OpenZeppelin | security

FFLONK Verifier Audit

ZKsync

September 26, 2024

Table of Contents

Table of Contents 2

Summary 4

Scope 5

Phase 1 5

Phase 2 5

Phase 3 5

System Overview 7

Security Model and Trust Assumptions 8

Design Considerations and Trade-Offs 8

Critical Severity 10

C-01 Lack of Input Validation Allows Proof Forgery for Arbitrary Public Inputs - Phase 1 10

Medium Severity 13

M-01 Missing Proof Validation - Phase 1 13

Low Severity 14

L-01 Incorrect Bound Check in Lagrange Polynomial Evaluations - Phase 1 14

L-02 Two Representations of the Point-At-Infinity - Phase 1 14

L-03 Missing Docstrings - Phase 1 15

L-04 Brittle Use of Free Memory Pointer - Phase 1 15

L-05 Misleading Documentation - Phase 1 16

L-06 Lack of Validation of Proof Size - Phase 3 17

Notes & Additional Information 17

N-01 Some Memory Variables Could Be Constants – Phase 2 17

N-02 Lack of SPDX License Identifier - Phase 1 18

N-03 Lack of Security Contact - Phase 1 18

N-04 State Variable Visibility Not Explicitly Declared - Phase 1 19

N-05 The Codebase Cannot Be Compiled With the Default Settings - Phase 1 19

N-06 Invalid Reverts - Phase 1 20

N-07 Code Quality and Readability Suggestions - Phase 1 20

N-08 Gas Optimizations - Phase 1 21

N-09 Overwriting of the Free Memory Pointer - Phase 3 22

N-10 Use of Magic Number - Phase 3 22

N-11 Inconsistent Representation of Hex Values - Phase 3 23

N-12 Missing or Misleading Documentation - Phase 3 23

N-13 Missing Documentation About Treatment of Point at Infinity - Phase 3 24
N-14 PLONK verify Function Has public Visibility - Phase 3 24

Conclusion 26

Summary

Type	ZK Rollup
Timeline	From 2024-07-22 To 2024-09-13
Languages	Solidity, Yul

Total Issues	22 (21 resolved)
Critical Severity Issues	1 (1 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	1 (1 resolved)
Low Severity Issues	6 (6 resolved)
Notes & Additional Information	14 (13 resolved)

Scope

The audit took place in three phases.

Phase 1

In the first phase, we audited the matter-labs/fflonk-verifier repository at commit 3881e18.

In scope was the following file:

```
fflonk-verifier
└── solidity/fflonk-foundry/src/FflonkYul.sol
```

Phase 2

In the second phase, we audited the matter-labs/fflonk-verifier repository at commit 3c09dfd. This audit was focused on an update of the verifier and had the same scope as phase 1:

```
fflonk-verifier
└── solidity/fflonk-foundry/src/FflonkYul.sol
```

Phase 3

In the third phase, we audited the matter-labs/fflonk-verifier repository at commit c9e99a8, which was an update of the FFLONK verifier with the same scope as phases 1 and 2:

```
fflonk-verifier
└── solidity/fflonk-foundry/src/FflonkYul.sol
```

In addition, we audited the matter-labs/era-contracts repository at commit 49868af, with the goal of ensuring that the validator calls to prove batches were compatible with the verifier. The scope was comprised of portions of the following files:

```
era-contracts/l1-contracts/contracts/state-transition
├── chain-deps
│   └── facets
│       └── Executor.sol
```

```
├── chain-interfaces
│   ├── IExecutor.sol
│   └── IVerifier.sol
└── Verifier.sol
```

Specifically, for `Executor.sol`, the proving flow consisting of the `proveBatchesSharedBridge` function and its subsequent function calls were audited. This audit was performed with the goal of ensuring completeness and soundness, assuming all prior steps were functioning as intended. As such, all prior operations such as committing batches, as well as latter operations such as executing batches, were deemed out of scope. These changes will be audited as part of a separate audit. As mentioned below, the circuit itself was also deemed out of scope.

The `IExecutor.sol` and `IVerifier.sol` interfaces were checked to ensure compatibility with both the current `Verifier.sol` and the new `FFlonkYul.sol` file. The current verifier, located in `Verifier.sol`, was assumed to be functioning correctly outside of this interface change.

All resolutions in this report related to `FflonkYul.sol` are contained at commit 8aaf8b7 in the fflonk-verifier repository. This commit marks the final state of the audited contract.

System Overview

The upgrade under review aims to reduce the gas required for on-chain verification. When L2 blocks are produced, the sequencer submits them to the L1 `Executor` contract and a prover generates a proof of the associated state transitions. This proof generation is done with layers of recursion, finally resulting in the final proof that is submitted on-chain. Upon submission of this proof, the `Executor` contract calls the ZKP verifier contract to verify its validity. It is here that the verifier determines whether or not the L2 state transitions are valid, which would lead to the finalization of the L2 blocks on L1.

The current on-chain verifier uses a custom PLONK protocol and the upgrade seeks to replace this with the FFLONK protocol instead. The trade-off that FFLONK makes is to increase the number of prover operations in favor of reducing the number of scalar multiplications for the verifier. As the proving takes place off-chain but the verification takes place on-chain, it is important to reduce the verifier's work in terms of gas. As of this writing, each scalar multiplication requires calling the `ecMul` precompile, which consumes at least 6000 gas. With the FFLONK protocol, the number of scalar multiplications done by the verifier is reduced from 16 to 5.

During phase 1, the codebase used powers of a 4-th root of unity for the openings of the combined polynomials of both the first and the second rounds. The update audited in phase 2 used 3-rd roots of unity for the second round instead. This update further decreases gas costs by reducing the number of openings of C_2 by 2, as well as the number of inverses sent for the corresponding Lagrange interpolations. During phase 3, the Lagrange inverses were batched using the Montgomery batched inversion technique, thereby reducing the number of elements sent by the prover in calldata to 1. Combined with other optimizations, this resulted in further reducing gas costs.

Security Model and Trust Assumptions

As stated earlier, the verifier is called by the `Executor` contract to perform validation of the ZK-SNARK. During phases 1 and 2 of this audit, the `Executor` code calling the FFLONK verifier had not been written. Therefore, those portions of the audit were performed with the assumption that the new `Executor` contract would have validations similar to the one that was in place. During phase 3 of the audit, the `Executor` contract was updated to integrate with the new verifier. As previously mentioned, the proof verification part of `Executor`, in conjunction with the verifier, was then checked for soundness and completeness, assuming all prior steps had been performed as intended. While the current ZKsync system only allows privileged addresses to call the external function in `Executor` to prove state transitions, the eventual goal is to decentralize the prover.

In addition, the combined preprocessed polynomial C_0 was taken as-is for the purpose of our audit. As such, the fact that the circuit represented by this commitment is well-implemented is considered a trust assumption. For the sake of transparency, we recommend that the Matter Labs team provides code and tools to help anyone who is interested check that the desired circuit is correctly represented by the commitment C_0. In the same vein, it is assumed that the trusted setup to produce the SRS was done properly and the toxic waste was discarded. We recommend adding verbiage that makes it clear how this process was performed, along with checks for the well-formedness of the SRS.

Design Considerations and Trade-Offs

As outlined above, there is a trade-off between the prover and verifier work when switching from PLONK to FFLONK. There is also a trade-off between proof size and verifier work. For example, if a prover calculates and sends more elements to the verifier, the verifier may only need to check the validity of these elements instead of computing them for itself. Since the proof is passed as calldata in the call to the verifier, additional proof elements result in increased gas costs. However, there is a reduction in the work the verifier has to do to compute these elements. A concrete example can be found in this codebase, where the Lagrange inverses are computed and sent to the verifier, which only validates them. This approach offers significant gas savings compared to having the verifier compute the inverses.

To further improve gas savings, the suggestion was made during phases 1 and 2 to use the Montgomery batched inversion technique to combine all the inverses into one. In reducing the number of elements sent in the calldata, the gas cost is reduced further. This was later implemented and audited in phase 3. A suggestion to switch to 3-rd roots of unity to reduce gas costs was also implemented and audited in phase 2.

After auditing the codebase over all three phases, the following is an additional suggested design change that could further reduce the gas costs of the verifier:

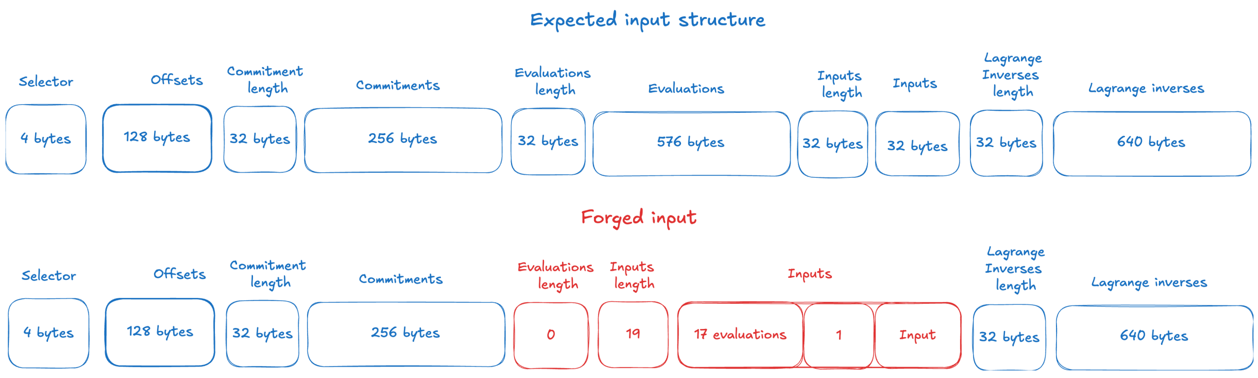
- Besides the inverses of the field elements involved in the computation of Lagrange polynomial evaluations, there are additional elements whose inverses are still computed by calling the `modexp` precompile with the $R-2$ power. Each such call costs ~1400 gas. Consider computing them as part of the Montgomery batched inversion.

Critical Severity

C-01 Lack of Input Validation Allows Proof Forgery for Arbitrary Public Inputs - Phase 1

During the validation of a proof, a transcript is built by appending elements sent by the prover. This transcript contains the public inputs, the commitments, and the evaluations. When adding some of these elements to the transcript, for example, the evaluations, the number of evaluations to be added is [read from a fixed offset of the calldata](). Challenges are then computed by [hashing this transcript]().

However, the length of the `evaluations` array, as read from calldata, is not validated to be equal to 18. This makes it possible to send an `evaluations` array of size 0, followed by an `inputs` array of size 19. The first 17 elements of the `inputs` would be the remaining evaluations, followed by a `1` ([validated]() to be the length of the public input), followed by the public input itself. The illustration of the layout of the expected calldata versus the forged calldata is shown below:



In addition, given that the inputs are calldata arrays and are only used in assembly, there is no implicit abi decoding of these arrays. Thus, they are not validated to be correctly abi encoded. A second possible attack vector could be the caller sending a sequence of bytes in calldata with the wrong length for the `evaluations` array (an invalid abi encoded array) to control the number of evaluations added to the transcript. [For example](), the length could be set to 0 in the calldata so that no evaluations are added to the transcript at all.

Since the number of elements added to the transcript depends on the value of the length of `evaluations`, in both of the aforementioned scenarios, an attacker would be able to change the evaluations without affecting the challenges since they are not added to the transcript. It is, in fact, possible to change these evaluations in both of these cases to forge proofs and successfully pass the pairing check for any public input of the attacker's choosing. This is done by solving the pairing equation using the evaluations as degrees of freedom. Here is an example of how such an attack could be carried out:

1. The malicious prover picks a public input. For example, in the case of a ZK rollup, the public input could correspond to an invalid state transition to drain funds from users.

2. The malicious prover first simulates a SHPLONK PCS opening, following the protocol outlined in Lemma 4.2 of the FFLONK paper, where the polynomials opened are $C_0(X)$, $C_1(X) = X$ and $C_2(X) = X$. Note that $C_0(X)$ here is the polynomial committed to in the verification key. The $C_1(X)$ and $C_2(X)$ polynomials can be set to arbitrary polynomials for the sake of this PCS, and are set to X for simplicity. The SHPLONK protocol is followed honestly.

    1. The protocol is adapted so that the challenges, noted γ and z in SHPLONK, are computed exactly the same way as α and y in the FFLONK verifier. For the sake of consistency, we will keep the notation of α and y for these challenges as we use z for the permutation polynomial.
    2. The commitments $[C_0]_1$, $[C_1]_1$, $[C_2]_1$, $[W]_1$, and $[W']_1$, as well as the Lagrange interpolation evaluations $R_0 := r_0(y)$, $R_1 := r_1(y)$ and $R_2 := r_2(y)$ are computed. The value of the challenges α and y are also saved to be used later.
    3. At the end of the protocol, these values pass the pairing check $e([C_0]_1 + \alpha\frac{Z_{T\setminus S_1}(y)}{Z_{T\setminus S_0}(y)}[C_1]_1 + \alpha^2\frac{Z_{T\setminus S_2}(y)}{Z_{T\setminus S_0}(y)}[C_2]_1 - (R_0 + \alpha\frac{Z_{T\setminus S_1}(y)}{Z_{T\setminus S_0}(y)}R_1 + \alpha^2\frac{Z_{T\setminus S_2}(y)}{Z_{T\setminus S_0}(y)}R_2) \cdot [1]_1 - \frac{Z_T(y)}{Z_{T\setminus S_0}(y)}[W]_1 + y[W']_1, [1]_2) \cdot e(-[W']_1, [x]_2) = 1$.

3. The malicious prover reuses the same commitments in the FFLONK proof. Denoting `REvals` as the array computed in the code that contains `r_0(y)`, `r_1(y)` and `r_2(y)`, the malicious prover wants to change the evaluations sent in the proof to solve the equation: $REvals[0] + \alpha * \frac{Z_{T\setminus S_1}(y)}{Z_{T\setminus S_0}(y)} * REvals[1] + \alpha^2 * \frac{Z_{T\setminus S_2}(y)}{Z_{T\setminus S_0}(y)} * REvals[2] = R_0 + \alpha * \frac{Z_{T\setminus S_1}(y)}{Z_{T\setminus S_0}(y)} * R_1 + \alpha^2 * \frac{Z_{T\setminus S_2}(y)}{Z_{T\setminus S_0}(y)} * R_2$

4. We denote T as the right-hand side of this equation, which the malicious prover can calculate using the values obtained in the previous steps. We also have that the verifier computes and/or checks the following constraints, which must be satisfied.

    1. $T_0(\zeta) \cdot Z_H(\zeta) = q_a(\zeta)a(\zeta) + q_b(\zeta)b(\zeta) + q_c(\zeta)c(\zeta) + q_m(\zeta)a(\zeta)b(\zeta) + q_{const}(\zeta) + PI(\zeta)$
    2. $T_1(\zeta) \cdot Z_H(\zeta) = z(\zeta)(a(\zeta) + \beta\zeta + \gamma)(b(\zeta) + k_1\beta\zeta + \gamma)(c(\zeta) + k_2\beta\zeta + \gamma)$ $-z(\zeta\omega)(a(\zeta) + \beta S_{\sigma_1}(\zeta) + \gamma)(b(\zeta) + \beta S_{\sigma_2}(\zeta) + \gamma)(c(\zeta) + \beta S_{\sigma_3}(\zeta) + \gamma)$

    3. $T_2(\zeta) \cdot Z_H(\zeta) = L_0(\zeta)(z(\zeta) - 1)$

5. Solving the equation can, for example, be done by sending the following $evaluations$:

    1. Set $z(\zeta) = 1$, which implies $T_2(\zeta) = 0$, by constraint iii above.
    2. Set $z(\zeta\omega) = 1$, $S_{\sigma_1}(\zeta) = \zeta, S_{\sigma_2}(\zeta) = k_1\zeta, S_{\sigma_3}(\zeta) = k_2\zeta$. This implies $T_1(\zeta) = 0$, by constraint ii above.
    3. Set $q_{const}(\zeta) = -PI \cdot L_0(\zeta), a(\zeta) = b(\zeta) = c(\zeta) = 0$. This implies $T_0(\zeta) = 0$.
    4. Thus, we have
        1. $\forall x \in S_1 = \{h_1, h_1\omega_4, h_1\omega_4^2, h_1\omega_4^3\}, C_1(x) = 0$. Therefore, $REvals[1] = 0$
        2. $\forall x \in S_2 = \{h_2, h_2\omega_4, h_2\omega_4^2, h_2\omega_4^3\}, C_2(x) = 1$. Therefore, $REvals[2] = \sum_{i=0}^{3} L_i^{(S_2)}(y) + \sum_{j=4}^{7} L_j^{(S_2)}(y)$, noted A, constant
    5. Set $T_1(\zeta\omega) = T_2(\zeta\omega) = 0$. This implies $\forall x \in S_3 = \{h_3, h_3\omega_4, h_3\omega_4^2, h_3\omega_4^3\}, C_{2_{shifted}}(x) = 1$
    6. Given the values of $S_{\sigma_1}(\zeta), S_{\sigma_2}(\zeta), S_{\sigma_3}(\zeta)$ from above, we have that $\forall x \in S_0 = \{h_0, h_0\omega_8, h_0\omega_8^2, h_0\omega_8^3, h_0\omega_8^4, h_0\omega_8^5, h_0\omega_8^6, h_0\omega_8^7\}$: $C_0(x) = q_a(\zeta) + xq_b(\zeta) + x^2q_c(\zeta) + x^3q_m(\zeta) + x^4q_{const}(\zeta) + x^5\zeta + x^6k_1\zeta + x^7k_2\zeta$. In our case, $q_a(\zeta) = 19$ as it is the first value read from the `evaluations` array.
    7. Let us set $q_c(\zeta) = q_m(\zeta) = 0$. Then
        1. $REvals[0] = (\sum_{i=0}^{7} L_i^{(S_0)}(y) \cdot (h_0 w_8^i)) \cdot q_b(\zeta) + C = B \cdot q_b(\zeta) + C$ where $C := \sum_{i=0}^{7} L_i^{(S_0)}(y) \cdot (19 + h_0^4 w_8^{4i}(-PI \cdot L_0(\zeta)) + h_0^5 w_8^{5i}\zeta + h_0^6 w_8^{6i} k_1\zeta + h_0^7 w_8^{7i} k_2\zeta)$ and where $B := \sum_{i=0}^{7}(h_0 w_8^i)L_i^{S_0}(y)$
    8. The malicious prover then solves $B \cdot q_b(\zeta) + C + \alpha^2 \frac{Z_{T \setminus S_2}(y)}{Z_{T \setminus S_0}(y)} \cdot A = T$. It is solved by setting $q_b(\zeta) = \frac{T - \alpha^2 \frac{Z_{T \setminus S_2}(y)}{Z_{T \setminus S_0}(y)} A - C}{B}$
6. The malicious prover sends the commitments $[C_0]_1$, $[C_1]_1$, $[C_2]_1$, $[W]_1$, $[W']_1$ and all the evaluations described above as part of the proof.
7. Because changing the evaluations did not change the challenges, the FFLONK pairing check is exactly the same as the one done in SHPLONK in step 2. Thus, the pairing check passes for this malicious public input, chosen by the prover.

If a ZK rollup relied on this verifier for its state transition and did not perform the checks previously mentioned, it would be possible to forge proofs and drain the rollup. Importantly, we note that as this audit was focused only on the verifier itself, we did not have access to the code that calls the verifier. It is, therefore, possible that checks identified as missing in this issue were in fact assumed to be made by the caller. However, upon reviewing the current contract calling the `verify` function, these checks are not performed by the caller.

Consider adding checks which ensure that the input arrays are correctly abi encoded and that their lengths match what is expected. We recommend using fixed-sized arrays as inputs. This would natively protect against size mismatches while also saving gas by not sending the offsets and lengths associated with dynamic types in calldata (~1200 gas). If such checks are expected to be made in the caller of the `verify` function, consider documenting this assumption in the verifier. As a general strategy against this kind of attack, we recommend ensuring that the transcript always contains everything, especially when the prover could be adversarial.

Update: *Resolved in pull request #3 at commit 9a89ab8. The team stated:*

> *The implementation now works with fixed-size arrays.*

Medium Severity

M-01 Missing Proof Validation - Phase 1

The `Verify` function does not validate the following:

- Whether the polynomial openings are valid field elements
- Whether the public witness values are valid field elements
- Whether the proof commitments are valid elliptic curve points
- Whether the proof commitments are on the correct subgroup

The codebase assumes the use of the BN254 curve which does not have non-trivial proper subgroups. Therefore, if only BN254 support is intended, checking whether the proof commitments are valid elliptic curve points will suffice, and it is not necessary to also check that they are on the correct subgroup. The check in bullet point 4 above can thus be skipped if is it the only supported curve. In addition to the check for bullet point 3, we also recommend the first two validations listed above, in order to future-proof the verifier.

Please note that dependence on precompiles to perform these checks would not always be effective. For example, the code for `point_add` circumvents this call to the precompile at times when `matched` is set to 1.

While the Fflonk paper does not explicitly mention these checks as it simply describes its novel contribution on top of Plonk, these checks are mentioned explicitly in the Plonk paper. In the interest of reducing the attack surface and increasing compliance with the Plonk specification, consider introducing these checks.

Update: *Resolved in pull request #3 at commit d7b8335. The team stated:*

> *The validations are now added.*

Low Severity

L-01 Incorrect Bound Check in Lagrange Polynomial Evaluations - Phase 1

The `precompute_lagrange_basis_evaluations_from_inverses_for_union_set` function computes the Lagrange polynomial evaluations $L_i^{(S_2)}(y)$, where $S_2 = \{h_2, h_2\omega_4, h_2\omega_4^2, h_2\omega_4^3\} \cup \{h_3, h_3\omega_4, h_3\omega_4^2, h_3\omega_4^3\}$. This function does a bound check to validate that it does not read past the end of the `lagrange_basis_inverses` array.

However, this check is inaccurate as it does not account for the union set. In practice, it checks that `start + num_polys = 16 <= lagrange_basis_inverses.length`, but the function reads up to index `start + 2 * num_polys - 1 = 19`.

Consider modifying the bound check to validate that `start + 2 * num_polys <= lagrange_basis_inverses.length`.

Update: *Resolved in pull request #3 at commit 9ddb5d3. The team stated:*

> *The bound check has now been fixed*

L-02 Two Representations of the Point-At-Infinity - Phase 1

In the `point_mul` function, there is an `if` block that sets the point (0, 1) to the value (0, 0). Therefore, there are two representations of the point-at-infinity when calling this function, (0, 1) and (0, 0). While not necessarily a practical issue currently, this inconsistency in representation increases the attack surface for a malicious prover who now has two options to represent this point. Furthermore, this inconsistency could lead to more serious vulnerabilities down the road when the code is refactored or if additional calling functions are created.

Consider removing this `if` block and making (0, 0) the only valid representation of the point-at-infinity.

Update: *Resolved in pull request #3 at commit 4f2e2f1. The team stated:*

> *The `point_mul` has now been corrected to have only one representation of point-at-infinity (0,0).*

L-03 Missing Docstrings - Phase 1

In the `FflonkYul` contract, multiple instances of missing docstrings were identified:

- The `verify` function should document the structure of the inputs and any assumptions related to the checks made by the caller.
- The core functions `check_main_gate_identity`, `precompute_lagrange_basis_evaluations_from_inverses`, `precompute_lagrange_basis_evaluations_from_inverses_for_union_set`, `compute_opening_points`, and `evaluate_r_polys_at_point_unrolled` should be documented.
- The helper functions `update_transcript`, `get_challenge`, `point_mul`, `point_add`, `pairing_check`, and `revertWithMessage` functions should be documented.

Consider thoroughly documenting all functions (and their parameters) that are part of any contract's public API. Functions implementing sensitive functionality, even if not public, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Update: *Resolved in pull request #3 at commit 8ced28d. The team stated:*

> *The functions have now been documented with the necessary details.*

L-04 Brittle Use of Free Memory Pointer - Phase 1

The `point_mul`, `point_add`, and `pairing_check` functions do not return their result. Instead, they leave it at the location of the free memory pointer, relying on the calling function to obtain the resulting value from that memory location. This lack of an explicit return results in brittle code that can be broken in a future refactoring of the calling function. Furthermore, for `point_mul` and `point_add`, the resulting elliptic curve point is represented by two words of memory, thereby making this connection even more tenuous. If the calling function overwrites these memory locations with other values, errors could be introduced into the codebase.

To make the code more robust, consider explicitly returning the output of the aforementioned functions using return variables in assembly.

Update: *Resolved in pull request #3 at commit c21ab95 and at commit 3c09dfd. The team stated:*

> *The concerned functions now return values instead of storing them at the free memory pointer.*

L-05 Misleading Documentation - Phase 1

Throughout the `FflonkYul.sol` file, multiple instances of misleading comments were identified:

- When updating the transcript, the comment in line 175 says, "commit third round commitment: copy-perm and lookup". However, the current codebase does not use lookup tables.
- The comment in line 190 of the code was commented out and could be removed.
- The denominator in the comment in line 302 should be in parenthesis to clarify the precedence of the operations.
- The point at which the vanishing polynomial is evaluated in `W'` is indicated as `x` whereas it should be `y`. Similarly, there are other instances of the incorrect use of `x` instead of `y`.
- The comment in line 660 should say `(alpha^2*Z_{T\S2}(y)/Z_{T\S0}(y)) * C2` and a comment should be inserted before line 664 `C0 + (alpha^2*Z_{T\S2}(y)/Z_{T\S0}(y)) * C2`. Similarly, the comment in line 667 should say `(alpha^2*Z_{T\S2}(y)/Z_{T\S0}(y)) * r2`.
- The comment on line 674 should say `(alpha*Z_{T\S1}(y)/Z_{T\S0}(y)) * C1` and a comment should be inserted before line 678 `C0 + (alpha*Z_{T\S1}(y)/Z_{T\S0}(y)) * C1 + (alpha^2*Z_{T\S2}(y)/Z_{T\S0}(y)) * C2`. Similarly, the comment on line 682 should say `(alpha*Z_{T\S1}(y)/Z_{T\S0}(y)) * r1 + (alpha^2*Z_{T\S2}(y)/Z_{T\S0}(y)) * r2`.
- The comment on line 685 should say `r0 + (alpha*Z_{T\S1}(y)/Z_{T\S0}(y)) * r1 + (alpha^2*Z_{T\S2}(y)/Z_{T\S0}(y)) * r2`.

Consider revising the aforementioned comments to improve consistency and more accurately reflect the implemented logic. This will make it easier for auditors and other parties examining the code to understand what each section of the code is designed to do.

***Update:** Resolved in pull request #3 at commit 343d11c and at commit 3c09dfd. The team stated:*

> *The comments are now fixed.*

L-06 Lack of Validation of Proof Size - Phase 3

The `verify` function of the `FflonkYul` contract expects a `_proof` argument with 24 elements. However, the length of `_proof` is currently not validated. If the proof is too short, missing evaluations will be filled with 0s instead of reverting. If the proof is too long, additional elements will be ignored. While we did not find any concrete security risk associated with this behavior, passing a short proof currently reverts later on in the execution with an unclear "Precompute Eval. Error [PALBE]" error, whereas passing a long proof does not necessarily revert. A mismatch in proof size could potentially happen during the migration from the PLONK to the FFLONK verifier.

Consider adding validation for the length of the proof in order to reduce the attack surface and facilitate debugging in case of proof length mismatch.

***Update:** Resolved in pull request #5 at commit 85ce7df. The Matter Labs team stated:*

> *The length check for the proof has now been integrated.*

Notes & Additional Information

N-01 Some Memory Variables Could Be Constants – Phase 2

The variables defined in the verification key, as well as some transcript elements such as the domain size, the number one, and omega, could be defined as constants instead of being stored in memory. This would clarify the fact that they are indeed constants, as well as save a bit of gas during execution.

Consider defining the variables mentioned above as constants.

Update: *Resolved in pull request #5 at commit 620ac25. The Matter Labs team stated:*

> *The variables are now made constants, and the necessary changes are reflected in the code.*

N-02 Lack of SPDX License Identifier - Phase 1

The `FflonkYul.sol` file lacks an SPDX license identifier.

To avoid legal issues regarding copyright and to follow the best practices, consider adding SPDX license identifiers to files as suggested by the Solidity documentation.

Update: *Resolved in pull request #3 at commit 2726bc8. The team stated:*

> *The license has now been added.*

N-03 Lack of Security Contact - Phase 1

Providing a specific security contact (such as an email or ENS name) within a smart contract significantly simplifies the process for individuals to communicate if they identify a vulnerability in the code. This practice is beneficial as it permits the code owners to dictate the communication channel for vulnerability disclosure, eliminating the risk of miscommunication or failure to report due to a lack of knowledge on how to do so.

The `FflonkYul` contract does not have a security contact.

Consider adding a NatSpec comment containing a security contact to the contract definition. Using the `@custom:security-contact` convention is recommended as it has been adopted by the OpenZeppelin Wizard and the ethereum-lists.

Update: *Resolved in pull request #3 at commit 2726bc8. The team stated:*

> *The Security Contact has now been added with the recommended NatSpec convention.*

N-04 State Variable Visibility Not Explicitly Declared - Phase 1

Within `FflonkYul.sol`, there are state variables that lack an explicitly declared visibility:

- The `DST_0` state variable
- The `DST_1` state variable
- The `DST_CHALLENGE` state variable
- The `FR_MASK` state variable
- The `Q_MOD` state variable
- The `R_MOD` state variable
- The `NUM_ALL_POLYS` state variable

For improved code clarity, consider always explicitly declaring the visibility of variables, even when the default visibility matches the intended visibility. Note that if these variables are declared to be `internal`, they would not increase the gas costs of the verifier. However, if these variables are declared to be `public`, they would increase both the deployment and runtime gas costs.

***Update:** Resolved in pull request #3 at commit 2726bc8. The team stated:*

> *The concerned variables now explicitly define the visibility.*

N-05 The Codebase Cannot Be Compiled With the Default Settings - Phase 1

The `FflonkYul` contract has a pragma to be compiled with Solidity version `0.8.24`. However, the Foundry config of the project sets the solc version to `0.8.25` which is inconsistent and prevents the contract from being compiled. In addition, the `FflonkYul.t.sol` test imports `FflonkYul` from `"../src/fflonk/FflonkYul.sol"` which does not exist. The correct path should be `"../src/FflonkYul.sol"`.

Consider addressing the above issues to prevent the code from not compiling.

***Update:** Resolved in pull request #3 at commit 2726bc8. The team stated:*

> *The imports and solidity versions have now been fixed as per the project configurations.*

N-06 Invalid Reverts - Phase 1

The code defines a `revertWithMessage` function. This function can be called with a length and a string as arguments to revert with an `Error(string)` error. However, some parts of the code use the assembly `revert` despite being passed the same arguments (a length and a string message) as if the `revertWithMessage` function had been called instead. Yet, the Yul `revert` function expects a memory pointer and a length as parameters. This mismatch would cause the reverts to fail as the string is mistakenly interpreted as the length.

In addition, some of the lengths given for the error messages are inaccurate:

- The length of this error is 21, not 22.
- Similarly, the length of this error is 20, not 21.

Consider addressing the aforementioned issues to improve the ability of callers to successfully debug potential reverts.

Update: *Resolved in pull request #3 at commit 2346c17. The team stated:*

> *The revert function has been corrected with the right lengths for the error strings.*

N-07 Code Quality and Readability Suggestions - Phase 1

Throughout the `FflonkYul.sol` file, multiple code quality and readability issues were identified:

- The inputs to the `verify` function are defined as `uint`. Consider explicitly defining them as `uint256`.
- There are multiple instances of numbers being hardcoded in the code. For example, corresponding to ω_8, ω_4, as well as powers of roots of unity (e.g., ω_8^2). Consider either defining these numbers as constants or adding comments if defining new constants is not desired.
- The `verify` function could be defined as `view` to improve code clarity.
- The ">" operator should be replaced with ">=" in error messages associated with checks which ensure that field values are `< R_MOD` [1] [2]. This will help make the error messages more accurate.
- Similarly, the number of inputs is checked to be equal to 1. Otherwise, the code reverts with "Inputs should be atmost 1" [1] [2]. Consider changing the error message to be more explicit (e.g., "Inputs length should be 1").

- The naming convention of variables in the `update_transcript` function is inconsistent, mixing both camel and snake case. Consider rewriting the `newState0` and `newState1` variables in snake case for consistency.
- The `NUM_ALL_POLYS` constant is unused and could be removed.

Consider making the changes outlined above to improve code quality and readability for auditors, code maintainers, and developers.

Update: *Resolved in pull request #3 at commit 22be764. The team stated:*

> *The suggestions have been acknowledged and implemented*

N-08 Gas Optimizations - Phase 1

Throughout the audited code, multiple opportunities for gas optimizations were identified:

- The proof currently contains three extra openings: $T_0(z)$, $T_1(z)$, and $T_2(z)$. These openings are currently validated by checking that they match what is recomputed by the verifier [1] [2] [3]. As these values are recomputed anyway, there always exists a unique value that the prover can send for these and sending them in calldata costs an extra ~1500 gas. As such, they could be removed from the proof to save gas. Note that the FFLONK specifications do not send these openings, as only 15 field elements are sent.
- `OPS_Y_POWS` stores 17 values, ranging from `y^0` to `y^16`. These values are computed and saved in memory inside the `initialize_opening_state` function. However, not all of these values are used. While some powers less than 8 remain unused throughout the protocol, notably, all powers of `y` exceeding 8 are not used at all.
- Each addition to the transcript computes two rolling hashes and stores them in memory. Unless there is a strong cryptographic argument to do so, consider maintaining a single rolling hash instead of two in order to save gas.
- Length of loops loaded from calldata or computed with `add` or `sub` could be loaded once and stored on the stack. For example, these lengths [1] [2] [3] could be stored on the stack to save gas.
- The `compute_opening_points` function could be optimized to remove the loops by storing `mload(PVS_R)` on the stack and computing the other elements with it.

Consider addressing the above instances to save gas.

Update: *Resolved in pull request #3 at commit cbf09ba and at commit 3c09dfd. The team stated:*

> *The optimizations have now been addressed.*

N-09 Overwriting of the Free Memory Pointer - Phase 3

The `point_mul`, `point_add`, `point_sub`, `pairing_check`, and `modexp` functions overwrite the free memory pointer location at `0x40`. In addition, all of these functions except for `point_mul` also overwrite the zero slot at `0x60`. According to the Solidity documentation, the free memory pointer points to `0x80` initially to avoid overwriting these designated slots.

The free memory pointer currently remains unused in the codebase as elements are written to memory using the locations defined by the constants at the beginning of `FflonkYul`. However, leaving such memory slots dirty introduces a risk in case of code reformatting or if these generic functions using elliptic curve points and field elements were to be copied to other files.

Consider storing values starting at memory location `0x80` for the aforementioned functions, while keeping in mind not to overwrite the first constant-defined memory location. Currently, starting at memory location `0x80` should not cause any overwriting, but this should be a consideration for any future refactoring of the code. Alternatively, consider documenting these functions more thoroughly, warning developers that these functions do indeed overwrite the free memory pointer and possibly the zero slot as well.

***Update:** Resolved in pull request #5 at commit 8aaf8b7. The Matter Labs team stated:*

> *The concerned functions now store values starting from 0x80, and a comment has been added.*

N-10 Use of Magic Number - Phase 3

The domain size, `8388608`, is hardcoded as a magic number in line 253, line 300, and line 302.

In order to future-proof this codebase and prevent a scenario during refactoring where only some of these values are updated but not others, consider defining a constant called `DOMAIN_SIZE` and using this constant throughout the codebase.

***Update:** Resolved in pull request #5 at commit 19ca355. The Matter Labs team stated:*

> *The DOMAIN_SIZE is now a constant.*

N-11 Inconsistent Representation of Hex Values - Phase 3

Throughout the `FflonkYul.sol` file, the hex values are sometimes represented with 2 digits and at other times with 3 digits, even when one of the digits is superfluous. One example of this is in line 189, where both `0x24` and `0x024` are used.

Consider standardizing the representation of hex values to improve the consistency of the codebase.

Update: *Resolved in pull request #5 at commit 19ca355. The Matter Labs team stated:*

> *The hex representations are now made consistent.*

N-12 Missing or Misleading Documentation - Phase 3

Throughout `FflonkYul.sol`, multiple instances of missing or misleading comments were identified:

- The comment in line 905, which states "Generates a new challenge with (uint32(2) || state_0 || state_1 || challenge_counter)" could be updated to "Generates a new challenge with (uint32(2) || state_0 || state_1 || uint32(challenge_counter))" to reflect the type of the challenge counter.
- The name of the constant `MEM_PROOF_LAGRANGE_BASIS_INVERSES` could be updated to `MEM_PROOF_LAGRANGE_BASIS_NUMERATORS` or `MEM_PROOF_LAGRANGE_BASIS` as these memory locations are first used to store the numerators and later on to store the Lagrange basis, but are not used to store the inverses.
- The first comment in the loop of the `precompute_partial_lagrange_basis_evaluations` function should say "h*w_i" instead of "h".
- In the `update_transcript` function, the memory slots `0x200` to `0x203` are assumed to be clean and are not explicitly cleaned. While we did not find any security risk associated with the current usage of this function, consider documenting this assumption explicitly to reduce the risk of error in case of code reformatting.

Consider revising the aforementioned comments to improve consistency and more accurately reflect the implemented logic, making it easier for auditors and other parties examining the code to understand what each section of code is designed to do.

***Update:** Resolved in pull request #5 at commit 19ca355. All the points were addressed, and `MEM_PROOF_LAGRANGE_BASIS_INVERSES` was renamed to `MEM_PROOF_LAGRANGE_BASIS_EVALS`. The Matter Labs team stated:*

> *The suggestions are now integrated.*

N-13 Missing Documentation About Treatment of Point at Infinity - Phase 3

The point at infinity was decided not to be accepted as a valid input for the commitment or for the second operand of the point_sub function due to security concerns by the Matter Labs team. While this could, in theory, pose completeness issues, we determined, in agreement with the team, that the security risks outweigh the completeness concerns in this case. However, this decision has not been documented in the code.

Consider adding documentation around the checks which ensure that the points are on the curve and the `point_sub` function. This will help document the decision made and avoid any confusion.

***Update:** Resolved in pull request #5 at commit 19ca355. The Matter Labs team stated:*

> *The checks and the function now highlight our decision based on the security concern.*

N-14 PLONK `verify` Function Has `public` Visibility - Phase 3

The current PLONK verifier defines the `verify` function with a public visibility. However, this function writes to the free memory pointer slot in memory. Thus, internal calls to the `verify` function would result in an invalid memory layout for the caller.

Consider defining the `verify` function as `external` for improved clarity, as this function should not be called internally by other contracts.

***Update:** Acknowledged, will resolve. This will be resolved after migration. The Matter Labs team stated:*

> *Acknowledged. The visibility is set to `external` for the Fflonk verifier per the Verifier interface. Thus, the verifier will have the correct visibility during the migration process.*

Conclusion

The audited code replaces the current custom PLONK verifier with FFLONK. This new verifier aims to reduce the on-chain gas costs of verifying L2 blocks.

We commend the Matter Labs team on their effort to reduce gas costs but note the complexity of the code, as it implements a complicated protocol in assembly. One critical issue was found that could have resulted in funds from rollups using this verifier being drained, assuming the future `Executor` contract would function in the same way as the current one. This issue stemmed from assumptions related to the structure of the inputs which were not validated. We also made multiple recommendations to enhance the quality of the codebase and to further reduce gas costs, many of which were addressed in later phases. In general, we recommend that for complex cryptographic changes such as these, a detailed spec be produced beforehand in order to reduce the possibility of error.

We thank the Matter Labs team for their responsiveness and for meeting with us regularly to explain the changes introduced by the update.