HEX-BLOOM: An Alternative to the Merkle Tree

Ripon Patgiri, *Senior Member, IEEE*

Abstract—Merkle tree is applied in diverse applications, namely, Blockchain, smart grid, IoT, Biomedical, financial transactions, etc., to verify authenticity and integrity. Also, the Merkle tree is used in privacy-preserving computing. However, the Merkle tree is a computationally costly data structure. It uses cryptographic string hash functions to partially verify the data integrity and authenticity of a data block. However, the verification process creates unnecessary network traffic because it requires partial hash values to verify a particular block. Moreover, the performance of the Merkle tree also depends on the network latency. Therefore, it is not feasible for most of the applications. To address the above issue, we proposed an alternative model to replace the Merkle tree, called HEX-BLOOM, and it is implemented using hash, Exclusive-OR and Bloom Filter. Our proposed model does not depends on network latency for verification of data block's authenticity and integrity. HEX-BLOOM uses an approximation model, Bloom Filter. Moreover, it employs a deterministic model for final verification of the correctness. In this article, we show that our proposed model outperforms the state-of-the-art Merkle tree in every aspect.

Index Terms—Merkle tree, Blockchain, Bitcoin, verification, authentication, integrity, privacy, Hash, Security.

1 Introduction

Merkle tree [1] is widely used nowadays due to the diverse requirements of security. Recent developments suggest that Merkle tree is adapted in diverse research domains including privacy-preserving computation [2], Blockchain [2], [3], [4], cryptography [5], [6], agriculture [7], Healthcare [8], [9], financial transactions [10], Smart Grid [11], Cloud Computing [12], Big Data [13], Wireless networking [14]. Therefore, the Merkle tree is modified to enhances its performance. Jakobsson *et al.* [15] presents fractal Merkle tree to enhance the time and space. Similarly, M. Szydlo [16] enhances Jakobsson's fractal Merkle tree. Buchmann *et al.* [17] improves the Merkle tree. Moreover, We have already witnessed diverse variants of the Merkle tree [4], [12], [14], [18], [19]. It shows that the Merkle tree is adapted in diverse applications and modified Merkle tree as per the requirements of the applications. Therefore, the Merkle tree has met wider applications in the diverse domain which demands an alternative to the Merkle tree, which features low space consumption, fewer network accesses, and low time complexity.

Merkle tree is a time-consuming data structure that wastes computational resources significantly. It is used to verify data blocks' authenticity and integrity. It allows verification of the data block's authenticity and integrity after successfully downloading the data block using Merkle root. However, it requires a few hash values, but it does not require the entire Merkle tree. This process creates network traffic. Each block requires a few hash values, which require network access. Therefore, it degrades the performance of the Merkle tree as well as the network traffic. Moreover, the Merkle tree requires high memory to store entire hash values; however, a user does not require the whole Merkle tree. Also, verification of a particular block is costly. Therefore, we propose an alternative model of the Merkle tree to address the above issue. Our proposed model uses **H**ash, **Ex**clusive-OR and **Bloom** Filter, HEX-BLOOM for short. It is two-fold; first, LinkedHashX, and second, Bloom Filter. We construct a deterministic model called LinkedHashX to verify the entire process's correctness. LinkedHashX uses a cryptographic hash function and XOR operation to provide an alternative model to the Merkle tree. LinkedHashX performs a hash on all data blocks and merges the data blocks' hashes into a single data block using XOR operation to create LinkedHashX root. User or creator of LinkedHashX does not maintain the entire process; instead, LinkedHashX root is maintained for future use. A user needs to reconstruct the LinkedHashX root and compares it with the original root. Secondly, we use Bloom Filter to verify the block's authenticity and integrity, an approximation of data structures. All data blocks are inserted into Bloom Filter during the construction of the LinkedHashX. A user requires LinkedHashX root and Bloom Filter to download to verify a data block's authenticity, integrity, and correctness.

Our key contributions are outlined below-

- HEX-BLOOM uses Bloom Filter to verify a data block's authenticity and integrity in $O(k)$ time complexity for k distinct hash functions.
- The total verification time complexity for data authenticity and integrity is $O(k\mathcal{L})$.
- The construction cost of HEX-BLOOM is two-folded, firstly, the construction cost of Bloom Filter, and secondly, the construction cost of LinkedHashX. The construction cost of Bloom Filter and LinkedHashX are $O(k\mathcal{L})$ and $O(\mathcal{L})$, respectively for $\mathcal{L}$ data blocks.
- The extra space complexity of Bloom Filter is μ which is derived in Equation (12). The extra space complexity of LinkedHashX is $O(1)$.
- The total communication cost of our proposed model is $O(1)$.
- Moreover, the time complexity of insertion and deletion is $O(k)$.

• *Department of Computer Science and Engineering, National Institute of Technology Silchar, Cachar-788010, Assam, India.*
E-mail: ripon@cse.nits.ac.in, and URL: http://cs.nits.ac.in/rp/

2 MERKLE TREE

Most of the Merkle tree implementation is binary; however, we assume m-ary Merkle tree for generalization.

Definition 1. *The m-ary or m-way tree is a tree of order m where*

- *It is a rooted tree,*
- *Each node can have at most $(m-1)$ keys,*
- *Each node can have at most m children,*
- *Keys are not in ordered.*

Definition 2. *An m-ary Merkle tree has leaf nodes, internal node and Merkle root which are defined below-*

- *The direct hash value of data blocks are known as leaf nodes in the Merkle tree.*
- *The Merkle tree concatenates m hash values and hashes the concatenated hash value to form a single value is called a parent node or an internal node.*
- *Merkle root is a root node of the tree that contains a hash value of its siblings.*

2.1 Merkle Tree

Merkle tree is complex to construct and maintain. There are many issues in the Merkle tree, for instance, the insertion of a new node. Also, it takes extra space complexity. Therefore, we analyze the complexities with respect to the number of blocks. Let $\mathcal{L}$ be the number of blocks. These blocks are hashed using the SHA2 hash function, and these hash values are used to build the Merkle tree. Therefore, there are $\mathcal{L}$ leaf nodes in the tree. Theorem 1 shows that an m-ary Merkle tree has $\mathcal{I} = \frac{(\mathcal{L}-1)}{(m-1)}$ internal nodes. It shows that there is an overhead of $\mathcal{I}$ hash functions. The cryptographic hash functions are slower than the non-cryptographic hash functions. Its cryptographic hash functions impact the Merkle tree's performance, for instance, SHA2.

2.2 Construction

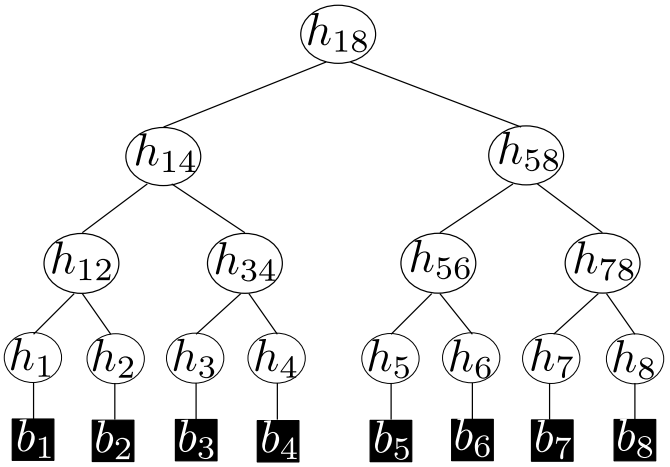

Fig. 1: Construction of the conventional binary Merkle Tree.

Figure 1 demonstrates the construction of the Merkle tree. Initially, all blocks are hashed using cryptographic string hash functions, for instance, SHA256. Then, the two consecutive hash values are concatenated, and the concatenated hash value is hashed using the same hash function to form their parents. Again, the same procedure is applied to the subsequent consecutive blocks. This process is rerun repeatedly until it becomes a single node, i.e., it repeats the process to get the Merkle root. Finally, the Merkle root is published publicly and can be distributed to peers. The creator of the Merkle tree maintains the tree, and the peers do not require the entire Merkle tree.

2.3 Verification

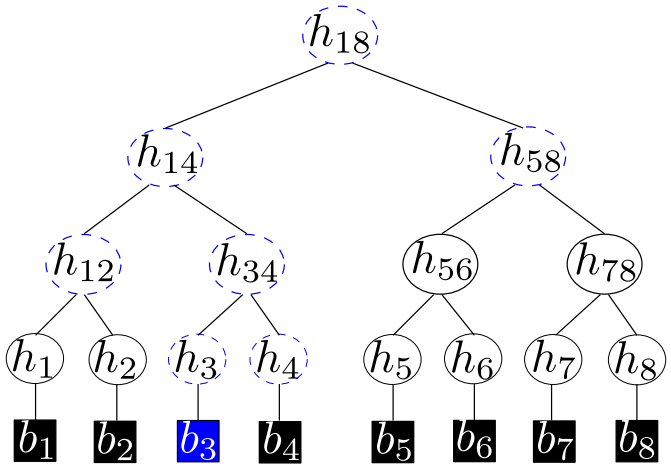

Fig. 2: Verification process of a particular block's authenticity in binary Merkle tree.

Figure 2 shows the process of verifying a particular block. Merkle tree verifies the authenticity of a specific block. For instance, a user has downloaded a block and needs to verify the block for its authenticity and integrity. In this case, the user does not require the entire Merkle to verify the block. It requires only a few hash values to verify the authenticity of the block, as shown in the dashed circle for block b_3 in Figure 2. Therefore, it features a partial verification process of a data block. Lemma 1 shows the total number of nodes in terms of the number of leaf nodes.

2.4 Analysis

Merkle tree is a computationally costly process. In this analysis, we analyze its time and space complexity. For generality, we assume an m-ary Merkle tree. Theorem 1 shows the total number of internal nodes for $\mathcal{L}$ blocks of data. Moreover, it shows the relation between the total number of leaf nodes and the internal nodes.

Theorem 1. *The total number of the internal nodes of m-ary Merkle tree is $\mathcal{I} = \frac{(\mathcal{L}-1)}{(m-1)}$.*

Proof. The relation between the leaf nodes and internal nodes is given in Equation (1).

$$\mathcal{L} = (m-1) * \mathcal{I} + 1 \tag{1}$$

Therefore, the total number of internal nodes can be derived from Equation (1). Thus, the total number of internal nodes is $\frac{(\mathcal{L}-1)}{(m-1)}$. □

Lemma 1. *The total number of nodes for $\mathcal{L}$ leaf nodes is $n = \mathcal{L} + \frac{(\mathcal{L}-1)}{(m-1)}$.*

Proof. The total number nodes comprises of leaf nodes and internal nodes. Thus, the total number nodes is $n = \mathcal{L} + \mathcal{I}$. Substituting $\mathcal{I}$ using Theorem 1, we get $n = \mathcal{L} + \frac{(\mathcal{L}-1)}{(m-1)}$. □

Theorem 2. *The m-ary Merkle tree has height $h = \lceil log_m n \rceil$ for n nodes.*

Proof. The m-ary Merkle tree is a complete tree, i.e., all leaf nodes are at the same level. Therefore, the total number of nodes is given in Equation (2).

$$n = \sum_{i=0}^{h} m^i = \frac{(m^{h+1}-1)}{(m-1)} \tag{2}$$

By solving the Equation (2), we get the height of the m-ary Merkle tree. The height is derived from the Equation (2) in Equation (3).

$$\begin{aligned} &\frac{(m^{h+1}-1)}{(m-1)} = n \\ &m^{h+1} - 1 = (m-1)n \\ &m^h = \frac{(m-1)n+1}{m-1} \\ &h = log_m(\frac{(m-1)n+1}{m-1}) \\ &h = log_m(m-1) + log_m n + log_m 1 - log_m(m-1) \\ &h = log_m n \end{aligned} \tag{3}$$

Thus, the height of the m-ary Merkle tree is $log_m n$. □

Theorem 2 shows the height of the Merkle tree in terms of the total number of nodes in the tree. The total number of nodes is shown in Lemma 1. Therefore, the total height is $log_m(\mathcal{L} + \frac{(\mathcal{L}-1)}{(m-1)})$ in terms of total number number of leaf nodes. Therefore, the height of the binary Merkle tree is $log_2(2\mathcal{L}-1)$.

Theorem 3. *The m-ary Merkle tree takes $O(log_m n)$ time complexity to insert a new node or delete a node.*

Proof. The insertion or deletion process is similar to the heap tree. The insertion process requires entire rehashing from bottom to top on the insertion path of the tree. The height of the m-ary tree is $log_m n$, and therefore, the m-ary Merkle tree takes $O(log_m n)$ time complexity to insert a new node or delete a node. □

Theorem 4. *The building time complexity of the m-ary Merkle tree is $O(n)$.*

Proof. In an m-ary Merkle tree, there are $\mathcal{L}$ hash function calls in the blocks, i.e., there are $\mathcal{L}$ leaf nodes. From Theorem 1, the total number of hash functions for the internal nodes is $\frac{(\mathcal{L}-1)}{(m-1)}$. It constitutes the total number of nodes where $\mathcal{L} + \frac{(\mathcal{L}-1)}{(m-1)} = n$. Therefore, the total number of the hash function calls is n. Thus, the total time complexity to build the m-ary Merkle tree is $O(n)$. □

Theorem 5. *The verification time complexity of a particular block in m-ary Merkle tree is $O(log_m n)$.*

Proof. All Merkle tree is not required to verify a particular block. A few hash values are needed to verify the correctness of the specific block; however, the tree's total height is the minimum requirement of the hash number. Thus, the verification time complexity of a particular block is $O(log_m n)$, which is the height of the tree. □

Theorem 6. *The verification time complexity of the total number of $\mathcal{L}$ blocks is $O(\mathcal{L}\ log_m n)$*

Proof. In an m-way Merkle tree, there are $\mathcal{L}$ blocks, and the total number of nodes in the tree is n as shown in Lemma 1. Each block requires a verification time complexity of $O(log_m n)$. Therefore, it requires $O(\mathcal{L}\ log_m n)$ time complexity for $\mathcal{L}$ blocks to verify each blocks. The $\mathcal{L} \approx n$, thus, the total time complexity to verify the authenticity of all blocks are $O(n\ log_m n)$. □

Theorem 7. *The extra space complexity of m-ary Merkle tree is $O(n)$.*

Proof. There are n nodes in the m-ary Merkle tree for $\mathcal{L}$ blocks. Therefore, the Merkle tree requires $O(n)$ extra spaces to construct the tree for the given set of blocks. The relation between blocks and nodes is given in Lemma 1. The tree node contains a hash value that is generated by the SHA2 hash function. Therefore, it consumes a large amount of extra memory. □

3 Issues of Merkle Tree

Merkle tree is not a searched tree; it's a simple m-ary or m-way tree. Definition 1 and 2 defines the m-ary tree. Merkle tree construction cost is given in Lemma 1, and Theorem 4. The total cost is given by Lemma 1 and Theorem 4. There are several issues in the Merkle tree, which are outlined below-

- Let us assume that the set of blocks $\mathcal{L}$ consists of millions of blocks. Then, it is not feasible for a conventional computer to construct the Merkle tree. Therefore, the Merkle tree wastes not only energy but also computational resources.
- Merkle tree construction requires $O(n)$ hash function calls which is time-consuming. As a result, it is not feasible for a large set of blocks.
- Merkle tree cannot be constructed by BTree, B$^+$Tree, AVL Tree, or any other search tree; otherwise, it violates the definition of Merkle tree.
- Merkle tree uses secure hash functions; for example, SHA2. However, these secure hash functions are costly, and a single operation also costs significantly, which cannot be neglected.
- Merkle tree requires extra spaces of $O(n)$, and it is costly due to large-sized memory requirements per node. It requires a mammoth-sized RAM for millions of data blocks.

3.1 Communication costs

Computation is much faster than communication. Communication involves many issues; for instance, it requires network access, increases network traffic and latency. Therefore, the Merkle tree can reduce its cost if the Merkle tree can reduce the network accesses. However, it is hard to reduce

the total number of network accesses in the Merkle tree structure. Merkle tree requires $\mathcal{L}$ network accesses to verify all data blocks, and it is the biggest drawback of the Merkle tree. For example, there are η users downloading the same file, and each user is verifying the data blocks, then it creates huge network accesses, which is exactly $\eta\mathcal{L}$. Moreover, the overall verification cost is $O(\eta\mathcal{L}\ log_m n)$.

4 Bloom Filter

Bloom Filter is an approximate membership filter with μ bit array, initially filled with zeros. The insertion process inserts 1 into k slots in the Bloom Filter by k independent hash functions. Query process checks whether all k slots are having 1 or not. If all slots contain 1, then it returns true; false otherwise. The deletion process resets all 1 by 0 in k slots in the bit array by the k independent hash functions.

Definition 3. *A Bloom Filter is a approximate membership filter that can answer "YES" or "NO" with a probability. Let* U *be the universe,* $\mathcal{S} = \{x_1, x_2, x_3, \ldots, x_n\}$ *be the set where* $\mathcal{S} \subset$ U*, and* $\mathcal{B}$ *be the bloom filter of* μ *bit array initially filled with 0. Let* $h = \{h_1, h_2, h_3, \ldots, h_k\}$ *be the* k *independent hash functions. Let* $\mathcal{S}$ *inserted into the Bloom Filter* $\mathcal{B}$ *where* $\mathcal{S} \in \mathcal{B}$ *using* k *independent hash functions. Let* x_i *be a random query and maps it to the Bloom Filter* $\mathcal{B}$ *using* $f : \ h(x_i) \mapsto \{0,1\}*$*. The true positive, false positive, true negative, and false positive are defined below-*

- ***True positive:*** *If* $x_i \in \mathcal{S}$ *and* $x_i \in \mathcal{B}$*, then the result of Bloom Filter is a true positive.*
- ***False positive:*** *If* $x_i \notin \mathcal{S}$ *and* $x_i \in \mathcal{B}$*, then the result of Bloom Filter is a false positive.*
- ***True Negative:*** *If* $x_i \notin \mathcal{S}$ *and* $x_i \notin \mathcal{B}$*, then the result of Bloom Filter is a true negative.*
- ***False negative:*** *If* $x_i \in \mathcal{S}$ *and* $x_i \notin \mathcal{B}$*, then the result of Bloom Filter is a false negative.*

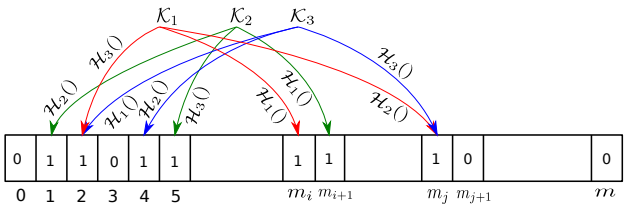

Fig. 3: Bloom Filter: Insertion and Query operations.

The μ is the number of bits in the bit array, the probability of a particular slot is not set to 1 by a specific hash function is $(1 - \frac{1}{\mu})$. The probability of that particular slot is not set to 1 by k hash functions is given by Equation (4).

$$(1 - \frac{1}{\mu})^k \tag{4}$$

We know that

$$e^{-1} = \lim_{\infty}(1 - \frac{1}{\mu})^{\mu} \tag{5}$$

Substituting Equation (4) by Equation (5), we get Equation (6).

$$(1 - \frac{1}{\mu})^k = ((1 - \frac{1}{\mu})^{\mu})^{k/\mu} \approx e^{-k/\mu} \tag{6}$$

If we insert n items into the bit array, then probability of that particular bit is still not set to 1 is given by Equation (7).

$$(1 - \frac{1}{\mu})^{nk} \approx e^{-kn/\mu} \tag{7}$$

The probability of that particular bit is set to 1 is given by Equation (8).

$$(1 - (1 - \frac{1}{\mu})^{nk}) \approx 1 - e^{-kn/\mu} \tag{8}$$

The probability of all slots of μ bit array to be 1 is given in the Equation (9).

$$\varepsilon = (1 - e^{-kn/\mu})^k \tag{9}$$

Equation (9) gives us the false positive probability. However, the value of k must be optimal for a certain number of inputs n; for instance, a large value of k increases the false positive probability, and also, a small value of k increases the false positive probability. Therefore, the value of k must be optimal to reduce the false positive probability.

$$k = \frac{\mu}{n} ln\ 2 \tag{10}$$

Equation (10) gives optimal false positive probability for given μ bits array and n input items [20]. Now, the value of k is replaced in Equation (9), we get Equation (11).

$$\varepsilon = \left(1 - e^{-(\frac{\mu}{n} ln 2)\frac{n}{\mu}}\right)^{\frac{\mu}{n} ln 2} \tag{11}$$

Taking ln both side of Equation (11), we get Equation (12).

$$\begin{aligned} ln\ \varepsilon &= -\frac{\mu}{n}(ln\ 2)^2 \\ \mu &= -\frac{n\ ln\ \varepsilon}{(ln\ 2)^2} \end{aligned} \tag{12}$$

Equation (12) gives us the required memory size in bits for a given number input items and a desired false positive probability.

Property 1. *The time complexity of insertion, query and deletion in Bloom Filter is* $O(k) \approx O(1)$ *where* k *is the total number of hash functions.*

Equation (10) gives the total number of hash function requirements. However, it is a pretty small number for large input items as shown in Property 1; for instance, it requires $k = 10$ hash functions for 10M inputs in standard Bloom Filter [21]. However, it takes $k = 5$ for the exact requirements in 2D Bloom Filter [22].

Property 2. *The extra space complexity of Bloom Filter is* $\mu = -\frac{n\ ln\varepsilon}{(ln\ 2)^2}$ *bits.*

Property 2 shows the relations among the required memory, error rate, and input items. It consumes $\mu = 17.14MiB$ for 10M input items and 0.001 false positive probability standard Bloom Filter [21]; however, it takes $k = 2MB$ for the same settings in 2D Bloom Filter [23], [24]. Thus, Bloom Filter uses a tiny amount of memory. There are also diverse fast approximation filters, namely, Morton filters [25], and XOR Filters [26]. Recent research suggests that Bloom Filters can be constructed with false positive free zone [27], [28].

TABLE 1: Comparison between Merkle tree and Bloom Filter. The $O(k) \approx O(1)$ because the value of k is nearly constant.

Features	Merkle tree	Bloom Filters
Building cost	$O(n)$	$O(k\mathcal{L})$
Insertion cost	$O(log_m n)$	$O(k)$
Deletion cost	$O(log_m n)$	$O(k)$
Updating cost	$O(log_m n)$	$O(k)$
Verification of a block	$O(log_m n)$	$O(k)$
Verification of all blocks	$O(n\ log_m n)$	$O(k\mathcal{L})$
Extra spaces	$O(n)$	μ
Type	Deterministic	Approximations

4.1 Comparison

Table 1 demonstrates the comparison between Merkle tree and Bloom Filter. Bloom Filters are faster than Merkle tree. Bloom Filter uses a non-cryptographic string hash function, while Merkle tree uses a cryptographic string hash function. Therefore, the Merkle tree is much slower than Bloom Filter. Patgiri *et al.* compares the performance of between the non-cryptographic string hash function and cryptographic string hash function [20]. Moreover, the construction cost of Bloom Filter is faster than Merkle tree. Similarly, verification of a particular block in Bloom Filter is also faster than Merkle tree. In short, Bloom Filter is much faster in all operations than Merkle tree. However, the Bloom Filter is an approximation data structure, while the Merkle tree is a deterministic data structure. Therefore, Bloom Filter cannot replace the Merkle tree due to its false positive probability.

5 LINKEDHASHX

We propose a new variant of Merkle Tree using hash and XOR operations, LinkedHashX for short, which is used to replace Merkle tree. LinkedHashX maintains its root. It does not require maintaining process.

Property 3. *If two same keys are XORed, then it produce zero output, i.e.,* $\mathcal{X}_1 \oplus \mathcal{X}_1 = 0$.

Definition 4. *Let* h_p, h_b, *and* h_d *be the previous hash value, the current hash value of a particular block, and the desired hash value, respectively. Then, LinkedHashX can be defined as given in Equation* (13).

$$h_d = h_p \oplus h_b \tag{13}$$

5.1 Construction

The construction of LinkedHashX is straightforward and simple. Equation (14) shows the construction process of $\mathcal{L}$ blocks. The last hash value of Equation (14) is the root of LinkedHashX.

$$\begin{aligned} &h_1 \oplus h_2 = h_{12} \oplus h_3 = h_{123} \oplus h_4 = h_{1234} \oplus h_5 \\ &= h_{12345} \oplus h_6 = h_{123456} \oplus h_7 = \\ &h_{1234567} \oplus h_8 = h_{12345678} \oplus \ldots = h_{12345678\ldots\mathcal{L}} \end{aligned} \tag{14}$$

Total hash functions calls are exactly $\mathcal{L}$ and the total number of XOR operations are $(\mathcal{L}-1)$. Therefore, the total construction cost is $(2\mathcal{L}-1)$. The XOR operations are much faster than the hash operations. However, conventional Merkle tree requires $(2\mathcal{L}-1)$ hash functions calls and $(\mathcal{L}-1)$ XOR operations for binary tree structure. LinkedHashX publishes the root $h_{1234\ldots\mathcal{L}}$ publicly along with the hash value of the root, i.e., $h_{root} = h(h_{123\ldots\mathcal{L}})$.

5.2 Insertion operation

To insert a new node, a new hash value of a transaction is XORed with root of Equation (14).

$$h_{12345678} \oplus h_i = h_{12345678i\mathcal{L}} \tag{15}$$

Equation (15) gives the the insertion operations. Therefore, it requires a single hash function call and an XOR operation. The total number of operations is 2, and hence, the time complexity is $O(1)$.

5.3 Delete operation

Delete operation is straightforward and straightforward by using Property 3. For instance, we would like to delete h_5, then perform XOR operation h_5 with the root of Equation (14).

$$h_{12345678} \oplus h_5 = h_{1234678\mathcal{L}} \tag{16}$$

Equation (16) shows the deletion process. It requires a single operation which is the XOR operation. The hash value h_5 is already calculated for LinkedHashX. Therefore, its time complexity is $O(1)$.

5.4 Update operation

Let us assume that the h_1 is to be updated to the new hash value h_u.

$$\begin{aligned} h_{12345678} \oplus h_1 &= h_{2345678\mathcal{L}} \\ h_{2345678} \oplus h_u &= h_{2345678u\mathcal{L}} \end{aligned} \tag{17}$$

Equation (17) demonstrates the update operation. It requires deleting the existing hash value h_1 by performing an XOR operation with the root of LinkedHashX. The new hash value h_u can be inserted into the root of LinkedHashX for the update. The two XOR operations and a hash function call are required to update a particular node. Therefore, its time complexity becomes $O(1)$.

5.5 Space complexity

LinkedHashX does not use any extra spaces, and therefore, its space complexity is $O(1)$. LinkedHashX maintains its root only; thus, there is no additional space requirement. On the contrary, most of the Merkle tree implementation uses at least $O(\mathcal{L})$ space complexity where $\mathcal{L}$ is the number of blocks.

5.6 Verification

One of the most significant disadvantages of LinkedHashX is its verification. Suppose a user downloads a block of a file. But it cannot verify partially in LinkedHashX, but the Merkle tree offers verification of the authenticity of a particular block. In LinkedHashX, it requires complete downloading of all blocks to verify the authenticity and integrity. Therefore, LinkedHashX cannot replace the Merkle tree due to the unavailability of partial verification of authenticity and integrity.

TABLE 2: Caption

Complexity	Merkle tree	LinkedHashX
Building cost	$O(n)$	$O(\mathcal{L})$
Insertion cost	$O(log_m n)$	$O(1)$
Deletion cost	$O(log_m n)$	$O(1)$
Updating cost	$O(log_m n)$	$O(1)$
Verification of a block	$O(log_m n)$	Doesn't support
Verification of all blocks	$O(n\ log_m n)$	$O(\mathcal{L})$
Extra spaces	$O(n)$	$O(1)$

5.7 Comparison

Table 2 compares the conventional Merkle tree and LinkedHashX. Merkle tree invokes n hash functions while LinkedHashX invokes $\mathcal{L}$ in the time of constructions. Therefore, LinkedHashX is much faster than Merkle tree. However, LinkedHashX does not permit partial verification of a block's authenticity and integrity. Therefore, LinkedHashX cannot be used directly in peer-to-peer networking. The insertion, deletion, and updating operations do not require in most applications.

6 HEX-BLOOM: THE REPLACEMENT

Merkle tree is costly in terms of time/space complexity and network access. Therefore, it requires replacing the Merkle tree with new design philosophy.

Our proposed system combines both LinkedHashX and Bloom Filter. However, Bloom Filter can be replaced with other approximation filters like Cuckoo Filter [29], Morton filters [25], and XOR Filters [26].

6.1 Construction

In the construction process, it requires to construct LinkedHashX as well as Bloom Filter. All hash values are inserted into Bloom Filter. Similarly, the construction of LinkedHashX requires $(\mathcal{L}+1)$ cryptographic hash functions and $(\mathcal{L}-1)$ XOR operations. Bloom Filter requires $\mathcal{L}$ insertion operation with $k \times \mathcal{L}$ non-cryptographic string hash functions. However, the non-cryptographic string hash function is much faster than the cryptographic string hash function [20]. The Bloom Filter and the root of LinkedHashX are maintained for future use.

6.2 Verification

The verification process requires the LinkedHashX root and Bloom Filter, which are downloaded from the trusted node or server. The LinkedHashX root contains the XOR value of all the hash values and the hash of the final results, i.e., $h_{123...\mathcal{L}}$ and h_{root}. The partial verification process requires Bloom Filter because LinkedHashX does not support partial verification of a block's authenticity. A block is verified in Bloom Filter for its authenticity after completely downloaded the block, and at the same time, the user needs to construct the LinkedHashX. Thus, the Bloom Filter gives approximate verification of a block's authenticity and integrity. LinkedHashX is constructed by the user while downloading the data blocks. When all processes are completed, the constructed LinkedHashX root is compared with the downloaded LinkedHashX root.

7 ANALYSIS

Bloom Filter results true or false depending on approximation. The true result of the Bloom Filter does not guarantee the existence of the item in the Bloom Filter. However, the false result of the Bloom Filter guarantees that the item is not a member of the Bloom Filter.

7.1 Communication cost

The Conventional Merkle tree process creates enormous network traffic. A set of data blocks n creates n network access for a single user. Suppose there are η users, and these users are interested in the same data, then network accessing cost becomes $O(\eta n)$. It creates unnecessary network traffic in verifying each block and transmitting the required hash values to verify each block's authenticity and integrity. In our proposed solution, A user downloads the LinkedHashX root and Bloom Filter only once. Then, the user verifies each block's authenticity and integrity in Bloom Filter. The complete proof can be done after the completion of data downloading. Also, it provides deterministic proof for the correctness of the downloaded data using LinkedHashX root. Thus, it reduces the communication cost significantly.

7.2 Comparison

TABLE 3: Comparison between Merkle tree and our proposed model.

Features	Merkle tree	Our solution
Building cost	$O(n)$	$O(k\mathcal{L})$
Verification of a block	$O(log_m n)$	$O(k)$
Verification of all blocks	$O(n\ log_m n)$	$O(k\mathcal{L})$
Extra spaces	$O(n)$	μ
Communication cost	$O(\mathcal{L})$	$O(1)$
Total non-cryptographic hash functions	NA	$k\mathcal{L}$
Total cryptographic hash functions	n	$\mathcal{L}$
Type	Deterministic	Approximations and deterministic

HEX-BLOOM is much faster than the existing state-of-the-art Merkle tree solution. The total construction cost of our proposed model is $O(k\mathcal{L})$, whereas the Merkle tree takes $O(n)$ time complexity. Here, the k is the number of non-cryptographic string hash function calls, and it can be ignored due to a small-sized value. Therefore, the Merkle tree is slower than our proposed system in construction. Furthermore, the verification of a block requires network access, and its performance depends on the latency. Also, it requires $O(log_m n)$ time complexity which consists of all cryptographic string hash functions. Besides, it requires $\mathcal{L}$ times network accesses to verify all block's authenticity. Therefore, HEX-BLOOM is significantly faster than the existing state-of-the-art Merkle tree because our proposed model does not require network access for verification. The state-of-the-art Merkle tree consumes $n \times \beta$ where β is the bit size of the hash value. On the contrary, our proposed model consumes μ memory for Bloom Filter and 2β bits for LinkedHashX root. Overall, HEX-BLOOM is a better choice than the Merkle tree in every aspect.

7.3 Drawback

Case 1. *There is a false positive in Bloom Filter if the constructed LinkedHashX root does not match with constructed LinkedHashX root.*

Case 1 states that Bloom Filter can return a false positive. The downloaded LinkedHashX root and constructed LinkedHashX root mismatches due to the false positive probability. A few blocks are corrupted or altered, and hence, Bloom Filter returns true for the corrupted or altered blocks. It leads to a complete download of the data blocks and found the mismatch between the downloaded LinkedHashX root and constructed LinkedHashX root. Then, there is an error, and therefore, the entire process has to be redone. In Case 1, it is not possible to diagnose the error and unable to find which block causes the error. However, it is improbable for the more extensive set of blocks. If the Bloom filter returns negative, the process is terminated.

8 Conclusion

In this article, we unearth the drawback of the Merkle tree and propose a new model to solve the issue of the Merkle tree, called HEX-BLOOM. HEX-BLOOM is significantly faster than the state-of-the-art Merkle tree. Moreover, it also consumes less memory than the state-of-the-art Merkle tree. We have demonstrated that the Merkle tree creates unnecessary network traffic, which is addressed in our proposed model. State-of-the-art Merkle tree requires network access to verify a block, and therefore, it is time-consuming, and its performance depends on the network latency. On the contrary, HEX-BLOOM depends on Bloom Filter to verify each block's authenticity. Therefore, it does not require network access in the verification process of each block. Overall, HEX-BLOOM outperforms the state-of-the-art Merkle tree in every aspect.

References

[1] R. C. Merkle, "A Digital Signature Based on a Conventional Encryption Function," in *Advances in Cryptology – CRYPTO '87*. Berlin, Germany: Springer, Dec 2000, pp. 369–378.

[2] D. Lee and N. Park, "Blockchain based privacy preserving multimedia intelligent video surveillance using secure Merkle tree," *Multimed. Tools Appl.*, pp. 1–18, Mar 2020.

[3] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," Manubot, Tech. Rep., 2019. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[4] M. Yu, S. Sahraei, S. Li, S. Avestimehr, S. Kannan, and P. Viswanath, "Coded merkle tree: Solving data availability attacks in blockchains," in *Financial Cryptography and Data Security*, J. Bonneau and N. Heninger, Eds. Cham: Springer International Publishing, 2020, pp. 114–134.

[5] J. Xu, L. Wei, Y. Zhang, A. Wang, F. Zhou, and C.-z. Gao, "Dynamic Fully Homomorphic encryption-based Merkle Tree for lightweight streaming authenticated data structures," *Journal of Network and Computer Applications*, vol. 107, pp. 113–124, Apr 2018.

[6] G. Becker, "Merkle signature schemes, merkle trees and their cryptanalysis," *Ruhr-University Bochum, Tech. Rep*, 2008.

[7] M. D. Borah, V. B. Naik, R. Patgiri, A. Bhargav, B. Phukan, and S. G. M. Basani, "Supply Chain Management in Agriculture Using Blockchain and IoT," in *Advanced Applications of Blockchain Technology*. Singapore: Springer, Sep 2019, pp. 227–242.

[8] S. Hariharasitaraman and S. P. Balakannan, "A dynamic data security mechanism based on position aware Merkle tree for health rehabilitation services over cloud," *J. Ambient Intell. Hum. Comput.*, pp. 1–15, Jul 2019.

[9] J. A. Alzubi, "Blockchain-based Lamport Merkle Digital Signature: Authentication tool in IoT healthcare," *Comput. Commun.*, vol. 170, pp. 200–208, Mar 2021.

[10] S. Dhumwad, M. Sukhadeve, C. Naik, M. K.N., and S. Prabhu, "A peer to peer money transfer using sha256 and merkle tree," in *2017 23RD Annual International Conference in Advanced Computing and Communications (ADCOM)*, 2017, pp. 40–43.

[11] H. Li, R. Lu, L. Zhou, B. Yang, and X. Shen, "An efficient merkle-tree-based authentication scheme for smart grid," *IEEE Systems Journal*, vol. 8, no. 2, pp. 655–663, 2014.

[12] J. Mao, Y. Zhang, P. Li, T. Li, Q. Wu, and J. Liu, "A position-aware Merkle tree for dynamic cloud data integrity verification," *Soft Comput.*, vol. 21, no. 8, pp. 2151–2164, Apr 2017.

[13] Y. Wang, Y. Shen, H. Wang, J. Cao, and X. Jiang, "Mtmr: Ensuring mapreduce computation integrity with merkle tree-based verifications," *IEEE Transactions on Big Data*, vol. 4, no. 3, pp. 418–431, 2018.

[14] Z. Sun, Y. Liu, J. Wang, F. Mei, W. Deng, and Y. Ge, "Non-cooperative game of throughput and hash length for adaptive merkle tree in mobile wireless networks," *IEEE Transactions on Vehicular Technology*, vol. 68, no. 5, pp. 4625–4650, 2019.

[15] M. Jakobsson, T. Leighton, S. Micali, and M. Szydlo, "Fractal Merkle Tree Representation and Traversal," in *Topics in Cryptology — CT-RSA 2003*. Berlin, Germany: Springer, Feb 2003, pp. 314–326.

[16] M. Szydlo, "Merkle Tree Traversal in Log Space and Time," in *Advances in Cryptology - EUROCRYPT 2004*. Berlin, Germany: Springer, May 2004, pp. 541–554.

[17] J. Buchmann, E. Dahmen, and M. Schneider, "Merkle Tree Traversal Revisited," in *Post-Quantum Cryptography*. Berlin, Germany: Springer, Oct 2008, pp. 63–78.

[18] U. Chelladurai and S. Pandian, "HARE: A new Hash-based Authenticated Reliable and Efficient Modified Merkle Tree Data Structure to Ensure Integrity of Data in the Healthcare Systems," *J. Ambient Intell. Hum. Comput.*, pp. 1–15, Apr 2021.

[19] Y. Zou and M. Lin, "Fast: A frequency-aware skewed merkle tree for fpga-secured embedded systems," in *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2019, pp. 326–331.

[20] R. Patgiri, S. Nayak, and N. B. Muppalaneni, "Is bloom filter a bad choice for security and privacy?" in *2021 International Conference on Information Networking (ICOIN)*, 2021, pp. 648–653.

[21] A. Kirsch and M. Mitzenmacher, "Less hashing, same performance: Building a better bloom filter," *Random Struct. Algorithms*, vol. 33, no. 2, p. 187–218, Sep. 2008.

[22] R. Patgiri, A. Biswas, and S. Nayak, "deepBF: Malicious URL detection using learned bloom filter and evolutionary deep learning," *CoRR*, vol. abs/2103.12544, 2021. [Online]. Available: https://arxiv.org/abs/2103.12544

[23] R. Patgiri, "robustBF: A high accuracy and memory efficient 2d bloom filter," 2021.

[24] S. Nayak and R. Patgiri, "countBF: A general-purpose high accuracy and space efficient counting bloom filter," 2021.

[25] A. D. Breslow and N. S. Jayasena, "Morton filters: Faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity," *Proc. VLDB Endow.*, vol. 11, no. 9, p. 1041–1055, May 2018. [Online]. Available: https://doi.org/10.14778/3213880.3213884

[26] T. M. Graf and D. Lemire, "Xor filters: Faster and smaller than bloom and cuckoo filters," *ACM J. Exp. Algorithmics*, vol. 25, Mar. 2020. [Online]. Available: https://doi.org/10.1145/3376122

[27] O. Rottenstreich, P. Reviriego, E. Porat, and S. Muthukrishnan, "Constructions and applications for accurate counting of the bloom filter false positive free zone," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 135–145. [Online]. Available: https://doi.org/10.1145/3373360.3380845

[28] S. Z. Kiss, E. Hosszu, J. Tapolcai, L. Ronyai, and O. Rottenstreich, "Bloom filter with a false positive free zone," *IEEE Transactions on Network and Service Management*, pp. 1–1, 2021.

[29] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 75–88. [Online]. Available: https://doi.org/10.1145/2674005.2674994