Contents

1 Introduction

Our main optimizations of Marlin Verifier includes the Following:

1 **Test hardness optimization**

2 **Verify batch optimizations**

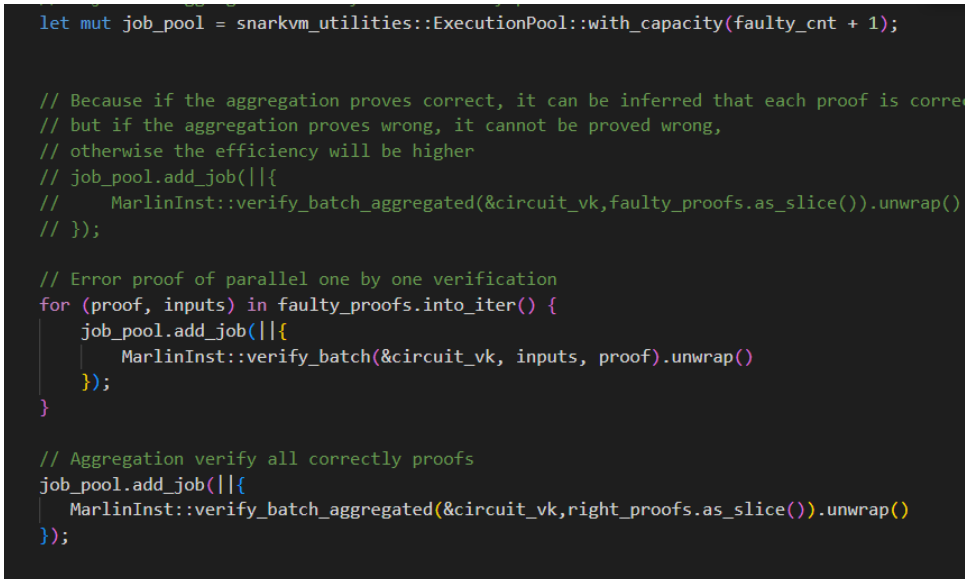
3 **Verify batch aggregation implementation**

2 Test hardness optimization

We implement the test hardness in **./snarkVM/dpc/examples/prize_marlin_verifier.rs**, where we modify the verify_group function based on the following rules.

- We **combine parallel and aggregation** methods within a round.
- We seperate proofs in a round into two groups based on the faulty value: **Faulty group and Right group**. For proofs in Right group, we use **verify_batch_aggregated function**, while for proofs in the Faulty group, we run **verify_batch function,** we run all those in parallel.
- The reason why we use verify batch aggregation for proofs in the Right group but not in the Faulty group is that, **if the verify batch aggregation for the Right group is true, we can infer each proof in the Right group is true. However, if the verify batch**

aggregation for the Faulty group is false, we cannot infer each proof in the Faulty group is false. So we use verify batch for proofs in the Faulty group , even if running verify batch aggregation will be faster.



```
let mut job_pool = snarkvm_utilities::ExecutionPool::with_capacity(faulty_cnt + 1);


// Because if the aggregation proves correct, it can be inferred that each proof is corre
// but if the aggregation proves wrong, it cannot be proved wrong,
// otherwise the efficiency will be higher
// job_pool.add_job(||{
//     MarlinInst::verify_batch_aggregated(&circuit_vk,faulty_proofs.as_slice()).unwrap()
// });

// Error proof of parallel one by one verification
for (proof, inputs) in faulty_proofs.into_iter() {
    job_pool.add_job(||{
        MarlinInst::verify_batch(&circuit_vk, inputs, proof).unwrap()
    });
}

// Aggregation verify all correctly proofs
job_pool.add_job(||{
   MarlinInst::verify_batch_aggregated(&circuit_vk,right_proofs.as_slice()).unwrap()
});
```

3 Verify Batch optimizations

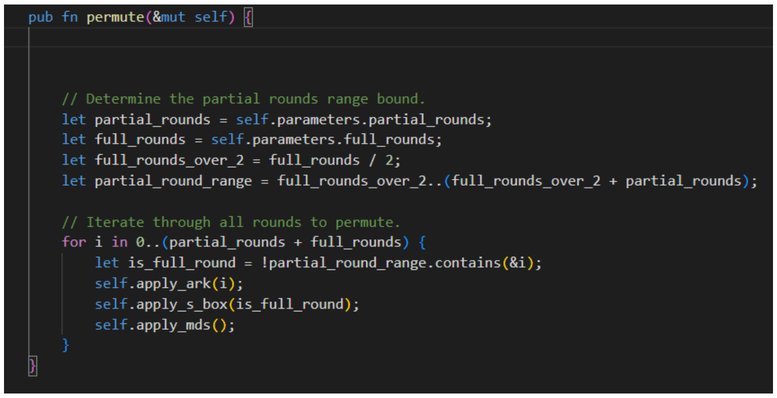
3.1 Sponge optimizations

We optimize the sponge functions based the following:

- **Poseidon hash optimizations**
- **init_sponge function optimization**

3.1.1 Poseidon Hash optimizations

The main process of those sponge functions is the computation of Poseidon hash. The original implementation of Poseidon hash uses the naive permute function, as we have seen **in ./snarkVM/algorithms/src/crypto_hash/poseidon.rs**



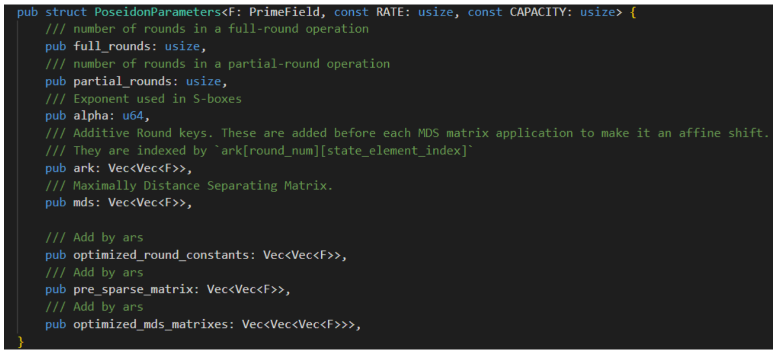
```rust
pub fn permute(&mut self) {


    // Determine the partial rounds range bound.
    let partial_rounds = self.parameters.partial_rounds;
    let full_rounds = self.parameters.full_rounds;
    let full_rounds_over_2 = full_rounds / 2;
    let partial_round_range = full_rounds_over_2..(full_rounds_over_2 + partial_rounds);

    // Iterate through all rounds to permute.
    for i in 0..(partial_rounds + full_rounds) {
        let is_full_round = !partial_round_range.contains(&i);
        self.apply_ark(i);
        self.apply_s_box(is_full_round);
        self.apply_mds();
    }
}
```

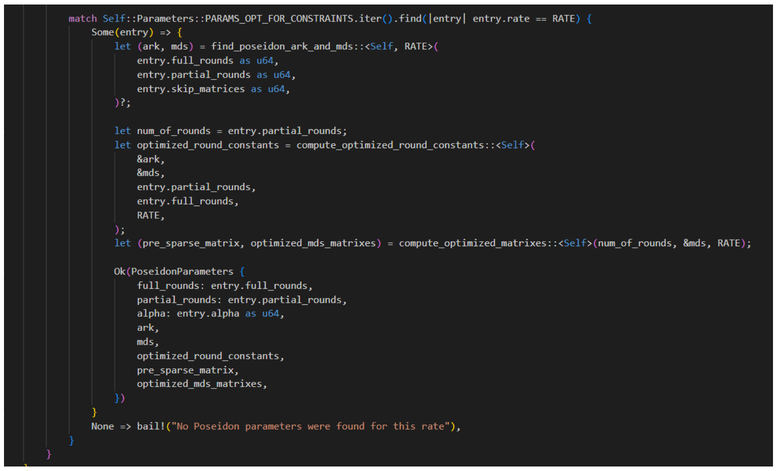
However, **the permute function can be optimized due to s-box function for the partial rounds**, Therefore, we optimize the sponge functions based the following:

- We implement our own **poseidon_round_function, which can greatly decrease the computation complexity of the partial rounds**, see **./snarkVM/algorithms/src/crypto_hash/poseidon.rs**
- **The implemention above need more Poseidon Paremeters, we generate those new Parameters with new functions and related matrix operations:** optimized_round_constants, pre_sparse_matrix,optimized_mds_matrixes, see the implamantations in ./snarkVM/fields/src/traits/poseidon_defau



```rust
pub struct PoseidonParameters<F: PrimeField, const RATE: usize, const CAPACITY: usize> {
    /// number of rounds in a full-round operation
    pub full_rounds: usize,
    /// number of rounds in a partial-round operation
    pub partial_rounds: usize,
    /// Exponent used in S-boxes
    pub alpha: u64,
    /// Additive Round keys. These are added before each MDS matrix application to make it an affine shift.
    /// They are indexed by `ark[round_num][state_element_index]`
    pub ark: Vec<Vec<F>>,
    /// Maximally Distance Separating Matrix.
    pub mds: Vec<Vec<F>>,

    /// Add by ars
    pub optimized_round_constants: Vec<Vec<F>>,
    /// Add by ars
    pub pre_sparse_matrix: Vec<Vec<F>>,
    /// Add by ars
    pub optimized_mds_matrixes: Vec<Vec<Vec<F>>>,
}
```



```
        match Self::Parameters::PARAMS_OPT_FOR_CONSTRAINTS.iter().find(|entry| entry.rate == RATE) {
            Some(entry) => {
                let (ark, mds) = find_poseidon_ark_and_mds::<Self, RATE>(
                    entry.full_rounds as u64,
                    entry.partial_rounds as u64,
                    entry.skip_matrices as u64,
                )?;

                let num_of_rounds = entry.partial_rounds;
                let optimized_round_constants = compute_optimized_round_constants::<Self>(
                    &ark,
                    &mds,
                    entry.partial_rounds,
                    entry.full_rounds,
                    RATE,
                );
                let (pre_sparse_matrix, optimized_mds_matrixes) = compute_optimized_matrixes::<Self>(num_of_rounds, &mds, RATE);

                Ok(PoseidonParameters {
                    full_rounds: entry.full_rounds,
                    partial_rounds: entry.partial_rounds,
                    alpha: entry.alpha as u64,
                    ark,
                    mds,
                    optimized_round_constants,
                    pre_sparse_matrix,
                    optimized_mds_matrixes,
                })
            }
            None => bail!("No Poseidon parameters were found for this rate"),
        }
    }
```

3.1.2 init_sponge function optimization

We optimize the init_sponge function based the following:

- **We cache the Poseidon Paremeters**, so that those parameters only need to be computed once for the first time.
- **We Cache the value of sponge.absorb_native_field_elements** for each batch_size, which depends only on circuit_commitments and batch_size
- We silghtly change the order of computing sponge.absorb_nonnative_field_elements, which seems to be a little better than the original.

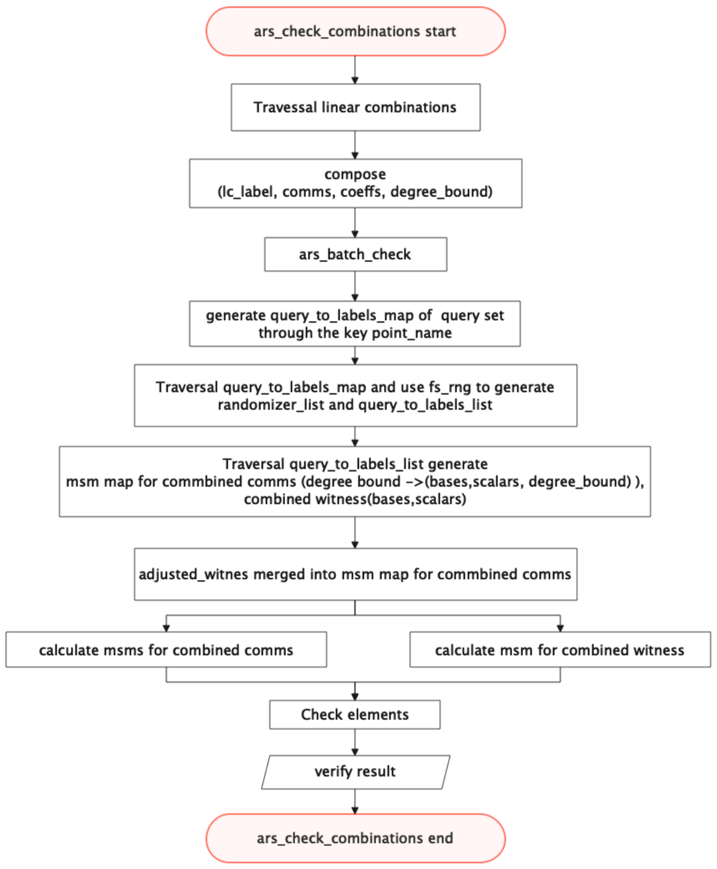
3.2 Check combinations optimizations

We optimize the check_combinations function based the following, see ars_check_combinations in **./snarkVM/algorithms/src/polycommit/sonic_pc/mod.rs**:

- **We eliminate the second random number batch_kzg_check_fs_rng in batch_check function**,which is time consuming but unnecessary.
- Based on the property of bilinear map, **we reorginze the check combinations function:** we eliminate the combine_commitments function for lcs, while keep the bases and scalars for each lc as input of batch check function . **We also reorginze the batch check function:** we eliminate the accumulate function for the query set, while we follow the logic inside the accumulate function to store the bases and scalars for combined_comms(with two kinds of

degree bound), combined_adjusted_witness and combined_witness. **At this point, we only need to compute 4 msms in parallel.**

- We further observe that in the the check_elems function, the None degree bound of combined_comms and the combined_adjusted_witness share the same H in G_2, so that they can be combined, so **we only need to compute 3 msms in parallel**. We modify the check_elems function without the input of combined_adjusted_witness


ars_check_combinations start
Travessal linear combinations
compose
(lc_label, comms, coeffs, degree_bound)
ars_batch_check
generate query_to_labels_map of query set through the key point_name
Traversal query_to_labels_map and use fs_rng to generate randomizer_list and query_to_labels_list
Traversal query_to_labels_list generate msm map for commbined comms (degree bound ->(bases,scalars, degree_bound)), combined witness(bases,scalars)
adjusted_witnes merged into msm map for commbined comms
calculate msms for combined comms
calculate msm for combined witness
Check elements
verify result
ars_check_combinations end

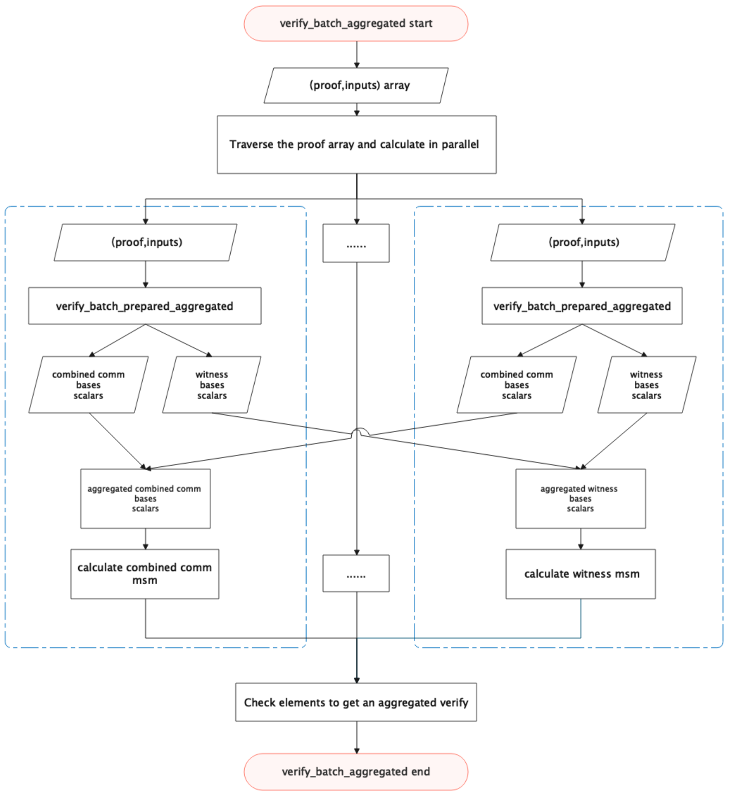
3.3 Modular multiplication optimization

We implement asm code for modular multiplication of Fp384 for the curve BLS377 in **./snarkVM/external/blst-0.3.10**, which speeds the modular multiplication around 10% . The new modular multiplication can both speed **poseidon_round_function** and **msm function.**

4 Verify batch aggregation implementation

Following the ideas above , we implement our own **verify_batch_aggregated function**, see **./snarkVM/algorithms/src/snark/marlin/marlin.rs**.

- For each proof, we run **verify_batch_prepared_aggregated function in parallel** to get the bases and scalars for combined_comms(with two kinds of degree bound) and combined_witness
- We combine the the bases and scalars of all the proofs together to get new bur longer bases and scalrs for combined_comms(with two kinds of degree bound) and combined_witness, then **we only need to compute 3 msms in parallel**, due to the property of bilinear map.
- **The check_elems function only need to be run once.**


verify_batch_aggregated start
(proof,inputs) array
Traverse the proof array and calculate in parallel
(proof,inputs)
......
(proof,inputs)
verify_batch_prepared_aggregated
verify_batch_prepared_aggregated
combined comm bases scalars
witness bases scalars
combined comm bases scalars
witness bases scalars
aggregated combined comm bases scalars
aggregated witness bases scalars
calculate combined comm msm
......
calculate witness msm
Check elements to get an aggregated verify
verify_batch_aggregated end