Implementation Characteristics of Hash Functions in Modern Proof Systems

Markus Schofnegger

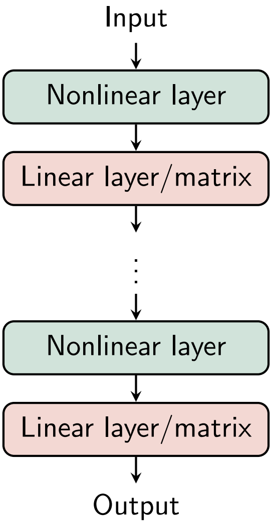
Vienna, February 2026

Outline

- Optimizing the permutation
- Usage in a STARK
- A STARK/HW-friendly permutation

Optimizing the Permutation: Native Performance

A Classical SPN



Analyzing the Components

- Most ZK-friendly permutations are described easily
  - Nonlinear layer in $\mathcal{O}(N)$
  - Linear layer (naively) in $\mathcal{O}(N^2)$
- Initial attemps were mostly focused on arithmetic efficiency
  - Linear layer “not so important” there
  - Only after some years, it got better algorithmically

Linear Layer: From $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log_2 N)$

- Many of the initial ZK-friendly permutations used MDS matrices
  - Maximum diffusion, good against statistical attacks
  - Not really necessary, but allows for easier analysis
- For a “random” MDS matrix, complexity is in general in $\mathcal{O}(N^2)$
- But can be reduced to $\mathcal{O}(N \log_2 N)$ using NTT-based multiplication
  - Assuming the matrix is circulant

MDS Layers from Reed–Solomon Codes[1]

- Reed–Solomon codes are MDS ($d = N - k + 1$)
- Let $H \subset \mathbb{F}^*$ be a multiplicative subgroup of order N, generated by g
- Message vector corresponds to polynomial evaluations over H:

$$\mathbf{x} = \left(f(g^0), f(g^1), \ldots, f(g^{N-1})\right) \in \mathbb{F}^N$$

- Codeword corresponds to evaluations over a coset τH with $\tau \notin H$:

$$\mathbf{y} = \left(f(\tau g^0), f(\tau g^1), \ldots, f(\tau g^{N-1})\right) \in \mathbb{F}^N$$

- We define the linear layer as the mapping $\mathbf{x} \mapsto \mathbf{y}$

[1] Ongoing work with Ulrich Haböck

From Interpolation to Convolution

We evaluate f at points $y = \tau g^j \in \tau H$ (coset of H).

Lagrange Interpolation

For nodes $x_i \in H$, we have:

$$f(y) = \sum_{x_i \in H} f(x_i) \cdot L_H(x_i, y)$$

- Naive evaluation would be $\mathcal{O}(N^2)$
- We need a faster way to compute this sum for all y

From Interpolation to Convolution cont.

Crucial Observation

For $H = \langle g \rangle$, the kernel depends only on the ratio y/x_i:

$$a_{i,j} = L_H(g^i, \tau g^j) = \frac{1}{N} \frac{-g^i(\tau^N - 1)}{g^i - \tau g^j} = \frac{1 - \tau^N}{N} \cdot \frac{g^i}{g^i(1 - \tau g^{j-i})} = \frac{1 - \tau^N}{N} \cdot \frac{1}{1 - \tau g^{j-i}}$$

Convolution Sum

Substituting this back yields a convolution sum:

$$y_j = f(\tau g^j) = \frac{1 - \tau^N}{N} \sum_{i=0}^{N-1} f(g^i) \cdot \frac{1}{1 - \tau g^{j-i}}$$

Convolution via Matrix Multiplication

Circulant Matrix Structure

The convolution $y_j = \sum x_i \cdot c_{(j-i)}$ uses coefficients

$$c_k = \frac{1-\tau^N}{N} \cdot \frac{1}{1-\tau g^k}$$

This is equivalent to $\mathbf{y} = A \cdot \mathbf{x}$ where A is the **circulant matrix**

$$A = \mathrm{circ}(c_0, c_{N-1}, c_{N-2}, \ldots, c_1)$$

Efficiency Gain

Benefit: Matrix-vector multiplication can be done using NTTs

$$\mathbf{y} = \mathrm{iNTT}(\mathrm{NTT}(\mathbf{x}) \circ \mathrm{NTT}(\mathbf{c}))$$

Linear Layer: From $\mathcal{O}(N \log_2 N)$ to $\mathcal{O}(N)$

- MDS matrices are not really needed in many cases
  - Fields are comparatively large, even the smaller ones
  - Degrees are mostly low

$\rightarrow$ A "weaker" matrix for which we know the branch number may be sufficient

- Based on ideas from Griffin-π [GHR+23], matrices got more efficient in Poseidon2$^\pi$
  - Complexity in $\mathcal{O}(N)$ for both external and internal rounds

Still more expensive than the nonlinear layers for Poseidon, *but much better than in the beginning!*

Special Instructions on Custom Hardware

- Custom hardware acceleration for ZK became a hot topic
- As usual, always a tradeoff
  - Area, throughput, reusability, . . .
- Hashing one of the bottlenecks in univariate STARKs

When do custom instructions make sense?

Focus on POSEIDON2$^\pi$: Accelerating the Internal Linear Layer

- Linear layer defined as

$$M_{\text{int}} = \begin{pmatrix} 0 & 1 & \dots & 1 \\ 1 & 0 & \dots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \dots & 0 \end{pmatrix} + \begin{pmatrix} d_0 & 0 & \dots & 0 \\ 0 & d_1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & d_{N-1} \end{pmatrix}$$

- Choose d_i as powers of 2
- Define instruction that facilitates sum building

→ Can be used in other workloads as well

→ Easier than very "specific" instructions such as POSEIDON2 components

Instruction Friendliness: Round Uniformity

- Partial rounds of POSEIDON are not ideal for vectorized implementations
- Main issue is non-uniformity
  - Only one state element goes through the S-box
  - Remainder of the state is unchanged (identity)
- Especially bad for single-permutation implementation
  - Most compute power of the vector register is wasted
  - Vectorized S-box instruction applied to only one element

This is not a big issue in the STARK workload, we will see later why.

Hardware Friendliness: Code Size

- Custom hardware often lacks branch prediction
  - Loops and branches are expensive
- Small code size for the permutation
  - Make unrolled implementation fit into instruction memory
  - Avoids control flow overhead

Loop (bad)

- Cycles overhead per iteration

```
label:
  add x[i], x[i]
  inc i
  cmp i, 16
  jne label
```

Unrolled (good)

- No control flow

```
add x[0], x[0]
add x[1], x[1]
...
add x[15], x[15]
```

Usage in STARKs and Memory Impacts

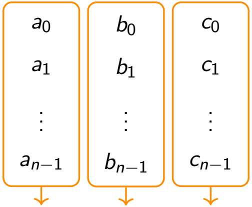
How and where are these hash functions used in STARKs?

- Focus on univariate STARKs
- Hash functions are used for the commitment
- ZK-friendly hash functions allow ZK-friendly opening
  - Important for recursive provers

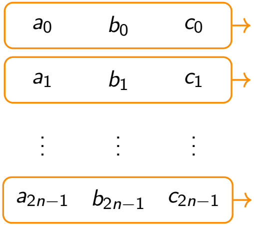
Commitment: LDE and Hashing

- Let us assume some arbitrary trace table
- LDE (low-degree extension) consists of one iNTT and one NTT
  - Applied for each column of the trace table
- Commitment for values after NTT
  - Usually twice or four times the size of the original trace table
- Later, row values have to be opened

Commitment: LDE and Hashing cont.



LDE (columns)



Hashing (rows)

- The LDE extends the columns, then we commit to the resulting values

Commitment: LDE and Hashing cont.

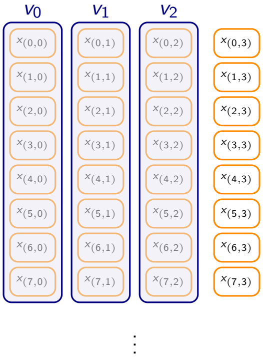
- The LDE extends the columns, then we commit to the resulting values

The Impact of Memory

- LDE implies column-major access, hashing implies row-major
- Large table sizes implies massive memory footprint
  - Over 2^{20} rows, many trace tables, can be around 10 GB of data
- **Different bottlenecks:**
  - LDE (NTT) is typically memory-bound
  - Hashing is typically compute-bound

Optimization: Vectorization/Bitslicing

- **Bitslicing:** Vectorize over calls, not state
- Parallelize multiple hashes (e.g., 16) in one vector register
- Solves non-uniformity of partial rounds
  - 100% utilization of vector units
  - Perform 16 permutations at once



The Persistent Memory Bottleneck

- Access pattern is still a problem
  - To fill vector register i, we need elements from 16 different rows:

$$\{x_{(0,i)}, x_{(1,i)}, \ldots, x_{(15,i)}\}$$

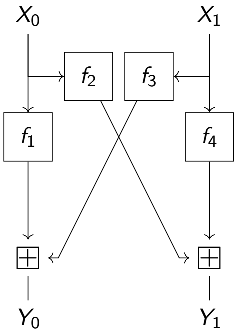
- Practical sizes make this hard
  - Columns are huge (e.g., 2^{20} elements)
  - $x_{(0,0)}$ and $x_{(0,1)}$ are separated by MBs of data
- Requires expensive access
  - Each element likely on a different memory page
  - $\longrightarrow$ Many page misses, cycles spent waiting for data

A STARK-Friendly Permutation from Beneš Networks

♻ Motivation: Reusing Existing Components

- LDE (NTT) and commitment (hashing) are adjacent steps
- NTT uses butterfly operations extensively
- **Idea:** Build a hash function using similar structures
  - Reuse arithmetic units and datapaths
  - "Free" implementation of the hash function if NTT hardware is present

Similarity to the Butterfly



- Beneš network application $\mathcal{B}(x, y)$:

$$u = f_1(x) + f_3(y)$$
$$v = f_2(x) + f_4(y)$$

- Invertible if $f_3 = f_4$ is a PP and $f_1 - f_2$ is a PP
- Resembles a butterfly operation

The Round Function

- Round function $\mathcal{R}$ composed of three layers:
  1. **Add Constant**: $\mathbf{x} \leftarrow \mathbf{x} + \mathbf{c}^{(r)}$
  2. **Nonlinear Layer** $\mathcal{F}$: Apply $\mathcal{B}$ pairwise

$$\mathcal{F}(x_1, \ldots, x_{2l}) = ||_{i=1}^{l} \mathcal{B}(x_i, x_{i+1})$$

  3. **Linear Layer**: Circular shift σ

$$\sigma(x_1, \ldots, x_{2l}) = (x_{2l}, x_1, \ldots, x_{2l-1})$$

- Simple and hardware-friendly structure
- No expensive matrix multiplication

Security Analysis

- Security relies on the complexity of the solving step
  - Dominated by quotient ring dimension D (grows exponentially)
- Solving complexity predictable
  - Complexity as $\mathcal{C}_{\mathtt{univ}} \approx D^{\omega}$ with $\omega \geqslant 2$
- Statistical analysis dominates the round number
  - Truncated paths and repeating patterns with some conditions

Efficiency: Linear Layers and Feistel Networks

- Linear layer is just a rotation
  - Essentially free in hardware (just wiring), cheap on software
  - Avoids more expensive matrix operations
- Same applies to GMiMC and similar designs
  - Maybe worth to renew cryptanalysis and benchmark them
  - Potential for higher efficiency in specific settings

Summary and Takeaways

- Memory wall in ZK workloads
  - Bandwidth/latency often limit performance more than compute
  - Careful memory management is crucial
- Hardware-friendly design
  - Avoid control flow, subcomponents must fit in instruction memory
  - Avoid partial rounds to maximize vector register utilization
- New designs and renewed analysis
  - Beneš and Feistel networks avoid expensive linear layers
  - Simple structures allow for easier hardware optimization

Thanks!

References

[GHR+23] Lorenzo Grassi, Yonglin Hao, Christian Rechberger, Markus Schofnegger, Roman Walch, and Qingju Wang. "Horst Meets Fluid-SPN: Griffin for Zero-Knowledge Applications". In: CRYPTO (3). Vol. 14083. Lecture Notes in Computer Science. Springer, 2023, pp. 573–606.