MARKS: Zero Side Effect Multicast Key Management using Arbitrarily Revealed Key Sequences

Bob Briscoe, <rbriscoe@jungle.bt.co.uk>; BT Research, B54/74, BT Labs, Martlesham Heath, Ipswich, IP5 3RE, England

Abstract. The goal of this work is to separately control individual secure sessions between unlimited pairs of multicast receivers and senders. At the same time, the solution given preserves the scalability of receiver initiated Internet multicast for the data transfer itself. Unlike other multicast key management solutions, there are absolutely no side effects on other receivers when a single receiver joins or leaves a session and no smartcards are required. The cost per receiver-session is typically just one short set-up message exchange with a key manager. Key managers can be replicated without limit because they are only loosely coupled to the senders who can remain oblivious to members being added or removed. The technique is a general solution for access to an arbitrary sub-range of a sequence of information and for its revocation, as long as the end of each sub-range can be planned at the time each access is requested.

1. Introduction

This paper presents techniques to maintain an individual security relationship between multicast senders and each receiver without compromising the efficiency and scalability of IP multicast's data distribution. We focus on issues that are foremost if the multicast information is being sold commercially. Of prime concern is how to individually restrict each receiver only to data for which it has paid.

We adopt an approach where the key used to encrypt sent data is systematically changed for each new unit of application data. The keys are taken from a sequence seeded with values initially known only to senders. A key sequence construction is presented where arbitrarily different sub-sequences can be revealed to each receiver by only revealing a small number of intermediate seed values rather than having to reveal the every key in each sub-sequence. Specifically a maximum of $O(\log(N))$ seeds need to be revealed *once per session* to each receiver in order to reconstruct a sub-sequence N keys long. This should be compared with the most efficient multicast key management solutions to date, that require a message of length $O(\log(n))$ to be multicast to *all* n receivers *every* time a receiver or group of receivers joins or leaves. Further, calculation of each key in the sequence only requires a mean of under two fast hash operations. (Notation is explained in Appendix B.)

In contrast, whenever a receiver is added or removed with the present scheme, there is zero side effect on other receivers. A special group key change isn't required because systematic changes occur sufficiently regularly anyway. No keys are sent over multicast, therefore reliable multicast isn't required. If key managers are delegated to handle requests to set-up receiver sessions, the senders can be completely oblivious to any receiver addition or removal. Thus, there is absolutely no coupling back to the senders. In many commercial scenarios (e.g. prepayment) key managers can be stateless, allowing performance to grow linearly with unbounded key manager replication. Resilience of the whole system would also be assured in such scenarios, even in the face of partial failures, due to the complete decoupling of all the elements.

Our thesis is that there are many applications that only rarely if ever require premature eviction, e.g. pre-paid or subscription pay-TV or pay-per-view. Thus, we don't present a solution for unplanned eviction, but instead concentrate on the pragmatic scenario of pre-planned eviction, which we believe is a novel approach. Each eviction from the multicast group is planned at each session set-up, but each is still allowed to occur at an arbitrary time. Nonetheless, we briefly describe how the occasional unplanned eviction can be catered for by modular combination with existing solutions at the expense of loss of simplicity and scalability.

Four other key sequence constructions are presented in a companion technical report [Briscoee99], which also presents a mathematical model that encompasses all five schemes and others in the same class (including the one-way function tree (OFT) [McGrew98]). Each scheme in the companion report has particular strengths; one is useful for sessions of unknown duration, another multiplies the effective key length against brute force attack (without increasing the operational key-length) and yet another is extremely simple and efficient in terms of message bandwidth but has limited commercial applicability. The scheme chosen for this paper is the simplest and is secure enough for most commercial scenarios.

In section 2, we discuss requirements and describe related work on multicast key management and other multicast security issues. In Section 3 we use an example application to put the paper into a practical context and to highlight the scalability advantages of using systematic key changes. In section 4 we present the key sequence construction that allows different portions of a key sequence to be reconstructed from various combinations of intermediate seeds. Section 5 discusses the efficiency and security of the construction. Section 6 very briefly describes variations on the approach to add other security requirements such as multi-sender multicast, a watermarked audit trail and unplanned eviction, although more detail and a wider literature review can be found in [Briscoee99]. Finally limitations of the approach are discussed followed by conclusions.

2. Background, Definitions and Requirements

When using Internet multicast, senders send to a multicast group address while receivers 'join' the multicast group through a message to their local router. For scalability, the designers of IP multicast deliberately ensured that any one router in a multicast tree would hide all downstream join and leave activity from all upstream routers and senders [Deering91]. Thus a multicast sender is oblivious to the identities of its receivers. Clearly any security relationship with individual receivers is impossible if they can't be uniquely distinguished. Conversely, if receivers have to be distinguished from each other, the scalability benefits start to be eroded.

If a multicast sender wishes to restrict its data to a set of receivers, it will typically encrypt the data at the application level. End-to-end access is then controlled by limiting the circulation of the key. A new receiver could have been storing away the encrypted stream before it joined the secure session. Therefore, every time a receiver is allowed in, the key needs to be changed (termed backward security [McGrew98]). Similarly, after a receiver is thrown out or requests to leave, it will still be able to decrypt the stream unless the key is changed again (forward security). Most approaches work on the basis that when the key needs to be changed, every receiver will have to be given a new key. Continually changing keys clearly has messaging side effects on all the other receivers than the one joining or leaving.

We define a 'secure multicast session' as the set of data that a receiver *could* understand, having passed one access control test. If one key is used for many related multicast groups, they all form one secure session. If a particular receiver leaves a multicast group then re-joins but she could have decrypted the information she missed, the whole transmission is still a single secure session. We envisage very large receiver communities, e.g. ten million viewers for a popular Internet pay-TV channel. Even if just 10% of the audience tuned in or out within a fifteen minute period, this would potentially cause thousands of secure joins or leaves per second.

We use the term 'application data unit' (ADU) as a more general term for the minimum useful atom of data from a security or commercial point of view (one second in the above example). The ADU equates to the aggregation interval used in Chang *et al* [Chang99] and has also been called a cryptoperiod when measured in units of time. ADU size is application and security scenario dependent. It may be an initialisation frame and its set of associated 'P-frames' in a video sequence or it may be ten minutes of access to a network game. Note that the ADU from a security point of view can be different from that used at a different layer of the application. ADU size can vary throughout the duration of a stream dependent on the content. ADU size is a primary determinant of system scalability. If a million receivers were to join within fifteen minutes, but the ADU size was also fifteen minutes, this would only require one re-key event.

However, reduction in re-keying requirements isn't the only scalability issue. In the above example, a system that can handle a million requests in fifteen minutes still has to be provided, even if its output is just one re-key request to the senders. With just such scalability problems in mind, many multicast key management architectures introduce a key manager role as a separate concern from the senders. This deals with policy concerns over membership and isolates the senders from much of the messaging traffic needed for access requests.

2.1 Related Work

Ballardie suggests exploiting the same scalability technique used for the underlying multicast tree, by delegating key distribution along the chain of routers in a core based multicast routing tree [IETF_RFC1949]. However, end-to-end security suffers from the complexity of requiring edge customers to entrust their keys to many intermediate network providers requiring a long chain of security associations. The Iolus system [Mittra97] sets up a similar distribution hierarchy, but only involving trusted end-systems. However, these gateway nodes also partition re-keying side effects by decrypting and re-encrypting the stream localising sub-group keys. This introduces a latency burden on every packet in the stream and requires strategically placed intermediate systems to volunteer their processing resource.

An alternative class of approaches involves a single key for the multicast data, but a hierarchy of keys under which to send out a new key over the same multicast channel as the data. These approaches involve a degree of redundant re-keying traffic arriving at every receiver in order for the occasional message to arrive that is decipherable by that receiver. The state of the art in this class is Chang *et al* [Chang99]. The group members are arranged as the leaves of a binary tree with the group session key at the root. Two auxiliary keys are assigned per layer of the tree. If each member is assigned a different user identity number (UID) this effectively assigns a pair of auxiliary keys to each bit in the UID space. The first of each pair is given to users with a 1 at that bit position in their UID and the other when there is a 0. When a single member leaves, a new group session key is randomly generated and multicast encrypted with every auxiliary key in the tree except those held by the leaving member. This guarantees (aside from the reliability of multicast) that every remaining member will get at least one message they can decrypt. A variant recognises the potential for aggregation of member removals if many occur within the timespan of one ADU. The group session key is multicast to the group multiple times, each encrypted with different logical combinations of the auxiliary keys in order to ensure all members but the leaving ones can decrypt at least one message. Finding this minimised set has the same solution as the familiar problem of reducing the number of logic gates and inputs in the field of logic hardware design. Wong *et al* [Wong98] take an approach that is a generalisation of Chang *et al*, analysing key graphs as a general case of trees. They find a tree of degree four rather than binary is the most efficient for large groups. The standardised approach to pay-TV key management also falls into this class [ITU-R.810]. A set of secondary keys is created and each receiver holds a sub-set of these in tamper-resistant storage. The group key is also unknown outside the tamper-resistant part of the receiver. In case the group key becomes compromised, a new one is regularly generated and broadcast multiple times under different secondary keys to ensure the appropriate receivers can re-key.

All work in this class of approaches uses multicast itself as the transport to send new keys. As 'reliable multicast' is still to some extent a contradiction in terms, all such approaches have to allow for some receivers missing the occasional multicast of a new key due to localised transmission losses. Some approaches include redundancy in the re-keying to allow for losses, but this reduces their efficiency and increases their complexity. Others simply ignore the possibility of losses, delegating the problem to a choice of a sufficiently reliable multicast scheme.

The Nark scheme [Briscoec99] falls into the same class as the present work because the group key is systematically changed for each new ADU in a stream. However, unlike with the present approach, a smartcard happens to be required to give non-repudiation of delivery and latency so its presence can also be exploited to control which keys in the sequence to reveal. Each receiver has a proxy of the sender running within her smartcard, so all smartcards can be sent one primary seed for the whole key sequence. The proxy on the smartcard then determines which keys to give out depending on the policy it was given by the key manager when the receiver set up the session. The present paper shows how to construct a key sequence such that it can be partially reconstructed from intermediate seeds, thus removing the need for a smartcard if non-repudiation is not a requirement.

Beyond the requirement we focus on, two taxonomies of multicast security requirements [Bagnall99, Canetti99] include many other possible combinations of security requirements for multicast. It is generally agreed that a modular approach is required to building solutions for combined requirements, rather than searching for a single monolithic 'super-solution'. Later, as examples of this modular approach, we show how a number of variations can be added to our basic key management schemes to achieve a selection of the more commercially important requirements.

3. Sender-Decoupled Architecture

We now describe a large-scale network game scenario to explain why systematic key changes allow sender decoupling, giving the scalability benefits asserted in the introduction. This motivates the need for key sequences that can initially be built from a small number of seeds. Use of a practical example also clarifies why it must be possible to reveal arbitrary portions of the key sequence to different customers. This motivates the need for reconstruction of any sub-range of the key sequence, also from a small number of intermediate seeds.

We deliberately choose an example where the financial value of an ADU (defined in Section 2) doesn't relate to time or data volume, but only to a completely application-specific factor. In this example, participation is charged per 'game-minute', a duration that is not strictly related to real-time minutes, but is defined and signalled by the game time-keeper. The game consists of many virtual zones, each moderated by a different zone controller. The zone controllers provide the background events and data that bring the zone to life. They send this data encrypted on a multicast address per zone, but the same ADU index and hence key is used at any one time in all zones. Thus the whole game is one single 'secure multicast session' (defined in Section 2) despite being spread across many multicast addresses. Players can tune in to the background data for any zone as long as they have the current key. The foreground events created by the players in the zone are not encrypted, but they are meaningless without reference to this background data.

Fig 3.1 only shows data flows relevant to game security and only those once the game is in progress, not during set-up. Clearly all players are sending data, but the figure only shows encrypting senders, S - the zone controllers. Similarly, only receivers that decrypt, R, are shown - the game players. A game controller sets up the game security (not shown but described below). Key management operations are delegated to a number of replicated key managers, KM, that use secure Web server technology.

The key to the secure multicast session is changed every game-minute (every ADU) in a sequence. All encrypted data is headed by an ADU index in the clear, which refers to the key needed to decrypt it. After the set-up phase, the game controller, zone controllers and key managers hold initial seeds that enable them to calculate the sequence of keys to be used for the entire duration of the game.

Game set-up

1. The game controller (not shown) unicasts a shared 'control session key' to all KM and S after satisfying itself of the authenticity of their identity. The easiest way to do this is for all S and KM to run secure Web servers so that the session key can be sent to each of them encrypted with each public key using client authenticated secure sockets layer (SSL) communications [Frier96]. The game controller also notifies all KM and S of the multicast address it will use for control messages, which they immediately join.
2. The game controller then generates the initial seeds to construct the entire key sequence and multicasts them to all KM and all S, encrypting the message with the control session key and using a reliable multicast protocol suitable for the probably small number of targets involved.
3. The game is announced in an authenticated session directory announcement [Handley97] regularly repeated over multicast (not shown). The announcement protocol is enhanced to include details of key manager addresses and the price per game-minute. Authenticated announcement prevents an attacker setting up

spoof payment servers to collect the game's revenues. Key managers as well as receivers listen to this announcement, in order to get the current price of a game-minute.

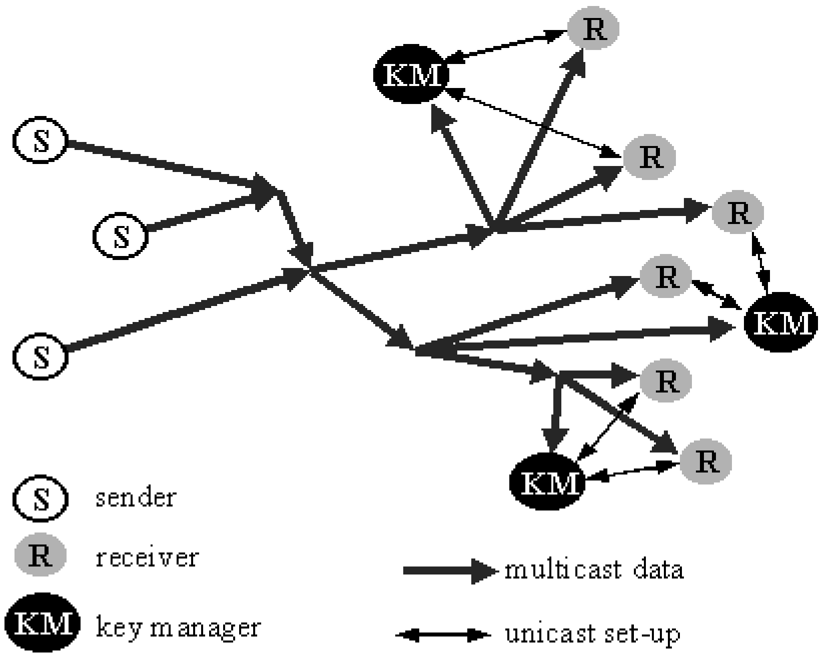


Fig 3.1 - Key management system design

Receiver session set-up, duration and termination

1. A receiver that wishes to pay to join the game, having heard it advertised in the session directory, contacts a KM Web server requesting a certain number of game-minutes using the appropriate form. This is shown as 'unicast set-up' in Fig 3.1. R pays the KM the cost of the requested game-minutes, perhaps paying in some form of e-cash or in tokens won in previous games. In return, KM sends a set of intermediate seeds that will allow R to calculate just the sub-range of the key sequence that she has bought. The key sequence construction described in the next section makes this possible efficiently. All this would take place over SSL with only KM needing authentication, not R.
2. R generates the relevant keys using the intermediate seeds she has bought.
3. R joins the relevant multicasts determined by the game application, one of which will always be the encrypted background zone data from one S. R uses a key from the sequence calculated in the previous step to decrypt these messages, thus making the rest of the game data meaningful.
4. Whenever the time-keeper signals a new game-minute (over the control multicast), all the zone controllers increment their ADU index and use the next key in the sequence. They all use the same ADU index. Each R notices that the ADU index in the messages from S has been incremented and uses the appropriate next key in the sequence.

5. When the game-minute index approaches the end of the sequence that R has bought, the application gives the player an 'Insert coins' warning before she loses access. The game-minutes continue to increment until the point is reached where the key required is outside the range that R can feasibly calculate. If R has not bought more game-minutes, she has to drop out of the game.

This scenario illustrates how senders can be completely decoupled from all receiver join and leave activity as long as key managers know the financial value of each ADU index or the access policy to each ADU through some pre-arrangement. There is no need for any communication between key managers and senders during the session. Senders certainly never need to hear about any receiver activity. If key managers need to avoid selling ADUs that have already been transmitted, they merely need to synchronise with the changing stream of ADU sequence numbers from senders. In the example, key managers synchronise by listening in to the multicast data itself. In other scenarios, it may be possible for synchronisation to be purely time-based, either via explicit synchronisation signals or implicitly by time-of-day synchronisation. In yet other scenarios (e.g. multicast distribution of commercial software), the time of transmission may be irrelevant. For instance, the transmission may be regularly repeated, with receivers being sold keys to a part of the sequence that they can tune in to at any later time.

In this example, pre-payment is used to buy seeds. This ensures key managers hold no state about their customers. This means they can be infinitely replicated as no central state repository is required, as would otherwise be the case if seeds were bought on account and the customer's account status needed to be checked. Thus performance can be linear with key manager replication and system resilience is independent of key manager resilience.

4. Key Sequence Construction

The following notations are used:

- `b(s)` is the notation used for a function that blinds the value of `s`. That is, a computationally limited adversary cannot find `s` from `b(s)`. An example of a blinding or one-way function is a hash function such as MD5 [IETF_RFC1321] or the standard Secure Hash 1 [NIST_Sha-1]. Good hash functions typically require only lightweight computational resources. Hash functions are designed to reduce an input of any size to a fixed size output. In all cases, we will use an input that is already the same size as the output, merely using the blinding property, not the size reduction property of the hash.
- `r(s)` is any computationally fast one-to-one function that maps from a set of input values to itself. A circular (rotary) bit shift is an example of such a function.

4.1 Binary Hash Tree (BHT)

The binary hash tree requires two blinding functions, $b_0()$ and $b_1()$, to be well-known. We will term these the 'left' and the 'right' blinding functions. Typically they could be constructed from a single blinding function, $b()$, by applying one of two simple one-to-one functions, $r_0()$ and $r_1()$ before the blinding function. As illustrated in Fig 4.1.1.

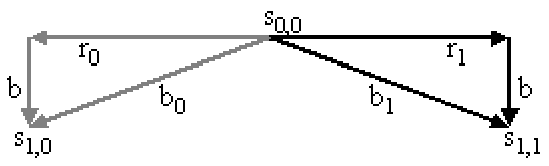


Fig 4.1.1 - Two blinding functions from one.

Thus: $b_0(s) = b(r_0(s));$ $b_1(s) = b(r_1(s))$

For instance, the first well-known blinding function could be a one bit left circular shift followed by an MD5 hash, while the second blinding function could be a one bit right circular shift followed by an MD5 hash. Other alternatives might be to precede one blinding function with an XOR with 1 or a concatenation with a well-known word. It seems advantageous to choose two functions that consume minimal but equal amounts of processor resource as this balances the load in all cases and limits the susceptibility to covert channels that would otherwise appear given the level of processor load would reveal the choice of function being executed. Alternatively, for efficiency, two variants of a hash function could be used, e.g. MD5 with two different initialisation vectors. However, it seems ill advised to tamper with tried-and-tested algorithms. The key sequence is constructed as follows:

1. The sender randomly generates an initial seed value, $s(0,0)$. As a concrete example, we will take its value as 128 bits wide.
2. The sender decides on the required maximum tree depth, D, which will lead to a maximum key sequence length, $N_0=2^D$ before a new initial seed is required.
3. The sender generates two 'left' and 'right' first level intermediate seed values, applying respectively the 'left' and the 'right' blinding functions to the initial seed:
   $s(1,0) = b_0(s(0,0));$ $s(1,1) = b_1(s(0,0))$.
   The sender generates four second level intermediate seed values:
   $s(2,0) = b_0(s(1,0));$ $s(2,1) = b_1(s(1,0));$
   $s(2,2) = b_0(s(1,1));$ $s(2,3) = b_1(s(1,1))$,
   and so on, creating a binary tree of intermediate seed values to a depth of D levels.
   Formally, if $s(d,i)$ is an intermediate seed that is d levels below the initial seed, $s(0,0)$:
   $$s(d,i) = b_p(s(d-1, \lfloor i/2 \rfloor)) \qquad (4.1.1)$$
   where $p=i \bmod 2$ (see Appendix B for notation)
4. The key sequence is then constructed from the seed values across the leaves of the tree. Strictly, the stream cipher in use may not require 128b keys, therefore a shorter key may be derived from the leaf seeds by truncation of the most (or

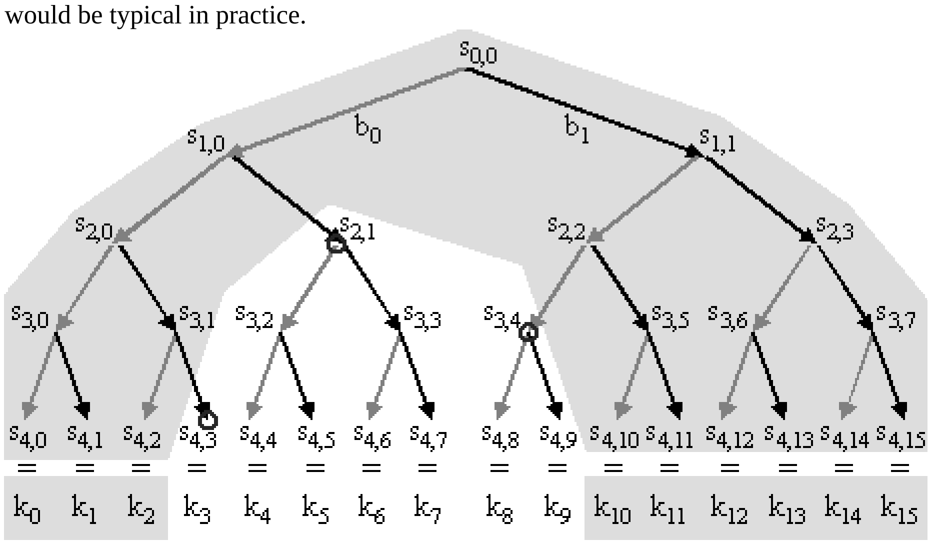
least) significant bits, typically to 64b. The choice of stream cipher is irrelevant as long as it is fast and secure.
That is, if `D=5`, $k_0 = s(5,0)$; $k_1 = s(5,1)$; ... $k_{31} = s(5,31)$.
Formally, $k_i = s(D,i)$ (4.1.2)

5. The sender starts multicasting the stream, encrypting ADU_0 with k_0, ADU_1 with k_1 etc. but leaving at least the ADU sequence number in the clear.
6. If the sender delegates key management, it must privately communicate the initial seeds to the key managers.

A receiver reconstructs a portion of the sequence as follows:

1. When a receiver is granted access from ADU_m to ADU_n, the sender (or a key manager) unicasts a set of seeds to that receiver (e.g. using SSL). The set consists of the intermediate seeds closest to the tree root that enable calculation of the required range of keys without enabling calculation of any key outside the range.
These are identified by testing the indexes, `i`, of the minimum and maximum seed using the fact that an even index is always a 'left' child, while an odd index is always a 'right' child. A test is performed at each layer of the tree, starting from the leaves and working upwards. A 'right' minimum or a 'left' maximum always needs revealing before moving up a level. If a seed is revealed, the index is shifted inwards by one seed. To move up a layer, the minimum and maximum indexes are halved with the maximum rounded down. The odd/even tests are repeated on the new indexes, revealing a 'right' minimum or 'left' maximum as before. The process continues until the minimum and maximum cross or meet. They can cross after either or both have been shifted inwards. They can meet after they have both been shifted upwards, in which case the seed where they meet needs revealing before terminating the procedure.
This procedure is described more formally, in C-like code in Appendix A
2. Clearly, each receiver needs to know where each seed that it is given resides in the tree. The seeds and their indexes can be explicitly paired when they are revealed. Alternatively, to reduce the bandwidth required, the protocol may specify the order in which seeds are sent so that each index can be calculated implicitly from the minimum and maximum index and the order of the seeds. This is possible because there is only one minimal set of seeds that allows re-creation of any one range of keys.
Each receiver can then repeat the same pairs of blinding functions on these intermediate seeds as the sender did to re-create the sequence of keys, k_m to k_n. (Equations `4.1.1` & `4.1.2`)
3. Any other receiver can be given access to a completely different range of ADUs by being sent a different set of intermediate seeds.

The creation of a key sequence with `D=4` is graphically represented in Fig 4.1.2. As an example, we circle the relevant intermediate seeds that allow one receiver to re-create the key sequence from $k_3$to k_9. The seeds and keys that remain blinded from this receiver are shown on a grey background. Of course, a value of D greater than 4 would be typical in practice.



$s_{0,0}$
b_0
b_1
$s_{1,0}$
$s_{1,1}$
$s_{2,0}$
$s_{2,1}$
$s_{2,2}$
$s_{2,3}$
$s_{3,0}$
$s_{3,1}$
$s_{3,2}$
$s_{3,3}$
$s_{3,4}$
$s_{3,5}$
$s_{3,6}$
$s_{3,7}$
$s_{4,0}$ $s_{4,1}$ $s_{4,2}$ $s_{4,3}$ $s_{4,4}$ $s_{4,5}$ $s_{4,6}$ $s_{4,7}$ $s_{4,8}$ $s_{4,9}$ $s_{4,10}$ $s_{4,11}$ $s_{4,12}$ $s_{4,13}$ $s_{4,14}$ $s_{4,15}$
k_0 k_1 k_2 k_3 k_4 k_5 k_6 k_7 k_8 k_9 k_{10} k_{11} k_{12} k_{13} k_{14} k_{15}

Fig 4.2.2 - Binary hash tree

Note that each layer can be assigned an arbitrary value of d as long as it uniquely identifies the layer. Nothing relies on the actual value of d or D. Therefore it is not necessary for the sender to reveal how far the tree extends upwards, thus improving security.

Often a session will have an unknown duration when it starts. Clearly, the choice of D limits the maximum length of key sequence from any one starting point. The simplest work-round is just to generate a new initial seed and start a new binary hash tree alongside the old if it is required. If D is known by all senders and receivers, a range of keys that overflows the maximum key index, 2^D, will be immediately apparent to all parties. In such cases it would be sensible to allocate a 'tree id' for each new tree and specify this along with the seeds for each tree.

5. Discussion

5.1 Storage and Processing Costs

The general approach is to use a small number of seeds to generate a larger number of keys, both at the sender before encryption and at the receiver before decryption. In either case, there may be limited memory capacity for the key sequence, which appears to require exponentially more memory than the seeds. We will now show that the tree construction requires minimal memory and minimal processing at either the sender or the receiver as each new key in the sequence is calculated. We assume the keys are used sequentially and once a key has been used it will never be required again. After this we will discuss the trade-offs between storage and processing that key managers may make, given that they have to be able to serve seeds from arbitrary points in the future tree at any time.

For senders and receivers using the BHT, it is most efficient to only store the seeds on the branch of the tree from a root to that key *following* the one currently in use. Note that there may be multiple roots, particularly for receivers, where each revealed seed is a root. In practice this principle translates into being able to deallocate memory for a parent seed immediately it has been hashed to produce its right child. If leaf seeds are also deallocated as soon as the next in the sequence is in use, this will ensure the tree only holds `log(N)` seeds in memory on top of any revealed seeds being held to generate the rest of the tree to the right of the current key.

Re-using the earlier example in Fig 4.2.2, we will now follow the key calculation sequence step-by-step. For brevity we will assume keys are synonymous with their corresponding leaf seeds:

1. `s(4,3)` is immediately available as one of the revealed seeds.
2. `s(4,4)` requires two hash operations from `s(2,1)`. The value of `s(3,2)` calculated on the way should be stored.
3. `s(4,3)` may be deallocated once `s(4,4)` is in use
4. `s(4,5)` requires one hash of the stored `s(3,2)`
5. `s(4,4)` and `s(3,2)` may then be deallocated
6. `s(4,6)` requires two hashes from `s(2,1)`. Again the value of `s(3,3)` calculated on the way should be stored.
7. `s(2,1)` may be deallocated as soon as it has been hashed
8. `s(4,5)` may be deallocated as soon as `s(4,6)` is in use
9. The process continues along similar lines until `s(4,9)` is finished with, when it is deallocated leaving no further seeds in memory.

It will be noted that, if the above seed storage strategy is adopted, one hash operation is required per key on the seeds in the penultimate layer, one hash every two keys on the next layer up, one hash every four keys on the next layer and so on. In other words, no branch of the tree ever requires the hash to be calculated more than once. Therefore:

$$\begin{aligned}\text{(mean no. of hashes per key)} &= \text{(no. of branches) / (no. of leaves)}\\ &= (2^{(D+1)} - 1) / 2^{D}\\ &< 2\end{aligned}$$

If memory is extremely scarce (e.g. an embedded device) but some clock cycles are spare, storage can be traded off against processing. Any intermediate seeds down the branch of the tree to the current key need to be calculated, but they don't all need to be stored. Those closest to the leaves should be stored (cached), as they will be needed soonest to calculate the next few keys. As intermediate seeds nearer to the root are required, they can be recalculated as long as the seeds originally sent by the key manager are never discarded until the sequence has left them behind.

Unlike senders or receivers, a key manager cannot guarantee to only access the key-space sequentially. It will have to respond to requests for seeds from anywhere in the tree. However, for most scenarios it is likely that requests will tend not to be randomly distributed. Therefore, a key manager can use an identical approach to the device with scarce memory. It can calculate seeds in any part of the tree from the initial seeds, but cache those being most frequently used. This simply requires a fixed size cache memory allocation and discard of the least recently used values in the store.

5.2 Efficiency

Table 5.2.1 shows various performance parameters of the BHT per secure multicast session, where:

- R, S and KM are the receiver, sender and key manager, respectively, as defined in Section 3
- `N (= n-m+1)` is the length of the range of keys that the receiver requires, randomly positioned in the key space
- w_s is the size of a seed (typically 128b)
- w_h is the size of the key management protocol header overhead
- t_s is the processor time to blind a seed (plus one relatively negligible circular shifting operation)

			BHT
per R	(unicast message size)/$w_s - w_h$ or (min storage)/w_s	min	`1`
		max	`2(log(N+2) - 1)`
		mean	`O(log(N) - 1)`
per R	(processing latency)/t_s	min	`0`
		max	`log(N)`
		mean	`O(log(N) /2)`
per R or S	(processing per key)/t_s	min	`1`
		max	`log(N)`
		mean	`2`
per S or KM	(min storage)/w_s		`1`
per S	(min random bits)/w_s		

Table 5.2.1 - Efficiency parameters of the BHT per secure multicast session

The unicast message size for each receiver's session set-up is shown equated to the minimum amount of storage each receiver requires. This is the storage required before starting the session, not once keys have started to be calculated. The minimum sender storage row has the same meaning. The processing latency is the time required for one receiver to be ready to decrypt incoming data after having received the unicast set-up message for its session. Note that there is no latency cost when other members join or leave, as in schemes that cater for unplanned eviction. The figures for processing per key assume sequential access of keys and the caching strategy described in Section

5.1. The exceptional cases when a session starts or ends are not included in the figures for per key processing. Only the sender (or a group controller if there are multiple senders) is required to generate random bits for the initial seeds. The number of bits required is clearly equal to the minimum sender storage of these initial seeds.

It can be seen that the only parameters that depend on the size of the group membership are those that are per receiver. The cost of two of these (storage and processing latency) is distributed across the group membership thus being constant per receiver. Only the unicast message size causes a cost at a key manager that rises linearly with group membership size, but the cost is only borne once per receiver session. Certainly, none of the per receiver costs are themselves dependent on the group size as in all schemes that allow unplanned eviction. Thus, the BHT construction is highly scalable.

5.3 Security

Each seed in the tree is potentially twice as valuable as its child. Therefore, there is an incentive to exhaustively search the seed space for the correct value that blinds to the current highest known seed value in the tree. For the MD5 hash, this will involve 2^{127} MD5 operations on average. It is possible a value will be found that is incorrect but blinds to a value that collides with the known value (typically one will be found every 2^{64} operations with MD5). This will only be apparent by using the seed to produce a range of keys and testing one on some data supposedly encrypted with it. Having succeeded at breaking one level, the next level will be twice as valuable again, but will require the same brute-force effort to crack. Note that one MD5 hash (portable source) of a 128b input takes about 4us on a Sun SPARCserver-1000. Thus, 2^{128} MD5s would take 4e25 years. MD5 optimised for its host architecture is about twice as fast.

Generally, the more random values that are needed to build a tree, the more it can contain sustained attacks to within the bounds of the sub-tree created from each new random seed. However, for long-running sessions, there is a trade-off between security and the convenience of a continuous key-space (as against concatenating BHTs side-by-side described earlier). The randomness of the randomly generated seeds is another potential area of weakness that must be correctly designed.

Any key sequence construction like that discussed here is vulnerable to collusion between valid group members. If a sub-group of members agree amongst themselves to each buy a different range of the key space, they can all share the seeds they are sent so that they can all access the union of their otherwise separate key spaces. Arbitrage is a variant of member collusion that has already been discussed. This is where one group member buys the whole key sequence then sells portions of it more cheaply than the selling price, still making a profit if most keys are bought by more than one customer. Protection against collusion with non-group members is discussed in Section 6.2 on watermarking.

Finally, the total system security for any particular application clearly depends on the strength of the security used when setting up the session. The example scenario in Section 3 describes the issues that need to be addressed and suggests standard cryptographic techniques to meet them. As always, the overall security of an application is as strong as the weakest part, which is more likely to be some 'human' element than the key sequence construction discussed here.

6. Requirement Variations

The key management scheme described in the current work lends itself to modular combination with other mechanisms to meet the additional commercial requirements described below.

6.1 Multi-Sender Multicast

A multi-sender multicast session can be secured using the BHT as long as all the senders arrange to use the same key sequences. They need not all simultaneously be using the same key as long as the keys they use are all part of the same sequence. Receivers can know which key to use even if each sender is out of sequence with the others as long as the ADU index is transmitted in the clear as a header for the encrypted ADU. The example scenario in Section 3 described how multiple senders might synchronise the ADU index they were all using if this was important to the commercial model of the application. If each sender in a multi-sender multicast uses different keys or key sequences, each sender is creating a different secure multicast session even if they all use the same multicast address. This follows from the distinction between a multicast session and a secure multicast session defined in Section 2.

6.2 Watermarked Audit Trail

Re-multicast of received data requires very low resources on the part of any receiver. Even if the value of the information received is relatively low there is always a profit to be made by re-multicasting data and undercutting the original price (arbitrage), as proved in Herzog *et al* [Herzog95]. In general, prevention of information copying is considered infeasible; instead most attention focuses on the more tractable problem of copy detection by uniquely 'watermarking' each copy of a work. If a watermarked copy is later discovered, it can be traced back to its source, thus deterring the holders of original copies from passing on further, illicit copies. Watermarks are typically applied to the least significant bits of a medium to avoid significantly degrading the quality. An approach such as Chameleon [Anders97] can be used to watermark the *keys* used to decrypt the stream of data and can therefore be combined with keys from the BHT.

In Chameleon a stream is ciphered by combining a regular stream cipher with a large block of bits (512kB in Chameleon's concrete example). Each receiver is given a long-term copy of the block to decipher the stream. The block is watermarked for each receiver in a way specific to the medium. Because the block is only used for the XOR operation, the position of any watermarked bits is preserved in the output, allowing the approach to be generic. Thus, the keys generated by the BHT construction can be treated as a sequence of intermediate keys from which a watermarked sequence of final keys is generated, thus enforcing watermarked decryption.

However, this approach suffers from an applicability limitation of Chameleon that has not been previously discussed to our knowledge. Chameleon doesn't detect 'semi-internal' leakage to users who legitimately hold a valid long term key block. Intermediate keys, rather than final ones, can be leaked to any such receiver. For instance, in the above network game example, a group of players can collude to each buy a different game-hour and share the (unwatermarked) intermediate keys that each buys between themselves. Thus a receiver not entitled to certain of the intermediate keys can create final keys watermarked with her own key block and hence decrypt the cipherstream. Although the keys and data produced are stamped with her own watermark, this only gives an audit trail to the target of the leak, not the source (shutting the stable door after the horse has bolted).

Chameleon *does* nonetheless create an audit trail for any keys or data that are passed to a completely unauthorised receiver - that is a receiver without a long-term key block, e.g. someone who has not played the game recently. In such cases the traitor who revealed the keys or data can be traced if the keys or data are traced. Similarly, there is an audit trail if one of the players passes on their long-term key block instead, as it also contains a watermark traceable to the source of the leak. Thus Chameleon 'raises the bar' against leakage, and is therefore still a valid candidate for modular combination with BHT.

6.3 Unplanned Eviction

As already pointed out, the BHT allows for eviction from the group at arbitrary times, but only if planned at the time each receiver session is set up. If pre-planned eviction is the common case, but occasionally unplanned evictions are needed, keys from the BHT can be combined with another scheme, such as LKH++ [Chang99] to allow the occasional unplanned eviction. To achieve this, as with watermarking above, keys from the sequence generated by the BHT are treated as intermediate keys. These are combined (e.g. XORed) with a group key distributed using for example LKH++ to produce a final key used for decrypting the data stream. Thus both the BHT intermediate key and the LKH++ intermediate key are needed to produce the final key at any one time.

Indeed, any number of intermediate keys can be combined (e.g. using XOR) to meet multiple requirements simultaneously. For instance, MARKS, LKH++ and Chameleon intermediate keys can be combined to simultaneously achieve low cost planned eviction, occasional unplanned eviction and a watermarked audit trail against leakage outside the long-term group.

Formally, the final key, $k_{i,j,...} = c(k'_i, k'_j, ...)$, where intermediate keys `k'` can be generated from sequences using a BHT construction or any other means such as Chameleon or LKH++ and `c()` is a combining function, such as XOR.

In general, combination in this way produces an aggregate scheme with storage costs that are the sum of the individual component schemes. However, combining LKH++ with MARKS, where most evictions are planned, cuts out all the re-keying messages of LKH++ unless an unplanned eviction is actually required.

7. Limitations and Further Work

Duplication of information costs so little that selling multiple copies at a unit price much greater than the cost of duplication always results in an economic incentives for potential buyers to collude. We discuss receiver collusion and arbitrage in Sections 5.3 & 6.2 but the best solution we can offer without requiring smartcards only offers the possibility of detecting collusion between a group member and a non-member.

Detecting intra-group collusion without requiring specialist hardware is left for further work.

We have assumed that knowledge of more than one value blinded in different ways from the same starting value doesn't lead to an analytical solution to calculate the original value. Until proofs exist showing any blinding function is resistant to analytical (as against brute force) attack, it won't be possible to prove whether an analytical attack has been made easier by our techniques.

Finally, through pressure of time, we have avoided analysis of trees of degree three and above. They potentially offer greater efficiency at the expense of additional complexity. For instance the experiments in Wong *et al* recommend a tree of degree four, but the pattern of usage that their tree is subjected to is only tenuously related to the present work.

7. Conclusion

We have presented a solution to manage the keys of very large groups. It preserves the scalability of receiver initiated Internet multicast by completely de-coupling senders from all receiver join and leave activity. Senders are also completely decoupled from the key managers that absorb this receiver activity. We have shown that many commercial applications have models that only need stateless key managers, in which cases unlimited key manager replication is feasible. These gains have been achieved by the use of systematic group key changes rather than receiver join or leave activity driving re-keying. Decoupling is achieved by senders and key managers pre-arranging the unit of financial value in the multicast data stream (the 'application data unit' with respect to charging). Using this model, there is zero side effect on other receivers (or on the senders) when one receiver joins or leaves. We also ensure multicast is not used for key management, only for bulk data transfer. Thus, re-keying isn't vulnerable to random transmission losses, which are complex to repair scalably when using multicast.

State of the art techniques that allow unplanned eviction from the group are still costly in messaging terms. In contrast we have focussed on the problem of planned eviction. That is, eviction per receiver after some arbitrary future ADU, but planned at the time the receiver requests a session. We have asserted that many commercial scenarios based on pre-payment or subscription don't require unplanned eviction but do require arbitrary planned eviction. Examples are pay-TV, pay-per-view TV or network gaming. To achieve planned but arbitrary eviction we have designed a key sequence construction that is used by the senders to systematically change the group key. It is designed such that an arbitrary sub-range of the sequence can be reconstructed by revealing a small number of seeds (16B each). We can reveal `N` keys to each receiver using `O(log(N))` seeds. The scheme requires on average just `O(log(N) /2)` fast hash operations to get started, then on average no more than just two more hashes to calculate each new key in the sequence. This implies under 10us of processing time to generate each ADU key with today's technology.

To put this work in context, for pay TV charged per second with 10% of ten million viewers tuning in or out within a fifteen minute period, the best alternative scheme (Chang *et al*) might generate a re-key message of the order of tens of kB every second, multicast to every group member. The present work requires a message of a few hundred bytes unicast just once to each receiver at the start of perhaps four hours of viewing. This comparison is not strictly fair as, unlike the present scheme, Chang *et al* and the other schemes of its class allow for unplanned eviction from the group, thus allowing accurate charging for serendipitous viewing. However, the purpose of this work is to present a far more scalable solution for commercial scenarios where unplanned eviction is not required. Another way of putting this is that the cost of scenarios requiring unplanned eviction might make them economically unviable compared to those that can make do with planned eviction.

Nonetheless, if unplanned eviction is occasionally required, we have shown how to combine our scheme with Chang's to get the best of both worlds. Combining schemes sums the storage requirements of each, but both are very low in this respect. We also show how to further combine with the Chameleon watermarking scheme to give rudimentary detection of information leakage outside the group.

Acknowledgements

Jake Hill, Ian Fairman, David Parkinson (BT)

References

[Anders97] Ross Anderson & Charalampos Manifavas (Cambridge Uni), "Chameleon - A New Kind of Stream Cipher" Encryption in Haifa (Jan 1997), <URL:http://www.cl.cam.ac.uk/ftp/users/rja14/chameleon.ps.gz>

[Bagnall99] Pete Bagnall, Bob Briscoe & Alan Poppitt, (BT), "Taxonomy of Communication Requirements for Large-scale Multicast Applications", Internet Draft (work in progress), Internet Engineering Task Force (17 May 1999) <draft-ietf-lsma-requirements-03.txt>

[Briscoec99] Bob Briscoe & Ian Fairman (BT), "Nark: Receiver-based Multicast Non-repudiation and Key Management", forthcoming in ACM conference on Electronic Commerce (Nov 1999), <URL:http://www.labs.bt.com/projects/mware/>

[Briscoee99] Bob Briscoe (BT), "MARKS: Zero Side Effect Multicast Key Management using Arbitrarily Revealed Key Sequences", BT Technical Report (Aug 1999), <URL:http://www.labs.bt.com/projects/mware/>

[Canetti99] Ran Canetti (IBM T.J. Watson), Juan Garay (Bell Labs), Gene Itkis (NDS), Daniele Micciancio (MIT), Moni Naor (Weizmann Inst. of Science), Benny Pinkas (Weizmann Inst. of Science), "Multicast Security: A Taxonomy and Efficient Constructions", Proceedings IEEE Infocomm'99, Vol2 708-716 (Mar 1999), <URL:http://www.wisdom.weizmann.ac.il/~bennyp/PAPERS/infocom.ps>

[Chang99] Isabella Chang, Robert Engel, Dilip Kandlur, Dimitrios Pendarakis, Debanjan Saha, (IBM T.J. Watson Research Center) "Key Management for Secure Internet Multicast using Boolean Function Minimization Techniques", Proceedings IEEE Infocomm'99, Vol2 689-698 (Mar 1999), <URL:http://www.research.ibm.com/people/d/debanjan/papers/infocom99.srm.pdf>

[Deering91] S. Deering, “Multicast Routing in a Datagram Network,” PhD thesis, Dept. of Computer Science, Stanford University, (1991).

[Frier96] A. Frier, P. Karlton and P. Kocher, (Netscape), "The SSL 3.0 Protocol", Nov 18, 1996.

[Handley97] Mark Handley (UCL), "On Scalable Internet Multimedia Conferencing Systems", PhD thesis (14 Nov 1997) <URL:http://www.aciri.org/mjh/thesis.ps.gz>

[Herzog95] Shai Herzog (IBM), Scott Shenker (Xerox PARC), Deborah Estrin (USC/ISI), "Sharing the cost of Multicast Trees: An Axiomatic Analysis", in Proceedings of ACM/SIGCOMM '95, Cambridge, MA, Aug. 1995, <URL:http://www.research.ibm.com/people/h/herzog/sigton.html>

[IETF_RFC1321] Ronald L. Rivest, "The MD5 Message-Digest Algorithm", Request for Comments (RFC) 1321, Internet Engineering Task Force (1992) <URL:rfc1321.txt>

[IETF_RFC1949] Tony Ballardie, "Scalable multicast key distribution", Request for Comments (RFC) 1949, Internet Engineering Task Force (May 1996) <URL:rfc1949.txt>

[ITU-R.810] ITU-R Rec. 810, "Conditional-Access Broadcasting Systems", (1992)

<URL:http://www.itu.int/itudocs/itu-r/rec/bt/810.pdf>
[McGrew98] McGrew, David A., & Alan T. Sherman, "Key establishment in large dynamic groups using one-way function trees," TIS Report No. 0755, TIS Labs at Network Associates, Inc., Glenwood, MD (May 1998). 13 pages.
[Mittra97] Suvo Mittra, "Iolus: A framework for scalable secure multicasting," Proceedings of the ACM SIGCOMM '97, 14-18 Sep 1997 Cannes, France.
[NIST_Sha-1] FIPS Publication 180-1, Secure hash standard, NIST, U.S. Department of Commerce, Washington, D.C. (April 1995).
[Wong98] Chung Kei Wong, Mohamed Gouda and Simon S Lam, "Secure Group Communications Using Key Graphs", Proceedings of ACM SIGCOMM'98 (Sep 98)
<URL:http://www.acm.org/sigcomm/sigcomm98/tp/abs_06.html>

Appendix A - Algorithm for Identifying Mminimum Set of Intermediate Seeds for BHT

In the following C-like code fragment

- the function `odd(x)` tests whether `x` is odd
- and the function `reveal(d,i)` reveals seed `s(d,i)` to the receiver

```
min=m; max=n;
for(d=D; ; d--) {         // working from leaves...
                          // move up tree 1 level ea loop
    if (min == max) {     // min & max have converged...
        reveal(d,min);    // ...so reveal sub-tree root..
        break;            // ...and quit
    }
    if odd(min) {         // odd min never left child...
        reveal(d,min);    // ...so reveal odd min seed
        min++;            // and step min in 1 to right
    }
    if !odd(max) {        // even max never right child..
        reveal(d,max);    // ...so reveal even max seed
        max--;            // and step max in 1 to left
    }
    if (min > max) break; // min & max cousins, so quit
    min/=2;               // halve min ...
    max/=2;               // ... & halve max ready for...
}                         // ... next level round loop
```

Appendix B - Notation

`O(x)` is notation for 'of order `x`'.
$\lfloor j/P \rfloor$ is notation for the value of `j/P` rounded down to the nearest integer (the floor function).
`j mod P` is notation for the remainder of `j/P`.