Espresso Turbo-PLONK verifier and BN254 Audit

Shresth Agrawal[1,2] Pyrros Chaidos[1,3]
Jakov Mitrovski[1,2]

[1] Common Prefix
[2] Technical University of Munich
[3] University of Athens

May 22, 2024
Last update: September 2, 2024

1 Overview

1.1 Introduction

Espresso Systems commissioned Common Prefix to audit their Solidity implementation of the Plonk verifier and its BN254 curve dependency.

Plonk [GWC19] is a state-of-the-art zero-knowledge proof system with a universal, updateable setup and an efficient verifier. Owing to its high efficiency and powerful arithmetization system, it is currently one of the most popular proof systems. The implementation used is intended to verify proofs produced by the Jellyfish PLONK ZKP cryptographic library. The codebase implements TurboPlonk, with 5 wires and additional gates customized for embedded curve operations and efficient arithmetic hashing. The codebase uses Yul and low-level Solidity to optimize the proof verification's gas cost.

Plonk requires a pairing-friendly group setting. For efficiency reasons, the implementation uses the BN254 curve, which Ethereum and other EVM chains have pre-compiled contracts for, making pairings and other curve operations practical. The deployment will be executed on an EVM chain, ensuring compatibility and leveraging the existing pre-compiled contracts. The codebase includes a wrapper of the pre-compile operations and additional helper functions for inversion, serialization, deserialization, validity, and more.

The primary objectives of the audit were to assess security, adherence to the relevant literature, performance optimizations, and code quality.

1.2 Audited Files

Audit start commits: [773cfae6, 4f2e93b]
Latest audited commits: [676053e, 5481965]

1. PlonkVerifier.sol
2. PolynomialEval.sol
3. Transcript.sol
4. BN254.sol
5. Utils.sol
6. IPlonkVerifier.sol

Supporting documentation:

1. Espresso's *Configurable Asset Privacy* specification is referred to as the *specification* document in the rest of this audit report.

1.3 Disclaimer

This audit does not give any warranties on the bug-free status of the given code, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. This audit report is intended to be used for discussion purposes only. We always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of the project.

The scope of the audit was limited exclusively to the Plonk verification and BN254 smart contracts, with no examination conducted on its associated dependencies. Furthermore, the audit does not encompass any reference string generation functionality in terms of code or execution. The reference string generation functionality involves creating the structured reference string (SRS), which is crucial for enabling both the prover and verifier to function efficiently and securely within the protocol.

1.4 Executive Summary

The audit was based on the verifier algorithm presented in Section 8.3 of the PLONK paper [GWC19]. As per the specification document, we augmented the verifier steps to extend to 5 wires and custom gates. The extended verifier algorithm and the translation of the paper's terms to the codebase structures and variable names can be found in Section 2.4.

Overall, the code is of good quality and adheres to best practices. The implementation is structured into modular functions that parallelize the

steps of the PLONK verifier. These functions are well documented and reference the relevant sections of the paper.

The main findings relate to edge cases of functions, deviations from the paper, and some operations that are curve-specific without being explicitly marked as such. The codebase seems to be based on some existing off-chain implementation. Some of the operations, such as bytes manipulation, memory expansion, etc., which are negligible off-chain, are gas-expensive on-chain. The codebase could be refactored to optimize these operations. We have also identified several lower-impact issues, including possible optimizations, code simplifications, inconsistencies, and minor deviations from best practices (namely, magic numbers and documentation lapses).

1.5 Findings Severity Breakdown

Our findings are classified under the following severity categories, according to their impact and their likelihood of leading to an attack.

Level	Description
High	Logical errors or implementation bugs that are easily exploited. In the case of contracts, such issues can lead to any kind of loss of funds.
Medium	Issues that may break the intended logic, are deviations from the specification, or can lead to DoS attacks.
Low	Issues harder to exploit (exploitable with low probability), can lead to poor performance, clumsy logic, or seriously error-prone implementation.
Informational	Advisory comments and recommendations that could help make the codebase clearer, more readable, and easier to maintain.

2 Findings

2.1 High

H01: Handle `evaluatePiPoly` and `evaluateLagrangeOne` for the case when ζ is a root of unity

Affected Code: PolynomialEval.sol (lines 145,115)

Summary: The current implementations of `evaluatePiPoly` and `evaluateLagrangeOne` produce erroneous results when ζ is a root of unity. Specifically, it always returns 0, whereas L_i should be returning 1 when ζ is ω^i.

Suggestion:

- For `evaluatePiPoly`, first check if ζ is a root of unity. This can be done by checking if the vanishing polynomial is 0 at ζ (this is already checked in line 151). If it is, then return the respective `pi[i]` such that $\zeta = \omega^i$. Otherwise, proceed with the calculation as is.
- For `evaluateLagrangeOne`, first check if ζ is a root of unity. This can be done by checking if the vanishing polynomial is 0 at ζ (this is already checked in line 120). Inside the if block, check if ζ is the first root of unity, i.e., $\zeta = \omega^1$ and return 1. Otherwise, return 0. The rest of the code can remain as is.

Status: Resolved [aba35f1]

2.2 Medium

M01: Use `addmod` for `_computeLinPolyConstantTerm`

Affected Code: PlonkVerifier.sol (lines 305,310,315,320)

Summary: The lines referenced above use `add` instead of `addmod`. Such calculations are implicitly performed modulo 2^{256}, instead of modulo the order of the scalar field, which is incorrect. In this particular instantiation, due to the size of the BN254 scalar field, the calculations will not overflow (i.e., for all previously reduced a and b, $a+b < 2^{256}$), and the addition will take place over the integers. Regardless, this behavior is specific to BN254 and not documented. Using the same code on a different curve would produce erroneous results.

Suggestion: Switch to using `addmod` throughout, or, alternatively, add appropriate documentation to indicate a BN254-specific optimization.

Status: Resolved [53a62f2]

M02: Start `evaluatePiPoly` and `evaluateLagrangeOne` enumeration from g^1 instead of g^0

Affected Code: PolynomialEval.sol (lines 145,115)

Summary: Currently, `evaluatePiPoly` and `evaluateLagrangeOne` start enumerating the group H from g^0, i.e., the coefficient for the public input is g^0 for the first, g^1 for the second, etc., instead of g^1 for the first, g^2 for the second, etc.

Suggestion: We suggest starting from g^1 for consistency with the paper, as this affects the alignment of the L_1 and PI polynomials.

Status: Acknowledged

M03: The `g1Deserialize` function allows deserialization of non-canonical points

Affected Code: BN254.sol (line 331)

Summary: As the field order is smaller than 2^{256}, there are multiple valid representations of the same field element. Traditionally, the smallest non-negative integer is used as the canonical representation. The `g1Deserialize` function currently does not validate whether the x-coordinate is canonical. For example, both 1 and 1+ `fieldOrder` are deserialized to the same point, $(1, 2)$. This can lead to unexpected behavior for higher-level applications that rely on the library for the uniqueness of the deserialize operation.

Suggestion: We suggest that the function reverts if the x-coordinate is not canonical.

Status: Resolved [cc33b1d, 70c8225]

M04: Transcript does not include common preprocessed input, or any SRS elements

Affected Code:

- Transcript.sol (line 81)
- IPlonkVerifier.sol (line 52)

Summary: The transcript currently does not include the entirety of the common preprocessed input (CPI), as specified in the Plonk paper. Rather, it includes the verification key, in its place. This is partly acceptable as the elements of the verification key serve as commitments to the elements of the CPI.

However, the `VerifyingKey` structure lacks any representative of the SRS, introducing an unexpected and undesired degree of freedom. In the paper, the $[x]_2$ is part of the verifier key, whereas in the code, it is represented as a "magic" value.

Suggestion: The $[x]_2$ value should be included in the `VerifyingKey` structure.
Status: Resolved [8e9790e, 1ce3efd]

M05: The `invert` function allows inversion of zero

Affected Code: BN254.sol (line 188)
Summary: The current implementation of the `invert` function allows the input 0 to be processed. This is incorrect as 0 does not have a multiplicative inverse in the field.
Suggestion: We recommend that the `invert` function reverts gracefully when the input is 0.
Status: Resolved [62b1d9c]

2.3 Low

L01: Inefficient handling of public inputs

Affected Code:
- PlonkVerifier.sol (line 93)
- Transcript.sol (line 153)

Summary: Currently, the functions `verify` and `appendVkAndPubInput` use a dynamically sized array for `publicInput`, but only validate and utilize the first eight public inputs. Sending an array with fewer than eight elements will revert the transaction (with an 'index out of bounds' error) while sending an array with more than eight elements will result in some public inputs not being validated and used. Additionally, using a dynamic length array incurs an additional gas cost to store the length of the array in memory.
Suggestion: We suggest implementing one of the following approaches:
- If the number of public inputs is fixed at eight, consider using a fixed-size array as input to the functions to avoid unnecessary gas costs.
- If the number of public inputs can vary, ensure dynamic input validation and addition of the input to the transcript to guarantee that all inputs are properly validated and used.

Status: Resolved [7b9c964, d40c9f7]

L02: Ambiguity of canonical arguments assumption

Affected Code: BN254.sol (lines 100,108,113)

Summary: As the field order is smaller than 2^{256}, there are multiple valid representations of the same field element. Traditionally, the smallest non-negative integer is used as the canonical representation; however, other representations exist. Adding any integer multiple of the `R_MOD` to the canonical representation produces a value that behaves identically modulo `R_MOD`, but not in the integers or when treated as a byte string. It is unclear if the library consistently treats arguments as canonical across all functions. While this does not cause issues when used by the PLONK verifier contract, it could lead to potential problems in other contexts.

Suggestion: We recommend that all functions (except `validateG1Point` and `validateScalarField`) that take a typed field element as an argument should assume that the representation is canonical and ensure that any field elements returned are also canonically represented. We have referenced above the functions that should be updated to reflect this assumption.

Status: Resolved [3547dfa, 70c8225]

L03: Challenge generation logic deviates from the protocol specification

Affected Code: Transcript.sol (line 45)

Summary: The transcript is a log of public inputs, verifying key, and all the messages exchanged between the prover and the verifier. According to the PLONK paper and the Fiat Shamir heuristic, challenges are generated by hashing the current state of the transcript.

However, the current implementation prepends the hash of the current transcript to the transcript itself and then hashes that to generate the new challenge. This additional step deviates from the protocol specification. Additionally, the function `getAndAppendChallenge` copies both the hash and the transcript to a new memory location each time it is called. This practice is inefficient and unnecessary.

Suggestion: We suggest directly hashing the transcript to align with the standard protocol specifications. Additionally, when generating multiple challenges for the same round (such as β and γ), concatenate an index to the transcript and hash the result. For example:

$$\beta = \mathcal{H}(\text{transcript}, 0)$$

$$\gamma = \mathcal{H}(\text{transcript}, 1)$$

Status: Resolved [a0e382d, 1ce3efd]

L04: Type safety issues

Summary: The code frequently wraps and unwraps library types to perform basic operations, which reduces readability and increases the risk of errors. There are inconsistencies in the types of arguments representing the same value across different functions. For example, in the `evaluateLagrangeOne` function, the variable ζ is represented by a `ScalarField` type, whereas in the `evalDataGen` and `evaluatePiPoly` functions, ζ is represented by a `uint256`.

Suggestion: We recommend using library types consistently throughout the codebase. Every typed variable should be assumed to be valid and canonical whereas every typecast (wrap/unwrap) operation should be validated or explicitly documented safe. We also recommend utilizing type-safe library functions to reduce the typecasting. Keep in mind that there is a gas cost trade-off when performing an additional function call.

Status: Acknowledged

L05: Suboptimal verifier code due to missing MSM implementation

Affected Code:

- PlonkVerifier.sol
- BN254.sol (line 173)

Summary: In the verifier code, particularly within the `_preparePolyCommitments` function, arrays of bases and scalars are allocated and passed between functions with the expectation of a valid Multi-Scalar Multiplication (MSM) implementation. However, the `multiScalarMul` function implementation only performs the naive iterative scalar multiplication and addition. This results in the following issues:

- The allocation (PlonkVerifier.sol (lines 154,482)) and copying of arrays (PlonkVerifier.sol (line 488 - L493)) result in unnecessary memory overhead.
- Severe overhead in code complexity and gas cost is introduced in the `_batchVerifyOpeningProofs` function.

Suggestion: We recommend refactoring the code to immediately calculate the dot products and cumulative sums instead of passing around the scalars and bases arrays and then applying the naive MSM algorithm.

Status: Acknowledged

2.4 Informational

I01: Inefficient memory allocation in transcript generation logic

Affected Code:

- PlonkVerifier.sol (line 173)
- Transcript.sol (line 81)

Summary: The current implementation of the function `_computeChallenges` in `PlonkVerifier.sol` and `appendVkAndPubInput` in `Transcript.sol` inefficiently manages memory by repeatedly calling `abi.encodePacked` to concatenate elements to the transcript. Each call to `abi.encodePacked` results in a new memory allocation and copying of the existing transcript, leading to excessive gas consumption.

Suggestion: Create a struct containing all the fields that are appended to the transcript. Hash the parts of the struct using low-level memory access to generate the relevant challenges. To allow for tight packing of the transcript, one can also use a bytes array instead of a struct and low-level memory writes to populate the array.

Status: Resolved [1ce3efd, d414934]

I02: Redundant endianness reversal in transcript generation and serialization logic

Affected Code:

- Transcript.sol (lines 45,81)
- PlonkVerifier.sol (line 173)
- BN254.sol (lines 313,331)

Summary: The current implementation reverses the endianness of all entries appended to the transcript. This is done to ensure compatibility with previous off-chain implementations. However, this step is redundant for the correct functioning of the Fiat-Shamir heuristic.

Suggestion: We suggest removing the endianness reversal step from the transcript generation and point serialization/deserialization to simplify the code and reduce gas cost.

Status: Resolved [720ad31, 1ce3efd]

I03: Gas optimization of `evaluatePiPoly`

Affected Code: PolynomialEval.sol (line 145-L231)

Summary: Currently, the batch inversion method is used to minimize the number of necessary inversions. This requires computing the product $\prod_{j \neq i}(\zeta - g^j)$ for each i. The current implementation performs $n(n-1)$ multiplications, but can be optimized to only perform $3n$ multiplications.

Suggestion: We suggest optimizing the product computation by implementing the product-of-array-except-self algorithm. This can be done by precomputing prefix and suffix products of the array consisting of $(\zeta - g^j)$. Then, multiply `prefix[i]` and `suffix[i]` to obtain the product $\prod_{j \neq i}(\zeta - g^j)$. This approach reduces the number of multiplications to $3n$ while keeping the number of inversions to 1. For $n = 8$, this reduces the number of multiplications from 56 to 24, reducing gas cost by ∼10K.

Code Suggestion: EspressoSystems/espresso-sequencer/pull/1716

Status: Resolved [ca07365, 7b9c964]

I04: Redundant fields in the `EvalDomain` struct

Affected Code: PolynomialEval.sol (lines 17,21)

Summary: The `EvalDomain` struct contains separate fields for **size** and `logSize`, even though **size** can be derived from `logSize` using a single-bit shift operation. Additionally, the `groupGenInv` field is defined but not used anywhere in the code.

Suggestion: We recommend removing the **size** and `groupGenInv` fields from the `EvalDomain` struct.

Status: Resolved [4ddfd04, 9990e9a]

I05: Inefficient runtime calculation of domain elements and eval domain

Affected Code: PolynomialEval.sol (line 235-L258)

Summary: Currently, domain elements and eval domain are computed at runtime, which leads to unnecessary gas consumption. This approach is inefficient if the public input size and the domain size remain fixed.

Suggestion: For a given verifying key, we recommend computing the domain elements and eval domain in the constructor and storing them as `immutable` constants. Similar to the `getVk` function, we can add `getDomainElements` and `getEvalDomain` functions which load these constants into memory.

Status: Resolved [9068eea]

I06: Inefficient use of memory for read-only arguments

Affected Code: PlonkVerifier.sol (line 91)

Summary: The `verify` function takes three arguments all of which are read-only.

Suggestion: We recommend using `calldata` for the `proof`, `publicInput`, and `verifyingKey` arguments and making the necessary changes to the rest of the codebase to support this update.
Status: Acknowledged

I07: Inconsistent definition and usage of `COSET_Ki` constants

Affected Code:
- Transcript.sol (line 104-L119)
- PlonkVerifier.sol (line 28-L35)

Summary: The `COSET_Ki` constants are used across multiple contracts. In the `PlonkVerifier` contract, these are defined as constants, whereas in the `Transcript` contract, they appear as magic numbers. This can lead to difficulties in maintaining the codebase.
Suggestion: We recommend refactoring the code to reference a single source of truth for the `COSET_Ki` constants. This can be achieved by defining these constants in a common library or a shared contract.
Status: Resolved [2f40358]

I08: Incomplete test coverage of library functions

Affected Code:
- BN254.sol (lines 83,91,100,108,113,135)
- BN254.sol (lines 140,145,150,173,188,234)
- BN254.sol (lines 313,331)

Summary: Several functions (`infinity`, `isInfinity`, `negate`, `add`, `mul`, `multiScalarMul`, `invert`, `validateScalarField`, `g1Serialize`, and `g1Deserialize`) in the library lack sufficient test coverage.
Suggestion: Write unit tests for all the functions to ensure the library's correctness and ease of maintenance.
Status: Acknowledged

I09: Redundant success checks for `staticcall()`

Affected Code: BN254.sol (lines 126-L131,163-L168)
Summary: The success of the `staticcall()` function is verified twice in the referenced code segments, leading to an extra, redundant check.
Suggestion: We recommend consolidating the verification into a single success check.
Status: Resolved [1ead65b]

I10: Inconsistent return type in the `g1Serialize` function

Affected Code: BN254.sol (line 313)
Summary: The `g1Serialize` function currently returns a `bytes` type, which is (i) inconsistent with the input type expected by the `g1Deserialize` function and (ii) gas inefficient.
Suggestion: We recommend changing the return type of `g1Serialize` to `bytes32`.
Status: Resolved [08d2094, 4f3bd26]

I11: Non-standard way of deriving randomness for batch verification

Affected Code: PlonkVerifier.sol (line 545-L550)
Summary: Batch verification constructs a randomizer by hashing together the `u` challenges of the individual proofs. The usual practice is to hash the terms being batched together. However, each individual term is included in the hash calculation that produces the corresponding `u` value. Thus, the change does not impact the security argument.
Suggestion: Document that the terms are implicitly represented by `u`.
Status: Resolved [318af16]

I12: Deviation from positive or negative $\mathbb{G}_1$ point encoding conventions

Affected Code: BN254.sol (line 286)
Summary: The encoding of $\mathbb{G}_1$ points assigns higher values to the encoding of positive Y-coordinates and lower values to the encoding of negative Y-coordinates, which is unconventional.
Suggestion: We suggest the following two options:

- Update the documentation to include a clear definition of what "positive" means in the context of $\mathbb{G}_1$ point encoding and clarify that the current design choices were made to ensure compatibility with the Arkworks library [Lib].
- Alternatively, update the code to match the conventional practice of encoding positive points as small and negative points as big.

Status: Acknowledged

I13: Inconsistent `staticcall` offsets and gas subtraction

Affected Code: BN254.sol (lines 126,163)

Summary: The current implementation calls `staticcall` with input and return offsets that do not match the sizes expected by the precompiled contracts `ecAdd` and `ecMul`. Additionally, the subtraction of 2,000 gas units is unnecessary.

Suggestion: We recommend updating the input and return offsets to reflect the exact sizes of the input and return data. Additionally, remove the unnecessary gas subtraction.

Status: Resolved [421f2b9]

I14: Redundant code segments

Affected Code: PlonkVerifier.sol (lines 411,485,516)

Summary: The final update of `vBase` in the `_preparePolyCommitments` function is unnecessary. Additionally, in the `_verifyOpeningProofs` function, the `sumEvals` variable can be omitted since it is initialized to 0 and `pcsInfo.eval` is only added to it once.

Suggestion: Remove the redundant code segments. Consider directly using `pcsInfo.eval` instead of `sumEvals` in the `_verifyOpeningProofs` function.

Status: Resolved [f2169b8]

I15: Unused function `_batchVerifyOpeningProofs`

Affected Code: PlonkVerifier.sol (line 540)

Summary: The `_batchVerifyOpeningProofs` function is not used within the codebase and is currently defined as an internal function.

Suggestion: If the function is intended to be used, consider exposing this functionality through a public or external function. Otherwise, remove the function from the codebase.

Status: Resolved [318af16]

I16: Redundant function `evaluateLagrangeOne`

Affected Code: PolynomialEval.sol (line 115)

Summary: The function `evaluateLagrangeOne` computes the first Lagrange polynomial evaluation. However, this evaluation is also calculated as an intermediate result in the function `evaluatePiPoly`.

Suggestion: We recommend removing the `evaluateLagrangeOne` function and modifying `evaluatePiPoly` to return both the first Lagrange polynomial evaluation and the public input polynomial evaluation. This change would optimize gas usage and reduce code complexity.

Status: Acknowledged

I17: Inconsistent comment

Affected Code: PlonkVerifier.sol (line 769)
Summary: There is a discrepancy between the code and the comment referenced above. While the code correctly uses `verifyingKey.sigma4` ($[s_{\sigma_4}]_1$), the comment incorrectly refers to $[s_{\sigma_3}]_1$.
Suggestion: We suggest updating the comment to reflect the formula accurately.
Status: Resolved [5d41633, e9282be]

I18: Unhandled case in `multiScalarMul` function

Affected Code: BN254.sol (line 173)
Summary: The current implementation of the `multiScalarMul` function does not handle the scenario where empty arrays are passed as arguments.
Suggestion: Consider adding a check for empty arrays and reverting with an appropriate error message.
Status: Resolved [3fdaf5a]

I19: `g1Serialize` function optimization

Affected Code: BN254.sol (line 313)
Summary: In the case when infinity is passed to `g1Serialize` function, it first creates a bitmask and then ors it with the encoding of the x-coordinate.
Suggestion: As the encoding of infinity is unique, we suggest directly returning the serialized infinity point.
Status: Resolved [8ca1c70, 4f3bd26]

A Additional Artifacts

In this section, we provide a brief outline of the data structures used inside the verifier as well as an outline of the verification algorithm of [GWC19], as modified by the additional wires and gates used.

A.1 Verification Data Structures

[4]This is the scalar that's multiplied with $[1]_1$ to produce $[E]_1$.
[5]These are the scalars that produce $[D]_1$ via an MSM.
[6]These are the bases that produce $[D]_1$ via an MSM.

struct PlonkProof			struct VerifyingKey		
Plonk Ref	Variable	Offset	Plonk Ref	Variable	Offset
$[a]_1$	`G1Point wire0`	0x00	n	`uint256 domainSize`	0x00
$[b]_1$	`G1Point wire1`	0x20	fixed $l = 8$	`uint256 numInputs`	0x20
$[c]_1$	`G1Point wire2`	0x40	$[s_{\sigma_1}]_1$	`G1Point sigma0`	0x40
$[d]_1$	`G1Point wire3`	0x60	$[s_{\sigma_2}]_1$	`G1Point sigma1`	0x60
$[e]_1$	`G1Point wire4`	0x80	$[s_{\sigma_3}]_1$	`G1Point sigma2`	0x80
$[z]_1$	`G1Point prodPerm`	0xA0	$[s_{\sigma_4}]_1$	`G1Point sigma3`	0xA0
$[t_{lowest}]_1$	`G1Point split0`	0xC0	$[s_{\sigma_5}]_1$	`G1Point sigma4`	0xC0
$[t_{low}]_1$	`G1Point split1`	0xE0	$[q_1]_1$	`G1Point q1`	0xE0
$[t_{mid}]_1$	`G1Point split2`	0x100	$[q_2]_1$	`G1Point q2`	0x100
$[t_{hi}]_1$	`G1Point split3`	0x120	$[q_3]_1$	`G1Point q3`	0x120
$[t_{higher}]_1$	`G1Point split4`	0x140	$[q_4]_1$	`G1Point q4`	0x140
$[W_\zeta]_1$	`G1Point zeta`	0x160	$[q_{M12}]_1$	`G1Point qM12`	0x160
$[W_{\zeta\omega}]_1$	`G1Point zetaOmega`	0x180	$[q_{M13}]_1$	`G1Point qM34`	0x180
$\bar{a}$	`ScalarField wireEval0`	0x1A0	$[q_O]_1$	`G1Point qO`	0x1A0
$\bar{b}$	`ScalarField wireEval1`	0x1C0	$[q_C]_1$	`G1Point qC`	0x1C0
$\bar{c}$	`ScalarField wireEval2`	0x1E0	$[q_{H1}]_1$	`G1Point qH1`	0x1E0
$\bar{d}$	`ScalarField wireEval3`	0x200	$[q_{H2}]_1$	`G1Point qH2`	0x200
$\bar{e}$	`ScalarField wireEval4`	0x220	$[q_{H3}]_1$	`G1Point qH3`	0x220
$\bar{s}_{\sigma 1}$	`ScalarField sigmaEval0`	0x240	$[q_{H4}]_1$	`G1Point qH4`	0x240
$\bar{s}_{\sigma 2}$	`ScalarField sigmaEval1`	0x260	$[q_{Ecc}]_1$	`G1Point qEcc`	0x260
$\bar{s}_{\sigma 3}$	`ScalarField sigmaEval2`	0x280			
$\bar{s}_{\sigma 4}$	`ScalarField sigmaEval3`	0x2A0			
$\bar{z}_\omega$	`ScalarField prodPermZetaOmegaEval`	0x2C0			

Table 1: Structure of proofs and verification keys.

struct Challenges		
Plonk Ref	Variable	Offset
α	`uint256 alpha;`	0x00
α^2	`uint256 alpha2;`	0x20
α^3 (unused)	`uint256 alpha3;`	0x40
β	`uint256 beta;`	0x60
γ	`uint256 gamma;`	0x80
ζ	`uint256 zeta;`	0xA0
v	`uint256 v;`	0xC0
u	`uint256 u;`	0xE0

Table 2: Structure of challenges.

struct PcsInfo				
Variable	Offset	PcsInfo Ref	Plonk Ref	Description
`uint256 u;`	0x00	chal.u	u	a random combiner –challenge
`uint256 evalPoint;`	0x20	zeta	ζ	the point to be evaluated at –also from challenge
`uint256 nextEvalPoint;`	0x40	zetaOmega	$\zeta\omega$	the shifted point to be evaluated at
`uint256 eval;`	0x60	eval	E[4]	the polynomial evaluation value
`uint256[] commScalars;`	0x80	commScalars	D[5]	scalars of poly comm for MSM
`G1Point[] commBases;`	0xa0	commBases	D[6]	bases of poly comm for MSM
`G1Point openingProof;`	0xc0	proof.zeta	$[W_\zeta]_1$	proof of evaluations at point 'eval_point'
`G1Point shiftedOpeningProof;`	0xe0	proof.zetaOmega	$[W_{\zeta\omega}]_1$	proof of evaluations at point 'next_eval_point'

Table 3: Structure of PcsInfo.

A.2 Verification Algorithm

Adjusted from [GWC19] to account for wires, and custom gates.

1. Validate the 13 `G1Point`s of the proof: 5 wires, 1 `prodPerm`, 5 split values, 2 W values (total 13). Implemented in PlonkVerifier.sol (line 111-L138).
2. Validate the 10 `ScalarField`s of the proof. Currently: 5 wire scalars, 4 permutation scalars, `prodPermZetaOmegaEval`. Implemented in PlonkVerifier.sol (line 111-L138).
3. Validate the l elements of public input. (Fixed to $l = 8$). Implemented in PlonkVerifier.sol (line 98-L105).
4. Compute challenges $\beta, \gamma, \alpha, \zeta, v, u \in \mathbb{F}$. Implemented PlonkVerifier.sol (line 173-L248) in `_computeChallenges` and also Transcript.sol (line 9).
5. Compute zero polynomial evaluation $Z_H(\zeta) = \zeta^n - 1$. See PolynomialEval.sol (line 89-L111) for implementation.
6. Compute Lagrange polynomial evaluation $L_1(\zeta) = \frac{\omega(\zeta^n - 1)}{n(\zeta - \omega)}$. See PolynomialEval.sol (line 115-L142) for implementation.
7. Compute public input polynomial evaluation $PI(\zeta) = \sum_{i \in [l]} (w_i L_i(\zeta)))$. See PolynomialEval.sol (line 144-L231) for implementation.
8. Compute $r_0 := PI(\zeta) - L_1(\zeta)\alpha^2 - \alpha(\bar{a} + \beta\bar{s}_{\sigma 1} + \gamma)(\bar{b} + \beta\bar{s}_{\sigma 2} + \gamma)(\bar{c} + \beta\bar{s}_{\sigma 3} + \gamma)(\bar{d} + \beta\bar{s}_{\sigma 4} + \gamma)(\bar{e} + \gamma)\bar{z}_\omega$. See PlonkVerifier.sol (line 281-L337) under `_computeLinPolyConstantTerm` for implementation.
9. Compute $[D]_1$, line by line. See PlonkVerifier.sol (line 668-L964) under `_linearizationScalarsAndBases` for implementation.
   (a) $\bar{a}[q_1]_1 + \bar{b}[q_2]_1 + \bar{c}[q_3]_1 + \bar{d}[q_4]_1 + \bar{a}\bar{b}[q_{M12}]_1 + \bar{c}\bar{d}[q_{M34}]_1 - \bar{e}[q_O]_1 + [q_C]_1 + \bar{a}^5[q_{H1}]_1 + \bar{b}^5[q_{H2}]_1 + \bar{c}^5[q_{H3}]_1 + \bar{d}^5[q_{H4}]_1 + \bar{a}\bar{b}\bar{c}\bar{d}\bar{e}[q_{ecc}]_1$
   (b) $((\bar{a} + \beta\zeta + \gamma)(\bar{b} + \beta k_1\zeta + \gamma)(\bar{c} + \beta k_2\zeta + \gamma)$
   $(\bar{d} + \beta k_3\zeta + \gamma)(\bar{e} + \beta k_4\zeta + \gamma)\alpha + L_1(\zeta)\alpha^2 + u) \cdot [z]_1$
   (c)
   (d) $-(\bar{a} + \beta\bar{s}_{\sigma 1} + \gamma)(\bar{b} + \beta\bar{s}_{\sigma 2} + \gamma)(\bar{c} + \beta\bar{s}_{\sigma 3} + \gamma)(\bar{d} + \beta\bar{s}_{\sigma 4} + \gamma)\alpha\beta\bar{z}_\omega \cdot [s_{\sigma 5}]_1$
   (e) $-Z_H(\zeta) \cdot ([t_{lowest}]_1 + \zeta^{n+2}[t_{lo}]_1 + \zeta^{2n+4}[t_{mid}]_1 + \zeta^{3n+6}[t_{hi}]_1 +$
   $+\zeta^{4n+8}[t_{highest}]_1)$
10. Compute $[F]_1 := [D]_1 + v \cdot [a]_1 + v^2 \cdot [b]_1 + v^3 \cdot [c]_1 + v^4 \cdot [d]_1 + v^5 \cdot [e]_1 + v^6[s_{\sigma 1}] + v^7[s_{\sigma 2}] + v^8[s_{\sigma 3}] + v^9[s_{\sigma 4}]$. See PlonkVerifier.sol (line 339-L417) under `_preparePolyCommitments` for implementation.
11. Compute $[E]_1 := (-r_0 + v\bar{a} + v^2\bar{b} + v^3\bar{c} + v^4\bar{d} + v^5\bar{e} + v^6\bar{s}_{\sigma_1} + v^7\bar{s}_{\sigma_2} + v^8\bar{s}_{\sigma_3} + v^9\bar{s}_{\sigma_4} + u\bar{z}_\omega) \cdot [1]_1$. See PlonkVerifier.sol (line 419-L445) under `_prepareEvaluations` for implementation.
12. $e([W_\zeta]_1 + u \cdot [W_{\zeta\omega}]_1, [x]_2) \stackrel{?}{=} e(\zeta \cdot [W_\zeta]_1 + u\zeta\omega \cdot [W_{\zeta\omega}]1 + [F]_1 - [E]_1, [1]_2)$. Implemented in PlonkVerifier.sol (line 447-L534).

A.3 Constant Sanity Checks

The following script checks that the subgroup and coset constants are well structured.

```
r=21888242871839275222246405745257275088548364400416034343698204186575808495617


d16=[16,
65536,
0x30641e0e92bebef818268d663bcad6dbcfd6c0149170f6d7d350b1b1fa6c1001,
0x00eeb2cb5981ed45649abebde081dcff16c8601de4347e7dd1628ba2daac43b7,
0x0b5d56b77fe704e8e92338c0082f37e091126414c830e4c6922d5ac802d842d4]

d17=[17,
131072,
0x30643640b9f82f90e83b698e5ea6179c7c05542e859533b48b9953a2f5360801,
0x1bf82deba7d74902c3708cc6e70e61f30512eca95655210e276e5858ce8f58e5,
0x244cf010c43ca87237d8b00bf9dd50c4c01c7f086bd4e8c920e75251d96f0d22]

d18=[18,
262144,
0x30644259cd94e7dd5045d7a27013b7fcd21c9e3b7fa75222e7bda49b729b0401,
0x19ddbcaf3a8d46c15c0176fbb5b95e4dc57088ff13f4d1bd84c6bfa57dcdc0e0,
0x36853f083780e87f8d7c71d111119c57dbe118c22d5ad707a82317466c5174c]


d19=[19,
524288,
0x3064486657634403844b0eac78ca882cfd284341fcb0615a15cfcd17b14d8201,
0x2260e724844bca5251829353968e4915305258418357473a5c1d597f613f6cbd,
0x6e402c0a314fb67a15cf806664ae1b722dbc0efe66e6c81d98f9924ca535321]

d20=[20,
1048576,
0x30644b6c9c4a72169e4daa317d25f04512ae15c53b34e8f5acd8e155d0a6c101,
0x26125da10a0ed06327508aba06d1e303ac616632dbed349f53422da953337857,
0x100c332d2100895fab6473bc2c51bfca521f45cb3baca6260852a8fde26c91f3
]

d5=[5,32,
0x2ee12bff4a2813286a8dc388cd754d9a3ef2490635eba50cb9c2e5e750800001,
0x9c532c6306b93d29678200d47c0b2a99c18d51b838eeb1d3eed4c533bb512d0,
0x2724713603bfbd790aeaf3e7df25d8e7ef8f311334905b4d8c99980cf210979d
]

for d in [d5,d16,d17,d18,d19,d20]:
  print("Checking d=",d[0])
  print(2**d[0]==d[1]) #size vs logsize
```

```
  print(1==d[1]*d[2]%r) #inverse of size
  print(1==d[3]*d[4]%r) #inverse of gen
  print(d[3]==pow(d[4],-1,r)) #extra inverse
  print(1==pow(d[3],d[1],r)) #gen has at most claimed order
  print(1!=pow(d[3],d[1]-1,r)) #gen does not have half order (no
                                    factors other than 2)


#Coset uniqueness verification

twoadicity=0
r0=r-1
while (0==r0%2):
  r0//=2
  twoadicity+=1

print (twoadicity)


K=[0]*5
K[1]=0x2f8dd1f1a7583c42c4e12a44e110404c73ca6c94813f85835da4fb7bb \\
                                    1301d4a;
K[2]=0x1ee678a0470a75a6eaa8fe837060498ba828a3703b311d0f77f010424 \\
                                    afeb025;
K[3]=0x2042a587a90c187b0a087c03e29c968b950b1db26d5c82d666905a689 \\
                                    5790c0a;
K[4]=0x2e2b91456103698adf57b799969dea1c8f739da5d8d40dd3eb9222db7 \\
                                    c81e881;
Z=[pow(x,r-1,r) for x in K]
print(Z)
R=[pow(x,r-2,r) for x in K]
#R has inverses of elements, we check that no
#non-trivial product is in the subgroup
for i in range(4):
  for j in range(4):
    if i==j :continue
    print (1!=pow((K[i+1]*R[j+1])%r,2**twoadicity,r))


print
```

References

GWC19. Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Paper 2019/953, 2019. https://eprint.iacr.org/2019/953.

Lib. Arkworks BN254 Library. ark-bn254. https://docs.rs/ark-bn254/latest/ark_bn254/.

About Common Prefix

Common Prefix is a blockchain research, development, and consulting company consisting of scientists and engineers specializing in many aspects of blockchain science. We work with industry partners who are looking to advance the state-of-the-art in our field to help them analyze and design simple but rigorous protocols from first principles, with provable security in mind.

Our consulting and audits pertain to theoretical cryptographic protocol analyses as well as the pragmatic auditing of implementations in both core consensus technologies and application layer smart contracts.