The parameters we discuss are: n, n_1, n_2, and n_3, (the number of bits of the key, the left contribution, the right contribution, and the size of the key verification, respectively), and the number of levels of keys that are added to the hint. The choice of these parameters is driven by the tradeoff between efficiency (communication and computation) and security.

7.1 Security Model

Our attacker model assumes a "reasonable" attacker who breaks a system by breaking the weakest link. The main application of this work is a broadcast environment where the receivers do not have tamper-resistant security devices. This implies that a user has access to the decryption keys, because they are stored in memory. Hence, an attacker can always obtain the current group key by subscribing to the service. From another perspective, how secret can a group key be if it is shared by 10^5 members? We judge our key distribution protocol as secure if it is considerably more difficult and expensive to get the key by breaking the key distribution protocol than by other means.

7.2 System Requirements

Besides the security requirements, we also have system requirements. We want to have have a key update protocol that has low computation and communication overhead. For the hint messages we require that at least 98% of all receivers can recover the key from the hint message. This implies that the hint message includes the keys from at least 6 levels. The hint to a key of level i may help a fraction of 2^{-i} members, in the case of a single member leave event (assuming that the key tree is balanced). Since $\sum_{i=1}^{6} 2^{-i} = 0.984$, 6 levels are sufficient to reach 98% of the members.

Furthermore we require that the key reconstruction be faster than requesting a key update by unicast from the key server. We assume that such a request message may take around 200 ms. Hence the requirement is that a fast (e.g. 800 MHz Pentium) workstation could reconstruct 6 keys in less than 200 ms.

As we discuss in the implementation section, our test workstation computes 5,000,000 PRF functions per second.

7.3 Parameters

The first parameter we choose is the group key size n. In this instance we assume that ELK is used for a medium-security environment, and we choose the key size of 64 bits.

Next, we want to achieve that the key reconstruction from a hint takes at most 200 ms, which allows us to compute n_1. In the worst case, a member needs to reconstruct 6 keys. Considering that it can compute 5,000,000 PRFs per second, this leaves 166,666 PRF computations per key. For each key guess, two PRF computations are necessary, one to compute the key, and the other to verify the key with the hint. Therefore, we chose to use 16 bits for n_1, which translates into $2 \cdot 2^{16}$ = 131,072 computations for each key. As we discussed previously, if the hint message is also 16 bits long, the member expects to get one additional false positive key per level. This implies that the member who computes 6 keys will end up with 7 candidate keys for the group key, which requires 6 times more work to compute the group key. For this reason, we make the key hint $n_3 = 17$ bits long, which reduces the false positives. (Table 2 shows the number of candidates for different levels. For our recommended parameters, the number of false positives averages around 0.5.)

We now compute the number of bits for n_2 based on a sample scenario.

Protecting Perishable Information Goods

Perishable information goods loose their value with time. An example is a live "pay-per-view" media transmissions such as a sports event video feed. Consider an example of perishable data that we want to protect for 10 minutes. The security requirements dictate that an attacker needs at least 1000 computers to break the key in less than 10 minutes.

We assume that the attacker has fast machines that compute 10^7 PRFs per second. The 1000 computers can hence compute $3.6 \cdot 10^{13}$ PRFs in 10 minutes. Since we know that the attacker needs at least $2 \cdot 2^{n_1+n_2}$ operations to break the key, we can derive $n_1 + n_2 \approx 44$, and $n_2 = 28$ bits. The key update per key is $n_1 + n_2 = 44$ bits long, and the total size per key of a hint is $n_2 - n_1 + n_3 = 29$ bits. Hence the savings of the hint message are 34%. For each key hint we also encode the position of the hint in the key tree with an additional bit (left or right). Hence the cost for 6 key hints is $6 * 30 = 180$ bits, which translates into 23 bytes.

7.4 Advantages

As computers get faster, the savings of ELK improve. We achieve the largest savings for the hint messages if $n_1 = n_2$, in which case the hint is only half the size of the key update. In this case the legitimate members perform 2^{n_1} operations, but the attacker cannot perform $2^{n_1+n_2} = 2^{2n_1}$ operations. Merkle used a similar argument to construct a public/private key encryption algorithm [20]. In his *Merkle Puzzle* system, he assumes that legitimate users can perform 2^n operations, but an attacker would need to perform

2^{n+m} operations to break the encrypted message. Brute-force search for a parameter was also used by Dwork and Naor to fight spam mail [10]. Similarly, Manber [18], and Abadi, Lomas, and Needham [1] used brute-force search to improve the security of the UNIX one-way password function.

The above examples demonstrate that ELK can trade off security and efficiency. It allows the content distributor to choose the desired level of security on a fine-grained scale and lower security directly translates into smaller key update messages. Note that previous protocols do not offer this feature. If one desires a lower security margin it is not safe to shorten the key length, because new attacks that exploit short key lengths become possible. For instance, an attacker may perform pre-computation and use memory lookup tables to break the short group key. ELK does not suffer from these attacks, as we show in the security analysis in appendix B.

8 Comparison and Related Work

Harney and Muckenhirn introduced GKMP [15], a centralized server approach that distributes group keys through unicast. We refer to this approach as Secure Key Distribution Center (SKDC). Mittra's Iolus aims for scalability through distributed hierarchical key servers [21]. Molva and Pannetrat involve routers on the multicast distribution path into the security [22].

The logical key tree hierarchy was independently discovered by Wallner et al. [34, 35], and by Wong, Gouda, and Lam [36, 37]. We call this algorithm the Logical Key Hierarchy (LKH). An optimization that halves the size of the key update message is described by Canetti et al. [6]. This optimization to LKH is called LKH+, which Harney and Harder described in more detail in their Internet Draft [14].

To reduce the overhead of join, the current approach is to simply compute a one-way function on each key that the new member obtains, which is proposed by the Versakey framework [8] and the LKH+ protocol [14].

Another method to halve the size of the key update message is the One-way Function Tree protocol (OFT) by Balenson, McGrew and Sherman [2, 19]. In OFT, the key of the node is derived from the two sibling keys. The protocol we present in this paper resembles the OFT approach.

Even though the LKH+ protocol greatly diminishes the overhead of a group key change to $O(\log(N))$ (where N is the number of group members), the constant key change with the resulting key update messages can still result in an unscalable protocol in large dynamic groups. If members join and leave frequently, the resulting key update traffic can overwhelm the group. Setia, Koussih, and Jajodia [30] attempted to solve this problem. They proposed periodic rekeying, which results in an aggregating members that join or leave during a short time interval. They do not address the issue of reliability of key updates, however. Their approach to scalability uses a hierarchy of key servers (similar to Iolus) that aggregate join and leave events. (Within one subgroup the Kronos protocol would minimize the communication overhead by using our ELK protocol.)

Briscoe designed the MARKS protocol [4]. MARKS is scalable and does not require any key update messages, but the protocol only works if the leaving time of the member is fixed when the member joins the group and so members cannot be expelled.

Trappe et al. [32] also observed that key updates should be distributed with the data. In fact, they propose that the key updates be embedded within the data, for example using techniques similar to watermarking. The focus of their work, however, is different. While the ELK key updates encode the same key as was used to encrypt the data, [32] proposed embedding future keys in current data.

Wong and Lam designed Keystone [38], which addresses the reliable delivery for key update messages, which is also the primary motivation of this work. Since most reliable multicast transport protocols do not scale well to large groups, they propose that the key server uses forward error correction (FEC) to encode the key update message. As long as the member receives a sufficient fraction of the key update packets, it can reconstruct the information. If too many packets are lost, the member uses a unicast connection to the key server to recover the missing keys. Since the authors assume independent packet loss, this scheme is quite effective. In practice, however, the packet loss in the Internet is correlated, which means that the probability of loss for a packet increases drastically if the previous packet was lost [3, 41]. This means that even though the key update packets are replicated, sending them in close succession introduces considerable vulnerability against correlated packet loss.

8.1 Comparison

In this work we focus on broadcast overhead, since we consider it to be the most important quantity. Unicast cost, memory overhead at the key server, and computation overhead are of lesser concern. (If a broadcasting company has 10^6 paying customers, it can also afford a server with sufficient memory to store the $2 \cdot 10^6$ keys.) By reducing the size of the key updates, one can have a higher level of redundancy (assuming that a constant fraction of the bandwidth is dedicated for key updates) which results in higher reliability.

Table 1 shows a comparison of the standard key distribution protocols. N is the number of users in the group and d is the height of the key tree. Assuming that the tree is balanced, we have $d = \lceil \log_2(N) \rceil$. The location information

	SKDC	LKH+	OFT	Keystone (binary)	ELK Full	ELK Hint (2^{-1})	ELK Hint ($1-2^{-i}$)
Single member join							
Broadcast size	Nn	d	dn	$2dn$	0	0	0
Multiple member join (j members)							
Broadcast size	jNn	dj	$a_j n$	$2a_j n$	0	0	0
Single member leave							
Broadcast size	$(N-1)n$	$(d-1)n$	dn	$2dn$	$(d-1)(n_1+n_2)$	n_2	in_2
Multiple member leave (j members)							
Broadcast size	$(N-j)n$	$(a_j-j)n$	$(a_j-1)n$	$2a_j n$	$(a_j-j)(n_1+n_2)$	$b_j n_2$	$c_j n_2$

Table 1. Comparison between current key distribution schemes. All quantities are in number of bits, and we do not account for the tree location information that needs to be passed along with each key. The parameters a_j, b_j, and c_j are explained in the text.

of a node in the tree also takes d bits. The quantity a_j determines the number of keys that change in the tree when j members join or leave.

The SKDC protocol is the simplest protocol, but since it is not based on key trees it clearly does not scale to large groups. The Keystone protocol is based on key trees, but since it does not incorporate recent developments to reduce the size of the key update messages it is more expensive than LKH+ or OFT. We can clearly see that ELK is the most efficient protocol, even in the case where ELK has the same security parameter as the other protocols ($n_1+n_2=n$). Our new join protocol does not require any broadcast message, since the entire tree changes in each interval. In particular for the case of j leaving members, ELK provides savings with a factor of a_j-j factor instead of the OFT's a_j-1 factor.

To display the overhead of the hint message, we list two cases: One where the hint helps half the members to reconstruct the group key, and the other that helps a fraction of $1-2^{-i}$ members. To simplify the table, we marked the size of the hint message as n_2[4] In the case when j members leave the group, the size of the hint message is more difficult to give. The expected number of keys that allows half the members to recover the group key is $b_j = \sum_{w=0}^{v-1} 2^w(1-(1-2^{-w})^j)$ with $v = \lceil -\frac{\log(1-2^{-1/j})}{\log(2)} \rceil$ The formula for the case where a fraction of $1-2^{-i}$ member wish to recover the group key from the hint is
$c_j = \sum_{w=0}^{v-1} 2^w(1-(1-2^{-w})^j)$ with $v = \lceil -\frac{\log(1-2^{-i/j})}{\log(2)} \rceil$

[4]If the hint size is $n_2-n_1+n_3$, and $n_3=n_1+1$, the hint size would actually be n_2+1.

9 Implementation

We implemented the ELK protocol. In this section, we discuss our choice for the cryptographic primitives and report measured performance numbers. We have not yet measureed the savings in a real multicast group communication environment.

9.1 PRF

As we pointed out in Section 7, the savings of the hint increases with the speed of the members. The only function that is relevant for the hint computation is the speed of the PRF, because the PRF is the only function that is used repeatedly to derive the lost key in the exhaustive search.

We use a MAC function to construct the PRF. For all the PRFs needed, the required input size is less or equal to n bits, and the output size is also always less or equal to n bits. In our application, we chose $n = 64$ bits. We compared the speed of a variety of MAC functions, HMAC with a hash function, and CBC-MAC based on a block cipher. We used the functions provided by the OpenSSL library [25], and the fast Rijndael (Advanced Encryption Standard) implementation provided by NIST [24]. The HMAC based on hash functions are slower than the fastest CBC-MACs. Rijndael performed well on our 800 MHz Pentium workstation with 1,200,000 MACs per second (the input, output, and key sizes are 128 bits). However, RC5 is faster with 5,000,000 MACs per second (with 64 bit input and output size). Since we do not need more than 64 bits input or output size, RC5 is over four times faster than Rijndael, and hence, we use one encryption with RC5 as our PRF function.

Path length	1	2	3	4	5	6
Reconstruction (ms)	18	42	51	102	124	153
Candidates	1.41	1.59	1.72	1.63	1.63	1.59
Member fraction	50%	25%	12.5%	6.25%	3.125%	1.5625%

Table 2. Reconstruction time and number of candidates for paths of varying length. The last row lists the fraction of members that recomputes the key path of the given length.

9.2 Encryption Function

To save space in our key updates and hints we cannot afford the data expansion caused by a block cipher. Therefore we used a stream cipher. Since the speed of the encryption is not as important as that of the PRF function and since we already use RC5 for the PRF function, we use RC5 in OFB mode as our stream cipher. We use the group key for the IV. This is secure because we never encrypt twice with the same key/IV pair.[5]

9.3 Results

We implemented ELK with the parameters of $n_1 = 16$ bits, $n_2 = 35$ bits, and $n_3 = 17$ bits to prevent an increase of false positive candidate keys. We report the performance results from the key reconstruction algorithm. Table 2 shows the average number of milliseconds to reconstruct a path of a certain length.

10 Conclusion

We summarize with some major contributions of ELK.

- ELK features smaller key updates than previous protocols. Most notably, ELK member join events do not require any data broadcast to current group members in the general case.
- ELK generates small hint messages that trades off communication overhead with member computation. These small hints enable legitimate receivers to derive a group key through computation in case they missed a key update message.
- ELK is one of the first protocols to provide reliability of key update messages without relying on reliable multicast protocols. Instead, ELK uses small key update footprints composed of hints carried in data packets. This approach allows the majority of the members to recover the new group key when the key update message is lost.

[5]An exception is the encryption of part of the hint message and part of the key update message. In this case, however, not only the key/IV pair is the same for both encryptions, but also the plaintext data.

11 Acknowledgments

We would like to thank David Wagner for his helpful feeback and fruitful discussions. We would also like to thank Nikita Borisov, Ran Canetti, Yongdae Kim, Radha Poovendran, and Gene Tsudik and for helpful discussions. We are also indebted to the anonymous referees for their detailed and insightful comments.

References

[1] Martin Abadi, T. Mark A. Lomas, and Roger Needham. Strengthening passwords, September 1997. SRC Technical Note 1997-033.

[2] D. Balenson, D. McGrew, and A. Sherman. Key management for large dynamic groups: One-way function trees and amortized initialization. Internet Draft, Internet Engineering Task Force, March 1999. Work in progress.

[3] M. Borella, D. Swider, S. Uludag, and G. Brewster. Internet packet loss: Measurement and implications for end-to-end QoS. In *International Conference on Parallel Processing*, August 1998.

[4] Bob Briscoe. MARKS: Zero side-effect multicast key management using arbitrarily revealed key sequences. In *First International Workshop on Networked Group Communication*, November 1999.

[5] John W. Byers, Michael Luby, Michael Mitzenmacher, and Ashutosh Rege. A Digital Fountain approach to reliable distribution of bulk data. In *Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 1998.

[6] R. Canetti, J. Garay, G. Itkis, D. Micciancio, M. Naor, and B. Pinkas. Multicast security: A taxonomy and some efficient constructions. In *INFOCOMM'99*, March 1999.

[7] Ran Canetti, Tal Malkin, and Kobbi Nissim. Efficient communication-storage tradeoffs for multicast

encryption. In Jacques Stern, editor, *Advances in Cryptology – EUROCRYPT '99*, number 1599 in Lecture Notes in Computer Science. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 1999.

[8] G. Caronni, M. Waldvogel, D. Sun, N. Weiler, and B. Plattner. The VersaKey framework: Versatile group key management. *IEEE Journal on Selected Areas in Communications*, 17(9), September 1999.

[9] Isabella Chang, Robert Engel, Dilip Kandlur, Dimitrios Pendarakis, and Debanjan Saha. Key management for secure Internet multicast using boolean function minimization techniques. INFOCOM 1999, September 1998.

[10] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In Ernest F. Brickell, editor, *Advances in Cryptology - Crypto '92*, pages 139–147, Berlin, 1992. Springer-Verlag. Lecture Notes in Computer Science Volume 740.

[11] S. Floyd, V. Jacobson, S. McCanne, C. G. Liu, and L. Zhang. A reliable multicast framework for lightweight sessions and application level framing. In *Proceedings of the ACM SIGCOMM 95*, pages 342–356, Boston, MA, August 1995.

[12] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM*, 33(4):792–807, October 1986.

[13] Li Gong. Enclaves: Enabling secure collaboration over the Internet. In *Proc. 6th USENIX Unix and Network Security Symposium*, 1996.

[14] H. Harney and E. Harder. Logical key hierarchy protocol. Internet Draft, Internet Engineering Task Force, April 1999. Work in progress.

[15] H. Harney and C. Muckenhirn. Group Key Management Protocol (GKMP) Specification / Architecture. Request for Comments RFC-2093 and RFC-2094, Internet Engineering Task Force, July 1997.

[16] Mingyan Li, Radha Poovendran, and C. Berenstein. Optimization of key storage for secure multicast. In *35th Annual Conference on Information Sciences and Systems (CISS)*, Johns Hopkins University, March 2001.

[17] F. J. MacWilliams and N.J.A. Sloane. *The Theory of Error-Correcting Codes*. North Holland, Amsterdam, 1977.

[18] Udi Manber. A simple scheme to make passwords based on one-way functions much harder to crack. *Computers and Security*, 15(2):171–176, 1996.

[19] David A. McGrew and Alan T. Sherman. Key establishment in large dynamic groups using one-way function trees, May 1998. `http://www.cs.umbc.edu/~sherman/Papers/itse.ps`.

[20] R. Merkle. Secure communication over insecure channels. *Communications of the ACM*, 21(4):294–299, 1978.

[21] Suvo Mittra. Iolus: A framework for scalable secure multicasting. In *ACM SIGCOMM*, September 1997.

[22] Refik Molva and Alain Pannetrat. Scalable multicast security in dynamic groups. In *Proc. 6th ACM Conference on Computer and Communications Security*, pages 101–112, Nov 1999.

[23] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, April 1996.

[24] NIST. Advanced Encryption Standard (AES) development effort. `http://csrc.nist.gov/encryption/aes/`, October 2000.

[25] OpenSSL. The OpenSSL project. `http://www.openssl.org`, 2000.

[26] V. Paxson. End-to-end Internet packet dynamics. *IEEE/ACM Transactions on Networking*, 7(3):277–292, June 1999.

[27] In V. S. Pless and W. Huffman, editors, *Handbook of Coding Theory*, volume 1, page 740. Elsevier, Amsterdam, Netherlands, 1998.

[28] M. O. Rabin. The information dispersal algorithm and its applications. In *Sequences: Combinatorics, Compression, Security and Transmission*, pages 406–419. Springer-Verlag, 1990.

[29] Irving S. Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8:300–304, 1960.

[30] Sanjeev Setia, Samir Koussih, and Sushil Jajodia. Kronos: A scalable group re-keying approach for secure multicast. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Oakland, CA, May 2000.

[31] M. Steiner, G. Tsudik, and M. Waidner. Cliques: A new approach to group key agreement. *IEEE Transactions on Parallel and Distributed Systems*, To appear in 2000.

[32] Wade Trappe, Jie Song, Radha Poovendran, and K. J. Ray Liu. Key distribution for secure multimedia multicast via data embedding. In *IEEE ICASSP 2001*, Salt Lake City, Utah, May 2001.

[33] Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, April 1996.

[34] D. Wallner, E. Harder, and R. Agee. Key management for multicast: Issues and architectures. Request for Comments (Informational) 2627, Internet Engineering Task Force, June 1999.

[35] Debby M. Wallner, Eric J. Harder, and Ryan C. Agee. Key Management for Multicast: Issues and Architectures. Technical report, IETF draft, July 1997.

[36] C. Wong, M. Gouda, and S. Lam. Secure group communications using key graphs. Technical Report TR-97-23, University of Texas at Austin, Department of Computer Sciences, August 1997.

[37] C. Wong, M. Gouda, and S. Lam. Secure group communications using key graphs. In *Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 68–79, 1998. Appeared in *ACM SIGCOMM Computer Communication Review*, Vol. 28, No. 4 (Oct. 1998).

[38] Chung Kei Wong and Simon S. Lam. Keystone: A group key management service. In *International Conference on Telecommunications, ICT 2000*, 2000.

[39] Avishai Wool. Key management for encrypted broadcast. In *5th ACM Conference on Computer and Communications Security*, pages 7–16, San Francisco, California, November 1998.

[40] X. Rex Xu, Andrew C. Myers, Hui Zhang, and Raj Yavatkar. Resilient multicast support for continuous-media applications. In *NOSSDAV*, 1997.

[41] M. Yajnik, S. Moon, J. Kurose, and D. Towsley. Measurement and modelling of the temporal dependence in packet loss. In *IEEE INFOCOM '99*, March 1999.

[42] Bennet Yee and Doug Tygar. Secure coprocessors in electronic commerce applications. In *Proceedings of The First USENIX Workshop on Electronic Commerce*, New York, New York, July 1995.

[43] Bennet S. Yee. *Using Secure Coprocessors*. PhD thesis, Carnegie Mellon University, 1994.

A Unicast Key Recovery Protocol

In case the group member did not receive the key update message and is in the unlucky part of the key tree, it cannot recover the group key from the hint message. In this case, it needs to request the keys from the key server. This request protocol is straightforward to design. The most important requirement is that the protocol is efficient and scales well. In contrast to previous work by Wong and Lam, we find that TCP is not the appropriate protocol for this key request protocol [38]. The reason is that TCP requires one round-trip message just to set up the connection (in most implementations, no data is sent in the initial SYN packet). This slows down the request unnecessarily. An advantage of TCP might be that the receiver could keep the connection open for future key requests. This approach, however, does not scale to large number of receivers because each TCP connection requires a considerable amount of state at the server. Therefore, we propose to use a light-weight key update protocol based on UDP, where the receiver achieves reliability through timeout and request retransmissions.

B Security Analysis

We perform our security analysis in a computational complexity framework. Our attacker model distinguishes between passive and active adversaries. Passive adversaries only eavesdrop on the group communication (in particular they are never group members), whereas active adversaries may be group members. For both cases, we analyze the computation complexity for the adversary to derive a group key while it is not a group member. To clarify the subsequent description, we refer to the passive adversary as Eve and to the active attacker as Mallory.

B.1 Security Analysis with Passive Adversary

We first look at the difficulty of the passive adversary (Eve) to compute the group key. We assume that Eve eavesdrops all traffic without loss (receives all data packets of a session), but Eve is never a group member and hence does not know any keys in the key tree. Clearly, an exhaustive search attack to find the group key takes $\Omega(2^n)$ operations, where n is the bit length of the group key. Eve cannot do better than brute force search $\Omega(2^n)$ possibilities by using key update messages or hint messages, because she does not know any of the keys used to decrypt the key update or any of the keys used for the pseudo-random function to compute the hints. Hence the following observation clearly holds:

Proposition 1. *With overwhelming probability, Eve needs to perform $\Omega(2^n)$ operations to determine the group key by exhaustive search.*

B.2 Security Analysis with Active Adversary

We now turn our attention to Mallory and we assume that he was a group member during some previous time period. We analyze his computation complexity to derive a group key while he is not in the group.

We first look at backward secrecy. When Mallory joins the group and receives all keys on his key path, can he derive previous group keys? Mallory might have recorded earlier key update messages, hints and data packets encrypted with previous group keys, but he cannot derive the group key better than brute-force $\Omega(2^n)$ possibilities because he does not have any of the decryption keys or the keys used for the pseudo-random function to compute the hints. Hence we obtain the following observation:

Proposition 2. *With overwhelming probability, Mallory needs to perform $\Omega(2^n)$ operations to brute-force any group key preceeding the time he joins the group.*

We now analyze forward secrecy, where Mallory tries to derive the group keys after he leaves the group. We assume that Mallory just left and hence knows all the keys on his key path because this scenario gives Mallory the greatest advantage. When Mallory leaves the group, the server updates the group key and the key tree as section 6.1.1 describes. We show that in this scenario the following observation is true:

Proposition 3. *After Mallory leaves the group, with overwhelming probability he needs to perform $\Omega(2^{n_1+n_2})$ operations to derive the new group key.*

Argument Sketch. The new group key is updated by the key server according to the description in section 6.2. Mallory has three ways to derive the new group key: from the key update messages, from the published hints, or brute-force all possibilities in the computation of generating the new group key.

The group key is distributed by encryption with the keys in the key tree that Mallory does not have, as described in LKH. Hence Mallory cannot do better than trying $\Omega(2^n)$ possibilities. Similarly, Mallory cannot gain advantage from the published hints either because he does not have the keys used in the pseudo-random function computation.

Note that the new group key K' is derived as $K' = \mathsf{PRF}_{\chi}^{\langle n \to n \rangle}(K_0)$, with $\chi = \mathsf{PRF}_{K_{0l}^{\alpha}}^{\langle n \to n_1 \rangle}(K_0) \mid \mathsf{PRF}_{K_{0r}^{\alpha}}^{\langle n \to n_2 \rangle}(K_0)$, where K_{0l} and K_{0r} are the left and right child keys of K_0 respectively. One of them is already updated depending on which side of the tree Mallory was in. Hence, Mallory does not know either K_{0l} nor K_{0r}. But Mallory knows K_0, so he only needs to brute-force all possibilities of χ which are $n_1 + n_2$ bits long.

Hence the best Mallory can do is to brute force $\Omega(2^{n_1+n_2})$ operations to compute the new group key. □

Note that even though Mallory now only needs to try $\Omega(2^{n_1+n_2})$ possibilities, where $n_1 + n_2$ might be much smaller than n, he cannot simply pre-compute a table with $\Omega(2^{n_1+n_2})$ entries and use it to derive new group keys later. The reason is that the new group key is computed using a pseudo-random function with the previous group key as input which keeps changing. So if Mallory wants to derive the new group key when he is not in the group any more, he needs to recompute the table with $\Omega(2^{n_1+n_2})$ entries with the latest group key as the input. This is the same amount of work as computing $\Omega(2^{n_1+n_2})$ possibilities. Hence Mallory does not gain advantage by doing precomputation and saving the result for later.