Documentation for the Prize3 submission solution

Teamname:Zproof

Email:candf1002@gmail.com

Discord:CandF1010#1333

Github: https://github.com/zproof/zprize-ecdsa-varuna

Benchmark:

Test machine: CPU : Intel 12400 Memory : 32G

	100bytes	1000bytes	50000bytes
Keccak256	430.3607s	499.1345s	837.9847s
ECDSA	264.230s	264.230s	264.230s
In total	694.5907s	763.3645s	1102.2147s

※This is the total time to prove **50** times ECDSA signature verification. And only count on proving time and generate witness time, without circuit building time.

※Varuna use Poseidon to do FS, I am not sure how much will be improved if this is sha256, which is 30times faster than Poseidon.

We use stark to prove keccak256, then wrap into plonky2, then wrap into Varuna, and for ECDSA we just prove in Varuna.

We have a fast version for Keccak but don't have time to complete the whole implementation, this is a unsecure version benchmark(but the same time with secure version)

	100bytes	1000bytes	50000bytes
Keccak256	222.2301s	268.1345s	837.9847s
ECDSA	264.230s	264.230s	264.230s
In total	486.4601s	532.3645s	1102.2147s

Overview

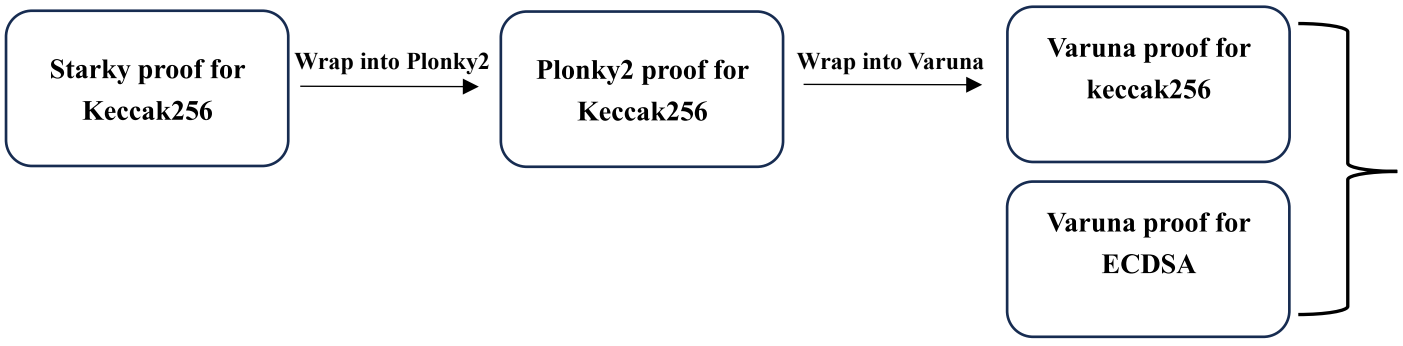
For Prize3: High Throughput Signature Verification, the goal is to utilize Varuna for verifying multiple ECDSA signatures.

Our approach involves splitting the complete ECDSA verification algorithm into two parts. The first part consists of the calculation of the keccak256 hash, while the second part encompasses the algebraic aspect of ECDSA verification (referred to as ECDSA verification hereafter). The reason why we do this is we want to use small field zkp system to prove hash, big field zkp system to prove elliptic curve operations.

- For proving Keccak256 hash function, we employ Starky, a proof system from Polygonzero, and subsequently utilize Plonky2's fast and efficient recursive proof system to wrap the Starky proof into a Plonky2 proof.
- For proving ECDSA verification, we employ the Varuna proof system..

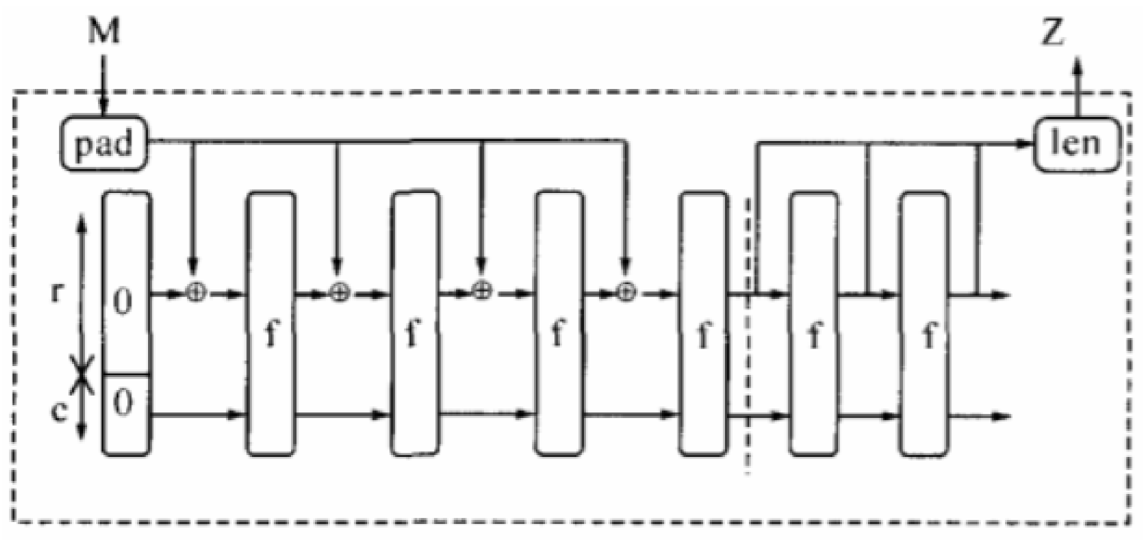
We merge 50 Keccak256 proofs using recursive composition into a single Plonky2 proof. Finally, to wrap the Plonky2 verification into Varuna, we use Gnark to generate R1CS of Plonky2 verify algorithm, and we implement this R1CS in Varuna, Ultimately, Combine with ECDSA, Varuna conducts the final verification.

Besides, we find some issues that need to be changed in Varuna implementation which we attached at the last page of the documentation. We also conducted other experiments, such as implementing ECDSA using xJsnark and Fri-based proof system. We attempted to optimize the R1CS outputs from these xJsnark and prove in Varuna, meantime leveraging Varuna's lookup argument for furthermore optimization. Also, use batch strategy to optimize ECDSA constraint numbers in Plonky2.However, due to Varuna support the batch prove feature, if we count on 50 times ECDSA verification in snark, Varuna performs better. But if we only consider one invocation scenario, Varuna is not efficient because of its prover complexity depends on non-zero elements of R1CS matrices. We will later in this documentation give a detailed analysis. Nevertheless, we provide our test results for transparency.


Starky proof for Keccak256
Wrap into Plonky2
Plonky2 proof for Keccak256
Wrap into Varuna
Varuna proof for keccak256
Varuna proof for ECDSA

Starky proof for Keccak256:

The structure of the Keccak256 hash algorithm:



It can be conceptualized as three main components:

1. Each "f," that is to say the keccakf[1600], takes a 1600-bit input and produces a 1600-bit output. It functions as a permutation mapping for a 1600-bit string and is repetitively invoked in SHA-3.
2. There is the entire sponge construction, responsible for padding the hash input, computing the input and output for each keccakf[1600], and ultimately outputting the final result of keccak256.
3. There is a logical operation structure that handles XOR operations with the previous keccakf[1600] output's first 1088 bits when processing each segment of 1088 bits of input.

For these three components, we can assign the names **keccakf**, **keccaksponge**, and **logic**, respectively.

We arithmetize each of these three components separately and employ the Starky proof system to generate distinct Stark proofs for each part(**keccakf**, **keccaksponge**, and **logic**). By incorporating crosstablelookup and combining these proofs, we can demonstrate the whole keccak256 computation(the structure of **Polygonzero's zkevm**).

※The arithmeticization for each part is generated (**constraint system: AIR**) in AIR. This can be directly read from the rust code. Then we got a table where each column corresponds to the values of certain wires. Rows to rows define a **transition function**. The operations between these rows involve repetitive computations. (I speculate that this is one of the reasons why using Starky to prove keccak is relatively efficient, as many repetitive computations occur in the hashing process.) Finally, the generation of proofs is accomplished using Starky(a stark based on the Goldilocks field).

Assume that for one invocation of keccak256:

keccak256(message)=hash_result

We split this computation into 3 parts:

keccaksponge(message,outputs_keccakfs,outputs_xors)

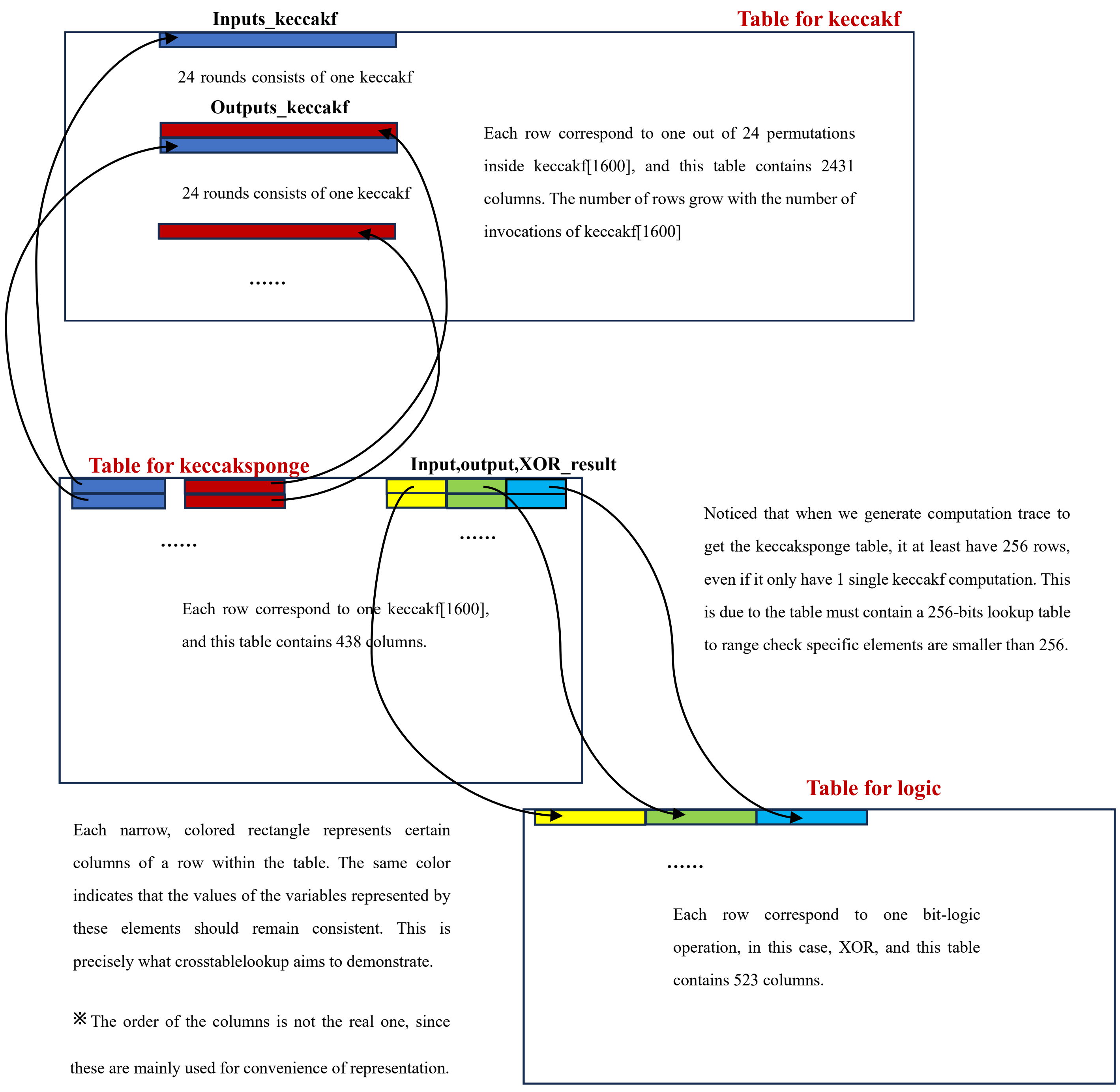
=hash_result

keccakf(inputs_keccakfs)=outputs_keccakfs

Logic(inputs_xors)=outputs_xors

For each computation, we utilize Starky to generate a proof, each proof also leverage a **crosstablelookup argument** to ensure that the mutual wires used in different tables maintain the same. Combining these all together allows us to sufficiently prove a single invocation of keccak256.

Cross table lookup



24 rounds consists of one keccakf

24 rounds consists of one keccakf

Each row correspond to one out of 24 permutations inside keccakf[1600], and this table contains 2431 columns. The number of rows grow with the number of invocations of keccakf[1600]

Each row correspond to one keccakf[1600], and this table contains 438 columns.

Noticed that when we generate computation trace to get the keccaksponge table, it at least have 256 rows, even if it only have 1 single keccakf computation. This is due to the table must contain a 256-bits lookup table to range check specific elements are smaller than 256.

Each narrow, colored rectangle represents certain columns of a row within the table. The same color indicates that the values of the variables represented by these elements should remain consistent. This is precisely what crosstablelookup aims to demonstrate.

※The order of the columns is not the real one, since these are mainly used for convenience of representation.

Each row correspond to one bit-logic operation, in this case, XOR, and this table contains 523 columns.

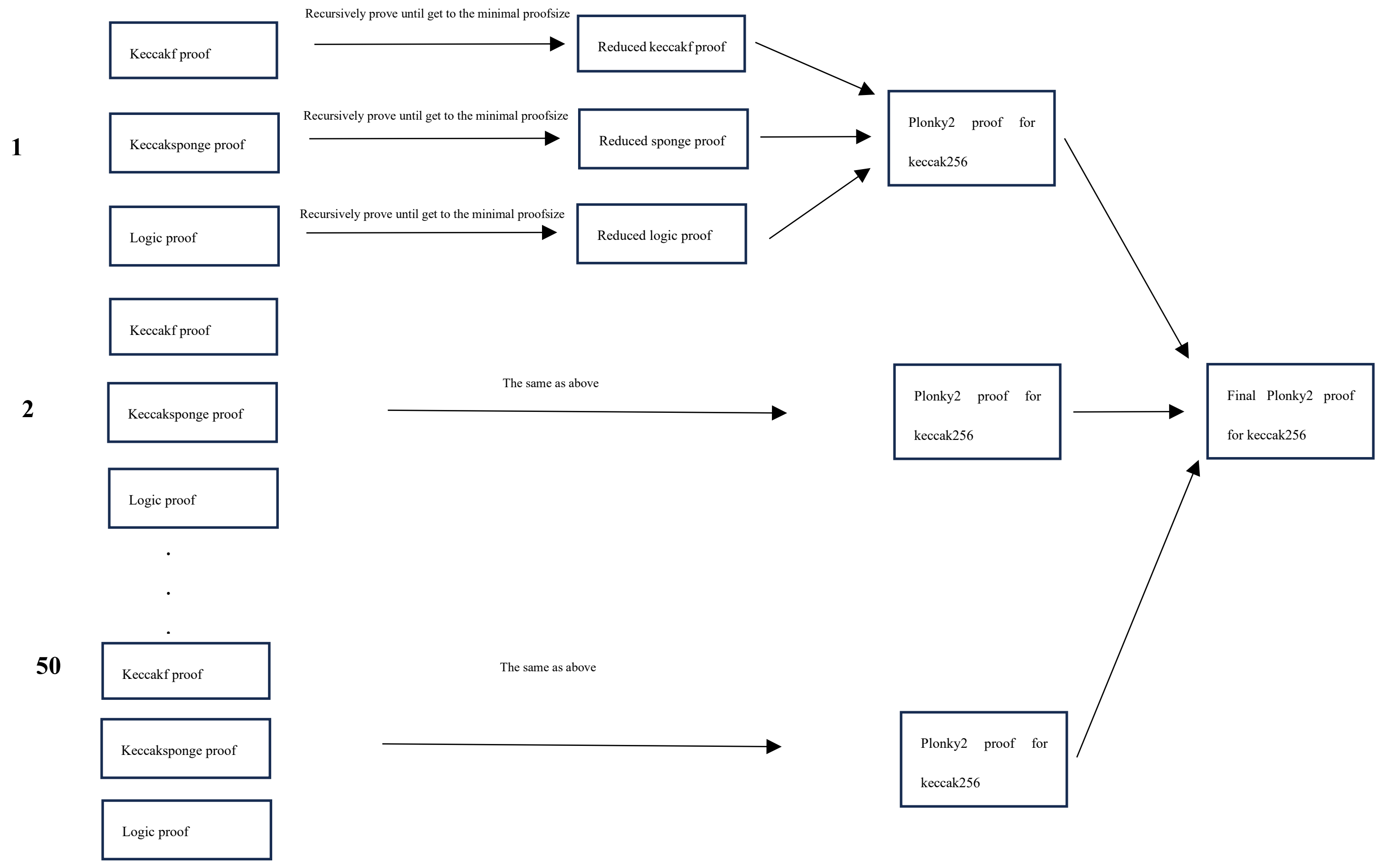
Notice that in the three tables above, the number of columns used to represent the same variables is not consistent. This discrepancy arises from differences in representation across the tables. A column represents an element of varying size, like it can be expressed with 34 columns, each element being 32 bits, and this set of 34 elements represents a 1088-bit input, it can also be expressed with 136 bytes（136*8=1088）. Therefore, when performing specific table lookups, a linear transformation needs to be applied to

Plonky2 proof for Keccak256:

First, we present the solution we use just as the following paragraph, later next section, we give a far more efficient solution that we don't have time to implement.

The solution we use: Once we have obtained these 3 Stark proofs, we then further merge three proofs into one proof(use Plonky2). And this is only one invocation of keccak256, we generate 3 stark proofs, since the benchmark runs with 50 invocations of ECDSA signatures, we need 50 invocations of keccak256, so we initially need to generate 50 multiply 3 equals 150 Starky proofs(50 keccakf proofs, 50 keccaksponge proofs, 50 logic proofs), and then we get 50 tuples of Starky proofs, it looks like: (keccakf proof, keccaksponge proof, logic proof). For each tuple, we use Plonky2 to recursively prove the correctness of the verification procedure of each Starky proof in this tuple. That is to say wrap 3 Starky to 1 Plonky2, and repeat this process 50 times. Finally, wrap plonky2 proof into Varuna, **and we also use gnark to implement plonky2 verifier.**

And the actual execution involves a bit more additional steps, since we can recursively prove the correctness of a proof's verification procedure over and over again to reduce the proof's size, until the proof size meets the recursion threshold(for plonky2 the size is 58860 kb(high rate config)). So we first reduce the 3 proofs in one tuple to the recursion threshold and then merge them into 1. **Additionally, at the last node when we generate the recursive proof,we need to use a config which its Poseidon hash is based on BLS12-377, this is crucially important, without this a huge number of NNA arithmetic will occur in the process of wrapping Plonky2 into Varuna.**


Recursively prove until get to the minimal proofsize
Keccakf proof
Reduced keccakf proof
Recursively prove until get to the minimal proofsize
1
Keccaksponge proof
Reduced sponge proof
Plonky2 proof for keccak256
Recursively prove until get to the minimal proofsize
Logic proof
Reduced logic proof
Keccakf proof
The same as above
2
Keccaksponge proof
Plonky2 proof for keccak256
Final Plonky2 proof for keccak256
Logic proof
.
.
.
50
Keccakf proof
The same as above
Keccaksponge proof
Plonky2 proof for keccak256
Logic proof

※ Another very important thing is that when wrapping the three Stark proofs into one Plonky2 proof, we need to verify the public inputs. Firstly, the previous cross table lookup ensures that the common variables among the three Starks are consistent. However, we also need to check whether the correct public inputs were used in proving Keccak, for example, the hash result: hash_result. After all, our goal is to prove Keccak256(msg) = hash_result.

A faster version for above solution

In the above process, combining multiple Stark proofs into one proof and combining multiple Plonky2 proofs into one proof are both very time-consuming tasks. Therefore, the direct idea is to, for small-scale hash inputs, place the proofs of these Keccak256 hashes in one circuit. This eliminates the step of combining 50 proofs into one(this time we only have 3 starks instead of 150,then one plonky2 proof instead of 50), saving a considerable amount of time. This requires making circuit modifications to the Keccak-sponge Stark. In essence, after implementing these changes, it is no longer a true "Keccak-sponge." Additionally, when combining the three Starks into one Plonky2, some constraints for ensuring integrity are added, and these can be completed within 1 to 2 seconds. The specifics are as follows:

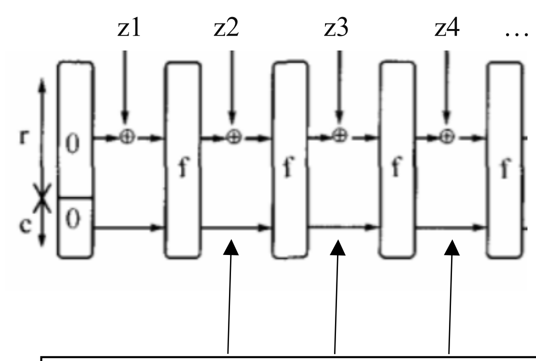
Assume, each message is length of 100 bytes, we have 50 such messages, named: x1,x2,…,x50.

First, we add bits string like : 100…000001 to each message xi such that xi'=xi||100…00001, length(xi)=1088bits. This is actually the same process of padding in sha3.

Second, we calculate :yi = the first 1088bits of keccakf[1600](xi).

Third, z1=x1', z2=y1 ⊕ x2', z3=y2 ⊕x3', … , z50=y49 ⊕ x50', then we have z1,z2,…,z50.

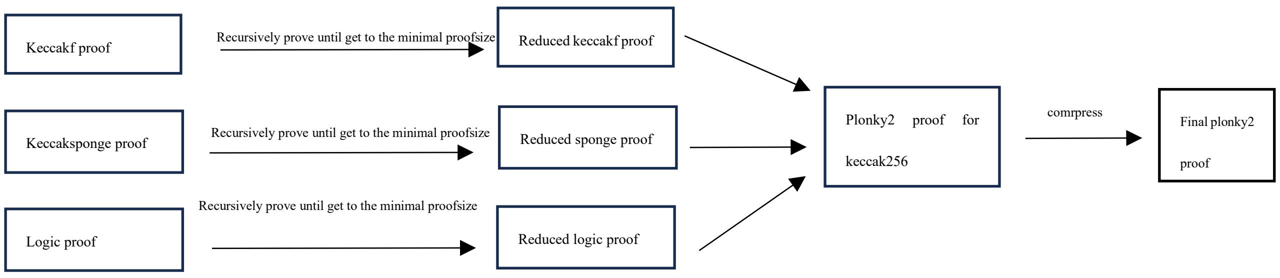
Next, we write a circuit or a set of constraints to illustrate the following function:



Zero bit string which is length of 512 and all bits are 0 , not 512 bits of the last output of Keccak-f

Actually, this function is not a real Keccak256 function , but in general once we prove this function in snark along with the constrains for the generation of {zi} from {xi}. We can optimize the above solution so that we don't have to do with 50 times recursion. And the constraints of generation of {zi} from {xi} are mainly just a number of XOR, precisely it is 1088×50, after test , this can be done within 1.7s in plonky2.

So at this time, we only have 3 starks, 1 plonky2 proof. And we compress this plonky2 proof, then wrap into Varuna.
Just one run of this:


Keccakf proof
Recursively prove until get to the minimal proofsize
Reduced keccakf proof
Keccaksponge proof
Recursively prove until get to the minimal proofsize
Reduced sponge proof
Logic proof
Recursively prove until get to the minimal proofsize
Reduced logic proof
Plonky2 proof for keccak256
comrpress
Final plonky2 proof

We apply this to benchmark level 100 bytes and 1000bytes. For 50000 bytes, we still have 150starks because of the memory is limited, we can not apply this to the level of 50000 bytes.
Unfortunately, some of the circuit constraints are still under active development, and we are working diligently to implement them. However, these issues will not impact the final result and the resulting time.

Varuna proof for ECDSA:

Solution : Gnark-style big integer module method + Varuna lookup argument to do rangecheck.

Specifically speaking, when we use Gnark-style big integer module method, every time we come across a multiplication gate, we do once module, therefore a new set of range check need to be done. After a whole ECDSA verification be transfer to a R1CS, if we use bit-decompositon to do range check, millions of R1CS constraints need to be proven.

So we use Varuna's lookup argument to do range check, we utilize a 16-bit table, contains 0..2^16-1. All the variables that need to be range checked can be done by using the table.

Like, a variable x length of 32bits, we first decompose it to x=x1+x2*2^16. Then check x1,x2 are both in the table.

Notice that, though we can lookup 3 elements in the table one time in the Varuna R1CS lookup setting, but all of our elements are going to be looked up in the same 16bit table(1 column 2^16-1 rows). In Varuna setting, we are looking elements like (0,x,0) in ((0,0,0),(0,1,0),…,(0,2^16-1),0) to ensure x is in the range of 2^16.

In the end, Our r1cs contains 110k normal constraints and 170k lookup constraints.

But the non_zero entries in each matrix are huge, we finally optimize it to the state like :
A and B matrix in r1cs contain 3000k non_zero entries, C matrix contains 500k non_zero entries.

Our starting point is to make more use of the lookup argument in Varuna to optimize the NNA (Non-Negative Arithmetic) problem. In practice, proving ECDSA verification with a SNARK involves proving scalar multiplication, and the operations on the elliptic curve group that lead to an increase in the number of constraints are essentially big integer multiplication. Even though it is possible to represent multiplication in O(m) constraints (as in xJsnark), this quickly leads to a rapid growth in the number of non-zero elements in the R1CS (Rank-1 Constraint System). However, this is already the best approach, as reducing the number further would result in a rank-deficient matrix, compromising the integrity of the constraints.

In scenarios with an equal number of constraints, Groth16 performs significantly better than Varuna, even when including table lookups. Groth16 can circuitize a lookup argument like gnark. But xJsnark produces results with 6 million constraints, as it uses bit decomposition for range checks, even with table lookups to address this issue, its performance is still not comparable to gnark's modular multiplication. xJsnark employs a greedy algorithm to decide when to perform modular reduction, and the former gnark is more suitable for scenarios where table lookups can be used since it performs modular reduction every time when it comes cross a multiplication gate(Because in this way, you don't worry overflow issues).

Plonky2 based on the goldilocks field can prove an ECDSA in just 20 seconds, which is remarkable. Unfortunately, it lacks batch verification capabilities, making it highly inefficient when verifying multiple ECDSAs on the same machine. However, even with a 20-second proof time, it is still slower than Groth16, which aligns with expectations. Simulating a 256-bit field with a 64-bit field results in significantly more constraints, and using a curve-based SNARK to prove ECDSA seems more effective than using goldilocks or Mersenne fields.

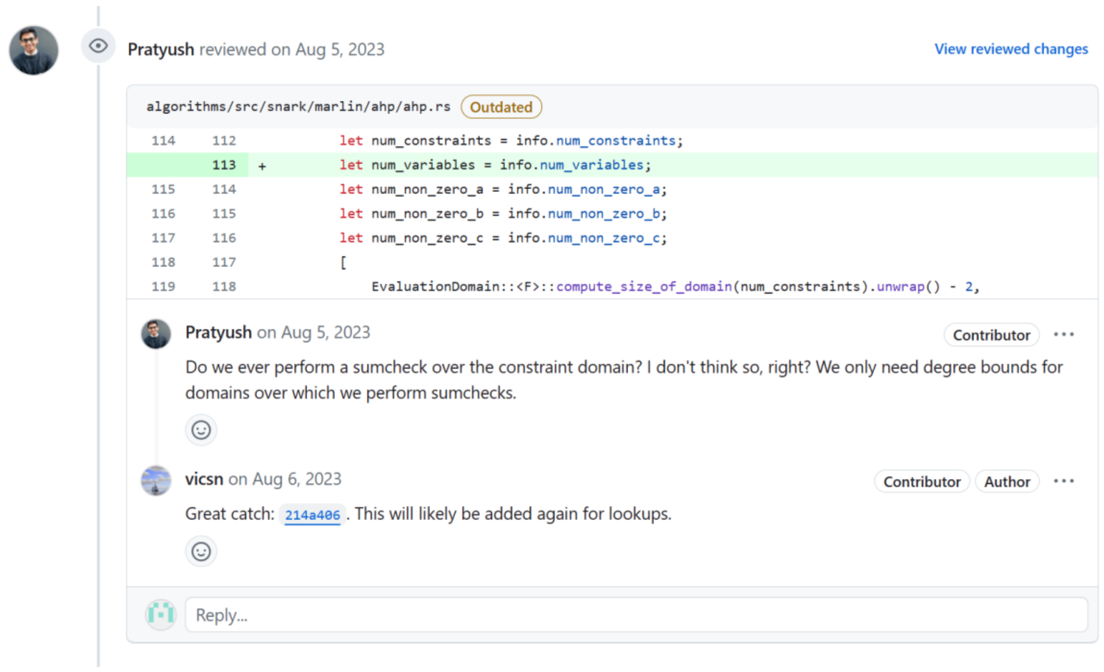
Furthermore, it is evident that using a smaller field for bitwise operations and a larger field for curve operations is a promising approach. This means using a curve-based SNARK to prove ECDSA and employing Mersenne or goldilocks fields, or even smaller binary fields, for hash proof.

In the end, we adopted the approach of using gnark, which performs modular reduction every time multiplication is encountered. After get the R1CS from ganrk, we utilized Varuna's table lookup protocol to address all range checks,. Finally, we employed Varuna's prove batch functionality to generate the proofs.

Something about Varuna.

※ During the implementation of the table lookup, we noticed that the Assignment struct was missing the relevant lookup components. Therefore, we manually added them ourselves in the following path: snarkVM > circuit > environment > src > helpers > assignment.rs.

※ And also, zprize-ecdsa-varuna/snarkVM/algorithms/src/snark/varuna/ahp/ahp.rs {}impl AHPForR1CS<F,MM> > get_degree_bounds. In this, we add num_constraints back since we need Lookup.



Pratyush reviewed on Aug 5, 2023 View reviewed changes

algorithms/src/snark/marlin/ahp/ahp.rs Outdated

```
        let num_constraints = info.num_constraints;
+       let num_variables = info.num_variables;
        let num_non_zero_a = info.num_non_zero_a;
        let num_non_zero_b = info.num_non_zero_b;
        let num_non_zero_c = info.num_non_zero_c;
        [
            EvaluationDomain::<F>::compute_size_of_domain(num_constraints).unwrap() - 2,
```

Pratyush on Aug 5, 2023 Contributor

Do we ever perform a sumcheck over the constraint domain? I don't think so, right? We only need degree bounds for domains over which we perform sumchecks.

vicsn on Aug 6, 2023 Contributor Author

Great catch: 214a406 . This will likely be added again for lookups.

Reply...

https://github.com/AleoHQ/snarkVM/pull/1701

※ We also found some programming errors in aleo-std-profiler that caused the prove process to stall and consume all memory on the machine. We have mitigated this by adding appropriate checks.

※ The root cause is multiple calls to end_timer! on the same TimerInfo object, which causes NUM_INDENT to underflow to a value very close to max(usize). We recommend letting end_timer! consume the TimerInfo, e.g. using drop(timer), to disable multiple calls to end_timer! at compile time. Of course, this would also require some modifications to the snarkVM code.