Verkle Trees

John Kuszmaul

Abstract

We present Verkle Trees, a bandwidth-efficient alternative to Merkle Trees. Merkle Trees are currently employed in a variety of applications in which membership proofs are sent across a network, including consensus protocols, public-key directories, cryptocurrencies such as Bitcoin, and Secure File Systems. A Merkle Tree with n leaves has $O(\log_2 n)$-sized proofs. In large trees, sending the proofs can dominate bandwidth consumption. Vector Commitments (VCs) pose a potential alternative to Merkle Trees, with constant-sized proofs. Unfortunately, VC construction time is $O(n^2)$, which is too large for many applications. We present Verkle Trees, which are constructed similarly to Merkle Trees, but using Vector Commitments rather than cryptographic hash functions. In a Merkle Tree, a parent node is the hash of its children. In a Verkle Tree, a parent node is the Vector Commitment of its children. A Verkle Tree with branching factor k achieves $O(kn)$ construction time and $O(\log_k n)$ membership proof-size. This means that the branching factor, k, offers a tradeoff between computational power and bandwidth. The bandwidth reduction is independent of the depth of the tree; it depends only on the branching factor. We find experimentally that with a branching factor of $k = 1024$, which provides a factor of 10 reduction in bandwidth, it takes 110.1 milliseconds on average per leaf to construct a Verkle Tree with 2^{14} leaves. A branching factor of $k = 32$, which provides a bandwidth reduction factor of 5, yields a construction time of 8.4 milliseconds on average per leaf for a tree with 2^{14} leaves. (The performance on a tree with 2^{14} leaves is representative of larger trees because the asymptotics already dominate the computation costs.) My role in this research project has been proving the time complexities of Verkle Trees, implementing Verkle Trees, and testing and benchmarking the implementation.

1 Introduction

Suppose that Alice has some files, F_0, F_1, ... , F_n, which she wants to store remotely, for example, at a company called Dropbox.

One way for Alice to do this would be to send her files across the network to Dropbox. Then Alice can query Dropbox for individual files F_i. Unfortunately, Alice has no way to verify that the file Dropbox responds with, D_i, is the same as the file F_i and has not been maliciously modified.

In light of this, Alice would like to be able to verify the integrity of the files she receives back from Dropbox. To achieve this generally, Alice will first generate a *digest* d over her files. She will store d on her local device and only then send her files to Dropbox. When querying Dropbox for individual files F_i, Alice will receive a file D_i, but also a membership proof, π_i for the file with respect to the digest d. Finally, Alice will verify the proof π_i against the digest d, which she has stored locally, to ensure that the file she received from Dropbox, D_i, is the same as the file she initially sent to Dropbox, F_i.

There are well-known data structures that Alice can use to do this. Merkle Trees and Vector Commitment Schemes would both work, but Merkle Trees could have membership proofs that are too large, which would result in costly bandwidth overheads as Dropbox sends membership proofs to Alice, and Vector Commitment Schemes can be computationally expensive to compute. We introduce *Verkle Trees* which offer a tradeoff between computational power and bandwidth.

Verkle Trees have the potential to reduce the proof size, and thus the bandwidth, in applications including consensus protocols, public-key directories, cryptocurrencies such as Bitcoin [1], encrypted web applications, and secure file systems. These are all applications in which Merkle Trees are currently employed.

In Section 2, we will discuss the background of our work, including Cryptographic Hash Functions, Merkle Hash Trees, and Vector Commitment Schemes. In Section 3, we discuss our system including its design and time complexities. In Section 4, we discuss our implementation of Verkle Trees. In Section 5 we present and analyze our results. Section 6 contains our conclusions and future work, and Section 7 is our acknowledgements.

2 Background

2.1 Hash Functions: A Simple Scheme

To see how Alice can verify the integrity of her files on Dropbox in practice, we must first discuss Cryptographic Hash Functions [2]. A Cryptographic Hash Function H takes an input of arbitrary length, and returns its *hash*, a short fixed-length binary string. SHA-256, for example, gives a 256-bit output. The most important property of Cryptographic Hash Functions, for our work, is their strong collision resistance. For a strongly collision resistant hash function [2], it is computationally infeasible to find two distinct inputs, m and m', which have the same output, i.e., $H(m) = H(m')$. It should be noted that given there are an infinite number of possible inputs to a Cryptographic Hash function such as SHA-256, and only a finite number of outputs, there do exist two distinct inputs that yield the same hash by the pigeonhole principle, but for a strongly collision resistant hash function, it is computationally infeasible to find two such inputs.

Using Cryptographic Hash Functions, we can quickly construct a scheme that will allow Alice to verify the integrity of the files she receives back from Dropbox. Alice will first hash each file and store these $O(n)$ hashes, $h_0 = H(F_0), h_1 = H(F_1), ..., h_n = H(F_n)$, locally on her computer as the digest d. Only then will she send her files to Dropbox. When Alice queries Dropbox for an individual file F_i, Dropbox will respond with a file D_i (note that in this scheme, the membership proof is empty, and thus Dropbox responds with the file alone). Alice verifies that the file she receives, D_i, is the same as the file she sent to Dropbox, F_i, by checking that $H(D_i) = h_i$. Dropbox cannot respond with a distinct file $D_i \neq F_i$ that will satisfy $H(D_i) = h_i = H(F_i)$ without breaking the strong collision resistance of the Cryptographic Hash function.

The problem with this scheme is that it requires Alice to store n digests on her local device. This is impractical, and we would like a scheme that allows Alice to store a constant-sized digest locally regardless of how many files she is storing on Dropbox. The well-known solution to this problem is the Merkle Tree [3], discovered by Ralph Merkle in the 1970s.

2.2 Merkle Trees

To construct a Merkle Tree, Alice will first hash each of her files. She will then continue hashing upwards as shown in Figure 1. To compute a given node in the Merkle Tree, Alice looks at its two children nodes, each of which contains a hash. Alice concatenates these two hashes and hashes again to compute the given node.

The root of the Merkle tree, R, serves as the digest; Alice will store it locally on her

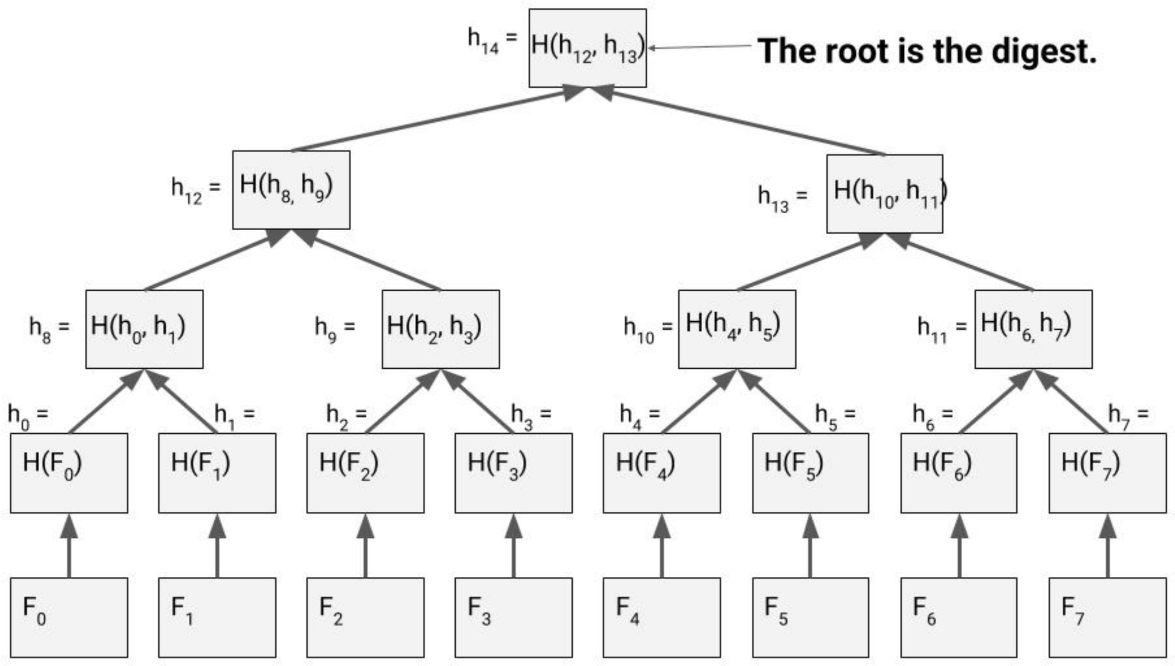

Figure 1: A Binary Merkle Tree

device. When Alice queries Dropbox for individual files F_i, Dropbox will respond with a file D_i, and the membership proof π_i, the Merkle proof for D_i. The Merkle proof of a leaf in a Merkle tree consists of the siblings of each of the nodes in the path from the leaf to the Merkle root (see Figure 2).

Alice can then verify D_i with respect to the digest, R, which she stores locally on her device, with the Merkle proof she receives from Dropbox (see Figure 2). Alice hashes up the tree along the path from the leaf D_3 up to the Merkle root using only the nodes that she received from Dropbox in the Merkle proof. Once Alice computes a new Merkle Root D_R, she checks that $D_R = R$. If it does, she can rest assured that $D_i = F_i$, because in order to forge a Merkle proof for a modified file $D_i \neq F_i$, Dropbox would have to break the strong collision resistance of the Cryptographic Hash function [3].

Merkle Trees are computationally fast, and a Merkle Tree over n files can be constructed in $O(n)$ time (See Figure 5).

Unfortunately, a Merkle Tree that contains many small files can have Merkle proofs that are then prohibitively large. Suppose Alice has $2^{30} \approx 10^9$ files. The depth of the tree will then be approximately 30, and since the Merkle Proof consists of one node (which is a hash) at each level in the tree (besides the first), the Merkle Proof will work out to be approximately one kilobyte (using a 256-bit hash such as SHA-256). Dropbox has to send both the file and a Merkle Proof across the network to Alice when queried for an individual file, so for

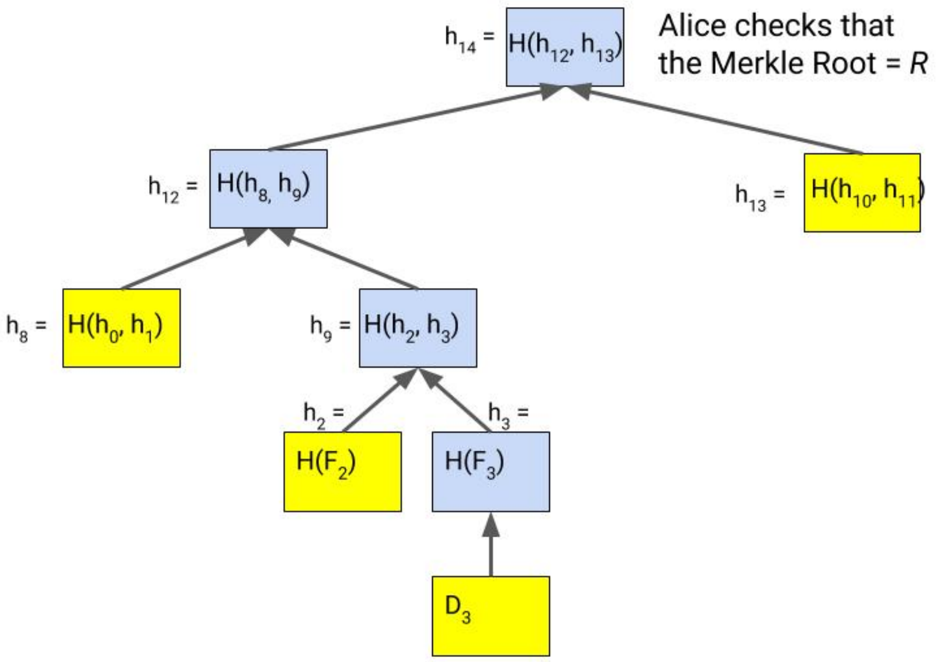

Figure 2: The Merkle Proof (in yellow)

small files, the Merkle Proof itself could create a large and expensive bandwidth overhead on Dropbox.

2.3 k-ary Merkle Trees

One possible solution is to use a k-ary Merkle Tree. In a binary Merkle Tree, the proof consists of one node at each level, so to reduce the size of the proof, a natural instinct is to reduce the height of the tree by giving it a branching factor of $k > 2$. Giving our Merkle tree a branching factor of k reduces the height of the tree from $\log_2 n$ to $\log_k n$, a $\log_2 k$ factor decrease in height. Unfortunately, the Merkle proof actually grows larger, from $O(\log_2 n)$ to $O(k \log_k n)$. This is because in a k-ary Merkle Tree, the Merkle proof actually consists of $(k-1)$ nodes at each level (besides the first); Alice needs to receive the other $k-1$ children of a node in order to concatenate together all k children and hash to compute the parent node.

Our work, Verkle Trees, reduce the proof size from $O(\log_2 n)$ to $O(\log_k n)$ for a branching factor of k. This results in a factor of $\log_2 k$ less bandwidth. So if we set $k = 1024$, the Verkle Proof will be 10 times smaller than the Merkle Tree. This comes at a cost of k times more computation when constructing and updating the tree. This is remarkable because it allows us to reduce the bandwidth by an order of magnitude at the cost of a reasonable

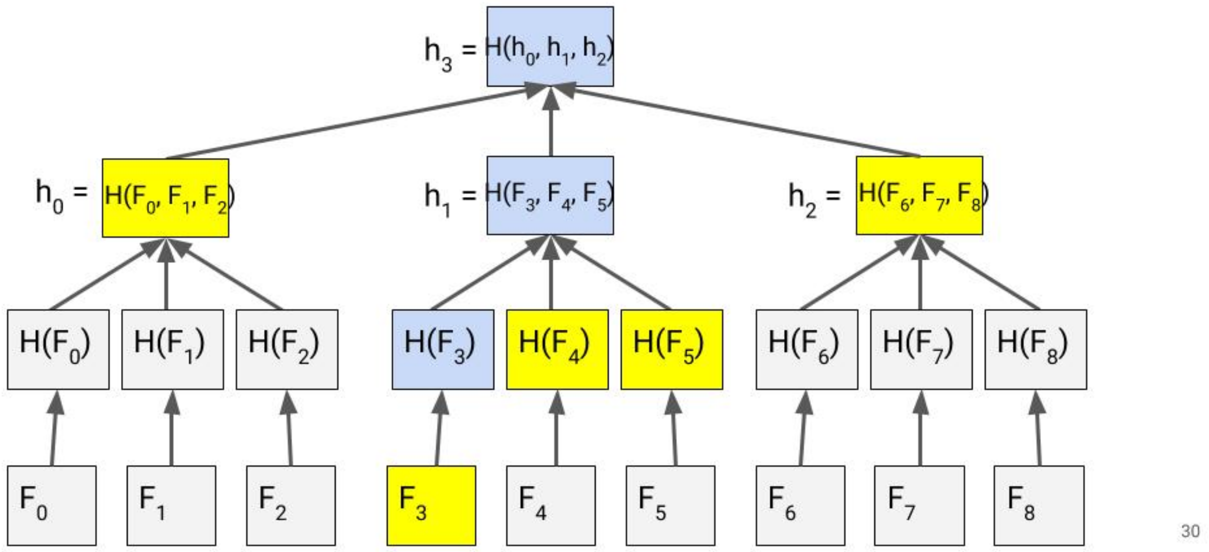

Figure 3: A 3-ary Merkle Tree: The Merkle Proof, in yellow, is larger!

increase in computational power. Note that bandwidth is typically much more expensive than computational power in the applications for which Merkle Trees are typically used.

Verkle Trees use Vector Commitment (VC) schemes.

2.4 Vector Commitment Schemes

Vector Commitment schemes [4] are an alternative scheme that Alice could use to verify the integrity of her files on Dropbox. A commitment C is computed over the $F_0, F_1, ..., F_n$ along with membership proofs $\pi_0, \pi_1, ..., \pi_n$ for each file $F_0, F_1, ...F_n$ respectively with respect to the commitment C. Thus C is the digest of the VC. Significantly, each membership proof is constant-sized, regardless of how many files are contained in the VC. The VC Scheme we use is by Catalano and Fiore and is constructed using billinear groups and the Computational Diffie-Hellman assumption. The background of the scheme itself is beyond the scope of this paper. It is remarkable that the proofs are constant-sized, thus reducing the bandwidth required to send a proof to a constant. Unfortunately, it takes $O(n^2)$ time to construct a VC along with each of its $O(n)$ membership proofs.

3 System

At their heart, Verkle Trees combine techniques used in Merkle Tree as well as Vector Commitment Schemes to provide a tradeoff between bandwidth and computational power.

3.1 Goal

Our goal is to create a new data structure, Verkle Trees, which reduce the bandwidth required for Merkle Trees. We will do so by reducing the membership proof size. This will come at the cost of increasing the computational power required to compute and update a Verkle Tree relative to that required to compute and update a Merkle Tree. The Verkle Tree is a k-ary tree, i.e., k is the branching factor of the tree. As we will see, the branching factor, k, determines the exact tradeoff between computational power and bandwidth that a k-ary Verkle Tree provides.

3.2 Verkle Tree Design

The key insight of the Verkle Tree is that we can construct a Merkle Tree, but replace Cryptographic Hash functions with Vector Commitments. Before computing a Verkle Tree over some files F_0, F_1, ... , F_n, we first select the branching factor of the tree, k. We then group our files into subsets of k files and compute a Vector Commitment, C, over each of the subsets of files. We also compute each VC membership proofs π_i for each file F_i in the subset with respect to C. We then continue computing Vector Commitments up the tree over previously computed commitments until we compute the root commitment. In Figure 4, we began with 9 files and branching factor of 3. After dividing the files into subsets of size $k = 3$, a Vector Commitment is computed over each subset along with the corresponding membership proofs. This leaves us with the commitments C_1, C_2, and C_3. We compute the Vector Commitment C_4 over these three commitment along with the membership proofs π_9, π_{10}, and π_{11} for the commitments C_1, C_2, and C_3 respectively with respect to the commitment C_4. The digest of the Verkle Tree is the root commitment, which is C_4 in this case.

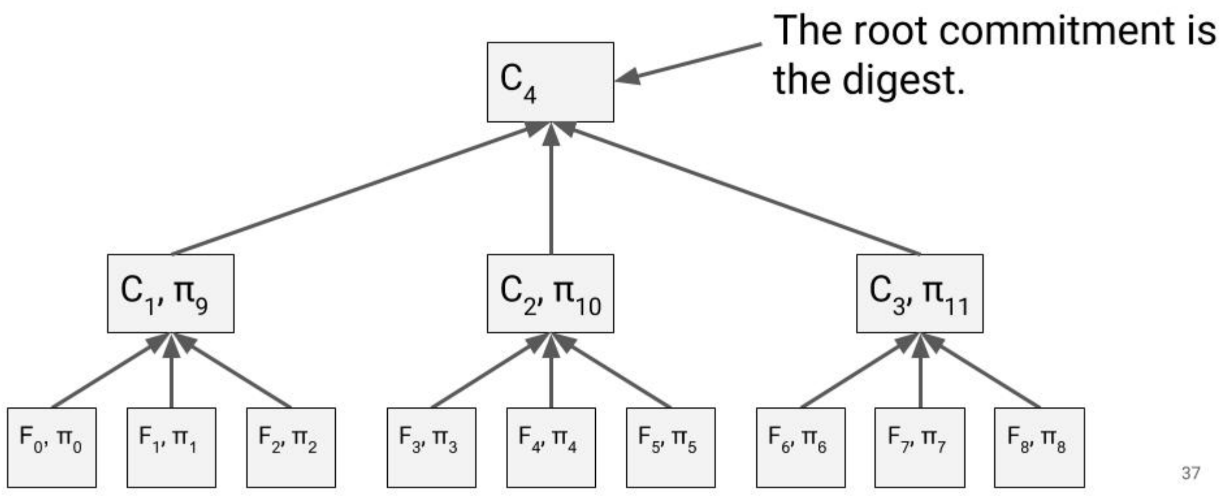

Figure 4: A Verkle Tree with $k = 3$

3.3 Time Complexities and Comparison

Scheme/Op.	Construct	Update File	Proof Size
Merkle Tree	$O(n)$	$O(\log_2 n)$	$O(\log_2 n)$
k-ary Merkle Tree	$O(n)$	$O(k \log_k n)$	$O(k \log_k n)$
Vector Commitment	$O(n^2)$	$O(n)$	$O(1)$
k-ary Verkle Tree	$O(kn)$	$O(k \log_k n)$	$O(\log_k n)$

Figure 5: Time Complexities

We began by looking at Merkle Trees, which are exceptionally fast and can be computed in $O(n)$ time. Unfortunately, their proof size of $O(\log_2 n)$ is quite large and can be costly in terms of bandwidth. We then briefly turned to k-ary trees, but their proofs are actually even larger than Merkle Trees, $O(k \log_k n)$. Vector Commitment Schemes reduce the proof size down to a constant, $O(1)$, but this comes at the cost of $O(n^2)$ computation to construct the Vector Commitment, which is very expensive.

The k-ary Verkle Tree, on the other hand, takes only $O(kn)$ time to construct. Furthermore, its proof size is only $O(log_k n)$, a factor of $O(log_2 k$ times smaller than the Merkle Tree's memberhsip proofs. We believe that this is a satisfactory tradeoff, because it allows the reduction of bandwidth at an unreasonable increase in computational power. Generally, bandwidth is cheaper than computational power.

If we set $k = 1024$, we achieve a 10 times reduction in proof size, and thus bandwidth, at the cost of a 1024 increase in computational power. If less computational power is desired, k can be lowered. The exact value of k may vary from application to application depending on the environment-specific optimization between computational power and bandwidth.

4 Implementation

Our implementation of Verkle Trees was written in C++ and uses the RELIC [5] library as well as the SHA-256 cryptographic hash function. We began by implementing a Vector Commitment scheme by Catalano and Fiore [4]. We then proceeded to implement a Verkle Tree as a prefix tree [6]. In a prefix tree, a leaf is represented by a key-value pair. The prefix of the leaf (which determines its location in the prefix tree), is determined by the hash of the leaf's key. This ensures that two distinct leaves will have distinct prefixes due to the collision resistant nature of SHA-256.

As far as we know, we are the first to implement the Vector Commitment scheme of Catalano and Fiore. We are also, to the best of our knowledge, the first to implement Verkle Trees.

5 Results

We have benchmarked our implementation of Verkle Trees. In particular, we measured the amount of time it took to construct the Verkle Tree by updating a prefix Verkle Tree one node at a time, starting with an empty tree. It would be faster to construct the entire Verkle Tree at once, but our benchmark represents a simulation in which a server is continuously receiving new leaves and then updating its prefix Verkle Tree to contain them.

Our graphs, which have logarithmic axes, demonstrate the adaptability of the Verkle Tree. Figure 6 demonstrates the time required to construct a Verkle Tree with branching factor $k = 1024$. While it is approximately linear, the Verkle Tree is actually relatively slow, taking 110.1 milliseconds per leaf to construct the largest tree shown with 2^{14} leaves. Recall, however, that this comes with an associated factor of 10 reduction in proof size relative to Merkle Trees.

If however, this is too computationally expensive, the branching factor k can be varied, as shown in Figure 7. The branching factor could be, for example, reduced from $k = 1024$

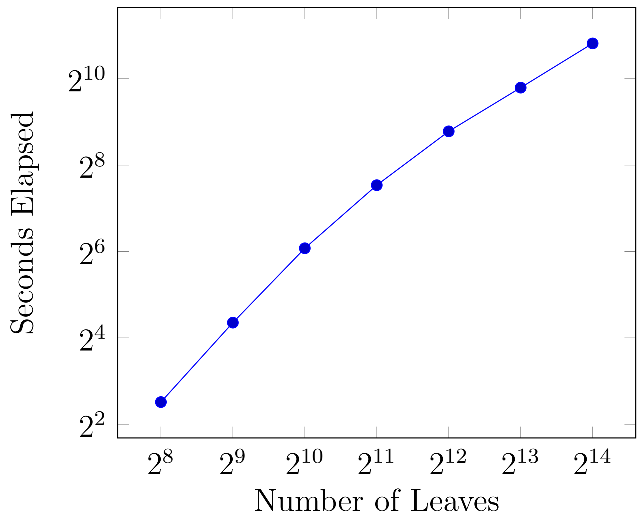

Figure 6: Tree Construction Time for $k = 1024$

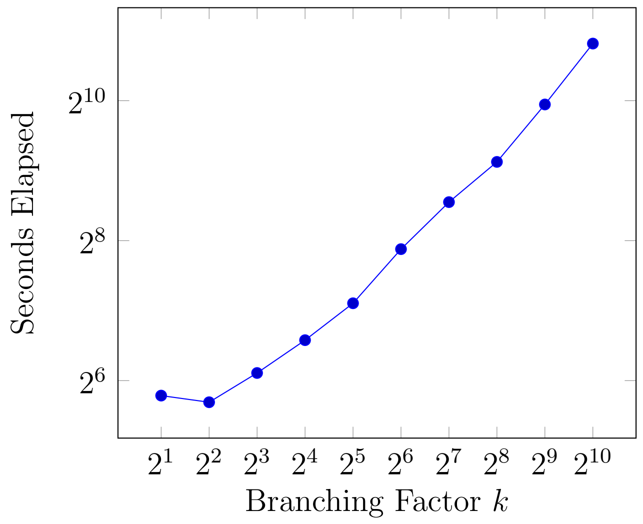

Figure 7: Tree construction time for 2^{14} leaves

to $k = 32$, which would result in only a factor of 5 reduction in proof size relative to Merkle Trees. This would greatly decrease the computational power required for the Verkle Tree, however, to only 8.4 milliseconds per leaf, which is approximately 13 times faster.

Thus applications which receive frequent updates to their tree may wish to choose a lower branching factor, whereas more static Verkle Trees can be maintained with a higher branching factor, thus saving bandwidth and reducing the size of membership proofs.

6 Conclusion and Future Work

We conclude from our preliminary benchmarks that Verkle Trees are a potential replacement of Merkle Trees for many applications. Verkle Trees have the advantage that the branching factor, k, is variable and can be adapted to satisfy different criteria.

In the future, we would like to further optimize our implementation of Verkle Trees by implementing it in parallel. Other optimizations we are considering include batching updates.

7 Acknowledgements

I would like to thank my mentor, Alin Tomescu, for his invaluable guidance. He provided the idea for this project.

I would also like to thank MIT PRIMES, Dr. Slava Gerovitch, and Dr. Srini Devadas for giving me this research opportunity.

References

[1] S. Nakamoto, “Bitcoin: A Peer-to-Peer Electronic Cash System,” https://bitcoin.org/bitcoin.pdf, 2008, Accessed: 2017-03-08.

[2] P. Rogaway and T. Shrimpton, “Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance,” 2004, Accessed: 2017-9-16.

[3] R. C. Merkle, “A digital signature based on a conventional encryption function,” in *Conference on the Theory and Application of Cryptographic Techniques.* Springer, 1987, pp. 369–378.

[4] D. Catalano and D. Fiore, “Vector commitments and their applications,” in *Public-Key Cryptography–PKC 2013.* Springer, 2013, pp. 55–72.

[5] D. F. Aranha and C. P. L. Gouvêa, “RELIC is an Efficient LIbrary for Cryptography,” https://github.com/relic-toolkit/relic.

[6] S. Ramabhadran, J. Hellerstein, S. Ratnasamy, and S. Shenker, “Sok: Research perspectives and challenges for bitcoin and cryptocurrencies,” http://static.usenix.org/event/sec09/tech/full_papers/crosby.pdf, Accessed: 2017-5-15.