Bloom Filters with Polynomial Commitment Multiproofs

Thanakul Wattanawong
j.wat@berkeley.edu

Noppapon Chalermchockcharoenkit
noppaponnc@berkeley.edu

May 2023

1 Introduction

Merkle Trees [8] are a cryptographic data structure used for verifying the integrity and authenticity of a data set. They consist of a binary tree of hash values, where each leaf node stores the hash of a data element, and each non-leaf node stores the hash of the concatenation of each child node. By requesting a proof, a client can query the server to verify the authenticity of a piece of data on the client side. Merkle Trees have found widespread use, serving as backbones for systems such as Ethereum [9], Google's Certificate Transparency [3], and Git [13].

The Bloom Tree [12] is a variant of Merkle Trees that instead constructs the Merkle Tree over chunks of a Bloom filter. Bloom filters can be much smaller than the set they represent, and thus lead to good efficiency gains in proof size at the cost of relaxing guaranteed correctness for presence proofs. Regardless, they still work extremely well for absence proofs and there are some applications that can either mitigate or work around the probabilistic nature of presence checking.

The contribution of this work is two-fold. First, we present a survey of the current literature around efficient Merkle Tree variants and techniques. Second, we present a novel scheme called Bloom Filters with Polynomial Commitment Multiproofs that combine multiproof polynomial commitments with Bloom Filters in order to reduce the proof size to $O(1)$ at the cost of extra computation on the client and server. We compare our method with that of the existing literature and find it is competitive in terms of proof size.

1.1 Problem Setting

Although Merkle Trees only require $O(log_2(n))$ hashes to be computed for a proof to verify the existence of a single leaf or data point, in practice this is still too large. Consider the setting from [6] where if Alice has 2^{30} files, she will need to download and check a proof of 30 hashes to verify one file. If a cryptographically secure hash-function such as SHA-256 is used, this cost can be around 1kB, which may exceed the size of the file itself!

As such, there is a need for more efficient hash trees that aim to reduce proof size as much as possible in order to accomodate low-bandwidth situations such as mobile or edge. Preferably, the ideal situation is if we can achieve a constant proof size with strong guarantees.

2 Preliminaries

2.1 Merkle Tree Construction

Merkle Trees are binary trees where leaves store the hash of data sets and intermediate nodes store the hash of their children's values concatenated. Concretely, given a data set partitioned into $x_1, \dots, x_n$ ordered values, each leaf node stores $h(x_i)$ where h is some hash function such as SHA-2, and each internal node with children v_1, v_2 stores $h(v_1 v_2)$.

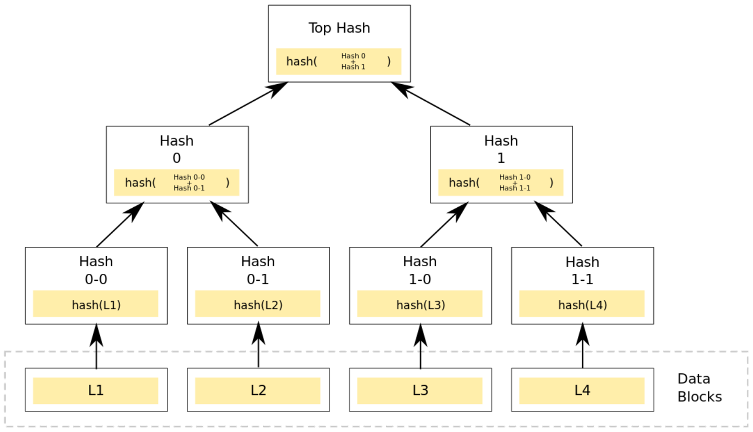

Figure 1: Visualization of a Merkle Tree.

2.2 Merkle proof

A Merkle tree can verify the existence of an element using an *audit proof* of size $O(\log_2 n)$. A client with a chunk of data can query the server for an *audit trail* consisting of nodes required to recompute the root hash. For example, in Figure 1, a client querying for $L2$ would receive $hash(0-0)$ and $hash(hash(1-0), hash(1-1))$. If the root hash matches, the proof validates that the data exists in the tree and is not corrupted. Merkle trees can be constructed in $O(n)$ time and a chunk can be updated in $O(log_2(n))$.

2.3 KZG Polynomial Commitment

Commitment schemes are cryptographic primitives that allows the prover to produce an output that *commits* to a value while hiding it. The message itself is secret and cannot be changed, but can be revealed later using an *opening* from the prover, after which the verifier can confirm the authenticity of a message. In this way, the verifying recipient does not know the contents of the message until the sender opens it, but they can verify that the contents of the message have not been tampered with in the meanwhile.

In KZG (Kate, Zaverucha, and Goldberg) Polynomial Commitment [5], we convert all data points (each point denoted as (x_i, y_i)) into a single polynomial, such that $p(x_i) = y_i \ \forall i$ — one way to do so is through Lagrangian Interpolation. The following section will explain how the prover can prove that $p(x_i) = y_i$ to the verifier while preserving both **correctness** and **soundness** using a proof (opening) of size $O(1)$.

Let G_1 and G_2 be two different elliptical curves, both of order p, with pairing $e : G_1 \cdot G_2 \Rightarrow G_T$. If G and H are the generators of G_1 and G_2 respectively, we can write $[x]_1 = xG \in G_1$ and $[x]_2 = xH \in G_2$ for any x in $\mathbb{F}_p$ [7].

Proving an evaluation at one point

1.) In a trusted setup, for some secret s, the elements $[s^i]_1$ and $[s^i]_2$ are distributed to both prover and verifier. The prover sends the commitment of the polynomial $p(x)$ evaluated at s, $[p(s)]_1$ to the verifier. Notice that both the prover nor the verifier know the value of s at this stage.

2.) Verifier asks for the evaluation for the committed polynomial at a point t, i.e. $p(t)$.

3.) Notice that $p(x) - p(t) = (x - t)h(x)$, where $h(x)$ is some polynomial. Rearranging gives $h(x) = \frac{p(x)-p(t)}{x-t}$. The prover again sends another commitment, but now of polynomial $h(x)$ evaluated at s, $[h(s)]_1$. $[h(s)]_1$ is commonly denoted as the proof π.

4.) As a recap, the verifier first receives $[p(s)]_1$ and then $[h(s)]_1$. Also, we know that $p(s) - p(t) = (s - t) \times h(s)$. The verifier can compute 1.) $[s - t]_2$ and 2.) $[p(t)]_1$ using the received $[p(s)]_1$ by themself. With the property of elliptical curves, where e is a bilinear mapping, the verifier can now verify whether $[p(s)]_1$ sent by the prover is the commitment of the polynomial $p(x)$ by checking $e([p(s) - p(t)]_1, [1]_2) = e([s - t]_2, [h(s)]_1)$.

Proving an evaluation at multiple points (Multiproofs)

Next we present the multiproof polynomial commitment method from [7] that allows the prover to prove multiple points at the same time using a single proof. Suppose the verifier wants to evaluate the polynomial at k different points: $(x_0, y_0), (x_1, y_1), ...,$
(x_{k-1}, y_{k-1}). The prover can find a polynomial going through all of these points using Lagragian Interpolation. Let's called this polynomial $I(x)$. If $p(x)$ goes through all the k points, $p(x) - I(x)$ will be equal to 0 at each x_i, which implies that $p(x) - I(x)$ is divisible by $Z(x) = \Pi_0^{k-1}(x - x_i)$. Therefore, just like the single point case mentioned above, we can write the equation $p(x) - I(x) = Z(x) \cdot h(x)$, and rearranging gives $h(x) = \frac{p(x)-I(x)}{Z(x)}$. The prover commits to $h(x)$ at s, creating a commitment $[h(s)]_1$, and the verifier can then verify by checking if $e([p(s) - I(s)]_1, [1]_2) = e([Z(s)]_2, [h(s)]_1)$.

3 Merkle Tree Variants

Next we summarize some key developments around Merkle Tree variants that are specially crafted for a particular application or have various benefits over the vanilla design.

3.1 Verkle Tree

Verkle Trees were proposed in 2019 by Kuszmaul in [6]. It's primary contributions include increasing the branch size to k instead of 2, and using vector commitments to have a constant intra-node proof size.

Naively, simply increasing the branching factor to k of a Merkle Tree can increase the speed of queries and construction as trees are relatively shorter. However, the size of the proofs also increase, as one must send all the child neighbors of each node along the Merkle path, and instead of there being only one at each level there are now k. Verkle Trees propose the use of a vector commitment scheme called Catalano-Fiore [2] for which the specific details are out of the scope of this survey paper. This scheme allows for $O(1)$ proof size at each internal node.

A vector commitment scheme is simply a scheme that maps $h(a_1, ..., a_k) \Rightarrow C$ at each intermediate node, where we also have a method of checking if for any value x and index i, $x = a_i \in C$ cheaply.

Using this method, the proof size is reduced from $O(log_2(n))$ to $O(log_k(n))$ due to the increased branching factor. Nonetheless, the construction time is increased from $O(n)$ to $O(kn)$.

Additionally, the cost of updating a chunk increases from $O(log_2(n))$ to $O(klog_k(n))$ as there are now k children on each internal node to be updated. As such, the k value can be seen as a knob to trade off compute for bandwidth.

Using vector commitment schemes to reduce the proof size has clear benefits to bandwidth reduction, and furthermore Verkle Trees allow this tradeoff to be adjusted with the k value. For example $k = 10$ would decrease the proof size by a factor of $log_2(10)$. Kuszman also created a reference implementation showing that Verkle Trees can have good empirical performance.

3.2 Merkle Patricia Trie

Patricia Trie A Patricia Trie [11] is a radix tree of with radix value 2. A radix tree itself is a tree where each internal node has at most r children, and a power of 2. They are also usually space-optimized, where if a node has only one child, it is merged with its parent. For example, in a regular trie storing the word "patricia" and "patrick", there would be nodes for p, a, t, and r separately, but in a radix tree, the "patri" would be stored together.

The Ethereum project uses Merkle Patricia Tries [9] as an authenticated key-value store that underpins the Ethereum Blockchain network, storing transactions, receipts, and state. The hashes returned by a Merkle proof query on each of these trees allow for quick and verifiable queries for many questions as whether a transaction was included on a particular block or the current balance on an account.

The Ethereum project uses Merkle Patricia Trees instead of standard Merkle Trees or Tries for two reasons. While the standard Merkle Tree is good in the offline setting where data can be broken up into chunks and held static, state in Ethereum consists of many key-value tuples, where keys and values may be updated frequently. The internal nodes in a Merkle Patricia tree store parts of the key, and the leaf nodes hold the values. However, for a large hash, the tree depth can be very big, and so a Patricia Tree can compress the height. As such Merkle Patricia Trees provide three main properties [10] in addition to the standard Merkle proof:

1. The new tree root hash can be quickly calculated after an update operation without rebuilding

the entire Merkle Tree once the underlying data has changed. As internal nodes in the MPT are simply nibbles of the key and keys change regularly, this property must be satisfied for performance.

2. The depth of the tree is bounded. This is easy to observe as Tries have depth bounded by their key length, and hash lengths are fixed.

3. The root of the tree depends only on the order, not the sequence of updates.

In terms of the larger picture, the MPT is a modified Merkle Tree that is designed to allow for rapid updates and minimize tree depth to guard against malicious attackers hoping to perform a denial-of-service attack on the server. This addresses many of the Merkle Tree's original limitations for example the inability to update the whole tree without recomputing, or the fact that the proof size can grow beyond what is practical.

3.3 Efficient Sparse Merkle Tree

Efficient Sparse Merkle Trees were first proposed in [4] by Dahlberg et al. They use the idea of a perfect Merkle Tree i.e. one that contains every single value that can be output from the hash function. Such trees are intractable to due to the large output space of most modern hash functions, but the authors observe that most realizations of such trees are sparse i.e. many branches lead to an empty leaf.

As such, an efficient representation that does not materialize the empty branches into memory is sufficient, and the authors present a caching-based approach.

One advantage of Sparse Merkle Trees in general is that they support efficient absence proofs. This is because the presence of a null leaf in the full tree efficiently indicates that the provided chunk is not in the Merkle Tree.

3.4 Bloom Tree

Bloom Filters Bloom filters [1] are memory-efficient probabilistic data structures used for membership testing. They consist of an m sized bit array initially set to 0, and k hash functions from $[U] \Rightarrow m$. To add a new element, compute all k hashes and set the bits in the array corresponding to the hashed values to 1. The construction of a bloom filter requires a bit array of m bits, all initially set to 0, and k hash functions. To add a new element to the filter, we compute its hashes once with each hash function, and converts the bits to 1 at the positions the hashes equate to. To perform membership testing, one can compute the k hashes using the same set of hash functions and check whether all of the corresponding indices are 1.

Bloom Filter can return a false positive when it contains multiple elements, as some indices of the bit array could be 1 as a result of other insertions. Therefore, during the membership testing, the indices in context all might have value equal to 1 even though the element is never inserted, which happens with probability approximately $(1 - e^{-kn/m})^k$. Notice that a false negative is never possible.

Bloom Trees [12] build first construct a bloom filter over the dataset. Then, the bloom filter is divided into a chunk size such as 32 bytes and a Merkle Tree is constructed over it.

Sparse Merkle Multiproof is a method of batching multiple Merkle Proofs in order to reduce the average number of hashes that must be transmitted per data checked. Consider the case where two adjacent elements that are sublings are checked. Then their proofs can be combined which can lead to less data transmitted. In general, any internal node where each of it's children is also sent can be safely omitted from the Merkle proof.

In order to perform a lookup, one computes the k hashes required to check a Bloom Filter, and then asks for a Sparse Merkle Multiproof of those k hashes to check whether the bits that correspond to each of the hash values are set to 1 i.e. present. If all are present, then the data is inside the Bloom Tree with high probability.

If the data is not present, only one proof of a single zero value needs to be sent. This makes absence proofs exceptionally efficient.

4 Bloom Filters with Polynomial Commitment Multiproofs

The idea that we propose is the Bloom Filter with Polynomial Commitment Multiproofs, which allows proofs of size $O(1)$ to be transmitted.

We do this by using a Polynomial Commitment scheme. Polynomial commitment schemes are an alternative option for providing presence or absence proofs, and they are commonly used as both a stand-alone and in combination with some Merkle Tree variants, including Verkle Tree. By utilizing the constant proof size property of the **KZG Polynomial Commitment Scheme**, we can transmit the presence or absence proof for a bloom filter in $O(1)$. One caveat is that we first must transmit a shared secret through a trusted setup, which is not a requirement in other schemes such as the Bloom Tree. Furthermore, polynomial commitment proofs can be fairly expensive to construct since Lagrangian interpolation takes $O(k^2)$ time.

Just like in Bloom Tree, we divide the bloom filter into chunks of bytes. However, instead of constructing a Merkle Tree on top of these chunks, we construct a polynomial $p(x)$ such that $p(i)$ = chunk i's content $\forall i$. Whenever a recipient requests a presence or an absence proof of element z, we first hash the element using each of the k hash functions and deduce which chunks of the bloom filter are needed. If all the values at the given indices are all one, the prover then can send a polynomial commitment multi-proof for the k chunks. On the other hand, if the index in context of any given chunks is zero, which means the element is not in the set, the prover sends a proof for any one chunk that contains the zero index. This is illustrated in Fig. 2.

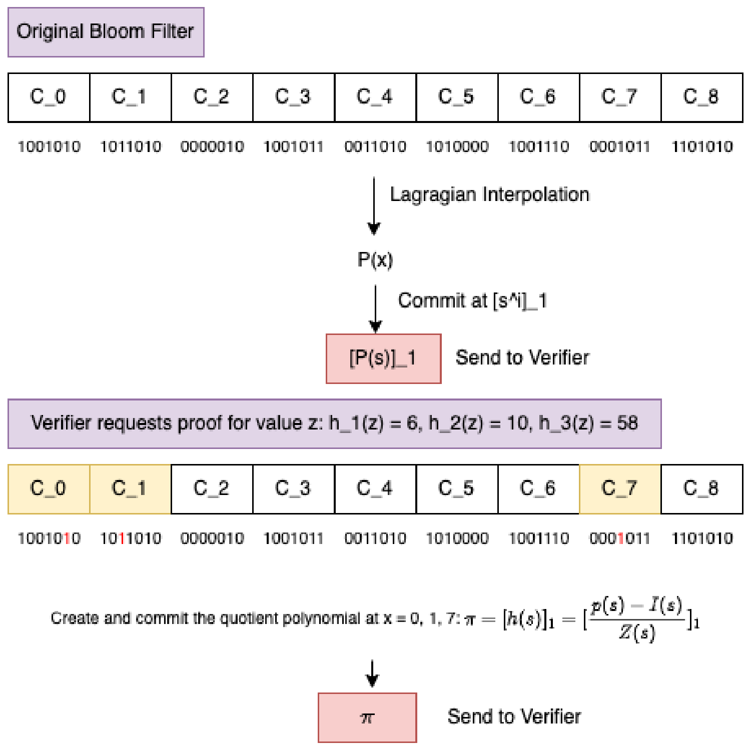

Figure 2: Polynomial Commitment Multiproof Workflow for Bloom Filter.

Notice that after the prover commits to their polynomial, they can't find another polynomial $q(x)$ with the same commitment as the original $p(x)$, i.e. $[p(s)]_1 = [q(s)]_1$. This is because $[p(s)]_1 - [q(s)]_1 = [p(s) - q(s)]_1 = 0 \Rightarrow p(s) - q(s) = 0$. Additionally, the prover can't intentionally create an incorrect proof, tricking the verifier into believing that $p(x_i) = y_i'$ for some $y_i' \neq y_i$. If so, the prover has to divide $p(x_i) - y_i'$ by $x - x_i$, resulting in a remainder.

The runtime complexity for proof construction and validation is $O(k^2)$, where k is the number of hash functions in the Bloom Filter due to the Lagrangian Interpolation computation on both the prover and the verifier side. Therefore, even though the use of Polynomial Commitment to transmit a Bloom Filter reduces the proof size down to $O(1)$, this comes at a cost of $O(k^2)$ computation to construct and validate, which is expensive compared to $O(\log_2(k))$ in Bloom Tree.

5 Comparison

Due to the large amount of inconsistent literature around Merkle Trees and their variants, we provide a useful table summary of all current approaches in 1 and also list our approach:

Table 1: Merkle Tree Variants

Name	**Techniques**	**Proof Size**	**Inclusion Pf.**	**Exclusion Pf.**
Merkle Tree	-	$O(log_2(n))$	Exact	Infeasible
Verkle Tree	Vector Commitment, n-ary tree	$O(log_k(n))$	Exact	Infeasible
Merkle Patricia Tree	Patricia Trie	$O(1)$	Exact	Exact
Efficient SMT	Sparse Perfect Merkle Tree	$O(1)$	Exact	Exact
Bloom Tree	Bloom Filter	$O(clog_2(c))$ incl., $O(log_2(c))$ excl.	Probabilistic	Exact
Bloom Filter w/ Polynomial Commitment Multiproofs	Bloom Filter & Polynomial Commit.	$O(1)$	Probabilistic	Exact

6 Conclusion

In this paper we have conducted a survey of efficient Merkle Trees and methods for verifying the presence/absence of data using a proof that is much smaller than the data they represent. We also presented our own scheme for presence/absence proofs that combines multiproof polynomial commitments with Bloom filters to achieve $O(1)$ proof size but requires computation quadratic to the number of hash functions used. We hope that we or others may create a reference implementation and see how the performance compares on a real workload in the future.

References

[1] Burton H Bloom. "Space/time trade-offs in hash coding with allowable errors". In: *Communications of the ACM* 13.7 (1970), pp. 422–426.

[2] Dario Catalano and Dario Fiore. "Vector commitments and their applications". In: *Public-Key Cryptography–PKC 2013: 16th International Conference on Practice and Theory in Public-Key Cryptography, Nara, Japan, February 26–March 1, 2013. Proceedings 16*. Springer. 2013, pp. 55–72.

[3] *Certificate Transparency : Certificate Transparency — certificate.transparency.dev*. `https://certificate.transparency.dev/`. [Accessed 08-May-2023].

[4] Rasmus Dahlberg, Tobias Pulls, and Roel Peeters. "Efficient sparse merkle trees: Caching strategies and secure (non-) membership proofs". In: *Secure IT Systems: 21st Nordic Conference, NordSec 2016, Oulu, Finland, November 2-4, 2016. Proceedings 21*. Springer. 2016, pp. 199–215.

[5] Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. "Constant-size commitments to polynomials and their applications". In: *Advances in Cryptology-ASIACRYPT 2010: 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings 16*. Springer. 2010, pp. 177–194.

[6] John Kuszmaul. “Verkle trees”. In: *Verkle Trees* 1 (2019).

[7] *KZG polynomial commitments — dankradfeist.de*. `https://dankradfeist.de/ethereum/2020/06/16/kate-polynomial-commitments.html`. [Accessed 07-May-2023].

[8] Ralph C Merkle. “A digital signature based on a conventional encryption function”. In: *Advances in Cryptology—CRYPTO’87: Proceedings 7*. Springer. 1988, pp. 369–378.

[9] *Merkle Patricia Trie | ethereum.org — ethereum.org*. `https://ethereum.org/en/developers/docs/data-structures-and-encoding/patricia-merkle-trie/`. [Accessed 25-Mar-2023].

[10] *Merkling in Ethereum | Ethereum Foundation Blog — blog.ethereum.org*. `https://blog.ethereum.org/2015/11/15/merkling-in-ethereum`. [Accessed 04-May-2023].

[11] *Radix tree - Wikipedia — en.wikipedia.org*. `https://en.wikipedia.org/wiki/Radix_tree`. [Accessed 04-May-2023].

[12] Lum Ramabaja and Arber Avdullahu. “The Bloom Tree”. In: *arXiv preprint arXiv:2002.03057* (2020).

[13] Teknikal$_{D}$*omain. Merkle trees in Git and Bitcoin — initialcommit.com*. `https://initialcommit.com/blog/git-bitcoin-merkle-tree`. [Accessed 08-May-2023].