24.10.29 Merkle Tree with Stake

Xun Zhang   Wuyun Siqin   Bingsheng Zhang
Zhejiang University, CHN
22221024@zju.edu.cn   3210101763@zju.edu.cn   bingsheng@zju.edu.cn

October 29 2024

1 Merkle Tree with Stake

We benched Merkle tree with inputs $(\mathbf{mvk}_i, \mathsf{stake_i})$ of every leaf node.

The benchmark code proving the following relations:

- For $i \in \{1 \ldots k\}$: $(mvk_i, \mathsf{stake}_i)$ lies in Merkle tree AVK, N following path $\boldsymbol{p}_i$.

Where **num_origin** is the total number of merkle tree leaves, while **num_aggregation** is the number of the leaf nodes we need to prove the mermbership(through a merkle path).

Here is the benchmark results:

degree	num_aggregation	num_origin	proof_time	proof_size	verify_time
19	16	32	5.2977s	704	5.6127ms
19	32	64	7.0434s	1056	6.4773ms
19	64	128	12.9528s	2112	7.8996ms
19	128	256	21.6548s	3872	8.0950ms
19	256	512	45.9381s	8448	10.3092ms
19	512	1024	93.9012s	17600	14.1707ms
19	1024	2048	203.7358s	37664	22.4600ms

Table 1: Merkle Tree Path Benchmark with stake

2 BLS Multisignature Combined with Merkle tree(stake version)

We also combined this version of merkle tree with BLS multi-signature($\mathsf{MSP.B}$is too costly).

The benchmark code prove the following relations:

- $\mathsf{MSP.AKey}(\mathbf{mvk})$: Takes a vector $\mathbf{mvk}$ of (previously checked) verification keys and returns an intermediate aggregate public key $ivk = \prod mvk_i$.
- $\mathsf{MSP.ASig}(\boldsymbol{\sigma})$: Takes as input a vector $\boldsymbol{\sigma}$ and returns $\mu \leftarrow \prod_1^d \sigma_i$.
- For $i \in \{1 \ldots k\}$: $(mvk_i, \mathsf{stake}_i)$ lies in Merkle tree AVK, N following path $\boldsymbol{p}_i$.

Here is the benchmark results:

degree	**num_aggregation**	**num_origin**	**proof_time**	**proof_size**	**verify_time**
19	16	32	26.3544s	3424	7.5312ms
19	32	64	29.1397s	4000	8.4109ms
19	64	128	32.8391s	4576	8.4429ms
19	128	256	51.2058s	7008	10.2677ms
19	256	512	75.1236s	10688	11.5886ms
19	512	1024	132.3989s	19104	14.7561ms
19	1024	2048	251.9375s	37664	20.0323ms

Table 2: BLS Multi-signature with Merkle Tree(stake)

3 ϕ Function in Mithril

This function mapping the stake $\mathsf{stake_i}$ of an individual user, or set of users to the probability of wining one of the lotteries.

We find the implementation code in Mithril, and this function also compare the ev_i and $\phi(\mathsf{stake_i})$: As you can see, the implementation is very complex, and it use an trick to compare a real and a float. In particular, ev is a natural in $[0,2^{512}]$, while ϕ is a floating point in [0, 1], and so what this check does is verify whether $p < 1 - (1 - phi\_f)^w$, with $p = ev/2^{512}$.

```rust
#[cfg(any(feature = "num-integer-backend", target_family = "wasm", windows))]
/// Checks that ev is successful in the lottery. In particular, it compares the output of `phi`
/// (a real) to the output of `ev` (a hash).  It uses the same technique used in the
/// [Cardano ledger](https://github.com/input-output-hk/cardano-ledger/). In particular,
/// `ev` is a natural in `[0,2^512]`, while `phi` is a floating point in `[0, 1]`, and so what
/// this check does is verify whether `p < 1 - (1 - phi_f)^w`, with `p = ev / 2^512`.
///
/// The calculation is done using the following optimization:
///
/// let `q = 1 / (1 - p)` and `c = ln(1 - phi_f)`
///
/// then          `p < 1 - (1 - phi_f)^w`
/// `<=> 1 / (1 - p) < exp(-w * c)`
/// `<=> q           < exp(-w * c)`
///
/// This can be computed using the taylor expansion. Using error estimation, we can do
/// an early stop, once we know that the result is either above or below.  We iterate 1000
/// times. If no conclusive result has been reached, we return false.
///
/// Note that         1             1               evMax
///             q = ----- = ------------------ = -------------
///                 1 - p    1 - (ev / evMax)    (evMax - ev)
///
/// Used to determine winning lottery tickets.
pub(crate) fn ev_lt_phi(phi_f: f64, ev: [u8; 64], stake: Stake, total_stake: Stake) -> bool {
    // If phi_f = 1, then we automatically break with true
    if (phi_f - 1.0).abs() < f64::EPSILON {
        return true;
    }

    let ev_max = BigInt::from(2u8).pow(512);
    let ev = BigInt::from_bytes_le(Sign::Plus, &ev);
    let q = Ratio::new_raw(ev_max.clone(), ev_max - ev);

    let c =
        Ratio::from_float((1.0 - phi_f).ln()).expect("Only fails if the float is infinite or NaN.");
    let w = Ratio::new_raw(BigInt::from(stake), BigInt::from(total_stake));
    let x = (w * c).neg();
    // Now we compute a taylor function that breaks when the result is known.
    taylor_comparison(1000, q, x)
}
```

4 An Optimization in Mithril Paper

There a optimization for $\phi(stake)$, and the following contents is from Mithril paper:

The main outlier is the evaluation of ϕ. Fortunately, we don't actually need to evaluate ϕ in the proof: we can replace stake in the tree with $\phi(stake)$ and proceed with the comparison directly. This gives us a circuit size of $O(klogq)$, and verifier complexity of $O(\frac{klog^4q}{log(klogq)})$ as verification is dominated by a multi-exponentiation based on the circuit size.

Which means that our benchmark results is also valid for this optimized solution, because we generated the $\mathsf{stake_i}$ randomly.