DTU Compute

Verkle tries for efficient data verification in peer-to-peer networks

Victor Brevig (s184469)

Christian Rosenkilde Husted Kjær (s183943)

Contents

1 Introduction

Communication between nodes in a peer-to-peer network often relies on some way to verify that the data received is correct. This could for example be verifying that files downloaded using Git are correct or if specific parts of a blockchain state are correct. Precisely, the problem considered in this paper consists of a *prover* and a *verifier*, which are two different participants in a network. The prover has access to all data in full, and the verifier wishes to verify that some part of this data is correct according to what is stored with the prover. Thus, the verifier requests proof for some specific data from the verifier, who in turn responds with a proof. The verifier is allowed to store small constant-sized data that together with the proof can verify the validity of what is requested. This paper explores different solutions to this problem and how they are placed in the trade-off spectrum between proof size and proof construction time. This spectrum is representative of a more general spectrum between network communication overhead and computational overhead.

The most widely used solution to the problem stated above is using merkle trees. If a dynamic data structure is needed as in the example of storing a blockchain state, a merkle *trie* can be used. Both are tree data structures based on hashing which follows the simple recursive rule: an inner node contains the hash of its children, and the hash of data chunks are stored in the leaves. Here, a verifier only needs to store the root value, and a proof consists of all sister nodes on the root-to-leaf path. Since merkle tries are based on hashing, these are simple and are very fast to prove and validate. The downside of this approach is that it requires very large proof sizes for large trie sizes.
To address the large proof sizes of merkle tries, verkle trees were introduced by John Kuszmaul in 2018 [1]. Verkle trees, and therefore also verkle tries, utilize vector commitments instead of regular hashing to be able to prove a child-parent relationship between two nodes in the trie. This approach allows the prover to only send the nodes on the root-to-leaf path along with a small proof for each. However, this does not come without a cost, as computing the vector commitments requires significantly higher computational resources.
In a more recent development and with the Ethereum blockchain in mind, Dankrad Feist showed how to reduce the proof size and construction time for proving multiple values at once by using verkle tries with multi-proofs using random evaluation [2]. This scheme is specific to using polynomial commitments as vector commitments and bundles multiple commitments into a single proof.

This paper serves to give a detailed insight into how these three schemes work as well as theoretically accessing their strengths in regards to proof size versus proof construction time. Furthermore, these are tested empirically. Lastly, a discussion regarding the test results is presented as well as a discussion about the benefits of implementing these schemes into different applications.

2 Merkle tree

A merkle tree is a hash-based data structure in which every leaf node in the tree is a hash of some block of data, and every non-leaf node is a hash of its children [3]. Merkle trees are commonly used in distributed systems and peer-to-peer networks for efficient data verification/membership proofs and are used in peer-to-peer networks such as Git, Tor, Bitcoin, Ethereum, and many more. The efficiency lies in the use of hashes instead of the full data blocks, as a hash is much smaller in size. For example, the SHA-256 hashing algorithm can take an enormous amount of data and output a 256-bit fingerprint. As merkle trees are used to verify any stored data, they can help ensure that data received from other peers in a peer-to-peer network has not been tampered with and that the data received is the expected data.

A conceptual illustration of a merkle tree with branching factor 2 can be seen in figure 2.1.

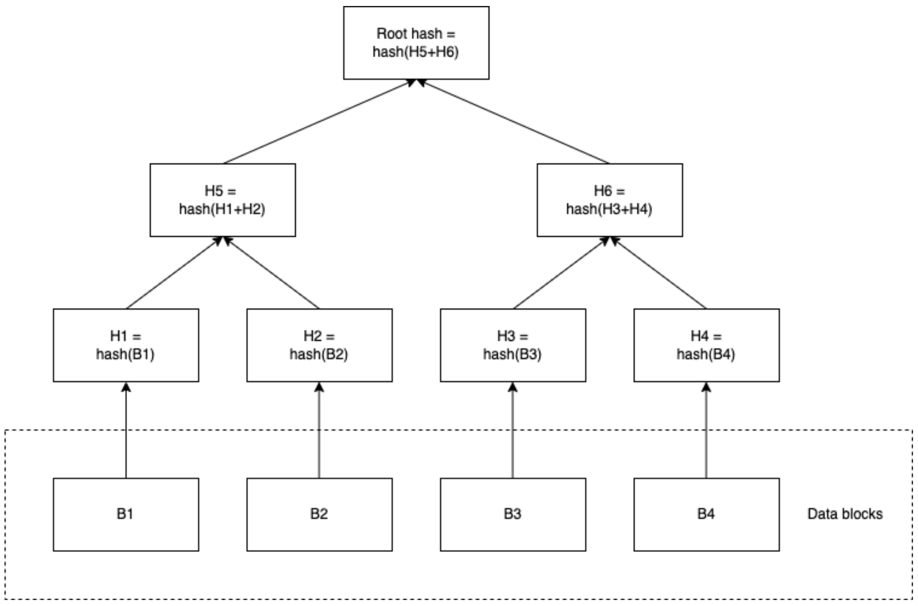

Figure 2.1: Binary Merkle tree example

For constructing a merkle tree, the data is split into blocks. Each of these blocks is hashed using some collision-resistant hash function, for example, SHA-256. Each pair of nodes are then continuously hashed in a bottom-up fashion until the root is reached. The root thus contains a hash to which every leaf node in the tree has contributed. Changing a single bit in any of the data blocks will result in a completely different root hash. This root hash is stored by all participants and is used for verification.
To understand how a merkle tree can be used for proof and verification, one can consider the following example: Alice stores some data for Bob, say B1 through B4 from figure 2.1, and Bob wishes to retrieve the data from B1 stored with Alice. Bob can query Alice for B1 and Alice would alongside B1 send a proof consisting of the hashes of the sister nodes on the leaf-to-root path. That is, Alice would send B1, H2, and H6. Bob can then verify that the B1 received is indeed the B1 he once upon a time stored with Alice, by hashing the hash of B1 appended to H2 and then hashing this hash appended to H6 resulting in

a final hash. Bob can then compare this hash to the root hash stored locally, if they are equal, then Bob can be certain that B1 is indeed not tampered with.

A merkle tree can have an arbitrary branching factor. For a balanced binary merkle tree, the depth is $\mathcal{O}(\log_2(n))$. In this case, there is only one sister node for each height in the root-to-leaf path, so the proof size is also $\mathcal{O}(\log_2(n))$. For a merkle tree with branching factor k, namely a k-ary tree, the proof size becomes $\mathcal{O}(k \cdot \log_k(n))$, since the height is $\mathcal{O}(log_k(n))$ and there are k sister nodes for each node on the root-to-leaf path. Thus asymptotically, the optimal proof size is achieved with a branching factor $k = 2$, resulting in a binary merkle tree. Inserting, deleting, or updating an element from the merkle tree requires one to update all the hashes on the root-to-leaf path by rehashing all the sister nodes on this path. This takes $\mathcal{O}(k \cdot \log_k(n))$ time assuming computing the a hash takes $\mathcal{O}(1)$ time. This is a fair assumption, since all the inner nodes are hashes themselves which have a constant size. As such, it is clear that the optimal branching factor is $k = 2$ with regards to both update times, proof size, and proof construction time. Even for binary merkle trees, a proof size of $\mathcal{O}(\log_2(n))$ can become quite large for large values of n. This is especially true for network bandwidth-constrained applications.

2.0.1 Merkle trie

For many applications, it is convenient to maintain the data wished to prove in a linear space dynamic data structure that supports lookup, insertion and deletion operations efficiently. A compact trie, also sometimes referred to as a Patricia trie, achieves exactly this while keeping the tree property. In general, a trie can store key-value pairs, where the values are stored in the leaves. Two elements share their path maximally on the longest common prefix of their two corresponding keys. A compact trie reduces the size of the tree by concatenating paths of single nodes in the tree into one edge [4]. The time to search for and insert/delete an element with a key of length m is $\mathcal{O}(m)$. In the following, it is assumed that the key has a fixed standardized length (e.g. 256 bits). Since the tree is no longer balanced, the root-to-leaf path is no longer $\mathcal{O}(\log_k(n))$ but is bounded by the fixed key length. However, this is usually not a problem since e.g. 256 bits is large enough for any application in practice. When inserting or deleting an element from a merkle trie, one has to perform the standard operations to maintain the trie property, but it is also necessary to rehash all sister nodes on the root-to-leaf path to maintain the correct hashes.

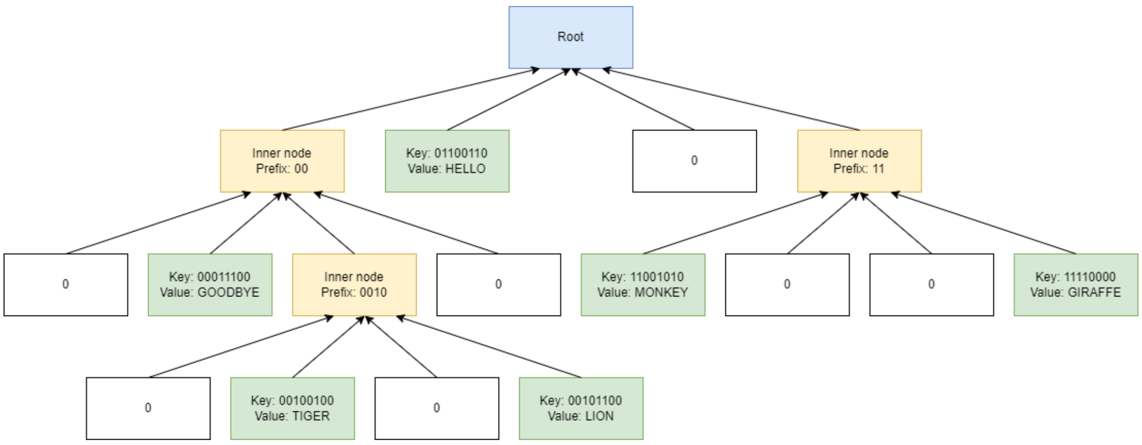

Figure 2.2: 4-ary Merkle trie with 6 inserted key-value pairs

Figure 2.2 shows an example of a merkle trie with branching factor 4 containing 6 key-value pairs. Note that for merkle tries, inner nodes are introduced. Inner nodes store the longest common prefix of its children and also stores the combined hash of all its children. To explain proof and verification, consider again an example with Alice and Bob. Perhaps Alice had stored the 6 key-value pairs from figure 2.2 for Bob, and Bob later wished to retrieve the value with key 00101100. Bob queries Alice for this data, and Alice would alongside the key-value pair send all sister nodes on the leaf-to-root path. These sister nodes are marked in red in figure 2.3.

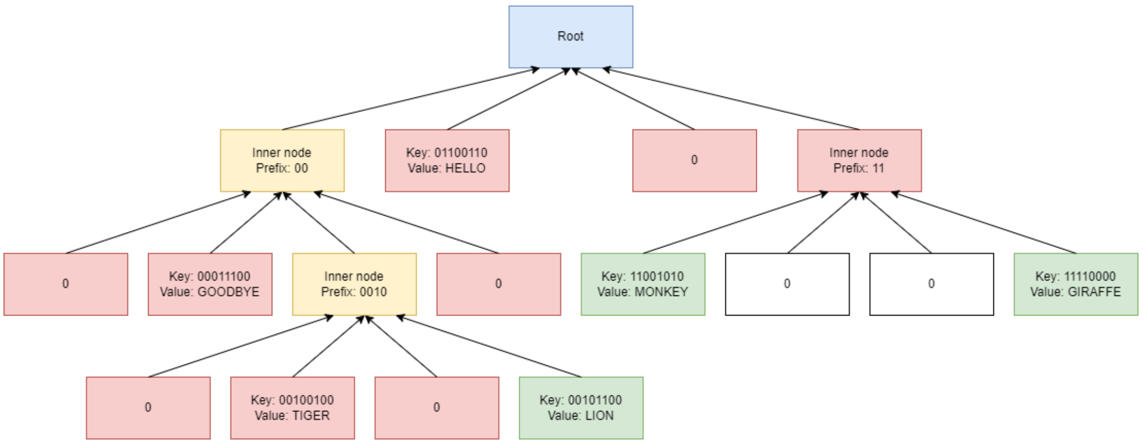

Figure 2.3: Example of sister nodes to include in proof of leaf with key 00101100

To verify Bob received the correct key-value pair, Bob would concatenate the key-value with its sister nodes from this layer and hash it. This hash value would be concatenated with the sister nodes from the next layer, which would also be hashed. This is continued until the root is reached, at which point Bob can compare the local root hash he has stored with that he has computed.

3 Verkle tries

As seen in merkle tries, constructing a proof for a key-value pair in the trie requires all of the sister nodes in the root-to-leaf path to the leaf containing the value. For large tries, this can become a lot. The idea behind verkle tries is to create proofs of relations between a node and its children. Thus, for any internal node, a verifier should be able to verify that a specific child is *actually* a child of that node in the trie. If this property is obtainable, then the verifier only needs the nodes on the root-to-leaf path plus the proofs linking them together to verify a key-value pair. The said property can be obtained through vector commitments. A vector commitment is a special type of hash function that takes a list of inputs and computes a commitment $h(x_1, x_2, ..., x_n) \rightarrow C$, with the requirement that given C and an element of the list x_i, there exists a short proof that C is, in fact, a commitment to a list where the $i'th$ value is x_i. Many types of vector commitment schemes exist, but this paper only investigates the use of polynomial commitments as vector commitments, since these are the simplest and most effective vector commitments [3]. In this study, KZG commitments are used as they are one of the simplest polynomial commitment schemes to use, but other schemes can also be applied. The idea is to compute and store a commitment for each parent and their children in the tree. Specifically, each parent node stores a commitment, such that together with a proof, a verifier can check if a given child is a child of that parent at a specific position.

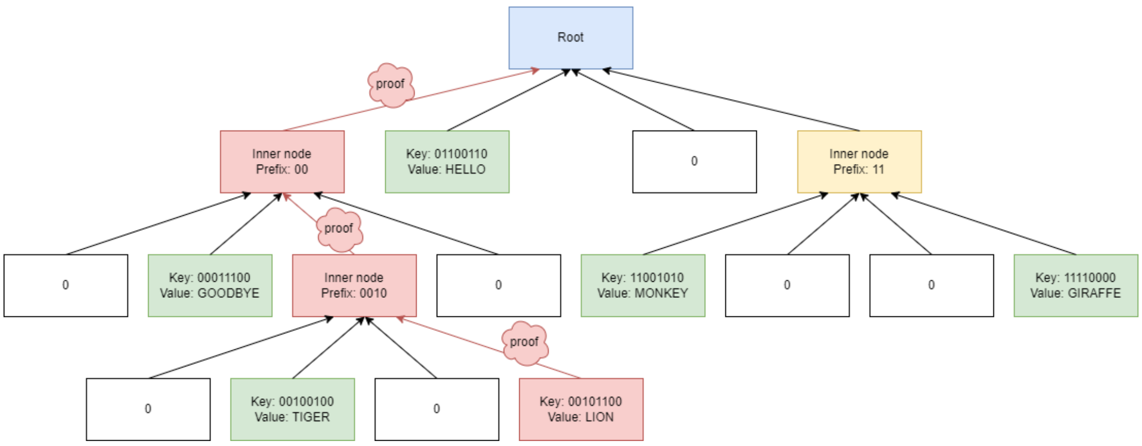

Figure 3.1: 4-ary Verkle trie with 6 inserted key-value pairs with proof example of leaf with key 00101100

Figure 3.1 shows an example of a verkle trie with a branching factor 4 containing 6 key-value pairs. The structure itself is almost identical to that of merkle tries, the only difference being that each internal node stores a vector commitment covering all of its children instead of a combined hash of its children. This way, the prover only has to send the commitments stored in the internal nodes on the root-to-leaf path of the leaf wished to prove, as well as the proofs for the parent-child relations on this path. Figure 3.1 shows an example of the necessary data that is needed for a verifier to prove the key-value pair $(00101100,$"LION"$)$. These are the shapes colored red. The verifier only needs to store the commitment of the root node to check against.

3.0.1 Elliptic curve math

Before diving into KZG commitments, this is a very brief introduction and overview of elliptic curves and the operations used in the following sections. An elliptic curve is a set of points satisfying the equation $y^2 = x^3 + ax + b$ for some constants a and b [5]. In practice prime fields are used $\mathbb{F}_p$, meaning everything is done modulo a prime number p.

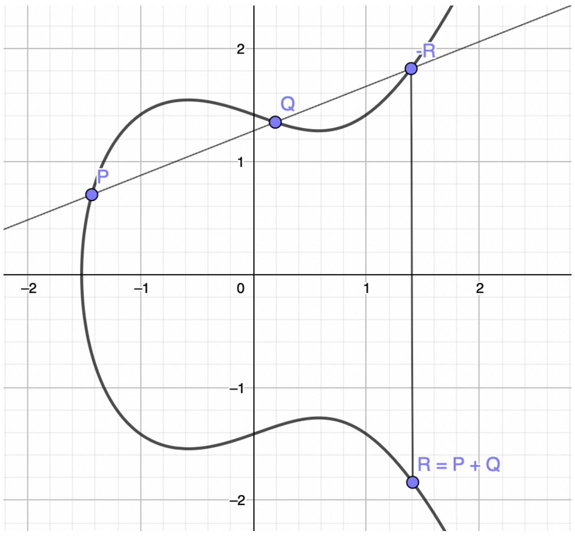

Figure 3.2: Example of elliptic curve with equation $y^2 = x^3 - x + 2$.

Two points P and Q can be added to get a new point on the curve $R = P + Q$ by extending the line between P and Q until the curve is hit again, and then taking the negative y value. An example of this is seen in figure 3.2. Similar to the 0 number regarding addition of integers, there exists a point of infinity O, such that $P + O = P$. Another operation is multiplication between a point and a constant n. This is the same as adding P to itself n times:

$$P \cdot n = \sum_{i=1}^{n} P \tag{3.1}$$

An elliptic curve has an *order*, which is a number n, such that $P \cdot n = O$ for any point P. Finally, a generator point G is a standardized point that in an analogous way represents 1 [6].
One can commit to a number p by calculating $[p] = p \cdot G$, which is a point on the curve. $[p]$ is a field element, i.e. $[p] \in \mathbb{F}_p$. From only knowing $[p]$, it is computationally infeasible to find p, as it is the same as solving the discrete logarithm problem. Effectively, this works as a one-way encryption scheme [6] [7].
Elliptic curves are homomorphic under addition, meaning that $[a] + [b] = [a + b]$. The same is *not* true for multiplication, i.e. $[a] \cdot [b] \neq [a \cdot b]$. Instead, one can use elliptic curve *pairings* [8]. In general, a pairing can be any mathematical operation e on two elements with the constraint that it must satisfy bi-linearity [6], meaning that the following must hold [6]:

$$\begin{aligned} e(P, Q + R) &= e(P, Q) \cdot e(P, R) \\ e(P + Q, R) &= e(P, R) \cdot e(Q, R) \end{aligned} \tag{3.2}$$

In the context of using pairings to perform multiplication of two field elements, this can be done if they are committed to in different curves and output on a third curve. If the first element is committed to on a curve $\mathbb{G}_1$ and the second element is committed to on another curve $\mathbb{G}_2$, then the pairing $\mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ outputs the multiplication commited to on a third curve $\mathbb{G}_T$ [8]:

$$e([a]_1, [b]_2) = e(G, H)^{ab} = [ab]_T \tag{3.3}$$

This section only briefly introduced some key concepts that are applied in KZG commitments.

3.0.2 KZG Commitments

KZG commitments, also referred to as Kate commitments, are polynomial commitments that allow a prover to compute a commitment that can be opened at any position, meaning that the value of the polynomial at a given position is equal to a claimed value. That is, for a polynomial $p(X)$, the verifier should be able to verify that $p(z) = y$ for some point (z, y) using a constant size proof. The trick behind KZG commitments is to look at the polynomial $p(X) - y$. If $p(z) = y$, it is certain that z is a root. A polynomial that has a root in z is also divisible by $X - z$ [8]. Thus, the polynomial remainder theorem can be utilized to take out the linear factor:

$$p(X) - y = (X - z) \cdot q(X) \tag{3.4}$$

This polynomial is clearly zero at $X = z$. The verifier can know that if y is indeed the correct evaluation of $p(z)$, then the polynomial $q(X)$ must exist such that equation 3.4 holds. For a fraudulent y', z would *not* be a root of $p(X) - y'$, and therefore it is not possible to take out the linear factor as seen in equation 3.4. Another way to look at it is that $p(X) - y$ cannot be divided by $(X - z)$ cleanly without a fractional remainder [3]. Thus, the equation will not hold when the verifier checks it. This is the crucial part of KZG commitments, namely to check that equation 3.4 holds given the verifier's own y value.
Because the verifier can only store something small of constant size and the proof should be of constant size, sending the whole polynomials $p(X)$ and $q(X)$ does not work. Luckily, the Schwartz-Zippel lemma says that to check if equation 3.4 holds, it is sufficient to check that it holds when evaluated in some random number $X = s$ [9].

$$p(s) - y = (s - z) \cdot q(s) \tag{3.5}$$

Neither the prover nor the verifier must know the number s, as the verifier must be confident that this is truly a random number that the prover has not tampered with. To keep s a secret, only a commitment to s on an elliptic curve is stored, namely $[s] \in \mathbb{F}_p$, where $\mathbb{F}_p$ is a prime field. To perform computation on elliptic curves, the two elliptic curves $\mathbb{G}_1$ and $\mathbb{G}_2$ are defined with the order p. Let G and H be generator points of $\mathbb{G}_1$ and $\mathbb{G}_2$. Thus for a number $x \in \mathbb{F}_p$, the group elements on the two curves are:

$$[x]_1 = xG \in \mathbb{G}_1 \text{ and } [x]_2 = xH \in \mathbb{G}_2 \tag{3.6}$$

It is now assumed that both prover and verifier have the group elements $[s]_1^i$ and $[s]_2^i$ for $i = 0, 1, \ldots, k - 1$. Computation and secure distribution of these are out of the scope of this paper. Note that as seen in section 3.0.1, s cannot be restored from $[s]_1^i$ and $[s]_2^i$. The number k is the number of points committed to. One can evaluate a polynomial at the secret point without knowing the secret as follows:

$$[p(s)]_1 = \left[\sum_{i=0}^{n} p_i s^i\right]_1 = \sum_{i=0}^{n} p_i \left[s^i\right]_1 \tag{3.7}$$

Here p_i are the coefficients of the polynomial p [8].

Because the verifier only has commitments to the secret number s, it is necessary to perform arithmetic on elliptic curve elements instead of normal integers. The left hand side of 3.4 as well as $s - z$ can easily be computed as field elements, since elliptic curves are additively homomorphic. However, multiplication is required on the right hand side. Since elliptic curves are not multiplicative homomorphic as mentioned in section 3.0.1, a solution is to use the $\mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ pairing [8]:

$$e([a]_1, [b_2]) = e(G, H)^{ab} = [ab]_T \tag{3.8}$$

Now, every operation needed to compute 3.4 as field elements is available. The equation then becomes:

$$[q(s) \cdot (s - z)]_T = [p(s) - y]_T \tag{3.9}$$

An typical interaction between a prover and a verifier goes as follows: The verifier stores the commitment $C = [p(s)]_1$ and wants to validate that $p(z) = y$. It sends z to the prover. Given z, the prover can calculate the proof $\pi = [q(s)_1]$ with its own y value corresponding to that z. The prover sends back the proof π with which the verifier can check if the following equation holds true (note that this is equivalent to equation 3.9):

$$e\left(\pi, [s - z]_2\right) = e\left(C - [y]_1, H\right) \tag{3.10}$$

The verifier checks equation 3.10 using the original y it wanted to check (this was never shown to the prover). If the equation holds, the verifier can be confident that indeed $p(z) = y$. The equation 3.10 is equivalent to equation 3.9 as a direct application of equation 3.8 (remember that the generator H works as the identity for points in $\mathbb{G}_2$ with regards to multiplication). Equation 3.9 is in turn just equation 3.4 evaluated on elliptic curve elements. Thus, the proof is correct.

3.0.3 Proving multiple values

So far, it has been shown how to prove the evaluation of a polynomial in a single point. This can be extended to proving the evaluation of a single polynomial in multiple points, while still maintaining a constant-sized proof. In this case, a verifier might wish to prove the evaluation of a polynomial for k points $(z_0, y_0), (z_1, y_1), \ldots, (z_{k-1}, y_{k-1})$. If this is true, the polynomial $p(X)$ must go through these points. It is possible to create another polynomial that also goes through all of these points using Lagrange interpolation:

$$I(X) = \sum_{i=0}^{k-1} y_i \prod_{\substack{j=0 \\ j \neq i}}^{k-1} \frac{X - z_j}{z_i - z_j} \tag{3.11}$$

Note that $p(X)$ and $I(X)$ can be very different polynomials. The degree of $p(X)$ is at least as large as the degree of $I(X)$, so $p(X)$ can be much larger. The only requirement is that both go through the k points. Since both $p(X)$ and $I(X)$ goes through the list of points, it is known that the polynomial $p(X) - I(X)$ has roots in $z_0, z_1, \ldots, z_{k-1}$. This is because both polynomials are equal to $y_0, y_1, \ldots, y_{k-1}$ at these values, so they will cancel out. Since $z_0, z_1, \ldots, z_{k-1}$ are roots, it is possible to take out linear factors:

$$p(X) - I(X) = (X - z_0)(X - z_1) \ldots (X - z_{k-1}) \cdot q(X) \tag{3.12}$$

For convenience, the zero polynomial $Z(X)$ is defined:

$$Z(X) = (X - z_0)(X - z_1) \ldots (X - z_{k-1}) \tag{3.13}$$

Using $Z(X)$, equation 3.12 can be rewritten as:

$$q(X) = \frac{p(X) - I(X)}{Z(X)} \tag{3.14}$$

Similar to a single-point-proof, the proof becomes $\pi = [q(s)]_1$ at the secret point s. Again, pairings are needed to deal with multiplication of field numbers. Thus, the verifier needs to check the following equation [8]:

$$e\left(\pi, [Z(s)]_2\right) = e\left(C - [I(s)]_1, H\right) \tag{3.15}$$

Which is equivalent to:

$$[q(s) \cdot Z(s)]_T = [p(s) - I(s)]_T \tag{3.16}$$

If the prover tries to trick the verifier into believing that $p(z_i) = y_i'$ for some i in the list, z_i would no longer be a root of $p(X) - I(X)$, and thus it is not possible to divide by $Z(X)$. In that case, equation 3.12 will not hold.

3.0.4 Application of KZG commitments in verkle tries

As previously stated, the motivation for using KZG commitments in verkle tries is that it enables the verifier to prove that an element is the ith child of an inner node. To obtain this, a KZG commitment C is stored in every inner node. The polynomial committed should go through the points $(0, y_0), (1, y_1), \ldots, (k-1, y_{k-1})$, where k is the branching factor. Note that the z-values are the indices, i.e. the position of the child node, and y_i is the child data for child i. For an inner node, y_i will be the commitment stored in that node. For a leaf node, the value in the key-value pair is used as y_i. Lagrange interpolation can be used to find a polynomial $p(X)$ that goes through all these points:

$$p(X) = \sum_{i=0}^{k-1} a_i \prod_{\substack{j=0 \\ j \neq i}}^{k-1} \frac{X - j}{i - j} \tag{3.17}$$

The polynomial commitments stored in each inner node must be maintained. When an insertion, deletion, or update of a key-value pair happens, one must update all commitments on the root-to-leaf path. This is similar to re-hashing in merkle trees.

When verifying a key-value pair, the verifier sends the key to the prover which collects all commitments on the root-to-leaf path. Along with the commitments, the prover computes a proof π for each, based on the index and values on the path searching down to the leaf. The root-to-leaf path is identified by performing a normal trie search for the key. Returning to figure 3.1, the prover must prove the following relations for the key-value pair $(00101100, \text{"LION"})$:

(3.18)

$$\begin{aligned} \pi_0 \text{ for } p_0(11) &= H(\text{"}LION\text{"}), \text{ where } C_0 = [p_0(s)]_1 \\ \pi_1 \text{ for } p_1(10) &= H(C_0), \text{ where } C_1 = [p_1(s)]_1 \\ \pi_2 \text{ for } p_2(00) &= H(C_1), \text{ where } C_2 = [p_2(s)]_1 \end{aligned} \tag{3.19}$$

Here, H is a collision-resistant hash function. Note that in this case, the z values are the indices represented in binary, i.e. 2 bits for a branching factor of 4. With these proofs and commitments, the verifier can go through the same chain, but initially using the value that must be proven, here ”LION”. If the chain of proofs is valid and the last commitment C_2 matches the root commitment stored locally with the verifier and the verifier is satisfied. If on the other hand, the value for the key 00101100 is actually something else, e.g. ”LIONS” or if any other key-value pair in the verkle trie is different from what is expected, then somewhere along the chain of proofs, one of them will be false.

4 Verkle trie with multi-proofs using random evaluation

It is possible to extend verkle proofs even further. While the structure of the verkle trie remains the same, it is possible to provide a single KZG proof that can prove many leaves at the same time. That is, instead of requiring one proof for each commitment along the path of a root-to-leaf for a single key-value pair, by utilizing the additively homomorphic property of polynomial commitment schemes such as KZG, a single fixed-size KZG proof can prove all parent-child links between commitments along the root-to-leaf paths for an unlimited number of key-value pairs [2].

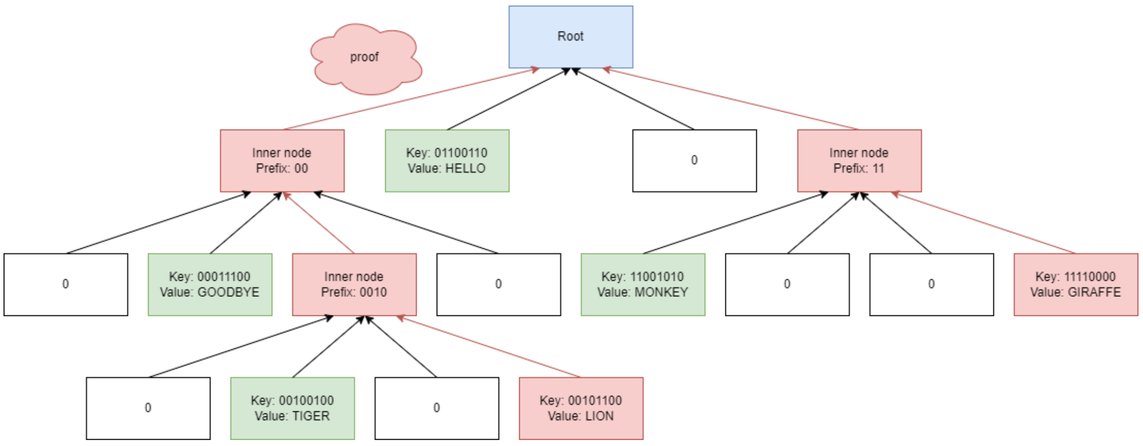

Figure 4.1: 4-ary verkle trie with multiproof with 6 inserted key-value pairs

In figure 4.1, an example of this can be seen, coloring the data that needs to be sent to the verifier for the key-value pair $(00101100, \text{"LION"})$ in red. All commitments along the paths from root-to-leaf as well as the key-value pairs still need to be provided, but now only a single KZG proof is needed, instead of a KZG proof for each link. Furthermore, there are additional savings on proof size to be had if any two leaves share any inner nodes on the path, as these now only need to be provided once.
As with the verkle tries previously described, for every inner node on the root-to-leaf path, one should be able to verify it's children at specific positions. When proving multiple values, there are more linkages, i.e. root-to-leaf chains of KZG proofs. However, it is still just sequences of proving statements of the form "node X is actually a child of node Y at position i". If there are m of such relations to prove, there will be m KZG commitments $C_0 = [p_0(s)]_1, C_1 = [p_1(s)]_1, \ldots, C_{m-1} = [p_{m-1}(s)]_1$ and m points $(z_0, y_0), \ldots, (z_{m-1}, y_{m-1})$. From these, one must prove the evaluations $p_0(z_0) = y_0, \ldots, p_{m-1}(z_{m-1}) = y_{m-1}$. Collecting these proofs into a single proof is exactly the core aspect of verkle tries with multi-proofs.

4.0.1 The proof

In order to construct the proof, a hash of all the commitments along with the points on the root-to-leaf paths is computed [2]:

$$r = H\left(C_0, \ldots, C_{m-1}, y_0, \ldots, y_{m-1}, z_0, \ldots, z_{m-1}\right) \tag{4.1}$$

Then, the prover can compute a polynomial $g(X)$:

$$g(X) = r^0 \frac{p_0(X) - y_0}{X - z_0} + r^1 \frac{p_1(X) - y_1}{X - z_1} + \ldots + r^{m-1} \frac{p_{m-1}(X) - y_{m-1}}{X - z_{m-1}} \tag{4.2}$$

The goal is to show that $g(X)$ is a polynomial and not a rational function. If it is the case that $g(X)$ is a rational function, then at least one of the quotients is *not* an exact divisor. As seen for the single KZG proof for $p(z) = y$ in section 3.0.2, the quotient polynomial $q(X) = \frac{p(X)-y}{(X-z)}$ is only a polynomial if z is a root of $p(X) - y$. The same concept goes for $g(X)$. Here, it is only a linear combination of quotients.
Because it is a random linear combination of quotients, a scenario could happen where two of them cancel out their remainders to give a polynomial. This is why r is introduced. As a consequence of the Fiat-Shamir heuristic, because r is chosen after all the inputs are fixed, it is computationally impossible for the prover to find inputs such that two remainders cancel [2]. Therefore, proving that $g(X)$ is a polynomial is sufficient.
Because any function that can be committed to through a KZG commitment *must* a polynomial, the prover simply computes and sends the commitment $D = [g(s)]_1$. The remaining part of the proof consists of convincing the verifier that D is indeed a commitment to the function $g(X)$. If it is indeed a commitment to $g(X)$, then $g(X)$ must be a polynomial and the verifier is satisfied. The correctness of D will be proved by evaluating it at a completely random point t and by helping the verifier check that the evaluation is indeed $g(t)$. Let $t = H(r, D)$. The polynomial $g(X)$ evaluated at t looks as follows [2]:

$$g(t) = \sum_{i=0}^{m-1} r^i \frac{p_i(t) - y_i}{t - z_i} \tag{4.3}$$

This equation can be split into two sums:

$$g(t) = \underbrace{\sum_{i=0}^{m-1} r^i \frac{p_i(t)}{t - z_i}}_{g_1(t)} - \underbrace{\sum_{i=0}^{m-1} r^i \frac{y_i}{t - z_i}}_{g_2(t)} \tag{4.4}$$

Here, the second term $g_2(t)$ can be computed by the verifier. This is because the verifier is sent all the commitments and (z_i, y_i) pairs on the root-to-leaf paths of the key-value pairs proven. Thus the verifier can compute r and thereafter $g_2(t)$. In these calculations, the verifier must input its own keys corresponding to the leaf values the verifier wants to prove.
The verifier does not have the polynomial coefficients required to evaluate $p_i(t)$. However, the verifier *does* have all the commitments C_i to the polynomials evaluated in s. The first term of $g(t)$ is the following polynomial:

$$h(X) = \sum_{i=0}^{m-1} r^i \frac{p_i(X)}{t - z_i} \tag{4.5}$$

Using the commitments, the verifier can compute a commitment to the polynomial $h(s)$:

$$E = [h(s)]_1 = \sum_{i=0}^{m-1} \frac{r^i}{t - z_i} C_i \tag{4.6}$$

An opening of the commitment E at t is exactly $g_1(t)$. The same commitment, E, can be computed by the prover. However, the prover can calculate $h(s)$ and then commit to it

instead of using the commitments as they have all the necessary inputs.
Now, $p_D(X)$ is defined as the polynomial committed to by D. If the prover behaves honestly, then this will be $g(X)$, which is exactly what the verifier wishes to confirm. Because commitments are $binding$, there can at most be one polynomial that the prover can open D at [2].
To conclude the proof, the verifier needs to check that $p_D(t) = h(t) - g_2(t)$ corresponding to equation 4.4. This can also be written as:

$$g_2(t) = h(t) - p_D(t) \tag{4.7}$$

As previously seen, the verifier can easily compute the left-hand side $y = g_2(t)$, however, the prover has to provide an opening to the commitment $E - D$ at t to prove that it is equal to y. This is the same as evaluating equation 4.7, but doing it over commitments. Finally, the KZG proof $\pi = [(h(s) - g(s) - y)/(s - t)]_1$ confirms that this is the case. In summary, the proof consists of D and π which is sent from the prover to the verifier [2].

4.0.2 Verification

To verify a proof, the verifier starts by computing both r and t. As explained above, the verifier can also compute the commitment E by equation 4.6. The verifier then computes $y = g_2(t)$ with it's own keys and checks if the following holds true:

$$e\left(E - D - [y]_1, [1]_2\right) = e\left(\pi, [s - t]_2\right) \tag{4.8}$$

Note that when substituting the definitions for E, D and π, and performing multiplication on the inputs too the pairings on the $\mathbb{G}_T$ curve, the left and right hand side checks out. If equation 4.8 holds, the verifier is convinced that when the commitment D is opened at a random point, the value is exactly equal to $g(t)$. The Schwartz-Zippel lemma states that it is impossible in practice to find that D opened at t has exactly the value of $g(t)$, unless D was a commitment to $g(X)$ [2]. Therefore, the verifier can trust that $g(X)$ must be a polynomial if equation 4.8 holds, and as such, the key-value pairs have been verified.

5 Implementation

Our implementation of Merkle Tree, Merkle Trie, Verkle Trie, and Verkle Trie with multi-proofs through random evaluation were written in typescript. We initially began our implementations in C++, however, this was abandoned once we were to implement the KZG commitments due to the lack of proper libraries to support it. Thus the absolute running times in the results section might have lower when implemented in a faster programming language. The implementation in typescript utilizes the libraries galois, rustbn, ffjavascript, and elliptic for their use of prime fields, BN128 curve operations, finite fields, and elliptic curve arithmetics respectively as well as the cryptography library for the use of SHA-256 hashes. We began by implementing binary merkle trees, followed by merkle tries. For merkle tries, we started by implementing a trie. In a trie, each leaf is represented by a key-value pair, where it is keyed based on the hash of the value of the leaf, which ensures that distinct leaves will have distinct keys due to the collision-resistant nature of SHA-256. Due to the trie being k-ary and not string-based, the hash used for the key was converted to a binary string, from where $\lfloor log_2(k) \rfloor$ bits were selected for matching on prefixes. As seen in figure 4.1, we are storing all k children. This was done in a fixed size array. If there is no child at some position, a NULL value is simply stored instead. Since we have the whole array, we could convert the $\lfloor log_2(k) \rfloor$ sized chunks of prefix bits to an integer and use that to index into the array of children. This is a different $\mathcal{O}(1)$ time solution for matching instead of using dictionaries as normally used when implementing a trie.
For the verkle trie, we started from the basic trie implementation and next implemented KZG polynomial commitments which we based on libkzg, a minimum-viable KZG polynomial commitment scheme implementation. For the verkle trie with multi-proofs through random evaluation, we started by implementing the necessary commitment scheme for the multi-proofs and applied that to the original trie. To make the tries dynamic, we implemented an insertion function which maintains the correct properties of the trie, both in the case of merkle and verkle tries. Furthermore, bottom-up recursive ”merkelize” and ”verkelize” functions were implemented to convert a normal compact trie to a merkle trie or verkle trie respectively, while only visiting each node once.
A ZIP-file with the source code is handed in with this paper. The code can be run using node.js. When in the correct folder, the typescript files can be run with the command ”npx ts-node FILENAME.ts”. The folder contains a separate folder for the implementation of the merkle tree, merkle trie, verkle trie and verkle trie with multi-proofs. Additionally, a folder called ”Globals” contains helper methods and most importantly the logic for creating KZG proofs in the ”KZG.ts” file. Finally, a folder named ”Tests” contains the files including the different tests conducted. These can be run to recreate the results found in the next section.

6 Experimental Evaluation and Discussion

6.0.1 Merkle trie experimental evaluation

For the first test, 100 key-value pairs were proven for a merkle trie containing 10000 key-value pairs. The results are shown in table 6.1.

Branching factor	Avg. proof size (bytes)	Avg. proof construction time (ms)
2	653.28	0.04
4	843.67	0.03
8	1216.72	0.01
16	1859.60	0.02
32	2824.40	0.03
64	4929.52	0.15
128	7404.40	0.16
256	11610.80	0.30
512	21213.44	0.55
1024	41298.48	2.57

Table 6.1: Proof size and proof construction time across different branching factors in a merkle trie for 100 key-values pairs proven and n = 10000

The proof size and construction times are averaged over 100 different key-value pairs proven. The reason for this is that the depth of a leaf in compact tries is depends on the number of other keys sharing some of the prefix with the leaf. Since the keys inserted in this trie are random (i.e. SHA-256 hashes), the depth, and therefore also the number of sister nodes required for a given key-value pair to be proven, has some randomness to it. Thus, taking the average of multiple proofs for different keys helps smoothing out this variance. The proof size was measured by adding up the byte sizes of the data sent from the prover to the verifier, for example 48 bytes per field element and 16 bytes for an integer.
The results in table 6.1 are mostly as expected, with the branching factor of $k = 2$ yielding the smallest proof size. The proof size increases quickly as the branching factor increases, also resulting in an increase of proof construction time. It was to be expected that the proof construction time would be lowest for a branching factor of $k = 2$ and not $k = 8$ as seen. However, at such small measures of time, it could be a result of predictive branching in the CPU or other background processes on the computer at compile time and thus small variation may occur. Proof verification times are similar to the proof construction time for merkle tries, as it is simply computing hashes up the root-to-leaf path and comparing that to the local root hash stored.

6.0.2 Verkle trie experimental evaluation

Similarly for the verkle trie, tests for both proof size and proof construction time were done for several branching factors. To further test the effect of proving multiple key-value pairs, the average of constructing 1, 10 and 100 proofs were measured respectively. In figure 6.1, results for a verkle trie containing 32768 key-value pairs are shown. As shown by the graphs in figure 6.1, neither the average proof size nor average proof construction

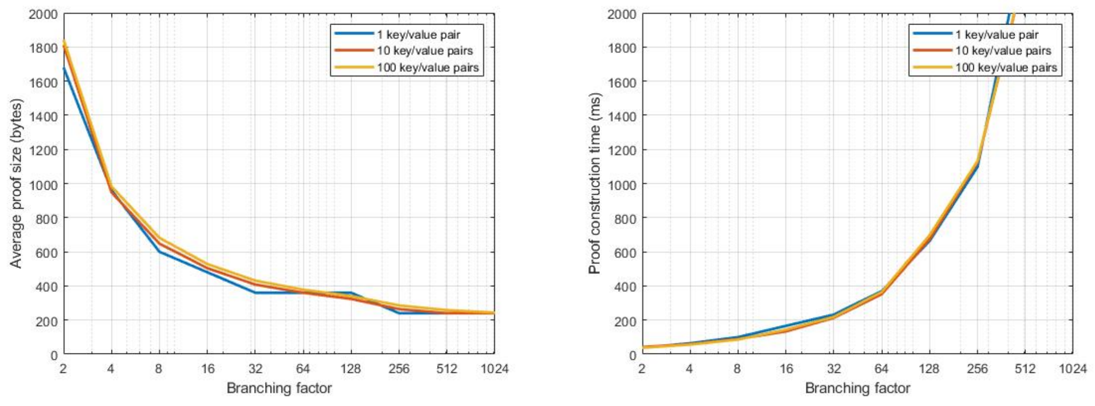

Figure 6.1: Average proof size and proof construction time across different number of key-value pairs proven and branching factors in a verkle trie with 32768 key-value pairs inserted

time seems affected by the number of key-value pairs proven. Proving multiple key-value pairs and taking the average only smooths out variance associated with the randomness of the key as previously explained. The non-existing effect of proving many key-value pairs at once was expected, since each key-value pair proven requires a disjoint set of commitments and proofs on the root-to-leaf path. This merely serves as a comparison to the verkle trie with multi-proofs. Thus, for the remaining experimental evaluations of verkle tries just 1 proven key-value pair will be used.
Figure 6.1 also quite clearly shows the trade-off between proof size and proof construction time that the branching factor provides. As the branching factor increases, the proof size decreases, and proof construction time increase. In fact, the proof construction time increases exponentially with the branching factor as can be shown by using a log chart on the y-axis as well. Although a smaller proof size is desired, the exponential growth of the proof construction time sets a limit for how wide the trie can get while maintaining an acceptable construction time. To further investigate the best branching factor, one has to look at how the size of the verkle trie affects this trade-off. In figure 6.2, results are shown for measuring an average proof size in bytes and proof construction time in milliseconds of proofs of 10 key-value pairs for different verkle trie sizes. The graphs in figure 6.2 are quite similar across the different trie sizes, indicating a low effect on growth of the proof size and construction time with regards to the branching factor. For all of the different trie sizes, the branching factor that best achieves a low proof size and a reasonable proof construction time seems to be around 32 to 64. Around this branching factor, the proof size seems to have almost flattened out, while the proof construction time remains less than half a second.

6.0.3 Verkle trie with multi-proof experimental evaluation

To test the verkle trie with multi-proof using random evaluation the same tests as for verkle tries were performed. As seen in figure 6.3, the amount of key-value pairs proven now greatly impacts both the average proof size and average proof construction time. The more pairs that are proven, the lower the average proof size and the lower the average proof construction time. The largest improvement of proven several pairs is however in terms of the proof construction time. The average time for constructing a proof for 100 pairs using a verkle trie of size 32768 with $k = 64$ takes around 360ms, whereas when using multi-proofs with the same setup the average construction time is lowered to an astounding 33ms, more than a factor 10 improvement. For $k = 512$, the normal verkle trie

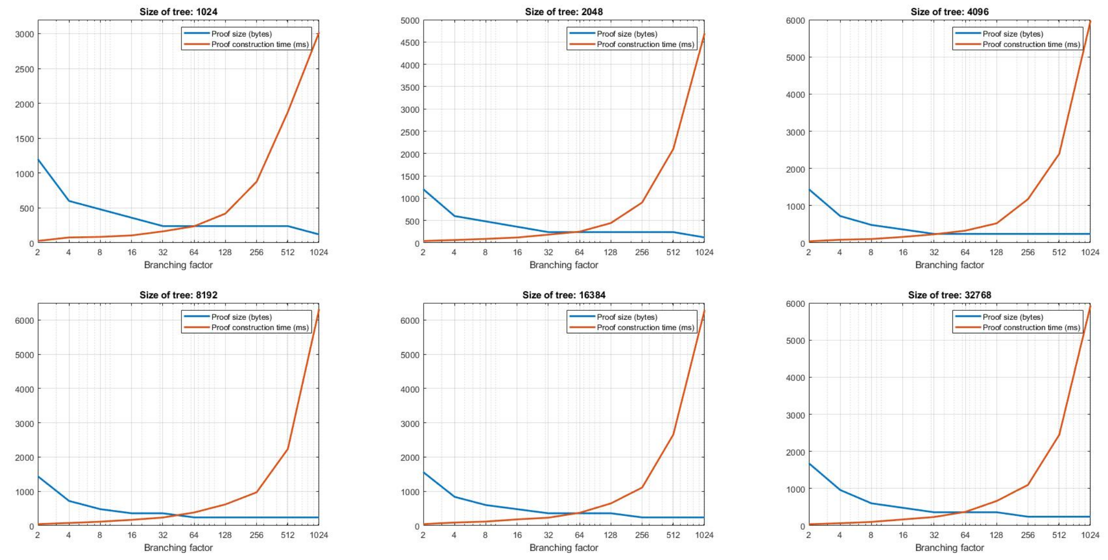

Figure 6.2: Proof size and construction time for different values of n.

sees an average construction time of 2266ms, whereas utilizing the multi-proofs sees an average construction time of just 153ms, almost an improvement of factor 15. The proof size meanwhile also decreases yet not quite as dramatically.

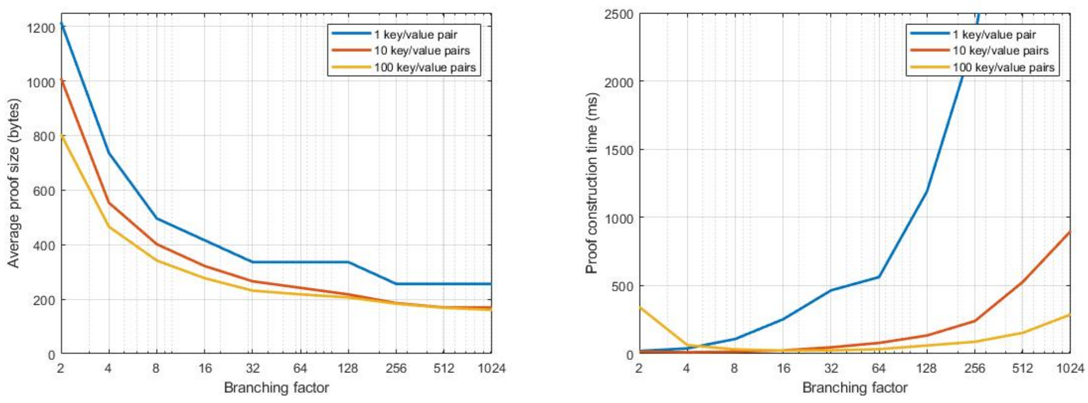

Figure 6.3: Average proof size and proof construction time across different number of key-value pairs proven and branching factors in a verkle trie with multi-proof with 32768 key-value pairs

The improvements of using multi-proofs is clear when proving several key-value pairs. However, when only proving a single key-value pair at a time, the use of multi-proofs cannot be justified as both the proof size and the construction time is greater than that of the normal verkle trie. The two graphs from figure 6.3 have been combined in figure 6.4 for a better overview.

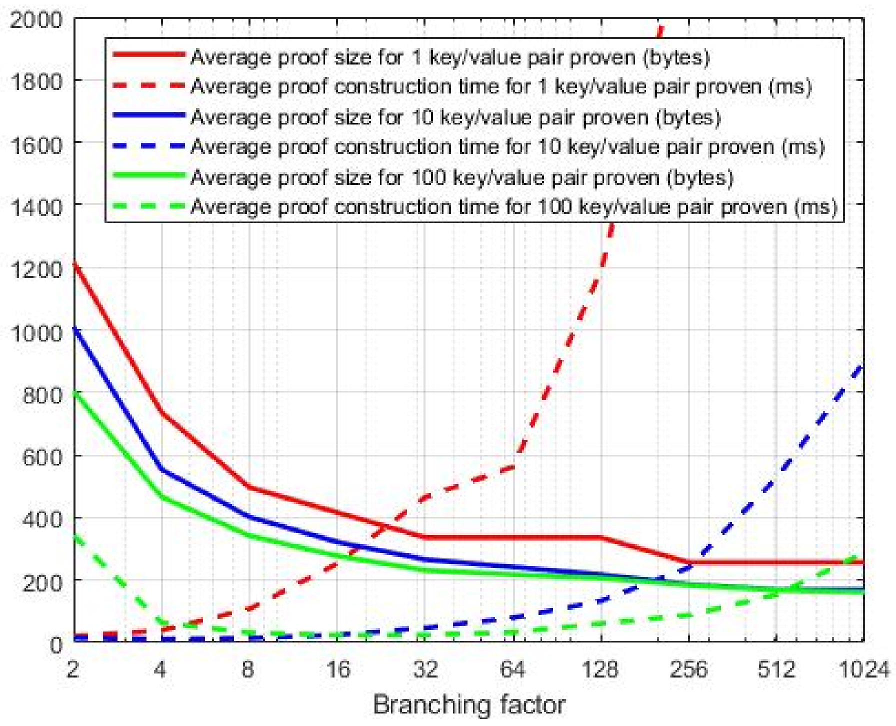

Figure 6.4: Average proof size and proof construction time across different number of key-value pairs proven and branching factors in a verkle trie with multi-proof with 32768 key-value pairs

Here, a reasonable branching factor is highly dependent on the number of key-value pairs proven. For 10 key-value pairs proven, a reasonable branching factor seems to be around $k = 128$ whereas for 100 key-value pairs proven, a branching factor of around 256 to 512 seems better. As the number of pairs increases, the best branching factor is likely to increase as well. As mentioned, proof verification is highly efficient and is therefore not considered for experimental evaluation.

6.0.4 Updates

As previously stated, both merkle tries needs to maintain correct hashes and verkle tries has to maintain correct commitments when a key/value pair is inserted, deleted or updated. There is no difference between verkle tries with and without multi-proofs, since the structure of the trie is the same. Insertion times for a merkle trie and verkle trie of different widths were tested. These tries has a fixed size of 16384, and the ”merkelize” and ”verkelize” functions described in section 5 were used to construct these tries fast. Since it is already known that the optimal branching factor for merkle tries is 2, it is not that interesting to investigate with regards to updates. However, for the verkle trie, especially with multi-proofs where the branching factor can range significantly, it is interesting to see how increasing the branching factor affects the updates of the trie. The following results for the verkle trie is an average time of 10.

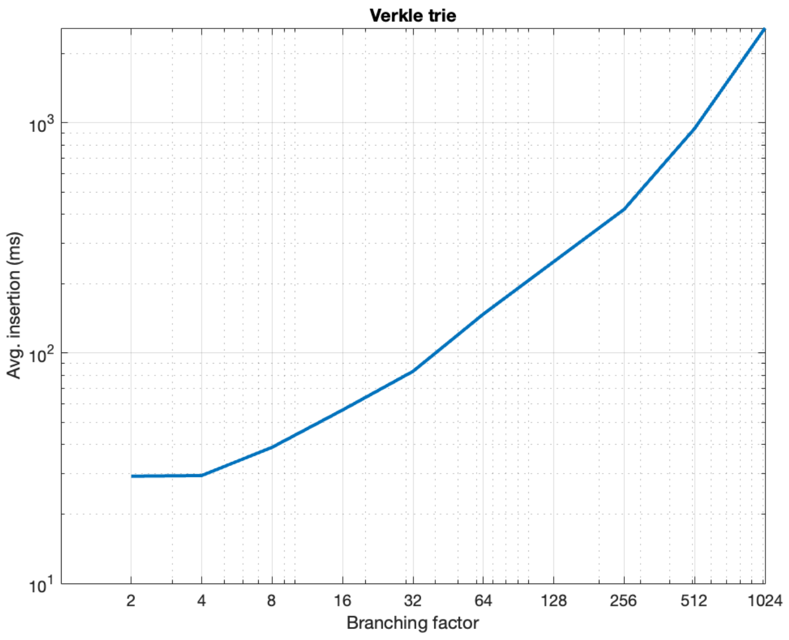

Figure 6.5: Double-logarithmic plot of average insertion time for different branching factors

Note that this test was run on a different computer than the others, so these results should not be compared to the other tests in absolute terms. Interestingly, the chart shows more than linear growth in a double-logarithmic plot indicating something larger than exponential growth. This speaks to the downside of verkle tries as well. Re-computing the polynomials and committing to them are much more computationally expensive than simply re-hashing the sister nodes. Together with the exponential growing proof construction time, this puts a limit to how wide the verkle tries can be in practice.

7 Discussion

The goal of using verkle tries instead of merkle tries is to make a shift in the trade-off space between network bandwidth in the form of proof size and computing power in the form of proof construction time. The results shown for the tests of the merkle trie in table 6.1 compared to that of the verkle trie implementations shows that the proof construction time is very small and the proof size is quite large in absolute terms. By the same token, the two versions of the verkle trie implemented both obtain very small proofs in the hundreds of bytes, but the proof construction time becomes quite large relatively. Depending on the application, one might seek different ends of this spectrum, although, with modern hardware, network bandwidth is often seen as a much scarcer resource than local computation.
Comparing the two verkle trie implementations, it was seen in figure 6.4 that the performance of the one utilizing multi-proofs increases as the number of key-value pairs proven increases. Especially, the proof construction time saw an effect of this. An application known to request many proofs at once may pick a high branching factor accordingly. For example, if an application on average requests proofs of at least 100 key-value pairs, a branching factor of 256 seems suitable. However, the designer of the application has to be sure that the application does not request batches of much fewer proofs, as the construction time becomes extremely large for those high branching factors. For the standard verkle trie not utilizing multi-proofs, the number of key-value pairs did not seem to affect the proof size and construction time as seen in figure 6.1. Similarly, the trie sizes did not seem to affect the best branching factor as seen in figure 6.2. For the standard verkle trie, the best branching factor seemed to be around 32 to 64, but can be shifted in either direction depending on the application requirements.

Datastructure	**Branching factor**	**Average proof size for (1 / 100) key-value pairs (bytes)**	**Average proof construction time for (1 / 100) key-value pairs (ms)**
Merkle Trie	2	672 / 734	0 / 0
Verkle Trie	32	360 / 430	232 / 218
Verkle Trie with multiproof	256	256 / 183	2325 / 87

Table 7.1: Overall comparison between the examined data structures for predetermined branching factors

In table 7.1 a comparison between merkle trie, verkle trie, and verkle trie with multi-proofs can be seen. Here, the branching factor was chosen independently for each data structure based on what we deemed as the most reasonable branching factor. The average proof size and average proof construction times are presented for 1 and 100 proven key-value pairs. Applications currently using a merkle trie could potentially reduce the network communication overhead by a factor of 4, while only increasing the average proof construction time by less than 100 milliseconds by switching to a verkle trie with multi-proofs. As mentioned previously, one must be certain that more than 100 pairs need to be proven at once. Otherwise, if only a single key-value pair needs to be proven, the bandwidth is still reduced by a factor 2.5, yet at a proof construction cost of over 2 seconds. The designer of an application should take the update times into consideration as well. As seen in figure 6.5, the insertion time grows very fast with the size of the branching factor. However, there exists techniques to bring this down asymptotically by storing extra pre-computed commitments and further utilizing the homormorphic property of elliptic curve commitments [3]. Instead of having to re-compute every KZG polyomial on the root-to-leaf path and commit to that, it is possible to perform an operation to update the existing commitment independent of the branching factor. This has not been the focus of this paper, but could be interesting to research further.

In general, it is hard to determine a single number as the optimal branching factor for verkle tries. It is possible to get an indication of where this optimum might be, but in the end, it is all about the constraints of the application. As an example, verkle tries with multi-proofs are scheduled to be implemented in the Ethereum blockchain to store the whole blockchain state instead of a hexary merkle trie which is currently used. Here, the key might be an Ethereum address (a 20-character hexadecimal number) and the value might be the cryptocurrency balance denominated in ETH or some other value. In this system, all regular nodes have the full state trie and the client software should be able to run on a regular consumer laptop. The goal is to create ”light clients” which can run on ultra-light hardware, thus not storing the whole state, but still be able to verify key-value pairs from the full state trie. Because the prover could be a consumer laptop, the proof construction cannot be too computationally heavy, so a very large branching factor might not be the best choice. The same goes for other peer-to-peer file-sharing systems such as BitTorrent which also currently uses merkle tries where the prover can be consumer hardware. Contrary, non-P2P applications such as GitHub might utilize the trade-off spectrum differently since the prover is a centralized data center, that has another level of computing power. Therefore, such an application might choose a higher branching factor to get smaller proof sizes.

8 Conclusion

This paper sought to provide understanding and experimental evaluation of merkle tries, and verkle tries with and without multi-proofs to solve the problem of data verification. It was seen how the merkle trie requires large proofs to perform the verification as all of the sister nodes on the root-to-leaf path must be provided. On the other hand, construction of the proof is very fast since no further computation is needed from the prover. These claims were validated by experimental evaluation.
Verkle tries reduce the proof size significantly by only needing to send commitments stored in nodes on the root-to-leaf path along with a small proof for each. KZG commitments are polynomial commitments that can be used as vector commitments to prove the parent-child relation of nodes. A KZG commitment is a commitment to a polynomial p on an elliptic curve. To construct a corresponding proof, the prover has to construct a polynomial q and commit to that. The larger the branching factor, the larger these polynomial has to be and computation on the elliptic curves is much more computationally heavy than doing standard arithmetic. It was shown through experiments that the proof construction time is exponential in the branching factor. Besides being much smaller in absolute terms than for merkle tries, the proof size decreases with the branching factor since the root-to-leaf path shortens. The optimal branching factor was found to be around 32 to 64.
To further reduce the proof size, verkle tries with multi-proofs combines all commitments and their corresponding points into one large polynomial which is committed to. This allows for the prover to only send commitments on the root-to-leaf path as with standard verkle tries, but only two field elements are required as additional proof. This makes it more efficient to prove many key-value pairs at once, since the root-to-leaf paths will share commitments and a constant sized KZG commitment is sufficient. The experimental evaluation confirmed that multiple key-value pairs proven result in smaller proof size and lower proof construction time. This means that an application can tune the branching factor to optimize how many key-value pairs are expected to be proven. However, the experimental evaluation showed that verkle tries using multi-proofs are strictly better regarding proof size than the standard verkle tries. The proof construction time becomes better when just a few more than one key-value pair are proven at once.
Overall, the three solutions correspond to different points on the trade-off spectrum between network communication overhead and computational overhead. Since sending data over a network is often regarded as more resourceful than local computation, verkle tries both with and without multi-proofs seem to be a good fit for many applications currently using merkle tries. Even when one of these solutions is chosen for an application, the branching factor can be chosen to further suit the needs. The experimental results gathered in this paper may give insight on how to best choose such a branching factor when designing an application using any of the three solutions investigated.

Bibliography

[1] John Kuszmaul. “Verkle Trees”. In: (2018).

[2] Dankrad Feist. *PCS multiproofs using random evaluation*. 2021. URL: https://dankradfeist.de/ethereum/2021/06/18/pcs-multiproofs.html.

[3] Vitalik Buterin. *Verkle trees*. 2021. URL: https://vitalik.ca/general/2021/06/18/verkle.html.

[4] Inge Li Gørtz. “Tries and Suffix Trees”. In: (), pp. 1–5.

[5] Nick Sullivan. *A (Relatively Easy To Understand) Primer on Elliptic Curve Cryptography*. 2013. URL: https://blog.cloudflare.com/a-relatively-easy-to-understand-primer-on-elliptic-curve-cryptography/.

[6] Vitalik Buterin. *Exploring Elliptic Curve Pairings*. 2017. URL: https://vitalik.ca/general/2017/01/14/exploring_ecp.html.

[7] Tom Walton-Pocock. *Kate Commitments: A Primer*. 2021. URL: https://hackmd.io/@tompocock/Hk2A7BD6U.

[8] Dankrad Feist. *KZG polynomial commitments*. 2020. URL: https://dankradfeist.de/ethereum/2020/06/16/kate-polynomial-commitments.html.

[9] David Wong. *How does PLONK work? Part 10: The Kate polynomial commitment scheme*. 2021. URL: https://www.youtube.com/watch?v=iwOovhLU8U4&t=1s.