Polygon zkEVM Security Review

Pil-stark and Snarkjs cryptography review

Auditors

Nirvan Tyagi, Lead Security Researcher

Wilson Nguyen, Lead Security Researcher

Report prepared by: Pablo Misirov

June 20, 2023

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Polygon zkEVM is a new zk-rollup that provides Ethereum Virtual Machine (EVM) equivalence (opcode-level compatibility) for a transparent user experience and existing Ethereum ecosystem and tooling compatibility.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of pil-stark and snarksjs according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 2 days in total, the team conducted an audit of two parts of the Polygon ZKEVM proving pipeline. In the first part, the team reviewed the optimizing modifications made to the fflonk verifier smart contract at the final stage of the pipeline. In the second part, the team reviewed custom PIL constraints for a few components of the STARK verifier including Fp3 multiplication, 1-of-4 Fp3 selector, and Poseidon hash evaluation. These constraints are used during the recursive composition stage of the proving pipeline.

Overall, no soundness issues were discovered in either part of the review. A number of further optimizations to the fflonk verifier and minor documentation / optimizations for the custom PIL constraints are suggested below, however none require immediate action from the Polygon team.

Summary

Project Name	Polygon
Scope	pil-stark
Commit	bb2404...686e
Scope	snarkjs
Commit 2	e70271...8854
Type of Project	Cryptography, zkEVM
Audit Timeline	May 30 to May 31
Two week fix period	May 31 - Jan 14

5 Findings

5.1 FFlonk Verifier Optimizations

5.1.1 Preliminaries

We will be reviewing and optimizing the FFlonk Verifier. Here we define the multiplicative groups, their orders, relevant constants, and the cosets.

- $H = \langle \omega \rangle,\ H = 256$
- $H_3 = \langle \omega_3 \rangle,\ H_3 = 3$
- $H_4 = \langle \omega_4 \rangle,\ H_4 = 4$
- $H_8 = \langle \omega_8 \rangle,\ H_8 = 8$
- $r \leftarrow \mathbb{F},\ \xi \leftarrow r^{24}$
- $h_0 = r^3,\ h_1 = r^6,\ h_2 = r^8,\ h_3 = r^8\omega^{1/3}$
- $h_0^8 = \xi,\ h_1^4 = \xi,\ h_2^3 = \xi,\ h_3^3 = \xi\omega$
- $S_0 = h_0 H_8 = r^3 H_8$
- $S_1 = h_1 H_4 = r^6 H_4$
- $S_2 = h_2 H_3 \cup h_3 H_3 = r^8 H_3 \cup r^8\omega^{1/3} H_3$

5.1.2 FFlonk Verifier Contract

In this section, we briefly and succintly describe the relevant operations of the FFlonk verifier in solidity.

Constants Available

- $\omega,\ \omega^{1/3}$
- $\{\omega_3,\ \omega_3^2\}$
- $\{\omega_4,\ \omega_4^2,\ \omega_4^3\}$
- $\{\omega_8,\ \omega_8^2,\ \ldots,\ \omega_8^7\}$

computeChallenges Compute every element of S_0, S_1, and S_2.

computeLiS0 For lagrange polynomial,

$$L_i^{(S_0)}(X) = \frac{1}{8h_0^7\omega_8^{7i}} \cdot \frac{X^8 - h_0^8}{X - h_0\omega_8^i}$$

Compute the denominators when evaluated on challenge y,

$$\text{LiS0dens} \left\{ 8h_0^7\omega_8^{7i} \cdot (y - h_0\omega_8^i) \right\}_{i \in [8]}$$

computeLiS1 For lagrange polynomial,

$$L_i^{(S_1)}(X) = \frac{1}{4h_1^3\omega_4^{3i}} \cdot \frac{X^4 - h_1^4}{X - h_1\omega_4^i}$$

Compute the denominators when evaluated on challenge y,

$$\text{LiS1dens} \left\{ 4h_1^3\omega_4^{3i} \cdot (y - h_1\omega_4^i) \right\}_{i \in [4]}$$

computeLiS2 I will only present an optimized version as the current method is complicated. See Section 5.1.4 for more details.

computeInversions Compute the inversions of denominators

$$\text{LiS0dens}, \text{LiS1dens}, \text{LiS1dens}$$

computeR0

- For the polynomial

$$\begin{aligned} C_0(X) = q_L(X^8) + X \cdot q_R(X^8) + X^2 \cdot q_O(X^8) \\ + X^3 \cdot q_M(X^8) + X^4 \cdot q_C(X^8) + X^5 \cdot S_{\sigma_1}(X^8) \\ + X^6 \cdot S_{\sigma_2}(X^8) + X^7 \cdot S_{\sigma_3}(X^8) \end{aligned}$$

  Compute the following evaluations

$$\{C_0(e) \mid e \in \mathrm{S}_0 = h_0\mathrm{H}_8\}$$

  from the given evaluations

$$\begin{aligned} q_L(h_0^8) = q_L(\xi), \quad q_R(h_0^8) = q_R(\xi), \quad q_O(h_0^8) = q_O(\xi) \\ S_{\sigma_1}(h_0^8) = S_{\sigma_1}(\xi), \quad S_{\sigma_2}(h_0^8) = S_{\sigma_2}(\xi), \quad S_{\sigma_3}(h_0^8) = S_{\sigma_3}(\xi) \end{aligned}$$

  By manually evaluating,

$$\begin{aligned} C_0(e) = q_L(\xi) + e \cdot q_R(\xi) + e^2 \cdot q_O(\xi) \\ + e^3 \cdot q_M(\xi) + e^4 \cdot q_C(\xi) + e^5 \cdot S_{\sigma_1}(\xi) \\ + e^6 \cdot S_{\sigma_2}(\xi) + e^7 \cdot S_{\sigma_3}(\xi) \end{aligned}$$

  for every $e \in \mathrm{S}_0 = h_0\mathrm{H}_8$.

- Compute the numerator $y^8 - h_0^8$.
- Compute the evaluation for y of

$$\begin{aligned} r_0(X) &= \sum_{i=0}^{7} C_0(h_0\omega_8^i) L_i^{(S_0)}(X) \\ &= \sum_{i=0}^{7} C_0(h_0\omega_8^i) \frac{1}{8h_0^7\omega_8^{7i} \cdot (X - h_0\omega_8^i)} \cdot (X^8 - h_0^8) \end{aligned}$$

  By computing,

$$\left\{ \frac{1}{8h_0^7\omega_8^{7i} \cdot (y - h_0\omega_8^i)} \cdot (y^8 - h_0^8) \right\}_{i \in [8]}$$

  using previously computed inverted values of LiS0dens and the numerator $y^8 - h_0^8$. Then, multiplying and summing using the evaluations of $C_0(X)$ over S_0.

computeR1 Using a similar strategy, evaluate the polynomial

$$r_1(X) = \sum_{i=0}^{3} C_1(h_1\omega_4^i) L_i^{(S_1)}(X)$$

on y where

$$C_1(X) := a(X^4) + X \cdot b(X^4) + X^2 \cdot c(X^4) + X^3 \cdot T_0(X^4)$$

given evaluations

$$a(h_1^4) = a(\xi), \quad b(h_1^4) = b(\xi), \quad c(h_1^4) = c(\xi)$$

and computing $T_0(h_1^4) = T_0(\xi)$.

computeR2 A similar strategy to compute $r_2(y)$ is used; however, we will avoid the discussion as the optimized description is easier. See Section 5.1.4 for more details

5.1.3 Optimizations of FFlonk Verifier: r_0 and r_1 computation

Overview The prior computation of $r_0(y), r_1(y), r_2(y)$ incur $O(n^2)$ field operations for n being their respective domain sizes. In this section, we provide a strategy to reduce the computation to $O(n \log n)$ and shaving off constant factors. The overall solution requires less multiplications, additions, and modular operations when comparing against the operations in the verifier.sol. More concretely,

1. By using a different form of lagrange polynomials over cosets, we can avoid calculating the coset values in computeChallenges. Furthermore, we shave off the excess multiplications and modular operations needed to compute the complex denominators in the old lagrange polynomial representation. This is expecially relevant since we have access to the underlying group elements as constants.
2. When naively using lagrange interpolation to evaluate $r(y)$, the full evaluations of the lagrange polynomials on y are needed. Instead, we can directly use barycentric evaluation which reduces repeated computation. Most notably, in the prior method, there was a repeat multiplication by the numerator divided by the order that could be removed out of the sum.
3. Finally, the a large chunk of the savings comes from viewing the evaluations of the $C(X)$ polynomials (needed to evaluate $r(y)$) as a fast fourier transform. By unrolling the fast fourier transform into concrete operations, the multiplications can drop from n^2 to $n \log n$ (for the relevant n).
4. We discuss additionally a different view of the r_2 polynomial, which hopefully can shave off some operations.

Lagrange Polynomials and Barycentric Evaluation over Cosets For ease of description, lets consider a group $\mathrm{H} = \langle \omega \rangle$ of order n and a coset representative c. This strategy is generic and applies to the computation of r_0, r_1 (we will discuss the edge case of r_2 later). Consider a polynomial $r(X)$ of degree less than n whose evaluations over the coset $c\mathrm{H}$ are $r_0, r_1, \ldots, r_{n-1}$. We can express $r(X)$ uniquely in lagrange basis as,

$$r(X) = \sum_{i=0}^{n-1} r_i \cdot L_i^{(c\mathrm{H})}(X)$$

However, notice that a lagrange polynomial for the coset cH can identically be represented as a lagrange polynomial over H with a shifted $X' = X/c$,

$$\begin{aligned} r(X) &= \sum_{i=0}^{n-1} r_i \cdot L_i^{(c\mathrm{H})}(X) \\ &= \sum_{i=0}^{n-1} r_i \cdot L_i^{(\mathrm{H})}\left(\frac{X}{c}\right) \\ &= \sum_{i=0}^{n-1} \frac{r_i \omega^i}{n} \cdot \frac{(X')^n - 1}{X' - \omega^i} \\ &= \frac{(X')^n - 1}{n} \cdot \sum_{i=0}^{n-1} \frac{r_i \cdot \omega^i}{X' - \omega^i} \end{aligned}$$

Hence to compute $r(y)$ for $y \in \mathbb{F}$,

$$r(y) = \frac{(y')^n - 1}{n} \cdot \sum_{i=0}^{n-1} \frac{r_i \cdot \omega^i}{y' - \omega^i}$$

the following quantities need to be computed

1. $y' = y/c$
2. $\frac{(y')^n - 1}{n}$

3. $\{r_i \cdot \omega^i\}_{i\in[n]}$
4. $\{\frac{1}{y'-\omega^i}\}_{i\in[n]}$

Step (1) can be done with a single inversion of c and multiplication. Step (2) can be done in one subtraction, $\log(n)+1$ multiplications (assuming $1/n$ is a given constant). Step (4) requires n subtractions (if we assume the ω^i are given as constants) and n inversions (of course all inversions can be batched). Step (3) is a special case which will expand on in the next section. With these quantities, $r(y)$ can be evaluated in $n+1$ multiplications and $n-1$ additions. By moving the left scalar out of the sum, you can save n multiplications over the naive method of computing each lagrange evaluation.

Efficiently Computing the Numerators Computing the numerators $\{r_i \cdot \omega^i\}_{i\in[n]}$ in (3) can be done efficiently using a fast fourier transform. Consider a polynomial $C(X)$ of the following form,

$$C(X) := f_0(X^n) + X \cdot f_1(X^n) + X^2 \cdot f_2(X^n) + \cdots + X^{n-1} \cdot f_{n-1}(X^n)$$

In the FFlonk verifier, we have the following equality (ignoring the special case of $r_2(X)$ which we will discuss later),

$$r_i \cdot \omega^i = C(c \cdot \omega^i) \cdot \omega^i \quad \text{for } i \in [n]$$

Plugging in the quantities,

$$\begin{aligned} C(c \cdot \omega^i) \cdot \omega^i &= f_0(c^n) \cdot \omega^i + (c \cdot \omega^i) \cdot f_1(c^n) \cdot \omega^i + (c \cdot \omega^i)^2 \cdot f_2(c^n) \cdot \omega^i + \cdots + (c \cdot \omega^i)^{n-1} \cdot f_{n-1}(c^n) \cdot \omega^i \\ &= f_0(c^n) \cdot \omega^i + (c \cdot f_1(c^n)) \cdot (\omega^i)^2 + (c^2 \cdot f_2(c^n)) \cdot (\omega^i)^3 + \cdots + (c^{n-1} \cdot f_{n-1}(c^n)) \end{aligned}$$

This is equivalent to evaluating the following polynomial on ω^i,

$$C'(X) = c^{n-1} f_{n-1}(c^n) + f_0(c^n)X + cf_1(c^n)X^2 + \cdots + c^{n-2} f_{n-2}(c^n)X^{n-1}$$

Thus, we can calculate the numerators by

1. Computing the vector of coefficients

   $$\text{coeffs}[c^{n-1} f_{n-1}(c^n),\ f_0(c^n),\ cf_1(c^n),\ c^{n-2} f_{n-2}(c^n)]$$

   We can assume (in the computation of $r0, r1$) that the quantities

   $$f_0(c^n),\ f_1(c^n),\ \ldots,\ f_{n-1}(c^n)$$

   are already given (with the exception of a few manually computed values). Computing the vector of coefficients coeffs takes $2n$ multiplications.

2. Compute a fast fourier transform on H in $n \log n$ multiplications,

   $$C'(H) = \text{FFT}(\text{H}, \text{coeffs}) = \{r_i \cdot \omega^i\}_{i\in[n]}$$

   which gives us the desired numerators. To be explicit, the fast fourier transform will have to be unrolled into concrete operations in solidity. This will avoid the repeat computation currently done in the verifier to compute $C(X)$ over each domain point in the coset.

5.1.4 Optimizations of FFlonk Verifier: r_2 computation

Lagrange Polynomials for Union of Two Cosets For ease of description, lets consider a group $\mathsf{H} = \langle\omega\rangle$ of order n and two coset representatives $c_1,\ c_2$. Define the set $Sc_1\mathsf{H} \cup c_2\mathsf{H}$ as the union of two cosets. Define the following constants,

$$s_1(c_1/c_2)^n - 1, \qquad s_2(c_2/c_1)^n - 1$$

Define $X_1' X/c_1$ and $X_2' X/c_2$. Define the lagrange polynomials for S, for $i \in [2n]$,

$$L_i^{(S)}(X)\frac{\omega^i}{n} \cdot \frac{(X_1')^n - 1}{(X_1' - \omega^i)} \cdot \frac{(X_2')^n - 1}{s_1} \text{if } i \in [0, n-1] \frac{\omega^i}{n} \cdot \frac{(X_1')^n - 1}{s_2} \cdot \frac{(X_2')^n - 1}{(X_2' - \omega^i)} \text{if } i \in [n, 2n-1]$$

Barycentric Evaluation for S Consider a polynomial $r(X)$ of degree less than n whose evaluations over S are $r_0, r_1, \ldots, r_{2n-1}$. We can express $r(X)$ uniquely in lagrange basis over S as,

$$\begin{aligned}
r(X) &= \sum_{i\in[2n]} r_i \cdot L_i^{(S)}(X) \\
&= \sum_{i\in[n]} r_i \cdot L_i^{(S)}(X) \quad + \sum_{i\in[n,2n-1]} r_i \cdot L_i^{(S)}(X) \\
&= \sum_{i=0}^{n} r_i \cdot \frac{\omega^i}{n} \cdot \frac{(X_1')^n - 1}{(X_1' - \omega^i)} \cdot \frac{(X_2')^n - 1}{s_1} \quad + \quad \sum_{j=n}^{2n-1} r_j \cdot \frac{\omega^j}{n} \cdot \frac{(X_1')^n - 1}{s_2} \cdot \frac{(X_2')^n - 1}{(X_2' - \omega^j)} \\
&= \frac{[(X_1')^n - 1] \cdot [(X_2')^n - 1]}{n} \cdot \left(\frac{1}{s_1} \cdot \sum_{i=0}^{n} \frac{r_i \cdot \omega^i}{(X_1' - \omega^i)} \quad + \quad \frac{1}{s_2} \cdot \sum_{j=n}^{2n-1} \frac{r_j \cdot \omega^j}{(X_2' - \omega^i)} \right)
\end{aligned}$$

Hence to compute $r(y)$ for $y \in \mathbb{F}$,

$$r(y) = \frac{[(y_1')^n - 1] \cdot [(y_2')^n - 1]}{n} \cdot \left(\frac{1}{s_1} \cdot \sum_{i=0}^{n} \frac{r_i \cdot \omega^i}{(y_1' - \omega^i)} \quad + \quad \frac{1}{s_2} \cdot \sum_{j=n}^{2n-1} \frac{r_j \cdot \omega^j}{y_2' - \omega^i)} \right)$$

The following quantities need to be computed

1. $y_1' = y/c_1$, $y_2' = y/c_2$
2. $\frac{[(y_1')^n - 1] \cdot [(y_2')^n - 1]}{n}$
3. $s_1 = (c_1/c_2)^n - 1$, $s_2 = (c_2/c1)^n - 1$
4. $1/s_1$, $1/s_2$
5. $\{r_i \cdot \omega^i\}_{i\in[n]}$, $\{r_j \cdot \omega^j\}_{j\in[n,2n-1]}$
6. $\{1/(y_1' - \omega^i)\}_{i\in[n]}$, $\{1/(y_2' - \omega^j)\}_{j\in[n,2n-1]}$

Steps (1), (2) (6) can be computed using similar strategies in Section 5.1.3. Step (3) can be performed efficiently by

- Computing $(c_1/c_2)^n$ and inverting to obtain $(c_2/c_1)^n$.
- Subtracting one from each.

Step (4) requires inverting both quantities. Step (5) requires performing the tricks in 5.1.3 to efficiently compute the numerators. This involves performing two FFTs over H.

5.1.5 FFTs over Smooth Domains

Let $\mathbb{F}$ be an FFT-friendly field with order $|\mathbb{F}| = p$ such that $(p-1)$ is divisible by powers of small primes. For example, the scalar field in FFlonk in the solidity verifier has order

$$p = 21888242871839275222246405745257275088548364400416034343698204186575808495617$$

$$\begin{aligned}(p-1) = 2^{28} * 3^2 * 13 * 29 * 983 * 11003 * 237073 * 405928799 * 1670836401704629 \\ * 138183644341974388644693380 81\end{aligned}$$

Let $H \subseteq \mathbb{F}^\times$ be a smooth multiplicative subgroup such that it's order $|H| = n$ factors into powers of small primes. The Cooley–Tukey FFT algorithm or Mixed-radix FFT (when generalized to finite fields) is an algorithm that can efficiently evaluate a polynomial of degree less than n over H. We can think of this as just an FFT algorithm where the domain size does not need to be a power of two.

The core idea is that instead of splitting each FFT step into two halves (even and odd), we can split the FFT step into r chunks where r is the current prime factor we are iterating on. This r is often referred to as a *radix*. For example, in a standard FFT the radix is always equal to 2.

On the following page is code written in Sage that can be executed in an online interpreter. We use the scalar field in the FFlonk verifier and a subgroup of size $n = 2^2 * 3^2 * 13$. The radixes will be 2, 3, and 13.

This code is overly generic to accommodate any subgroup size and choice of radix's, but if a specific subgroup factorization is known then the code can tailored for those radix's. Note, that this algorithm is identical to the standard FFT when the subgroup order n is a power of 2.

```
p = 21888242871839275222246405745257275088548364400416034343698204186575808495617
# F is a field whose size is p
F = GF(p)
# define x to be a variable
R.<x> = PolynomialRing(F)

# g is the multiplicative generator with order (p-1)
g = F.multiplicative_generator()

# Implementation of Mixed-Radix FFT / Cooley-Tukey FFT for finite fields
# vals are the coeffs
# w is the multiplicative generator
# For n = p1^e1 * p2^e2 * ..., factors = [(p1, e1), (p2, e2), ...]
def fft_helper(vals, w, factors):
    n = len(vals)
    # Base case
    if n == 1:
        return vals

    # r is the current radix
    r = factors[0][0]

    # decrementing exponent
    # [(p1, e1), (p2, e2), ...]
    # goes to [(p1, e1-1), (p2, e2), ...]
    # or [(p2, e2), ...] if e1 = 1
    factors = list(factors)
    factors[0] = (factors[0][0], factors[0][1]-1)
    if factors[0][1] == 0:
        factors = factors[1:]

    # Recursive Step
    # acc = [fft0, fft1, fft2, ...]
    acc = list()
    for i in range(r):
        acc.append(fft_helper(vals[i::r], w^r, factors))

    # Allocate list for evaluations
    evals = [0 for _ in range(n)]

    # Accumulate recursive steps
    shift = n // r
    # acc = [fft0, fft1, fft2, ...]
    # Iterate as [t0 = (fft0[0], fft1[0], fft2[0]), t1 = (fft0[1], fft1[1], fft2[1]),...]
    for i, t in enumerate(zip(*acc)):
        for j in range(r):
            evals[i + shift * j] = sum((w^(i+shift*j))^k * t[k] for k in range(r))
    return evals

# w is the multiplicative generator
# vals are the coeffs
def fft(w, vals):
    factors = list(factor(w.multiplicative_order()))
    return fft_helper(vals, w, factors)

# Domain Size
n = 2^2 * 3^2 * 13
# w is generator for group of size n
w = g^((p-1)/n)

coeffs = [F.random_element() for _ in range(n)]
poly = sum([c*x^i for i,c in enumerate(coeffs)])
fft_evals = fft(w, coeffs)
true_evals = [poly(w^i) for i in range(n)]
print(fft_evals == true_evals)
```

5.2 Custom PIL Constraints

In this review, the team verified the correctness of the custom PIL constraints corresponding to custom templates used in the STARK verifier circom descriptor. The steps for this proving pipeline are as follows:

- (A) STARK verifier described using a circom descriptor including custom templates
- (B) STARK verifier circom descriptor compiled to R1CS constraints where custom templates do not generate constraints
- (C) Circom-generated R1CS constraints are written as PIL constraints for a STARK verifier PIL descriptor
- (D) Custom PIL constraints are added to complete the STARK verifier PIL descriptor to capture the circom custom templates

This review addresses step (D) of this pipeline, examining the custom PIL constraints for 3 custom templates:

- Fp3 multiplication (circuits.gl/cmul.circom)
- 1-of-4 Fp3 selector (circuits.gl/treeselector4.circom)
- Poseidon hash evaluation (circuits.gl/{poseidon.circom, poseidon_constants.circom})

The relevant files for PIL constraint generation for the C12 and C18 PIL descriptors (referring to the number of committed witness polynomials) that were reviewed in this audit include:

- src/compressor/{compressor12.pil.ejs, compressor12_setup.js}
- src/compressor/{compressor18.pil.ejs, compressor18_setup.js}

The review was conducted on the following commit: bb2404

5.2.1 Fp3 Multiplication (CMUL)

No issues found.

5.2.2 1-of-4 Fp3 Selector (TreeSelector4)

INFO: Documentation for "keys" constraints assume binary constraints The comments on lines 326, 335, 344, 353 claim that the "keys" polynomial will only be 1 if the key bit polynomials a[12] and a[13] are set to appropriate bits. This is true if a[12] and a[13] are restricted to {0,1}, however that constraint does not exist. In fact, the TreeSelector4 functions even when a[12] and a[13] are not restricted in this manner.

More accurately, the following bijections are given from the existing PIL constraints:

- keys1 = 0 a[12] = 1 or a[13] = 1
- keys2 = 0 a[12] = 0 or a[13] = 1
- keys3 = 0 a[12] = 1 or a[13] = 0
- keys4 = 0 a[12] = 0 or a[13] = 0

This enforces that there is no setting of a[12] and a[13] such that ALL of keys{1,2,3,4} can be set to 0, meaning that the selector is applied for at least one branch.

Suggestion: Without loss of generality, replace comment with "keys1 will be non-zero unless a[12]=0 or a[13]=0.

5.2.3 Poseidon Hash Evaluation (Poseidon / CustPoseidon)

POSSIBLE SOUNDNESS ISSUE: No binary enforcement on CustPosedion Selection Key In lines 93-100, the witness polynomial a[8] is used to select the custom Poseidon input order, however it assumes that a[8] takes on a binary value {0,1}. If a[8] is unrestricted, a prover can set a[8] to make the input a different value. Unclear if this exploitable in the context of how it is used within the STARK verifier due to the collision-resistance of Poseidon.

Suggestion: Include a polynomial constraint to restrict a[8] to binary when reading custom Poseidon inputs:

POSEIDONCUSTFIRST * (1 - a[8]) * (a[8]) = 0

OPTIMIZATION: POSEIDONM polynomial can be used to capture multiple rounds In line 196, POSEIDONM, POSEIDONAFTERPART, POSEIDONFIRST, POSEIDONCUSTFIRST are all used as a selector to propagate the values stored in the poseidonM polynomial to the next row. While POSEIDONAFTERPART, POSEIDONFIRST, and POSEIDONCUSTFIRST all have other important functionality, POSEIDONM is just used because it also needs to include the propagation from Round 30 -> output row.

Suggestion: The polynomial constraint can be simplified by defining POSEIDONM to additionally be set to 1 at the same points as POSEIDONAFTERPART, POSEIDONFIRST, and POSEIDONCUSTFIRST.

POSEIDONM * poseidonM<%- i + 12 + 1 %> = 0;