Computation-Efficient Multicast Key Distribution

Lihao Xu, *Senior Member*, *IEEE*, and Cheng Huang, *Member*, *IEEE*

Abstract—Efficient key distribution is an important problem for secure group communications. The communication and storage complexity of multicast key distribution problem has been studied extensively. In this paper, we propose a new multicast key distribution scheme whose *computation* complexity is significantly reduced. Instead of using conventional encryption algorithms, the scheme employs MDS codes, a class of error control codes, to distribute multicast key dynamically. This scheme drastically reduces the computation load of each group member compared to existing schemes employing traditional encryption algorithms. Such a scheme is desirable for many wireless applications where portable devices or sensors need to reduce their computation as much as possible due to battery power limitations. Easily combined with any key-tree-based schemes, this scheme provides much lower computation complexity while maintaining low and balanced communication complexity and storage complexity for secure dynamic multicast key distribution.

Index Terms—Key distribution, multicast, MDS codes, erasure decoding, computation complexity.

1 Introduction

In many applications, multicast is an efficient means of distributing data in terms of resources (such as network bandwidth, server computation, and I/O load) usage. The privacy of a multicast communication session is usually ensured using (symmetric) encryption. All the designated receivers or members in a multicast group share a session (encryption) key. In many applications, however, the multicast group membership changes dynamically, i.e., some new members are authorized to join a new multicast session, whereas some old members should be excluded. Thus, session keys shall change dynamically to ensure both *forward secrecy* and *backward secrecy* of multicast sessions. The forward secrecy is maintained if an old member who has been excluded from the current and future sessions *cannot* access the communication of the current and future sessions, and the backward secrecy is guaranteed if a new member of the current session *cannot* recover the communication data of past sessions. Each session thus needs a new key that is only known to the current session members, i.e., session keys need to be *dynamically* distributed to authorized session members.

In this paper, we study how a multicast group key can efficiently be distributed in *computation*. We adopt a common model where session keys are issued and distributed by a *central group controller* (GC), as it has much less *communication complexity*, as compared to *distributed* key exchange protocols, which is a very desired property in most wireless applications [35], [36], [37], [22], [38], [39]. The resources needed for the GC to distribute session keys to group members include communication, storage, and computation resources. The *communication complexity* is usually measured by the number of data bits that need to be transmitted from the GC to group members to convey information of session keys, whereas the *storage complexity* is measured by the number of data bits that the GC and group members need to store to obtain session keys. Another similarly important but usually undernoticed, if not ignored, factor is the *computation complexity*, which can be measured by the number of computation operations (or the computation time on a given computing platform) that the GC and group members need to distribute and extract session keys. Hereafter, the problem of how resources can effectively be used to distribute session keys is referred to as the *group key distribution* problem.

The group key distribution problem has been studied extensively in the larger context of key management for secure group communications [26], [27], mainly on balancing the storage complexity and the communication complexity. There are two trivial schemes for distributing a session key to a group of n members. The first one is that the GC shares an individual key with each group member, which can be used to encrypt a new group session key. In this scheme, the communication complexity is $O(n)$, whereas the GC needs to store $O(n)$ key information, each member stores $O(1)$ key information, and $O(n)$ encryption and decryption operations are needed. In the second scheme, the GC shares an individual key with each *subset* of the group, which can then be used to multicast a session key to a designated subset of group members. Now, both the communication complexity and the computation complexity reduce to $O(1)$, but at the cost of increasing the storage complexity to $O(2^n)$ for both the GC and each group member. It is easy to see that neither scheme works for practical applications with a reasonable group size n. Thus, research efforts have been made to achieve low communication and storage complexity for group key distribution.

Static secret sharing via broadcast channel was studied in [32] and [20]. However, this *threshold*-based scheme can only distribute a session key to a designated group of

- *L. Xu is with the Department of Computer Science, Wayne State University, 5143 Cass Avenue, 431 State Hall, Detroit, MI 48202. E-mail: lihao@cs.wayne.edu.*
- *C. Huang is with Microsoft Research Labs, One Microsoft Way, Redmond, WA 98052. E-mail: cheng.huang@microsoft.com.*

Manuscript received 12 Feb. 2007; revised 15 June 2007; accepted 24 July 2007; published online 16 Aug. 2007.
Recommended for acceptance by P. Mohapatra.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0050-0207.
Digital Object Identifier no. 10.1109/TPDS.2007.70759.

members for *one-time* use. Once a session key is distributed to the group, any member can calculate the secret information that other members in the same group hold. Thus, the scheme does not provide forward or backward secrecy. A secure lock method based on the Chinese Remainder Theorem was proposed in [11]. However, its prohibitively high communication complexity and computation complexity make it only practical for a very small group with limited number of members. Various theoretical measures and schemes for group key distribution were introduced in [14]. Along the same line, many research efforts have been made on balancing communication complexity and storage complexity of the group key distribution problems, for example, [8], [18], [6], [7], [35], and [36].

For a multicast group with a large number of members, *key-tree*-based schemes were introduced to decompose a large group into multiple layers of subgroups with smaller sizes [37], [22], [38], [39]. Using these schemes, a group membership change can be effectively handled in the corresponding subgroups without affecting other ones. Thus, the communication complexity is reduced, but at the cost of increase in storage and computation complexity together with extra communication delays. For a group of n members, key-tree-based schemes have a communication complexity of $O(logn)$ and a storage complexity of $O(n)$ for the GC and $O(logn)$ for each group member. It has been shown that if a group member can store at most $O(logn)$ keys, then the lower bound of the communication complexity is $O(logn)$ if a *structure-preserving* protocol is used for group key distribution [10]. Thus, the key-tree-based schemes are of practical interest for a variety applications because of its balance between communication complexity and storage complexity.

Although most research on group key distribution has been on balancing communication complexity and storage complexity, very few efforts have been made to reduce *computation complexity* [33]. It has been long assumed that expensive encryption and decryption operations are necessary to distribute group keys. Although broad-sense multicast is becoming increasingly practical over general Internet using various technologies such as overlay networks, multicast certainly gains most on true broadcast communication media such as wireless networks. In such wireless systems, multicast receivers (group members) are often of various lightweight *mobile* devices or *sensors*. Although it is becoming increasingly affordable to embed considerable computing power (and storage capacity) into these devices, their battery power will remain to be limited for a long time ahead. Computation complexity is thus more important than storage complexity for these devices in many applications. Hence, it becomes at least equally, if not more important, to reduce the computation complexity of the group key distribution problem, which has been understudied so far.

In this paper, we propose a new *dynamic* group key distribution scheme that drastically reduces computation complexity and yet maintains at least the same security degree of using symmetric encryption algorithms without increasing communication or storage complexity. In our scheme, information related to session keys is encoded using *error control codes* rather than encryptions. In general, encoding and decoding of a proper error control code have much (at least one order, although this is hard to strictly quantify analytically) lower computation complexity than existing encryption and decryption algorithms, which has been verified by our experiments described later in Section 3.4. Thus, the computation complexity of key distribution can be significantly reduced. The similar idea of using error control codes to achieve privacy was employed in [32], [20], and [4]. The major difference between these schemes and ours is that our scheme allows *dynamic* group membership changes with very low storage complexity, whereas the other schemes only work for a predefined *static* group.

The security strength of this scheme will be evaluated, as well as its communication, storage, and computation complexity. Aside from its low computation complexity, this scheme also has low storage complexity, i.e., $O(1)$ for an individual group member and $O(n)$ for the GC, where n is the number of group members. Based on the basic scheme using error control codes, concrete design parameters are derived to apply this scheme to key trees. Experiments are conducted to show great reduction of our scheme in computation complexity than using other commonly used traditional encryption algorithms on 3-ary balanced key trees.

This paper is organized as follows: A basic scheme using error control codes to distribute multicast keys is described in Section 2. Section 2 also evaluates the security and resource consumption of the basic scheme. Section 3 applies the general scheme to a balanced 3-ary key tree to achieve low communication, computation, and storage complexity. Implementation and experimental results are given as well. Section 4 concludes this paper.

2 The Basic Scheme: Dynamic Key Distribution Using Maximum Distance Separable Codes

2.1 Maximum Distance Separable Codes

Maximum Distance Separable (MDS) codes are a class of error control codes that meet the Singleton bound [19, chapter 11]. Letting $GF(q)$ be a finite field with q elements [19, chapter 4], an (n, k) (block) error control code is then a mapping from $GF(q)^k$ to $GF(q)^n : E(m) = c$, where $m = m_1 m_2 \cdots m_k$ is the original message block, $c = c_1 c_2 \cdots c_n$ is its code word block, and $E(\cdot)$ is an encoding function, with $k \leq n$. If a decoding function $D(\cdot)$ exists such that $D(c_{i_1} c_{i_2} \cdots c_{i_k}, i_1, i_2, \cdots, i_k) = m$ for $1 \leq i_j \leq n$ and $1 \leq j \leq k$, then this code is called an (n, k) MDS code. For an (n, k) MDS code, the k original message symbols can be recovered from *any* k symbols of its code word block. The process of recovering the k message symbols is called *erasure decoding*. All the symbols are defined over $GF(q)$, and usually, $q = 2^m$. The well-known Reed-Solomon (RS) codes [28] are a class of widely used MDS codes. Notably, the RS codes and other MDS codes can be used to construct secret-sharing and threshold schemes [32], [20].

2.2 Description of the Basic Scheme

For a dynamic multicast group, a session key is issued by a GC. Using this session key, the GC can establish a secure

multicast channel with the authorized group members. Every time group memberships change because of the *join* or *leave* of some group members, the GC reissues a new session key, which is independent of all the old session keys. This *rekeying* procedure ensures the security of the current session and that of the old sessions, i.e., the newly joined members cannot recover the communications of the old sessions, and those old members who left the group cannot access the current session. Thus, both the backward secrecy and the forward secrecy of group communication are maintained.

The complexity of the rekeying operation is asymmetric between a new member's join and an old member's leave. When a new member joins, the GC can easily multicast the new session key encrypted by the current session key to all the current members, followed by a unicast to the new member to send the new session key encrypted by a predetermined encryption key shared between the GC and the new member. Thus, join is easy, with low communication and computation cost. However, when an old member leaves, the current session key cannot be used to convey the new session key information securely, since it is also known to the old member. Thus, hereafter, we will focus on the rekeying operation for a single member leave. The same idea can easily be extended to other rekeying operations such as batch rekeying [17].

In any key distribution schemes, a basic operation is needed to distribute a piece of secret data to a small group of n members, where each member shares a different individual key with the GC. In all current existing schemes, this operation is fulfilled by the GC using n encryptions, followed by n unicasts. Now, we describe a new scheme that realizes this operation by using *one* erasure decoding of certain MDS code, followed by one multicast to all the n members. We call this scheme the *basic* scheme of key distribution. We will then show that this basic scheme can be easily integrated into any key distribution scheme, especially the schemes based on key trees, to reduce computation cost.

The basic scheme consists of three phases: 1) the initialization of the GC, 2) the join of a new member, and 3) the rekeying procedure whenever a group member leaves. Here again, a targeted multicast group has n members.

2.2.1 Group Controller Initialization

Initially, the GC constructs a *nonsystematic* (L, n) MDS code C over $GF(q)$ and a secure *one-way* hash function $H(\cdot)$ whose codomain is $GF(q)$. (For a nonsystematic code, *none* of the original message block symbols directly appears in the corresponding code word block [19].) The domain of $H(\cdot)$ can be an arbitrary space F that is large enough so that $H(\cdot)$ has a secure one-way property: given any arbitrary $y \in GF(q)$, it is impossible or computationally hard to derive $x \in F$ such that $H(x) = y$. Since other strong properties such as second-preimage resistance [21, chapter 9.2] are *not* necessary, the hash function H can be implemented more efficiently. The GC then makes both the MDS code C and the one-way hash function H public.

2.2.2 Member Initial Join

Whenever a new member i is authorized to join the multicast group for the first time, the GC sends it (using a secure unicast) a pair (j_i, s_i), where s_i is a random element in $H(\cdot)$'s domain F, and j_i is a positive integer satisfying $j_i \neq j_k$ for all k's, where k is a current member of the multicast group. The pair (j_i, s_i) will be used as member i's *seed key* (denoted as S_i) and is kept in the GC's local database, as long as member i remains a potential member of the multicast group.

2.2.3 Rekeying

Whenever some new members join or some old members leave a multicast group, the GC needs to distribute a new session key to all the current members. As already discussed, we will focus on the rekeying operation when an old member leaves. After an old member leaves, the GC needs to distribute a new key to n remaining members to achieve both forward and backward secrecy of the session key.

The GC executes the rekeying process in the following steps:

1. The GC randomly chooses a *fresh* element r in F, which has not been used to generate previous keys.
2. In the remaining group of n members, for each member i of the current group with its seed key (j_i, s_i), the GC constructs an element c_{j_i} in $GF(q) : c_{j_i} = H(s_i + r)$, where $+$ is a simple combining operation in F, for example, string concatenation.
3. Using all the c_{j_i}'s in the above step, the GC constructs a code word c of the (L, n) MDS code C: set the (j_i)th symbol of the code word c to be c_{j_i}. Since C is an (L, n) MDS code, the code word c is uniquely determined by its n symbols. Using an efficient erasure decoding algorithm for C, the GC can easily calculate the n corresponding message symbols $m_1 m_2 \cdots m_n$.
4. The GC sets the new session key k to be the first message symbol $m_1 : k = m_1$.
5. The GC multicasts r and $m_2 \cdots m_n$.

The above rekeying process is illustrated in Fig. 1a.

Note that when $n = 1$, i.e., there is only one member remaining, then in step 3 above, the (L, n) MDS code becomes a trivial *repetition* code, i.e., all the symbols in a code word are the *same* [19]. Hence, the decoding algorithm in step 3 becomes trivial, i.e., $m_1 = c_{j_i}$, without the need to compute any other m_i $(i \geq 2)$. This is also intuitive: the GC in this case simply sets a session key that is solely derived from the remaining member's seed key (j_i, s_i).

Upon receiving r and $m_2 m_3 \cdots m_n$ from the GC, an authorized member i of the current group executes the following steps to obtain the new session key:

1. Calculate $c_{j_i} = H(s_i + r)$ with its seed key (j_i, s_i).
2. Decode the first message symbol m_1 from the $(n-1)$ message symbols $m_2 \cdots m_n$, together with its code word symbol c_{j_i}.
3. Recover the new session key k, i.e., $k = m_1$.

This key recovery process, as shown in Fig. 1b, finishes the whole rekeying procedure. Note that in step 2 of the key recovery process, virtually, all MDS codes in use, for

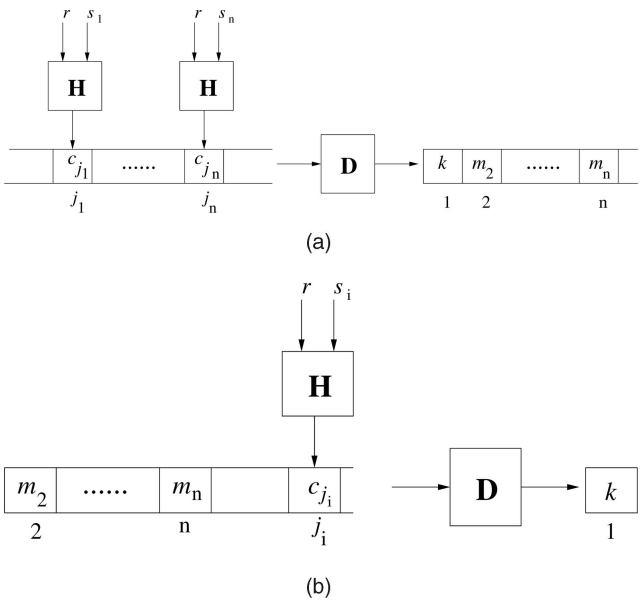

Fig. 1. Rekeying process of the basic scheme. (a) GC's operations. (b) Operations of members.

example, the RS code, are linear, i.e., any single code word symbol is a *linear combination* of all the n original message symbols. This decoding process is essentially for solving a single linear equation with only *one* unknown. Thus, it is equivalent to an encoding operation with much lower computation than a general erasure decoding operation for multiple unknowns.

The MDS property of the code C ensures that each authorized member, with its c_{i_j} and $m_2 \cdots m_n$, can derive the new session key $k = m_1$ as the GC. It is worth noting that the new session key k is only the first message symbol m_1, which is independent of all the other message symbols $m_2 \cdots m_n$. Thus, any unauthorized receiver cannot deduce m_1 just from $m_2 \cdots m_n$, since it needs one more symbol of the code word c. Any stale seed key (j'_i, s'_j) cannot generate a valid symbol of the current code word c, since the pair (j'_i, s'_j) is not used in the generation of c. Thus, both the forward and the backward secrecy are achieved.

2.3 Evaluation of the Basic Scheme

As can be seen from the above basic scheme, for all the authorized group members, the GC generates a new session key by generating a *common* new message word, the first symbol of which is used as the new session key. The new session key is decided by a random element r in F, as well as all the seed keys of the current authorized group members. The random element r and the $(n-1)$ message symbols are multicast in *plaintext* to all the group members, and the computation cost is thus much lower than using encrypted point-to-point communications for the rekeying process in all existing schemes. The computation operations needed for this new scheme are erasure decoding of a chosen MDS code, a one-way hash function, and some simple combining functions, all of which can be implemented efficiently and are computationally much cheaper than traditional encryptions.

As already noted, the purpose of this basic scheme is to replace separate encryptions, followed by unicasts, and be used as a building block for any key distribution schemes whenever applicable. Thus, we examine the communication, computation, and storage costs of the basic scheme and compare them with those of conventional rekeying schemes using point-to-point unicasts secured by separate encryptions.

For simplicity, hereafter, we assume that $q = 2^m$, and the size of a new session key is l bits.

2.3.1 Security

Since r and $m_2 \cdots m_n$ are multicast in plaintext and are thus known to all interested parties, including unauthorized receivers who attempt to access the current session, the security of the new session key relies on the secrecy of a code word symbol c_{j_i} that authorized member i of the current multicast group has. The secrecy of c_{j_i}, in turn, depends on the secrecy that the seed key member i has. Thus, an attacker who attempts to deduce the new session key k has three potential options:

1. brute-force attack, i.e., guessing the session key k itself,
2. guessing a symbol c_k of the code word, or
3. guessing a current member's seed key.

The effort that an attacker makes to deduce a session key depends on the parameters of the scheme, namely, the size of finite field $GF(2^m)$, the size t of member i's seed key component s_i, and the size of a random number r. Intuitively, when proper hash function $H(\cdot)$ and a random number generator are chosen, the larger these parameters are, the more the effort that an attacker needs to make, and thus, the more secure this scheme is. The following theorem states the exact sizes of the parameters to ensure the security of this scheme:

Theorem 1. *Over a finite field $GF(2^m)$, if $m = t = l_r = l$, then the effort that an attacker needs to make to deduce a session key from the basic scheme is* no less *than that of a brute-force attack, where t is the size of member i's seed key component s_i, l_r is the size of random number r, and l is the size of session key k.*

Proof. The effort that an attacker needs to make can be measured by the *entropy* of the information that an attacker attempts to recover [12]. We give an information-theoretical proof to show that the entropy of either a symbol c_k of code word or a current member's seed key is no less than the entropy of the session key itself. Hence an attacker makes no less effort than a brute-force attack of directly guessing the session key itself.

First, the entropy of a random session key k is $\mathcal{H}(k) = l$, where $\mathcal{H}(X)$ is the entropy of a random variable X.

If an attacker chooses to guess a symbol c_{j_i} in the code word directly, then the entropy of c_{j_i} is $\mathcal{H}(c_{j_i}) = log_2\, q = m = l$. Even though the correct location of a symbol in the code word is needed to generate the corresponding session key, an attacker can exploit the fact that for any location j_i, there always exists a correct symbol c_{j_i}, which is a symbol of the code word that generates a new session key. Thus, the attacker can first pick up an arbitrary location j_i, where $n \leq j_i \leq L$, and then guess the code word

symbol c_{j_i} at that location. Thus, the entropy of getting a correct code word symbol is $\mathcal{H}(c_{j_i}) = m = l$.

On the other hand, a seed key consists of two components: an element s_i in F and an integer j_i. It is easy to see that $\mathcal{H}(s_i) = t = l$. In addition, notice that the two components s_i and j_i of a seed key S_i are chosen *independently*; thus, $\mathcal{H}(S_i) = \mathcal{H}(s_i) + \mathcal{H}(j_i) = l + log_2 L$ when an (L, n) MDS code is used.

Finally, the inputs to the one-way hash function $H(\cdot)$ are s_i and r, each of which has size l, and the output is c_{j_i}, a symbol of a code word, whose size is l, as shown in Fig. 1a. Considering the fact that r is also known to the attacker, the effort that an attacker needs to deduce s_i from r and c_{j_i} is the *conditional* entropy [12] $\mathcal{H}(s_i|r, c_{j_i}) = \mathcal{H}(s_i) = l$ if $H(\cdot)$ is a properly chosen secure hash function. This corresponds to the scenario where the attacker is a current member but wants to deduce another member's seed key for future use. This is a more serious attack, since it compromises the security of multiple future sessions. In this case, the attacker is able to compute any code word symbol c_{j_i}, and it can simply try deducing s_i's for all possible c_{j_i}'s for future use.

Combining all the above possible attacks, the amount of information that an attacker needs to guess to obtain an l-bit session key k is at least l bits, i.e., the attacker needs to make as much effort as a brute-force attack to deduce a session key from the basic scheme. □

Now, we examine the security strength of this scheme against a *conspiracy attack*, where some old members cooperate to deduce the current session key. Since its freshness is determined by a fresh random number r, a new session key is totally independent of all the old session keys. This guarantees the *forward secrecy* and the *backward secrecy*, i.e., old members cannot access the new session, and new members cannot recover communications of all the previous sessions. One possible way for an old member to get a new session key is to calculate the seed key of a current member from old keys. It is easy to compute a symbol c_k of a code word c from a message word m once k is known. However, because of the secure one-way property of the hash function $H(\cdot)$ used to the generate c, it is impossible (or at least computationally hard) to compute a seed key (s_i, j_i) from c_{j_i}, even if c_{j_i} is known. Cooperation among d old members helps reduce neither the number of possible locations for j_i of a current member nor the computational hardness of getting s_i from c_{j_i}, since j_i can be recycled from old ones, as long as the current member i has unique j_i, which is different from that of all other current members. Thus, this scheme is *resilient* to any conspiracy attack.

Finally, it is conjectured that for all (L, n) MDS codes over $GF(q)$, where $q = 2^m$, the largest possible L is $q + 1 = 2^m + 1$ [19]. Thus, for a session key of l bits, this scheme can support up to 2^l members for one group.

2.3.2 Complexity

When a new member i is authorized to join a multicast group, the GC assigns a seed key pair (j_i, s_i) to it. This seed key pair remains valid until member i leaves the multicast group permanently. Thus, the seed key pair is unicast only *once* to an authorized member. Compared to the rekeying procedure that is needed whenever there is a membership change for the multicast group, this communication cost (in terms of the number of the bits transmitted) is negligibly small.

In the rekeying procedure, the GC needs to multicast a fresh random number r and $(n-1)$ symbols of the new code word $m_2 \cdots m_n$. Each of the code word symbols has m bits. The random number r is used to guarantee that the new session key is different from all the old keys used. The length of r determines the total number of sessions that the scheme supports. Letting r be l_r bits long, then this scheme can support up to 2^{l_r} sessions. Thus, the number of bits that the GC needs to multicast for a rekeying procedure is $l_r + (n-1)m$. As already discussed in the previous section, it is secure enough to set $l_r = m = l$; thus, the *communication complexity* of this basic scheme is nl, which is the *same* as that of separate encryptions followed by unicasts. Hence, the basic scheme does *not* incur additional communication complexity to the conventional schemes that it replaces.

With regard to *storage complexity*, a current member i only needs to store its own seed key pair (j_i, s_i). Since $j_i \leq L$ and s_i is t bits long, member i needs to store $\lceil log_2 L \rceil + t$ bits locally. On the other hand, the GC needs to store the seed keys of all the current members of the multicast group, i.e., the GC needs to store $n(\lceil log_2 L \rceil + t)$ bits. Since it is secure to set $m = t = l$ and $L = 2^m + 1$ for session keys of size l, an individual member's storage complexity is $(2l + 1)$, and the GC's is $n(2l + 1)$, both approximately about *twice* those for conventional schemes.

Finally we examine the computation cost of this scheme. Just as all other key distribution schemes, a seed key pair of a current member is distributed only once by using a secure unicast when the member joins the multicast group for the first time. In this join procedure, the GC needs only one encryption operation, and the newly joined member needs only one decryption operation. Most computations are thus carried in the rekeying procedure. During the rekeying procedure, the GC needs n hashing operations with n current members in the multicast group and one erasure decoding operation for n symbols. Symmetrically, a current member needs one hashing operation and one erasure decoding operation for only *one* symbol, i.e., the session key. The erasure decoding operations for an (L, n) MDS code only need $O(n^2)$ arithmetic operations if standard erasure decoding algorithms are used. Fast decoding algorithms only need $O(nlogn)$ operations [19]. Particularly, when a proper MDS code is chosen, both encoding and decoding only need $O(mn^2)$ binary exclusive OR operations [3], [5], [40].

3 Practical Key Distribution: Applying the Basic Scheme to Key Trees

The basic key distribution scheme reduces computation complexity by replacing computationally expensive encryption and decryption operations with more efficient erasure decoding operations of MDS codes. This basic scheme has the same communication complexity as conventional key distribution schemes using secure unicasts. Thus, the basic scheme can be readily used as a building block to replace

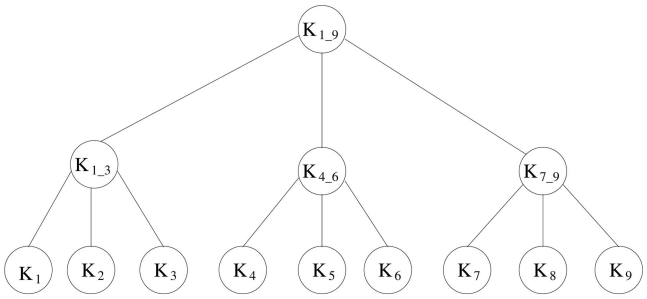

Fig. 2. A key tree for a nine-member group.

encrypted unicasts in any key distribution schemes, particularly schemes with low communication complexity.

To reduce the communication complexity of rekeying operations, a key-tree-based scheme and many of its variations have been proposed [37], [22], [38], [39], [34]. This scheme reduces the communication complexity of rekeying operations to $O(logn)$, whereas each member needs to store $O(logn)$ keys, and the GC needs to store $O(nlogn)$ keys, where n is the multicast group size. This is the most practical key distribution scheme, which balances the communication and storage complexity for dynamic multicast key distribution.

Here, we briefly describe a basic key-tree-based scheme for the rekeying operation.

3.1 Key-Tree-Based Rekeying Scheme

The main idea of reducing the rekeying communication complexity of this scheme is to have the GC distribute *subgroup* keys in addition to individual member keys and the group session key. These keys are arranged in a *logical* tree hierarchy, where the group session key serves as the root, the individual member keys are the leaves, and the subgroup keys correspond to intermediate nodes. Each member stores all the keys along the path from the corresponding leaf to the root in the tree. Then, each subgroup key can be used to securely multicast to the members that are leaves of the corresponding subtree. During the rekeying process, the GC can thus securely multicast to a subgroup of members using their shared subgroup key instead of individual member keys.

Fig. 2 shows a key tree for a nine-member group. K_i $(1 \leq i \leq 9)$ is the individual key of member i. K_{1_9} is the group session key that is shared by all the members. Finally, K_{1_3}, K_{4_6}, and K_{7_9} are three subgroup keys for the three corresponding subgroups respectively. For example, K_{1_3} is shared by members 1 through 3, who form the first subgroup.

Now, suppose that member 9 leaves the group. Then, a rekeying operation is needed to change the current group session key K_{1_9} and the corresponding subgroup key K_{7_9}. This can be achieved by the GC multicasting five messages (of equal size) to the remaining eight members: $E_{K_7}(K_{7\_8})$, $E_{K_8}(K_{7\_8})$, $E_{K_{7\_8}}(K_{1\_8})$, $E_{K_{1\_3}}(K_{1\_8})$, and $E_{K_{4\_6}}(K_{1\_8})$. Here, K_{1_8} is the new group session key, K_{7_8} is the new subgroup key for the new subgroup formed by members 7 and 8, and $E_k(m)$ is the encrypted version of message m using key k.

Although the key tree can be arbitrary, it has been shown that a balanced tree is required to achieve low communication complexity of the rekeying operation. In general, it is easy to show that the communication complexity of a rekeying operation on a balanced d-ary key tree is $O(dlog_dn)$, and each member needs to store $O(log_dn)$ keys. It is not hard to show that $dlog_dn$ is minimized when d is 3 [34]. The only cost of this key-tree-based scheme is the storage increase in the GC: the GC now needs to store all the additional subgroup keys as well. The total number of the keys stored on the GC increases to $\frac{dn-1}{d-1}$ from $n+1$ for a d-ary balanced key tree.

3.2 MDS Code-Based Rekeying on a Key Tree

The GC initialization and each member's initial join can be performed exactly the same on a key tree as in the basic scheme described in Section 2.2. Thus, we focus on the adaption of the basic scheme for rekeying on a key tree.

As in other key-tree-based rekeying schemes, MDS codes are used to rekey from bottom (leaves) up. In Fig. 2, when member 9 leaves, the new subgroup key K_{7_8} is rekeyed before the server changes the new group session key to K_{1_8}. When MDS codes are used for the rekeying process, each node (leaf or intermediate) key becomes a pair of (j_i, s_i), as discussed in the previous section. The GC stores all the key pairs on the key tree. Whenever encryptions are needed for rekeying a subgroup key, a new MDS code word is constructed from all the key pairs ((j_i, s_i)'s) of the corresponding immediate child nodes and then multicast by the GC. Note that in the rekeying process, each level of the key tree may use the same or different MDS codes. However, for the simplicity of implementation, the same MDS code can be used for all levels, since the security of the basic scheme does *not* depend on the MDS code.

In Fig. 2, when member 9 leaves the group, the GC first uses the key pairs $K_7 = (j_7, s_7)$ and $K_8 = (j_8, s_8)$, together with a fresh random r, to construct a code word of an $(L, 2)$ MDS code and then follows the rekeying procedure of the basic scheme, as described in the previous section. After proper decoding, members 7 and 8 share a new subgroup key K_{7_8} or, more accurately, its component s_{7_8}, which is only known to them, aside from the GC. Note here that $K_{7\_8} = (j_{7\_8}, s_{7\_8})$, but j_{7_8} can be predetermined *publicly* once the key tree structure is decided. This finishes the rekeying of K_{7_8}. Next, the GC constructs another code word of an $(L, 3)$ MDS code from the subgroup keys K_{1_3}, K_{4_6}, and K_{7_8}, and the decoding output from this code word produces a new group session key K_{1_8}, which is shared by all the remaining group members.

Note that when the key tree is a d-ary balanced tree, only an $(L, d-1)$ MDS code is needed to rekey the immediate subgroup key shared by the leaf node corresponding to the just-left old member. Then, another (L, d) MDS code is needed for rekeying all the other subgroup keys and the new group session key. Since the rekeying scheme based on MDS codes does *not* change the communication and storage complexity of the underlining key-tree-based rekeying scheme, the communication complexity still remains to be $O(dlog_dn)$.

3.3 MDS Code Implementation for the Rekeying Operation

As pointed out previously, d needs to be 3 to minimize the communication complexity during rekeying. Hence, only

two types of MDS codes are needed, which are $(L,2)$ and $(L,3)$ codes. In fact, the rekeying scheme only needs two specific MDS codes, i.e., an $(L,2)$ code and an $(L,3)$ code. Although any general (n,k) MDS code can be used for the rekeying purpose by setting $k=2$ or $k=3$, there are a number of optimization techniques that can be applied for special implementations of the $(L,2)$ and $(L,3)$ codes. As we shall see in the following, these techniques turn out to make the codes used for rekeying extremely fast, even though they do *not* readily extend to the implementations of general MDS codes.

3.3.1 Reed-Solomon (RS) Implementation: Vandermonde Representation versus Cauchy Representation

We choose the RS codes [28] as the MDS codes, since it is the most widely used MDS code. Among numerous ways of constructing an RS code, the two popular ones are *Vandermonde* and *Cauchy* representations. In the Vandermonde representation, a *Vandermonde Matrix* is used as a *generator matrix* for the chosen RS code [19], which is a traditional description of the RS code. Recently, a Cauchy representation for the RS code has been proposed to make general encoding and decoding of an RS code faster by using mostly XOR operations instead of more expensive finite-field operations [5]. In this representation, a *Cauchy Matrix* is used as a generator matrix.

For our rekeying purpose, an RS decoding operation is equivalent to solving a group of linear equations. It thus mainly involves two steps: 1) inverting a coefficient matrix and 2) multiplying the inverse matrix to get the values of the unknowns. In general, the second step has quite similar complexity among different representations for the same code. Thus, if one code representation has lower complexity in the first step (inverting the coefficient matrix), its overall decoding operation will be more efficient. It is well understood that the inversion of the Cauchy matrix requires less complexity than that of the Vandermonde matrix. Therefore, in general, Cauchy-matrix-based RS codes are considered as more efficient than Vandermonde-matrix-based RS codes [5], [25].

Quite contrary, we observe that Vandermonde representations are, in fact, more efficient for $(L,2)$ and $(L,3)$ RS codes in terms of decoding operations for the rekeying process. The main reason is that the inverse Vandermonde matrix is much simpler than the inverse Cauchy matrix for $k=2$ and $k=3$. Taking $k=3$ as an example, a more computation-efficient variant of a Vandermonde-matrix-based RS code can be represented as follows [23], [24]:

$$\begin{bmatrix} 1 & i & i^2 \\ 1 & j & j^2 \\ 1 & k & k^2 \end{bmatrix} \begin{bmatrix} m_1 \\ m_2 \\ m_3 \end{bmatrix} = \begin{bmatrix} c_i \\ c_j \\ c_k \end{bmatrix}, \tag{1}$$

where i, j, and k are the positions of members assigned by the GC when they join the multicast group. They are also considered to be elements in the corresponding finite field $\mathrm{GF}(2^m)$, which is used to construct the RS codes. Then, the inverse matrix can be represented as

$$\begin{bmatrix} m_1 \\ m_2 \\ m_3 \end{bmatrix} = \begin{bmatrix} \frac{jk}{(i\oplus j)(i\oplus k)} & \frac{ki}{(j\oplus i)(j\oplus k)} & \frac{ij}{(k\oplus i)(k\oplus j)} \\ \frac{j\oplus k}{(i\oplus j)(i\oplus k)} & \frac{k\oplus i}{(j\oplus i)(j\oplus k)} & \frac{i\oplus j}{(k\oplus i)(k\oplus j)} \\ \frac{1}{(i\oplus j)(i\oplus k)} & \frac{1}{(j\oplus i)(j\oplus k)} & \frac{1}{(k\oplus i)(k\oplus j)} \end{bmatrix} \begin{bmatrix} c_i \\ c_j \\ c_k \end{bmatrix}. \tag{2}$$

Note that m_1 is just the multicast session key here.

Similarly, the RS codes constructed from the Cauchy matrix can be represented as

$$\begin{bmatrix} \frac{1}{x_i\oplus y_1} & \frac{1}{x_i\oplus y_2} & \frac{1}{x_i\oplus y_3} \\ \frac{1}{x_j\oplus y_1} & \frac{1}{x_j\oplus y_2} & \frac{1}{x_j\oplus y_3} \\ \frac{1}{x_k\oplus y_1} & \frac{1}{x_k\oplus y_2} & \frac{1}{x_k\oplus y_3} \end{bmatrix} \begin{bmatrix} m_1 \\ m_2 \\ m_3 \end{bmatrix} = \begin{bmatrix} c_i \\ c_j \\ c_k \end{bmatrix}, \tag{3}$$

where x_i, x_j, and x_k change values based on the positions of the members, whereas y_1, y_2, and y_3 are constants. The inverse matrix turns out to be much more complicated than that of the Vandermonde matrix, so we simply present one single entry to further elaborate. Letting $d_{1,1}$ be the entry of the inverse matrix at row 1 and column 1, then

$$d_{1,1} = \frac{(x_i\oplus y_2)(x_i\oplus y_3)(x_j\oplus y_1)(x_k\oplus y_1)}{(x_i\oplus x_j)(x_i\oplus x_k)(y_1\oplus y_2)(y_2\oplus y_3)}(x_i\oplus y_1). \tag{4}$$

It is true that the common terms (for example, $y_1\oplus y_2$, $y_2\oplus y_3$, and $y_1\oplus y_3$) can be precomputed for each given code construction, but still, every entry requires more operations than the inverse of the Vandermonde matrix.

Based on the above observation, we choose to use the Vandermonde matrix to construct the $(L,2)$ and $(L,3)$ RS codes, instead of using the Cauchy matrix, as conventional wisdom suggests.

3.3.2 Optimized Lookup Table for Multiplication and Division

Finite-field multiplication and division are usually implemented via lookup tables. Each element (except 0) in a finite field can be represented as a power to a primitive element (say, α). Hence, the multiplication of two elements a and b can be done by first adding the power values together and then calculating the exponential value as

$$a\times b = \alpha^{log_\alpha a + log_\alpha b}. \tag{5}$$

It is straightforward to precompute a logarithmic table and an exponential table for a given finite field. With that, the multiplication can simply be implemented by lookup tables as

$$a\times b = \exp[\log[a]+\log[b]]. \tag{6}$$

Similarly, the division can be implemented as

$$a\div b = \exp[\log[a]-\log[b]]. \tag{7}$$

Careful readers might immediately spot the potential problems of the exponential table, as 1) element a or b might be 0, which will not yield correct results from the lookup table, 2) the index to the table might exceed the table boundary in the multiplication case, and 3) the index to the table might become negative in the division case. These issues can essentially be addressed by augmenting the exponential table in the following ways: 1) augmenting the table such that the sum of two power values is always covered, 2) further augmenting the table such that 0 maps to a *remote* entry in the table such that a further operation

(plus/minus) results in a region nearby, where all corresponding values are 0, and 3) shifting the pointer to the table such that negative indices are allowed. Of course, the last one is programming language specific. However, even without programming language support, the same goal can still be achieved simply by adding a constant offset to a computed (negative) index. These techniques are also applicable to general implementations of the RS codes.

Considering our special needs of implementing the $(L, 2)$ and $(L, 3)$ codes, we discover that the lookup table idea can be extended to further simplify operations. The most complicated calculation in (2) requires three multiplications and two divisions (for example, $\frac{c_i jk}{(i \oplus j)(i \oplus k)}$). Hence, instead of performing this calculation as a series of binary operations, where each operation is similar to (6) or (7), we can build a special augmented exponential table to accommodate the entire 5-ary calculation as

$$\begin{aligned}&(a \times b \times c) \div (d \times e)\\ &= \exp[\log[a] + \log[b] + \log[c] - \log[d] - \log[e]].\end{aligned} \tag{8}$$

In addition, notice that only element a could take value 0 (b, c, d, and e *cannot* be 0 due to the construction of the RS codes), which means that the table needs just slightly further augmentation to accommodate this special case. Indeed, when a finite-field GF(256) is used, an exponential table of size 2,304 is sufficient. Assuming that negative index is supported, then the table spans from $[-1, 535, 768]$. $\exp[-1, 535, -512]$ always maps to 0 to handle the special case when $a = 0$ (the logarithmic table maps 0 to $-1, 024$ here, i.e., $\log[0] = -1, 024$), and $\exp[-511, 768]$ handles normal 5-ary operations (multiplications together with divisions). It is conceivable that such exponential table can easily fit into the fast cache of modern computing devices.

3.3.3 Size of the Finite Field

In most current applications, a session key needs to be at least 128 bits to be considered reasonably secure. According to the basic scheme, $m = l_r = t = 128$ bits, so the RS codes should be constructed in a finite field of $\mathrm{GF}(2^{128})$. In such a field, each element is represented by 128 bits, and the total number of elements is 2^{128}. Apparently, this field is too large to have an efficient and meaningful implementation, as it is impossible to have a logarithmic table or an exponential table of this size. Instead, we choose to construct the RS codes in a much smaller field of $\mathrm{GF}(2^8)$, where each element is represented by 8 bits, and there are total 256 elements in the field. This way, a 128-bit key is composed of 16 elements in the finite field.

Before moving forward, we first address some potential concerns of very careful readers. We argue that reducing the size of the finite field does *not* compromise the security strength of the proposed scheme by any means. Let us revisit the basic scheme. When a member i joins the multicast group, the GC sends it a secret pair (j_i, s_i), where s_i is a random element, and j_i is essentially a position. If a $\mathrm{GF}(2^{128})$ finite field is used, then the entire 128-bit s_i is *one* single element, and the entire 128-bit j_i represents *one* single position. Now, supposing a finite field of GF(256) is used, then s_i corresponds to 16 random elements in the field, and j_i corresponds to 16 random positions. Note that these 16 elements/positions are completely independent. Later on, when a member departs and the rekeying process is triggered, a random r is generated by the GC. This 128-bit r is again *one* single element in $\mathrm{GF}(2^{128})$ while corresponding to 16 random elements in GF(256). Hence, although the 128-bit key is generated as one single element in $\mathrm{GF}(2^{128})$, it is generated as the composition of 16 elements when a finite field of GF(256) is used. Note that the key is generated by 16 independent procedures from completely independent values. Since guessing 16 independent 8-bit values is no easier at all than guessing one 128-bit value, reducing the field size does *not* affect the security strength of our scheme.

3.4 Comparison with Traditional Cryptographic Schemes

To evaluate the proposed scheme, a multicast key distribution scheme is implemented to disseminate 128-bit session keys among a 3-ary balanced key tree. The proposed scheme is compared with traditional cryptographic schemes. As the communication and storage complexity are the same among all the schemes, it suffices to simply compare the computation complexity.

The comparison considers the following scenario, where each three-member group has one member that departs. These departures are not constrained to happen at the same time, but in practice, they might tend to be close, for example, at the end of one movie broadcast, etc. This makes a *batch* process possible, which means that all remaining members could be rekeyed at once.

Before reporting the experimental results, it is worth pointing out that any one-way hash function used in the proposed scheme can be simplified from general-sense hash function implementations. For instance, we use the MD5 algorithm [31, chapter18] as an exemplary hash function in our evaluation, which produces a 128-bit hash output from any arbitrary length input. A general MD5 input consists of three components: 1) input data, 2) padding bits, and 3) a final 64-bit field for length. In our case, as the input is always 128 bits, we can preset the final length field to represent 128. Moreover, all the rest bits can be set to 0 and removed from the MD5 logic. This tends to make the MD5 algorithm more efficient. Obviously, the same method can be readily applied to other hash algorithms, for example, SHA-1 and SHA-256 [31, chapter 18], should the MD5 algorithm be considered insufficient or using longer session keys becomes necessary.

Experiments are conducted to compare the computation complexity of the proposed scheme with the conventional cryptographic schemes. The particular cryptographic algorithms compared in the experiments are CAST-128 [29], IDEA [16], AES/Rijindael [1], and RC4 [31, chapter 17]. All keys are set to be 128 bits. To make the comparison as fair as possible, we use the widely adopted and highly optimized software-based cryptography implementation, i.e., the Crypto++ [13] package. We use the same optimization flags to compile both the cryptography package and our own optimized RS code implementation. We disable the hardware acceleration (SSE/SSE2) and tend not to compare under that scenario, simply because it is not ubiquitous over all platforms yet (for example, not available on mobile devices).

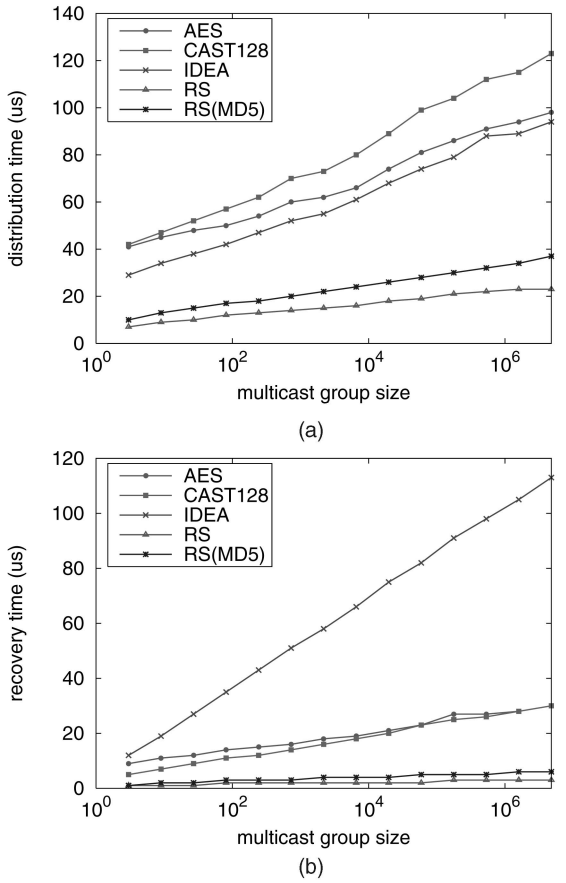

Fig. 3. Computation time for key distribution (the *RS(MD5)* shows the proposed scheme, whereas the *RS* curve excludes the hash function). (a) GC's computation time for key dissemination. (b) A member's computation time for key recovery.

Fig. 3 shows the computation time of the key dissemination and recovery using different schemes under various multicast group sizes. The experiments are carried out on a Pentium 4 2.53-GHz machine with a 512-Mbyte memory running Linux Redhat 9.0. It is clear that using one-way hash functions adds none-trivial computation complexity. Nevertheless, the proposed scheme still outperforms the conventional cryptographic schemes by a significant margin.

The computation time of the key distribution is also compared to conventional stream ciphers, as shown in Table 1, for a selected multicast group size. Notice that the computation times of both the GC and the member using the RC4 cipher are significantly larger than using other schemes. Even though RC4 itself is a fast stream cipher, its key scheduling process has dominant effect in this particular scenario, where only 128-bit data is encrypted/decrypted using any given key. Results under other multicast group sizes are similar, which are thus not duplicated here.

Finally, it is worth noting that our basic scheme simply reduces computation complexity by replacing cryptographic encryption and decryption operations with more efficient encoding and decoding operations. It is orthogonal to any other schemes that use different rekeying protocols and procedures. This basic scheme can always be combined with any rekeying schemes that use cryptographic encryption and decryption operations. For example, this basic scheme can be readily adapted to incorporate the so-called *one-way function tree* scheme [33], where a different rekeying protocol on a key tree is used other than the traditional scheme, as described in Section 3.1, to further reduce the computation complexity. We leave this simple exercise to interested readers.

TABLE 1
Computation Time Comparing to the RC4 Approach
(Multicast Group Size of 59,049)

time (us)	RS	RS(MD5)	AES	IDEA	CAST-128	**RC4**
GC	19	28	81	74	99	**227**
member	2	5	23	82	23	**61**

4 Conclusion

We have presented a dynamic multicast key distribution scheme using MDS codes. The computation complexity of key distribution is greatly reduced by employing only erasure decoding of MDS codes instead of more expensive encryption and decryption computations. Easily combined with key trees or other rekeying protocols that need encryption and decryption operations, this scheme provides much lower computation complexity while maintaining low and balanced communication complexity and storage complexity for dynamic group key distribution. This scheme is thus practical for many applications in various broadcast capable networks such as Internet and wireless networks.

Acknowledgments

Some preliminary results of this work were presented at the 2003 IEEE International Symposium on Information Theory, Yokohama, Japan, 29 June-4 July, 2003. This work was in part supported by US National Science Foundation (NSF) Grants ANI-0322615 and IIS-0541527.

References

[1] *AES Algorithm (Rijndael) Information,* http://csrc.nist.gov/CryptoToolkit/aes/rijndael/, 2007.

[2] M. Abdalla, Y. Shavitt, and A. Wool, "Towards Making Broadcast Encryption Practical," *IEEE/ACM Trans. Networking,* vol. 8, no. 4, pp. 443-454, Aug. 2000.

[3] M. Blaum, J. Bruck, and A. Vardy, "MDS Array Codes with Independent Parity Symbols," *IEEE Trans. Information Theory,* vol. 42, no. 2, pp. 529-542, Mar. 1996.

[4] R. Blom, "An Optimal Class of Symmetric Key Generation Systems," *Advances in Cryptology—Proc. Workshop Theory and Application of Cryptographic Techniques (EUROCRYPT '84),* pp. 335-338, 1984.

[5] J. Bloemer, M. Kalfane, M. Karpinski, R. Karp, M. Luby, and D. Zuckerman, "An XOR-Based Erasure-Resilient Coding Scheme," Technical Report TR-95-048, Int'l Computer Science Inst., Aug. 1995.

[6] C. Blundo and A. Cresti, "Space Requirement for Broadcast Encryption," *Advances in Cryptology—Proc. Workshop Theory and Application of Cryptographic Techniques (EUROCRYPT '95),* pp. 287-298, 1995.

[7] C. Blundo, A. De Santis, A. Herzberg, S. Kutten, U. Vaccaro, and M. Yung, "Perfectly Secure Key Distribution in Dynamic Conferences," *Advances in Cryptology—Proc. Workshop Theory and Application of Cryptographic Techniques (EUROCRYPT '93),* pp. 471-486, 1993.

[8] C. Blundo, L.A. Frota Mattos, and D.R. Stinson, "Trade-Offs between Communication and Storage in Unconditionally Secure Schemes for Broadcast Encryption and Interactive Key Distribution," *Advances in Cryptology—Proc. 16th Ann. Int'l Cryptology Conf. (CRYPTO '96),* pp. 387-400, 1996.
[9] R.E. Bryant and D.R. O'Hallaron, *Computer Systems: A Programmer's Perspective.* Prentice Hall, 2002.
[10] R. Canetti, T. Malkin, and K. Nissim, "Efficient Communication-Storage Tradeoffs for Multicast Encryption," *Advances in Cryptology—Proc. Int'l Conf. Theory and Application of Cryptographic Techniques (EUROCRYPT '99),* May 1999.
[11] G.H. Chou and W.T. Chen, "Secure Broadcasting Using the Secure Lock," *IEEE Trans. Software Eng.,* vol. 15, no. 8, pp. 929-934, Aug. 1989.
[12] T.M. Cover and J.A. Thomas, *Elements of Information Theory.* John Wiley & Sons, 1991.
[13] W. Dai, *Crypto++ Library,* http://www.eskimo.com/~weidai/cryptlib.html, 2007.
[14] A. Fiat and M. Naor, "Broadcast Encryption," *Advances in Cryptology—Proc. 13th Ann. Int'l Cryptology Conf. (CRYPTO '94),* pp. 480-491, 1994.
[15] H. Harney and E. Harder, *Logical Key Hierarchy Protocol,* IETF Internet draft, work in progress, Mar. 1999.
[16] X. Lai, J. Massey, and S. Murphy, "Markov Ciphers and Differential Cryptanalysis," *Advances in Cryptology—Proc. Workshop Theory and Application of Cryptographic Techniques (EUROCRYPT '92),* pp. 17-38, 1992.
[17] X.S. Li, Y.R. Yang, M.G. Gouda, and S.S. Lam, "Batch Rekeying for Secure Group Communications," *Proc. 10th Int'l World Wide Web Conf. (WWW '01),* May 2001.
[18] M. Luby and J. Staddon, "Combinatorial Bounds for Broadcast Encryption," *Advances in Cryptology—Proc. Int'l Conf. Theory and Application of Cryptographic Techniques (EUROCRYPT '98),* pp. 512-526, 1998.
[19] F.J. MacWilliams and N.J.A. Sloane, *The Theory of Error Correcting Codes.* North-Holland Math. Library, 1977.
[20] R.J. McEliece and D.V. Sarwate, "On Sharing Secrets and Reed-Solomon Codes," *Comm. ACM,* vol. 26, no. 9, pp. 583-584, Sept. 1981.
[21] A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone, *Handbook of Applied Cryptography,* fourth ed. CRC Press, 1999.
[22] S. Mittra, "Iolus: A Framework for Scalable Secure Multicasting," *Proc. ACM SIGCOMM '97,* pp. 277-288, Sept. 1997.
[23] J.S. Plank, "A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-Like Systems," *Software: Practice and Experience,* vol. 27, no. 9, pp. 995-1012, Jan. 1999.
[24] J.S. Plank and Y. Ding, "Correction to the 1997 Tutorial on Reed-Solomon Coding," *Software: Practice and Experience,* vol. 35, no. 2, pp. 189-194, Feb. 2005.
[25] J.S. Plank and L. Xu, "Optimizing Cauchy Reed-Solomon Codes for Fault-Tolerant Network Storage Applications," *Proc. Fifth IEEE Int'l Symp. Network Computing and Applications (NCA '06),* July 2006.
[26] S. Rafaeli and D. Hutchison, "A Survey of Key Management for Secure Group Communication," *ACM Computing Surveys,* vol. 35, no. 3, pp. 309-329, 2003.
[27] O. Rodeh, K. Birman, and D. Dolev, "The Architecture and Performance of Security Protocols in the Ensemble Group Communication System," *ACM Trans. Information and System Security,* vol. 4, no. 3, pp. 289-319, Aug. 2001.
[28] I.S. Reed and G. Solomon, "Polynomial Codes over Certain Finite Fields," *J. SIAM,* vol. 8, no. 10, pp. 300-304, 1960.
[29] C. Adams, *The CAST-128 Encryption Algorithm,* IETF RFC 2144, http://www.faqs.org/rfcs/rfc2144.html, May 1997.
[30] V. Rijmen, A. Bosselaers, and P. Barreto, *Optimised C Code V3.0 (of AES/Rijndael),* http://www.esat.kuleuven.ac.be/ rijmen/rijndael-fst-3.0.zip, 2007.
[31] B. Schneier, *Applied Cryptography,* second ed. John Wiley & Sons, 1996.
[32] A. Shamir, "How to Share a Secret," *Comm. ACM,* vol. 24, no. 11, pp. 612-613, Nov. 1979.
[33] A.T. Sherman and D.A. McGrew, "Key Establishment in Large Dynamic Groups Using One-Way Function Trees," *IEEE Trans. Software Eng.,* vol. 29, no. 5, pp. 444-458, May 2003.
[34] J. Snoeyink, S. Suri, and G. Varghese, "A Lower Bound for Multicast Key Distribution," *Proc. IEEE INFOCOM '01,* Apr. 2001.
[35] D.R. Stinson, "On Some Methods for Unconditionally Secure Key Distribution and Broadcast Encryption," *Designs, Codes and Cryptography,* vol. 12, pp. 215-243, 1997.
[36] D.R. Stinson and T. van Trung, "Some New Results on Key Distribution Patterns and Broadcast Encryption," *Designs, Codes and Cryptography,* vol. 14, pp. 261-279, 1998.
[37] M. Waldvogel, G. Caronni, D. Sun, N. Weiler, and B. Plattner, "The VersaKey Framework: Versatile Group Key Management," *IEEE J. Selected Areas in Comm.,* vol. 7, no. 8, pp. 1614-1631, Aug. 1999.
[38] D.M. Wallner, E.J. Harder, and R.C. Agee, "Key Management for Multicast: Issues and Architectures," *IETF Internet draft,* Sept. 1998.
[39] C.K. Wong, M. Gouda, and S.S. Lam, "Secure Group Communications Using Key Graphs," *Proc. ACM SIGCOMM '98,* Sept. 1998.
[40] L. Xu and J. Bruck, "X-Code: MDS Array Codes with Optimal Encoding," *IEEE Trans. Information Theory,* vol. 45, no. 1, pp. 272-276, Jan. 1999.

Lihao Xu received the BSc and MSc degrees in electrical engineering from the Shanghai Jiao Tong University in 1988 and 1991, respectively, and the PhD degree in electrical engineering from the California Institute of Technology in 1999. He was with the Department of Computer Science, Washington University, St. Louis, as an associate professor in July 2005 and as an assistant professor from September 1999 to June 2005. From 1991 to 1994, he was a lecturer in the Department of Electrical Engineering, Shanghai Jiao Tong University. Since August 2005, he has been an associate professor of computer science at the Wayne State University. His research interests include distributed computing and storage systems, error-correcting codes, information theory, and data security. He is a senior member of the IEEE.

Cheng Huang received the BS and MS degrees in electrical engineering from Shanghai Jiao Tong University in 1997 and 2000, respectively, and the PhD degree in computer science from Washington University, St. Louis, in 2005. He is currently a member of the Communication and Collaboration Systems Group, Microsoft Research, Redmond, Washington. His research interests include peer-to-peer applications, distributed storage systems, erasure correction codes, multimedia communications, networking, and data security. He is a member of the IEEE.