RZ 3215 (# 93261) 03/13/00
Computer Science/Mathematics 10 pages

Research Report

Efficient Downloading and Updating Applications on Portable Devices using Authentication Trees

Luke O'Connor and Günter Karjoth

IBM Research
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

Efficient Downloading and Updating Applications on Portable Devices using Authentication Trees

Luke O'Connor and Günter Karjoth

IBM Research, Zurich Research Laboratory, 8803 Rüschlikon, Switzerland

Abstract

Consider the problem of securely downloading n application blocks $B_1, B_2, \ldots, B_n$ from an application provider to a smart card (SC) with the possibility of updating a block B_i at a later time. In this paper we describe a scheme called Ordered Authentication Trees (OTA), which solves the problem of downloading and updating application blocks with the following properties: (1) a single signature based on an authentication tree is computed for the blocks, (2) only $O(\log n)$ additional memory is required by SC beyond the memory for the blocks themselves, (3) block B_i can be verified as correct upon receipt at the SC, and (4) blocks can be updated in $O(\log n)$ time. All previously known solutions require $O(n)$ memory and/or delay requirements for the download/update of n code blocks. The OTA scheme also generalizes to other portable devices that share common characteristics with smart cards.

Keywords: Portable device, smart card, authentication tree, digital signature.

1. INTRODUCTION

A general computing device such as a PC or a workstation stores applications in some permanent storage media, such as a hard disk, and then reads an application into main memory (RAM) for execution as required. Subject to available memory, new applications can be freely added while old applications can be easily removed or updated. However for a smart card (SC), given its limited memory capacity, applications are typically loaded into its ROM at the time of fabrication. This approach is convenient for the large scale production of SCs supporting one or a few fixed applications. But this approach is less suitable if the set of applications to be supported is expected to change, or if old applications need to be updated, or if the number of cards supporting a given application are not to be produced in large quantities.

A more flexible approach is to design the SC so that applications can be downloaded to the card as required. For example, in the ESPRIT project CASCADE [CASCADE, 1997], an SC was designed where the pre-installed software consists of a small boot kernel, libraries for basic I/O and cryptography, and a secure downloading mechanism, where other applications and systems code are downloaded securely to a FLASH memory (approximately 16 KB) to the card. As an application may be quite large with respect to the amount of RAM or bandwidth available to the SC, it is anticipated that an application will be partitioned into blocks $B_1, B_2, \ldots, B_n$ and each block will be downloaded from an Application Provider (AP) in a separate communication to the SC. Also, if an application is to be updated, then only those blocks that have been modified need be re-installed on the SC. We note that the blocks may represent either application code and/or application data. Using this scheme of block partitioning, there are two parameters of interest in evaluating a given solution with respect to downloading and updating blocks: delay requirements and memory requirements.

Delay Requirements. The downloading or updating of blocks should be 'on-the-fly' in the sense that blocks that are incorrect due to some error should be detected quickly to avoid wasting bandwidth and memory. If a block B_i arrives at time t but cannot be verified (e.g. by a hash check) until the arrival of block B_{i+d} at time $t + d$, we say that the verification of B_i is *delayed* for d blocks. For a given scheme, we are interested in the maximum delay for block verification. □

Memory Requirements. During block verification, storage is required for intermediate calculations and the caching of intermediate values that will be used for future blocks verifications. As memory on SCs is limited we seek solutions that minimize memory requirements. □

For code downloading, the worst case delay and memory requirements are both $O(n)$. An $O(n)$ delay means that all blocks must be received before verification can begin, whereas $O(n)$ memory means that an additional linear amount of space beyond the storage for the n blocks themselves is required. A protocol demonstrates a gain in efficiency if either the memory or verification delay is reduced from $O(n)$ to possibly $O(\log n)$ or even $O(1)$. We say that block verification is 'on-the-fly' if the maximum delay is $O(1)$.

1.1 RELATED WORK

Our research has been partly motivated by the fact that little work has been done to address the code download/update problem for portable devices. According to its authors, the CASCADE project report [Dhem, 1998] describes the first protocols that considered these problems. The solution in this case was to produce a hash vector $H = (H_1, H_2, \ldots, H_{n+1})$ of $(n + 1)$ components such that H_{n+1} is randomly chosen and $H_i \leftarrow h(H_{i+1}, B_i)$ for some hash function $h(\cdot)$ such as SHA-1 [SHA, 1994]. The AP signs H_1, and then sends the following $n + 2$ messages to the SC:

$$Sign(H_1),\ (H_1, B_1),\ (H_2, B_2),\ \ldots,\ (H_n, B_n),\ H_{n+1}.$$

The SC first verifies the signature on H_1 and then proceeds to verify the hash chain used to form the hash vector. Owing to the form of the chain defined above, each block B_i can be verified after the next pair (H_{i+2}, B_{i+1}) has been received, yielding a constant delay of $O(1)$. However if a block B_i is to be updated, then the hash

chain must be recomputed from position i forward due to the linear nature of the hash chain. This scheme will be denoted as 'CASCADE with hashes'.

As noted by Dhem [Dhem, 1998], the n hash values $H_1, H_2, \ldots, H_n$ need not be sent by the AP, because these values can be generated by the SC. We will refer to this scheme as 'CASCADE without hashes'. However the penalty for this reduced transmission is that the code block verification cannot begin until H_{n+1} has been received, meaning that the maximum block delay before verification is $O(n)$ when no hashes are sent. Regardless of whether the hashes are sent at the time of download, they have been discarded by the time of update and thus incur an $O(n)$ update time for a block.

A problem related to efficient application downloading/updating is that of signing digital data streams. The solution that has commonalities with our results is the Wong-Lam scheme [Wong and Lam, 1999] based on authentication trees [Merkle, 1989]. Their protocol breaks a data stream into n packets $P_1, P_2, \ldots, P_i, \ldots, P_n$ that are collected into a transmission group TG. The packets of TG are then arranged to be the leaves of an authentication tree T, and the hash of the tree is computed and signed by the sender to produce $Sign(TG)$. When TG is transmitted, each packet P_i is sent with the sequence of hash values that were used to form the path in the hash tree from the leaf representing P_i to the root of the authentication tree T. The signature $Sign(TG)$ on the authentication tree T is also sent with each packet and packet hash path. This permits each packet P_i to be verified as it is received, even though other packets in the TG of P_i may have been lost or reordered. To verify a given packet P_i, the receiver is typically required to recompute the path in the authentication tree from the leaf representing P_i to the root of T, and then verify $Sign(TG)$ based on the computed root hash. The full hash path from the leaf to the root must be computed for the first packet received, but the verification of subsequent packets can be optimized by reusing hash values that were previously computed, verified and then cached. The next received packet P_{i+1} is verified by hashing it until a node in the cache is reached that was previously authenticated. The cache structure suggested by Wong-Lam mimics the structure of the original authentication tree that the sender used to compute the signature on TG. The receiver then requires a storage of the size $O(n)$ because this is the size of the authentication tree.

1.2 NEW CONTRIBUTIONS

The solution to the application code download/update problem presented in this paper is called *Ordered Tree Authentication*, or simply OTA. OTA gives a $O(1)$ verification delay and logarithmic time for block update. This is achieved by sending a particular *ordered* sequence of hash values from the authentication tree along with the blocks to be authenticated, thus allowing verification of nodes of the tree besides the root. The OTA algorithm improves the approaches described in Section 1.1 in two aspects. First, the OTA algorithm needs a significantly lower amount of verification data to be transmitted with each data packet than Wong-Lam. Second, the OTA algorithm lowers the amount of storage required at the receiving end without increasing the time for verification at the receiver. Thus, the OTA algorithm allows safe transmission between an application or service provider and a portable device having a limited storage or memory capacity and/or restricted processing power, such as smart cards and the like. A comparison between previous schemes and the proposed OTA method is given in Table 1.

The remainder of this paper is as follows. Section 2. begins with an example of an authentication tree that will be used to display the workings of OTA. The OTA scheme is described in Section 3., where Section 3.1 details the format of the blocks at the AP for transmission, and Section 3.2 details the process that the SC uses to verify the blocks. The problem of incrementing blocks is discussed in Section 4.. Conclusions are presented in Section 5..

2. AUTHENTICATION TREES

As orginally proposed by Merkle [Merkle, 1989], an authentication tree is a data structure used to authenticate individual data items such as the blocks $B_1, B_2, \ldots, B_n$. The basic idea is to select a labeled binary tree T with $n = 2^d$ leaves and to associate B_i with the i-th leaf. For simplicity, we assume that n is a power of two but this is not required for the construction. The length of the path from the root to a node x is the *depth*

Table 1 Summary of time and storage requirements for block download and update.

	Download		*Update*	
Method	*Storage*	*Max Delay*	*Storage*	*Max Delay*
CASCADE with hashes	$O(1)$	$O(1)$	$O(n)$	$O(1)$
CASCADE without hashes	$O(n)$	$O(n)$	$O(1)$	$O(n)$
Tree Authentication	$O(\log n)$	$O(n)$	$O(\log n)$	$O(\log n)$
Wong-Lam	$O(n)$	$O(1)$	N/A	N/A
OTA	$O(\log n)$	$O(1)$	$O(\log n)$	$O(\log n)$

of x in T. The root is at depth 0 and there are 2^i nodes at depth i. An authentication tree T has *height* d, the largest depth of any node in T. Tree T has exactly n leaves associated with the values of $B_1, B_2, \ldots, B_n$ and exactly $n-1$ internal nodes with two children each.

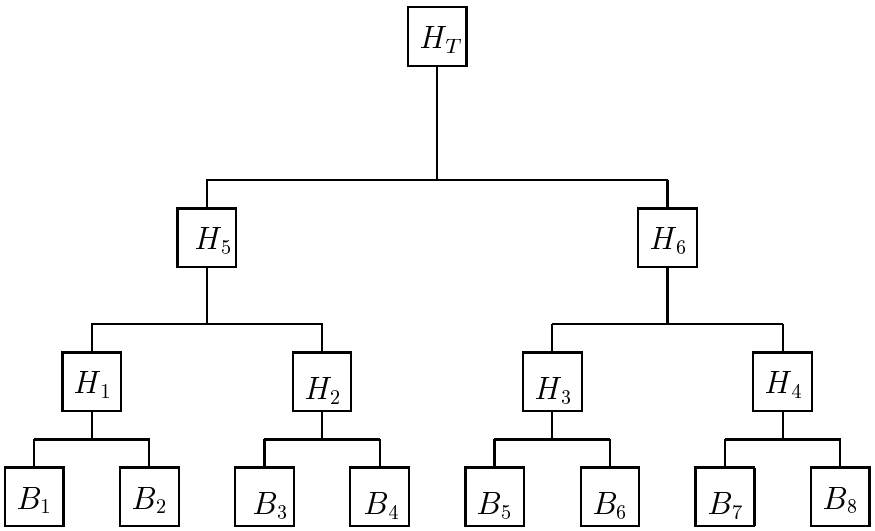


Figure 1 An authentication tree for $n = 8$ and $d = 3$.

To compute the hash of the tree, the i-th leaf is labeled $H(B_i) = h(B_i)$, where B_i is associated with the leaf. Then, beginning at depth d and proceeding to the root at depth 0, each internal node j is labeled

$$H_j = h(L(H_j)||R(H_j))$$

where $||$ denotes concatenation and $L(H_j)$ and $R(H_j)$ are the labels of the left and right child, respectively, of node j. The label at the root, denoted H_T, is a hash value that depends on $B_1, B_2, \ldots, B_n$. The structure of an authentication tree on $n = 8$ values $B_1, B_2, \ldots, B_8$ is shown in Figure 1.

The AP signs the hash H_T of T, then sends H_T, its signature, and the blocks $B_1, B_2, \ldots, B_n$. To verify the signature on the blocks, the SC must repeat all the hashing computations on T to determine H_T. Note that no block can be rejected as corrupted until all blocks have been received, because the locally computed value of H_T is not available until that time. Further, if the generated root hash does not match the received root hash, then the incorrect block(s) cannot be identified and all blocks must be retransmitted. Thus, the verification delay for basic tree authentication is $O(n)$. By recursively calculating the nodes as blocks are

received, the verification of T requires a memory of the size $O(\log n)$. Table 2 illustrates the verification of the authentication tree shown in Figure 1.

Table 2 Storage requirements to verify a tree authentication for $n = 8$.

Received block	*Compute*	*Hash Storage*
B_1	$H(B_1)$	$H(B_1)$
B_2	$H(B_2), H_1$	H_1
B_3	$H(B_3)$	$H_1, H(B_3)$
B_4	$H(B_4), H_2, H_5$	H_5
B_5	$H(B_5)$	$H_5, H(B_5)$
B_6	$H(B_6), H_3$	H_5, H_3
B_7	$H(B_7)$	$H_5, H_3, H(B_7)$
B_8	$H(B_8), H_4, H_6$	H_5, H_6
$Sign(H_T)$	H_T	$Sign(H_T)$

An advantage of the tree authentication over other methods such as linear hashing is that an individual block can be updated in a logarithmic number of messages (or by a single message with a logarithmic number of components). To update B_i to B_i', the AP first associates the i-th leaf with B_i' and then recomputes the hash values of the tree to yield the new root value H_T'. The AP then signs the new root value H_T' and then sends H_T', its signature, and B_i'. The SC then recomputes the hash tree of its blocks after replacing B_i with B_i', and verifies that the newly computed root hash equals the received value of H_T'. If the hashes agree, and the signature is correct, then B_i is updated as B_i'. Thus, using tree authentication, n blocks can be downloaded in time $O(n)$ using $O(\log n)$ memory, and a block can be updated in $O(\log n)$ time and with $O(\log n)$ memory.

3. ORDERED TREE AUTHENTICATION

To verify a block, the verifier needs the tree signature and the siblings of each node in the block's path to the root. With this information, the verifier computes the hash values of each of the block's ancestors in the tree. That is, it first computes the hash value of the block, and then recursively the hash values of each ancestor by concatenating the last computed hash value with the corresponding node's sibling. For example, block B_3 is verified if $H'(B_3) = h(B_3)$, $H_2' = h(H'(B_3)||H(B_4))$, $H_5' = h(H_1||H_2')$, $H_T' = h(H_5'||H_6)$, and $H_T = H_T'$, where H_T is contained in the tree signature.

Note that the above calculation also verifies nodes $H(B_4)$, H_1, H_2, H_5, and H_6. If their values are cached, the verification of other blocks can be shortened. We define the *last verified node* for a block B_i to be the closest verified ancestor on the block's path to the root of the authentication tree. For example, after the above calculation, block B_2's last verified node is H_1 and block B_4's last verified node is $H(B_4)$. In general, it is sufficient to verify a block against its last verified node.

In the OTA scheme, we exploit the above observations. By sending the tree signature first, the root becomes the last verified node for all blocks. Sending block B_1 and the siblings of each node in the block's path the to root, verifies not only the path from $H(B_1)$ to the root but also the siblings of the nodes on the path. We observe that removing the nodes on the path from leaf $H(B_1)$ to the root splits the authentication tree into d subtrees, where d is the height of the tree. The roots of the subtrees are the siblings of the nodes on the path. Each sibling is the last verified node of its children. In the OTA scheme, blocks are sent in sequence. Each message contains one block and all the nodes that constitute the block's path to its last verified node.

In the remainder of this section, we specify the OTA scheme in detail via pseudo-code description of two routines called *SendBlocks* and *ReceiveBlocks*.

3.1 SENDING BLOCKS

As *SendBlocks* will act on the authentication tree T for the blocks, we assume the following tree structure and operations based on the concept of a *node*. For our purposes a *node* has a parent, a left and a right child, and a hash value. Let function $Parent(\cdot)$ return the parent node and let function $Right(\cdot)$ return the hash value of the right sibling. Let H_T denote the hash of the authentication tree. As by construction the leaves of the authentication tree correspond to blocks, we also let *Node*(B_i) denote the leaf corresponding to B_i.

The pseudo-code for the *SendBlocks* routine is shown in Figure 2. This routine will be run by the AP that wishes to send the blocks $B_1, B_2, \ldots, B_n$ to the SC. Each code block B_i will be sent in a message M_i, where M_i will contain B_i and possibly some additional hash values from T.

```
 1.  int depth ← log(n)
 2.  boolean cache[0 .. depth];
                                   /* generate and sign authentication tree, send signature */
 3.  T ← authentication tree for B_1,
B_2, ..., B_n;  4.  Send(Sign(H_T));
                                   AP signature on H_T  5.  cache[0]
← true;
                                   /* generate message blocks */
 6.  for i from 1 to n
do  7.  M_i ← B_i;
 8.  node ← Node(B_i);
 9.  j ← depth; 10.  while
cache[j] = false do                /* siblings of each node in the
path to the last verified node */ 11.  M_i ←
(M_i||Right(node)); 12.  cache[j] ←
true; 13.  node ← Parent(node); 14.  j ← j-1; 15.  od;  /* while */
16.  Send(M_i) 17.  cache[j] ←
false; 18.  od;                                        /* for */
```

Figure 2 Pseudo-code for the *Sendblocks* routine.

When the *SendBlocks* routine is executed, all n blocks $B_1, B_2, \ldots, B_n$ are passed as inputs. Routine *SendBlocks* maintains the following variables: $depth$ holds the height of the corresponding authentication tree T of the blocks, $cache$ is a boolean array of size $depth + 1$, and $node$ is a pointer into tree T. A field $cache[i]$ indicates whether a node on depth i constitutes the verified root of a subtree for a block not yet received.

First, *SendBlocks* constructs and hashes the authentication tree T for the blocks, with the resulting signature $Sign(H_T)$ sent to the SC. Also $cache[0]$ is set to true to indicate that the root hash has been sent. Next, the messages corresponding to the blocks that will be used to verify this signature are created. Routine *SendBlocks* has a main **for** loop (steps 6 to 18) that executes n times and processes B_i at the i-th iteration to generate M_i, which is sent to the receiver at step 16. At each iteration, message M_i is initialized to B_i (step 7), variable $node$ points to *Node*(B_i), the leaf node corresponding to B_i (step 8), and auxiliary variable j is set to the depth of the leaf nodes of T (step 9). Further processing adds any additional hash values to M_i that will be required at the SC to perform the verification of B_i after it is received.

We now explain how the additional hashes for B_i are determined. The **while** loop from steps 10 to 15 traverses the path from $Node(B_i)$ to the last verified node towards the root of the authentication tree by repeatedly assigning *node* to its parent (step 13). At each new node referenced by *node* for which the hash value of the right sibling has not been sent as part of a previous message (that is $M_1, M_2, \ldots, M_{i-1}$), this hash value is appended to M_i (step 11) and the cache field of the corresponding depth is set to true (step 12). Note that the **while** loop is executed for odd-numbered messages only; therefore there is always a right sibling. Equivalently, if *cache*[j] is false, then $Right(node)$ is appended to M_i at step 11, and $cache[j]$ is set to true to

indicate that this hash value need not be sent with any future message. This process of appending hash values to M_i continues in the **while** loop until $cache[j] = true$ indicates that the hash value of this depth was sent by a previous message. At this point, the message M_i for block B_i is complete and can be sent to the SC. The creation of the next message M_{i+1} for the block B_{i+1} begins at the $(i+1)$-st iteration of the main loop of *SendBlocks*. The main loop continues in this manner until all n messages for all n blocks have been generated and sent, at which point *SendBlocks* exits.

Example 1 Consider executing *SendBlocks* on the authentication tree of Figure 1 with the blocks $B_1, B_2, \ldots, B_8$. After the signature on H_T has been sent, the contents of the eight messages $M_1, M_2, \ldots, M_8$ corresponding to the blocks $B_1, B_2, \ldots, B_8$ are

$$\begin{aligned} M_1 &= \{B_1, H(B_2), H_2, H_6\}, \\ M_2 &= \{B_2\}, \\ M_3 &= \{B_3, H(B_4)\}, \\ M_4 &= \{B_4\}, \\ M_5 &= \{B_5, H(B_6), H_4\}, \\ M_6 &= \{B_6\}, \\ M_7 &= \{B_7, H(B_8)\}, \\ M_8 &= \{B_8\}. \end{aligned}$$

We note that the longest message is M_1 and that even-indexed messages contain no hash values because $H(B_{2i})$ will be sent with M_{2i-1}. □

We first prove a crucial property of the *SendBlocks* routine, the proper termination of the **while** loop.

Lemma 1 *Before processing message M_i, $1 \leq i \leq n$, there is at least one field in the $cache$ set to true.*

Proof. Before processing message M_1, $cache[0] = true$ holds (step 5). At every iteration of the **for** loop, only the field with the highest index set to true is set to false (step 17). However, the inner **while** loop will flip all fields holding value false, starting from $cache[depth]$ up to the field that corresponds to the depth of the last verified node. This means that all fields of the cache are only set to false if $cache[depth]$ is the only true field (the **while** loop is not executed). □

In other words, data structure $cache$ behaves like a counter if we regard its fields to represent a binary number, where $true$ denotes 1 and $cache[0]$ is the most significant bit. Its initial value is $n = 2^{\text{depth}}$. The **for** loop decrements its value at every iteration. After n iterations the counter is zero.

Lemma 2 *Message M_i contains exactly those hash values that are required to compute the hash of a subtree of T for which node $H(B_i)$ is the leftmost child.*

Proof. To compute the above subtrees, it is sufficient that message M_i contains block B_i together with the siblings of each node in the path from leaf $H(B_i)$ to block B_i's last verified node. The proof is by induction.

Base case. Before creating message M_1, $cache[0] = true$ holds (step 5). Execution of the **while** loop adds to message M_1 the siblings of each node in block B_1's path to its last verified node, root H_T. In addition, fields $cache[depth]$ to $cache[1]$ are set to true, and field $cache[0]$ is set to false.

Induction step. Let us assume that message M_i has been sent. Case analysis.

1. M_{i+1} is an even-indexed message: Then message M_i carried the hash value of block B_{i+1}, the sibling of block B_i, and field $cache[depth] = true$. Thus, no hash value is added to message M_{i+1} but $cache[depth] = false$ holds afterwards.

2. M_{i+1} is an odd-indexed message: Then block B_{i+1} is the left child of its ancestor and its last verified node is at depth j, $0 < j < depth$. Thus, execution of the **while** loop adds to message M_{i+1} the siblings

of each node in block B_{i+1}'s path to its last verified node at depth j. In addition, fields $cache[depth]$ to $cache[j+1]$ are set to true, and field $cache[j]$ is set to false.

□ We now prove some simple properties concerning the messages generated by *SendBlocks*.

Lemma 3 *Let $B_1, B_2, \ldots, B_n$ be a set of $n = 2^d$ blocks and $M_1, M_2, \ldots, M_n$ the corresponding n messages generated by SendBlocks for these n blocks. Then no message M_i contains more than d hash values, and no more than $n-1$ hash values are sent in total with the n messages.*

Proof. The **while** loop of *SendBlocks* cannot execute more than d times before the root of the authentication tree is reached, which proves the first statement of the lemma. To prove the second statement we observe that hash values appended to messages by *SendBlocks* must be labels of right children in the authentication tree. As there are $2n-1$ nodes in the authentication tree, at most $n-1$ hashes of right children can be sent in messages. □

The proof of this lemma shows that on average each message generated by *SendBlocks* contains at most one additional hash value.

3.2 RECEIVING BLOCKS

The *ReceiveBlocks* routine executes at the SC and verifies code blocks by processing the messages generated by the *SendBlocks* routine at the AP. The pseudo-code for the *ReceiveBlocks* routine is shown in Figure 3. *ReceiveBlocks* maintains a data structure called *cache*, which is an array of hash values. The fields of *cache* can be addressed from 0 up to d, the depth of the authentication tree T. For each depth i of the authentication tree T, *cache*$[i]$ holds hash value $H(j)$ of a node that was verified last in the processing of a previous message. In the setup phase (steps 2–5 of Figure 3), the *ReceiveBlocks* routine first reads and verifies the signature $Sign(H_T)$ on the authentication tree T, extracts the hash value H_T of the root, and then stores the hash H_T of the authentication tree in field *cache*$[0]$.

```
 2.   depth ← log(n);
 3.   cache[0 .. depth];                                   /* array of hash
values */
 4.   sig ← read();                                       /* read Sign(H_T) */  5.   cache[0]
← H_T;
 6.   for i from 1 to n do                                /* read messages and verify blocks */
 7.     M_i ← read();
 8.     B_i ← block from M_i;
 9.     h ← h(B_i); 10.      j
← depth;
11.     while cache[j] = 0 do 12.      cache[j] ← head(M_i); 13.      h
← h(h||cache[j]); 14.      j
← j-1; 15.      od;                  /* while */ 16.      if cache[j] = h 17.      then cache[j]
← 0; 18.      else error 19.      fi 20.      od;                                   /* for */
```

Figure 3 Pseudo-code for the *ReceiveBlocks* routine.

ReceiveBlocks has a main **for** loop (steps 6–20) that executes n times, receiving and verifying the i-th block B_i at the i-th iteration. At the i-th iteration, it receives message M_i (step 7), extracts block B_i from M_i (step 8), computes the hash value $H(B_i)$ of that block, and stores the computed hash value in the temporary variable h (step 9). Next *ReceiveBlocks* simulates the computation of the hash path for B_i in T until it reaches a node which has already been verified in the processing of a previous message $M_1, M_2, \ldots, M_{i-1}$ (steps 10 - 15). For each not yet verified node, *ReceiveBlocks* extracts the hash value of its right sibling from the received message M_i and stores the value in the corresponding field of the cache (step 12), computes the hash value of the parent node, and stores it in temporary variable h (step 13). Finally, the routine compares the computed

hash value stored in variable h with the hash value of the already verified intermediate node (step 16). If the values are equal, block B_i is considered verified and *ReceiveBlocks* clears the hash value of the already verified intermediate node in *cache* (step 17). Otherwise, *ReceiveBlocks* indicates an error (step 18). The main **for** loop continues in this manner until all n messages for all n blocks have been received and verified, at which point *ReceiveBlocks* exits.

Table 3 Storage requirements for OTA for $n = 8$.

Received message	*SC Compute*	*SC Verify*	*SC Hash Storage*
$Sign(H_T)$	–	–	H_T
$\{B_1, H(B_2), H_2, H_6\}$	$H(B_1), H_1, H_5, H_T$	H_T	$H(B_2), H_2, H_6$
$\{B_2\}$	$H(B_2)$	$H(B_2)$	H_2, H_6
$\{B_3, H(B_4)\}$	$H(B_3), H_2$	H_2	$H(B_4), H_6$
$\{B_4\}$	$H(B_4)$	$H(B_4)$	H_6
$\{B_5, H(B_6), H_4\}$	$H(B_5), H_3, H_6$	H_6	$H(B_6), H_4$
$\{B_6\}$	$H(B_6)$	$H(B_6)$	H_4
$\{B_7\}, H(B_8)$	$H(B_7), H_6$	H_4	$H(B_8)$
$\{B_8\}$	$H(B_8)$	$H(B_8)$	–

Example 2 The operation of the *ReceiveBlocks* routine as run on the messages produced by *SendBlocks* from Example 1 is shown in Table 3. After the signature on H_T has been verified, field *cache*$[0]$ holds H_T. All other fields are empty. Next, *ReceiveBlocks* receives the first message: $M_1 = \{B_1, H(B_2), H_2, H_6\}$. As the fields *cache*$[3]$, *cache*$[2]$, and *cache*$[1]$ are empty, the **while** loop extracts the three hash values $H(B_2)$, H_2, and H_6 from the message. These hash values are the siblings of the nodes on the path for B_1 and thus allow node H_T to be computed and verified. As a side effect, nodes $H(B_2)$, H_2, and H_6 are also verified and thus stored in the cache. As node H_T is verified, field *cache*$[0]$ is cleared.

Let us assume that the next message received contains an even-indexed block. Then the previous message contained the block's hash value and was stored in field *cache*$[3]$. Thus, the even-indexed block can be immediately verified and field *cache*$[3]$ will be cleared. In general, whenever a node is verified, the corresponding field in the cache is cleared (step 17).

Finally, let us consider the case where the next message received contains an odd-indexed block B_{2j+1}. We know from the discussion above that the previous iteration (iteration $2j$) of the **for** loop cleared field *cache*[3] and thus at least the hash value of the next block B_{2j+2} will be extracted from the message. If the parent node has not been verified previously, the hash value of the parent's right child will be extracted from the message. Note that intermediate nodes are verified before their right child are verified. Thus, walking up towards the root until an already verified node is reached, the shortest hash path to verify the code block is obtained. □

Lemma 4 The *ReceiveBlocks* routine verifies B_i given $M_1, M_2, \ldots, M_i$ in constant time using $O(\log n)$ storage.

Proof. The proof is by induction.

Base case. Message M_1 carries the right siblings of each node in the path from $Node(B_1)$ to but excluding the root H_T (Lemma 2) that has been sent before and is kept in field $cache[0]$. With this information, SC can compute the root value of T by iteratively computing the hash value of each parent node by combining the left child (first by computing the hash value of block B_1; later using the result of the last computation) with the hash value of the right child contained in message M_1. Block B_1 in message M_1 is verified. In addition, fields $cache[depth]$ to $cache[1]$ keep the hash values of the siblings, and field $cache[0]$ is set to zero.

Induction step. Let us assume that blocks $B_1, B_2, \ldots, B_i$ have been verified. Consider verifying block B_{i+1} with message M_{i+1}. Case analysis.

1. M_{i+1} is an even-indexed message: According to Lemma 2, message M_i carries the hash value of block B_{i+1}, the (right) sibling of block B_i. Hash value $H(B_{i+1})$ was stored in field $cache[depth]$. Thus, block B_{i+1} can immediately be verified. Field $cache[depth]$ is set to zero.

2. M_{i+1} is an odd-indexed message: Then block B_{i+1} is the left child of its ancestor and its last verified node is at depth j, $0 < j < depth$. Thus, message M_{i+1} contains the siblings of each node in block B_{i+1}'s path to its last verified node at depth j. Execution of the **while** loop calculates the hash value of block B_{i+1}'s last verified node and compares that value with the hash value stored at field $cache[j]$. In addition, fields $cache[depth]$ to $cache[j+1]$ keep the hash values of the siblings, and field $cache[j]$ is set to zero.

Routine *ReceiveBlocks* stores the intermediate hash values in array $cache$ of size $\log n + 1$. □

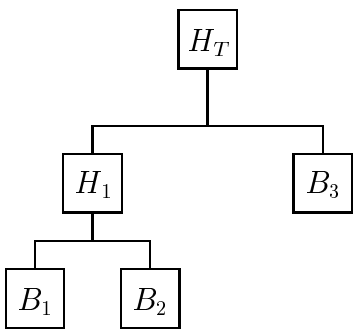
4. INCREMENTAL BLOCKS

It may also be the case that a given application of n blocks $B_1, B_2, \ldots, B_n$ is to be increased to have $n+1$ blocks with the addition of a new block B_{i+1}. Adding a new block can be considered a special update operation in OTA. Let H_T be the OTA authentication tree for the blocks $B_1, B_2, \ldots, B_n$, which will consist of a binary tree with internal nodes and leaves, where each internal node will have two children (excluding the case of $n = 1$). Each leaf is a block B_i, and is positioned at some depth d in H_T. To add a new block B_{n+1} to H_T, one considers the set of leaves that are at the minimum depth d in H_T, and pick one at random, here denoted B_i. Then the node for B_i is replaced by an internal node that has B_i and B_{n+1} as its children.

To add the new B_{i+1} block to those existing in the SC, the AP adds B_{n+1} to the H_T as described above and then computes the new root value H'_T. Then, the AP sends H'_T, its signature $Sign(H'_T)$, the index i, and B_{n+1} to the SC, where i indicates the leaf in the current authentication tree that will become the parent of the new block. The SC inserts B_{n+1} into the authentication tree, recomputes the hash tree and verifies that the newly computed root hash equals the received value of H'_T. If the hashes agree, and the signature is correct, then B_{n+1} is added to the blocks. The above protocol naturally extends to the case where m new blocks are added.

5. CONCLUSIONS

The OTA scheme presented for downloading to and/or updating and authenticating data or applications on a portable device has clear advantages for the chosen specific example of a smart card as portable device as well as for other applications. The algorithm can easily be adapted and applied to any problem where complex applications must be downloaded to or updated in a device having constraints in memory, processing speed and/or bandwidth or where updating time and/or security play significant roles.



Message	*SC Storage*
$Sign(H_T)$	H_T
$\{B_1, H(B_2), H(B_3)\}$	$H(B_2), H(B_3)$
$\{B_2\}$	$H(B_3)$
$\{B_3\}$	–

Figure 4 An authentication tree for $n = 3$ and $d = 2$.

x

If the number of blocks to be downloaded is not a power of 2, there is a simple change to adapt the two routines. Before executing the **while** loop, variable *j* is assigned the value of the height of the authentication tree (j $\leftarrow$ depth). It is sufficient to replace the constant *depth* by a function, $Depth(\cdot)$, that returns the depth of the leaf associated with the current block to be processed. If we take, for example, an application that consists of three blocks, we get the authentication tree in Figure 4. The major difference is in sending/receiving the third block. After the second message has been processed, $cache[1] = H(B_3)$ holds. Setting the initial value of variable j to 1 (j $\leftarrow Depth(B_3)$ instead to 2 (j $\leftarrow$ depth) skips the processing of the (missing) nodes at depth 2.

References

[CASCADE, 1997] See `http://www.dice.ucl.ac.be/crypto/cascade/cascade.html`.

[SHA, 1994] (1994). FIPS 180-1, Secure Hash Standard, Federal Information Processing Standards Publication 185, US Department of Commerce/NIST, National Technical Information Service, Springfield, Virginia, 1994. Available at http://csrc.nist.gov/fips/fip180-1.ps.

[Dhem, 1998] Dhem, J.F. (1998). *Design of an efficient public key cryptographic library for RISC-based smart cards*. PhD thesis, Université catholique de Louvain. Available at `http://www.dice.ucl.ac.be/crypto/dhem/dhem.html`.

[Merkle, 1989] Merkle, R.C. (1989). A certified digital signature. *Advances in Cryptology, CRYPTO 89, Lecture Notes in Computer Science, vol. 218, G. Brassard ed., Springer-Verlag*, pages 218–238.

[Wong and Lam, 1999] Wong, C.K. and Lam, S.S. (1999). Digital signatures for flows and multicasts. *IEEE/ACM Transactions on Networking*, 7(4):502–513.