Two tree-based associative hash functions

Jelle van den Hooff

Abstract

Cryptographic hash functions are frequently used to fingerprint data. However, such existing hash functions are not associative, making them expensive to maintain with changing inputs or when two files are concatenated in an archive, as a non-associative hash must be completely recalculated in linear time. We present two associative hash functions whose output grows logarithmically with the length of their inputs. Each hash function consists of a Merkle hash over a history-independent balanced tree. The Merkle hash guarantees our hash function's security, and relies on a history-independent tree for correctness. The first tree we use is the treap, which is simple, but not guaranteed to remain balanced for certain inputs. The second tree we use is SeqHash, which is guaranteed to remain balanced, but is more complicated and has a slightly higher overhead. We analyze both hash functions, and show experimental results that match our analysis.

1 Introduction

Hash functions are frequently used to summarize and authenticate data. While powerful, existing cryptographic hash functions are not associative. Imagine creating an archive file consisting of several large input files. Using SHA512 to compute a hash of the concatenation requires recomputing the hash from the start, taking time linear in the inputs' lengths.

This paper introduces two novel associative hash functions that can be reused and combined in sub-linear time. An associative hash function consists of two functions, hash and concat. The function hash computes a fingerprint $\text{hash}(s)$ of a sequence s, and concat computes two concatenation of two fingerprints $\text{concat}(\text{hash}(a), \text{hash}(b)) = \text{hash}(a||b)$. The size of $\text{hash}(s)$ can depend on the length of s. Our goal is to bound the size $|\text{hash}(s)| = O(\log |s|)$, and to bound the runtime of $\text{concat}(\text{hash}(a), \text{hash}(b))$ by $O(\log |a| + \log |b|)$ (both with high probability).

To the best of our knowledge, no existing associative hash function satisfying the above definition is secure. For example, the Tillich-Zemor associative hash function [4] has been shown to exhibit collisions [1]. However, that might be because our definition of an associative hash function lets fingerprints grow, while traditional hash functions force the size $|\text{hash}(s)|$ to be constant (e.g. 512 bits for SHA512). Assuming an underlying non-associative cryptographic hash function, we show an upper bound of $O(\log |s|)$ for associative hash function size. It remains an open question if an associative hash function of size $O(1)$ exists.

Our hash functions are based on a hierarchically-computed hash. We built a tree over the input sequence, and compute a recursive Merkle hash [2] over that tree where the hash of each node is a cryptographic hash of the hashes of its children, and the hash of the tree is the hash of the root node. Using a Merkle hash is appealing because it ensures our construction remains secure as long as the underlying cryptographic hash function is secure. The challenge in building a hash function using a Merkle hash is finding the right underlying data structure.

For a hash function to be useful it must be deterministic and always compute the same fingerprint for the same input. In our setting, the tree underlying the Merkle hash must have the same shape and internal structure for the same input sequence no matter how the tree was built. Yet most balanced trees can take different shapes depending on how they are constructed. For example, constructing a red-black tree by appending leafs from left-to-right leads to a different shape than appending the same leafs from right-to-left (Figure 1). To be useful for our hash functions, the tree must always have the same shape. The requirement that two data structures have the same representation regardless of the order of operations used to construct them is formalized in the notion of history-independence [3].

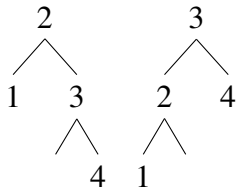

Figure 1: Two red-black trees storing the same data inserted either from left-to-right (left tree) or from right-to-left (right tree). Because the trees have different shape, the recursively calculated Merkle hashes $H(2, 1, H(3, nil, 4))$ and $H(3, H(2, 1, nil), 4)$ of the two trees have different values and not useful for fingerprinting.

The challenge this paper focuses on is picking and using a history-independent tree as an associative hash function. By re-framing the problem of building a hash function into a problem of building and manipulating trees, we transform a cryptography problem into a data structures problem. We outline our approach in §2. To start out, we transform a treap into an associative hash function in §3. However, in our setting, a treap is not guaranteed to be balanced. To improve our design, we use the history-independent balanced tree SeqHash. We describe SeqHash in §4, and cover our final construction in §5. Finally, we evaluate the performance of both the treap- and SeqHash-based hash functions in §6.

2 From history-independent tree to associative hash function

The core building block of our hash functions is an efficient implementation of a history-independent balanced binary tree that takes an input s and builds a balanced tree T_s of height $O(\log |s|)$. This tree must support efficient merging, so that we can merge trees T_a and T_b into $T_{a||b}$ corresponding to the concatenation $a||b$ in time $O(\log |a| + \log |b|)$.

Using the tree as a basic building block, we can compute a Merkle hash of the tree by recursively hashing nodes. However, the tree itself is not suitable as an associative hash, as it is too large to pass around. While we could easily distribute the small hash of the root of T_s, that root hash can not be concatenated: We need to know the structure of T_a and T_b to compute $T_{a||b}$, and that is lost when looking at just the hashes of the roots of the trees. Our solution is to store a part of the trees T_a and T_b large enough to calculate the Merkle hash of T_a and T_b but small enough to pass around.

We call this small part of the tree T_s the *fringe* F_s (Figure 2). We require that this fringe F_s has size at most $O(\log |s|)$ so it can be passed around efficiently, that we can can calculate the Merkle of T_s from F_s, and that we can combine two fringes F_a and F_b into the fringe $F_{a||b}$. We show that such a fringe exists for both the treap in §3 and for SeqHash in §5.

Our final construction in both cases consists of two functions, hash and concat. The function hash computes the fringe F_s from s, and concat computes $F_{a||b}$ from F_a and F_b. The fringe F_s has size $O(\log |s|)$ and can be computed in time $O(n)$. The concat function has runtime $O(\log |a| + \log |b|)$.

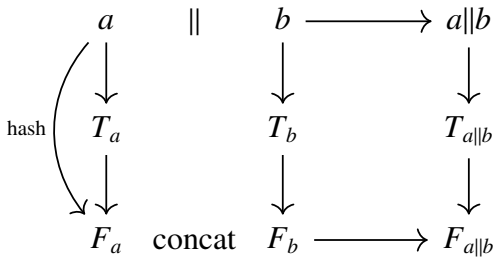

Figure 2: A diagram showing the relationship between hash, concat, F, and T. The sequence s has corresponding tree T_s with fringe F_s. We can compute F_s using $\text{hash}(s) = F_s$. Two sequences a and b can be concatenated to obtain $a||b$. Using concat, we can compute $F_{a||b}$ using $\text{concat}(F_a, F_b) = F_{a||b}$.

3 A treap-based associative hash function

A treap is a randomized balanced binary (search) tree. A treap satisfies two properties. First, it is an ordered tree, where the in-order traversal of the nodes with their values corresponds to the original order of the values in the tree. Second, it assigns a priority to each node and satisfies the heap property that each node has a higher priority than its children (breaking ties by order of values). These two properties combined completely determine the shape of the tree.

Normally each treap node is given a random priority, and that ensures that the treap is balanced. However, a treap is naturally transformed into a history-independent data-structure by assigning each node a priority derived from a pseudo-random function f that maps node values a_i to priorities $p_i = f(a_i)$. This derandomized treap is history-independent because the two treap properties combined completely constrain the shape of the tree.

To use the treap as an associative hash function, we need to extract a fringe F_s from a treap T_s that still supports efficient merging. To understand what a fringe F_s should look like, we will consider the algorithm that merges two trees T_a and T_b. Intuitively, the root node of the merged tree $T_{a||b}$ must be either the root node of T_a, or the root node of T_b, as one of those must have the highest priority in all of $a||b$. The node to the left of T_a's root node will keep their structure, as none of their priorities or values will change. Similarly, the nodes to the right of T_b's root node keep their place. Only the node in between the root nodes might change (Figure 3).

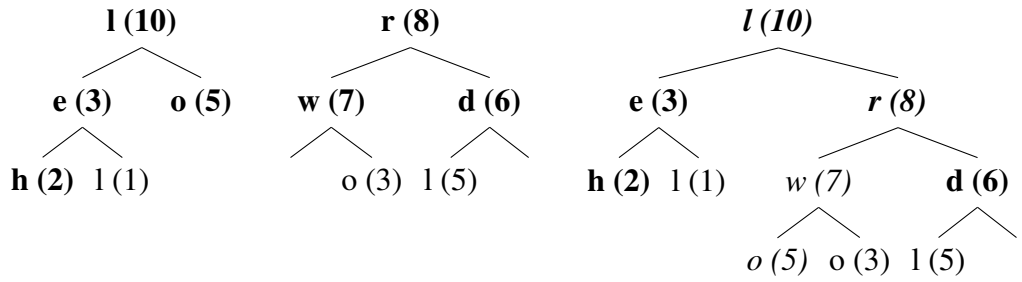

Figure 3: Three treaps containing, from left to right, the sequences "hello", "world", and "helloworld". All treap nodes contain both a value a_i and a priority $p_i = f(a_i)$ formatted as a_i (p_i). Nodes in the treaps' fringes are typeset **bold**. Nodes changed when merging "hello" and "world" into "helloworld" are *italic*.

The actual algorithm to merge two treaps is simple (Figure 4). This algorithm is efficient as it only descends into one node. This means that if the treaps are balanced, merging two treaps take time $O(\log|a| + \log|b|)$. More importantly, merging two treaps only requires access to a small part of the original trees.

To construct a fringe to form our hash function, we will store only the nodes that could be accessed by the merge algorithm. Luckily, there are not many of those nodes. When merging two trees a and b, the algorithm only ever access the right child of a, and only ever access the left child of b. To support any kind of merging, we only need to store the nodes on the outside of the tree: Nodes that can be reached either by only going left, or by only going right from the root of the tree. We call these nodes the fringe of the tree (Figure 3).

```
def treap_merge(a, b):
    if a == None or b == None:
        return a or b
    if a.priority > b.priority:
        return a.update(right=treap_merge(a.right, b))
    else:
        return b.update(left=treap_merge(a, b.left))
```

Figure 4: The treap merge algorithm for computing $T_{a\|b}$ from T_a and T_b. The update function returns a new node with the given field updated to the new value.

3.1 Treap analysis

Both the performance of concat and the amount of space required to hold a hash are directly proportional to the size of the fringe F_s. Intuitively, the size of the fringe F_s is proportional is at most twice the height of the tree, or $O(\log |s|)$ for sequences s with balanced trees.

We can explicitly calculate the expected size of fringe using some simple tools from probability. To start, let us determine what values are part of the left branch of the fringe. The leftmost value in the input sequence is always going to be part of the fringe, as it cannot be any node's right child. The parent of the leftmost value is going to be the first value in the sequence that has higher priority than the leftmost value. It is part of the fringe as, similar to before, it can also not be any later node's right child. To find all nodes in the left branch of the fringe, we keep looking for values that have a greater priority than any values seen before.

To count the number of values in the fringe, we consider the probability that an element will be a part of the left fringe. An element is part of the left fringe if and only if it has a higher priority than all elements seen so far. If all priorities are independent random variables, element i has a higher priority than all other elements with probability $1/i$. That gives an expected number of elements in the left fringe is then $\sum_1^n 1/i = H_n$. The complete fringe has two sides, with one shared element, so the total number of elements in the fringe is $2H_n - 1$ in expectation.

This analysis holds if all probabilities are independent. This is the case if f is pseudo-random, all inputs to f are distinct, and have been picked without considering f. To get the desired behavior from f, we assume the random oracle model and use a cryptographic hash function from f. It is harder to ensure that the inputs to f are distinct. In some settings, such as hashing log file entries, it might be feasible to assume a unique timestamp or identifier in each value of s. In those settings, the above analysis gives us a bound on the size of the fringe and the runtime of the merge function.

In other settings, the treap breaks down. For example, for inputs $s = a_1 a_2 \cdots a_n$ where $a_1 = a_2 = \ldots = a_n$ the priorities assigned to each node are identical, and the resulting tree becomes a linked list. While such a tree might be compressed (by observing that all nodes are identical), an even worse scenario is an adversary that sample several input elements, and sorts them by their priority $f(a_i)$ to construct a sequence where all a_i are distinct and $f(a_1) < f(a_2) < \cdots < f(a_n)$.

Intuitively, the problem with the treap's design, is that each node's priority has a global effect on the treap's shape: A node with high priority is going to be at the top of the treap,

no matter what, and that means that either identical nodes, or adversarially sampled nodes, are always going to be a threat to treap's balanced shape. It would be far better to use the pseudo-randomness locally. For example, if we could just consider the relative priority of adjacent nodes, make a local decision on their relative position based on that randomness, and then re-sample a priority to make a decision for their combined position relative to other nodes, we would be in much better shape. This intuition is the motivation behind SeqHash's design.

4 SeqHash

The SeqHash tree [5] is a derandomized history-independent balanced binary tree. The shape of a SeqHash is defined by construction. At a high level, SeqHash builds a tree level by level, and follows a merging process at each level to determine which nodes in each level should be combined into a new node. This section describes the construction in detail.

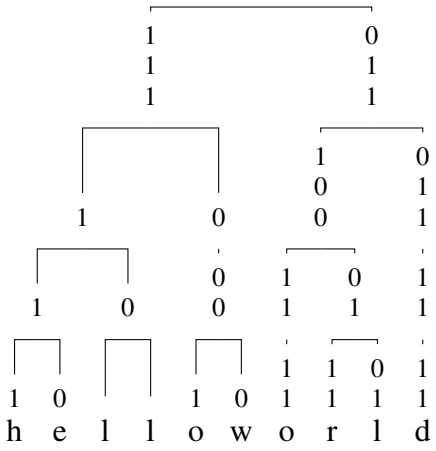

Figure 5: A SeqHash tree constructed over the sequence "helloworld". The tree has five levels. The leafs of the tree are individual characters of "helloworld". At that first level, many nodes merge. The 'h' and 'e' merge because they generate a 1-bit and 0-bit for their first bit. The two adjacent 'l's merge because they are identical. However, the 'o' and 'r' do not merge as 'r' does not generate a 0-bit. The 'd' does not merge in the first round as its only adjacent node, the 'l', merges with another node, the 'r'.

The first level of a SeqHash consists of leaf nodes with all the values in the SeqHash in their original order. Those leafs are merged to form the second level of the SeqHash,

and those nodes get merged again to form the third level, and so forth. This process continues until all nodes have been merged and a single root node remains on the final level.

To determine which nodes to merge on a single level, SeqHash follows merging process based on nodes generating pseudo-random bits (Figure 5). Each node's hash is used to seed a generator, creating a stream of bits coming out of each node. To merge nodes, SeqHash performs a bit-by-bit process: First, we sample a new bit from each node's generator, so that each node has either a 0-bit or a 1-bit attached. Then, we consider adjacent pairs of nodes. Whenever the node on the left generates a 1-bit, and the node on the right generates a 0-bit, we merge the two nodes and create a new node with the two existing nodes as children. This new node takes their place in the next level. Requiring a 1-bit on one side and a 0-bit on the other side ensures that the merging is well defined, and that there is only one possible interpretation of the merging rule.

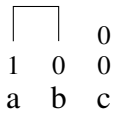

Figure 6: A zoomed-in view on the SeqHash merging process. The two nodes 'a' and 'b' merge at their first bits because 'a' generates a 1-bit and 'b' generates a 0-bit; correspondingly, 'b' and 'c' do not merge at their first bits because 'b' does not generate a 1-bit. After merging, 'b' no longer participates on this level. Even if 'b' were to generate a 1-bit for its second bit, it would not merge with 'c', as each node can merge at most once each level.

After merging two nodes, the merged nodes are no longer considered for the remainder of the level (Figure 6). The merging continues until merging is no longer possible. Some nodes might remain unmerged, but can also not merge anymore because both of their neighbors have been merged with their other neighbors. Such individual nodes pass through and become part of the next level, and have another chance to merge on that level.

Adjacent nodes with the same seed will never merge on a level. To handle this edge-case, SeqHash merges any sequence of identical nodes at the beginning of each level. This merge is multi-way: If there are more than two identical adjacent nodes, they all nodes get the same parent. This means that SeqHash is not a binary tree.

4.1 SeqHash analysis

SeqHash's runtime depends on both the number of levels, and the length of each level. The number of levels is bound by $O(\log n)$. The length of each level is $O(1)$ with high probability.

The number of levels is guaranteed to be bound by $O(\log n)$, as a large number of nodes merge each each level. A level ends only after no pairs of adjacent nodes remain unmerged. This means that at least $2/3$ of the nodes on each level must merge, and so the total number of nodes decreases by $1/3$ each level. That gives a guaranteed bound on the total height of the tree of $O(\log_{3/2}(n))$.

The length of each level is bound as each level will only consume a limited number of bits. Consider any pairs of adjacent nodes. If they are identical, they will immediately merge at the beginning of the level. Otherwise, for every bit generated, that pair of nodes will merge with probability $1/4$ (if they have not already merged yet), and so the expected number of bits generated is 4 and with high probability the nodes merge after $O(1)$ bits. By applying the union bound over all $O(n)$ nodes, we learn that each level ends after $O(\log n)$ bits with high probability. This analysis holds as long as the pseudo-random bit generator outputs pairwise independent bits for any two inputs.

In the face of an adversary trying to come up with input values that will consume many bits of randomness, the analysis changes. Specifically, an adversary might by themselves feed many values into the pseudo-random bit generator, and try and find values that merge only after very many bits. SeqHash remains efficient even in this case using a simple extension of the union bound. Instead of considering $O(n)$ pairs of adjacent nodes in the input, we must consider all pairs of nodes the adversary might have considered. We assume the adversary has considered at most $O(h)$ different hashes resulting in at most $O(h^2)$ pairs of hashes. We can apply the union bound again over all those $O(h^2)$ pairs and obtain a bound of $O(\log h)$ bits per level. This bound is comparable to the strength of cryptographic constructions where the challengers have an exponential advantage over adversaries.

Because h appears only in the analysis of SeqHash, the performance guarantees of SeqHash adapt automatically to the amount of effort put in by an adversary without having to predetermine an upper bound on h.

5 A SeqHash-based associative hash function

The SeqHash tree can be used as the basis of an associative hash function by defining a suitable fringe of the tree. Unlike the treap's fringe, the SeqHash's fringe is more involved and is based on a partial evaluation of the SeqHash construction.

The key idea of the partial evaluation is that it is not necessary to know all input values to determine if two nodes will merge in a SeqHash. Specifically, two adjacent nodes that generate a 1-bit and a 0-bit for their first bit will merge no matter what the other nodes generate. So even if we do not know all inputs, we can still partially evaluate the SeqHash construction for a sub-sequence. This partially evaluated SeqHash can then later be combined with another partially evaluated SeqHash to construct a SeqHash for the combined sequence.

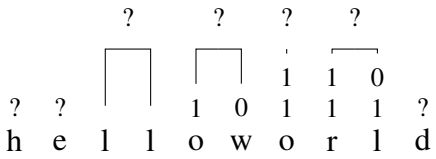

Figure 7: A partially-evaluated SeqHash tree constructed over the sequence "helloworld". In a partially-evaluated SeqHash, the fate of some nodes remains unknown as it depends on nodes that might be added to the left or to the right of the sequence. For example, the 'h' is marked as unknown as it is the leftmost element of the sequence, and it might merge with another 'h' as part of the duplicate-merging step at the beginning of a level. The 'e' is also marked as unknown, as it might merge with 'h' if the 'h' does not merge with its left neighbor. Not all nodes remain unknown. The two 'l's will merge as part of the duplicate-merging step, no matter what nodes are prepended before or after "helloworld". Similarly, the 'o' and 'w' will always merge as they generate a 1-bit and 0-bit for their first bits.

To compute a partially evaluated SeqHash, the construction process is expanded to consider two potential unknown values at the beginning and end of the input sequence (Figure 7). Whenever a node might merge with those unknown values, that node itself is marked as unknown as well. For example, the first node is always marked as unknown, as it might merge with its (yet to be added) left neighbor if they have same value. After that, we do not know if the first node merges or not, and that uncertainty propagates. For example, the second node might be marked as unknown if it generates a 0-bit for its first bit and the first node generates a 1-bit for its first bit. Luckily, though, this uncertainty propagates only one bit at a time: A node can be marked only unknown the bit after the

its neighbor was marked as unknown. This means that the total number of unknown nodes should be limited (as we will cover in our analysis below).

The final output of a partial evaluation is a sequence of unknown nodes for both sides of each level. This partial evaluation can be turned into a complete evaluation by running the merging algorithm, now with all inputs known, and computing the fate of the remaining unknown nodes.

The partial evaluation is the SeqHash equivalent of the fringe. All nodes that are not unknown nodes will never merge with another node, and so we need not store them for use in the fringe. Instead, we store only the unknown nodes, and obtain a fringe suitable for an associative hash function.

5.1 Fringe analysis

The size of the fringe and the runtime of the merge is determined by the number of unknown nodes on each level.

Just like a full SeqHash, a fringe will have $O(\log n)$ levels for an input of length n. Since each level's length is bound by $O(\log n)$, the total number of unknown nodes is easily bound by $O(\log^2 n)$. However, a better bound exists, as the total length of each level is too strong for the fringe. Instead, the fringe consists only of the unknown nodes on each level. That number is a lot lower: Each level has $O(1)$ fringe nodes with high probability, or $O(\log h)$ for an adversary that has computed a total of $O(h)$ values of the pseudo-random generator, giving an $O(\log n)$ or $O(\log n \log h)$ bound.

To derive the $O(1)$ fringe nodes bound, we apply the union bound on a pair of nodes $O(1)$ from the end of the sequence. That pair of nodes merges (either with each other or with one of their neighbors) with high probability within $O(1)$ steps. If an adversary has actively tried to make this merge take long, we can instead apply the upper bound on total round length from before, and get a fringe of size at most $O(\log h)$ per level.

6 Evaluation

Experiments confirm that the SeqHash-based hash function performs well in all cases, while the treap-based hash function degrades with certain inputs.

We used three different data sets: the text of Shakespeare's Hamlet, the binary package file for MacTeX 2014, and SHA512 hashes of the integers. The Shakespeare text

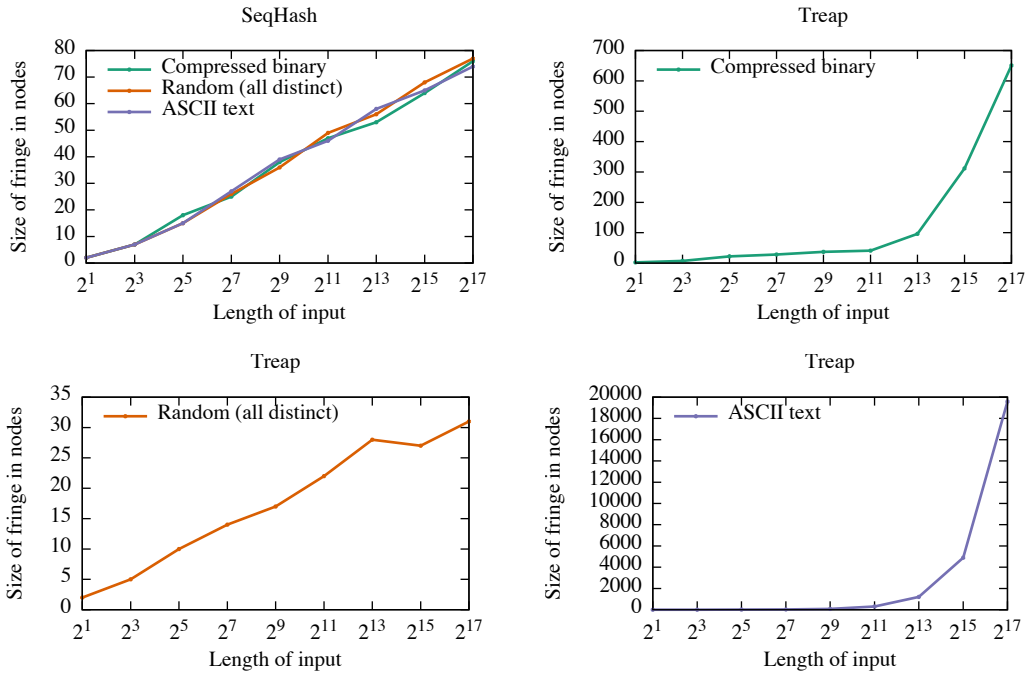

Figure 8: Plots of the size of the hash functions over different data sets with varying input lengths. The SeqHash-based hash function performs similar on all input data sets, while the size of the treap-based hash function depends heavily on input characteristics: The more duplicates in the input, the worse the treap's performance. On both the text and binary input, the size of the treap's hash grows linearly.

represents human-readable text, and the individual inputs are the bytes of the text. The binary package represents compressed random data, and it is also split at the byte level. Both the Shakespeare text and the binary package contain many duplicate inputs, as there are only 256 bytes and many bytes appear very frequently. To test inputs where there are (almost) no duplicates, we used the SHA512 hashes of the integers.

To measure the asymptotic behavior of the hash function, we calculated the size of each hash for prefixes of varying lengths. To measure worst-case behavior, we sampled 20 different hash functions, and took the maximum size of the hash function.

The results show that SeqHash contains a logarithmic number of nodes in its fringe for all inputs, while the treap only performs well on the integers input where there are no duplicates (Figure 8). This aligns well with our analysis of the treap (§3) that only gave guarantees when all inputs were distinct.

Human-readable text contains more duplicates than compressed binary data, as there are fewer characters in the English alphabet than the total number of distinct bytes. Our analysis suggested that the treap would performs worse with more duplicates, as duplicates have the same priority. This is visible in the performance of the treap, which scales worse on the Shakespeare text than on the binary package file.

For the integers, where the treap performs well, the treap seems to perform better than the SeqHash, generating approximately half as many output nodes. However, the total number of hashes is equal, as each treap node contains two hashes while each SeqHash node contains only: Each treap node holds both the value stored in the node, and a hash for the node not stored as part of the fringe. In the text case, the value stored inside of the node is only a single byte and can almost be ignored, but in the case where the treap works well, the value stored in each node is in all likelihood going to be a large 32-byte hash. The SeqHash then performs almost as well as the treap on all data sets, and significantly better on certain real-world inputs.

7 Conclusion

We have shown two associative hash functions constructed from a Merkle hash of a treap and a SeqHash tree. Our analysis predicted that SeqHash would perform well in all cases, while the treap would perform only well with distinct inputs. Our evaluation confirmed this analysis.

References

[1] M. Grassl, I. Ilic, S. Magliveras, and R. Steinwandt. Cryptanalysis of the tillich-zémor hash function. Cryptology ePrint Archive, Report 2009/376, 2009. http://eprint.iacr.org/.

[2] R. C. Merkle. A digital signature based on a conventional encryption function. In *Proceedings of the 7th Annual International Cryptology Conference (CRYPTO)*, pages 369–378, Santa Barbara, CA, Aug. 1987.

[3] M. Naor and V. Teague. Anti-persistence: History independent data structures. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing (STOC)*, Heraklion, Greece, July 2001.

[4] J.-P. Tillich and G. Zémor. Hashing with SL 2. In *Advances in Cryptology-CRYPTO'94*, pages 40–49. Springer, 1994.

[5] J. van den Hooff, M. F. Kaashoek, and N. Zeldovich. VerSum: Verifiable computations over large public logs. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS 2014)*, Scottsdale, AZ, Nov. 2014.