Efficient (Non-)Membership Tree from Multicollision-Resistance with Applications to Zero-Knowledge Proofs

Maksym Petkus*

maksym@petkus.info

Abstract. Many applications rely on accumulators and authenticated dictionaries, from timestamping certificate transparency[1] and memory checking to blockchains and privacy-preserving decentralized electronic money[2], while Merkle tree and its variants are efficient for arbitrary element membership proofs, non-membership proofs, i.e., universal accumulators, and key-based membership proofs may require trees up to 256 levels for 128 bits of security, assuming binary tree, which makes it inefficient in practice, particularly in the context of zero-knowledge proofs.

Building on the hardness of multi-collision we introduce a novel (non-)membership, optionally key-value, accumulator with up to 2x smaller tree depth while preserving the same security level, as well as multiple application-specific versions with even shallower trees, up to 6x smaller depth, that rely on the low-entropy source. Moreover, solving for special case of adversarial attacks we introduce key index variants which might be a stepping stone for an entropy-free accumulator.

Notably, unlike other constructions, e.g., [Woo+14]; [GHW21], this work, although may, doesn't depend on the dynamic depth of the tree which is simpler and more suitable for constant-size ZKP circuits, while ensuring a substantially smaller upper bound on depth.

Efficient in practice construction in the adversarial context, e.g. blockchain, where the tree manager doesn't need to be trusted, i.e., operations can be carried out by an untrusted party and verified by anyone, is the primary goal. Example instantiations are considered, where special treatment is given to the application of representing serial numbers, aka nullifiers, see [BS+14]. Nevertheless, the constructions are self-sufficient and can be used in other contexts, without blockchain and/or zero-knowledge proofs, including non-adversarial contexts, see 5.1.

Furthermore, our findings might be of independent interest for other use cases, such as hash tables, databases and other data structures.

Keywords: dictionary, accumulator, tree, Merkle tree, multi-collision, set membership, non-membership, zero-knowledge proof, zk-SNARK, blockchain, hash table, memory checking.

*This work have been performed while employed at Chronicled Inc.

[1] Mel+15.

[2] Hop+20.

Contents

1 Introduction

Many cryptographic applications rely on accumulators, from certificate transparency [LK12]; [Mel+15] to privacy-preserving decentralized electronic money [BS+14], while Merkle tree and its variants are efficient for arbitrary element membership proofs, non-membership proofs and key-based membership proofs, on other hand, may require a sparse tree of depth 2λ for λ bits of security [Bau04], which makes it inefficient in practice, particularly in the context of zero-knowledge proofs.

While there are constructions, such as Merkle Patricia tree [Woo+14], which are efficient on average in the absence of an adversary, however, it's possible to maliciously factor the keys in order to increase the depth and subsequently slow down the speed of operations on the data structure.

Building on the hardness of multi-collision we introduce a basic trust-less version of (non-)membership, optionally key-value, accumulator with almost 2x smaller tree depth ($\approx \lambda$) while preserving the λ security level. Additionally, we build specialized constructions with even shallower, up to 5+ times, trees that rely on a source of entropy, and authority trees depth of which is 6+ times smaller. Notably, the construction doesn't depend on dynamic tree depth which is suitable for ZKP circuits.

Such an accumulator can be used to represent a database, a global state of a blockchain, a blacklist to prove absence against or keep track of serial numbers[3], aka nullifiers, and privately check or ensure its (non-)existence.

The combination of tools depends on the application and environment, e.g., it can be a server of certificate authority or a blockchain that can use it either as a key-value storage accumulator or to store nullifiers. We focus on blockchain applications but these techniques can be adapted to other contexts.

1.1 Overview and Contributions

We start by considering simple Merkle tree-based constructions (2.1, 2.2) and their limitations, such as (non-)membership proof time complexity which serves as a motivation for improvement. We then analyze the probability of collision in the simple construction, aka birthday collision, and introduce the primary observation that the probability of collision decreases dramatically as we increase the arity of collision (2.3).

Building on the insight the first tree construction sketch based on the hardness of multicollision is introduced (2.4) leading to an almost 2-fold depth reduction compared to the baseline, which is roughly equal to the security parameter λ for adversarial setting where input keys can be sampled maliciously to find multicollisions. While a substantial improvement over the state of the art [Bau04], further improvement is still desired both for ZKP (non-)membership proof complexity and insertion complexity, especially if the tree is stored as part of the blockchain state.

Efficiency improvement proceeds considering the concrete use-case, namely storing the nullifiers, our approaches can be generally described as relying on a random oracle model where randomness can be seen as derived post-factum combined with rate-limiting, such that a malicious actor can't query an oracle before deciding whether to submit the key for insertion, similarly to universal hashing.

Construction based on constraining epoch's sample space is proposed (4.1), it further reduces tree depth by 4-fold compared to baseline (for specific parameters). The solution is suitable for environments with relatively

[3] BS+14.

few users, such as enterprise blockchains, nevertheless, it's still desirable to reduce the depth even further, while ideally being able to support many users.

Next, we turn to the rate-limiting approach with post-factum randomization (4.2), effectively making the number of samples equal to the system throughput instead of the computing power of an adversary. The construction in 4.2.1 relies on the derivation of the key index from the independently randomized input, specifically from a randomized commitment. Such an approach further reduces the depth by 5-fold compared to the baseline for the practical instantiations supporting 2^{45} transactions, it is suitable for systems where each transaction consumes an input that can be randomized, such as [BS+14].

As the next step, we're seeking to support cases where randomized input cannot be provided, starting with commitment bootstrapping (4.2.2) that introduces distinguishability concern, which we address with the requirement of unconstrained key index (dummy nullifier) to fall onto an empty bucket (4.2.3), which paves the way to front-running, i.e., making bucket non-empty before honest user's transaction gets processed. We counteract front-running with the requirement to provide epoch-specific proof of work when no randomized input is used, and relaxing the empty bucket rule to allow for multiple elements, rendering attacks computationally infeasible.

While front-running is addressed, a malicious actor could engage in the distinguishability attack via strategic partial filling of buckets that allows to use statistical methods to assign the likelihood of a nullifier being a dummy and perform broad estimations which we solve through the introduction of key index variants (4.2.3.5) that allow to fall back on another bucket if the corresponding one is busy. While seemingly more complex the membership proof and amortized non-membership proof complexities are essentially the same.

Shifting attention to efficient general-purpose constructions, we introduce authority accumulator (5.1) as an example of a non-adversarial case, or alternatively, a setup where the hash function used cannot be learned by an adversary (e.g., universal hashing, keyed hash function). Following up with VRF-based key index derivation construction (5.2) that is suitable for malicious choices of keys, and a VDF-based construction (5.3) as an additional alternative.

Finally, in section 6 we analyze bucket occupancy to judge average time complexity, concluding that even if there are some high arity collisions, most of the non-empty buckets will contain just a few elements, effectively rendering constant time lookup complexity in the bucket. This finding is of independent interest, such as for the design of efficient hash tables. We also discuss history-independent (6.1) and history-dependent (6.2) approaches in more detail.

1.2 Comparison to Alternatives

Comparison of time and space complexities for λ bits of security (accumulator size is $\mathcal{O}(1)$ for all):

Construction	(Non-)mem. proof upper bound	Lookup	Insert	Tree Size[4]	HI[5]
Sorted Merkle Tree	$2\log n$	$\log n$	n	$2n$	✗
Sparse Merkle Tree[7]	2λ	$\log n$	2λ	$4n$	✓
Sorted Reference Merkle Tree[8]	$\log n$	$2\log n$	$3\log n$	$3n + 3n$	✗
Merkle Patricia Tree[9]	$\approx 1.3\lambda$	$\log n$	$\log n$	$4n$	✓
Jellyfish Merkle Tree[10]	2λ	$\log n$	$\log n$	$4n$	✗
This work, basic construction, section 2.4	λ	1[6]	$\log n$	$2n$	✓
This work, optimized construction 4.2.1	$\log n$	1[6]	$\log n$	$2n$	✓

We're considering the upper bound on the (non)-membership proof since 1) this has an influence on the size of the zero-knowledge proof circuit and 2) higher tree depth can be targeted to perform DoS attacks, i.e., if in the design of the system, the average case is considered ($\mathcal{O}(\log n)$) however the worst-case is staged and exploited by a malicious user.

Note that lookup time complexity doesn't assume an additional indexing data structure, which would trade storage size for faster time. For the tree data structure size estimates, if a key-based lookup structure is needed we account for an additional 2x overhead, assuming a hash table which is approximately the same as storing reference to node's children instead, such as in a sparse Merkle tree.

Sorted reference Merkle tree is impractical without a supporting data structure such as an AVL tree, however, it will increase storage requirement since each node not only stores its value but also pointers to the left child, right child and the parent. Hence, about $4*(2n-1)*\log_2(2n-1)$ bits of storage would be needed for a perfectly balanced AVL tree. This yields at least 41 TiB for $n = 2^{40}$ which is almost the size of the sorted Merkle tree, even having that the actual key values are stored in the sorted Merkle tree only, hence the number of reads doubles, otherwise even more storage is needed. It requires $\mathcal{O}(2\log_2 n)$ on average for lookup versus $\mathcal{O}(1)$ for this work.

Merkle Patricia tree is a substantial improvement over the sparse Merkle tree [Bau04], intuition being the fact that instead of depth increasing proportional to the length of colliding key index prefix, it adds only one more level due to the use of extension node. For an attacker to increase the depth of the tree a set of key indices has to be selected such that if we take one of the indices all of the rest will share the prefix of distinct length with the given one, ensuring that branching is happening at many levels. Therefore, more samples are required on average, meaning that for the same security parameter, the maximum depth of the tree is smaller.

Jellyfish, a variant of the Merkle Patricia tree, on the other hand, simplifies the construction by removing extension nodes altogether resulting in the depth proportional to the length of the shared prefix index.

[4] at full capacity, including supporting data structures
[5] history-independent, see 6.1
[6] amortized
[7] [Bau04]
[8] [LK12]; [Tzi+21], assumes supporting data structure (e.g., AVL tree that references primary tree's leaf indices), otherwise lookup is $\mathcal{O}(n)$
[9] arity of 2 is used for fair comparison; non-membership proof size is empirical, based on randomized simulations of up to 2^{25} key samples
[10] [GHW21]

2 Accumulator from Hardness of Multicollision

2.1 Key-Value Merkle Tree

In the introduction we've mentioned that a naive Merkle tree would have to be 2λ levels tall to achieve λ bits of security, e.g., 256 levels for 128 bits of security, which is due to the birthday paradox. Such an accumulator would hash a key k to obtain an index key $h = \mathsf{H}(k)$ and treat it as a path to the leaf (aka, leaf index) and the leaf itself would represent a flag, e.g., single bit 0 or 1 meaning element is or isn't present in the accumulator.

Inserting an element would entail flipping the leaf's bit from 0 to 1 and updating the parent hashes. This allows us to prove both membership and non-membership since the leaf's location is deterministic, i.e., addressable. Optionally, the leaf can also encode the value corresponding to the key, effectively making it a key-value accumulator.

We can utilize sparse Merkle tree [Bau04] to efficiently compute the initial accumulator and alleviate the storage requirement, however, the maximum supported depth must still be 256 levels, which is expensive in certain contexts, such as in ZKP circuits.

2.2 Sorted Merkle Tree

Another approach is to keep the sorted tree, such that starting from leaf 0 every element is specified in an ascending order. This way we can prove membership as in the classic Merkle tree, and we prove non-membership by revealing two adjacent leaves and proving that the key would be between them if it was in the tree. We would need to account for 2 edge cases where the key in question is lesser than the key in the 0-index leaf, in which case we just show that it's lesser and if it's greater than the last we just show that it's greater.

Such a tree can be shallow, just to accommodate the expected number of inserted elements during its lifetime, however, each insertion operation incurs $O(n)$ time complexity to shift all the greater leaves to the right, which is infeasible in most contexts. Notably, some optimizations can be done, which we'll come back to later.

2.3 Intuition

We start by looking closely into the generalized Birthday paradox, i.e., the probability of multi-collision.

We first observe that the number of samples required on average to reach s-collision, i.e., s people sharing the same birthday, increases close to linearly with s, e.g., while for 2-collision it's 23, 1398 samples are needed on average for 10-collision, see [Ste99].

This leads to an insight: since the "hardness" of multi-collision is higher than of the 2-way collision, if we replace a leaf with a bucket that contains a list of elements that share the same path we might be able to reduce the depth of the tree, while keeping security at the same level.

Here's an example where a 6-collision is necessary to overflow a bucket, a 3-fold increase in s compared to a birthday 2-collision, however, requires 460 samples on average, a 20-fold increase in the number of samples.

In fact, the accumulator from section 2.1 can be seen as a special case of such generalization, where the bucket size is one and we're seeking to avoid 2-collision.

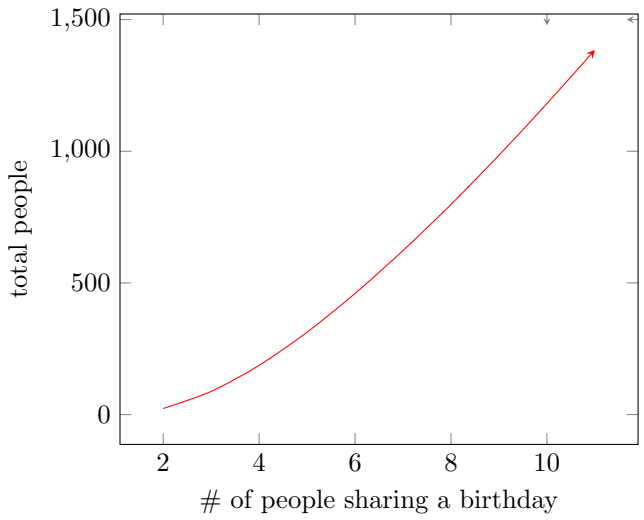

Figure 1: Number of people needed for at least 2, 3... of them sharing a birthday with 50% chance

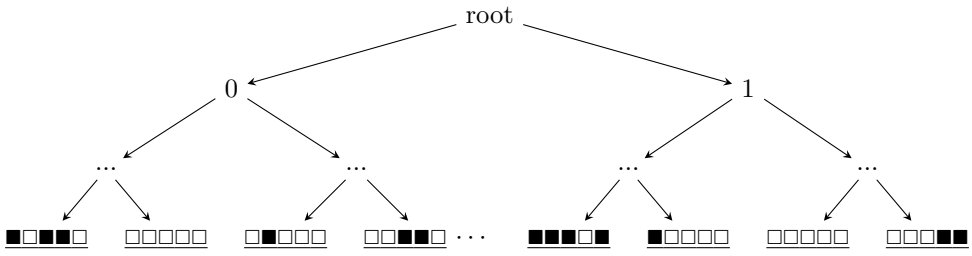

Figure 2: Tree with buckets of 5-element capacity, where ■ - element, □ - empty cell

2.4 General-Purpose Construction Sketch

We need two components to construct the accumulator 1) an addressable binary hash tree with depth d^{A} (A stands for addressable) and a bucket that can hold a set of up to $s-1$ elements, where s is the arity of collision that should be improbable to occur.

To insert a key-value element (k, v) we get the first d^{A} bits of the key index $h = \mathsf{H}(k)$ to signify the path to the bucket $k^{\mathrm{A}} = h_{0..d^{\mathrm{A}}-1}$ in the addressable tree, and we take the rest $|h| - d^{\mathrm{A}}$ bits, i.e., $k^{\mathrm{B}} = h_{d^{\mathrm{A}}..|h|-1}$ to append (k^{B}, v) to the elements of the bucket, consecutively $h = k^{\mathrm{A}} \| k^{\mathrm{B}}$, i.e., a concatenation of the bucket address and the bucket element's key. Notably value v is optional.

The (non-)membership proof is then an authentication path to the corresponding bucket along with the list of all the bucket elements that went into the bucket and requires checking that the bucket element of the key is (or is not) in the list.

In order to set the parameters for security level we need a good approximation of the amount of work necessary to hit the first s-collision with 50% probability which has been explored in [Suz+06]. On average the following amount of work is needed to overflow a single bucket:

$$n^{\frac{s-1}{s}} * (s!)^{\frac{1}{s}} \tag{1}$$

Hence we derive that the amount of work provides $\log_2\left(n^{\frac{s-1}{s}} * (s!)^{\frac{1}{s}}\right)$ bits of security.

Additionally, we can calculate an upper bound probability of s-collisions after work q. Let $C(n, q, s)$ denote an s-collision event having n buckets and q samples (queries), concretely that there's at least one bucket with at least s elements. Using the formula (see [RS09] theorem 2.9 assuming ideal hash function, i.e., $\mu(h) = 1$, [Suz+06] theorem 2):

$$\Pr[C(n, q, s)] \leq \binom{q}{s} * n^{-(s-1)} \tag{2}$$

For example, using a bucket size of 10 we can reduce the depth of the addressable tree to 140 levels. Following is an exposition of how bucket size influences the addressable tree depth while keeping the security level constant at 128 bits.

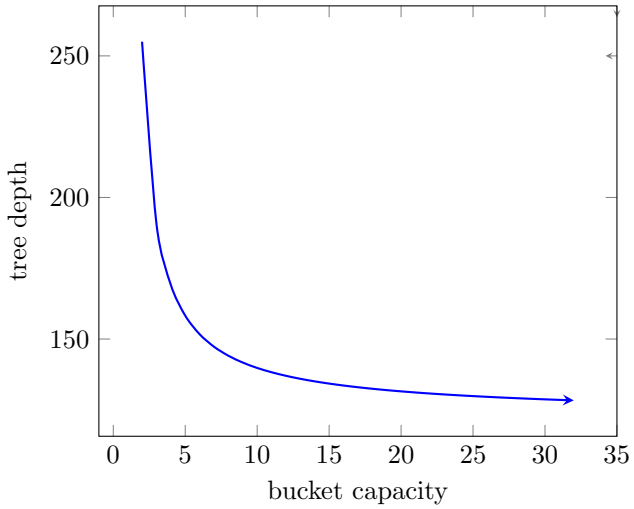

Figure 3: Tree depth as a function of bucket capacity, for 128 bits of security

Notably, with an increase in the capacity of the bucket the depth falls sharply only initially, becoming less influential, therefore, for the actual implementation a tradeoff between tree depth and complexity of the bucket, in terms of storage and proof time and size, shall be made for the specific setup used. For example, for tree depth $d^{\text{A}} = 128$ and the bucket capacity of 32, and the respective bucket part of key index size of $256 - 128 = 128$ bits, the bucket hash would consume 4096 bits of input (assuming there's no associated value), in order to prove (non-)membership of an element alongside with 32 comparisons.

We can improve the construction by sorting the bucket and putting abridged elements into the bucket tree, a sorted Merkle tree of depth $d^{\text{B}} = \log_2(s - 1)$, since computational overhead is negligible for small buckets, as described in 2.2. This would allow us to prove the non-membership of the bucket with $2d^{\text{B}} - 1$ siblings (and hashes) and 2 comparisons with adjacent elements, instead of $s - 1$ elements and $s - 1$ comparisons, reducing the hashing complexity, see figure 4.

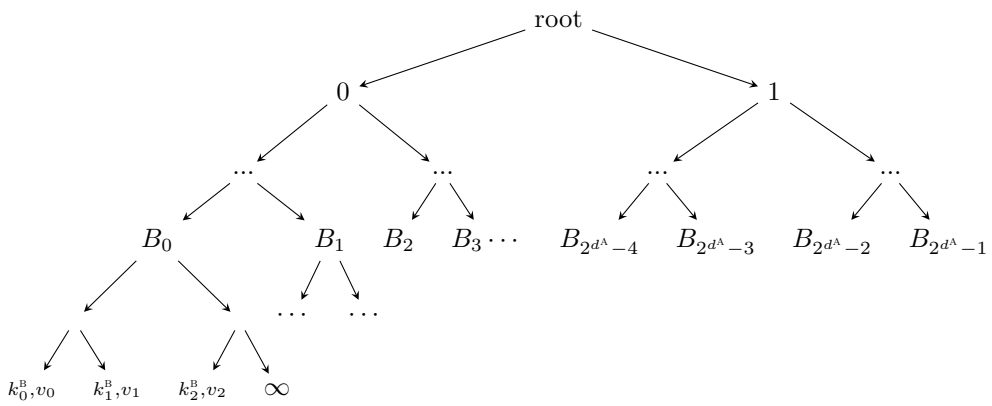

Figure 4: Tree with 4-element capacity buckets

Where B_{k^A} is the root hash of the bucket tree of depth $d^B = 2$ which contains leaves that represent all the index keys that share d^B-bits prefix, i.e., list sort($\{k^B \mid k^A || k^B \in K\}$) where K is the set of all the key indices in the accumulator, $k^A \in \{0,1\}^{2^{d^A}}$ and ∞ signifies default value of an empty leaf.

Yet another improvement is possible through replication of the bucket key from the next leaf in the current leaf (adopted from [GT00]), i.e., a leaf would represent (k_i^B, v, k_{i+1}^B), which is enough information to prove non-membership, and results in almost 2-fold decrease in hash function invocations for bucket tree, therefore the updated non-membership proof in the bucket requires only d^B hashes. Note that we also can optionally store value v associated with the key, in which case it's serialized into a constant bit length either as raw data if it's short or otherwise a hash of the respective data.

Structuring the bucket as a tree allows us to find optimal parameters by trading bucket tree depth for addressable tree depth, see figure 5.

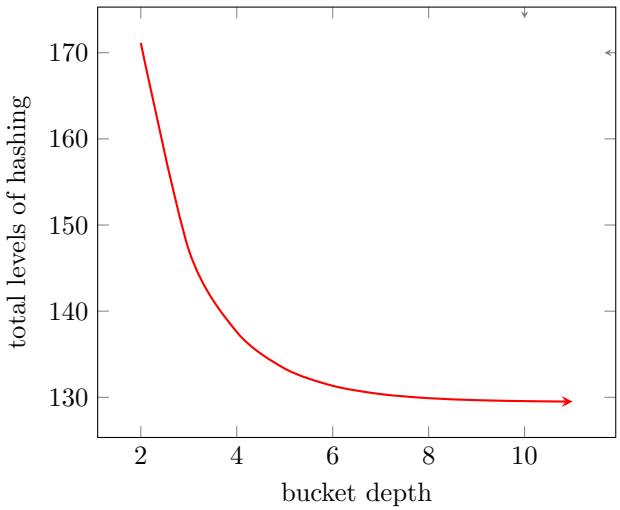

Figure 5: Combined depth dependence on the bucket depth for 128 bits of security

This yields close to optimal parameters, assuming the hash function is perfect, i.e., $\mu_s(h) = 1$, achieving 128

bits of security, meaning that it's infeasible to fill out even a single bucket, with an addressable tree of depth 125 and bucket tree of depth 6, such that non-membership proof requires hashing $125 + 6 = 131$ (or $122 + 8$) levels, with the total tree depth of 131, which would require about $2^{127.72}$ of queries to achieve a multi-collision with 50% probability. That's an almost $2\times$ improvement compared to the original construction in 2.1. Similarly, for 96 bits of security, the combined depth is roughly 99.

From the perspective of storage burden established efficient solutions for sparse Merkle trees can be used with different variations and respective tradeoffs, see [DPP16]. Still, for certain applications, it's preferable to reduce the depth even further.

2.5 Update and Deletion

To make the scheme complete we can trivially add update and deletion functionality. To update the value under an existing key of the bucket element, all that needs to be done is to lookup the leaf using the respective key index h, modify the corresponding value v and update the parent hashes all the way to the root, as it is done in Merkle tree.

For the deletion we first lookup the corresponding leaf $l_i = (k_i^{\mathrm{B}}, v_i, k_{i+1}^{\mathrm{B}})$, and the preceding leaf $l_{i-1} = (k_{i-1}^{\mathrm{B}}, v_{i-1}, k_i^{\mathrm{B}})$, update the reference of the preceding leaf $l_{i-1}.k_i^{\mathrm{B}} = l_i.k_{i+1}^{\mathrm{B}}$, if there's a preceding leaf otherwise we skip this step, and empty the deleted leaf $l_i = \perp$, complete the change by updating hashes of the respective parent nodes. The operation can be performed having either the complete tree data structure or just the membership proof for the leaves (l_{i-1}, l_i), which do share the path at least down to the bucket root. Importantly leaves from the other buckets (e.g., adjacent) need not be considered, i.e., for non-membership proof if =the k^{B} is not in the respective bucket k^{A}, then it can't be in any other bucket.

Note that this is a procedure for the history-dependent deletion (section 6.2), for history-independent deletion (section 6.1), we would need to additionally shift one step left all the non-empty leaves that are to the right of the deleted one in the bucket.

3 Security

3.1 Objectives

In order to improve on the basic construction it's important to specify the security properties.

Adversarial model:

- probabilistic polynomial-time stateful adversary
- exclusive access to the accumulator for its lifetime, i.e., up to its capacity, meaning that it can decide the exact order of its own operations
- has unlimited storage
- computationally bound

The objective, then, is to ensure that the following properties are honored:

- Correctness

  - membership case: it's possible to prove and verify membership of an element that is part of the accumulator with all but negligible probability
  - non-membership case: it's possible to prove and verify non-membership of an element that isn't part of the accumulator with all but negligible probability
- Soundness: the probability of proving (non-)membership for an element that isn't (is) part of the accumulator is negligible
- Availability: given a key that is not part of the accumulated set, the probability of successfully adding the key and its respective value to the accumulator is all but negligible. Specifically, the key has to fall onto a bucket with at least one leaf still available to be filled, otherwise, it's a bucket overflow and the operation is rejected

3.2 Building Blocks

It's worth noting, that although we reduced the depth of the tree, the key index $h = k^{\mathrm{A}} \| k^{\mathrm{B}}$ is still represented in full, i.e., partly as the address of the bucket and partly as a suffix in the bucket's leaf, inheriting the birthday collision resistance-based security. Hence, correctness and soundness are addressed, as long as the hash function used to derive h is secure, regardless of the depth of the tree. Therefore, it remains to show that the probability of bucket overflow is negligible.

For availability, we relax the definition to signify overflow of any bucket, concretely, for the choice of addressable tree depth d^{A}, bucket tree depth d^{B}, security parameter λ and $q \leq 2^{\lambda}$ work of PPT adversary the probability of overflow of any bucket is negligible:

$$\Pr\left[C\left(2^{d^{\mathrm{A}}}, q, 2^{d^{\mathrm{B}}} + 1\right)\right] \leq \mathsf{negl}(\lambda) \tag{3}$$

It naturally follows from the upper bound in (2). Thus, we rely on the following building blocks:

1. Random oracle model (ROM) to represent the uniform distribution of the hash function $\mathsf{H}()$
2. Collision-resistance to ensure infeasibility of collision of the key index
3. Multi-collision resistance to ensure infeasibility of bucket overflow

Although we rely on ROM, in practice it's instantiated with a suitable hash function. The choice of function is unlikely to be ideal, however, the notion of s-balance from [RS09] can be utilized as a compensatory measure.

4 Efficient Constructions for Nullifiers and Combined Keys

We start the optimization process by considering an accumulator for serial numbers[11], also known as nullifiers[12], which is an important cryptographic primitive in privacy-preserving blockchains and it's useful to keep nullifiers in an accumulator for applications such as privacy-preserving rollups. More generally, however, this section covers use-cases where the key index is deterministic but isn't required to be derived only from the

[11] BS+14.
[12] Hop+20.

user's input, i.e., external input is also used, and nullifier is a great example of that, e.g., in [Hop+20] to derive nullifier not only note commitment data itself (user input) but also its position in the note commitments tree (external input) are used together. While we focus on the specific practical application, the following techniques can be useful in other settings and we'll apply them for the general case.

Even though we had a major improvement in terms of depth, the number of levels of the resulting tree is still non-trivial, especially compared to the realistic number of members of the accumulator in its lifecycle.

We come from the angle of blockchains that utilize zero-knowledge proofs for the established "first commit, then reveal unlinkable serial number (nullifier) pattern" (see [STS99]; [BS+14]) where the nullifier may or may not be dummy (see [Hop+20]), i.e., there's no record/note that's being consumed.

All of the following options rely on the notion of the existence of the unbiased source of entropy, i.e., such that an adversary cannot influence it and its output is uniformly distributed. Therefore, a randomness beacon or distributed verifiable random function are potential choices ([HMW18]; [Gal+21]), however, similar techniques can be used in other environments and architectures, including centralized solutions. Within the environment, we focus on the practical application of accumulators for nullifiers which can yet again be mapped onto other use cases without loss.

Reducing the depth of the tree entails a reduction in the work complexity necessary for an attacker to either target specific buckets or successfully generate a random s-collision. Therefore our approach is to bound the amount of work an attacker can perform in the lifetime of the accumulator.

As for randomness we assume that it is derived from a cryptographically secure source of entropy and the adversary has no control nor influence on it.

4.1 Constrained Epoch Sample Space

The approach is to constrain the user's sample space, so that only a relatively insignificant amount of work can be done in the lifetime of the system, at the same time we need to make sure that the user's anonymity is preserved. Assuming the number of users is relatively small, a user can derive the nullifier from their own namespace and a nonce η.

To ensure the privacy of the user and unlinkability of its transactions the namespace will be represented by a secret seed uniformly derived at random. User pre-commits to the secret random seed ς in a setup transaction using a cryptographic commitment, hence the nullifiers of the same user, derived from the seed ς and nonce η are indistinguishable from random, assuming that the pseudorandom function family $\mathsf{PRF}()$ is pre-image and collision resistant, we can derive nullifier as $\mathsf{PRF}_{\varsigma}(\eta)$.

This still allows an adversary to attempt to pick the secret seed such that in combination with a nonce it'll yield desired nullifier properties, such as mapping on a specific subset of buckets.

Therefore, as a countermeasure, the ledger operation is split into epochs of constant duration and at the beginning of each epoch, an epoch randomness ϵ_i is generated from a source of entropy. A new seed commitment, submitted by the user at epoch t, does not become effective until the next epoch randomness is generated, hence, one can't predict what ϵ_{t+1} will be at the time of submission, which extends to inability to know in advance the set of nullifiers that will be available in any of the future epochs, therefore, the seed ς can't be purposefully factored anymore to yield desired nullifier(s). Once committed, the secret seed can be used indefinitely until the user chooses to update it.

The nullifier is derived from $\mathsf{PRF}_{\varsigma}(\epsilon_i \| \eta)$, (where i is the latest epoch number) and is a part of the value of the record or note commitment (see [BS+14]) that's revealed when the record is nullified (also consumed, spent). In case the input commitment is a dummy (see section 4.7 of [Hop+20]) it's desirable for the transaction to look indistinguishable from the transactions consuming existing commitments and, therefore shall be accompanied by a nullifier, hence, instead of being an opening from the note commitment it is freshly derived the same way it would've been derived for the note commitment using the current epoch's randomness.

We claim that this is an effective solution as long as the number of users $|U|$ and therefore pre-committed random seeds is relatively small, hence assuming that an attacker controls all of the users the sample space is limited to $|U| * |N|$ per epoch, where N is the set of nonces available. The total number of samples in the lifetime of the system is $|U| * |N| * |E|$, where E is the set of all of the epochs.

Henceforth, we can rely on concrete security, namely the construction is (q, ε)-secure if any adversary sampling at most $q = |U| * |N| * |E|$ queries succeeds in overflowing a bucket with probability at most ε. This also means that the capacity of the tree is limited to q elements.

For instance, assuming that 1 transaction has 1 nullifier, if there are at most 2^{16} users, the nonce can be chosen from $\left[0, 2^{18}\right)$ and an epoch is 10 minutes long, in 1000 years there will be about 2^{26} epochs, henceforth, the upper bound on total number of samples is $2^{16} * 2^{18} * 2^{26} = 2^{60}$. Note that epoch randomness is only relevant during the epoch, i.e., no one can post transactions using an expired epoch randomness retroactively, hence there will be fewer samples in reality, which we acknowledge as a margin of safety.

Hence, for scheme to be $(2^{60}, 2^{-128})$-secure, we may choose the depth of the addressable tree to be $d^{\mathrm{A}} = 56$ and the depth of the bucket to be 7, in which case the probability of $(2^7 + 1)$-collision in the lifetime of the system, 2^{60} samples, is no higher than 2^{-151}, i.e., $\Pr\left[C(2^{56}, 2^{60}, 2^7 + 1)\right] \leq 2^{-151}$ (as per upper bound in (2)). This brings the combined depth of the tree to 63 which is about 4 times shallower than the original construction.

Notably, a user is allowed to modify his random seed ς, however, importantly, one cannot use it until the next epoch which will bring an unpredictable ϵ, therefore, factoring the seed to facilitate multi-collision is infeasible.

This approach, however, may not be suitable for certain applications since the nullifier of the commitment is known by its creator, instead in some cases it's desired to be known only to the owner of the commitment, e.g., the recipient, that's why the private key is also sometimes used (see [BS+14]) to derive the nullifier, otherwise the sender may be able to learn when recipient's commitment is consumed. The user should also be careful to not re-use the same nonce within an epoch, which would render one or more of the commitments unusable. The collision-prevention approaches, such as combining nullifier arguments with the position of the commitment might be used but also will extend the number of possible samples since in the adversarial setting the position can be chosen to gain an advantage.

On the other hand, the approach requires a relatively small amount of entropy, since the ϵ is generated only once per epoch, therefore is suitable for limited entropy environments.

4.2 Post-Factum Randomization

The idea is to make the key index (e.g., nullifier) unpredictable at the time of committing to all of the user's inputs and infeasible to retroactively modify it, effectively limiting the number of samples to the upper bound

of the transaction throughput of the system, while keeping them uniformly distributed. We achieve this through the introduction of post-processing of the user's input (transaction) which binds external randomness, e.g., a random beacon [HMW18]; [Gal+21], to it and is used to eventually derive the nullifier. In practice, this can be achieved either by re-randomization of the respective commitment before storing, e.g., through the homomorphic property of Pedersen commitment [Ped92], or via association of commitment with the respective randomness r_{cm}, e.g., storing the $(\mathsf{cm}, r_{\mathsf{cm}})$ pair.

While this approach covers nullifiers derived from existing commitments, keeping the dummy nullifiers unconstrained opens up a gateway for collision attacks, therefore, we'll consider three constructions with their respective tradeoffs.

4.2.1 Nullifiers from Randomized Commitment

We start with the basic case where the commitment creation undergoes the following stages:

1. User generates transaction with original commitment cm as usual and posts it to the ledger
2. If valid, the commitment is randomized when the transaction is processed within a block with the block randomness σ_b, where b is the block number, $\mathsf{rcm} = \mathsf{RCM}_{\sigma_b}(\mathsf{cm})$ which is stored instead of the cm.

When the commitment is consumed in one of the future transactions the randomized version will be used in order to construct the nullifier, e.g., $\mathsf{PRF}_z(\mathsf{rcm})$, where z is the application-specific data bound to the commitment cm, such as commitment position, private key or nullifier key [Hop+20].

The solution assumes that adversary has no control over, or ability to predict, the external randomness (e.g., block randomness), which, for example, can be achieved with the use of a distributed random beacon, currently deployed as part of multiple notable blockchain networks. Since the joint randomness is unknown at the time of transaction generation, the number of samples an attacker can try is effectively limited by the throughput of the system.

Therefore, the best an adversary can do is produce as many transactions as possible to try and generate an s-collision, hence the amount of work one can do is proportional to the number of transactions. If the throughput of the system is l transactions a year and its expected lifetime is y years then an attacker is bound by $\mathcal{O}(l \cdot y)$ of work. Therefore in the choice of parameters, we need to ensure that $n^{\frac{s-1}{s}} * (s!)^{\frac{1}{s}} \gg l \cdot y$ (formula (1)), i.e., the probability of an s-collision is negligible.

For example, assuming that 1 transaction = 1 nullifier, if the system can process 1000 transactions/sec then in 1000 years there will be no more than 2^{45} of nullifiers, therefore having 41 levels in the addressable tree and 7 in the bucket tree, the probability of $(2^7 + 1)$-collision is no more than 2^{-166}, as per (2), i.e., $(2^{45}, 2^{-128})$-secure. That's the total of 48 levels of the tree, 5.3 times smaller than the naive, addressable tree-only, construction.

There's one caveat, however, if we want to also accommodate dummy nullifiers, i.e., nullifier for a dummy note commitment that doesn't exist (see section 4.7 of [Hop+20]), which is important to achieve transaction indistinguishability on number of commitments consumed, i.e., in simplest case, an observer should not be able to deduce whether the transaction creates completely new commitment or consumes an existing one.

In the case of Zcash, and similar protocols, at least one of the input notes will be non-dummy to pay the transaction fee, therefore the dummy nullifiers can be derived from the first non-dummy randomized input commitment. For our example, we will consider the JoinSplit transfer case with 2 inputs and 2 outputs.

We modify the derivation of nullifier to $\mathsf{PRF}_z(\delta \| \mathsf{rcm})$, where $\delta \in \{0,1\}$ is a single bit signifying whether the respective nullifier is dummy or not. This means that 2 distinct nullifiers can be derived from each non-dummy commitment. Adversary, trying to exploit this, would maximize the number of nullifiers available to be exposed by always consuming 1 non-dummy commitment, exposing 2 nullifiers and producing 2 new commitments, and so on with every transaction. Therefore, each transaction consumes a single commitment, exposes 2 nullifiers and produces 2 new commitments which in turn can be used to expose 4 nullifiers and to produce 4 new commitments and so on. This process can be modeled with a binary tree where each node represents a consumed commitment (and transaction) which exposes 2 nullifiers and produces 2 new commitments and all the leaves represent all of the unconsumed commitments with 2 respective nullifiers each ready to be selectively exposed. Hence, we can express the total number of nullifiers through the number of transactions, counting 2 nullifiers for each node and 2 nullifiers available in each leaf, i.e.:

$$2 * |T| + 2 * (|T| + 1) \Rightarrow$$
$$4 * |T| + 2$$

Where T is the set of all of the transactions in the lifetime of the ledger.

Going back to 2^{45} transactions upper bound, this translates to $2^{47} + 2$ nullifiers where $2^{46} + 2$ of them not yet exposed, however, available to the adversary to choose from. This increase can be countered by the addressable tree depth of 43, which translates to 2^{-164} upper bound probability of $(2^7 + 1)$-collision, yielding the combined tree with 50 levels.

Contrary to the solution in 4.1, this one is less restrictive and allows anonymity of the recipient from the sender and other modifications, but it needs new randomness every block. However, it is amenable to variable levels of entropy, such that if entropy is not sufficient at the time of transaction processing it can be accumulated until the desired level is reached and only then finalized, effectively increasing the latency.

What if the application does not have the input commitment in certain cases? That would remove the foundation from which to derive the dummy nullifier. We'll explore the remaining solutions that address the challenge.

4.2.2 Bare Commitment Bootstrapping

For certain applications, we ought to treat general case where none of the input commitments might exist.

We can accommodate both dummy and real nullifiers with a 2-step flow:

1. initially new commitment cm is submitted without the nullifier, randomized as $\mathsf{rcm}^{\text{new}}$ and left in pending state on the ledger
2. follows up with another transaction where the user is constrained to either deterministically derive the real nullifier from the randomized consumed commitment (if it exists) $\mathsf{PRF}_{z^{\text{old}}}(0 \| \mathsf{rcm}^{\text{old}})$ or deterministically derive the dummy nullifier from the created commitment's randomization $\mathsf{PRF}_{z^{\text{new}}}(1 \| \mathsf{rcm}^{\text{new}})$, which would move $\mathsf{rcm}^{\text{new}}$ from pending state and make it available for consumption

The approach, however, requires 2 transactions to execute the operation which is inconvenient and might leak some additional information, e.g., through analysis on how fast one follows up with the second transaction.

Hence we can simplify through an introduction of bootstrapping transaction that each user needs to do once (although not limited) which gives a randomized commitment that can be used to transact as usual, unless the user was the recipient of a commitment before in which case bootstrapping can be avoided.

The user starts by preparing and submitting a transaction with 0 inputs and one or more output commitments. The output commitment(s) get randomized and stored on the ledger.

These commitment(s) can then be used as a regular transaction similar to 4.2.1. Even if one of the next transactions doesn't have an input commitment, one of the randomized commitments can be supplied to derive the dummy commitment from, which would not add an extra membership proving cost for constant-size circuits, since an unused slot of input commitment membership proving can be used. As long as randomized commitment wasn't used to derive a dummy nullifier yet, it can be used to construct regular transactions with zero inputs.

As in the section 4.2.1, if an adversary chooses to maximize the number of nullifiers (samples) in 2 inputs 2 outputs case, the total number of nullifiers will be the same less the 2 nullifiers that will not be present in the bootstrapping transaction, i.e.:

$$4 * |T|$$

The downside is that the bootstrapping transaction publicly indicates that there are no inputs as well as that it can be used to derive an approximate number of users, although a user might choose to run multiple such transactions either for convenience or to obscure the picture, moreover as mentioned users might bootstrap by receiving commitments from another user.

It also might be necessary to keep track of the consumed commitments, for which dummy nullifiers have not been exposed yet, to seed the future dummy-input transactions.

Depending on the application the approach might be a reasonable tradeoff, for when it's not see the following solution.

4.2.3 Empty Bucket Rule

It is still might be desirable to retain transaction indistinguishability, hence we'll address it in this section.

An easier alternative is to remove the constraint on the dummy nullifier, introduced in 4.2.1, but require it to fall on an empty bucket instead, thereby ensuring that the newly introduced freedom can't be exploited to target buckets, i.e., at most, the adversary can easily produce only a single such element.

The way to enforce this is to provide the public signal to the zero-knowledge proof circuit, say $\tau \in \{0, 1\}$ that will communicate whether the target bucket is empty (0) or not (1). The user will specify τ at the time of proof generation based on the current state of the accumulator and the ledger will verify that the τ corresponds to the current state of the bucket in the accumulator.

The circuit will enforce that for the dummy nullifier the bucket must be empty, i.e., $\delta \to \tau = 0$, while for the regular commitment-derived nullifier it'll not be enforced. The nullifier will still be derived as $\mathsf{PRF}_z(\delta \| \mathsf{rcm})$ for non-dummy commitment, however, in the dummy nullifier case the rcm will be enforced to be 0 and z will be the arbitrary value provided by the user, which will be generated at random.

Hence, it's allowed to use any, user-generated, dummy nullifier as long as it's not falling on a non-empty bucket, therefore, it shall be ensured that empty buckets are always available during the lifecycle of the ledger,

which means that the total number of buckets shall be greater than the upper bound of the number of nullifiers. To produce a suitable dummy nullifier user would sample z at random, derive the nullifier and check if the corresponding bucket is empty, if it's not, sample another z and so on until the empty bucket is yielded.

Another important factor is the prospect of nullifier bucket collision with the nullifiers of all other transactions concurrently generated by other users because when it happens the first nullifier to be stored will make the bucket non-empty, thereby the conflicting nullifiers will be rejected because of the mismatch, $\tau \neq 0$. When it happens the user's action will depend on the case:

1. if the nullifier was for an existing commitment, prepare the same transaction with the non-empty bucket flag set, $\tau = 1$, and submit to the ledger, this time it will not be rejected even if there's a collision;
2. for dummy nullifier, sample new z from scratch as described above and submit again.

We can observe two issues:

- inconvenience, it might require multiple transactions to execute a single operation;
- privacy degradation, if following the rejection due to collision there's no follow-up transaction with the same nullifier, it may be concluded with non-trivial confidence that the nullifier was dummy.

Thus, it's desirable to make the probability of such collision negligible. Consequently, we need to consider an upper bound of the number of competing nullifiers, i.e., the number of samples, and the number of empty buckets, i.e., the sample space.

Similarly to existing privacy-preserving protocols, such as Zcash, we limit the duration of time ϵ_{T} a transaction has from its construction to being processed by the ledger, if the duration is exceeded the transaction is rejected. We'll consider ϵ_{T} to be the range of time any given nullifier is competing with other nullifiers for empty buckets, which is usually several minutes long, although in practice a transaction is processed much faster. Moreover, we'll assume the worst-case scenario: all of the competing nullifiers are dummy, i.e., all of them are trying to target empty buckets.

Our goal, then, is to make the probability of collision within the ϵ_{T} timeframe negligible. Therefore, we turn back to the idea of the system throughput. If the system can handle at most θ transactions per second, assuming we have 2 nullifiers per transaction, that sets an upper bound on the set of competing nullifiers to $\epsilon_{\mathrm{T}} * \theta * 2$. For example if, as before, the $\theta = 1000\,\mathrm{tps}$ and transaction expiration is set $\epsilon_{\mathrm{T}} = 10\,\mathrm{min.}$, that would translate to $2^{20.1946}$ nullifiers.

Using square root approximation we derive that the sample space needs to be larger than 2^{40} just to keep the probability of single collision within ϵ_{T} at 50%, however, to achieve the negligible probability of 2^{-128} or less the sample space has to be about 2^{167}, hence $d^{\mathrm{A}} = 167$, which can be calculated using the birthday collision probability approximation:

$$1 - e^{-\frac{\left(2^{20.1946}\right)^2}{2*2^{167}}} \approx 2^{-128}$$

This, obviously, isn't compatible with our goal of keeping the accumulator small, however, we can reuse our insight into the hardness of multi-collision yet again and instead of enforcing the target bucket for the dummy nullifier to be empty we can enforce it to have less than the certain number of elements a (allowance), such that the probability of a-collision is negligible.

The ledger logic will depend on τ of the transaction:

- if bucket declared as non-empty $\tau = 1$, accept only if bucket is indeed non-empty;
- if $\tau = 0$, accept only if there are less than a elements in the bucket;
- reject in all other cases.

The change will allow an adversary to pretend that the bucket is empty and feasibly fill up buckets of choice, but only up to $a - 1$ elements, the rest will be reserved for use by regular nullifiers. Consequently, we need to account for the reduction of reserved space in our choice of the size of the bucket to keep the security at the former level.

To sum up, we need to adjust the sizes of the addressable tree and the bucket tree such that:

1. for an honest user, while sampling a dummy nullifier, it should consistently fall on an empty bucket with high probability, i.e., it shouldn't take more than a few samples on average;
2. probability of collision with all other nullifiers within ϵ_{T} is negligible.

4.2.3.1 Example Instantiation

As for the first point we might want around 99% probability that an honest user's dummy nullifier falls on an empty bucket after just 3 attempts. We can find the maximum acceptable probability of the bucket being busy in any given sample as $1 - (\Pr[\text{busy}])^3 = 0.99$, hence, $\Pr[\text{busy}]$ shall be around 0.215 to achieve the desired property, meaning that the accumulator shall have about 4 empty buckets for each non-empty bucket. Two nullifiers per transaction case yields about $2^{45.85}$ nullifiers in the lifetime of the ledger, which in the worst case scenario means that $2^{45.85}$ buckets will be non-empty. Hence, if we pick the addressable tree height to be 48 that would yield $\Pr[\text{busy}] \approx 0.224$ when the accumulator is at its peak capacity.

Picking $a = 7$ for the second point, yields $2^{-158.9}$ probability of an honest a-collision in ϵ_{T}, which for the bucket of size 2^6 leaves 58 elements unaffected, hence an $(s-a+1)$-collision probability is no more than 2^{-345}.

4.2.3.2 Front-running Problem

While we made a-collision and $(s - a + 1)$-collision close to impossible in nominal circumstances, because we also made the dummy nullifier unconstrained, an adversary would need to perform about 2^{43} of work to find a random 7-collision for our example parameters. Notably, depending on the setting, it might be a concern, since to prevent an honest user from successfully transacting with a nullifier(s) that falls on an empty bucket, an adversary would need to:

- observe the transaction's nullifiers before it's processed by the ledger;
- do the work to sample an a-collision;
- compute zero-knowledge proof, form transactions with at least $a - 1$ nullifiers;
- ensure that the front-running transactions will be executed before the honest user's transaction.

However, an adversary may be able to perform the sampling and the proof computation in advance, hence it'll be up to the adversary's network connectivity advantage and the order of transaction processing on the ledger.

Assuming transactions are processed in the order they were received, block producers have low-latency high-speed connection and immediately share incoming transactions, users are communicating transactions directly to the block producers and adversary can't eavesdrop on an honest user's transaction communication channel

nor can it artificial delay dissemination of the transaction to the block producer, front-running becomes less of a concern, especially since adversary needs to get multiple of its transactions executed for the front-running attack to be successful.

On the other hand, if the user's transaction has to make multiple hops through mempools of different nodes before getting to the block producer, adversary could use one's connectivity and speed advantage to get its transactions to the block producer faster, additionally, if transactions are ordered based on some other criteria, such as fee amount, it might make it even easier.

4.2.3.3 Addressing Front-running

To tackle front-running we first ensure that an adversary can't precompute a colliding transaction in advance through re-introduction of the epoch randomness ϵ (see section 4.1), such that the user's zero-knowledge proof now have to attest that the dummy nullifier is derived from $\mathsf{PRF}_z(1\|\epsilon_i)$ where z is arbitrary user input and ϵ_i is the active epoch randomness which expires after ϵ_{T} (same as transaction expiration, for simplicity).

Additionally, to give an advantage to the user over an adversary the protocol will constrain z, where instead the user will specify arbitrary z' and will be required to provide proof of work (PoW) [DN93] for the choice of randomness z' and the active epoch randomness ϵ_i (constituting the challenge), such that z, and thus, nullifier depends on the result of the proof of work, hence the nullifier is unknown until the work is performed, i.e., $z = \mathsf{PRF}_{z'}(\pi_{\mathrm{POW}})$, such that $\mathsf{POW}^{\mathrm{verify}}(\epsilon_i\|z', \mathsf{diff}, \pi_{\mathrm{POW}}) = 1$ where π_{POW} is the proof of work, diff is the difficulty. Thus, the adversary must perform the work for each nullifier they sample to identify whether it's suitable for the attack.

Such construction ensures that the adversary can't precompute the work before the epoch starts, and is required to perform significantly larger computation than an honest user, even when proof of work is negligible, i.e., on the order of tens of milliseconds.

As we've established already in a nominal circumstance a-collision is next to impossible, hence if it does happen more likely than not there's malicious behavior. Therefore, the difficulty diff can be adjusted automatically by the ledger when $(a-1)$-collision does happen. To achieve this the first time an element is added to a bucket the timestamp can be stored, later when the bucket reaches the size of a elements it can be checked whether ϵ_{T} time has not yet passed since its first element, and if so, incrementally increase the difficulty allowing for the grace transition period and making sure not to issue another increase until the new difficulty went into effect. Notably, such situations may be mostly avoided altogether through dynamic difficulty correlation to Moore's law [Wik23c].

As before, affected transaction(s) of the honest user(s) can be re-generated and submitted again, hence successful front-running is just delaying the processing of the user's intent but not preventing it.

In the choice of parameters, we shall ensure that it takes adversary more than ϵ_{T} time to find an a-collision, while simultaneously it takes less time for an honest user, hence rendering adversary efforts futile. To achieve this we adopt an insufficient advantage principle, meaning that even if the adversary has up to α times more computational power than a user it is still unlikely to produce the multicollision within the timeframe. Formally, if it takes on average o_{POW} computational operations to produce a single proof of work and the user's hardware can perform R_{u} operations per second we have (where q_ϵ is the number of samples needed to attain probability p of an a-collision):

$$\frac{o_{\mathrm{POW}}}{R_{\mathrm{u}}} \ll \epsilon_{\mathrm{T}} < \frac{q_\epsilon * o_{\mathrm{POW}}}{\alpha * R_{\mathrm{u}}}$$

Notably, the time it takes the user to produce the proof of work shall be way less than ϵ_{T}. We can update the relation by claiming that it should take the user at least m times less time than an α times more powerful adversary.

$$\frac{o_{\mathrm{POW}}}{R_{\mathrm{u}}} \leq \frac{q_\epsilon * o_{\mathrm{POW}}}{m * \alpha * R_{\mathrm{u}}}$$

Which can be simplified to $m * \alpha \leq q_\epsilon$, meaning that the adversary cannot reach probability that makes it feasible to sample a-collision within an epoch. We are making an assumption that as the computation power, that can be used for malicious purposes, grows so does the computation power of a user, hence the proof of work difficulty can be proportionally increased.

If the epoch randomness ϵ_i lasts for ϵ_{T} time, it's convenient for epochs to overlap so that transactions that are generated closer to the end of the epoch have enough time to be processed before expiring, hence epochs are released with κ cadence, i.e., every κ units of time. For a computationally bound adversary, to maximize its chances of success, it's beneficial to use the latest epoch randomness as soon as it's available to ensure that samples will stay relevant as long as possible, compared to using the oldest relevant epoch randomness which would render the work useless faster. Hence, it's akin to a sliding window, where when new randomness ϵ_i is issued the oldest relevant randomness $\epsilon_{i-\frac{\epsilon_{\mathrm{T}}}{\kappa}}$ expires, thus, adversary can discard the uncommitted samples tied to $\epsilon_{i-\frac{\epsilon_{\mathrm{T}}}{\kappa}}$ and start using ϵ_i for the upcoming workload.

Consequently, considering the case just before the new epoch, assuming a steady sampling rate, the size of the adversary's relevant samples pool stays constant (i.e., progress done in ϵ_{T}) with equal amounts added and removed as the previous epoch expires and gives way to another one, however, its composition changes as above, hence we'll consider every such instance of pool composition as new.

To reflect the change in setting, we introduce the notion of $(q_s, q_\epsilon, \varepsilon)$-security, meaning that after q_s randomized samples (regular nullifiers) in lifetime of the accumulator, and up to q_ϵ samples at adversary's disposal per epoch duration, the probability that adversary succeeds is no higher than ε. Concretely:

1. $\Pr\Big[C\Big(2^{d^{\mathrm{A}}}, q_s, s-a+1\Big)\Big] \leq \varepsilon$
2. $\Pr\Big[C\Big(2^{d^{\mathrm{A}}}, q_\epsilon, a\Big)\Big] \leq \varepsilon$
3. $1 - \Big(1 - \Pr\Big[C\Big(2^{d^{\mathrm{A}}}, q_\epsilon, a\Big)\Big]\Big)^{|E|} \leq \varepsilon$
4. $2^{d^{\mathrm{A}}} > q_s$, to ensure availability of empty buckets
5. $\frac{o_{\mathrm{POW}}}{R_{\mathrm{u}}} \ll \epsilon_{\mathrm{T}} < t^{\mathcal{A}}_{q_\epsilon}$, where $t^{\mathcal{A}}_{q_\epsilon}$ is time it takes adversary to obtain q_ϵ samples

Point 3 means that in the lifetime of the accumulator, a new a-collision within any epoch isn't feasible, as opposed to any a-collision across epochs which can happen, however since honest users' dummy nullifiers fall onto empty buckets those long-range collisions are inconsequential.

We ought to also consider unexposed nullifiers for unconsumed commitments an adversary has control over since they can be used to aid in the search of the multicollision, therefore we'll account for them in the adversary's relevant samples pool.

Let's examine what parameters are reasonable to instantiate with. For this setting, we'll use 2 inputs and 2 outputs transitions such that one, regular, nullifier can be derived from each commitment. That means that in order to maximize available unexposed nullifiers an adversary can produce 2 commitments per transaction

with both inputs being dummy, which yields $|T| * 2$ dummy nullifiers exposed and $|T| * 2$ unexposed regular nullifies at the end of the ledger lifecycle. Also, within ϵ_{T} we expect up to $\epsilon_{\mathrm{T}} * \theta * 2$ nullifiers from honest users which target empty buckets, to be processed, which contribute to the probability of a-collision. Thus:

- $q_s = |T| * 2$, i.e., max cumulative quantity of regular exposed and unexposed nullifiers
- $q_\epsilon = |T| * 2 + \epsilon_{\mathrm{T}} * \theta * 2 + q^{\mathcal{A}}_{\epsilon_{\mathrm{T}}}$, a respective sum of
  - max unexposed nullifiers, i.e., no commitments were ever consumed
  - nullifiers from all of the transactions in the epoch, assuming worst case: at capacity and all targeting empty buckets
  - $q^{\mathcal{A}}_{\epsilon_{\mathrm{T}}}$ is an upper bound on the number of samples adversary can draw in ϵ_{T}

Notably, in q_ϵ we don't account for already exposed nullifiers since users would ensure that dummy nullifiers fall onto empty buckets while regular nullifiers stay unaffected.

Having 1000 years of operation at $\theta = 1000$ transactions per second, $\epsilon_{\mathrm{T}} = 10\,\text{minutes}, \kappa = 1\,\text{min.}$, hence $|T| = 3.1536 * 10^{13}, |E| = 5.256 \times 10^8$, adversary that is $\alpha = 2^{40}$ times more powerful and takes $m = 2^{13}$ times more time to draw $q^{\mathcal{A}}_{\epsilon_{\mathrm{T}}}$ samples, hence, $q_s = 2^{46}, q_\epsilon \approx 2^{53}$. For the choice of $\varepsilon = 2^{-128}$, the $(q_s, q_\epsilon, \varepsilon)$-security can be achieved with the following configuration:

$$d^{\mathrm{A}} = 50,\ d^{\mathrm{B}} = 7,\ a = 103,\ s = 27,\ q^{\mathcal{A}}_{\epsilon_{\mathrm{T}}} = 2^{53},\ \frac{o_{\mathrm{POW}}}{R_{\mathrm{u}}} = 250\,\text{ms} \tag{4}$$

The configuration yields these concrete results:

1. $\Pr\big[C(2^{50}, 2^{46}, 27)\big] \leq 2^{-151}$
2. $\Pr\big[C(2^{50}, 2^{53}, 103)\big] \leq 2^{-185}$
3. $1 - \big(1 - 2^{-185}\big)^{5.256 \times 10^8} < 2^{-156}$
4. $2^{50} > 2^{46}$
5. $250\,\text{ms} \ll 10\,\text{min.} < 34\,\text{min.}$

For comparison, the fastest modern-day supercomputers are up to 6 orders of magnitude faster than a consumer-grade computer [Wik23b], which is way slower than up to about a trillion-fold assumed adversary advantage. For the configuration it would translate into around 71 years of computation, hence 3 ms proof of work difficulty is currently reasonable.

The setup also makes Pr[busy] ≤ 0.0625, meaning that an honest user will do fewer iterations on average to sample a dummy nullifier that lands on an empty bucket.

If desired, the construction can be combined with a randomized dummy nullifier from 4.2.1 to ensure that a user only needs to generate proof of work once and then derive dummy and regular nullifiers from its own randomized commitments, therefore increasing proof of work generation time is acceptable, e.g., if takes the honest user 5 minutes then the adversary has to be 2^{52} (4.5×10^{15}) times more powerful to reach the negligible probability bound within 10 minute epoch time.

As the computation power increases the dynamic PoW difficulty ensures that adversary will be kept at bay.

Even if a-collision, due to increased Adversarial computational power, ever occurs, it'll not affect regular nullifiers and will only result in temporary degradation in the success rate of transactions with dummy nullifiers and will be alleviated as soon as the proper PoW difficulty is set.

There are several problems with such a scheme: 1) when the user specifies that the target bucket is non-empty, $\tau = 1$, it unambiguously means that a regular nullifier is used, i.e., not dummy; 2) when instead nullifier falls on an empty bucket, $\tau = 0$, it slightly increases the probability of a nullifier to be dummy; 3) if there are very few collisions, of arity 2 or higher, when there should've been many, it points to the fact that the majority of nullifiers are dummy. Hence we need to ensure the indistinguishability of types of nullifiers.

4.2.3.4 Indistinguishability

Adversary may choose to carry out an attack on indistinguishability, concretely, one may set out to make as many buckets as possible non-empty, thereby forcing the user to set $\tau = 1$ for randomized nullifiers, using statistical methods that would allow to estimate the number of randomized nullifiers, and from there the number of dummy nullifiers.

Therefore, the first step is to reduce the number of buckets adversary can affect by increasing the bucket's capacity by c and modify the user's behavior to let dummy nullifiers fall onto non-empty buckets as long as it holds no more than c elements. This effectively reduces the number of affected buckets by the factor of c, requiring adversary to find a multitude of c-collisions.

So far we've ensured that a-collision is infeasible within the epoch's timeframe. Furthermore, if adversary chooses to commit arbitrary multi-collisions found within epoch, prioritizing highest arity, up to the available bandwidth, it'll be hard to achieve a $(c + a - 1)$-collision across epochs since the effort will be spread out, however, if a smaller predefined target set of buckets is used, an adversary would only need to commit nullifiers that fall within that set, ensuring that accumulator's throughput is used effectively. For instance, limiting target buckets set to 2^{20} would allow assumed adversary to find a-collision in just about 8 epochs, the reduced target set ensures that suitable nullifiers found will fit into the throughput, moreover, it'll also be feasible to find c-collisions in the same fashion.

While the above doesn't prevent honest users from transacting, adversary can fill many buckets with c-collisions, increasing coverage to obtain higher confidence in the statistics.

Thus, second step is to combine current scheme with the randomized dummy nullifier from 4.2.1 to ensure that a user only needs to generate proof of work once and then derive dummy and regular nullifiers from its own randomized commitments, allowing such nullifiers to be accepted with $\tau = 1$, thereby removing direct association of $\tau = 1$ with regular, non-dummy, nullifier. Following is the list of the possible types of nullifiers along with the conditions in which they are accepted.

Nullifier Type	$\tau = 0$	$\tau = 1$
Dummy nullifier	✓	×
Randomized nullifier	✓	✓
Randomized dummy nullifier	✓	✓

While we've made it harder to distinguish types of nullifiers, because $\tau = 1$ only permits nullifiers that were derived from randomized commitments, it can provide a source of estimation of how many of the nullifiers are of such type and, consequently, the number of dummy nullifiers can be derived. Concretely, if adversary managed to fill out $|B_c|$ buckets with c-collisions and there are Θ randomized nullifiers of honest users then

the expected number of nullifiers with $\tau = 1$ is $\Theta * \frac{|B_c|}{n}$, hence, having the number of colliding randomized nullifiers it's easy to calculate how many are there in total, including non-colliding, and subtract that from all honest user nullifiers to get the approximate number of dummy nullifiers.

Although the expected number of collisions is useful, the actual result may vary and it's important to know the likelihood of different outcomes to see how differences in expectation and outcome might affect an analysis by an observer, e.g., adversary. Let's say that the random variable X represents all possible outcomes of the randomized nullifier collision with the filled buckets $|B_c|$, then the specific outcome, i.e., that the number of collisions is x, the probability can be computed as

$$\Pr[X = x] = \binom{\Theta}{x}\left(\frac{|B_c|}{n}\right)^x\left(1 - \frac{|B_c|}{n}\right)^{\Theta - x}$$

which follows the normal distribution, i.e., the majority of outcome probabilities are aggregated in the standard deviation proximity of the mean, e.g., if the mean is large it's extremely unlikely that in reality there will be just a few collisions, and if that's the case then absence/insufficiency of collisions points to the fact that the most of the nullifiers are dummy.

Hence, we can conclude that if we want to achieve indistinguishability, to an observer it should look as if there are zero dummy nullifiers and all of the nullifiers are randomized, which represents the expected probability distribution. Therefore, as long as the actual probability distribution is hard to distinguish from the expected, i.e., having almost identical mean and standard deviation, and not reaching statistical significance with the application of statistical methods, then it'll be close to impossible to derive any conclusions from the available information. More generally, the more expected collisions there are the higher the statistical confidence that could be achieved when deviations occur, therefore, the goal is either to minimize the variance or make the probability of even a single collision very unlikely.

Assuming computationally favorable setup parameters this means that the share of dummy nullifiers must only be a fraction of total nullifiers, in practical terms meaning that a user should only use dummy nullifiers as a bootstrap mechanism and preferably there should be only a few users compared to the number of transactions, i.e., each user executing many transactions, or the absolute majority of users are bootstrapped by their peers.

It's a viable approach, although narrowing the range of applications, hence we propose a solution to alleviate the limitations.

4.2.3.5 Randomized Nullifier Indistinguishability via Key Index Variants

Instead of providing useful statistical information to an observer, we can make the number of expected collisions next to zero, hence making it extremely unlikely that a user would ever need to use $\tau = 1$. We already have a mechanism for dummy nullifiers to avoid crowded buckets, we can try to re-use a similar mechanism for the randomized nullifiers.

The approach is to use deterministic key index variants for a nullifier (key) if the respective bucket of the first key index variant is already filled with c or more elements, then the second key index variant will be derived and so on until either there's a bucket with available space or the limit is reached and the $\tau = 1$ must be used. In the end, only a single key index variant will be stored with the highest priority being on the first key index variant, then on the second one and so on. Hence, the non-membership proof is simple, starting with the first key index variant:

1. show that the key index variant is absent in the respective bucket
2. show that the bucket has less than c elements, otherwise start at step 1 for the next key index variant, unless all the key index variants are evaluated

For brevity from now on we'll use "variant" to mean "key index variant".

Notably, when the nullifier is not yet exposed, such as when non-membership is relevant, an adversary doesn't know which bucket(s) to attack, therefore, they can't be easily targeted to increase the complexity of non-membership proofs, conversely, when nullifier is exposed there's no use in proving non-membership anymore, hence filling up such buckets is of no detriment.

Membership proof is trivial, the user just needs to prove membership of one of the possible variants, therefore it's best if it can be computed in constant time instead of sequential derivation, such as a hash chain.

Concretely, i-th variant of a nullifier is derived as $\vartheta_i = \mathsf{H}(\mathsf{nf} \| i)$, having $i \in [1, \ell]$ and represented in fixed-length binary form, where ℓ is the number of available variants.

While it might seem that the introduction of variants substantially increases the computational complexity of the non-membership proof, with reasonable setup parameters proving non-membership for more than one variant is an exception rather than the rule, we'll come back to this in the example instantiation. In the context of zero-knowledge proofs, using proving schemes that don't consume proving time for execution paths not taken, such as SuperNova [KS22], Valida, MidenVM or RISC Zero, ensures minimal overhead and an amortized constant time, i.e., single nullifier variant non-membership proof.

If the user submits a transaction with nullifier nf and $\tau = 0$ to the ledger, then ϑ_1 is computed and if the respective bucket has the capacity, i.e., less than c, it's appended, otherwise ϑ_2 is derived and the process continues similarly until a suitable bucket is found or all the variants are exhausted. If instead $\tau = 1$ is used, then the ϑ_1 is inserted in the respective bucket, which will have the capacity with overwhelming probability.

Algorithm 1 Preparing transaction on client side

```
function PrepareAndSend(δ, rcm, z, args)
    nf ← PRF_z(δ‖rcm)
    τ ← 0^1
    if SendAndExecute(nf, τ, args) then          ▷ Sends transaction to the ledger
        return ⊤
    else                                          ▷ Improbable fallback
        if ¬δ then
            τ ← 1^1
            return SendAndExecute(nf, τ, args)
        else
            return ⊥          ▷ Upon an unlikely failure, the client retries with a new sample of z
        end if
    end if
end function
```

Algorithm 2 Adding nullifier to the accumulator on the ledger

```
function InsertNullifier(nf, τ)
    i ← 1
    while i ≤ ℓ do
        ϑ_i ← H(nf‖i)
        if IsMember(ϑ_i) then
            return ⊥                                   ▷ Reject duplicate
        else if ¬IsBucketCfilled(ϑ_i) ∨ τ then
            return InsertVariant(ϑ_i)
        end if
        i ← i + 1                                      ▷ Try next variant
    end while
    return ⊥                                           ▷ Couldn't find a bucket with availability
end function
```

We forgo the separate allocation of allowance a which leaves bucket capacity at $c + s - 1$. To make the non-membership proof efficient, we need to ensure that the information on the bucket fullness is readily available, therefore we add a unique encoding for bucket root node computation which depends on whether bucket occupancy is c or higher.

There are multiple ways to achieve this, for one, let's say we chose to use unique binary personalization using $\mathsf{pers} : \mathbb{N} \longrightarrow \{0,1\}^{\mathsf{pl}}$, of fixed length pl, at each level l of the tree, starting at 1, hence, each non-leaf node is computed as

$$\mathsf{node} = \mathsf{H}(\mathsf{pers}(l)\|\mathsf{left\_child}\|\mathsf{right\_child})$$

To compute the bucket root node we use $l = d^{\mathrm{B}}$ when the bucket has less than c elements, and $l = d^{\mathrm{B}} + d^{\mathrm{A}} + 1$ otherwise.

We need to identify how many variants should be available for each nullifier, i.e., ℓ, so that it's infeasible for an adversary to prevent an honest user from submitting the nullifier in an indistinguishable fashion, i.e., with $\tau = 0$. While we don't allocate separate space for allowance a, it's still helpful to account for it. As was said previously an adversary can't produce a-collision within an epoch's time, and for the analysis we relax our assumptions and presume that an adversary can generate $(a-1)$-collision for any chosen buckets within an epoch and that's enough to succeed. Having $a - 1 < c$, an adversary would have the best shot at succeeding if as many buckets as possible are filled with $c - a + 1$ elements, which we assume in worst case scenario is carried out using whole accumulator's capacity as expressed as the total number of transactions, 2 nullifiers each, $|T| * 2$, yielding $ß = \left\lfloor \frac{|T|*2}{c-a+1} \right\rfloor$ "staged" buckets. Buckets that contain less than $c-a+1$ elements are said to be infeasible to fill to c elements within an epoch. Thus, for an adversary to succeed, all of the variants of any nullifier shall fall on the staged buckets. Let A_i be an event of i-th variant falling on a staged bucket, then $\Pr[A_i] = \frac{ß}{n}$, since events are independent we have $\Pr\left[\bigcap_{i=1}^{\ell} A_i\right] = \left(\frac{ß}{n}\right)^{\ell}$.

Therefore, ℓ shall be as large as to make it very unlikely for all variants to fall on staged buckets, i.e., $\ell = \min\left\{x \in \mathbb{N} : \left(\frac{ß}{n}\right)^{x} \leq \epsilon\right\}$.

Having ℓ, we can also identify an expected number of variant non-membership proofs on average to prove nullifier non-membership, represented by random variable Y. A user only needs to prove non-membership of one

more variant when the current variant's bucket has c elements or more, considering the worst-case scenario, an adversary would need to fill as many buckets as possible with c elements, i.e., $\left\lfloor \frac{|T|*2}{c} \right\rfloor$.

Let C be an event of variant falling on a bucket with c elements, then $\Pr[C] = \frac{|T|*2}{c \cdot n}$, and expected number of non-membership proofs is $E[Y] = g_\ell(1)$, where:

$$g_\ell(x) = \Pr\left[\bar{C}\right] * x + \Pr[C] * g_\ell(x+1) \tag{5}$$

$$g_\ell(\ell+1) = 0 \tag{6}$$

Which furthermore can be simplified:

$$E[Y] = \sum_{i=0}^{\ell-1} \Pr[C]^i - \ell \cdot \Pr[C]^\ell \tag{7}$$

We can see that for small $\Pr[C]$, $E[Y]$ approaches 1, $\lim_{\Pr[C]\to 0} E[Y] = 1$, i.e., on average only a single variant non-membership proof is necessary. Moreover, a similar result applies to the complexity of adding a new nullifier to the tree.

Lastly, we need to assess whether increased freedom, namely the ability to fallback to the next variant is reducing security in any way, effectively ensuring that PoW is not wasted if the first variant falls on c-filled bucket, i.e., helping an adversary.

We can assess that such an ability doesn't allow adversary to fill more buckets than before, since it's the accumulator's throughput that's a limiting factor. In the absence of adversary with extremely high probability, there will be no buckets that contain c or more elements, hence secondary variants will not be used, thus those fallback variants will only be used if adversary chooses to c-fill buckets, in worst case scenario as many as possible, i.e., up to $\left\lfloor \frac{|T|*2}{c} \right\rfloor$.

Effectively such circumstance ensures that the c-filled buckets are excluded from the sampling set, i.e., from $2^{d^{\mathrm{A}}}$, since regardless of how many consecutive variants will fall on c-filled bucket(s), with overwhelming probability a bucket with available space will be found. Hence, the resulting sampling space is reduced from $2^{d^{\mathrm{A}}}$ to $2^{d^{\mathrm{A}}} - \left\lfloor \frac{|T|*2}{c} \right\rfloor$, in worst case scenario, which makes it somewhat easier for an adversary to find a-collision, therefore, such reduction shall be accounted for.

We can update the notion of $(q_s, q_\epsilon, \varepsilon)$-security to include indistinguishability:

1. $\Pr\left[C\left(2^{d^{\mathrm{A}}}, q_s, s\right)\right] \leq \varepsilon$
2. $1 - \left(1 - \Pr\left[C\left(2^{d^{\mathrm{A}}} - \left\lfloor \frac{|T|*2}{c} \right\rfloor, q_\epsilon, a\right)\right]\right)^{|E|} \leq \varepsilon$
3. $\Pr\left[\bigcap_{i=1}^{\ell} A_i\right] \leq \varepsilon$
4. $\frac{o_{\mathrm{POW}}}{R_{\mathrm{u}}} \ll \epsilon_{\mathrm{T}} < t^{\mathcal{A}}_{q_\epsilon}$, where $t^{\mathcal{A}}_{q_\epsilon}$ is time it takes adversary to obtain q_ϵ samples

Point 3 ensures that there's always a bucket available that has sufficient capacity, which also means that the same mechanism can be used for dummy nullifiers and there will never be a need to re-sample the dummy nullifier if the bucket is busy.

Note, that we no longer need to ensure that there are empty buckets available, moreover we leave τ and s in place as a safety mechanism for cases when a) adversary learns the user's nullifiers before they were exposed in a transaction; b) PoW adjustment wasn't adequate for a period of time.

We can now update our example instantiation (4) to reflect the changes, while maintaining $\left(2^{46}, 2^{53}, 2^{-128}\right)$-security:

$$d^{\mathrm{A}} = 49,\ d^{\mathrm{B}} = 8,\ c = 229,\ a = 129,\ s = 28,\ \ell = 14,\ q^{\mathcal{A}}_{\epsilon_{\mathrm{T}}} = 2^{53},\ \frac{o_{\mathrm{POW}}}{R_{\mathrm{u}}} = 250\,\mathrm{ms} \tag{8}$$

These parameters yield $E[Y] = 1.00049$ (one extra variant non-membership proof per 2040 proofs), i.e., the absolute majority of non-membership proofs require only one variant non-membership proof, assuming the worst-case scenario which is only reached at the end of the lifecycle of the accumulator in about 1000 years. Notably, we were able to achieve indistinguishability, through the elimination of collision overflow, while maintaining the overall tree depth unchanged.

5 Efficient General-Purpose Constructions

Now that we considered multiple application-specific constructions and respective primitives that make them possible, we can apply the learnings to the general-purpose constructions.

5.1 Authority Accumulator

One notable learning is that if we assume that the samples are uniformly distributed and there's no malicious activity to factor the key then the combined size of the accumulator is only slightly larger than its capacity. Thus, for the case when the accumulator is managed authoritatively, e.g., a government keeping list of sanctioned organizations, certificate authority issuing certificates, etc., the accumulator can be kept small and very efficient without the complexities of the considered above applications.

In fact, for the accumulator capacity q and bucket size $s - 1$, we can find the addressable tree depth d^{A}, such that $\Pr\left[C(2^d, s, q)\right] \leq 2^{-128}$, i.e., probability of filling up a bucket after q elements are inserted is negligible:

$$d^{\mathrm{A}} = \left\lceil \frac{\log_2 2^{128} \binom{q}{s}}{s - 1} \right\rceil$$

For example, fixing bucket tree depth $d^{\mathrm{B}} = 7$, for capacity $n = 2^{24}$ the addressable tree depth is $d^{\mathrm{A}} = 20$ totaling to 27, and for capacity $n = 2^{32}$ the depth is $d^{\mathrm{A}} = 28$. On average combined accumulator depth is just about 3 levels deeper than the smallest Merkle tree that has enough capacity to hold n elements, i.e., $\log_2(n) + 3$, for $2^9 \leq n \leq 2^{83}$, which is next to optimal, for comparison an AVL tree has a maximum depth of $1.4405 \log_2(n + 2) - 0.3277$, see [Knu98], i.e., about 31% more than our construction for $n = 2^{32}$, while also requiring more storage.

Note that in order to achieve the same security with the trivial solution from the section 2.1 it would require to use the tree of depth 175 and 191 for 2^{24} and 2^{32} capacities respectively, hence this work provides 6.48x and 5.45x reductions respectively.

5.2 VRF-Based

Verifiable random function $\mathsf{VRF}(*)$, introduced in [MRV], allows to open up authority accumulator from section 5.1 to other users, whereby authority would be responsible for computing the deterministic index of the key from given key-value pair (k, v), similarly to [Mel+15], i.e., $\mathsf{VRF}(k)$. This way, the marginal tree depth overhead also applies.

The VRF itself can either be centralized or distributed, however, a distributed setting would require rebuilding an accumulator from scratch if VRF setup reconfiguration occurs, since key indices will change.

Assuming the adversary can't evaluate the key $\mathsf{VRF}(k)$ by any other means apart from submitting a key-value pair insertion operation which will be added to the accumulator, the number of samples is limited by the capacity of the accumulator.

5.3 VDF-Based

Verifiable delay function [Bon+18] can provide both determinism of result for any given input, as well as requirement on the number of sequential steps of computation before the result is provided. We can use these properties to build general-purpose accumulator in cases where VRF is not appropriate.

The idea is to multiply the amount of work necessary to produce s-collision, thereby allowing for reduction of the tree depth. If it takes w work (samples) on average to find s-collision, then if the addition of VDF takes w_{DS} operations to derive a single key index from key k then the total work becomes $w \times w_{\mathrm{DS}}$. Hence, the number of indices that's possible to derive with w work is $q_{\mathrm{VDF}} = \frac{w}{w_{\mathrm{DS}}}$, respectively the updated goal is to set accumulator parameters such that it takes q_{VDF} samples on average to find s-collision, effectively meaning that we can reduce depth while keeping security at the same level.

We can achieve this either using random delay oracle (see [Bon+18]) or the VDF directly, relying on its determinism and unpredictability properties. Using notation from [Bon+18], to derive key index h, first the VDF evaluation is carried out $(\nu, \pi_{\mathrm{VDF}}) = \mathsf{VDF.Eval}(\mathsf{ek}, k)$, followed by $h = \mathsf{PRF}_0(\nu)$. The accumulator insertion request, therefore, consists of $(k, v, \nu, \pi_{\mathrm{VDF}})$, which is verified before being accepted using $\mathsf{VDF.Verify}()$, which can be carried out either natively or through zero-knowledge proof. Notably, the verification takes only a short time compared to evaluation, hence the modification has minimal impact on accumulator performance.

For example for 128 bits of security, if it takes $w_{\mathrm{DS}} = 2^{28}$ operations to evaluate the VDF on a key, we need to set accumulator parameters such that s-collision is feasible after 2^{100} key indices are sampled. Having addressable tree depth $d^{\mathrm{A}} = 94$ and bucket size $s - 1 = 256$ satisfies the requirements, resulting in a combined depth of 102, or about 1.28 times shallower than in section 2.4. We can satisfy 96 bits of security with $d^{\mathrm{A}} = 62$ and $s - 1 = 256$, i.e., 70 levels combined, a 1.41 times reduction.

This tree size improvement, of course, comes at an additional cost to the user and, therefore, may not be suitable for some applications.

6 Bucket Operational Efficiency

We've omitted bucket insertion or removal time complexity so far, but it's important to address it for practical efficiency. Although s-collision may be infeasible after w samples, there might be many less than s-collisions, we can generalize the number of days with the birthday collision formula [Wik23a, Number of days

with at least two birthdays]:

$$n - \sum_{i=0}^{s-1} n\left(\frac{1}{n}\right)^{i}\binom{q}{i}\left(\frac{n-1}{n}\right)^{q-i} \tag{9}$$

Notably the formula $\binom{q}{s} \times n^{-(s-1)\mu_s(h)}$ from [RS09] section 2.4, is asymptotic to the above, however, it's especially suited for cases where $\mu_s(h) \neq 1$, although, as before (section 2.4) we assume that $\mu_s(h) = 1$ for simplicity, although final construction must account for less than ideal hash function for margin of safety.

For example, for $d^{\mathrm{A}} = 59$, $n = 2^{59}$ and $s = 65$, the probability of 65-collision is negligible after 2^{60} work.

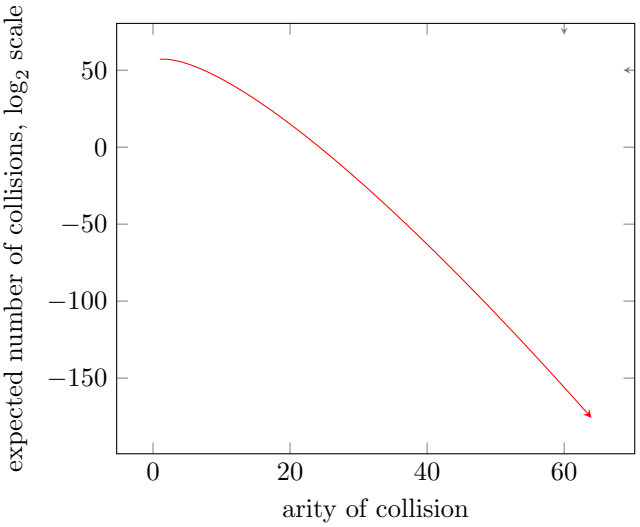

Figure 6: Expected number of buckets with the specific number of elements

For instance, we can expect about 2^{57} buckets with a single element, 2^{44} 10-element buckets, and about one bucket with 24 elements, thus multi-element buckets are quite common and we need to consider how to best deal with them.

It's helpful, to look at the cumulative share of busy buckets up to a certain number of elements per bucket in relation to all the non-empty buckets, figure 7. For instance, more than 99% of all the busy buckets are expected to have 6-collision or less, and more than 83% of 3-collision or less. To complete the picture, let's calculate the expected bucket occupancy as relation of the number of elements versus the number of busy buckets:

$$\frac{q}{n - n\left(\frac{n-1}{n}\right)^{q}} \tag{10}$$

For our example this yields about 2.3 elements on average per bucket, hence it's reasonable to store bucket trees as sparse Merkle trees [Bau04]; [DPP16] and assume constant time performance.

However, the more we shift the balance towards the larger bucket size the larger the expected occupancy will be, for instance, $q = 2^{32}, d^{\mathrm{A}} = 28, d^{\mathrm{B}} = 7$, yields about 16-element buckets on average. Generally, security level being 128 bits, as we choose larger buckets, $d^{\mathrm{B}} > 1$, the average occupancy will get up to 25%. Considering such cases or the fact that malicious actor might try to commit partial multi-collisions as a form of DoS attack, especially in constructions such as in section 4.2.3, our solution will depend on whether history-independence property is desired.

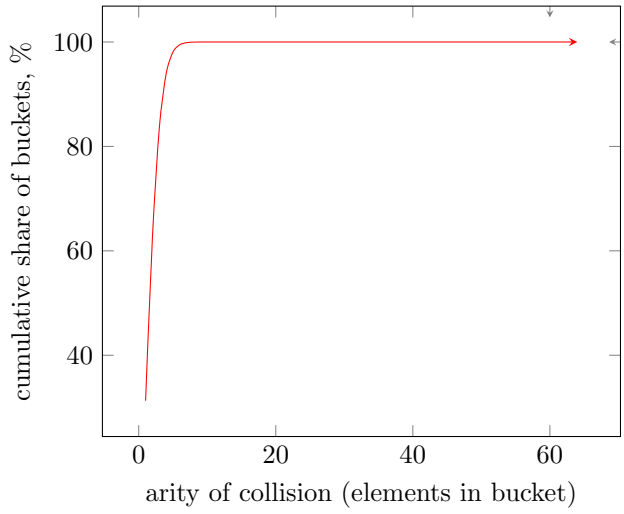

Figure 7: Cumulative share of buckets, containing up to the given number of elements

6.1 History-Independent

It's important to note that if the bucket is history-independent then the accumulator as a whole is history-independent since keys are mapped to specific buckets deterministically and regardless of the order of processing the buckets will contain exactly the same elements.

Keeping buckets sorted provides history independence, however, this might prompt an $O(s)$ reorganization in the worst case which would require reconstruction of the bucket tree from scratch, involving recomputing hashes of all parent nodes. Hence it's best to keep the bucket size small, however, as we've seen in figure 5, a bucket tree with a depth less than 6 will need to pay unproportionate extra in size of the addressable tree, hence it's a balancing act.

For our example, the bucket size of 2^5 with the addressable tree depth of $d^{\mathrm{A}} = 62$ provides the necessary level of security, moreover, more than 99% of non-empty buckets are expected to contain 2 or fewer elements, and in 99.9999% of non-empty buckets the worst case is an insert or removal operation in bucket with 4 elements, which is negligible and may be considered amortized constant-time.

6.2 History-Dependent

As mentioned in section 2.2, we can do further optimization of the sorted tree, building on top of optimization from section 2.4 where leaf also contains the key of the leaf that is next in the ordering sequence, i.e., leaf l_i contains $\left(k_i^{\mathrm{B}}, v, k_{i+1}^{\mathrm{B}}\right)$ where in non-membership-proof of $k_{\notin}^{\mathrm{B}}$ we make sure that $k_i^{\mathrm{B}} < k_{\notin}^{\mathrm{B}} < k_{i+1}^{\mathrm{B}}$.

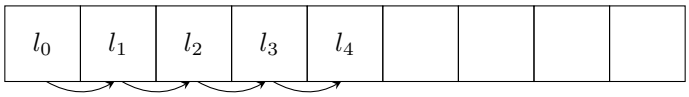

Figure 8: leaves are in the ascending order of the key

Considering that all the data needed for membership and non-membership proofs is already available in the leaf, we can conclude that the ordering of the leaves doesn't matter as long as each leaf contains the key of

the next element in the ordered sequence.

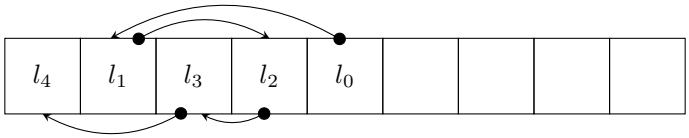

Figure 9: leaves in order of insertion "pointing" to the next leaf, in the ordered sequence

Hence, upon insert operation, we can append an element onto the first available leaf and make sure to specify the correct next value (if there's any) and update the next value of the previous by-order leaf (if there's any). Notably, the same observation was independently made in [LK12].

Hence, now every time we insert only at most $O(2d^{\mathrm{B}})$ updates need to be made. However, keeping it as is would require $O(s-1)$ search complexity, to find the two adjacent elements in the linked list-like structure, which might be reasonable for smaller buckets.

To make search complexity improvement we can keep a supplemental sorted data structure which would contain the list of leaf indices sorted by the bucket key k^{B} of the respective leaves. For example, we can use a sorted array for $O(d^{\mathrm{B}})$ search complexity and $O(s-1)$ insertion or removal complexity, although it's worth noting that changes to this data structure don't need invocation of hashing, hence it's more efficient when sorting data is kept independent of the bucket tree. Moreover, for buckets of size s only $s \log_2 s$ bits of storage are required, which is only a fraction of bucket storage, i.e., a bucket size of 32 only needs 20 bytes to store the sorted structure.

Other sorted data structures, such as AVL[13] tree can be used for $O(d^{\mathrm{B}})$ insert and delete operations. However, since the number of multi-collisions is dropping exponentially, as per figure 6, most non-empty buckets will contain only a few elements, therefore it might be reasonable to use simple data structure and only when it's warranted, i.e., when the bucket reaches a certain size.

With a supporting data structure in place, it's notable that there's no need to store the key of the next leaf in the current one, the key can be fetched from the next leaf when needed, such as for hashing up to the root or constructing a (non-)membership proof.

Even though we update 2 leaves when we insert an element, we only do so in the bucket ($\mathcal{O}(2d^{\mathrm{B}}-1)$) and the update of the bucket root hash in the whole tree is only computed for d^{A} nodes.

7 Future Work

Apart from applications to cryptographic accumulators, learnings from the multi-collision probability and bucket occupancy analysis may have a broad range of applications, including in databases and hash table implementations. For instance, for hash tables it may provide a useful estimate on how many elements to expect at any given location on average, while separately taking care of edge-case overflows.

While the focus of the work was on symbiosis with zero-knowledge proofs, in some scenarios it's not required, hence our construction can be easily adapted for such needs.

[13] AVL62.

Introduction of key index variants in section 4.2.3.5 might be a stepping stone to either reducing or eliminating the need for post-randomization.

Additionally, the approach can be modified to utilize higher tree arity to improve storage efficiency and/or to benefit from better performing hash functions.

8 Conclusions

We've constructed an efficient key-value, and optionally history-independent[14], accumulator in terms of constant-time lookup complexity, rounds of hashing for (non-)membership proofs, number of updates per insert and storage requirement. The primary insight is that the larger the arity of collision is, the substantially more samples is needed, asymptotically $\mathcal{O}\left(\frac{n}{\mathbf{e}}\right)$ increase per each arity increment, i.e., s.

For the subset of applications that include nullifiers, we've utilized unpredictability of the final key index along with rate-limiting to establish an upper bound on the total number of samples which facilitated a substantial tree depth reduction of more than 5 fold.

Additional analysis of the expected occupancy of bucket trees (section 6) led to the conclusion that amortized lookup complexity is $\mathcal{O}(1)$.

9 Acknowledgements

We thank Daniel Lubarov and Kai Geffen for helpful comments that made this work better.

[14] NT01.

10 References

[AVL62] Georgii Maksimovich Adelson-Velskii and Evgenii Mikhailovich Landis. "An algorithm for organization of information". In: *Proc. USSR Acad. Sci.* Vol. 146. 2. USSR Academy of Sciences. 1962, pp. 263–266 (cit. on p. 31).

[Bau04] Matthias Bauer. "Proofs of Zero Knowledge". In: *CoRR* cs.CR/0406058 (2004). URL: http://arxiv.org/abs/cs/0406058 (cit. on pp. 3, 5, 6, 29).

[Bon+18] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. *Verifiable Delay Functions*. Cryptology ePrint Archive, Paper 2018/601. https://eprint.iacr.org/2018/601. 2018. URL: https://eprint.iacr.org/2018/601 (cit. on p. 28).

[BS+14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. *Zerocash: Decentralized Anonymous Payments from Bitcoin*. Cryptology ePrint Archive, Paper 2014/349. https://eprint.iacr.org/2014/349. 2014. URL: https://eprint.iacr.org/2014/349 (cit. on pp. 1, 3, 4, 11, 12, 13).

[DN93] Cynthia Dwork and Moni Naor. "Pricing via Processing or Combatting Junk Mail". In: *Advances in Cryptology—CRYPTO'92: 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August 16–20, 1992. Proceedings*. Vol. 740. 1993, p. 139 (cit. on p. 19).

[DPP16] Rasmus Dahlberg, Tobias Pulls, and Roel Peeters. *Efficient Sparse Merkle Trees: Caching Strategies and Secure (Non-)Membership Proofs*. Cryptology ePrint Archive, Paper 2016/683. https://eprint.iacr.org/2016/683. 2016. URL: https://eprint.iacr.org/2016/683 (cit. on pp. 10, 29).

[Gal+21] David Galindo, Jia Liu, Mihair Ordean, and Jin-Mann Wong. "Fully Distributed Verifiable Random Functions and their Application to Decentralised Random Beacons". In: *2021 IEEE European Symposium on Security and Privacy (EuroSP)*. 2021, pp. 88–102. DOI: 10.1109/EuroSP51992.2021.00017 (cit. on pp. 12, 14).

[GHW21] Zhenhuan Gao, Yuxuan Hu, and Qinfan Wu. *'Jellyfish Merkle tree*. Tech. rep. Facebook Diem, Tech. Rep, 2021 (cit. on pp. 1, 5).

[GT00] Michael T Goodrich and Roberto Tamassia. *Efficient authenticated dictionaries with skip lists and commutative hashing*. Tech. rep. Technical Report, Johns Hopkins Information Security Institute, 2000 (cit. on p. 9).

[HMW18] Timo Hanke, Mahnush Movahedi, and Dominic Williams. "DFINITY Technology Overview Series, Consensus System". In: *ArXiv* abs/1805.04548 (2018). URL: https://api.semanticscholar.org/CorpusID:44106332 (cit. on pp. 12, 14).

[Hop+20] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. *Zcash Protocol Specification*. [Version 2020.1.14]. 2020. URL: https://github.com/zcash/zips/blob/36b35dbf4a7f6be54617fb52906d87816582d4e6/protocol/protocol.pdf (cit. on pp. 1, 11, 12, 13, 14).

[Knu98] D.E. Knuth. *The Art of Computer Programming: Sorting and Searching, Volume 3*. Pearson Education, 1998. ISBN: 9780321635785. URL: https://books.google.com/books?id=cYULBAAAQBAJ (cit. on p. 27).

[KS22] Abhiram Kothapalli and Srinath Setty. *SuperNova: Proving universal machine executions without universal circuits*. Cryptology ePrint Archive, Paper 2022/1758. https://eprint.iacr.org/2022/1758. 2022. URL: https://eprint.iacr.org/2022/1758 (cit. on p. 24).

[LK12] Ben Laurie and Emilia Kasper. "Revocation transparency". In: *Google Research, September* 33 (2012) (cit. on pp. 3, 5, 31).

[Mel+15] Marcela S Melara, Aaron Blankstein, Joseph Bonneau, Edward W Felten, and Michael J Freedman. "CONIKS: Bringing key transparency to end users". In: *24th USENIX Security Symposium (USENIX Security 15)*. 2015, pp. 383–398 (cit. on pp. 1, 3, 28).

[MRV] Silvio Micali, Michael Rabin, and Salil Vadhan. "Verifiable Random Functions". In: () (cit. on p. 28).

[NT01] Moni Naor and Vanessa Teague. "Anti-Persistence: History Independent Data Structures". In: *Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing*. STOC '01. Hersonissos, Greece: Association for Computing Machinery, 2001, 492–501. ISBN: 1581133499. DOI: 10.1145/380752.380844. URL: https://doi.org/10.1145/380752.380844 (cit. on p. 32).

[Ped92] Torben Pryds Pedersen. "Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing". In: *Advances in Cryptology — CRYPTO '91*. Ed. by Joan Feigenbaum. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 129–140. ISBN: 978-3-540-46766-3 (cit. on p. 14).

[RS09] Somindu C. Ramanna and Palash Sarkar. *On Quantifying the Resistance of Concrete Hash Functions to Generic Multi-Collision Attacks*. Cryptology ePrint Archive, Paper 2009/525. https://eprint.iacr.org/2009/525. 2009. URL: https://eprint.iacr.org/2009/525 (cit. on pp. 8, 11, 29).

[Ste99] Steven Finch. *Minimal number of people to give a 50% probability of having at least n coincident birthdays in one year*. [Online; accessed 20-December-2020]. 1999. URL: https://oeis.org/A014088 (cit. on p. 6).

[STS99] Tomas Sander and Amnon Ta-Shma. "Auditable, Anonymous Electronic Cash". In: *Advances in Cryptology — CRYPTO' 99*. Ed. by Michael Wiener. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 555–572. ISBN: 978-3-540-48405-9 (cit. on p. 12).

[Suz+06] Kazuhiro Suzuki, Dongvu Tonien, Kaoru Kurosawa, and Koji Toyota. "Birthday Paradox for Multi-collisions". In: *Information Security and Cryptology – ICISC 2006*. Ed. by Min Surp Rhee and Byoungcheon Lee. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 29–40. ISBN: 978-3-540-49114-9 (cit. on pp. 7, 8).

[Tzi+21] Ioanna Tzialla, Abhiram Kothapalli, Bryan Parno, and Srinath Setty. *Transparency Dictionaries with Succinct Proofs of Correct Operation*. Cryptology ePrint Archive, Paper 2021/1263. https://eprint.iacr.org/2021/1263. 2021. URL: https://eprint.iacr.org/2021/1263 (cit. on p. 5).

[Wik23a] Wikipedia contributors. *Birthday problem — Wikipedia, The Free Encyclopedia*. [Online; accessed 6-November-2023]. 2023. URL: https://en.wikipedia.org/w/index.php?title=Birthday_problem&oldid=1182788018 (cit. on p. 28).

[Wik23b] Wikipedia contributors. *Computer performance by orders of magnitude — Wikipedia, The Free Encyclopedia*. [Online; accessed 26-October-2023]. 2023. URL: https://en.wikipedia.org/w/index.php?title=Computer_performance_by_orders_of_magnitude&oldid=1178716171 (cit. on p. 21).

[Wik23c] Wikipedia contributors. *Moore's law — Wikipedia, The Free Encyclopedia*. [Online; accessed 27-October-2023]. 2023. URL: https://en.wikipedia.org/w/index.php?title=Moore%27s_law&oldid=1181898128 (cit. on p. 19).

[Woo+14] Gavin Wood et al. “Ethereum: A secure decentralised generalised transaction ledger”. In: *Ethereum project yellow paper* 151.2014 (2014), pp. 1–32 (cit. on pp. 1, 3).