fbMHT: A file-base MHT for efficient data integrity verification

Di Lu

July 8, 2022

Abstract

Merkle Hash Tree (MHT) has been widely adopted to achieve effective data integrity verification in various fields, such as cloud computing, BitTorrent downloading, blockchain, and database. However, due to the needing of maintaining the tree in the memory, preserving and handling an MHT in the memory-constrained systems, such as embedded systems, is not realistic. This report describes a file-based MHT (fbMHT) design and implementation for efficient data integrity verification. In fbMHT, an MHT, a complete binary tree, is organized in a sequential file, for which a set of file-based MHT operations are implemented, such as fbMHT creation, node searching, node updating, and node insertion. With fbMHT, the MHT can be maintained and handled in a file instead of occupying excessive memory when the MHT continuously grows. Hence, this scheme suits embedded systems with limited or strict memory utilization requirements. Memory-occupation analysis proves that fbMHT can achieve a small memory footprint and the performance evaluation shows the feasibility, usability, and limitations.

1 Introduction

To address the performance issue of integrity protection for large amounts of data, Merkle Hash Tree (MHT) has been studied for a long time and is suitable in such scenario, which only needs $O(\log_2 n)$-sized Merkle proof to finish integrity verification for a data set with n pieces of data. Figure 1 shows an example of an MHT (the left part of the figure) with 8 leaf nodes and the Merkle proof (the right part of the figure) when verifying the integrity of data piece d'_1 (d'_1 is supposed to be different from d_1, which may be tampered with). In the MHT, the value (h_i) of the each leaf node can be obtained by $hash(d_i)$, and each interior node's value is generated by hashing the concatenated digests of its children. E.g., $h_{56} = hash(h_5||h_6)$, in which $x||y$ denotes a concatenation of x and y. For the Merkle proof, only the nodes in the dashed circles (h_2, h_{34} and h_{58}) are necessary in integrity verification of d'_1. Thus, only the three hashes need to be loaded into the memory, which only takes a very small amount of space.

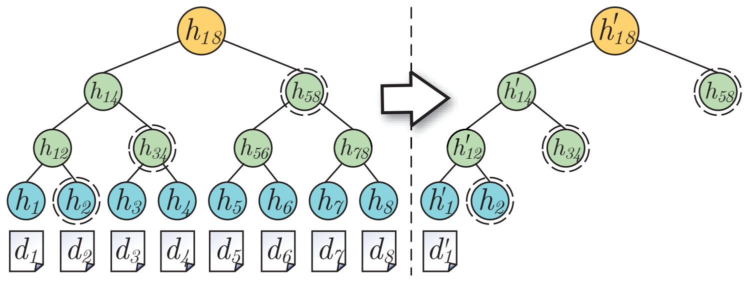


Figure 1: An example of an MHT tree and the Merkle proof of data piece d_1

However, due to the limited computational resources of embedded systems, loading and maintaining the entire MHT in the memory with limited capacity is not realistic; particularly, the MHT can continuously grow with the increase of data. Hence, in our design, the MHT will be created and maintained in the form of a sequential file on the external storage, such as TF-card and flash ROM, which has a much larger capacity compared to the memory. However, there are two major challenges while storing MHT file on the external storage:

1. Since only a file pointer can be moved forward or backward while handling an MHT file, a portion of the memory is also used to temporarily store block data being processed to improve the efficiency of file operations. Thus, the occupied memory used to improve file operation performance needs to be controlled and minimized.

2. Considering the external storage has a lower speed than the memory, the file pointer should avoid being moved in long-distance frequently in designing the MHT file-related algorithms. E.g., when locating a block blk_x, the effective way is to directly move the file pointer to each block in the path from the MHT root to blk_x instead of resetting the file pointer to the beginning of the file every time.

3. With the growth of the data, the MHT file size will also increase. Thus, it is inevitable for the file pointer to jump over a long distance during locating a block while handling an MHT file in a large size. Particularly, when inserting a new block into the MHT file, a large number of existing data blocks need to be moved to produce extra space, which can increase the non-negligible performance overhead.

Due to the challenges mentioned above, our design goals can be summarized as follows:

• *Design Goal 1* ***(G1)***. Small footprint in the memory: the amount of memory used to improve file operation performance should be minimized according to the platform conditions.

• *Design Goal 2* ***(G2)***. Frequent file pointer long-jump-proof: when handling the MHT file, every file pointer move should be effective, and frequent long-distance jump should be avoided.

• *Design Goal 3* ***(G3)***. Excessive data moving-proof: to avoid moving lots of data blocks while inserting a new block to the MHT file.

2 System Design

2.1 File Structure

An MHT file can be divided into two main part: 1) the file header (FH) and 2) the data block area (DBA). The file header stores the meta data of the file, whereas each data block in DBA preserves the information about an MHT node, including *data index*, *data hash*, *node level*, and the information about its parent and brother node, which will be detailed in table 1 of section 2.2. Figure 2 shows the internals of an MHT file. The file header has two vital fields: 1) the root block (node) pointer (Φ_{rn}), which points to the offset of the MHT root block; and 2) the first supplementary block (node) pointer (Φ_{fsn}), which indicates the offset of the first supplementary block ($SpBlk - 1$) in the file (details about the supplementary block will be elaborated in section 2.2-A).

2.2 Key Algorithms

To realize basic operations on fbMHT of single-file version, four fundamental algorithms are proposed as follows, including *building an MHT file*, *searching/updating a designated data block* and

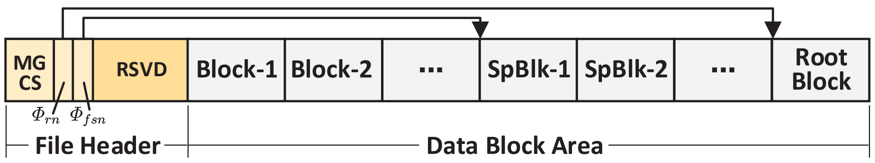


Figure 2: An overview of a single-file fbMHT structure

Table 1: Node structure of the double-linked queue

Attributes	Comments
index	node index used to identify a node and support binary search
tmp_idx	index of the right most leaf node of the current node's left child
level	node level which starts from 0 (0 refers to leaf node)
hash	data hash value
lchild_idx	left child node's index
lchild_os	distance from the current node to its left child in the queue
rchild_idx	right child node's index
rchild_os	distance from the current node to its right child in the queue
parent_idx	parent node's index
parent_os	distance from the current node to its parent in the queue

inserting a new data block. To illustrate these algorithms, some essential data structure and symbols are given in table 1 and 2 below. Note that, in table 1, *index* can be considered as the unique integer identifier of the data, which can be used to data retrieval. *tmp_idx* is set to temporarily store the index of the rightmost leaf node of an interior node's left child sub-tree and used as the index of the interior node's parent, which makes fbMHT support binary search when querying data with a given index. Additionally, attributes *lchild_idx*, *rchild_idx* and *parent_idx* are also used for realizing binary search in MHT. *level* indicates a node's level in MHT, and 0 refers to a leaf node, whereas the maximum refers to the root. Attributes *lchild_os*, *rchild_os* and *parent_os* record the distance from the current node to the corresponding nodes in the queue, by which we can quickly locate them in MHT file (how to calculate and use these values will be detailed in the discussion on algorithm 1). In table 2, since the MHT file is sequential, the file pointer f_p, which indicates the offset (relative to the beginning of the file) of the block to be processed, can be only moved forward or backward in the file.

A. Building an MHT file

A-1. A Memory-consuming (MMCS) Version

The primary challenge of building an MHT file is to generate a serialized MHT during processing the data set. Hence, we have adopted a double-linked queue to sequentially output MHT nodes to fill the MHT file. Algorithm 1 gives the procedure of building a new MHT file. At the beginning of the algorithm, a double-linked queue and an output file handler are initialized, and the header of the MHT file will be reserved and then completed when the algorithm ends. The main body of the algorithm can be divided into three main stages: **1)** *building the left complete sub-tree (BLCST)* (*line 3-12*), **2)** *supplementing the right complete sub-tree (SRCST)* (*line 14-16*), and **3)** *processing the root (PR)* (*line 17-21*) (here, *"left"* and *"right"* are relative to the root). These three stages will be detailed as follows.

1) BLCST: to facilitate management and expansion, the MHT in our design is considered as a complete binary tree. However, in reality, the number of data pieces in data set ***D*** (denoted as

Table 2: Essential symbols used in algorithms

Symbols	Comments
$\boldsymbol{D}$	data set to be processed
d_i	i^{th} data piece in $\boldsymbol{D}$, $1 \leq i \leq \|\|\boldsymbol{D}\|\|$
Q_{db}	the double-linked queue
Q_{hdr}, Q_{tail}	Q_{db}'s header and tail
q_x	a node of Q_{db}, $q_x \in Q_{db}$
f_p	a file pointer indicating the offset where data will be written
f_{MHT}	MHT file
s_{blk}	MHT block size
f_{hdr}	the header of the MHT file
Φ_{fsn}	the offset where the first supplementary node (fsn) locates
Φ_{rn}	the offset where the root node (rn) locates

N and $N = ||\boldsymbol{D}||$) usually does not satisfy $\log_2(N) \in \{0, \mathbb{Z}^+\}$, which implies that we cannot build a complete binary tree due to lacking of $2^{\lceil \log_2(N) \rceil} - N$ data pieces (Theorem 1). Hence, stage SRCST is used to supplement nodes for MHT, which makes it a complete binary tree (SRCST will be detailed in the discussion of stage 2). Figure 3(a) shows an MHT with insufficient leaf nodes cannot become a complete binary tree, whereas figure 3(b) shows a complete MHT in which supplementary nodes ($s_1 \sim s_3$) are provided.

By observing figure 3(a), we can find that the left sub-tree of the root is complete even if it has a incomplete right sub-tree. In fact, for an MHT constructed from left to right with any number of data pieces, the left sub-tree of the root node is always a complete binary tree (Theorem 2). Hence, the stage BLCST is firstly processed in algorithm 1 (line 3-12).

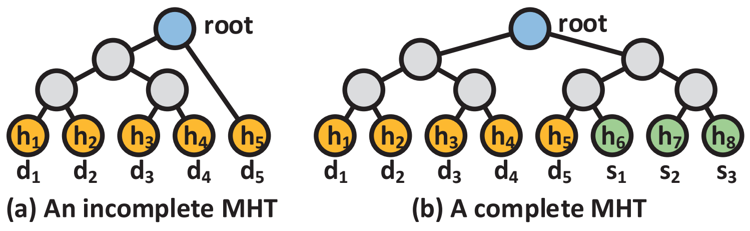
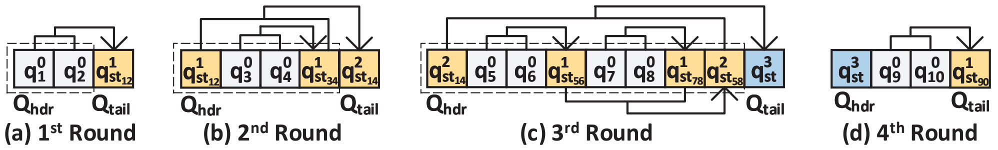


Figure 3: (a) an incomplete MHT with insufficient leaf nodes, where the hash values (h_x) of data pieces (d_x) are stored; (b) a complete MHT with supplementary nodes, $s_1 \sim s_3$.

As an important premise, all the data pieces in $\boldsymbol{D}$ are increasingly sorted by data index. Then, the main loop of this stage sequentially traverses each data piece (d_i) in data set. Once a data piece (d_i) is picked from $\boldsymbol{D}$, it will be hashed to create a new queue node q_i via invoking function `MakeQNode`, which will be then enqueued. After that, we will merge nodes in pairs that have the same level to the greatest extent possible via calling `CombineNodes` (this function will be detailed in algorithm 2). The merging process will pause when a node with higher level has been generated, and all the nodes before the new merged node will be dequeued and written to the MHT file in order. Finally, when the traversal of $\boldsymbol{D}$ is finished, the stage BLCST ends. Note that there will be two results: a) $||Q_{db}|| > 1$, which means $||\boldsymbol{D}||$ is not a power of 2, which needs to be supplemented with extra nodes to build a complete MHT (detailed in stage SRCST), and b) $||Q_{db}|| = 1$, which implies that $||\boldsymbol{D}||$ is exactly a power of 2 and the remained node is MHT root (detailed in stage

PR). For result a), the queue's header is the left sub-tree root (denoted as q_{st}) of the MHT which currently has the maximal level. Figure 4 shows an example of the variation of a queue during the execution of stage BLCST when building an MHT with 10 leaf nodes. q_x^y denotes a queue node with index x and level y. Particularly, $q_{st_{jk}}^y$ indicates that the node is generated by merging both q_j^y and q_k^y, which is also their parent node. We can find that only the nodes with the maximal level remains in the tail (Q_{tail}) in the first three rounds, including $q_{st_{12}}^1$, $q_{st_{14}}^2$ and q_{st}^3, and all the nodes before these three nodes will be dequeued and written to the MHT file sequentially (the dashed frames contain the nodes to be dequeued). In the 4^{th} round, the stage BLCST is over since the last two nodes (q_9^0 and q_{10}^0) in $\boldsymbol{D}$ are processed. However, the two nodes can be only merged to create a sub-tree root with level 1 ($q_{st_{90}}^1$), which cannot be merged with q_{st}^3. Thus, node q_{st}^3, as the queue's header, is the MHT's left sub-tree root.



q_1^0 q_2^0 $q_{st_{12}}^1$
Q_{hdr} Q_{tail}
(a) 1st Round
$q_{st_{12}}^1$ q_3^0 q_4^0 $q_{st_{34}}^1$ $q_{st_{14}}^2$
Q_{hdr} Q_{tail}
(b) 2nd Round
$q_{st_{14}}^2$ q_5^0 q_6^0 $q_{st_{56}}^1$ q_7^0 q_8^0 $q_{st_{78}}^1$ $q_{st_{58}}^2$ q_{st}^3
Q_{hdr} Q_{tail}
(c) 3rd Round
q_{st}^3 q_9^0 q_{10}^0 $q_{st_{90}}^1$
Q_{hdr} Q_{tail}
(d) 4th Round

Figure 4: An example of the variation of a queue during the execution of stage BLCST when building an MHT with 10 leaf nodes. Figures (a)-(d) show the queue's status in each round of iterations. Note that q_{st}^3 in the blue square is the root of the left sub-tree of the MHT root.

2) SRCST: as the discussion on stage BLCST, when N is not an n^{th} integer power of 2 ($n \in \{0, \mathbb{Z}^+\}$), the data pieces in data set $\boldsymbol{D}$ are insufficient to build a complete MHT. When stage BLCST finishes, there will be

$$\zeta = \sum_{i=0}^{k-1} b_{k-i}(2^{k-i} - 1) \tag{1}$$

nodes left in the queue (Theorem 3), and $2^{\lceil \log_2(N) \rceil} - N$ supplementary leaf nodes are needed (Theorem 1). For example, for figure 4, $N = 10$, thus, according to equation 1 and theorem 3, $n = 2, k = 2, b_2 = 1, b_1 = 0$, there are 3 nodes left in the queue except the queue's header q_{st}^3 (referring to figure 4(d)), and 6 supplementary leaf nodes need to be added.

By observing SRCST, we can find that the main objective is to build the "right sub-tree" of the MHT with the supplementary leaf nodes, which are similar to stage BLCST. Assuming the level of the queue's header at the beginning of SRCST is l_{hdr}, the end condition of the main loop of SRCST is to create a new node with the same level, which can be merge with the header node to create the MHT root. Note that we use $hash(0)$ (in our design, the hash function is `SHA256`) to create the supplementary nodes whose indices are the maximum by default. For example, in 32-bit system, the maximum signed integer is `0x7FFFFFFF`. During the process of SRCST, the offset of the *"first supplementary node"* in the MHT file, denoted as $\varPhi_{fsn}$, will be recorded, which will be further written into the corresponding section of the MHT file header. Function `SupplementRCST` implements such a process which is shown as algorithm 5. Note that algorithm 5 can be called if and only if $||Q_{db}|| > 1$.

3) PR: in this stage, there is only one node left in the queue (header node), which is the MHT root (Theorem 4). Note that the offset of the root node in the MHT file, denoted as $\varPhi_{rn}$, will be recorded, which will be also further written into the corresponding section of the MHT file header.

Algorithm 2 shows the implementation of the function `CombineNodes`(), which is a *"Greedy Algorithm"* and merges nodes with the same level to the greatest extent. The algorithm searches for a node that can be merged with the tail node by scanning backward from the tail node and

Algorithm 1: Building a new fbMHT

```
Input: $\boldsymbol{D}$: Data sequence to be processed
Output: $f_{MHT}$: A new created fbMHT file
1  Initialization;
2  FileWrite ($f_p$, $f_{hdr}$);
3  foreach $d_i$ in $\boldsymbol{D}$, $1 \leqslant i \leqslant ||\boldsymbol{D}||$ do
4      $q_i \leftarrow$ MakeQNode (SHA256 ($d_i$));
5      Enqueue ($Q_{db}$, $q_i$);
6      CombineNodes ($Q_{db}$);
7      if There exists a new created subtree $q_{st}$ then // $q_{st}$ is returned by CombineNodes
8          while $Q_{hdr}.level < q_{st}.level$ do
9              $q_{pop} \leftarrow$ Dequeue ($Q_{db}$);
10             FileWrite ($f_p$, $q_{pop}$);
11         end
12     end
13 end
14 if $||Q_{db}|| > 1$ then
15     SupplementRCST ($Q_{db}$,$f_p$);
16 end
17 if $||Q_{db}|| = 1$ then
18     $p_{pop} \leftarrow$ Dequeue($Q_{db}$);
19     $\Phi_{rn} \leftarrow f_p$;
20     FileWrite ($f_p$, $q_{pop}$);
21 end
```

Algorithm 2: *CombineNodes*() : Creating sub-trees via combining nodes in the same level

```
Input: $Q_{db}$: the double-linkded queue
Result: a new sub-tree root is created
1  $q_{ptr} \leftarrow Q_{tail}$;
2  while $q_{ptr}.prev \neq Q_{hdr}$ do
3      if $q_{ptr}.level > Q_{tail}.level$ then
4          break;
5      end
6      if $q_{ptr}.level = Q_{tail}.level$ then
7          $q_{st} \leftarrow$ MakeSubtree($q_{ptr}, Q_{tail}$) ; // $q_{st}$ increases
8          Enqueue ($Q_{db}$, $q_{st}$);
9          ProcessOffset ($q_{st}$, $q_{ptr}$, $Q_{tail}$);
10         ProcessIdx ($q_{st}$, $q_{ptr}$, $Q_{tail}$);
11     end
12     $q_{ptr} = q_{ptr}.prev$;
13 end
14 return $q_{st}$;
```

then enqueues the merged nodes. In the beginning, we temporarily store the queue's tail as q_{ptr}, since the tail node will be modified when a new merged node is enqueued. Note that there are two end conditions of the main loop:

1) $q_{ptr}.level > Q_{tail}.level$: in this case, a node with a higher level is found (denoted as q^*), which means a complete binary tree rooted at q^* has been established and the nodes between q^* and Q_{tail} cannot be merged with q^* to create a new node with higher level. Hence, the loop will abort.

2) $q_{ptr} = Q_{hdr}$: in this case, the pointer q_{ptr} has reached the queue's header, thus, the loop will abort.

Once two nodes with the same level are found, function `MakeSubtree(lchild, rchild)` is used to create a sub-tree root (denoted as q_{st}), whose left and right children are *lchild* and *rchild* respectively, and $q_{st}.level = lchild.level + 1$ (or $rchild.level + 1$). Then, q_{st} will be enqueued, and the queue's tail (Q_{tail}) becomes q_{st}. After that, the attributes *lchild_os*, *rchild_os*, *parent_os*, *lchild_idx*, *rchild_idx* and *parent_idx* of q_{st}, q_{ptr} and Q_{tail} will be updated by functions `ProcessOffset` and `ProcessIdx` respectively. The offset information will be further used to locate the node block in the MHT file, whereas the indices are used to implement binary search on node blocks in the MHT file. The implementations of the functions `ProcessOffset` and `ProcessIdx` will be detailed in algorithms 3 and 4.

Algorithm 3: $ProcessOffset(root, lchild, rchild)$: Computing offset attributes within the nodes of a minimal sub-tree

```
Input: $root$: the root of the sub-tree
       $lchild$: the left child of the sub-tree
       $rchild$: the left child of the sub-tree
Result: The related offset attributes within $root$, $lchild$ and $rchild$ are updated
1 $d_{pl} \leftarrow$ the distance (absolute value) from $root$ to the left child ($lchild$);
2 $d_{pr} \leftarrow$ the distance (absolute value) from $root$ to the right child ($rchild$);
3 $root.lchild\_os \leftarrow -d_{pl}$;
4 $root.rchild\_os \leftarrow -d_{pr}$;
5 $lchild.parent\_os \leftarrow d_{pl}$;
6 $rchild.parent\_os \leftarrow d_{pr}$;
```

Algorithm 3 shows the implementation of function `ProcessOffset`$(root, lchild, rchild)$, which computes relative distance between the *parent* and *lchild*/*rchild* nodes of a minimal sub-tree (denoted as d_{pl} and d_{pr}). Here, a minimal sub-tree refers to a tree composed of three nodes: *root* (*parent*), *left child* (*lchild*) and *right child* (*rchild*). In the queue, d_{pl} and d_{pr} are the number of nodes locating between *parent* & *lchild* and that between *parent* & *rchild* respectively, which can be used as the relative offsets in the MHT file. Since we take the position of the root node as the starting point for distance computation, both *lchild* and *rchild* are located in front of the parent node. Thus, attributes *lchild_os* and *rchild_os* of the parent node (root) are negative (line 3-4), whereas *parent_os* of *lchild* and *rchild* are positive values (line 5-6). For example, as shown in figure 4(c), nodes $q^1_{st_{56}}$, $q^1_{st_{78}}$ and $q^2_{st_{58}}$ form a minimal sub-tree, in which $q^2_{st_{58}}$ is *root* (parent), $q^1_{st_{56}}$ and $q^1_{st_{78}}$ are *lchild* and *rchild* respectively. In this case, $q^2_{st_{58}}.lchild\_os = -4$, $q^2_{st_{58}}.rchild\_os = -1$, $q^1_{st_{56}}.parent\_os = 4$ and $q^1_{st_{78}}.parent\_os = 1$. When locating node blocks in the MHT file, since the offset values of all the nodes except the MHT root are relative, once the root block has been located by the file pointer f_p according to $\varPhi_{rn}$, all the tree nodes can be accessed by moving f_p from the current position by referring the offset attributes.

As mentioned in the discussion on algorithm 2, the index attribute of each node can be used to implement binary search in the MHT file. Algorithm 4 shows the implementation of the function `ProcessIdx`, which gives the details on how to designate an index for each node to realize a BST (binary search tree) from bottom to the top. Same as the function `ProcessOffset`, a minimal sub-tree, including a root, a left child (*lchild*) and a right child (*rchild*), is the fundamental processing unit for `ProcessIdx`. According to the characteristics of BST, we stipulate that $root.index \leq lchild.index$ and $rchild.index > root.index$. Hence, we divide the index assignment into two cases based on the types of *lchild* and *rchild* nodes:

1) *lchild* and *rchild* are leaf nodes: in this case, *root.index* is *lchild.index*, and *rchild.index* will be temporarily stored in *root.tmp_idx* for the further use.

2) *lchild* and *rchild* are interior nodes: in case 1), *root* is an interior node which can be either

Algorithm 4: *ProcessIdx*$(root, lchild, rchild)$: Computing index of nodes, including leaf and interior nodes

```
Input: root: the root of the sub-tree
       lchild: the left child of the sub-tree
       rchild: the left child of the sub-tree
Result: The related offset attributes within root, lchild and rchild are updated
1  if $lchild.level$ and $rchild.level$ equal to 0 then
2      $root.index \leftarrow lchild.index$;
3      $root.tmp\_idx \leftarrow rchild.index$;
4  else
5      $root.index \leftarrow lchild.tmp\_idx$;
6      $root.tmp\_idx \leftarrow rchild.tmp\_idx$;
7  end
8  $root.lchild\_idx \leftarrow lchild.index$;
9  $root.rchild\_idx \leftarrow rchild.index$;
10 $lchild.parent\_idx \leftarrow root.index$;
11 $rchild.parent\_idx \leftarrow root.index$;
```

a left or right child of a higher-level root (denoted as $root^*$), thus, if $root$ is the left child ($lchild$) of $root^*$, $root^*.index$ is $root.tmp\_idx$, whereas $root^*.tmp\_idx$ will be $root.tmp\_idx$ if it is the right child ($rchild$) of $root^*$. In this way, the index of a certain leaf node that acts as the right node can be passed up and assigned to a proper interior node (including the MHT root). The proof of theorem 5 indicates the correctness of algorithm 4.

Algorithm 5: *SupplementRCST*(Q_{db}, f_p) : Supplementing new nodes to the queue Q_{db} to build a complete right sub-tree of the MHT.

```
Input: $Q_{db}$: the queue used to build fbMHT
       $f_p$: the MHT file pointer
Result: A newly built complete binary tree as the right sub-tree having been written into the MHT file,
        and only the MHT root left in $Q_{db}$
1  Firstly, $||Q_{db}|| > 1$ must be satisfied before calling this function;
2  $l_{hdr} \leftarrow Q_{hdr}.level$;
3  while $Q_{tail}.level <= l_{hdr}$ do
4      $q_0 \leftarrow$ MakeQNode(SHA256(0));
5      Enqueue ($Q_{db}$, $q_0$);
6      CombineNodes ($Q_{db}$);
7      if There exists a new created subtree $q_{st}$ then        // $q_{st}$ is returned by function CombineNodes
8          while $Q_{hdr}.level < q_{st}.level$ do
9              $q_{pop} \leftarrow$ Dequeue ($Q_{db}$);
10             if $q_{pop}$ is the first supplementary node then
11                 $\Phi_{fsn} \leftarrow f_p$;
12             end
13             FileWrite ($f_p$, $q_{pop}$);
14         end
15     end
16 end
17 Now, only the MHT root node ($q_{rt}$) left in $Q_{db}$;
```

A-2. A Memory-saving (MMSV) Version

With the growth of $||\boldsymbol{D}||$, the queue (Q_{db}) size can become too large to be maintained by the memory constrained systems, such as low-end embedded systems with limited memory (e.g., STM32F407VET6 MCU has only 192KB RAM), during the process of building MHT file. For example, let Q denotes the queue used to build an MHT file, and at a certain moment, the queue's

Table 3: An example on how the queue's size varies with level L

L	1	2	4	6	8	10	12	14	16	18
S_Q (KB)	0.34	0.62	2.26	8.82	35.07	140.07	560.07	2.24×10^3	8.96×10^3	3.58×10^4

header is q_x^l (l denotes the node level). According to algorithm 1, a new sub-tree with root q_y^l needs to be constructed to combine with q_x^l to form a bigger tree with root q_{xy}^{l+1}. After enqueuing q_{xy}^{l+1}, all the nodes before it can be dequeued. Thus, it can be easily obtained that there are $2^{l+1}+1$ nodes in Q before dequeuing happens (Theorem 7). Table 3 shows how the queue size (denoted as S_Q) varies with level L (each MHT file block's size is 70 bytes). Obviously, when $L = 10$, $S_Q = 140.07KB$, which cannot be neglected for some low-end MCUs (e.g., for STM32F407VET6, the availability and usability can be seriously affected if $S_Q = 140.07KB$, which has only 192KB RAM).

Hence, to reduce the memory consumption during building MHT file, we have proposed "Memory-saving Queue (MSQ)" to adapt for such scenario to satisfy our design goal ***G1***. Algorithm 6 gives the detailed implementation on MSQ-based MHT file creation. As an important premise, data set $\boldsymbol{D}$ has been processed to satisfy $\log_2 ||\boldsymbol{D}|| \in \{\mathbb{Z}^+\}$. Thus, if $\boldsymbol{D}$ has been extended, there are supplementary elements with indices idx_{IV} and data 0 in $\boldsymbol{D}$ (idx_{IV} denotes an invalid index value. For example, it can be set to the maximum integer in 32-bit system, `0x7FFFFFFF`). Algorithm 6 can be divided into three parts:

1. Making sub-trees and dequeuing useless nodes (line 3-8): in this part, each data element (d_i) in $\boldsymbol{D}$ is will be converted into a corresponding queue node q_i and be enqueued. Meanwhile, the queue will be check by function `CombineNodesWithSameLevels` if any two nodes with the same level can construct a sub-tree. After finishing sub-tree construction, the "useless nodes" will be dequeued for memory saving (the "useless nodes" will be further explained in function `CombineNodesWithSameLevels`).

2. Processing root node (line 9-12): after finishing part 1, there will be only one node left in the queue, which is the root and has been written into the MHT file. Thus, the queue can be emptied by a dequeuing operation and released.

3. Updating file header (line 13): finally, the header of MHT file will updated, including the values of Φ_{fsn} and Φ_{rn}.

As the key sub-routine, function `CombineNodesWithSameLevels` adopts a greedy policy to search any possible node pairs from Q_{tail} to Q_{hdr} to make sub-trees, and removes the nodes *which have been written into the MHT file and will never be used to construct any new sub-trees* (an explanation on "useless nodes"). Algorithm 7 shows the detailed implementation on function `CombineNodesWithSameLevels`. The main loop keeps checking the two elements at the end of the queue ($q_{rchild} \leftarrow Q_{tail}$ and $q_{lchild} \leftarrow Q_{tail}.prev$) whether they can be combined into a new sub-tree whose root becomes the new queue tail (line 2-7). Now, q_{lchild} becomes $Q_{tail}.prev.prev$, whereas q_{rchild} becomes $Q_{tail}.prev$; and both of them will be dequeued by `DequeueSpPos` and written into the MHT file (the queue node property $is\_written$ indicates that whether it has been written into the MHT file). Different from the conventional dequeuing operation, `DequeueSpPos`(Q, q) will locate node q in Q and pop it, instead of popping any nodes from the queue's header. During this period, the offset of the first supplementary node (Φ_{fsn}) will be recorded for the further file header update (line 13-15). Note that if the popped node is an interior node, which has been written into MHT file, the index of the file block corresponding to the node should be updated by `UpdateBlockIndexInfo`, since the properties of such interior nodes can be changed when new sub-trees has been created.

Algorithm 6: Building a new fbMHT, a MSQ-based version

```
   Input: $\boldsymbol{D}$: Data sequence to be processed
   Output: $f_{MHT}$: A new created fbMHT file
1  Initialization for $Q_{db}$ ;
2  FileWrite ($f_p$, $f_{hdr}$);
3  foreach $d_i$ in $\boldsymbol{D}$, $1 \leqslant i \leqslant ||\boldsymbol{D}||$ do
4      CombineNodesWithSameLevels ($f_p$, $Q_{db}$);
5      $q_i \leftarrow$ MakeQNode (SHA256 ($d_i$));
6      Enqueue ($Q_{db}$, $q_i$);
7  end
8  CombineNodesWithSameLevels ($f_p$, $Q_{db}$);
9  $q_{pop} \leftarrow$ Dequeue($Q_{db}$);
10 if $q_{pop}.is\_written$ and $q_{pop}.level > 0$ then
11     $\Phi_{rn} \leftarrow$ offset of the block related to $p_{pop}$;
12 end
13 Updating $f_{hdr}$ with $\Phi_{rn}$ and $\Phi_{fsn}$;
```

After finishing processing nodes q_{lchild} ($Q_{tail}.prev.prev$) and q_{rchild} ($Q_{tail}.prev$), the sub-tree root (Q_{tail}) will be written into the MHT file without being dequeued (line 24-27), which is remained to construct another sub-tree with a higher level.

Algorithm 7: `CombineNodesWithSameLevels`(f_p, Q_{db}): Combining nodes with the same levels and dequeuing the new combined sub-tree in time.

```
   Input: $f_p$: the file pointer for the new built MHT file
   Output: $Q_{db}$: the queue used to build MHT file
1  while $Q_{tail} \neq Q_{hdr}$ and $Q_{tail}.level = Q_{tail}.prev.level$ do
2      $q_{lchild} \leftarrow Q_{tail}.prev$;
3      $q_{rchild} \leftarrow Q_{tail}$;
4      $q_{st} \leftarrow$ MakeSubtree($q_{lchild}, q_{rchild}$) ;
5      Enqueue($Q_{db}, q_{st}$);
6      ProcessOffset($q_{st}, q_{lchild}, q_{rchild}$);
7      ProcessIdx($q_{st}, q_{lchild}, q_{rchild}$);
8      $\Delta \leftarrow \{Q_{tail}.prev.prev, Q_{tail}.prev\}$;
9      foreach $\delta_i$ in $\Delta$, $1 \leqslant i \leqslant ||\Delta||$ do
10         $q_{ptr} \leftarrow \delta_i$;
11         $q_{pop} \leftarrow$ DequeueSpPos($Q_{db}, q_{ptr}$);
12         if $q_{pop}.is\_written = FALSE$ then
13             if $q_{pop}.level = 0$ and $q_{pop}.index$ is invalid and $\Phi_{fsn}$ has not been recorded then /* recording
                the offset of the first supplementary block */
14                 $\Phi_{fsn} \leftarrow f_p$;
15             end
16             FileWrite($f_p, q_{pop}$);
17         end
18         else
19             if $q_{pop}.level > 0$ and $q_{pop}.is\_written = TRUE$ then
20                 UpdateBlockIndexInfo($f_p, q_{pop}$);
21             end
22         end
23     end
24     if $Q_{tail}.is\_written = FALSE$ then
25         $Q_{tail}.is\_written = TRUE$;
26         FileWrite($f_p, Q_{tail}$);
27     end
28 end
```

Table 4: Symbols used in figure 5

Symbols	Comments
Q	the queue used in building an MHT file
Q_{hdr} & Q_{tail}	the queue's header and tail
Hdr	the MHT file header
q_x^l	an arbitrary queue node with level l
q_x^{l*}	an arbitrary queue node with level l which has been written into the MHT file
$q_{st_{xy}}^l$	the sub-tree root node in Q with level l, which is created by combining q_x^{l-1} and q_y^{l-1}
b_x^l	referring to an in-MHT-file block corresponding to q_x^l
f_{MHT}	the MHT file
`dq`	the dequeuing operation (`Dequeue`)
`wrt`	writing the node into the MHT file without dequeuing it
$\otimes$ in red	releasing the dequeued nodes

Figure 5 shows an example that how the queue and the MHT file varies when processing a data set with 4 elements (table 4 illustrates the symbols used in this figure). We can find that once two nodes with the same levels have been combined into a sub-tree, both of the nodes are written into the MHT file and dequeued, which refer to "useless nodes". Then, the sub-tree root is written into without being dequeued, which is further used to build another sub-tree with a higher level. For example, in sub-figure (1), $q_{st_{12}}^1$ is created by combining q_1^0 and q_2^0, and both of q_1^0 and q_2^0 are dequeued written into the MHT file, but the sub-tree root $q_{st_{12}}^1$ is just written into the MHT file and remained in the queue (shown as sub-figure (2), $q_{st_{12}}^{1*}$). Here, q_1^0 and q_2^0 are "useless nodes". In sub-figure (3), $q_{st_{14}}^2$ is created by combining $q_{st_{12}}^{1*}$ and $q_{st_{34}}^{1*}$, but both of $q_{st_{12}}^{1*}$ and $q_{st_{34}}^{1*}$ are dequeued and released since they have been already written into the MHT file. Note that, since $q_{st_{12}}^{1*}$ is an interior node, when it is first written into the MHT file, it's *parent_idx* and *parent_os* are not determined till $q_{st_{14}}^2$ has been written into the file. Hence, $b_{st_{12}}^1$ has to be updated by function `UpdateBlockIndexInfo`. In sub-figure (4), the only node remained in Q is the root node ($q_{st_{14}}^{2*}$), which has been written into the MHT file. Hence, it just needs to be dequeued and released.

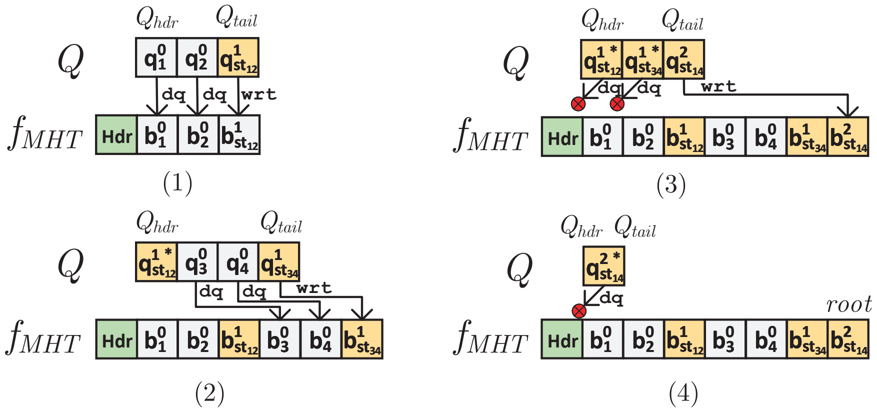


Figure 5: An example of building an MHT with memory-saving policy.

Theorem 8 indicates that the maximum number of nodes in the queue during building the MHT file can be denoted as $l + 3$ before dequeuing happens, in which l denotes the level of the queue's header node. Similarly, when the size of the MHT file block is 70B, the queue size (S_Q) variation

Table 5: A comparison of two methods of building an MHT file on how the queue's size varies with level l.

l	1	2	4	6	8	10	12	14	16	18
S_Q (KB) in MMCS version	0.34	0.62	2.26	8.82	35.07	140.07	560.07	2.24×10^3	8.96×10^3	3.58×10^4
S_Q (KB) in MMSV version	0.27	0.34	0.48	0.62	0.75	0.89	1.03	1.16	1.30	1.44

with l can be shown as table 5. Compared with the memory-consuming (MMCS) version, the memory-saving version (MMSV) only takes a very small portion of memory (e.g., when $l = 18$, S_Q in MMSV version is only about 1.44KB, but in MMCS version, S_Q is about 34.96MB!), which is very suitable for the embedded systems.

B. Searching A Block in MHT File

Algorithm 8: $SearchBlockByIndex(f_p, index)$: searching the corresponding block in MHT file by the given index

```
Input: $f_p$: MHT file pointer
       $index$: the given index
Output: The offset of the searched block
1  Moving $f_p$ to $f_{hdr}$;
2  Extracting root block offset ($\Phi_{rn}$) from $f_p$;
3  Moving $f_p$ to $\Phi_{rn}$;
4  while $f_p.level > 0$ do
5      if $index \leq f_p.index$ then
6          Moving $f_p$ to $f_p.lchild\_os$;
7      else
8          Moving $f_p$ to $f_p.rchild\_os$;
9      end
10 end
11 if $f_p.index = index$ then
12     return the block pointed by $f_p$; // Matched block found
13 end
14 return null ; // There is no matched block
```

Whether it is integrity verification or block update, searching a leaf block with the given index is one of the fundamental operations on an MHT file. Algorithm 8 gives an implementation based on binary searching. Different from handling a queue in the memory, the file pointer (f_p) is the only handler when dealing with the MHT file, which indicates the offset of the beginning of a block. Since searching starts from the root block, the root block offset Φ_{rn} will be first extract from the MHT file header and f_p will be moved to Φ_{rn}. Here, f_p is not only a file pointer but also the pointer to the current block, by which each attribute of the block can be visited via $f_p + os_x^{attr}$ (os_x^{attr} indicates the offset of attribute x from the block beginning). In the main loop, f_p will be moved to the left or right child blocks according to the comparison between the given index and that of the current block pointed by f_p till a leaf block is reached. Finally, if $f_p.index = index$, the matched block is found, otherwise, there is no such block with the given index.

C. Data Integrity Verification

Algorithm 9 shows the implementation of data integrity verification function `DataIntVerify` which has four parameters: the file pointer (f_p), the *index* and digest (h_d) of the data to be verified, and the MHT root hash used as the standard value. Firstly, the corresponding data block will be located based on the specified *index*. Then, the hash value of each interior node in the path of the Merkle proof will be re-computed till a new root hash (h) is generated. Finally, h will be compared

Algorithm 9: $DataIntVerify(f_p, index, h_d, h_r^*)$: verifying the integrity of data with the specified index.

```
Input: $f_p$: MHT file pointer
       $index$: the index of the specified data
       $h_d$: the specified data
       $h_r^*$: the pre-stored root hash
Output: $True$, if the verification succeeds, otherwise, $False$ will be returned.
1  if SearchBlockByIndex($f_p, index$) $\neq null$ then          // Now $f_p$ points to the matched data block
2      $h = h_d$;
3      while $f_p \neq root\ block$ do
4          $idx \leftarrow f_p.index$;
5          Moving $f_p$ to $f_p.parent\_os$;
6          if $idx = f_p.lchild\_idx$ then
7              Moving $f_p$ to $f_p.rchild\_os$;
8          else
9              Moving $f_p$ to $f_p.lchild\_os$;
10         end
11         $h \leftarrow$ SHA256($f_p.hash || h$);
12         Moving $f_p$ to $f_p.parent\_os$;
13     end
14     if $h = h_r^*$ then
15         return $True$;
16     else
17         return $False$;
18     end
19 end
```

with h_r^* to determine whether the data has been tampered with. Note that to re-compute the hash values from the data block to the root block (the path of the Merkle proof), the brother of the current block pointed by f_p needs to be located in each iteration of the loop (referring to line 5-10).

D. Updating A Block in MHT File

Algorithm 10: $UpdateBlockByIndex(f_p, index, d)$: updating the corresponding block in MHT file with new data d according to given index

```
Input: $f_p$: MHT file pointer
       $index$: the given index
       $d$: the new data
Result: Finishing updating the block and updating the relevant Merkel proof
1  if SearchBlockByIndex($f_p, index$) $fails$ then
2      return failed to update block;
3  end
4  $blk \leftarrow$ ReadBlock($f_p$);
5  $blk.hash \leftarrow$ SHA256($d$);
6  FileWrite($f_p, blk$);
7  UpdateMerkleProof($blk, f_p$);
```

When a piece of data has been modified, the hash values of the nodes located in the corresponding Merkle proof need to be updated as well. Algorithm 10 shows the internals of function `UpdateBlockByIndex`, in which the hash values of the relevant blocks will be updated when a data block is modified. Firstly, the specified data block can be located according to the given *index* via invoking function `SearchBlockByIndex` (line 1-3). Then, the data block with *index* will be updated with new data hash (`SHA256`(d)) (line 4-6). After that, the hash of each node in the path

(Merkle Proof path) from d to the MHT root will be updated, which is finished by the function `UpdateMerkleProof` whose implementation is detailed in algorithm 11. Shown as algorithm 11, to re-compute and update the hash values of these in-path nodes, the brother of the current block pointed by f_p needs to be located in each iteration of the main loop, so that their parent's hash can be re-computed (line 2-9). Note that, to facilitate in-block hash-update, the block pointed by f_p is read into a buffer blk via function `ReadBlock`, which will be written to the file after hash-update via function `FileWrite`.

Algorithm 11: $UpdateMerkleProof(blk, f_p)$: Updating each node in the Merkle Proof path.

```
Input: blk: a buffer preserving an MHT block
       f_p: the MHT file pointer
Result: The hash of each node (block) in the Merkle Proof path will be updated.
1  while $blk \neq$ root block do
2      $idx \leftarrow blk.index$;
3      $h \leftarrow blk.hash$;
4      Moving $f_p$ to $blk.parent\_os$;
5      $blk \leftarrow$ ReadBlock($f_p$) if $idx = blk.lchild\_idx$ then
6          Moving $f_p$ to $blk.rchild\_os$;
7      else
8          Moving $f_p$ to $blk.lchild\_os$;
9      end
10     $blk \leftarrow$ ReadBlock($f_p$);
11     $h^* \leftarrow$ SHA256($h||blk.hash$);
12     Moving $f_p$ to $blk.parent\_os$;
13     $blk \leftarrow$ ReadBlock($f_p$);
14     $blk.hash \leftarrow h^*$;
15     FileWrite($f_p, blk$);
16     Moving $f_p$ to $blk.parent\_os$;
17     $blk \leftarrow$ ReadBlock($f_p$);
18 end
```

E. Inserting A Block in MHT File

When the new pieces of data need to be added after the establishment of the MHT file, the data blocks corresponding to the new added data also need to be created and inserted into the MHT file. Thus, the key challenge is how to insert the new data blocks to the created MHT file. To address this issue, two cases on block-insertion are discussed below:

1) The special case: supposing d' denotes the new added data, we have $d'.index > d_{rm}.index$, in which d_{rm} denotes the right-most leaf node of the MHT, which has the maximum index in the tree. In this case, the new added blocks can be only appended to the file instead of moving the original in-file data blocks. Algorithm 12 shows the implementation of the block insertion for the special case. Since the supplementary blocks can be used to accommodate the new added blocks, in the following, the insertion process will be detailed based on the amount of the supplementary blocks.

A) *A valid Φ_{fsn}*: there are supplementary blocks in the MHT file, and the first one can be replaced by the new added blocks (line 3-6) via invoking function `UpdateBlockByIndex`. Note that, after finishing update, Φ_{fsn} must be updated with offset of the next supplementary block.

B) *An invalid Φ_{fsn}*: it implies that the amount of the original data pieces used to create the MHT file satisfies $\log_2 ||\boldsymbol{D}|| \in \{0, \mathbb{Z}^+\}$, which means the MHT needs to be expanded with

Algorithm 12: *InsertNewBlock*(f_p, idx, d) : (*A Special Version*) inserting a new block with index idx into the MHT file, which is related to the new data (d)

```
Input: $f_p$: MHT file pointer
       $idx$: index of the new data
       $d$: the new data
Result: Finishing inserting the new block and updating the relevant Merkle proof
1  Moving $f_p$ to $f_{hdr}$;
2  Extracting the first supplementary block offset ($\Phi_{fsn}$) from $f_p$;
3  if $\Phi_{fsn}$ is valid then
4      Moving $f_p$ to $\Phi_{fsn}$;
5      UpdateBlockByIndex($f_p, idx, d$);
6      Updating $\Phi_{fsn}$ in $f_{hdr}$;
7  else
8      Moving $f_p$ to $f_{hdr}$;
9      Extracting the root block offset ($\Phi_{rn}$) from $f_p$;
10     Moving $f_p$ to $\Phi_{rn}$;
11     $root_{orig} \leftarrow$ restoring the current root node structure from $f_p$;
12     Creating a new queue $Q'$;
13     Enqueue($Q', root$);
14     Moving $f_p$ to $\Phi_{rn}$;
15     SupplementRCST ($Q'$,$f_p$);
16     if $||Q'|| = 1$ then
17         $p_{pop} \leftarrow$ Dequeue($Q'$);
18         $\Phi_{rn} \leftarrow f_p$;
19         FileWrite ($f_p$, $q_{pop}$);
20     end
21     Moving $f_p$ to $\Phi_{fsn}$;
22     UpdateBlockByIndex($f_p, idx, d$);
23     Updating $\Phi_{rn}$ and $\Phi_{fsn}$ in $f_{hdr}$;
24 end
```

new supplementary blocks to provide space for the new blocks to be inserted. Thus, there will be additional $||\boldsymbol{D}||$ supplementary blocks added to the MHT file, and the total amount of the blocks becomes $2 \cdot ||\boldsymbol{D}||$ which also satisfies $\log_2(2 \cdot ||\boldsymbol{D}||) \in \mathbb{Z}^+$ (the expanded MHT is still a complete binary tree). Line 8-20 in algorithm 12 shows the process of the MHT file expansion, which is similar to the stage SRCST in building an MHT file (line 14-16 of algorithm 1). After finishing expanding the MHT file, the new block to be inserted will be stored in the first supplementary block via calling `UpdateBlockByIndex`. Finally, Φ_{fsn} and Φ_{rn} in the MHT file header (f_{hdr}) will be updated.

2) The common case: the new data d' can be inserted at any proper place according to the characteristics of the BST (Binary Search Tree). Thus, the key challenge is how to deal with the nodes (blocks) whose indices are larger than $d'.index$ while inserting d' into the MHT file. Figure 6 shows such an example. The MHT file is created from four data pieces with indices 1, 2, 3 and 5, and Hdr denotes the file header, d^l_x denotes the data block has an index of x and is at level l. When inserting a new node d^0_4, it should replace d^0_5, and node d^0_5 and the subsequent nodes need to be moved to the right for a block size. However, the new node insertion will make the MHT be an incomplete binary tree, which needs to be extended.

Algorithm 13 shows a detailed implementation for the common case of node insertion. Firstly, the new data d' to be inserted will be searched in the MHT file with its index $d'.index$ by calling function `SearchBlockByIndex`. If there exists a matched block, it will be updated with the new data d' (including the index and the hash value), and the Merkle Proof from the block to the root

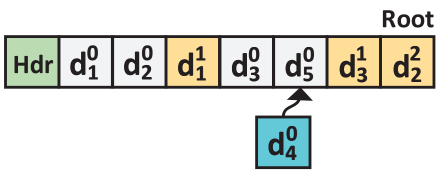


Figure 6: An example of a common case for node insertion.

will be updated as well (line 1-6). Note that the function `SearchBlockByIndex` can not only return the position (offset os_{ins}) of the matched block if it exists, but also return the appropriate position where d' should be inserted as a new block. Hence, when no matched block exists, the block pointed by os_{ins} should be replaced by the new data d'.

Line 8-38 of algorithm 13 shows the insertion process of the new data d'. Firstly, we create a copy of the MHT file (denoted as f_{MHT}^{dup}, line 9) handled by f_p^* for restoring the blocks whose indices are larger than $d'.index$ after finishing new block insertion. In contrast, f_p is used to handle the original MHT file (denoted as f_{MHT}^{orig}), into which the new data will be inserted. Then, if $\Phi_{fsn} = 0$, it implies that there is no extra block to accommodate d', and f_{MHT}^{orig} needs to be extended via function `ExtendMHT`, which can create a new sub-tree (denoted as T_{st}^{ext}) with the same number of blocks (all the blocks are supplementary blocks) as the original MHT file. T_{st}^{ext} will be appended to f_{MHT}^{orig} to form the final extended MHT file (denoted as f_{MHT}^{ext}). Next, Φ_{fsn} and Φ_{rn} will be updated.

After that, a new block is created based on d', which will be written into the block indicated by offset os_{ins}. Then, the Merkle Proof from the current block to the root will be updated by function `UpdateMerkleProofEx`. As an extended version of `UpdateMerkleProof`, in `UpdateMerkleProofEx`, both hash value and index of each interior node in the Merkle Proof path will be updated. In the main loop (line 21-37), each leaf node from offset os_{rd} in f_{MHT}^{dup} will be traversed and be copied to the position indicated by os_{wrt} in f_{MHT}^{orig}. After finishing each copy operation, the Merkle Proof related to the copied node (in f_{MHT}^{orig}) needs to be updated (line 33). Function `FindNextLeafNode`(f_p, os) can return the first leaf node from offset os, and will stop when encountering a supplementary node or the root node. Finally, when the loop ends, the MHT file duplicate f_{MHT}^{dup} will be removed.

3 Evaluations

In this section, the performance of the operations on the fbMHT has been evaluated, including *Building an MHT file*, *Data integrity verification*, *MHT block updating* and *MHT block insertion*. We mainly evaluate the time consumption of conducting these operations on a PC and a Raspberry PI 4B development board, respectively, and compare their evaluation results. For each experiment in the following, the result data are the average values after 100 rounds of experiments. Table 6 gives the configurations of our performance evaluation platforms.

3.1 Building An MHT File

In this part, we conduct a performance evaluation on building an MHT file with both MMCS and MMSV algorithms on the PC and RPI-4B platforms, respectively. Figure 7 shows the evaluation results. To facilitate our discussion, we let $N_{dblk} = ||\boldsymbol{D}||$ denote number of data blocks, and let t_{MMSV}^{x} and t_{MMCS}^{x} denote the time consumption of these two algorithms to run on platform x , respectively. In addition, to provide distinct data presentation, the y-axis is logarithmic coordinate.

Algorithm 13: $InsertNewBlockCommonVer(f_p, idx, d)$: (A $Common$ $Version$) inserting a new block with index idx and data d into the MHT file

```
Input: $f_p$: MHT file pointer
       $idx$: index of the new data
       $d$: the new data
Result: Finishing inserting the new block and updating the relevant nodes in Merkle Proof path
1  $f_p \leftarrow$ SearchBlockByIndex($f_p, idx$) ;
   /* With SearchBlockByIndex, $f_p$ can be moved to the location where $d$ should be
      inserted/updated.  The offset of the location will be recorded in $os_{ins}$.       */
2  $os_{ins} \leftarrow f_p$ ;
3  if $f_p$ is valid then
4      UpdateBlockByIndex($f_p, idx, d$);
5      return $f_p$;
6  end
7  else
8      ResetFp($f_p$);
9      $f_p^* \leftarrow$ FileCopy($f_p$);
10     Getting $f_{hdr}$ from $f_p$;
11     Getting $\Phi_{fsn}$ from $f_{hdr}$;
12     if $\Phi_{fsn} = 0$ then
13         ResetFp($f_p$) ;
14         ExtendMHT($f_p$);
15     end
16     $blk_{new} \leftarrow$ building a new MHT block with $idx$ and $d$;
17     Moving $f_p$ to $os_{ins}$;
18     FileWrite($f_p, blk_{new}$);
19     UpdateMerkleProofEx($blk_{new}, f_p$);
20     $os_{wrt} \leftarrow os_{rd} \leftarrow os_{ins}$;
21     while $os_{wrt} < \Phi_{rn}$ and $os_{rd} < \Phi_{rn}$ do
22         Moving $f_p$ to $os_{wrt}$;
23         $blk \leftarrow$ ReadBlock($f_p$);
24         $blk \leftarrow$ FindNextLeafNode($f_p, os_{wrt}$);
25         CheckOffset($os_{wrt}, \Phi_{rn}$);
26         Moving $f_p^*$ to $os_{rd}$;
27         $blk \leftarrow$ FindNextLeafNode($f_p^*, os_{rd}$);
28         CheckOffset($os_{rd}, \Phi_{rn}$);
29         if $blk$ is a supplementary node then
30             break;
31         end
32         FileWrite($f_p, blk$);
33         UpdateMerkleProofEx($blk, f_p$);
34         $os_{wrt} \leftarrow os_{wrt} + s_{blk}$;
35         $os_{rd} \leftarrow os_{rd} + s_{blk}$;
36         ResetFp($f_p$);
37         Updating $\Phi_{fsn}$ in $f_{hdr}$ pointed by $f_p$;
38     end
39 end
```

Obviously, regardless of the number of data blocks (the x-axis) and the experimental platform, MMSV algorithm takes much more time on building the MHT file. For example, when $N_{dblk} = 128$, $t^{PC}_{MMSV} = 874.23ms$ and $t^{RPI4B}_{MMSV} = 1106.87ms$. However, for MMCS algorithm, $t^{PC}_{MMCS} = 1.52ms$ and $t^{RPI4B}_{MMCS} = 3.87ms$. When $N_{dblk} = 524288$, whether on PC or RPI platforms, the MMSV algorithm is no longer practical (t^{PC}_{MMSV} is nearly 6585s and t^{RPI4B}_{MMSV} is about 5244s), whereas

Table 6: The configurations of performance evaluation platforms

Configurations \ Platforms	PC (Lenovo Thinkpad T470s)	Raspberry PI 4B (RPI-4B)
CPU	Intel Core i7 7500U @2.7GHz (Dual cores, four threads)	Broadcom BCM2711 @1.5GHz (ARM Cortex-A72, Four cores)
RAM	DDR4 16GB	LPDDR4 4GB
External Storage	SAMSUNG MZVLB512HAJQ-000L7, 512GB	Sandisk TF-card, 16GB
OS	Ubuntu 20.04 64-bit with kernel 5.13	Raspbian 11 with kernel 5.15

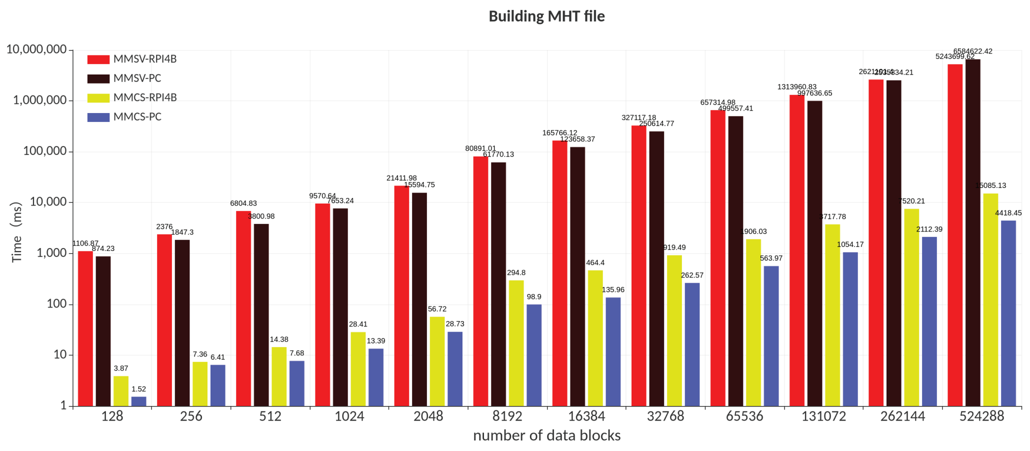


Figure 7: Time consumption on building MHT files with different number of data blocks.

MMCS algorithm only takes 4.4s and 5.1s on PC and RPI-4B platforms, respectively. Hence, although the memory occupation can be reduced by using MMSV algorithm during MHT file construction, the frequent access to the external storage (TF-card, SSD or harddisk) greatly affect the performance of the algorithm. Thus, for the memory constrained devices, an MHT file can be built by using MMSV algorithm from a new created data set with a small amount of data blocks. If the data set already contains a large amount of blocks, the MHT file can only be built by MMCS algorithm on a platform with enough memory and moved to the original device.

3.2 Data Integrity Verification

In this part, the time consumption of verifying the integrity of a specified data block is evaluated on both PC and RPI-4B platforms. To facilitate our discussion, let t_{IV}^{x} denote the time consumption of data integrity verification on platform x. Figure 8 shows the evaluation results. It is obvious that the verification algorithm takes only about $125\mu s$ on Raspberry Pi when there are 524288 leaf nodes in the MHT file. This is because the height of the Merkle Proof is only $\log_2 524288 = 19$ from the leaf node to the root. Due to the higher performance hardware, integrity verification on the PC takes less time than on the RPI-4B. In fact, although the Raspberry Pi consumes more time, the time difference between the Raspberry Pi and the PC is still within an acceptable range and does not affect the system's usability.

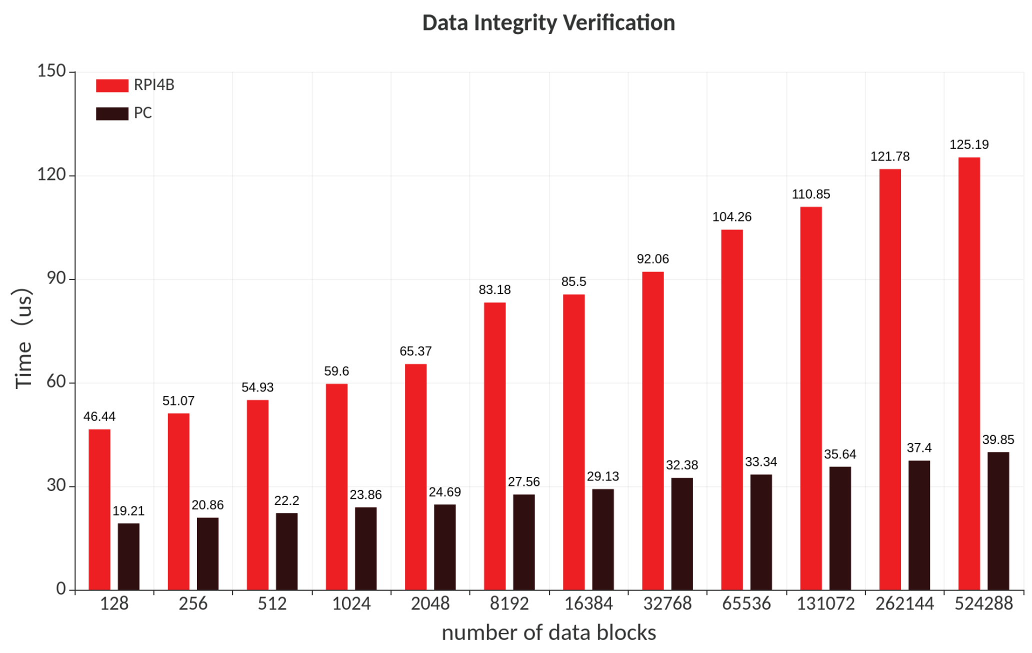


Figure 8: Time consumption on data integrity verification under the different number of blocks within MHT files.

3.3 MHT Block Update

In block update performance evaluation, we randomly select a leaf node of the MHT to be updated in each test round, and the node's hash will be replaced with a new one generated from a new data block. Figure 9 shows the average time consumption of updating a leaf node of the MHT file under different MHT scales. Similarly, to facilitate our discussion, let t_{BU}^{x} denote the time consumption of updating a leaf node on platform x. Obviously, with the increase of the number of leaf nodes, the corresponding t_{BU}^{x} also increases. However, even if $N_{dblk} = 524288$, $t_{BU}^{PC} = 197.42\mu s$ and $t_{BU}^{RPI4B} = 475.99\mu s$, which are acceptable in practical applications. This is because the node-searching operation is involved in data block update, which travels from the root to the leaf node to be updated along with the Merkle Proof and has the time complexity of $O(\log_2 N_{dblk})$. Note that the same node-searching is also included in Data Integrity Verification. However, since there are update operations, including re-computing hash value and re-writing the file block, in the Block Update procedure, we have $t_{BU}^{x} > t_{IV}^{x}$.

3.4 MHT Block Insertion

In this section, we evaluate the performance of inserting a new leaf node into the MHT file with two different algorithms. One deals with the particular insertion scenario (denoted as INS_{sp}), whereas the other is for the common case (denoted as INS_{cmn}). Similarly, to facilitate our discussion, we let t_{INSSP}^{x} and t_{INSCMN}^{x} denote the time consumption of executing insertion algorithms on platform x in both special and common versions.

A) The Special Version

According to section 2.2-E, for INS_{sp}, the new node to be inserted has a larger index than the right-most leaf node of the current MHT file. Hence, the new node can be directly inserted into the MHT file by replacing the first spare supplementary node. If no proper supplementary node is

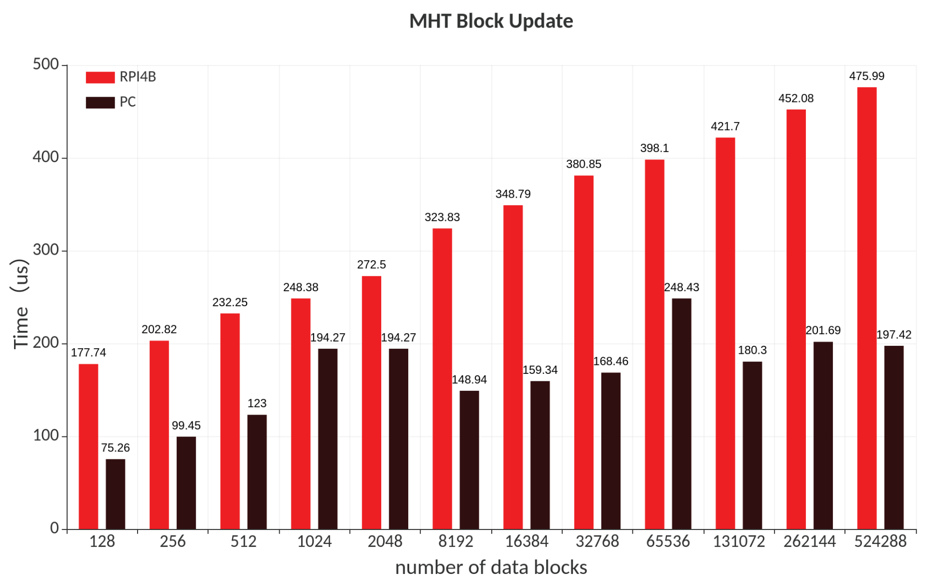


Figure 9: Time consumption on block update under the different number of blocks within MHT files.

available, the MHT file will be extended.

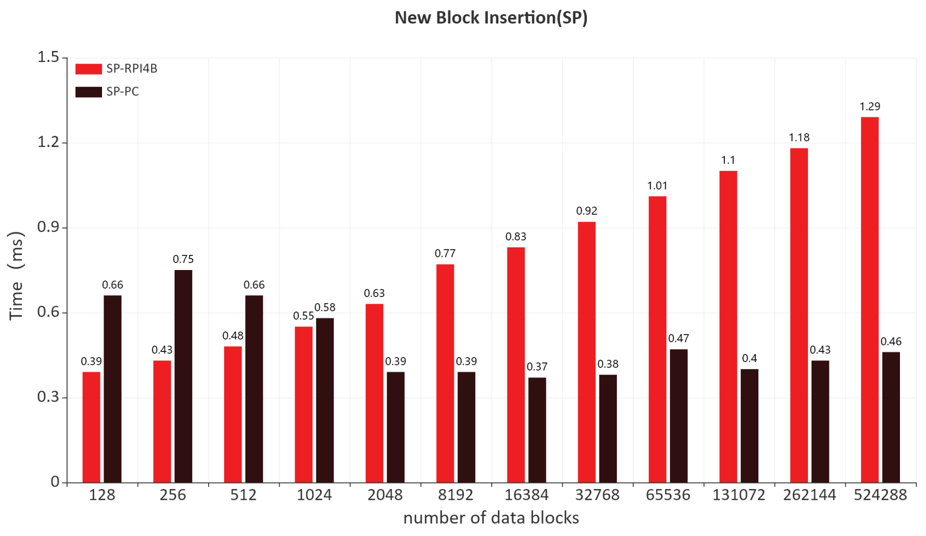


Figure 10: Time consumption on new block insertion under the different number of blocks within MHT files (the special version).

Figure 10 shows the evaluation results of node insertion with INS_{sp}. Obviously, when $N_{dblk} = 128, 256, 512$ and 1024, the algorithm which runs on PC spends more time than the other points. This is because MHT extension is involved in these processes to produce extra supplementary nodes for the new inserted nodes. For Raspberry Pi, increasing the data blocks brings slight growth in the time consumption of inserting a new node. For example, when $N_{dblk} = 524288$, we have $t^{RPI4B}_{INSSP} = 1.29ms$, which is acceptable in the practical applications.

B) The Common Version

For the node insertion algorithm in the common version, it can be obtained that $t_{INSCMN} > t_{INSSP}$ regardless of adopting PC or Raspberry Pi (figure 11). Particularly, when $N_{dblk} = 524288$, we have $t^{PC}_{INSCMN} = 74286.6ms$ and $t^{RPI4B}_{INSCMN} = 231750ms$. This indicates that inserting a new node to the MHT file randomly takes nearly 4 minutes to finish this process on the Raspberry Pi! Hence, for application scenarios that require real-time performance, the common version algorithm is not applicable. In fact, the significant drop in performance is mainly caused by data copy on the external storage during the node insertion. Similar to building an MHT file with MMSV algorithm, frequent file I/O can greatly affect the performance.

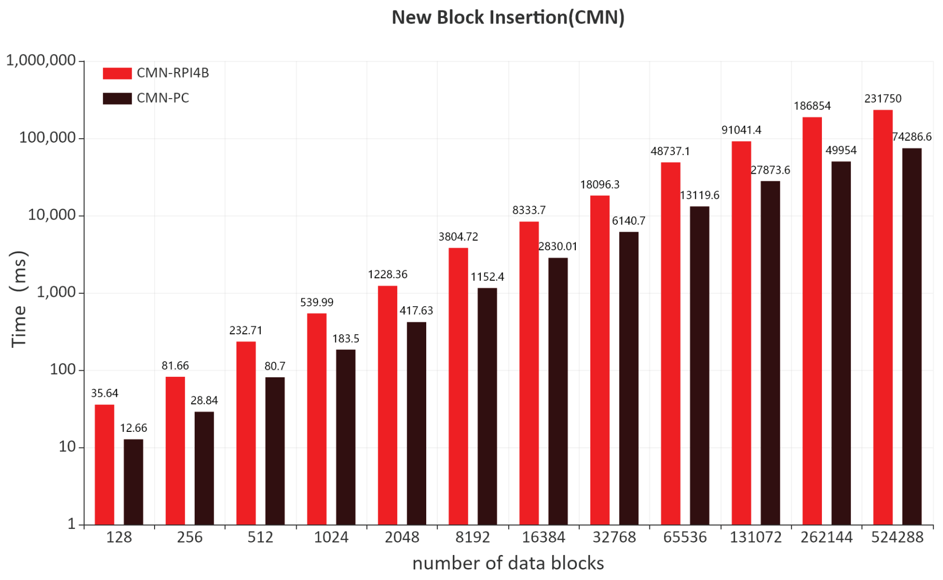


Figure 11: Time consumption on new block insertion under the different number of blocks within MHT files (the common version).

Therefore, to address the performance issue, the common version insertion algorithm needs to be optimized in future work. For example, we should avoid copying a large amount of data in data insertion.

4 Conclusion

In this report, we propose fbMHT, a file-based MHT that implements an MHT tree in a sequential file. With fbMHT, a memory-constrained system, such as some embedded system, can handle the MHT via file oriented operations instead of maintaining the whole tree in the memory. In an MHT file, we re-organize the 2-D tree structure into a sequential one and design a set of operations on the MHT file, including building an MHT file, node searching, data integrity verification, data block update and data block insertion. Memory-occupation analysis proves that fbMHT can achieve a limited memory footprint and the performance evaluation shows the feasibility, usability, and limitations.

References

[1] Andrews, G. E. *Number Theory: Chapter 1: The Basis Representation.* Dover Publications, Inc.

A Proofs of Theorems

Theorem 1 *For a data set with* $N(N > 0 \wedge N \in \mathbb{Z}^+)$ *data pieces as leaf nodes, if* $\log_2(N) > 0 \wedge \log_2(N) \notin \mathbb{Z}^+$*, it lacks of* $2^{\lceil \log_2(N) \rceil} - N$ *data pieces to build a complete MHT.*

Proof 1 *Since* $N(N > 0 \wedge N \in \mathbb{Z}^+)$ *and* $\log_2(N) > 0 \wedge \log_2(N) \notin \mathbb{Z}^+$*, we assume that it lacks of* $x(x \in \mathbb{Z}^+)$ *leaf nodes to build a complete binary tree, we have* $k = \log_2(N + x)$ *and* $k \in \mathbb{Z}^+$*. That means* $N + x$ *is the minimum integer larger than* N *to build a complete binary tree, and* k *can be denoted as* $k = \lceil \log_2(N) \rceil$ *as well. We have*

$$\begin{aligned} k &= \log_2(N + x) \\ \lceil \log_2(N) \rceil &= \log_2(N + x) \\ x &= 2^{\lceil \log_2(N) \rceil} - N \end{aligned}$$

■

Theorem 2 *For an MHT constructed from left to right with* $N(N \geqslant 0 \wedge N \in \mathbb{Z}^+)$ *data pieces as the leaves, the left sub-tree of the root node is always a complete binary tree.*

Proof 2 *For any integer* $N(N \geqslant 0 \wedge N \in \mathbb{Z}^+)$*, there are two cases to be discussed:*
1) If $\log_2(N) \in \{0, \mathbb{Z}^+\}$*,* N *nodes are just sufficient to construct a complete binary tree. Note that if* $\log_2(N) = 0$*,* $N = 1$*, which implies that the leaf node is also the MHT root.*
2) If $\log_2(N) > 0 \wedge \log_2(N) \notin \{0, \mathbb{Z}^+\}$*,* N *nodes are insufficient to construct a complete binary tree. However, to build a complete left sub-tree needs at least* $2^{\lceil \log_2(N) \rceil}/2$ *nodes. If* $N > 2^{\lceil \log_2(N) \rceil}/2$ *can hold, the theorem is proved.*

$$\begin{aligned} &\frac{2^{\lceil \log_2(N) \rceil}}{2} \div N = \frac{2^{\lceil \log_2(N) \rceil - 1}}{N} = \frac{2^{\lceil \log_2(N) \rceil - 1}}{2^{\log_2(N)}} = 2^{\lceil \log_2(N) \rceil - 1 - \log_2(N)} \\ &\because 0 < \lceil \log_2(N) \rceil - \log_2(N) < 1 \\ &\therefore \lceil \log_2(N) \rceil - \log_2(N) - 1 < 0 \\ &\therefore 2^{\lceil \log_2(N) \rceil - 1 - \log_2(N)} < 1 \\ &\therefore N > 2^{\lceil \log_2(N) \rceil - 1} \end{aligned} \tag{2}$$

Therefore, with N *nodes, a complete left sub-tree can be constructed.* ■

Theorem 3 *Supposing there are* n *leaf nodes left when stage BLCST ends, there are*

$$\zeta = \sum_{i=0}^{k-1} b_{k-i}(2^{k-i} - 1)$$

nodes (including leaf and interior nodes) in total left in the queue except the header node. Here, n *is expressed by a* k*-bit binary number* $b_k b_{k-1} b_{k-1} \ldots b_1$ $(b_i, (1 \leq i \leq k)$ *can be 0 or 1).*

Proof 3 *Since the process of BLCST is to build a complete binary trees till there are insufficient leaf nodes, if the nodes left in the queue are more than 1, that means* n *leaf nodes cannot create a complete binary tree but a series of smaller complete binary trees. Considering only an integer power of 2 nodes are sufficient to build a complete binary tree, thus, the problem above can be converted into another form that "how to express* n *with sums of a series of powers of 2?". With Basis*

*Representation theorem (Theorem 1-3 in chapter 1 of "*Number Theory*"[1]), n can be expressed by a k-bit ($k \geq 1$) binary number $b_k b_{k-1} b_{k-1} \dots b_1$ ($b_i, (1 \leq i \leq k)$ can be 0 or 1). n can be expressed as*

$$n = \sum_{i=0}^{k-1} b_{k-i} 2^{k-1-i} = b_k 2^{k-1} + b_{k-1} 2^{k-2} + b_{k-2} 2^{k-3} + \dots + b_1 2^0$$

Considering each complete binary tree has $2m - 1$ nodes (including interior and leaf nodes. m denotes the number of leaf nodes), thus, except the queue's header, the number of the nodes (including leaf and interior nodes) remained in the queue can be expressed as

$$\begin{aligned} \zeta &= \sum_{i=0}^{k-1} b_{k-i}(2 \cdot 2^{k-1-i} - 1) = \sum_{i=0}^{k-1} b_{k-i}(2^{k-i} - 1) \\ &= b_k(2^k - 1) + b_{k-1}(2^{k-1} - 1) + b_{k-2}(2^{k-2} - 1) + \dots + b_1(2^1 - 1) \end{aligned} \quad (3)$$

■

Theorem 4 *After finishing stages BLCST and SRCST, there is only one node in the queue, which is the MHT root.*

Proof 4 *The end condition of stage SRCST is the MHT root node, whose level is 1 larger than the queue's header, has been created, thus, all the nodes before the root node will be dequeued when SRCST ends. In this case, only the root node remains in the queue, which is also the queue's header.*

Theorem 5 *Algorithm 4 can make the MHT support binary search.*

Proof 5 *Let a triad $(q_l^k, q_r^k, q_{st}^{k+1})$ denote an arbitrary minimal sub-tree (denoted as T_{mst}), in which q_l^k and q_r^k denotes the left and right children with level k and q_{st}^{k+1} denotes the sub-tree root with level $k+1$. We need to prove that for any T_{mst}, $q_l^k.index \leq q_{st}^{k+1}.index$ and $q_r^k.index > q_{st}^{k+1}.index$ hold.*

If q_l^k and q_r^k are leaf nodes ($k = 0$), according to the algorithm, $q_{st}^1.index \leftarrow q_l^0.index$ and $q_r^0.index$ is temporarily stored in $q_{st}^1.index$. Since all the leaf nodes are arranged in ascending order, we have $q_r^0.index > q_{st}^1.index = q_l^0.index$. Thus, the proposition is proved.

If q_l^k and q_r^k are interior nodes ($k > 0$), both of them are sub-tree roots of their child nodes. According to the algorithm, "root.tmp_idx ← rchild.tmp_idx" guarantees that the index of the right most leaf node of any sub-tree rooted at node q_x can be passed up during the construction of the MHT. Therefore, $q_l^k.tmp\_idx$ stores the index of its right most leaf node (denoted as q_m^0). Let $q_{m-(2^k-1)}^0$ denotes the left most leaf node of q_l^k (since the sub-tree rooted at q_l^k, denoted as $T_{q_l^k}$, is a complete binary tree, it has 2^k leaf nodes), we have:

$$q_m^0.index > q_{m-1}^0.index > q_{m-2}^0.index > \dots > q_{m-(2^k-1)}^0.index$$

'

$$\begin{aligned} &\because q_m^0.index \text{ is the largest one in } T_{q_l^k}, \\ &\therefore q_m^0.index > q_l^k.index. \\ &\because q_{st}^{k+1}.index \leftarrow q_l^k.tmp\_idx = q_m^0.index, \\ &\therefore q_{st}^{k+1}.index > q_l^k.index. \end{aligned}$$

For q_r^k, $q_r^k.index = q_{r_{lchild}}^{k-1}.index$ ($q_{r_{lchild}}^{k-1}$ is the left child of q_r^k), whereas $q_{r_{lchild}}^{k-1}.index$ is equal to the index of the right most leaf node (denoted as q_p^0) of the sub-tree rooted at $q_{r_{lchild}}^{k-1}$ (denoted as $T_{q_{r_{lchild}}^{k-1}}$), which satisfies $q_p^0.index > q_m^0.index$ since all the leaf nodes are arranged in ascending order (in fact, $T_{q_{r_{lchild}}^{k-1}}$ is also a complete binary tree, thus, there are $2^{k-1} - 1$ nodes between q_p^0 and q_m^0). Therefore, $q_r^k.index = q_p^0.index > q_{st}^{k+1}.index = q_m^0.index$, and the proposition is proved. ■

Theorem 6 *When building an MHT file, the maximum size of the queue in the current iteration is determined by the level of the queue's header.*

Proof 6 *When building an MHT, whether using algorithm 1 (the memory-consuming version, MMCS) or algorithm 6 (the memory-saving version, MMSV), a sufficient and necessary condition for performing the dequeue operation is that a new sub-tree is constructed, that is, a new node is enqueued with the same level as the header node. At this moment, the queue reaches the maximum size at this iteration.* ■

Theorem 7 *Assuming the queue's header is q_x^l, for algorithm 1, the maximum queue size in the current iteration is $2^{l+1} + 1$ (l denotes the node level, and $l = 0$ indicates the leaf nodes).*

Proof 7 *Without loss of generality, we assume that $\log_2 ||D|| \in \{\mathbb{Z}^+\}$. According to algorithm 1 and theorem 6, when another node q_y^l has been enqueued, q_x^l and q_y^l can be combined to construct a new sub-tree with root $q_{st_{xy}}^{l+1}$. At this moment, the queue reaches its maximum size in the current iteration. Since q_y^l is also a sub-tree root of a complete binary tree, it has $2_{l+1} - 1$ nodes. Thus, there are $2^{l+1} - 1 + 2 = 2^{l+1} + 1$ nodes in total (including q_x^l and $q_{st_{xy}}^{l+1}$). The proposition is proved.*
■

Theorem 8 *Assuming the queue's header is q_x^l, for algorithm 6, the maximum queue size in the current iteration is $l + 3$ (l denotes the node level, and $l = 0$ indicates the leaf nodes).*

Proof 8 *Without loss of generality, we assume that $\log_2 ||D|| \in \{\mathbb{Z}^+\}$. According to algorithm 6, once there is a new sub-tree constructed by combining two nodes with the same levels, both of the nodes will be dequeued (via `Dequeue` or `DequeueSpPos`) and written into the MHT file, whereas the sub-tree node will be written into the file without dequeuing. Thus, it can be obtained that if the queue header is q_x^l, the following nodes are $q_{a_1}^{l-1}$,$q_{a_2}^{l-2}$,..., $q_{a_{k-1}}^1$,$q_{a_k}^0$. Thus, the number of nodes in the queue is $1 + l$. Once $q_{a_{k+1}}^0$ has been enqueued, $q_{a_k}^0$ and $q_{a_{k+1}}^0$ can be combined to construct $q_{st_{a_{k,k+1}}}^1$, which can be then combined with $q_{a_{k-1}}^1$. Hence, before performing dequeue operation, the queue reaches its maximum size, $1 + l + 1 + 1 = l + 3$ (the three 1s refer to q_x^l, $q_{a_{k+1}}^0$ and $q_{st_{a_{k,k+1}}}^1$). The proposition is proved.* ■