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. Solutions are presented for single and for multi-sender multicast. Further, we show how each receiver's data can be subject to an individual, watermarked audit trail. 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 each session end can be planned at the time each access is requested. It might therefore also be appropriate for virtual private networks or for information distribution on other duplicated media such as DVD.

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 to extract only the 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 the senders. Four constructions of sequences are presented where arbitrarily different portions of the sequence can be revealed to each receiver by only revealing a small number of intermediate seed values rather than having to reveal every key in the sequence. The first construction requires only two seed values to be revealed to each receiver in order to enable the reconstruction of any part of the sequence, however it is flawed for most practical applications for reasons that will be described later. The other three constructions avoid this flaw at the minimal extra expense of requiring a maximum of $O(\log(N))$ seeds to be revealed *once per session* to each receiver in order to reconstruct a part of the key 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 in the most lightweight scheme. (Mathematical notation used is explained in Appendix C.)

In contrast, whenever a receiver is added or removed in any of the constructions presented here, there is zero side effect on other receivers. A special group key change doesn't have to be initiated because systematic changes occur sufficiently regularly anyway. No keys need sending over the 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.

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 show how the occasional unplanned eviction can be catered for by modular combination with existing solutions at the expense of loss of simplicity and scalability.

By focussing on this pragmatic scenario, the single set-up message per receiver session can be with a highly replicated key manager rather than the sender. Key manager replication potential is limited if user account changes need to be synchronised across all replicas to prevent fraud. However the set-up exchange might, for instance, simply be a check that the receiver's identity entitles it to listen to information up to a certain security classification. Or a prepayment might give a certain viewing time. In both these scenarios and many others, key managers can be stateless allowing any amount of key manager replication. Thus performance can be linear with key manager replication and system resilience is independent of key manager resilience. The constructions we present decouple the sender from the key managers but the coupling between receivers and key managers depends on the commercial model of the application using the secure session. Key manager statelessness is therefore outside the scope of this paper.

In section 2, we discuss requirements and describe related work on multicast key management and other specific 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 four key sequence constructions that allow different portions of a key sequence to be reconstructed from various combinations of intermediate seeds. Section 5 discusses the efficiency and security of the constructions. Section 6 describes variations on the approach to add other security requirements such as multi-sender multicast, a watermarked audit trail and unplanned eviction. We also discuss how to apply the approach to non-multicast multi-party scenarios such as virtual private networks or circulation of data copies on DVD. Finally limitations of the approach are discussed followed by conclusions.

2. Background 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.

2.1 Multicast Key Management

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. For performance, an ADU may only be partially encrypted with the remainder in the clear [Kunkel97]. 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.

Ever since Internet multicast became a feasible proposition, scalability improvements have been sought over the earlier group key management schemes that scale linearly with group size (e.g. Ingemarsson *et al* [Ingemar82]). A second class of solutions extend Diffie-Hellman public key cryptography in order to act as a group key, but they require numbers of public key operations that also scale linearly with group size. 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, this suffers from a lack of end-to-end security, requiring edge customers to entrust their keys to many intermediate network providers. The Iolus system [Mittra97] sets up a similar distribution hierarchy, but only involving trusted end-systems. However, it also addresses the side effects of re-keying the whole group by requiring the gateway nodes in the hierarchy to decrypt and re-encrypt the stream with a new key known only to their local sub-group. 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 logical key hierarchy (LKH) [Wallner97] gives each receiver its own key then creates the same number of extra keys, one for each node of a binary tree of keys with each member's key at the leaves. For `n` receivers, LKH therefore requires the centre to store `O(2n)` keys but each receiver need only store `O(log(n))`. The root of the tree is the group key under which data is encrypted. When a member joins or leaves, all the keys on their branch to the root are replaced in one long message multicast to the whole tree. Each new key is included twice; each encrypted under one of the two keys below it. Each re-key message is therefore `O(2log(n))` times the key length. The closer a receiver is in the logical tree to the changed leaf key, the more decrypt operations it will need to extract its new keys from this message, the maximum being `O(log(n))`. Perlman has suggested an improvement to LKH, termed LKH+, where a one way function could be used by all those with knowledge of the existing key to compute the next one [Perlman]. The joining member would only be told the new key and not be able to work back to the old one. Unfortunately, the same argument cannot be applied for backward security. Wong *et al* [Wong98] take an approach that is a generalisation of LKH, analysing key graphs as a general case of trees. They find a tree similar to LKH (but of degree four rather than binary) is the most efficient for large groups.

The one-way function tree (OFT) technique [McGrew98] is in the same class of approaches as LKH. Incidentally, OFT is also presented in [Balen99], which is particularly notable for its comprehensive and accurate review of the literature. Like LKH, all members have their own key, and a binary tree of keys is built over them with the root also being the group key. However, the keys at each intermediate node are a combination of the hashes of the two keys below, rather than being freely generated. Thus Perlman's suggestion cannot be applied to OFT because the group key is not independent of the keys lower in the tree. As a result, LKH+ becomes more efficient than OFT in most scenarios. 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.

Chang *et al* [Chang99] also falls into this class but offers two advances over LKH+. We term it LKH++ for brevity. The group members are still arranged as the leaves of a binary tree with the group session key at the root, but instead of assigning an auxiliary key to each node, as in LKH, just 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. Thus storage per member is still `O(log(n))` but storage at the controller is `O(2log(n))` as opposed to `O(2n)` with LKH. 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. The auxiliary keys are then all changed by a hash keyed with the new group session key to ensure future rekeying messages cannot be opened by members who have left. This costs a message of `O(log(n))` as with LKH. The second improvement recognises the potential for aggregation of member removals if many occur within the timespan of one ADU. With this variation, 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. The combinations are chosen primarily to minimise the number of messages and secondarily to minimise the number of keys combined per 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. The resulting message size to re-key depends on which members leave but is typically smaller than the sum of all the otherwise discrete leave messages, and can be smaller than the message for a single member leaving.

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.

LKH+ [Perlman] is not actually in the same class of approaches as LKH and OFT. It relies on a pre-ordained new group key known in advance by all authorised recipients. Dillon's approach [Dillon97] falls into the same class. Documents transmitted over satellite are encrypted with a key generated from a one way hash of a seed keyed with the document ID. The seed is stored at the receiver in a tamper-resistant security engine having been transmitted under the public key of that security engine, the private key being installed in the security engine at manufacture. To request a document from a catalogue, the receiving computer requests that a key be generated by its associated security engine which it pre-loads into its satellite receiver, to be used when the next encrypted document broadcast is scheduled. Any documents that arrive with no corresponding key awaiting them are discarded. The distinguishing feature of this class of solutions is that the group key used to encrypt a document is specific to that document ID. Thus each broadcast document is encrypted using a different key rather than the key only being changed in synchrony with the addition or removal of receiver interest.

The Nark scheme [Briscoec99] also falls into this class. As with the present approach, 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 so it 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.

2.2 Multicast 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. It is possible to 'watermark' different copies of a copyrighted digital 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. Such bits are in different locations with different regularity in different media, therefore there is never likely to be a generic approach [Schneier96]. The most generic scheme discussed to date is Chameleon [Anders97]. In Chameleon a stream is ciphered by combining a regular stream cipher with a large block of bits. Each receiver is given a long-term copy of the block to decipher the stream. In the concrete example given, four 64b words in the 512kB block are chosen by indexing the block with the output of the regular stream cipher. Then all four are XORed together with each 64b word of the stream. The block given to each receiver is watermarked in a way specific to the medium. For instance, the least significant bit of every 16b word of an audio stream might be the only place where a watermark can be stored without degrading the content significantly. Because the block is only used for the XOR operation, the position of any watermarked bits is preserved in the output.

Naor *et al* [Naor98] formalises a pragmatic approach to such 'traitor tracing' by proposing a parameter that represents the minimum number of group members that need to collude to eliminate a watermark. The elimination criteria are that none of the conspirators are identifiable, and it is assumed that the copyright owner will want to avoid accusing innocent members. For instance, watermarking at least the square root of the total number of bits that could hold a watermark in the Chameleon scheme would protect against conspiracies of four or less members.

Watercasting [Brown99] is a novel, if rather convoluted way to embed an individual watermark in each receiver's copy of multicast data. Multicast forwarding is modified by including active networking elements at strategic branch points. These elements drop redundant data inserted into the original stream in order to produce a different drop pattern on each forwarded branch. A chain of trusted network providers is required for watercasting, each of which has to be willing to reveal their authenticated tree topology to each sender.

In this paper, for completeness, we report how it is possible to add an audit trail back to the copier of multicast information using watermarking. Our approach is not novel in this respect, simply re-using Chameleon. However, we include it to demonstrate our modular approach to the addition of mechanisms.

2.3 Other Requirements

Beyond the two requirements we have focussed on so far, two taxonomies of multicast security requirements [Bagnall99, Canetti99] include many other possible combinations of security requirements for multicast.

We have placed sender authentication outside the scope of this paper, but its importance merits a brief survey of the literature. A sender may merely need to prove it is one of the group of valid receivers in which case use of the group encryption key suffices. If receivers require each sender to authenticate their messages individually, public key signing leads to an unscalable solution because of the sheer volume of heavy asymmetric key operations required. Balenson *et al* [Balen99] and Canetti *et al* [Canetti99] both provide up to date reviews of more efficient approaches to this problem, the latter also offering a compromise solution where senders add as many one bit message authentication codes to their messages as there are receivers, each keyed with a secret known to only one receiver.

The need for proof of delivery is recognised in the above taxonomies, but solutions are rarely discussed in the academic literature. Proof of delivery is a very different problem to acknowledgement of delivery. It has to be possible to prove the receiver did indeed receive data when they might deny reception. Pay-TV and pay-per-view systems invariably use the tamper-resistant processing and storage capabilities of the local receiver to record which products or programmes have been requested in order to form a bill at a later time (e.g. [ITU-R.810, Dillon97] as already cited). Where an isochronous stream is being transmitted, Nark [Briscoec99] even allows late delivery of an ADU to be proved. Thus a refund can be claimed where quality of service is degraded for a particular leaf of a multicast tree. This is achieved by the receiver holding back its request to its smart card for the key to each ADU until it is satisfied that the ADU has been received on time. Thus, any record of a successful key request generated by the smartcard is sufficient proof that the ADU arrived on time.

The scenario where the data is an ordered stream and access is allowed between a start and an end point is not the only one to cater for. Access might be given only to certain categories of data, or to a certain total amount of data from anywhere within the stream. A more random access approach might be required for non-sequential application name spaces [Fuchs98]. Few multicast security approaches specifically cater for such scenarios. This paper is no exception, because a secure session is assumed to be a bounded portion of a linear sequence space. However, variations to cater for non-ordered and multi-dimensional key sequences are discussed later.

We have described four multicast security requirements beyond basic privacy key management. 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 the basic key management schemes to achieve a selection of the above 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. The subsequent section will describe four different key sequence constructions that meet these requirements, but for the purposes of describing the architecture, they will all be discussed collectively, and simply termed 'MARKS constructions'.

We deliberately choose an example where the financial value of an ADU (defined in Section 2.1) 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 2.1) 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, which is not shown in the figure, but is 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 (unless a staged set-up is used).

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 would be for all S as well as all 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. The key managers listen to this announcement as well as the receivers, in order to get the current price of a game-minute. The announcement must also specify which key sequence construction is in use.

Receiver session set-up, duration and termination

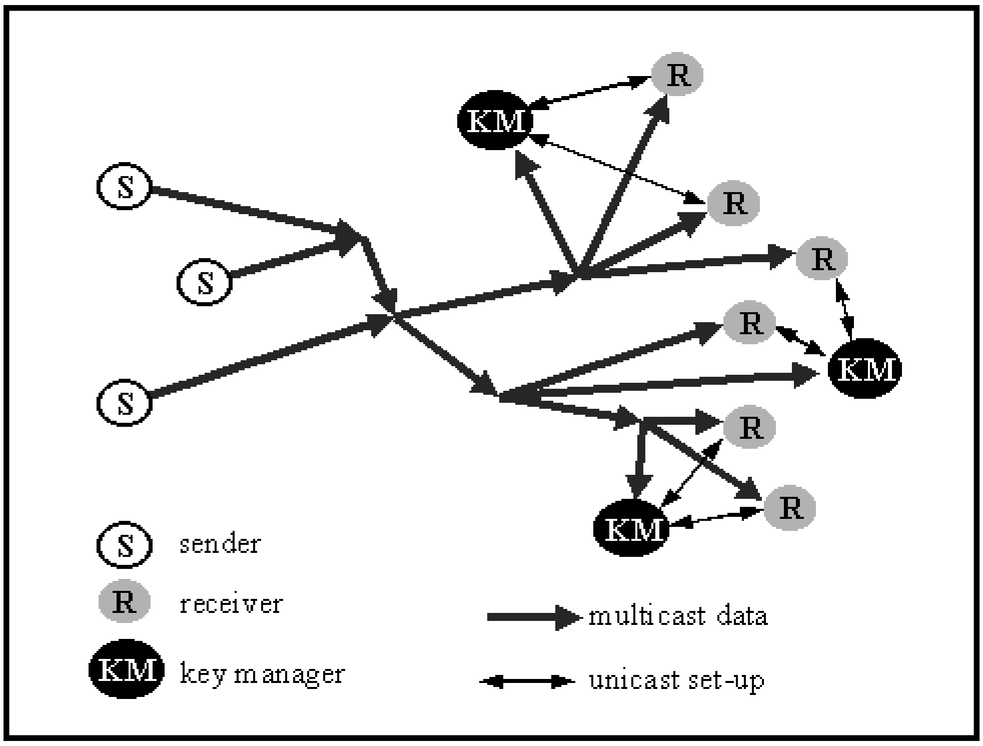


Fig 3.1 - Key management system design

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 giving her credit card details, or 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 constructions described in the next section make 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.

4. Key Sequence Construction

In all the key sequence constructions below, the following notations are used:

- `b(v)` is the notation used for a function that blinds the value of `v`. That is, a computationally limited adversary cannot find `v` from `b(v)`. An example of a blinding or one-way function is a hash function such as the MD5 hash [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 of the hash, not the size reduction property.
- $b^h(v)$ means the function `b()` applied repeatedly to the previous result, `h` times in all.
- `r(v)` 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.
- $c(v_1, v_2, ...)$ is a function that combines the values of v_1 , v_2 etc. such that given the result and all but one of the operands, the remaining operand can be trivially deduced. `c()` should also be chosen such that, if the bits of the operands are independent and unbiased, the bits of the result will also be independent and unbiased. The XOR function is a simple example of such a combinatorial function. `c()` should also ideally be the function that can be used to trivially deduce the remaining operand, as is the case with XOR, that is: $v_1 = c(c(v_1, v_2, ...), v_2, ...)$.

A common model for all the constructions will be presented in Section 4.5, but it is clearer to introduce each scheme on its own terms first.

4.1 Bi-Directional Hash Chain (BHC)

The bi-directional hash chain construction only proves to be secure in a limited form, but we persist in describing it as the limited version forms the basis of a later scheme. There may also be scenarios where the unlimited form is of use that the authors haven't imagined. The key sequence is constructed as follows:

1. The sender randomly generates two initial seed values, `v(0,0)` & `v(0,1)`. As a concrete example, we will take these values as 128 bits wide.
2. The sender decides on the required maximum key sequence length, `H`
3. The sender repeatedly applies the same blinding function to each seed to produce two seed chains of equal length, `H`. The values are therefore `v(0,0)` to `v(H-1,0)` and `v(0,1)` to `v(H-1,1)`. As the term `H-1` appears frequently, for brevity, we will introduce another constant `G=H-1`.
   Thus formally, $v(h,0) = b^h(v(0,0))$; $v(h,1) = b^h(v(0,1))$ (4.1.1).
4. To produce key, k_0, the sender combines the first seed from chain zero, `v(0,0)`, with last from chain one, `v(G,1)`.
   To produce key, k_1, the sender combines the second seed from chain zero, `v(1,0)`, with penultimate from chain one, `v(G-1,1)`
   etc.
   Formally, $k_h = c(v(h,0), v(G-h,1))$ (4.1.2)
   Strictly, the stream cipher in use may not require 128b keys, therefore a shorter key may be derived from the result of this combination 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.
5. The sender starts multicasting the stream, encrypting ADU_0 (application data unit 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 two initial seed values to the key managers. New initial seed pairs can be generated and communicated to key managers in parallel to streaming data encrypted with keys calculated earlier.

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 seeds `v(m,0)` and `v(G-n,1)` to that receiver.
2. That receiver produces seed chains `v(m,0)` to `v(n,0)` and `v(G-n,1)` to `v(G-m,1)` by repeatedly applying the blinding function to the seeds sent using `(4.1.1)`.
3. The receiver produces keys k_m to k_n, using `(4.1.2)` as the sender did.

   However, any seeds `v(h,0)` where `(h < m)` or `v(h,1)` where `(h > n)`, cannot feasibly be know by this receiver without an exhaustive search of the blinded seeds that 'precede' those the sender has revealed. Therefore, keys outside the range k_n to k_m cannot feasibly be calculated by this receiver.
4. Any other receiver can be given access to a completely different range of ADUs by sending the relevant seeds at the bounds of that range; the 'start' seed from the first chain and the 'end' seed from the second chain.

The creation of this key sequence is graphically represented in Fig 4.1.1:

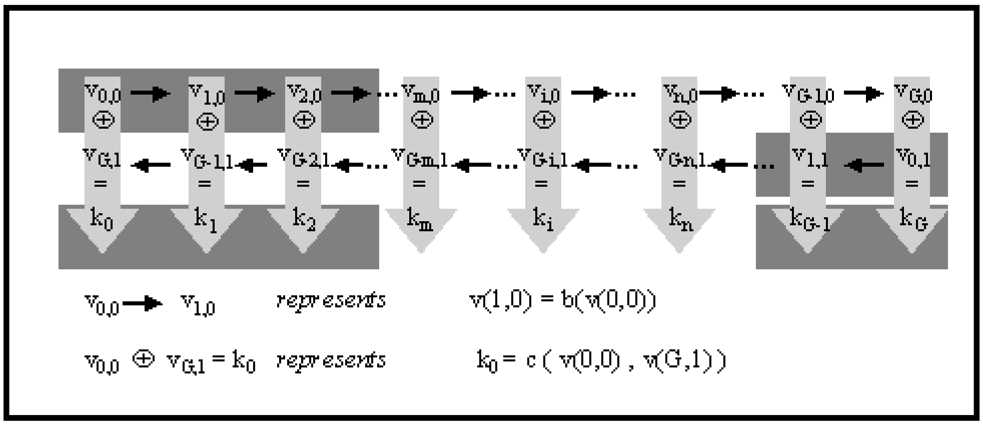


Fig 4.1.1 - Bi-directional hash chain

The ranges of seeds with a dark grey background represent those blinded from the first mentioned receiver. This leads to the keys with a dark grey background also being blinded from this receiver.

Therefore, each receiver can be given access to any contiguous range of keys by sending just two seeds per receiver per session. Unfortunately, this construction is of limited use unless each receiver can be restricted to only ever having one range of keys revealed within one sender sequence. If a receiver is granted access to an early range then another later range (say k_0 to k_1 then k_{G-1} to k_G) it can then calculate all the values between the two (k_0 to k_G). This is because seeds `v(0,0)`, `v(G-1,1)`, `v(G-1,0)` and `v(G,1)` will have had to be revealed, but `v(0,0)` and `v(G,1)` alone reveal the whole sequence.

One way round this restriction is to regularly restart the second chain with a new seed value (i.e. keeping `H` low) and to disallow two accesses for one receiver within `H` ADUs of each other. However, this requires holding per customer state at the key manager. There may be niche applications where this scheme is appropriate, such as commercial models where customers can only extend a subscription, not withdraw then re-instate it. In such cases, this would be an extremely efficient scheme.

A second way round this restriction is to note that two disjoint chains are only possible if there is room for a gap between two minimally short chains. In other words, a chain with `H<4` will always be secure. Such a short chain doesn't seem much use, but later we will use this feature to build a hybrid construction from short BHC fragments.

4.2 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 4.2.1.

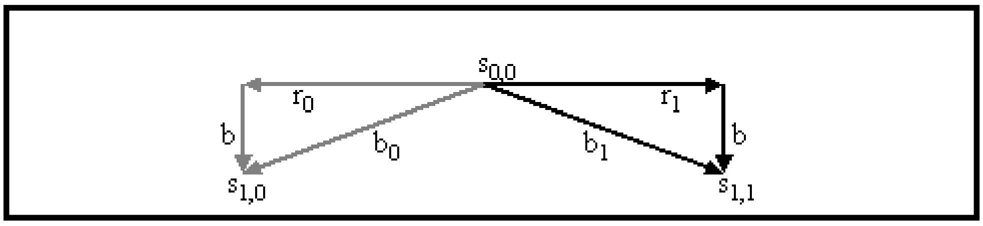


Fig 4.2.1 - Two blinding functions from one.

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.

This key sequence is constructed as follows:

1. The sender randomly generates an initial seed value, $s(0,0)$. Again, 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, $s(0,0)$:
   $s(d,i) = b_p(s(d-1, \lfloor i/2 \rfloor))$
   (4.2.1)
   where $p=0$ for even i and $p=1$ for odd i (see Appendix C for notation)
4. The key sequence is then constructed from the seed values across the leaves of the tree or truncated derivations of them as before.
   That is, if $D=5$, $k_0 = s(5,0)$; $k_1 = s(5,1)$; ... $k_{31} = s(5,31)$.
   Formally, $k_i = s(D,i)$ (4.2.2)
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. New initial seeds can be generated and communicated to key managers in parallel to streaming data encrypted with keys calculated earlier.

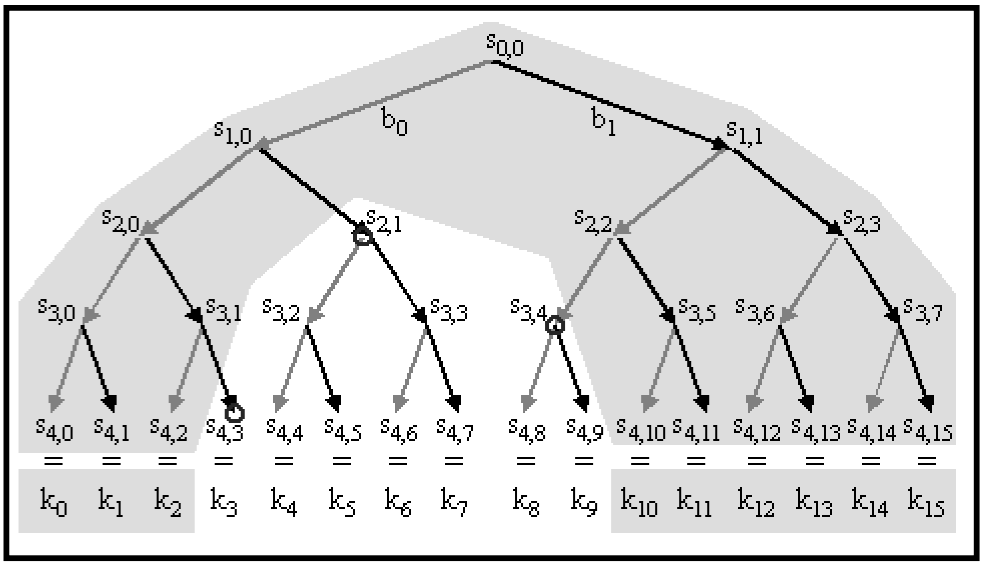
A receiver reconstructs a portion of the sequence as follows:

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, so that, before moving up a layer, the minimum and maximum are always even and odd respectively. To move up a layer, the minimum and

maximum indexes are halved with the maximum rounded down. This ensures the difference between them predictably reduces to half the previous difference 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 `4.2.1` & `4.2.2`)
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 4.2.2:



$s_{0,0}$
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

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.

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.

Another way to avoid this upper limit, is to make `D` variable instead of constant, e.g. $D = D_0 + f(i)$. Fig 4.2.3 shows such a continuous BHT where $D_0=4$ and where `D` rises by one every M keys. In this example M takes a fixed value of 7. However, there is little point in adding this complexity as the only seeds common to the different branches of the tree are those along the far right-hand branch of the tree, $s(d,2^d)$. If any of these were ever revealed the whole future tree would have been revealed. Therefore, this 'improvement' can never be used to add efficiency when revealing arbitrary ranges of keys to receivers and all it saves is the sender very occasionally passing a new initial seed in a trivial message to the key managers. On the contrary, it introduces a security weakness, as it creates a set of seeds of 'infinite' value for which any amount of exhaustive searching will be worthwhile. On the other hand, regularly having to generate a new initial seed, as in the first work-round, sets a ceiling on the vulnerability of the BHT to attack.

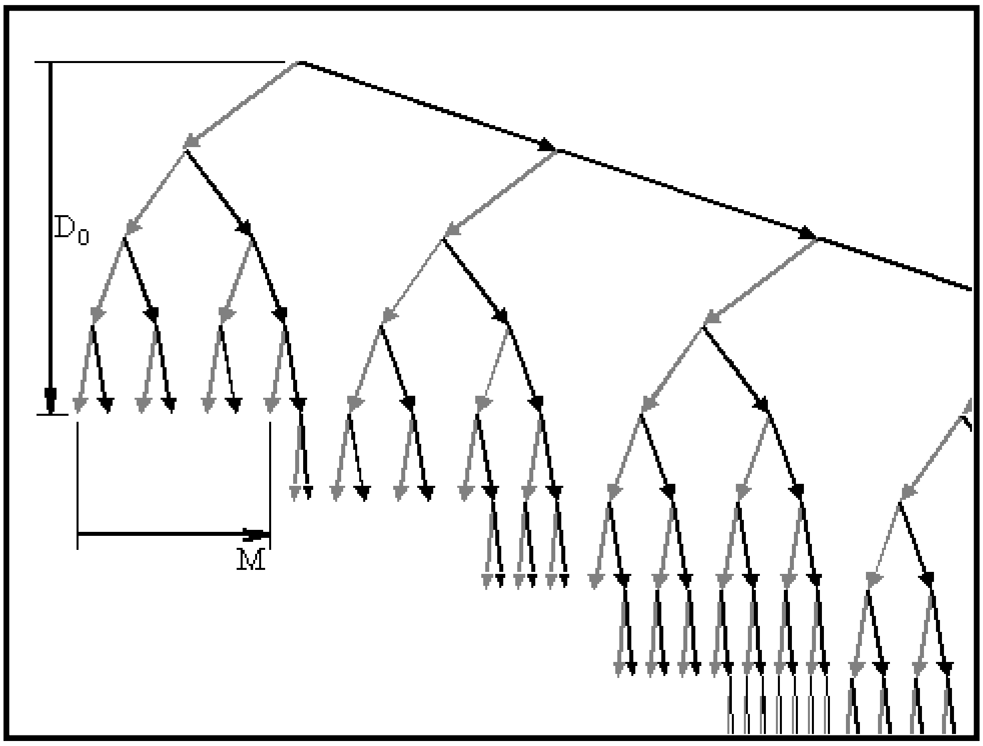


Fig 4.2.3 - Continuous binary hash tree

4.3 Binary Hash Chain-Tree Hybrid (BHC-T)

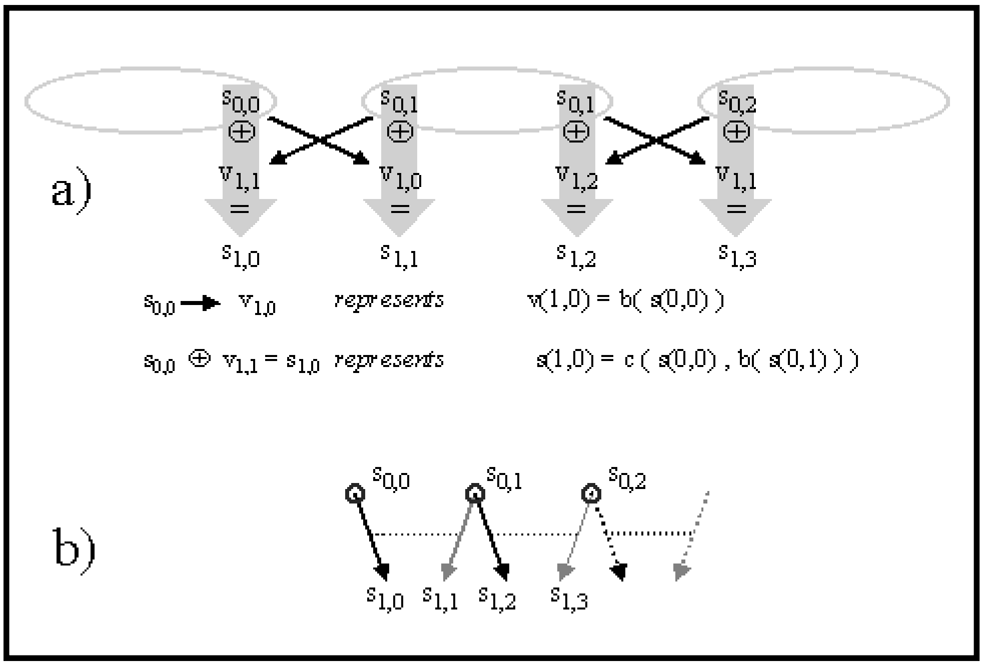
This construction is termed hybrid because a binary hash tree (BHT) is built from fragments of bi-directional hash chains (BHCs) that are just two seeds long. For understanding only we will start the explanation building the tree in the root to leaf direction in order to construct a BHC fragment, as shown in Fig 4.3.1. This is for explanation only. Later we will recommend the best way to build the tree is from the side rather than the root.

1. Let us assume we have two initial seed values generated randomly, `s(0,0)` and `s(0,1)`. Again, as a concrete example, we will take their values as 128 bits wide.
2. We now apply the same blinding function to each seed to produce two blinded seeds `v(1,0)` and `v(1,1)`.
3. To produce child seed, `s(1,0)`, we combine the first seed, `s(0,0)`, with the blinded second seed, `v(1,1)`. To produce child seed, `s(1,1)`, we combine the second seed, `s(0,1)`, with the blinded first seed, `v(1,0)`.
4. If we now randomly generate a third initial seed, `s(0,2)` and blind it to produce `v(1,2)`, we can combine the second and third initial seeds and their opposite blinded values in the same way to produce two more child seeds, `s(1,2)` and `s(1,3)`. This means that every parent seed produces four children, two when combined (incestuously) with its sibling to one side and the other two when combined with its half-sibling to the other side. In consequence, this construction produces a binary tree if new child seeds are blinded and combined as their parents were because the number of seeds doubles in each generation. However, the tree only branches under the middle of the top row of seeds (assuming more than two initial seeds are created along this row). The edges of the tree 'wither' inwards if built from the top (but see later).

    Formally, to allow us to keep the notation consistent with later more general models, we give the blinded values, `v(h,j)`, a tree height index, `h`, one more than twice the depth index, `d`, of their corresponding seed value:

    ```
    v(h,j) = b(s((h-1)/2, j)).
    ```
    Then
    ```
    s(d,i) = c(s(d-1,     i/2), v(2d-1,  i/2+1) ) for even i
           = c(v(2d-1,(i-1)/2), s(d-1, (i+1)/2) ) for odd i      (4.3.1).
    ```

Fig 4.3.1a) illustrates two pairs of parent seeds of the BHC-T hybrid, `(s(0,0), s(0,1))` and `(s(0,1), s(0,2))`. The rings identify the parent seed that is common to each pair, although the outer values in the outer rings fall off the edge of the diagram, because we focus on the descendants of just the central parent seed, `s(0,1)`. Fig 4.3.1b) shows the same three parents producing the same four children. In order to better illustrate how a binary tree is being formed, the blinded seeds are hidden from view as they are never communicated. The ringed parent seeds in the lower diagram represent the same three ringed seeds shown in the upper diagram. The two dotted arrows that continue the sequence to the right show how parent seed `s(0,2)` would produce another two children if there were another parent to the right. The dotted lines joining each pair of arrows represent the fact that both parents above this line combine to produce both children below it. We will represent this construction in later diagrams using the simplified form.



a)
$s_{0,0}$ ⊕ $v_{1,1}$ = $s_{1,0}$
$s_{0,1}$ ⊕ $v_{1,0}$ = $s_{1,1}$
$s_{0,1}$ ⊕ $v_{1,2}$ = $s_{1,2}$
$s_{0,2}$ ⊕ $v_{1,1}$ = $s_{1,3}$
$s_{0,0} \rightarrow v_{1,0}$ represents $v(1,0) = b(s(0,0))$
$s_{0,0} \oplus v_{1,1} = s_{1,0}$ represents $s(1,0) = c(s(0,0), b(s(0,1)))$
b)
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$
$s_{1,0}$ $s_{1,1}$ $s_{1,2}$ $s_{1,3}$

Fig 4.3.1 - Binary hash chain-tree hybrid elements

Fig 4.3.2a shows how three initial seeds would construct a binary tree in the centre with the edges withering inwards. The indexes are not meant to be readable. Fig 4.3.2b shows a magnified part of the same tree. To be consistent with the common model given later, the index, `i`, of every 'right' child is set to zero modulo the degree of the tree. That is, for a right child in a binary tree, `(i)mod2=0`. This convention applies to all the constructions where children are derived by combining parents. For the BHT the opposite applies; `(i)mod2=0` for a 'left' child.

As with the binary hash tree, the keys used to encrypt successive ADUs are the sequence of seeds at the leaves of the tree or truncated derivations of them. The figure shows how to reveal an example range of keys, k_3 to k_9 by revealing the ringed seeds to a particular receiver.

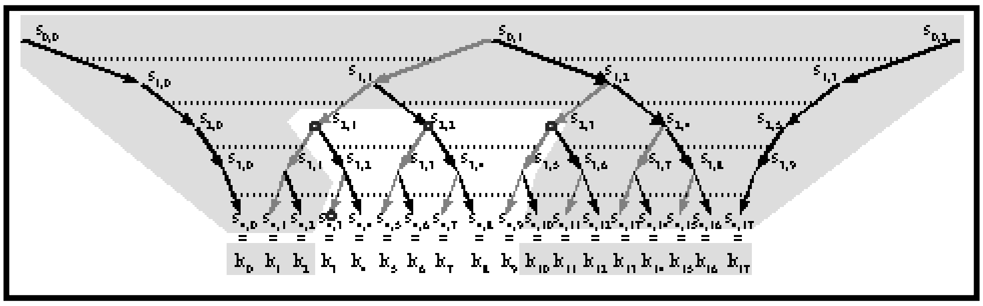
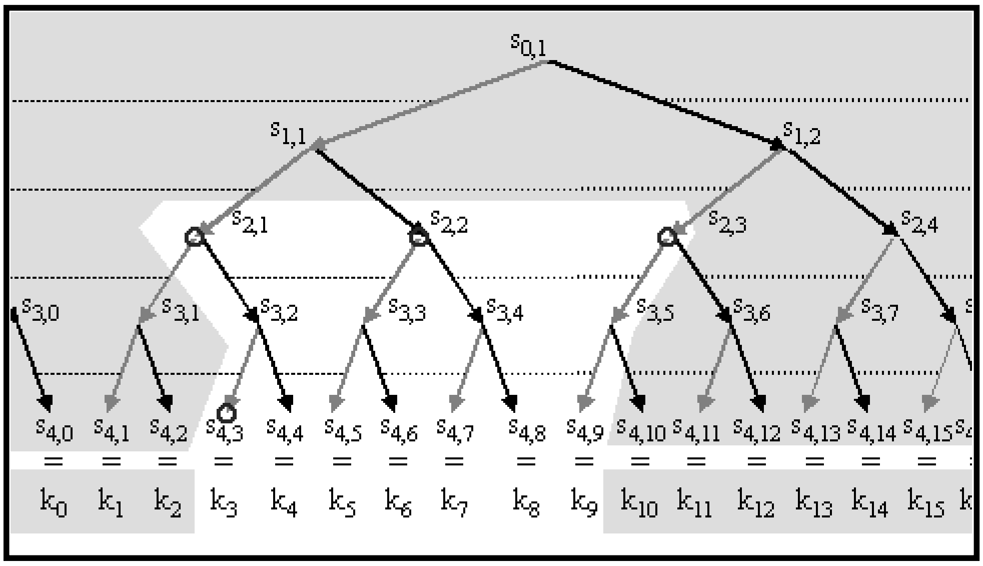


Fig 4.3.2a - Binary hash chain-tree hybrid (zoomed out)



$s_{0,1}$
$s_{1,1}$
$s_{1,2}$
$s_{2,1}$
$s_{2,2}$
$s_{2,3}$
$s_{2,4}$
$s_{3,0}$
$s_{3,1}$
$s_{3,2}$
$s_{3,3}$
$s_{3,4}$
$s_{3,5}$
$s_{3,6}$
$s_{3,7}$
$s_{4,0}$ $s_{4,1}$ $s_{4,2}$ $s_{4,3}$ $s_{4,4}$ $s_{4,5}$ $s_{4,6}$ $s_{4,7}$ $s_{4,8}$ $s_{4,9}$ $s_{4,10}$ $s_{4,11}$ $s_{4,12}$ $s_{4,13}$ $s_{4,14}$ $s_{4,15}$
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 4.3.2b - Binary hash chain-tree hybrid

We now move to a further twist in this construction in order to explain how to build the tree from the side rather than the root. It was noted earlier that the XOR function was chosen because if the XOR of two operands produces a third value, any two of these three values may be XORed to produce the third. This is illustrated in Fig 4.3.3, where the values of all the seeds are the same as in Fig 4.3.1. If `s(0,1)` is initially unknown, but `s(0,0)` and `s(1,1)` are known, `s(0,1)` then `s(1,0)` may be derived because of this 'twist' property:

```
s(0,1) = c(s(1,1), b(s(0,0)) ) then
s(1,0) = c(s(0,0), b(s(0,1)) )
```

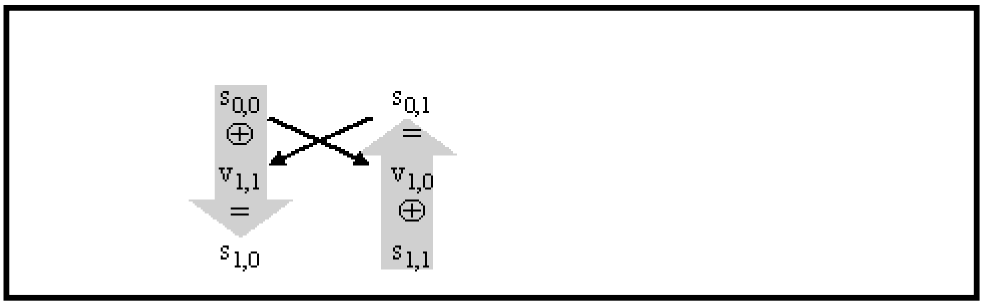


Fig 4.3.3 - Hash chain-tree twist

Fig 4.3.4 shows how a sender can build the BHC-T hybrid construction from the 'side'. The order of seed creation is shown by numbered circles. Seeds that can be created in any order are all allocated the same number followed by a distinguishing letter. The darker circles next to ringed nodes represent seeds that have to be randomly generated. We shall call these primary seeds. These fix the values of all subsequent intermediate seeds until the next ringed node.

1. The sender randomly generates the 128 bit value of seed 1.
2. Seeds 2 & 3 are then generated. They form the diagonal corners of a box of four seeds, thus setting the opposite corner values, 4 then 5 by the 'twist' algorithms:

   Formally, either of these formulae may be used depending on which neighbouring seed values are known:
   ```
   s(d,i) = c(s(d+1,  2i), v(2d+1, i+1) )
          = c(v(2d+1,i-1), s(d+1, 2i-1) )                  (4.3.2).
   ```
   where
   `v(h,j) = b(s((h-1)/2, j))` as before.

   To be consistent with the common model given later, seed 1 should have `i=1` as for the BHC-T built from the top.

   Note that if `d=0` for the root seed, `d` becomes increasingly negative in the leaf to root direction.
3. Seed 6 must then be generated, forming another pair of diagonal corners with 3.
4. This reveals the opposite corners, seeds 7 then 8 by equation `(4.3.2)`.

5. Seeds 8 and 3 then form the top corners of another box of four, setting seeds 9a & 9b by equations `(4.3.1)`.
6. The pattern continues in a similar fashion after seed 10 has been randomly generated. An advantage of this construction is that the tree can grow indefinitely - it is not necessary to decide any limits in advance.
7. The sender starts multicasting the stream, encrypting ADU_1 with k_1, ADU_2 with k_2 etc. but leaving at least the ADU sequence number in the clear.
   That is, $k_i = s(D,i)$ where $D=0$ (4.3.3)
8. If the sender delegates key management, it must privately communicate the primary seeds to the key managers. New primary seeds can be generated and communicated to key managers in parallel to streaming data encrypted with keys calculated earlier.

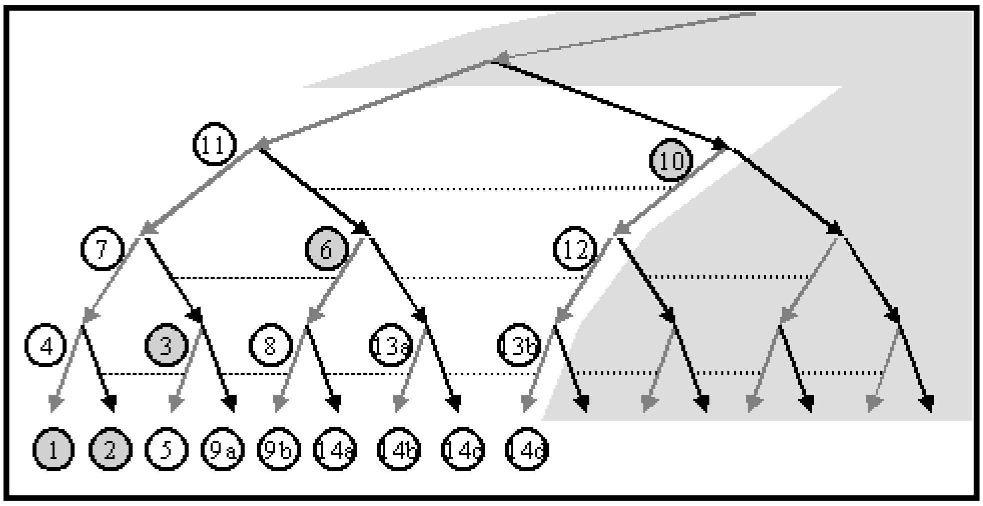


Fig 4.3.4 - Hash chain-tree growth

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. The set consists of the smallest set of intermediate seeds in the tree that enable calculation of the required range of keys.
   These are identified by testing the indices, i, of the minimum and maximum seed in a similar but mirrored way to the BHT. A 'left' minimum or a 'right' maximum always needs revealing before moving up a level. If a seed is revealed, the index is shifted inwards by one seed, so that, before moving up a layer, the minimum and maximum are always even and odd respectively. To move up a layer, the minimum and maximum indexes are halved with the maximum rounded up. The odd/even tests are repeated on the new indexes. The process continues until the minimum and maximum are two or three apart. The values cannot jump to less than two apart if they were more than three apart on the previous loop. The difference between the min and max always predictably reduces to one more than half the previous difference because the max is rounded up. If they are two apart they are revealed along with the seed between them. If they are three apart, they are only revealed along with both seeds between them if the minimum is odd. If it is even, it will be worth moving up one more layer so nothing is revealed and one more round is allowed. Before the tests start, exceptional initial conditions are tested for; where the requested range is already less than two wide.
   This procedure is described more formally, in C-like code in Appendix B
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. For instance, the algorithm in Appendix B will always reveal the same seeds in the same order for the same range of keys.
3. Each receiver can then repeat the same pairs of blinding and combining functions on these intermediate seeds as the sender did to re-create the sequence of keys, k_m to k_n. (Equations `4.3.1`, `4.3.2` & `4.3.3`)
4. Any other receiver can be given access to a completely different range of ADUs by being sent a different set of intermediate seeds.

Because the BHC-T can be built from the side, it is ideal for sessions of unknown duration. The continual random generation of new intermediate root seeds limits its vulnerability to attack but allows continuous calculation of the sequence. To further limit vulnerability, the sender could delay the generation of future seeds in order to deny any receiver the ability to calculate keys beyond a certain future point in the sequence. This would limit the time available for a brute force search of the seed-space. Nonetheless, building the tree from the side causes the numbers of keys dependent on each new root seed (and consequently the value of an attack on that seed) to grow exponentially.

The value of a root seed can be bounded by regularly incrementing the level defined to be the leaf level, moving it one layer closer to the root after each sequence of M keys (except the first).

Formally this requires equation (4.3.3) to be replaced with:

$$k_i = s(-\lfloor i/M \rfloor, i) \qquad \text{for } i<M$$
$$k_i = s(1-\lfloor i/M \rfloor, i) \qquad \text{for } i>=M \qquad (4.3.4)$$

This is illustrated in Fig 4.3.5 with M=8. Of course, in practice M would be a lot larger in order to ensure all reasonable length receiver sessions could be described efficiently without hitting the top left-hand branch of the tree.

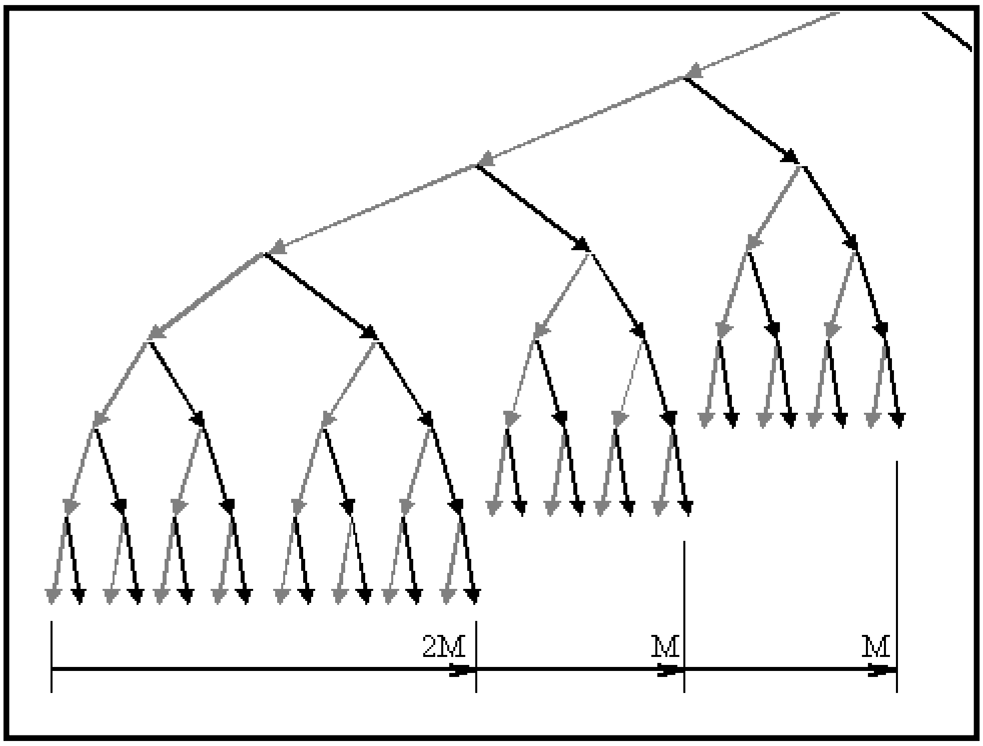


Fig 4.3.5 - Continuous binary hash chain-tree hybrid

We noted earlier that a BHC was only intrinsically secure when H<4. The BHC fragments used in this chain-tree hybrid have H=2, which ensures the security of the hybrid scheme. This also suggests that a binary chain-tree hybrid could be constructed from chain fragments of length three (H=3) without compromising security. In this case, each parent seed would produce six children when paired with its sibling and half-sibling, giving a threefold growth in tree width at each level (a ternary tree - BHC3-T). This construction is shown in Fig 4.5.5 but a full analysis is left for future work. It has the potential to be more efficient than BHC-T, if a little more complex.

4.4 Binary Hash Tree II (BHT2)

We now present a further binary tree based construction that combines the BHT and the BHC-T approaches in a way that greatly tightens security against brute force attack. We use the same notation for the seeds, $s_{d,i}$, with the origin for d being at the root as for BHT, its value rising as it approaches the leaves. One element of the tree is shown in Fig 4.4.1. We use two blinding functions in this construction, $b_0()$ and $b_1()$, which we will term 'left' and 'right' respectively, as was the case with the BHT.

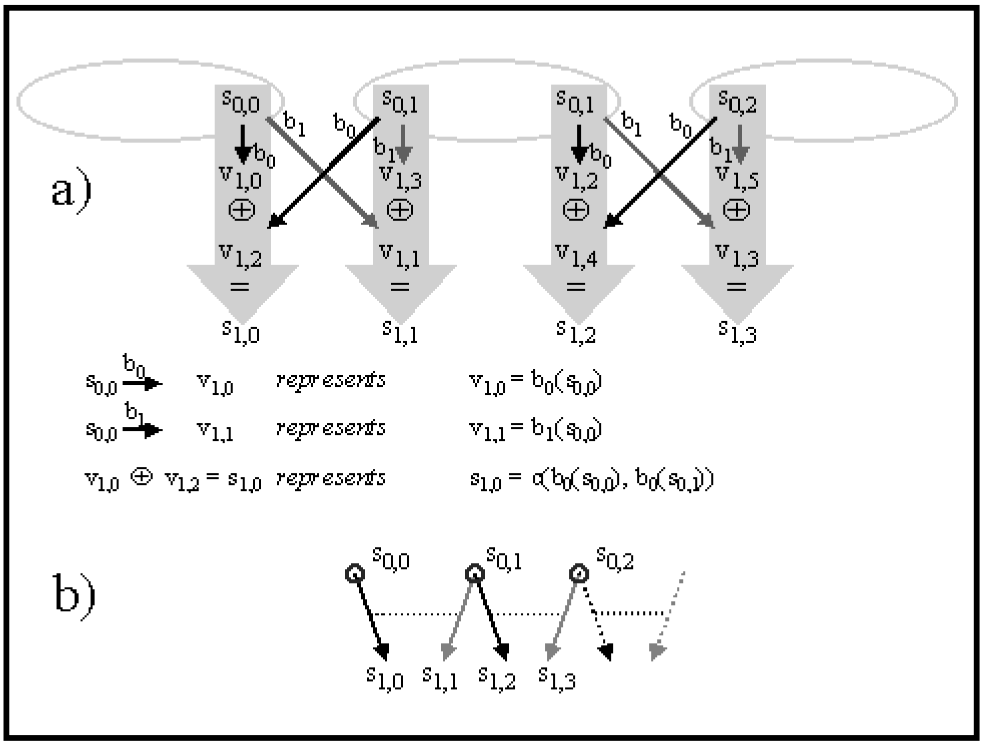
1. Let us assume we have two randomly generated initial seed values, s(0,0) and s(0,1). Again, as a concrete example, we will take their values as 128 bits wide.
2. The sender decides on the required maximum tree depth, D.
   We produce two blinded values from each of these initial seeds, one with each of the blinding functions.
   $v(1,0) = b_0(s(0,0)); \quad v(1,1) = b_1(s(0,0));$

$v(1,2) = b_0(s(0,1))$; $v(1,3) = b_1(s(0,1))$.

3. To produce child seed, $s(1,0)$, we combine the two left blinded seeds, $v(1,0)$ and $v(1,2)$.
   To produce child seed, $s(1,1)$, we combine the two right blinded seeds, $v(1,1)$ and, $v(1,3)$.
4. If we now randomly generate a third initial seed, $s(0,2)$, we can combine the second and third initial seeds in the same way to produce two more child seeds, $s(1,2)$ and $s(1,3)$. As with the BHC-T hybrid, this means that every parent seed produces two children enabling us to build a binary tree, but with the edges 'withering' inwards. In fact, if layer d contains n_d seeds, $n_{(d+1)}=2n_d-2$. As long as more than two initial seeds are used, the tree will tend towards a binary tree.

   Formally:
   $$s(d,i) = c(v(2d-1, i), v(2d-1, i+2)) \qquad (4.4.1)$$
   where
   $$\begin{aligned} v(h,j) &= b_0(s((h-1)/2, j/2)) && \text{for even } j \\ &= b_1(s((h-1)/2,(j-1)/2)) && \text{for odd } j. \end{aligned}$$
5. The key sequence is then constructed from the seed values across the leaves of the tree.
   Formally, $k_i = s(D,i)$ (4.4.2)
6. 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.

Fig 4.4.1a) illustrates two parent seed-pairs of the BHT2, $(s(0,0), s(0,1))$ and $(s(0,1), s(0,2))$. The rings identify the parent seed that is common to each pair in both parts a) and b) of the figure, in exactly the same fashion as was used to illustrate the BHC-T hybrid. As before, Fig 4.4.1b) shows how a tree of seeds built with BHT2 can be represented, hiding the intermediate blinded values from view for clarity. Once these internal values are hidden, the resulting BHT2 looks identical to the BHC-T hybrid in Fig 4.3.2.



a)
$s_{0,0}$ $s_{0,1}$ $s_{0,1}$ $s_{0,2}$
b_1 b_0 b_1 b_0
$v_{1,0}$ $v_{1,3}$ $v_{1,2}$ $v_{1,5}$
$v_{1,2}$ $v_{1,1}$ $v_{1,4}$ $v_{1,3}$
$s_{1,0}$ $s_{1,1}$ $s_{1,2}$ $s_{1,3}$
$s_{0,0} \xrightarrow{b_0} v_{1,0}$ represents $v_{1,0} = b_0(s_{0,0})$
$s_{0,0} \xrightarrow{b_1} v_{1,1}$ represents $v_{1,1} = b_1(s_{0,0})$
$v_{1,0} \oplus v_{1,2} = s_{1,0}$ represents $s_{1,0} = c(b_0(s_{0,0}), b_0(s_{0,1}))$
b)
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$
$s_{1,0}$ $s_{1,1}$ $s_{1,2}$ $s_{1,3}$

Fig 4.4.1 - Binary hash tree II elements

The algorithm to calculate which seeds to reveal in order to reveal a range of keys is also identical to that for the BHC-T hybrid in Appendix B, thus the ringed seeds in Fig 4.3.2 would still reveal k_3 to k_9 to a particular receiver.

The maximum number of keys across the leaves of a BHT2 built from three initial seeds (at layer 0) to depth D is 2^D+2. If a continuous tree is required, the keys can be defined to step down the layers of intermediate seeds rather than stay level across them, similar to the continuous BHT shown in Fig 4.2.3, but with similar security problems too.

We have shown how to build a binary tree only using two of the combinations of the four blinded values in (4.4.1). Taking the four values two at a time, gives six possible combinations:

```
c1 = c(v(1,0),  v(1,1) )
c2 = c(v(1,2),  v(1,3) )
c3 = c(v(1,0),  v(1,2) )
c4 = c(v(1,1),  v(1,3) )
c5 = c(v(1,0),  v(1,3) )
c6 = c(v(1,1),  v(1,2) )
```

`c1` and `c2` are dependent on only one parent seed each. Therefore, revealing the parent alone reveals a child, ruling out the use of either. Further, `c6 = c(c3, c4, c5)` and `c5 = c(c3, c4, c6)` etc. Therefore revealing any three of these combinations implicitly reveals the fourth. Nonetheless, any three of these combinations can be used rather than just the two used in the BHT2. Analysis of the resulting ternary tree (BHT3) is left for future work.

4.5 Common Model

Having presented four key sequence constructions, we now present a common model, which allows all of these schemes and others like them to be described in the same terms.

We define two co-ordinate planes

- a 'blinding' plane with discrete values, `v`, sitting at co-ordinates `(h,j)` such that, in general, values at one `h` co-ordinate are blinded to produce the values at `h+1`, the specific mappings depending on the scheme;
- a 'combining' plane with discrete values, `s`, sitting at co-ordinates `(d,i)`, which are the result of combining values from the blinding plane in ways that again depend on the scheme

Each construction is built from elementary mathematical 'molecules' in the blinding plane. Figs 4.5.1-4.5.5 show these molecules as a collection of thick black arrows representing the blinding functions mapping from one value of `v` to the next, starting from the `h=0` axis. To show how the construction grows in the direction of the `j` axis, the thick but very light-grey arrows represent blinding of adjacent values that complete the next molecule. A molecule is defined by three constants:

- `H`, the height of one molecule along the `h` axis of the blinding plane
- `P`, the number of blinding functions used within one molecule
- `Q`, the number of values that are combined from each molecule in the blinding plane to produce each value in the combining plane

The initial values, `v`, of one molecule in the blinding plane map directly from the previous values, `s`, in the combining plane (shown as chain dashed lines in Figs 4.5.1-4.5.5):

if $h \bmod H = 0$; $v(h,j) = s(h/H, j)$ (4.5.1).

Subsequent values in a blinding plane molecule are blinded from previous values (shown as thick arrows):

if $h \bmod H \neq 0$; $v(h,j) = b_p(v((h-1), \lfloor j/P \rfloor))$ (4.5.2).
where $p = j \bmod P$.

The resulting final values in the blinding plane molecule are then combined to produce the next values in the combining plane (shown as thin lines):

$s(d,i) = c(v(h_0,j_0), \ldots v(h_q,j_q), \ldots v(h_{(Q-1)},j_{(Q-1)}))$ (4.5.3).

Where h_q are j_q are defined for each construction as functions of the parameter q.

Thus, `d` increments one in the combining plane for every `H` along the `h` axis in the blinding plane.

Table 4.5.1 gives the values of `H`, `P` and `Q` and the formulae for h_q are j_q that define each construction introduced in the earlier sections. It also refers to the figures that illustrate each construction using this common model. Note that constructions with `Q>2` have not been analysed. The last column of the table shows that the one-way function tree (OFT) [McGrew98] also falls within the scope of this general class of constructions, with `n` being the number of receivers. It is also briefly discussed below. The final row gives the minimum number of initial seed values that each construction needs to get started. Other initial values might be used to produce different constructions.

	BHT2	BHT	BHC	BHC-T	BHC3-T	OFT
Fig	4.5.1	4.5.2	4.5.3	4.5.4	4.5.5	4.5.6
H	2	2	H	2	3	2
P	2	2	1	1	1	1
Q	2	1	2	2	2	2
h_q	Hd - 1		H(d-1) + q(H-1) + (1-Qq)(i mod H)			Hd - 1
j_q	i + Pq		⌊i/H⌋ + q			Qi + q
min set of initial conditions	s(0,0), s(0,1), s(0,2)	s(0,0)	s(0,0), s(0,1)	s(0,0), s(0,1), s(0,2)	s(0,0), s(0,1), s(0,2)	s(0,0), ..., s(0,n)

Table 4.5.1 - Coefficients and formulae of the common model defining each key sequence construction

In all cases, unless a continuous construction is desired, the keys constructed from the sequence can be defined by:

$$k_i = s(D,i) \qquad (4.5.4)$$

where $D=\log(N_0)$

where N_0 is the maximum number of keys required

If a continuous construction is required, this can be replaced with an equation such as (4.3.4).

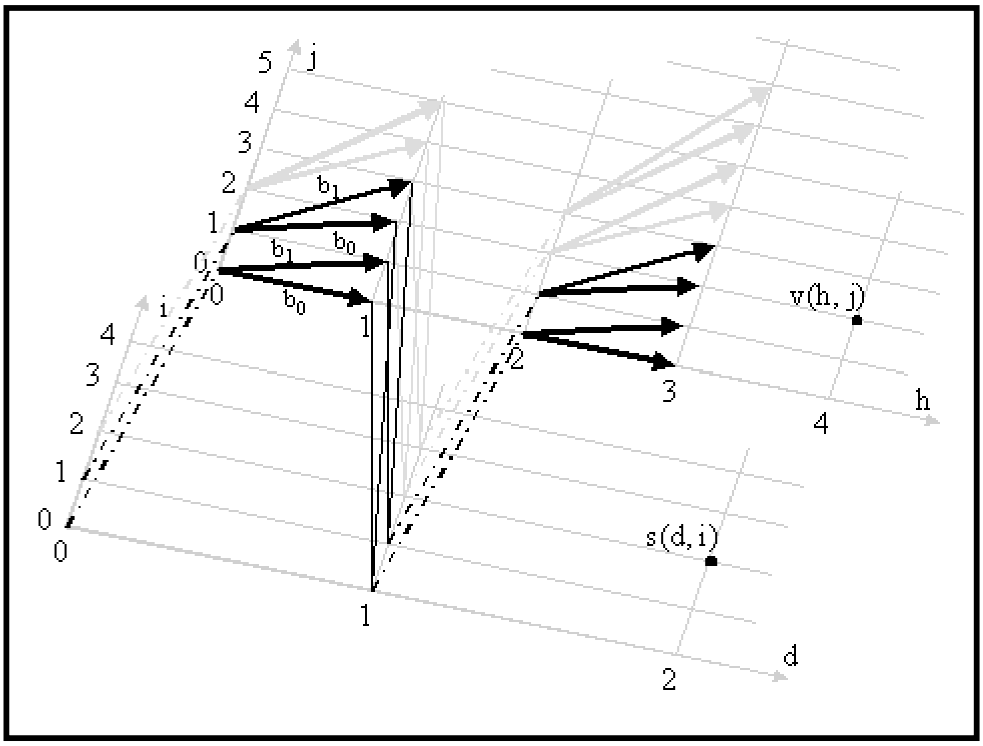


Fig 4.5.1 - BHT2 - Binary hash tree II

The formulae for the BHT2 and BHT are fairly straightforward. The h co-ordinate in the blinding plane increases at double the rate of the d co-ordinate in the combining plane (i.e. H=2) and the combination in the BHT2 increments simply and equally across the i and j axes.

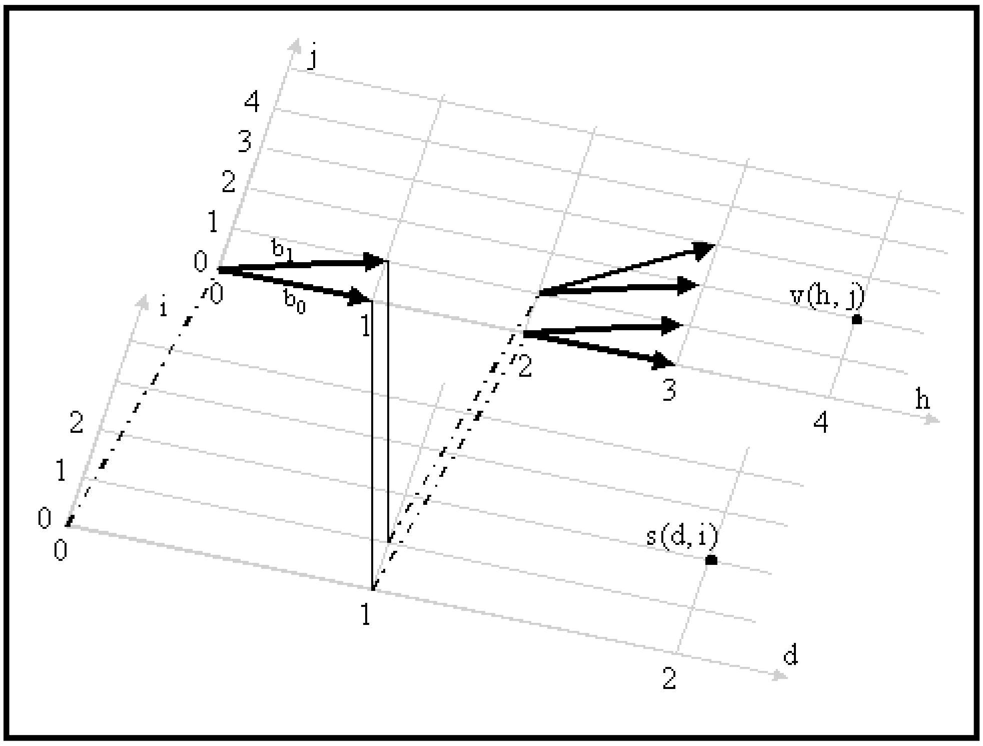


Fig 4.5.2 - BHT - Binary hash tree

Note that the BHT is the degenerate case of this model. It involves no combination, therefore the combination plane is redundant. Consequently, equations (4.5.1) and (4.5.3) are redundant and the `h` co-ordinates in the blinding plane could be halved (rounding up) to remove the gaps between the molecules. This was the approach taken in section 4.2, but the formulae in Table 4.5.1 preserve consistency with the model common to all the constructions.

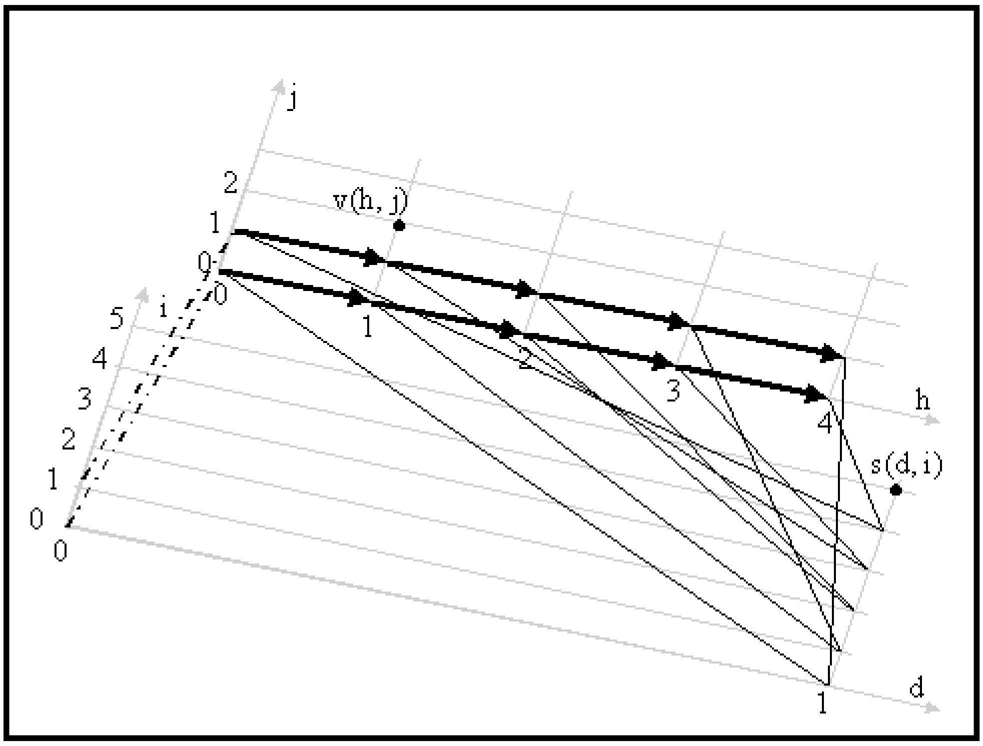


Fig 4.5.3 - BHC - Bi-directional hash chain

The BHC always has only one molecule, of height `H`. Fig 4.5.3 shows a BHC with `H=5`. The combining plane is always just one deep, with all the seeds arranged along the i axis.

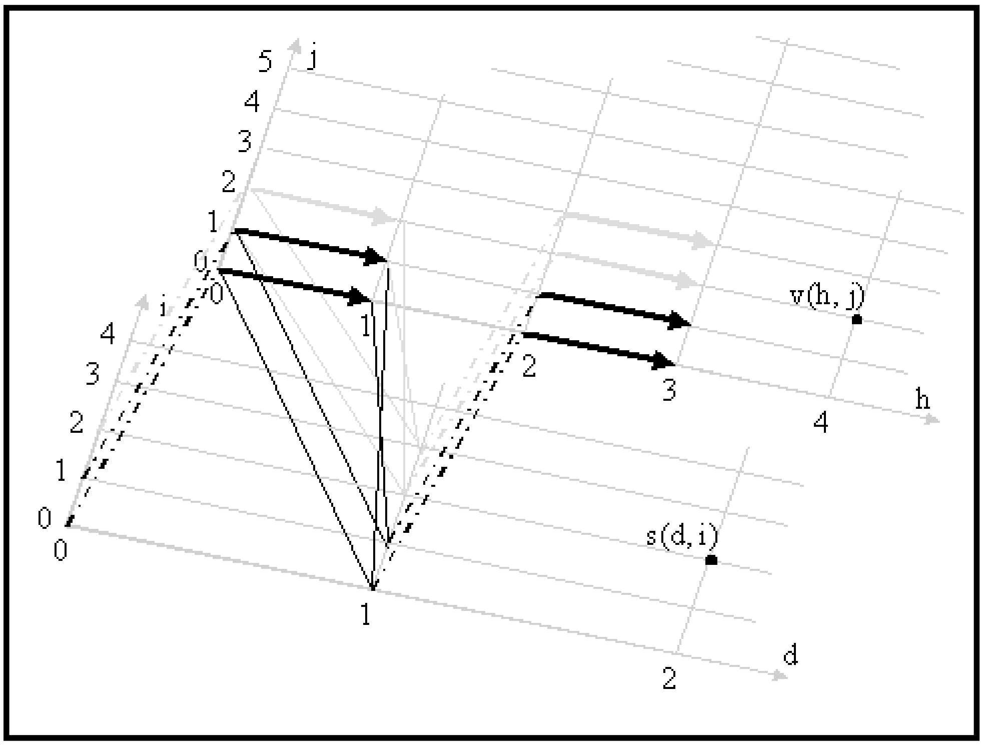


Fig 4.5.4 - BHC-T - Bi-directional hash chain-tree hybrid

The molecules of the BHC-T (Fig 4.5.4) are two high (H=2) along the h axis. Those of the BHC3-T (Fig 4.5.5) have H=3 and start to relate visually to the longer BHC. In particular, the mapping from the long molecule in the blinding plane to the wide one in the combining plane can be clearly seen. The BHC-T formula for h_q appears complex, but it is simply arranged to hold the base value of each molecule constant while a BHC-like molecule is built, before moving on to the next molecule.

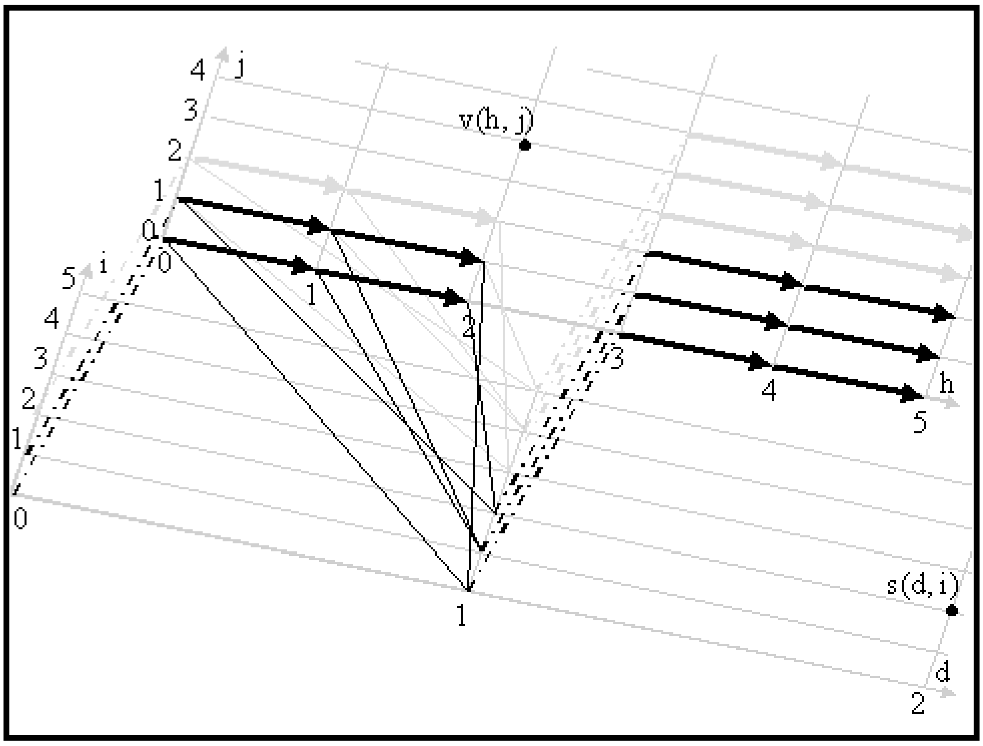


Fig 4.5.5 - BHC3-T - Bi-directional hash chain-tree hybrid III

The one-way function tree (OFT) [McGrew98] in Fig 4.5.6 happens to conform to the same mathematical model as the constructions in this paper. However, OFT is used very differently; instead of generating a large number of seeds from a few initial seeds, the purpose is to calculate a single group key. The group controller combines the individual keys it allocates to each member, recalculating the group key whenever any one member changes. When membership changes, a new leaf key is generated and everyone is multicast the blinded values from this leaf to the root, each encrypted with the keys at the complementary nodes along this branch of the tree. If the number of members of the group is not a power of two, the balance of the members are added at the appropriate layer (value of `d`) instead of at `d=0`. Thus the initial conditions change as members join and leave. In summary, in contrast to the current work, the group key is changed every membership change rather than changing it systematically and the tree starts from the leaves and moves towards the root rather than vice versa.

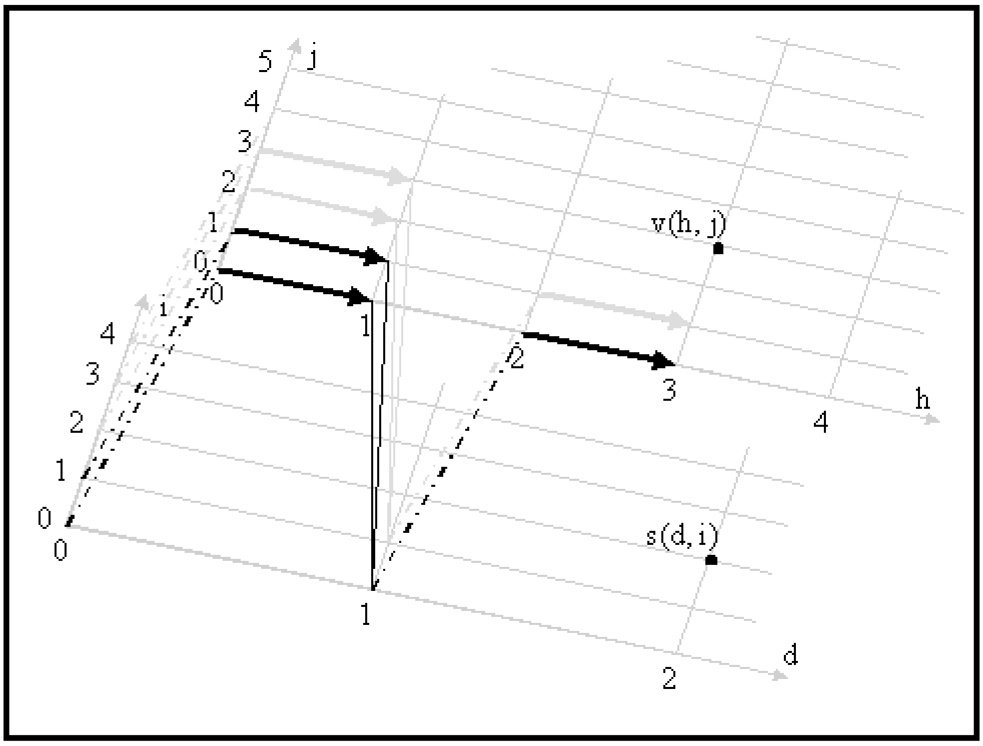


Fig 4.5.6 - OFT - One-way function tree

5. Discussion

5.1 Storage and Processing Costs

In all the MARKS constructions, a small number of seeds is used 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. As has already been noted, the bi-directional hash chain with `H>3` is extremely efficient in messaging bandwidth terms, but insecure. Therefore we will confine discussion to the tree-based constructions.

We will now show that all the tree constructions require 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. We will concentrate on the BHT first then expand the scope to cover the BHT2 or BHC-T.

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, shown again in Fig 5.1.1, 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
   Here, we reach the point in the process illustrated in Fig 5.1.1. The initial seeds are shown ringed. The seeds in memory are the dark blobs and those seeds that have been used and then deallocated are the light blobs.
9. The process continues along similar lines until `s(4,9)` is finished with, when it is deallocated leaving no further seeds in memory.

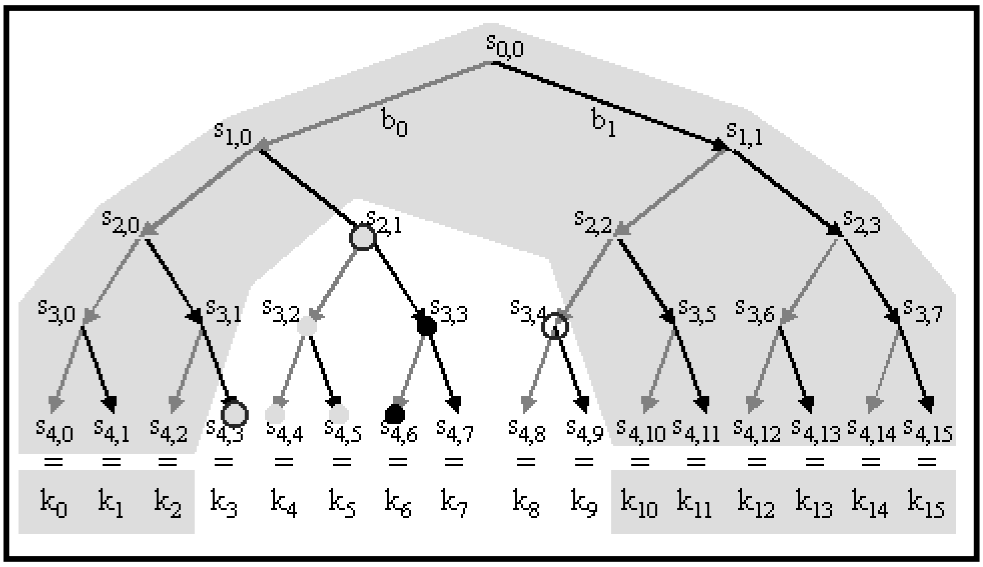

$s_{0,0}$
b_0
b_1
$s_{1,0}$
$s_{1,1}$
$s_{2,0}$
$s_{2,1}$
$s_{2,2}$
$s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$ $s_{3,4}$ $s_{3,5}$ $s_{3,6}$ $s_{3,7}$
$s_{4,0}$ $s_{4,1}$ $s_{4,2}$ $s_{4,3}$ $s_{4,4}$ $s_{4,5}$ $s_{4,6}$ $s_{4,7}$ $s_{4,8}$ $s_{4,9}$ $s_{4,10}$ $s_{4,11}$ $s_{4,12}$ $s_{4,13}$ $s_{4,14}$ $s_{4,15}$
= = = = = = = = = = = = = = = =
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 5.1.1 - BHT storage and processing

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:

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

The same principles apply for analysing the storage and processing requirements of the BHC-T and BHT2 constructions. The only difference is that the *pairs* of parents down to the key following that in use must be stored, rather than just single parents. Also, the left parent of a pair can be deallocated once its two children have been found, but the right parent is needed for another pair of children to the right. It is also worth caching the blinded value(s) of the right-hand parent for when they are required again. Fig 5.1.2 shows a snapshot of the same example BHC-T or BHT used earlier. `s(4,9)` is about to be calculated, therefore `s(3,4)` and `s(3,5)` have just been calculated and stored. When `s(3,5)` was calculated `s(2,2)` was no longer required. As soon as `s(4,8)` was in use, `s(4,7)` was deallocated. As the procedure is similar to the BHT, it will not be described in full.

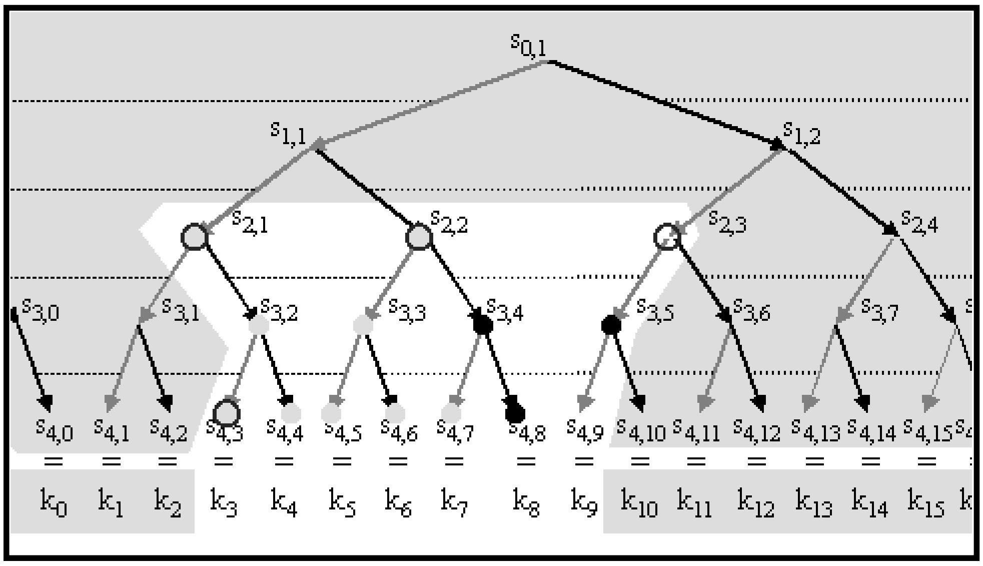

s0,1
s1,1
s1,2
s2,1
s2,2
s2,3
s2,4
s3,0
s3,1
s3,2
s3,3
s3,4
s3,5
s3,6
s3,7
s4,0 s4,1 s4,2 s4,3 s4,4 s4,5 s4,6 s4,7 s4,8 s4,9 s4,10 s4,11 s4,12 s4,13 s4,14 s4,15
k0 k1 k2 k3 k4 k5 k6 k7 k8 k9 k10 k11 k12 k13 k14 k15

Fig 5.1.2 - BHC-T or BHT2 storage and processing

Similarly, the mean hash processing required per key for the BHC-T or BHT is found using the same analysis as for the BHT.

If memory is extremely scarce (e.g. an embedded device) but some clock cycles are spare, storage can be traded off against processing. With all the tree constructions, 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 BHT, BHC-T and BHT2 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 and/or combining operation)

			BHT	**BHC-T**	**BHT2**
per R	(unicast message size)/w_s - w_h or (min storage)/w_s	min	1	3	3
		max	2(log(N+2) - 1)	2logN	2logN
		mean	O(log(N) - 1)	O(log(N))	O(log(N))
per R	(processing latency)/t_s	min	0	0	0
		max	log(N)	2(log(N) - 1)	4(log(N) - 1)
		mean	O(log(N) /2)	O(log(N) - 1)	O(2(log(N) - 1))
per R or S	(processing per key)/t_s	min	1	1	2
		max	log(N)	log(N) - 1	2(log(N) - 1)
		mean	2	2	4
per S or KM	(min storage)/w_s		1	3	3
per S	(min random bits)/w_s				

Table 5.2.1 - Various parameters of BHT, BHC-T and BHT2 per secure multicast session[i]

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 strategies described in Section 5.1. 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, all the constructions presented are highly scalable.

Comparing the schemes with each other, perhaps surprisingly, the hybrid BHC-T and BHT2 are very nearly as efficient as the BHT in messaging terms. They both only require an average of one more seed per receiver session set-up message. If N is large, this is insignificant compared to the number of keys required per receiver session. On average BHC-T requires twice as much processing and BHT2 four times as much as BHT. However, we shall see that the security improvements are well worth the cost.

5.3 Security

BHT. With the BHT, 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.

BHC-T. With the BHC-T hybrid, the strength against attack depends on which direction the attack takes. If we take a single element of the BHC-T, it has four seed values - two parents and two children as shown in Table 5.3.1 and also illustrated in Fig 5.3.1. Given only any one of the four values, none of the others can ever be calculated as there is insufficient information to test correctness. Given three of the four values, the fourth can always be calculated with just one blinding operation. Given just two of the values, the table lists how difficult it is to calculate the other two, depending on which two are given. The letter 'i' represents an input value and the values in the cells represent the number of blinding function operations necessary to guarantee finding the pair of output values given the inputs. w is the number of bits in the number-space (128 for MD5). Fig 5.3.1 shows the same information graphically, with input values ringed and blinded values shown over a grey background.

parents	s(0,0)	i	$1+2^{(w+1)}$	i	$1+2^w$	i	2
	s(0,1)	i		$1+2^w$	i	2	i
children	s(1,1)	2	i	i	$1+2^w$		i
	s(1,2)		i	$1+2^w$	i	i	2

Table 5.3.1 - Revealing and blinding seed pairs in BHC-T

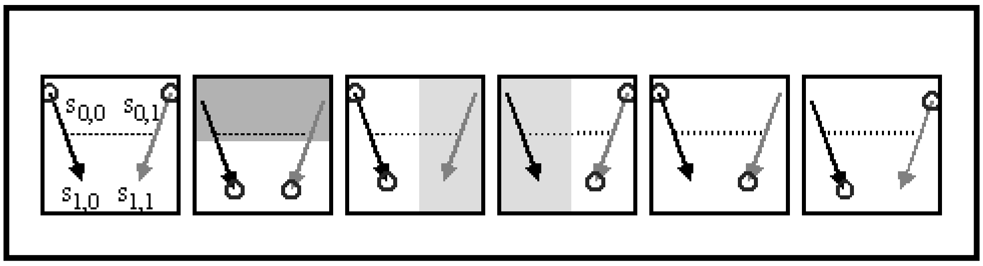


Fig 5.3.1 - Revealing and blinding seed pairs in BHC-T

If a parent and child down one side of the 'square' are given, the opposite parent can be searched exhaustively, with each value tested by blinding it and comparing it with the XOR of the two given values. Thus, success is guaranteed after 2^w blinding operations for a 'sideways' attack.

If only the two child values are given, the exhaustive search for one of the parents is slightly more involved. That is, one parent value, `s(0,1)` is guessed, and it is only correct if the following is true:

```
c(s(0,1), b(c(s(1,1), b(s(0,1))))) = s(1,2)
```

Thus, success is guaranteed after $2^{(w+1)}$ blinding operations for an 'upwards' attack.

The probability of finding two unknown values that are compatible with the two given values but are also not the *correct* pair of values (a double collision) is small in this construction. If such a pair does turn up, they can only be tested by producing keys with them and testing the keys on encrypted data. The lesser probability of a double collision therefore slightly reduces the complexity of the attacker's task.

A sideways attack can only gain at most one seed at the same level as the highest seed already known. An attack to the right ends at an even indexed child as only one value is known in the next 'box' to the right. Similarly, attacking to the left is blocked by an odd indexed child. An upward attack is then the only remaining option. One successful upward attack gives no extra keys, but when followed by a sideways attack reveals double the keys of the last sideways attack.

BHT2. The strength of the BHT2 against attack takes a similar form to that of the BHC-T hybrid, except the strength against upward attack is designed to be far greater. As with BHC-T, just one known value from a 'square' of four can never reveal any of the others. However, unlike BHC-T, three values do not necessarily immediately give the fourth. If only one parent is unknown, 2^w blinding operations are required to guarantee finding it. Given just two of the values, Table 5.3.2 lists how difficult it is to calculate the other two, depending on which two are given. As before, the values in the cells represent the number of blinding function operations necessary to guarantee finding the pair of output values given the inputs. Fig 5.3.2 shows the same information graphically, with input values ringed and blinded values shown over a grey background.

parents	s(0,0)	i	2^{2w}	i	$3+2^w$	i	$3+2^w$
	s(0,1)	i		$3+2^w$	i	$3+2^w$	i
children	s(1,1)	4	i	i	$3+2^w$		i
	s(1,2)		i	$3+2^w$	i	i	$3+2^w$

Table 5.3.2 - Revealing and blinding seed pairs in BHT2

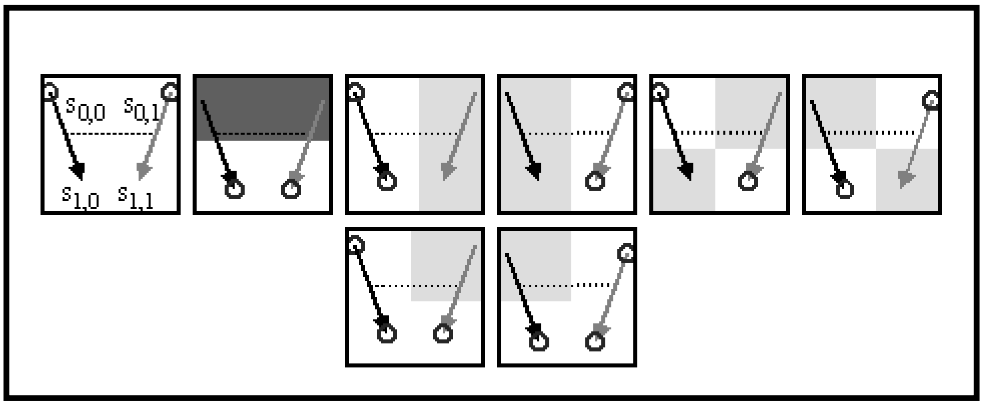


Fig 5.3.2 - Revealing and blinding seed sub-sets in BHT2

If a parent and either child down one side of the 'square' are given, the opposite parent can be searched exhaustively, with each value tested by blinding it and comparing it with the XOR of the two known values. Thus, success is guaranteed after 2^w blinding operations for a 'sideways' attack. The same applies if a parent and the opposite child are given.

If only the two child values are given, the exhaustive search for the parents is designed to be much more involved in BHT2. For each guess at the right parent value, `s(0,1)`, it must be left blinded then the left parent value has to be exhaustively searched to find a left blinded value which, when combined with the first left blinded guess gives the given value of the left child. However, when these two parent guesses are right blinded, they are unlikely to combine to give the correct right child. Thus, the next guess at the right parent has to be combined with an exhaustive search of the blinded values of the left parent and so on. This is equivalent to solving the following simultaneous equations, given only `s(1,1)` and `s(1,2)`:

$$c(b_0(s(0,0)),\ b_0(s(0,1))) = s(1,1)$$
$$c(b_1(s(0,0)),\ b_1(s(0,1))) = s(1,2)$$

To guarantee success therefore requires an exhaustive search of the square matrix of combinations of the two parents, that is 2^{2w} blinding operations. The greater strength against brute force attack in the child to parent direction is shown in the figure by a darker grey background. An alternative would be to store all the left and right blinded values of one parent to save keep recalculating them. However just the unindexed left blinded values of every possible value of one parent would consume more than 5e27TB of storage, the cost of which makes other means of attack more economically worthwhile!

The same comments about double collisions apply to BHT2 as did to BHC-T, except the wrong pair of values would only appear if four hash collisions were stumbled upon simultaneously - an event with vanishingly small probability.

Sideways attacks in BHT2 are confined to at most one 'box' either way as they are in BHC-T. Therefore, to gain any significant number of keys, an upward attack soon has to be faced. 2^w blinding operations for a sideways attack will probably be more expensive than legally acquiring the keys being attacked. Once an upward attack has to be faced, 2^{2w} blinding operations are definitely an incentive to find another way.

General Security. 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 discussed in the parts of sections 4.2, 4.3 and 4.4 on continuous trees. The randomness of the randomly generated seeds is another potential area of weakness that must be correctly designed.

All the MARKS constructions are 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 6.3 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 using any of the MARKS constructions is as strong as the weakest part.

6. Requirement Variations

The key management schemes described in the current work lend themselves 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 MARKS constructions 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.

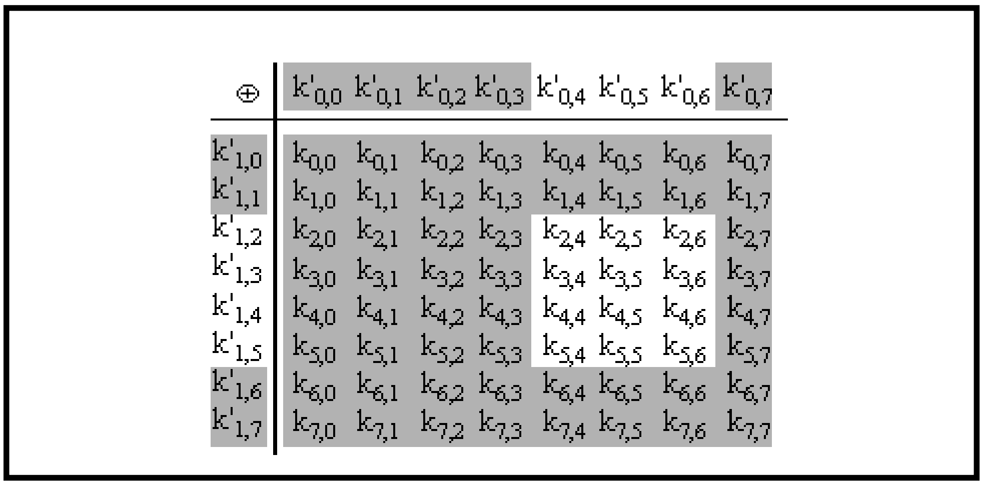
If each sender in a multi-sender multicast uses different keys or key sequences, each sender is creating a different secure multicast session even if they all use the same multicast address. This follows from the distinction between a multicast session and a secure multicast session defined in Section 2.1. In such cases each secure multicast session must be created and maintained separately from the others. However, there may be some scope for what is termed amortised initialisation [Balen99]. That is, distinct secure multicast sessions can all use the same set-up data to save messaging. For instance, the commercial model might be that customers always have to buy the same ADUs from every one of a set of related senders if they buy any at all from each. In such a scenario, each sender might combine a MARKS sequence of keys common to all senders with a long-term key specific to that sender. The customer could buy the relevant seeds for the common range of keys, then buy an additional long-term key for each sender she wished to decrypt.

6.2 Non-Sequential and Multi-Sequential Key Access

The MARKS constructions are designed to be efficient when giving each receiver access to a key sequence that is an arbitrary sub-range of a wider sequence, but not where the data isn't sequential or where arbitrary disjoint parts of a sequence are required. Thus MARKS is targeted at data streams that are naturally sequential in one dimension, such as real-time multimedia streams.

However, once a receiver has access to a range of keys, clearly there is no compulsion to access them in sequential order. For instance, the receiver may store away a sub-range of a stream of music being multicast over the Internet encrypted using one of the MARKS key sequences. Using an index of the tracks downloaded, the receiver could later pick out tracks to listen to in random order, using the relevant keys taken out of order from the MARKS sequence.

MARKS can also be used to restrict access to data that is sequential but in multiple dimensions. Some examples of such applications are described in Fuchs *et al* [Fuchs98]. A two dimensional key sequence space is shown in Fig 6.4.1.



$\oplus$	$k'_{0,0}$	$k'_{0,1}$	$k'_{0,2}$	$k'_{0,3}$	$k'_{0,4}$	$k'_{0,5}$	$k'_{0,6}$	$k'_{0,7}$
$k'_{1,0}$	$k_{0,0}$	$k_{0,1}$	$k_{0,2}$	$k_{0,3}$	$k_{0,4}$	$k_{0,5}$	$k_{0,6}$	$k_{0,7}$
$k'_{1,1}$	$k_{1,0}$	$k_{1,1}$	$k_{1,2}$	$k_{1,3}$	$k_{1,4}$	$k_{1,5}$	$k_{1,6}$	$k_{1,7}$
$k'_{1,2}$	$k_{2,0}$	$k_{2,1}$	$k_{2,2}$	$k_{2,3}$	$k_{2,4}$	$k_{2,5}$	$k_{2,6}$	$k_{2,7}$
$k'_{1,3}$	$k_{3,0}$	$k_{3,1}$	$k_{3,2}$	$k_{3,3}$	$k_{3,4}$	$k_{3,5}$	$k_{3,6}$	$k_{3,7}$
$k'_{1,4}$	$k_{4,0}$	$k_{4,1}$	$k_{4,2}$	$k_{4,3}$	$k_{4,4}$	$k_{4,5}$	$k_{4,6}$	$k_{4,7}$
$k'_{1,5}$	$k_{5,0}$	$k_{5,1}$	$k_{5,2}$	$k_{5,3}$	$k_{5,4}$	$k_{5,5}$	$k_{5,6}$	$k_{5,7}$
$k'_{1,6}$	$k_{6,0}$	$k_{6,1}$	$k_{6,2}$	$k_{6,3}$	$k_{6,4}$	$k_{6,5}$	$k_{6,6}$	$k_{6,7}$
$k'_{1,7}$	$k_{7,0}$	$k_{7,1}$	$k_{7,2}$	$k_{7,3}$	$k_{7,4}$	$k_{7,5}$	$k_{7,6}$	$k_{7,7}$

Fig 6.4.1 - Multi-dimensional key sequences

For instance, access to multicast stock quotes could be sold both by the duration of the subscription and by the range of futures markets subscribed to. Each quote would then need to be encrypted with two intermediate keys XORed together.

Thus the 'final keys' actually used for encryption would be:

$$k_{i,j} = c(k'_{0,i}, k'_{1,j}).$$

One intermediate key would be from a sequence $k'_{0,i}$ where i increments every minute. The other intermediate key could be from a sequence $k'_{1,j}$ where j represents the number of months into the future of the quote. A trader specialising in one to two year futures would not only buy the relevant sub-range of $k'_{0,i}$ depending on how long she wanted to subscribe, but she would also buy the range of intermediate keys $k'_{1,12}$ to $k'_{1,24}$.

6.3 Watermarked Audit Trail

An approach such as Chameleon [Anders97] (described earlier) can be used to watermark the keys used to decrypt the stream of data. Thus, the keys generated by any of the MARKS constructions are treated as intermediate keys. The sender creates a sequence of final keys by combining each intermediate key with a long-term key block (512kB in the concrete example) as described in Section 2.2. Each receiver is given a long-term watermarked version of the same block to produce a watermarked sequence of final keys from her sequence of intermediate keys, thus enforcing watermarked decryption.

However, this approach suffers from a general flaw with Chameleon. It creates an audit trail for any keys or data that are passed by a traitorous authorised receiver to a completely unauthorised receiver - that is a receiver without a long-term key block. In such cases the traitor who revealed the keys or data can be traced if the keys or data are traced. However, intermediate keys, rather than final ones, can be passed to any receiver who has, at some time, been given a long-term key block that is still valid. Thus a receiver not entitled to certain of the intermediate keys (which are not watermarked) 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. Hence, there is little deterrent against this type of 'internal' traitor.

Returning to the specific case of the MARKS constructions, this general flaw with Chameleon means that either the intermediate seeds or the intermediate keys can be passed around internally without fear of an audit trail. For instance, in the above network game example, a group of players can collude to each buy a different game-hour and share the intermediate seeds they each buy between themselves. To produce the real keys, each player can then use her own watermarked long-term key block that she would need to play the game. No audit trail is created to trace who has passed on unwatermarked intermediate seeds. However, there is an audit trail if any of the players tries to pass the watermarked keys or data to someone who has not played the game recently and therefore doesn't have a valid long term key block of their own. Similarly, there is an audit trail if, instead, one of the players passes on their long-term key block, as it also contains a watermark traceable to the traitorous receiver.

6.4 Unplanned Eviction

As already pointed out, the MARKS constructions allow 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, any of the MARKS schemes can be combined with another scheme, such as LKH++ [Chang99] to allow the occasional unplanned eviction. To achieve this, as with watermarking above, the key sequences generated by any of the MARKS constructions 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 MARKS 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, $k_{i,j,...} = c(k'_{0,i}, k'_{1,j}, ...)$

where intermediate keys k' can be generated from sequences using MARKS constructions or any other means such as those described in the previous two sections on watermarking and multi-dimensional key sequences.

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.

6.5 Other Group Key Management Scenarios

This paper has so far used multicast data distribution for all the example scenarios. We now present two alternative group keying scenarios to illustrate solutions for a potentially wider set of problem domains.

Virtual Private Network (VPN). A large company may allow its employees and contractors to communicate with other parts of the company from anywhere on the Internet by setting up a VPN. One way to achieve this is to give every worker a group key used by the whole company. Consequently, every time a worker joins or leaves the company, the group key has to be changed. Instead the key should be changed regularly in a sequence determined by one of the MARKS constructions, whether or not workers join or leave. As each new employment contract is set up, seeds are given to each worker that allows her to calculate the next keys in the sequence until her contract comes up for renewal. Any worker that leaves prematurely is treated as an unplanned eviction (see Section 6.4).

Digital Versatile Disk (DVD). DVD originally stood for digital *video* disk, because its capacity was suited to this medium. However, it can clearly be used to store less hungry media like software or audio. Instead of pressing a different sparsely filled DVD for each selection of audio tracks or software titles, each DVD could be produced filled to capacity with many hundreds of related tracks or titles (the ADUs). Each ADU could be encrypted with a different key from a sequence created using one of the MARKS constructions. These DVDs could be mass-produced and given away free (e.g. as cover disks on magazines). Anyone holding one of these DVDs could then buy seeds over the Internet that would give access to a range of keys to unlock some of the ADUs on the DVD. MARKS is ideally suited to such scenarios because the encryption key cannot be changed once the DVD is pressed, so commercial models that use physical media don't tend to rely on unplanned eviction. This scheme could usefully be combined with Chameleon to watermark the keys and data (as described in Section 6.3).

7. Limitations and Further Work

Each construction presented has strengths and weaknesses in terms of its efficiency and security. The trade-offs between these have already been discussed. Here we confine ourselves to inherent limitations suffered by all the schemes.

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 & 6.3 but the best solution we can offer without requiring smartcards or the complexity of 'watercasting' 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.

In the BHT and BHT2 constructions, 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 solutions 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. When one of a replicated set of stateless key managers fails it has no effect on transactions in progress on sister servers, thus isolating the overall system from problems, improving resilience. We have presented a worked example of a large-scale network game charged per minute to illustrate these points.

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). A systematic key change can then be signalled by incrementing the ADU index declared in the data. 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.

Traditional key management solutions have successfully improved the scalability of techniques to allow unplanned evictions of group members, however the best techniques 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 choice of key sequence constructions that are used by the senders to systematically change the group key. They are designed such that an arbitrary sub-range of the sequence can be reconstructed by revealing a small number of seeds (16B each). All the practical schemes can reveal `N` keys to each receiver using `O(log(N))` seeds. The schemes differ in the processing load to calculate each key, which is traded off against security. The heaviest scheme requires on average just `O(2(log(N) - 1))` fast hash operations to get started, then on average no more than just four more hashes to calculate each new key in the sequence, which can be done in advance. The lightest scheme requires half this already low processing load, implying under 10us of processing time to generate each ADU key with today's technology.

Of the constructions presented, the binary hash tree (BHT) is the simplest. The binary hash chain-tree hybrid (BHC-T) is slightly more complex and slightly more secure but equally as efficient as the BHT. However, the BHC-T is better suited for sessions where the duration is open-ended. Finally the binary hash tree II (BHT2) emulates double the key strength of the BHT with about half the efficiency. Given the BHT security is probably sufficient in most circumstances, we recommend it for its simplicity.

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.

Finally we have briefly described how our key sequence constructions could be used in other group key management scenarios such as for VPNs or for information distribution on DVD.

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>

[Balen99] D. Balenson, D. McGrew & A. Sherman (TIS Labs at Network Associates), "Key Management for Large Dynamic Groups: One-Way Function Trees and Amortized Initialization", February 26, 1999, <draft-balenson-groupkeymgmt-oft-00.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/>

[Brown99] Ian Brown, Colin Perkins & Jon Crowcroft (UCL), "Watercasting: Distributed Watermarking of Multicast Media", submitted to NGC'99, (Nov 1999), <URL:ftp://cs.ucl.ac.uk/darpa/watercast.ps.gz>

[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).

[Dillon97] Douglas M Dillon (Hughes), "Deferred Billing, Broadcast, Electronic Document Distribution System and Method", International patent publication no. WO 97/26611 (24 July 1997).

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

[Fuchs98] M. Fuchs, C. Diot, T. Turletti, M. Hoffman, "A Naming Approach for ALF Design", in proceedings of HIPPARCH workshop, London, (June 1998) <URL:ftp://ftp.sprintlabs.com/diot/naming-hipparch.ps.gz>

[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>

[Ingemar82] I. Ingemarsson, D. T. Tang, and C. K. Wong, "A Conference Key Distribution System", IEEE Transactions on Information Theory, vol. IT-28, pp. 714-720, 1982.

[ITU-R.810] ITU-R Rec. 810, "Conditional-Access Broadcasting Systems", (1992) <URL:http://www.itu.int/itudocs/itu-r/rec/bt/810.pdf>

[Kunkel97] Thomas Kunkelmann, Rolf Reinema & Ralf Steinmetz (Darmstadt Tech Uni), "Evaluation of Different Video Encryption Methods for a Secure Multimedia Conferencing Gateway", 4th COST 237 Workshop, Lisboa, Portugal, Springer Verlag LNCS 1356, ISBN 3-540-63935-7 (Dec 1997), <URL:http://www.ito.tu-darmstadt.de/publs/cost97.ps.gz>

[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.

[Naor98] Moni Naor & Benny Pinkas (Weizmann Inst of Sci, Rehovot), "Threshold Traitor Tracing", CRYPTO '98. <URL:http://www.wisdom.weizmann.ac.il/~bennyp/PAPERS/ttt.ps>

[NIST_Sha-1] FIPS Publication 180-1, Secure hash standard, NIST, U.S. Department of Commerce, Washington, D.C. (April 1995).

[Perlman] Radia Perlman, observation concerning LKH [Wallner97] from the conference floor - termed "LKH+"

[Schneier96] Schneier B, "Applied cryptography", 2nd Edition, John Wiley & Sons (1996).

[Wallner97] Wallner, Debby M., Eric J. Harder, and Ryan C. Agee, "Key management for multicast: Issues and architectures," IETF Internet Draft (work in progress) (15 Sep 1998) draft-wallner-key-arch-01.txt

[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>

Notes

[i] The exceptional cases for BHC-T when `N<=3` are not shown (in all these cases the number of seeds is just `N`). The exceptional cases when a session starts or ends are not included in the figures for per key processing.

Appendix A - Algorithm for Identifying Minimum 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;
if (min > max) error();   // reject min > max
for(d=D; ; d--) {         // working from leaves of tree...
                          // move up the tree one level each loop
    if (min == max) {     // min & max have converged...
        reveal(d,min);    // ...so reveal root of sub-tree...
        break;            // ...and quit
    }
    if odd(min) {         // odd min values are never left children...
        reveal(d,min);    // ...so reveal odd min seed
        min++;            // and step min inwards one seed to right
    }
    if !odd(max) {        // even max values are never right children...
        reveal(d,max);    // ...so reveal even max seed
        max--;            // and step max inwards one seed to left
    }
    if (min > max) break; // min & max were cousins, so quit
    min/=2;               // halve min ...
    max/=2;               // ... and halve max ready for...
}                         // ... next level up round loop
```

Appendix B - Algorithm for Identifying Minimum Set of Intermediate Seeds for BHC-T and BHT2

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;
if (min > max) error();   // reject min > max
d=D;                      // working from leaves of tree
if (max <= min+1) {       // requested min & max are adjacent/the same...
    reveal(d,min);        // ...so reveal left...
    if (max <> min)       // requested min & max are not the same...
        reveal(d,max);    // ...so reveal right too...
    quit();               // ...and quit
}
for(d=D; ; d--) {         // move up the tree one level each loop
    if (max <= min+3) {   // min & max are two or three apart...
        if (max < min+3) {    // min & max were two apart...
            reveal(d,min);    // ...so reveal left,...
            reveal(d,max);    // ...right
            reveal(d,min+1);  // ...and centre...
            break;            // ...and quit
        } else {                  // min & max were three apart, so...
            if (odd(min)) {       // ...only if min is odd...
                reveal(d,min);    // ...reveal left...
                reveal(d,min+1);  // ...left centre...
                reveal(d,max-1);  // ...right centre...
                reveal(d,max);    // ...and right...
                break;            // ...and quit
            }                     // (if min even, reveal nothing)
        }
    }
    if odd(min) {          // odd min values are never right children...
        reveal(d,min);     // ...so reveal odd min seed
        min++;             // and step min inwards one seed to right
    }
    if !odd(max) {         // even max values are never left children...
        reveal(d,max);     // ...so reveal even max seed
    } else                 // odd max values need rounding up when
halved...
        max++;             // ...so step max outwards one seed to right
    min/=2;                // halve min and ...
    max/=2;                // ... halve max ready for...
}                          // ... next level up round loop
```

Appendix C - 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`.