Optimizing Backend Verification in zk-Rollup Architectures

Mehdi Beriane[1] and Muhammed Ali Bingol[2]

[1] Blockchain developer
mehdi@tokamak.network
[2] ZKP Researcher
muhammed@tokamak.network

Abstract. Zero-knowledge rollups represent a critical scaling solution for Ethereum, yet their practical deployment faces significant challenges in on-chain verification costs. This paper presents a comprehensive implementation of the Tokamak zkEVM verifier, specifically optimized for the BLS12-381 elliptic curve operations introduced by EIP-2537. We detail the complete verification architecture, from EVM-compatible data formatting for pairing checks, multi-scalar multiplication (MSM), and elliptic curve addition, to the non-interactive protocol design between prover and verifier.

Our key contribution lies in novel optimization techniques that substantially reduce on-chain verification costs. Through strategic polynomial aggregation and scalar factorization, we minimize $\mathbb{G}_1$ exponentiations from 40 to 31, achieving gas savings of 108,000 units per verification. Additionally, we introduce a dynamic barycentric interpolation method that replaces computationally intensive FFT operations, resulting in 92-95% gas reduction for sparse polynomial evaluations. We further present proof aggregation strategies that minimize precompile calls while maintaining the 128-bit security guarantees of BLS12-381.

Our implementation demonstrates that careful protocol design and mathematical optimizations can make zk-rollup verification economically viable on Ethereum. The techniques presented are compatible with the upcoming Pectra upgrade and provide a blueprint for efficient on-chain verification of complex zero-knowledge proofs. Experimental results show total gas costs reduced from 857,200 to 748,450 units for complete proof verification, making our approach practical for high-throughput rollup deployments.

1 Introduction

Ethereum is a decentralized blockchain platform that enables the execution of smart contracts (self-executing programs that run on a global network of nodes). At the heart of Ethereum's execution environment is the Ethereum Virtual Machine (EVM), a stack-based virtual machine that processes smart contract bytecode. When we refer to "on-chain" operations, we mean computations that are executed directly within smart contracts on the Ethereum blockchain, where every operation consumes "gas" (a unit of computational effort that users must pay for in Ethereum's native currency).

zk-SNARKs (Zero-Knowledge Succinct Non-interactive Arguments of Knowledge) have attracted growing attention for their role in enhancing blockchain scalability and enabling secure transactions. A zk-SNARK is a cryptographic protocol that enables a prover to convince a verifier of the truth of a statement involving private data without disclosing the underlying information.

Early constructions such as Pinocchio [1] and Groth16 [2] laid the groundwork by introducing efficient proof systems with succinct, verifiable outputs. Over time, advancements have focused on reducing trusted setup requirements, improving prover efficiency, and enabling recursive composition. These developments have made zk-SNARKs practical for real-world applications, particularly in privacy-preserving protocols and scalable rollups.

To optimize certain computationally intensive operations, the EVM includes precompiled contracts (or "precompiles") that are native implementations of specific functions that would be prohibitively expensive to execute in EVM bytecode. These precompiles are hardcoded into Ethereum client implementations and can be called like regular smart contracts but execute much more efficiently. Common precompiles include cryptographic operations such as SHA-256 hashing and elliptic curve arithmetic.

A zkEVM rollup relies on an on-chain verifier to validate node states by checking proofs generated by the prover. The verifier processes these proofs along with setup parameters and performs a bilinear pairing to ensure their correctness. For these operations, we use the BLS12-381 elliptic curve [3], chosen for its strong security guarantees. Given the computational complexity of these operations, EIP-2537 [4] precompiles are required for efficient on-chain execution, with the deployment expected in an ethereum recent update, Pectra [5].

Applying zk-SNARKs to Ethereum, referred to as zkEVM, requires on-chain verification of cryptographic proofs to ensure state validity. Each verification operation on Ethereum consumes gas, with costs determined by the computational complexity of the underlying operations.

Jang and Judd have proposed a zk-SNARK protocol [6], which has been applied to Ethereum to develop Tokamak zk-EVM [7], an EVM-native zero-knowledge system. A distinguishing feature of this design is its two-step compilation process, which avoids the circuit size explosion typically caused by the EVM's indeterministic and adaptive behavior. The first compiler decomposes deterministic logic into reusable subcircuits, while the second compiler assembles and wires them to represent a given EVM function. This modular approach allows the same function to be reused across inputs, significantly reducing circuit size compared to other zk-EVM systems [8,9].

In this paper, we address the challenge of implementing gas-efficient on-chain verification for Tokamak zkEVM proofs. Specifically, given a Tokamak zkEVM proof π consisting of group elements in $\mathbb{G}_1$ and field elements in $\mathbb{F}_p$, we seek to minimize the total verification cost for the Pectra update while maintaining the security guarantees of the BLS12-381 curve.

1.1 Related Work

ZkSync Era [8] modified the original PlonK protocol to enable polynomial aggregation at the protocol level. Their approach introduces custom gate constraints and recursive proof composition, allowing the verifier to aggregate multiple polynomial commitments into a single aggregated polynomial before the pairing check. This protocol-level modification achieves approximately 700,000 gas savings per batch. However, this approach requires altering the underlying SNARK construction and maintaining a modified proof system, which increases implementation complexity and potential security surface.

Polygon zkEVM [9] takes an algorithmic optimization approach by moving computationally intensive operations off-chain. Specifically, they compute Montgomery batch inverses off-chain and only verify the correctness on-chain using simple modulo multiplications. This eliminates the need for on-chain IFFT computations, saving approximately 200,000 gas units. While effective, this approach is limited to specific mathematical operations and requires additional off-chain infrastructure.

Our Approach: The Tokamak zkEVM faces a distinct challenge due to our use of the BLS12-381 curve and the Tokamak zkSNARK's unique structure with 19 $\mathbb{G}_1$ elements. Unlike zkSync's protocol modifications or Polygon's off-chain computations, we pursue mathematical optimizations within the verification algorithm itself.

1.2 Our contributions:

In this paper, we provide a detailed explanation of how we improved the efficiency of the Tokamak zk-SNARK between the prover and the on-chain verifier. Our contributions include:

1. First Implementation of Tokamak zk-SNARK: This work presents the first complete implementation of the Tokamak zk-SNARK protocol for on-chain verification. We provide a comprehensive implementation that demonstrates the practical feasibility of the Tokamak construction in real-world blockchain environments, bridging the gap between theoretical protocol design and production deployment.

2. Polynomial Aggregation and Scalar Factorization: By strategically aggregating certain polynomials and factorizing scalars, we reduced the number of $\mathbb{G}_1$ exponentiations from 40 to 31, resulting in gas savings of 108,000 units. This optimization restructures the computation of some polynomials to minimize elliptic curve operations while maintaining protocol soundness.

3. Barycentric Interpolation for Sparse Polynomial Evaluation: We replace computationally intensive FFT operations with a dynamic barycentric interpolation method for evaluating a specific polynomial at a random challenge point. This optimization achieves:

- 92-95% gas reduction for sparse polynomial evaluations (from 180,000 to 8,000-15,000 gas)
- 60-75% gas reduction even for dense cases
- Adaptive performance that automatically adjusts to input sparsity without circuit-specific hardcoding
- Elimination of 2,300 modular exponentiations required by traditional IFFT approaches

4. Complete Gas-Optimized Implementation: Our combined optimizations reduce total verification costs from 857,200 to 748,450 gas units, making zkEVM verification economically viable for high-throughput rollup deployments. The implementation is fully compatible with the upcoming Pectra upgrade and EIP-2537 precompiles.

For comparison, Polygon uses an off-chain batch inverse computation using Montgomery modular batch inversion [10]. ZkSync altered the original PlonK zk-SNARK with custom gate constraints and recursive proof composition, achieving 70,100 gas savings per batch. Our approach achieves greater savings through mathematical optimizations without modification of the protocol.

1.3 Organization:

The rest of this paper is organized as follows: In Section 2.2, we detail some background information about Elliptic curve on-chain operations as required by the EIP-2537 released during the Pectra upgrade. Section 4.2 introduces the Elliptic curve functions designed for the Rollup's on-chain verifier. It includes $\mathbb{G}_1$ scalar multiplication, $\mathbb{G}_1$ addition and $\mathbb{G}_1$ subtraction. Section 4 presents the verifier's backend integrated protocol and highlighting efforts put into optimizing the overall efficiency. Finally, Section 5 provides quotations related to the efficiency of the protocol.

2 Preliminaries

2.1 Notations

Our operations will be conducted within bilinear groups $\mathbb{G}_1$, $\mathbb{G}_2$ or $\mathbb{G}_T$ each of prime order p, together with respective generators G_1, G_2 and G_T. These groups are equipped with a non-degenerate bilinear pairing $e : G_1 \times G_2 \to G_T$ with $e(G_1, G_2) = G_T$. We write $\mathbb{G}_1$ and $\mathbb{G}_2$ additively, and $\mathbb{G}_T$ multiplicatively. For $k \in \mathbb{F}_p$, we denote $[K]_1 := k \cdot G_1$, $[K]_2 := k \cdot G_2$. We use the notation $\mathbb{G} := \mathbb{G}_1 \times \mathbb{G}_2$. Given an element $h \in \mathbb{G}$, we denote by h_1(or h_2) the G_1(or G_2) element of h. We denote $\mathbb{G}_1^*$, $\mathbb{G}_2^*$ the non-zero elements of $\mathbb{G}_1$, $\mathbb{G}_2$ and denote $\mathbb{G}^* := G_1^* \times G_2^*$.

Symbols & Notation	Description
$\mathbb{G}_1$, $\mathbb{G}_2$ or $\mathbb{G}_T$	bilinear groups
$\mathbb{Z}_{256}$	$\{0, .., 2^{256} - 1\}$ (EVM word)
$\mathbb{F}_r$	finite field of order r.
s_D	The number of subcircuits in a subcircuit library.
m_D	The total number of wires of all subcircuits in a subcircuit library ($l < m_D$).
l_D	The total number of input and output wires of all subcircuits in a subcircuit library ($l_D < m_D$).
s_{max}	The maximum number of subcircuit placements that a circuit can be composed of.
m_I	$l_D - l$: The number of interface wires of all subcircuits in a subcircuit library.

Table 1. Symbols & Notation Table

2.2 Precompiled operations in EIP-2537

The precompiles at addresses 0x0b, 0x0c, and 0x0f are designed to perform elliptic curve additions (G1ADD), elliptic curve scalar multiplications (G1MSM), and bilinear pairings, respectively, on the BLS12-381 curve. Unlike the BN254 curve, the BLS12-381 curve has larger field elements and group elements, which require more storage.

- BN254: Base field $\mathbb{F}_p$ where $p \approx 2^{254}$ (32 bytes), scalar field $\mathbb{F}_r$ where $r \approx 2^{254}$ (32 bytes)
- BLS12-381: Base field $\mathbb{F}_q$ where $q \approx 2^{381}$ (48 bytes), scalar field $\mathbb{F}_r$ where $r \approx 2^{255}$ (32 bytes)

This section describes how the data is appropriately formatted to be suitable for the usage of these precompiles. Indeed, the standard follows a specific format for inputs and outputs that must be carefully considered.

Data Format Definitions: For BLS12-381 operations, we define the encoding function: bytes64$(v) : \mathbb{F}_q \to \{0,1\}^{512}$ which converts a 48-byte (384-bit) field element $v \in \mathbb{F}_q$ into a 64-byte representation by padding with 16 leading zero bytes to fit the EVM's 256-bit word alignment. For elliptic curve points, we use the notation:

- (x_0, y_0): coordinates of a first input point $P_1 \in \mathbb{G}_1$
- (x_1, y_1): coordinates of a second input point $P_2 \in \mathbb{G}_1$
- (x_3, y_3): coordinates of a resulting point $P_3 = P_1 + P_2 \in \mathbb{G}_1$

EC Addition: The input consists of a concatenation of each coordinate (total of 256 bytes) arranged in a specific order. Each coordinate is represented as a 64-byte value and added as follows:
input = (bytes64(x_0) + bytes64(y_0) + bytes64(x_1) + bytes64(y_1))
output = (bytes64(x_3) + bytes64(y_3))
The gas price for performing a G1 addition is established at 375 gas units.

Scalar Multiplication: Similarly to G1ADD, precompile at 0x0c expects inputs and outputs to be converted/concatenated in a specific format to handle the larger field elements. Indeed, the precompile expects 160*k bytes as an input that is interpreted as byte concatenation of k slices each of them being a byte concatenation of encoding of G1 point (128 bytes) and encoding of a scalar value (32 bytes). The gas price defined for using this precompile is 12,000 gas units
input = (bytes64(x_0) + bytes64(y_0) + bytes32(s))
output = (bytes64(x_1) + bytes64(y_1))

Bilinear Pairing: BLS12-381 EC pairing is done through precompile at address 0x0f and takes 384*k bytes as an input that is interpreted as byte concatenation of k slices each of them being a byte concatenation of an encoded $\mathbb{G}_1$ point (128 bytes) and an encoded $\mathbb{G}_2$ point (256 bytes). The output is a scalar value ($\in \mathbb{F}_p$), either 0 or 1. The cost for using the pairing precompile is $32,600 \times k + 37,700$ with k being the number of pairings within the equation.

Therefore, the input account for a total of 1,152 bytes. The pairing can accept a relatively high number of inputs for a single statistical and the gas price depends on the number of slices. In this example, there are 3 slices of 384 bytes each and the number of gas units has been evaluated at 135,500 (32,600 x 3 + 37,700). An additional slice would have cost 32,600 gas units more.

3 Tokamak zk-SNARK

The Tokamak zkSNARK is a succinct non-interactive argument of knowledge that combines the efficiency of Groth16 with field-programmable circuit derivation to achieve both universality and reduced verifier preprocessing. This section formally defines the cryptographic primitive underlying our zkEVM verifier implementation.

3.1 Design Philosophy

Traditional zkSNARKs like Groth16 require a new trusted setup for each circuit, limiting their practical deployment. The Tokamak zkSNARK addresses this limitation by introducing a field-programmable circuit architecture. Instead of creating entirely new circuits, programs are constructed by placing and wiring together predefined subcircuits from a library, similar to how programmers compose functions from standard libraries.

3.2 Mathematical Structure and Components

Here, we formally define the mathematical structures that constitute the Tokamak zkSNARK system:

Common Reference String (CRS): The setup algorithm generates a structured reference string σ_V where:

$$\sigma_V := \big([\alpha]_2, [\alpha^2]_2, [\alpha^3]_2, [\alpha^4]_2, [\gamma]_2, [\delta]_2, [\eta]_2, [x]_2, [y]_2\big) \tag{1}$$

Preprocessed Commitments: For universal applicability, the verifier maintains preprocessed commitments to the subcircuit library:

$$\text{Preprocessed inputs} = [s^{(0)}(x,y)]_1, [s^{(1)}(x,y)]_1, [s^{(2)}(x,y)]_1, [L_i(y)K_i(y)]_1 \tag{2}$$

where $s^{(i)}(x,y)$ are setup polynomials defined by the trusted setup, with $s^{(2)}(x,y) = x$ as specified in the protocol.

Public Inputs: The public inputs consist of:

$$\text{Public inputs} = (a_0, a_1, \ldots, a_{l-1}) \in \mathbb{F}_p^l \tag{3}$$

where l represents the number of public input/output wires, and these values are encoded into polynomials $A_{pub} = \sum_{j=0}^{l-1} a_j M_j(\chi)$ during verification.

Proof Structure: A complete Tokamak proof π consists of:

$$\begin{aligned}\pi = \big(&[U]1, [V]1, [W]1, [QA, X]1, [QA, Y]1, Vx, y,\\ &[B]1, [R]1, [QC, X]1, [QC, Y]1, [\Pi\chi]1, [\Pi\zeta]1,\\ &Ry, z, R'y, z, R''y, z, [M\zeta]1, [M\chi]1, [N\zeta]1, [N\chi]1,\\ &[Omid]1, [Oprv]1, [Opub]_1, [A]_1\big)\end{aligned} \tag{4}$$

These elements are organized into three interconnected components that work together to prove the statement's validity:

- **Arithmetic component** (first 6 elements): Proves that the computation was executed correctly according to the circuit logic.
- **Copy component** (next 13 elements): Ensures that values are consistently propagated between connected subcircuits.
- **Binding component** (final 4 elements): Guarantees that both components reference the same underlying data.

3.3 Subcircuit Library Architecture

The Tokamak zkSNARK operates on a subcircuit library $\mathcal{L}$ that contains reusable components for circuit construction. The system is parameterized by several key values that define the structure and constraints of this architecture:

- s_D: The number of subcircuits in the subcircuit library $\mathcal{L}$. This represents the total variety of computational building blocks available for circuit construction.
- m_D: The total number of wires across all subcircuits in the library. This includes all internal wires within each subcircuit as well as input/output interfaces.
- l_D: The total number of input and output wires across all subcircuits in the library ($l_D < m_D$). These are the "interface wires" that allow subcircuits to connect to each other.
- s_{max}: The maximum number of subcircuit placements that can compose a derived circuit. This parameter bounds the complexity of programs that can be verified using the system.
- m_I: The number of interface wires, calculated as $m_I = l_D - l$, where l represents the public input/output wires. These are the connecting wires used for data transfer between subcircuit instances.

3.4 Verification process

The verifier checks the proof validity through a single pairing equation that aggregates all proof components. This verification requires:

- 8 random challenges generated via Fiat-Shamir transformation
- 10 pairing operations (compared to 3 in standard Groth16)
- Linear combinations of proof elements weighted by the challenges

The additional complexity compared to Groth16 is the price paid for universality (the ability to verify different programs without changing the underlying cryptographic setup).

Verification algorithm: The complete verification algorithm is formally defined as follows:

Algorithm 1 Tokamak zkSNARK Verification

function VERIFY(σ, π, a) ▷ σ: CRS, π: proof, a: public inputs
  Load proof elements from π (19 $\mathbb{G}_1$ + 4 $\mathbb{F}_p$ elements)
  Generate challenges: $(\theta_0, \theta_1, \theta_2, \kappa_0, \chi, \zeta, \kappa_1, \kappa_2) \leftarrow \text{FiatShamir}(\pi)$
  Compute derived polynomials: $[F]_1, [G]_1, t_n(\chi), t_{s_{max}}(\zeta), K_0(\chi)$
  Barycentric interpolation: $A_{pub} \leftarrow \text{BarycentricEval}(a, \chi)$
  Polynomial aggregation: Compute $[LHS_A]_1, [LHS_B]_1, [LHS_C]_1$ using optimization factors
  Aggregate commitments: $[LHS]_1 \leftarrow [LHS_B]_1 + \kappa_2([LHS_A]_1 + [LHS_C]_1)$
  Auxiliary terms: $[AUX]_1 \leftarrow$ linear combination of $[\Pi_\chi]_1, [\Pi_\zeta]_1, [M_\chi]_1, [M_\zeta]_1, [N_\chi]_1, [N_\zeta]_1$
  Final pairing: return $\text{PairingCheck}([LHS]_1 + [AUX]_1, \text{other proof elements})$
end function

The algorithm implements the optimized verification process where polynomial aggregation reduces $\mathbb{G}1$ operations from 40 to 31, and barycentric interpolation efficiently computes *Apub* for sparse inputs.

4 Verifier's Backend Integrated Protocol

The Tokamak zkSNARK on-chain verifier implements the verification logic for the backend protocol, which governs the prover-verifier interaction. This component includes the following stages (i) loading the proof into storage variables, (ii) computing random challenges, and (iii) executing final bilinear pairing. For full protocol specifications, refer to the accompanying paper [6].

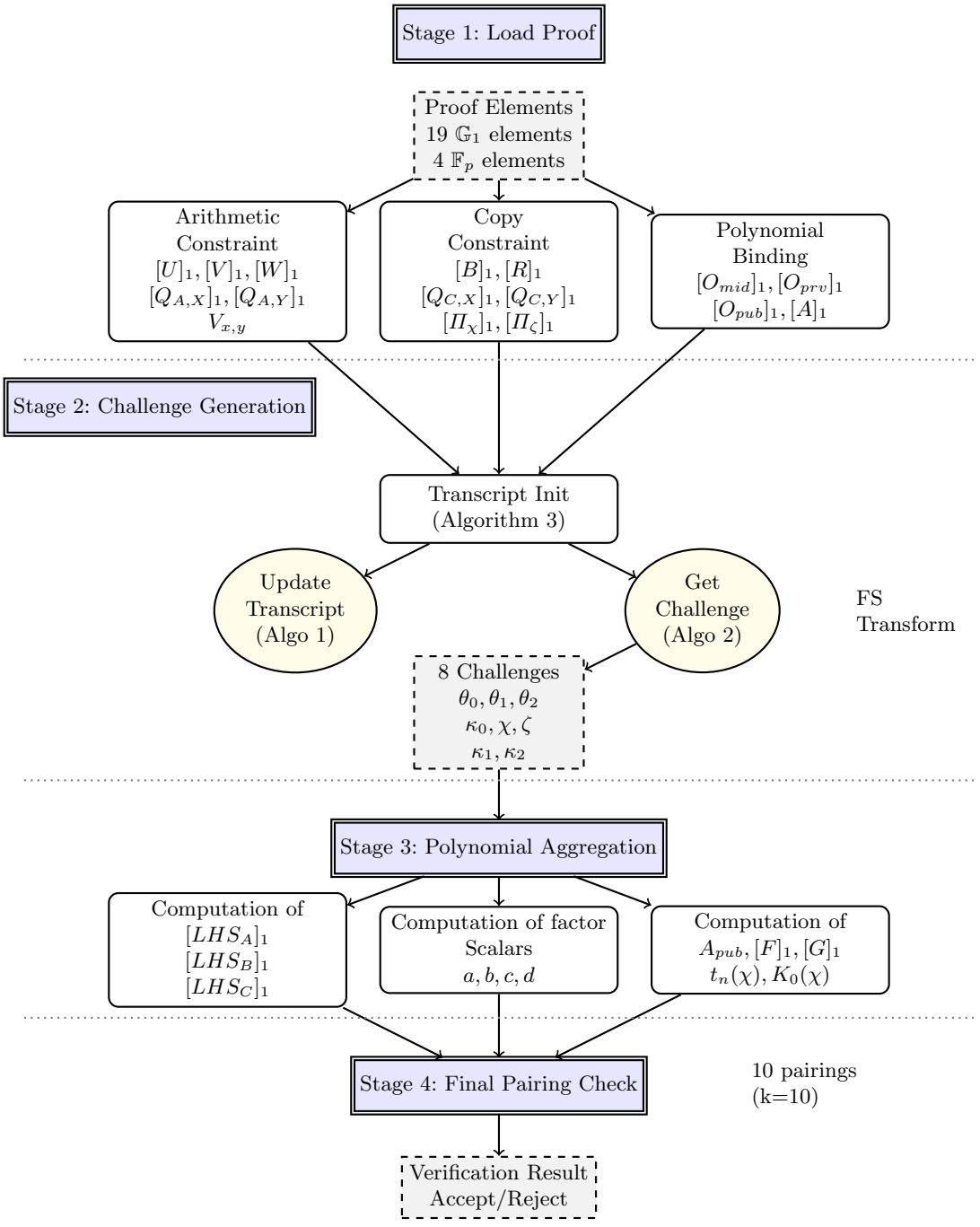

Fig. 1. Complete verification protocol flow showing the four main stages: (1) proof loading and parsing, (2) challenge generation via Fiat-Shamir transformation, (3) polynomial aggregation and optimization, and (4) final pairing verification. The optimizations in Stage 3 reduce $\mathbb{G}_1$ exponentiations from 40 to 31, saving 108,000 gas units.

4.1 Problem Formulation

Given a zkEVM proof consisting of 19 $\mathbb{G}_1$ elements and 4 $\mathbb{F}_p$ elements, along with fixed protocol parameters (s_D,m_D,l_D,s_{max},m_I) determined by the trusted setup, we aim to minimize the total gas cost $C_{total} = n_{MSM} \cdot 12,000 + n_{ADD} \cdot 375 + C_{pairing}$, where n_{MSM} and n_{ADD} represent the number of multi-scalar multiplications and elliptic curve additions respectively, and $C_{pairing} = 32,600k + 37,700$ for k pairing operations. Throughout this paper, we assume the availability of EIP-2537 precompiles (addresses 0x0b, 0x0c, 0x0f) and work within the constraints of the EVM's 256-bit word size for storing 384-bit BLS12-381 field elements. The challenge is to achieve this optimization without compromising the soundness of the verification protocol or requiring modifications to the underlying SNARK construction, thereby enabling economically viable zkEVM verification that can process high-throughput rollup batches within Ethereum's block gas limits.

4.2 Tokamak zk-SNARK EC operations implementation

Let $\mathbb{Z}_{256}$ be an integer subset such that $\mathbb{Z}_{256} := \{0, .., 2^{256} - 1\}$ (EVM word).
Let $\mathbb{F}_q$ be a finite field of order q (BLS12-381 field).
Let $\mathbb{G}_1 \subset (\mathbb{Z}_{256})^4$ be a group over $\mathbb{F}_q$.

The verifier is implemented using Solidity's inline assembly. As a result, each function has been adapted accordingly.

- $\mathbb{G}_1$ point multiplication: Takes one $\mathbb{G}_1$ point and one $\mathbb{Z}_{256}$ element as input. The function stores in memory the $\mathbb{G}_1$ point (each $\mathbb{Z}_{256}$ variable is stored into a single slot). the scalar is stored in memory at slot 0x80. The total accumulation of the memory is 160 bytes serves as an input for the precompile at address **0x0c** which returns the result ($\in \mathbb{G}_1$).
- $\mathbb{G}_1$ addition: takes two $\mathbb{G}_1$ points as input. The function stores in memory both points into 8 different slots. The total accumulation of the memory is 256 bytes serves as an input for the precompile at address **0x0b** which returns the result ($\in \mathbb{G}_1$).
- $\mathbb{G}_1$ subtraction: Our zkSNARK requires performing a $\mathbb{G}_1$ point subtractions. This operation is more complex because it involves coordinate negation under the base field modulus ($\in \mathbb{F}_q$), which is truncated into two slots. To perform subtraction on two $\mathbb{F}_q$ elements stored in Solidity, we perform subtraction on each slot separately and handle any potential borrow between the slots. This computation is performed on the y coordinates of each point before performing the same operation as for the $\mathbb{G}_1$ addition (accumulation of 256 bytes).

We assume that each $\mathbb{G}_1$ is properly encoded according to our storage layout before invoking these functions. Specifically, each coordinate of a $\mathbb{G}_1$ point is decomposed into two 256-bit storage slots to accommodate the 384-bit field elements of BN254. The encoding follows a consistent pattern: coordinates are split such that the lower-order bits are stored in PART2, while the higher-order bits, padded with 32 leading zeros, are stored in PART1. This decomposition ensures compatibility with the EVM's 256-bit word size while maintaining the full precision required for elliptic curve operations. For a comprehensive discussion of this storage optimization and its implementation details, see Section 4.

4.3 Loading the proof

This function loads the zk-SNARK proof, ensures that it is properly formatted, and stores it in memory.

- The first step is to load the public inputs:

Public input is composed of preprocessed commitments $s^{(0)}, s^{(1)}, s^{(2)}, [L_i(y)K_i(y)]_1 \in \mathbb{G}_1^4$ and $a \in \mathbb{Z}_{256}^n$ where n is the number of wires (it can differ from one proof to another). Note that $s^{(2)}$ and $[L_i(y)K_i(y)]_1$ are defined by the trusted setup and are supposed to be constant. These components are hardcoded within

the contract.
Indeed, according to the protocol:

$$s^{(2)}(\omega^j_{M_I}, \omega^i_{smax}) = \omega^j_{M_I} \Leftrightarrow s^{(2)}(X, Y) := X \tag{5}$$

where X is defined by the trusted setup.

– The second step loads the proof:

The proof is composed of 19 and 4 elements in $\mathbb{G}_1$ and $\mathbb{F}_p$ respectively :

1. Arithmetic constraint argument: $[U]_1$, $[V]_1$, $[W]_1$, $[Q_{A,X}]_1$, $[Q_{A,Y}]_1$, $V_{x,y}$
2. Copy constraint argument: $[B]_1$, $[R]_1$, $[Q_{C,X}]_1$, $[Q_{C,Y}]_1$, $[\Pi_\chi]_1$, $[\Pi_\zeta]_1$, $R_{y,z}$, $R'_{y,z}$, $R"_{y,z}$, $[M_\zeta]_1$, $[M_\chi]_1$, $[N_\zeta]_1$, $[N_\chi]_1$
3. Polynomial binding argument: $[O_{mid}]_1$, $[O_{prv}]_1$, $[O_{pub}]_1$, $[A]_1$,

Optimization: A key gas optimization in our zkEVM verifier restructures how proof data is passed to the contract. The standard approach passes the proof as a single uint256[] array, where each BLS12-381 coordinate (48 bytes) is split across two 32-byte words with 16 bytes of zero-padding in the first word. Our optimized version uses two arrays: uint128[] for the first 16 bytes and uint256[] for the remaining 32 bytes of each coordinate. In practice, this optimization saves approximately 40,000 gas - far more than just the calldata savings alone. This significant reduction comes from multiple factors: eliminating zero-padding in calldata, reducing memory operations during proof loading, and more efficient data access patterns in the assembly code. The optimization requires no algorithmic changes since the verifier already processes coordinates in two parts internally, making it a pure efficiency gain.

4.4 Challenge computation

To achieve non-interactive verification, the Fiat-Shamir (FS) transformation is used. This process is implemented through three interconnected algorithms that work together to generate the necessary challenges for proof verification. Algorithm 2 defines the core transcript update mechanism, which maintains the state of the Fiat-Shamir heuristic by progressively hashing proof elements. Algorithm 3 extracts deterministic challenges from the updated transcript state using domain separation tags. Finally, Algorithm 4 orchestrates the complete challenge generation process by systematically updating the transcript with all the proof components and extracting the eight required challenges $(\theta_0, \theta_1, \theta_2, \kappa_0, \chi, \zeta, \kappa_1, \kappa_2)$ in the correct sequence.

– Update Transcript:

Let $\mathbb{Z}_{256}$ be an integer subset such that $\mathbb{Z}_{256} := \{0, .., 2^{256} - 1\}$ (EVM word).
Let $\mathbb{Z}_8$ be an integer subset such that $\mathbb{Z}_8 := \{0, \cdots, 256 - 1\}$ (bytes).
Let $\mathbb{Z}_{800}$ be an integer subset such that $\mathbb{Z}_{800} := \{0, \cdots, 2^{800} - 1\}$.
Let $\mathbb{G}_1 \subset (\mathbb{Z}_{256})^4$ be a group over $\mathbb{F}_{\text{ıı}}$.
Let $c = a||b$ for $a, c \in \mathbb{Z}$ and $b \in \mathbb{Z}_{256}$ denote a concatenation of a, b such that $c := a2^{256} + b$.
Let BE : $a \in \mathbb{Z}_{800} \to \mathbf{b} \in (\mathbb{Z}_8)^{100}$, where $a = \sum_{i=0}^{99} a_i 2^{8i}$, such that $\mathbf{b} = (a_{99}, a_{98}, \cdots, a_0)$.
Let keccak256 : $(\mathbb{Z}_8)^{100} \to \mathbb{Z}_{256}$ denote a Keccak-256 hash function.

$$\text{Update}_T : (\mathbb{Z}_p)^3 \to (\mathbb{Z}_{256})^2 \tag{6}$$

Inputs: $(x, T_0, T_1) \in (\mathbb{Z}_p)^3$
Output: Updated states $\to (U_0, U_1) \in (\mathbb{Z}_p)^2$
Procedure:
Note that the on-chain verifier uses this function to store the output in memory layouts used as inputs for the same function.

Algorithm 2 Updating Transcripts

function UPDATE$_T(x, T_0, T_1)$ ▷ Where x is the input, T_0 and T_1 are transcripts to be updated
  $a \leftarrow \text{BE}((((0||T_0)||T_1)||x)))$
  $b \leftarrow \text{BE}((((1||T_0)||T_1)||x)))$
  $U_0 \leftarrow \text{keccak256}(a)$
  $U_1 \leftarrow \text{keccak256}(b)$
  return (U_0, U_1)
end function

– Get Challenge:

Let $\mathbb{Z}_{256}$ be an integer subset such that $\mathbb{Z}_{256} := \{0, .., 2^{256} - 1\}$ (EVM word).
Let $\mathbb{Z}_{32}$ be an integer subset such that $\mathbb{Z}_{32} := \{0, \cdots, 2^{32} - 1\}$ (32 bit words).
Let $\mathbb{F}_r$ be a finite field of order r (where r is a 254-bit prime).
Let $n \in \mathbb{Z}_{32}$.
Let $c = a||b$ for $a, c \in \mathbb{Z}$ and $b \in \mathbb{Z}_{256}$ denote a concatenation of a, b such that $c := a2^{256} + b$.
Let $\text{BE} : a \in \mathbb{Z}_{800} \mapsto \mathbf{b} \in (\mathbb{Z}_8)^{100}$, where $a = \sum_{i=0}^{99} a_i 2^{8i}$, such that $\mathbf{b} = (a_{99}, a_{98}, \cdots, a_0)$.
Let $\text{keccak256} : (\mathbb{Z}_8)^{100} \mapsto \mathbb{Z}_{256}$ denote a Keccak-256 hash function.

$$\text{GetChallenge} : \mathbb{Z}_{32} \times (\mathbb{Z}_{256})^2 \to \mathbb{F}_r \tag{7}$$

Inputs: $(n, T_0, T_1) \in \mathbb{Z}_{32} \times (\mathbb{Z}_{256})^2$
Output: $u \in \mathbb{F}_r$
Procedure:

Algorithm 3 Get challenge

function GETCHALLENGE(n, T_0, T_1) ▷ Where n is the challenge number, T_0 and T_1 are the final transcripts
  $N \leftarrow n << 224$
  $a \leftarrow \text{BE}((((2||T_0)||T_1)||N)))$
  $\tilde{u} \leftarrow \text{keccak256}(a)$
  $u \leftarrow \tilde{u} \wedge \text{FRMASK}$
  return u
end function

Our zk-SNARK involves the computation of 8 challenges generated successively by hashing specific polynomial commitments. Therefore, we have implemented the following InitializeTranscript function which updates the state of both transcripts defined above successively and generates challenges consecutively in accordance with the requirements.

– Transcript Initialization

$$\text{InitializeTranscript} : \mathbb{G}_1^9 \times \mathbb{Z}_{256}^3 \to \mathbb{F}_r^8 \tag{8}$$

Inputs: $([U]_1, [V]_1, [W]_1, [Q_{A,X}]_1, [Q_{A,Y}]_1, [B]_1, [R]_1, [Q_{C,X}]_1, [Q_{C,Y}]_1, V_{x,y}, R_{y,z}, R'_{y,z}, R''_{y,z}) \in \mathbb{G}_1^9 \times \mathbb{Z}_{256}^3$
Outputs: $(\theta_0, \theta_1, \theta_2, \kappa_0, \chi, \zeta, \kappa_1, \kappa_2) \in \mathbb{F}_r^8$
Procedure: The procedure involves calling the UpdateTranscript function multiple times, as well as the GetChallenge function once the necessary data has been hashed to obtain the target challenge. The full logic is described in Appendix B.

4.5 Final pairing

The final pairing operation requires aggregating all polynomial commitments that constitute the proof, along with additional parameters whose computation is detailed in the following section. This chapter presents our optimization approach for reducing the number of $\mathbb{G}_1$ exponentiations through strategic polynomial and field element factorization.

Polynomial aggregation: Based on zkSNARK protocol, the verifier accepts the transcript only if the following equation holds:

$$\begin{pmatrix} e([LHS]_1 + [AUX]_1, [1]_2)e([B]_1, [\alpha^4]_2 \\ e([U]_1, [\alpha]_2)e([V]_1, [\alpha^2]_2)e([W]_1, [\alpha^3]_2) \end{pmatrix} = \begin{pmatrix} e([O_{pub}]_1, [\gamma]_2)e([O_{mid}]_1, [\eta]_2)e([O_{prv}]_1, [\delta]_2) \\ e(\kappa_2[\Pi_\chi]_1 + \kappa_2^2[M_\chi]_1 + \kappa_2^3[N_\chi]_1, [x]_2) \\ e(\kappa_2[\Pi_\zeta]_1 + \kappa_2^2[M_\zeta]_1 + \kappa_2^3[N_\zeta]_1, [y]_2) \end{pmatrix} \tag{9}$$

where

$$\begin{cases} [LHS]_1 := [LHS_B]_1 + \kappa_2([LHS_A]_1 + [LHS_C]_1) \\ [AUX]_1 := \kappa_2\chi[\Pi_\chi]_1 + \kappa_2\zeta[\Pi_\zeta]_1 + \kappa_2^2\omega_{m_I}^{-1}\chi[M_\chi]_1 + \kappa_2^2\zeta[M_\zeta]_1 + \kappa_2^3\omega_{m_I}^{-1}\chi[N_\chi]_1 + \kappa_2^3\omega_{s_{max}}^{-1}\zeta[N_\zeta] \end{cases}$$

with

$$\begin{cases} [LHS_A]_1 := V_{x,y}[U]_1 - [W]_1 + \kappa_1([V]_1 - V_{x,y}[1]_1) - t_n(\chi)[Q_{A,X}]_1 - t_{s_{max}}(\zeta)[Q_{A,Y}]_1 \\ [LHS_C]_1 := \kappa_1^2\big((R_{x,y} - 1)[K_{-1}(x)L_{-1}(y)]_1 + \kappa_0(\chi - 1)(R_{x,y}[G]_1 - R'_{x,y}[F]_1) \\ \qquad\qquad + \kappa_0^2K_0(\chi)(R_{x,y}[G]_1 - R''_{x,y}[F]_1) - t_{m_l}(\chi)[Q_{C,X}]_1 - t_{s_{max}}(\zeta)[Q_{C,Y}]_1\big) \\ \qquad\qquad + \kappa_1^3([R]_1 - R_{x,y}[1]_1) + \kappa_2([R]_1 - R'_{x,y}[1]_1) + \kappa_2^2([R]_1 - R''_{x,y}[1]_1) \\ [LHS_B]_1 := (1 + \kappa_2\kappa_1^4)[A]_1 - \kappa_2\kappa_1^4A_{pub}[1]_1 \end{cases}$$

Optimization: Optimizing the number of $\mathbb{G}_1$ exponentiations relies on recomputing $[LHS_A]_1$, $[LHS_B]_1$ and $[LHS_C]_1$ as follows:

$$\begin{aligned} [LHS_A]_1 &:= V_{x,y}[U]_1 - [W]_1 + \kappa_1[V]_1 - t_n(\chi)[Q_{A,X}]_1 - t_{s_{max}}(\zeta)[Q_{A,Y}]_1 \\ [LHS_C]_1 &:= \kappa_1^2(R_{x,y} - 1)[K_{-1}(x)L_{-1}(y)]_1 + a[G]_1 - b[F]_1 - \kappa_1^2t_{m_i}(\chi)[Q_{C,X}]_1 \\ &\qquad - \kappa_1^2t_{s_{max}}(\zeta)[Q_{C,Y}]_1) + c[R]_1 + d[1]_1 \\ [LHS_B]_1 &:= (1 + \kappa_2\kappa_1^4)[A]_1 \end{aligned}$$

where

$$\begin{aligned} a &= \kappa_1^2\kappa_0R_{x,y}\big((\chi - 1) + \kappa_0K_0(\chi)\big) \\ b &= \kappa_1^2\kappa_0\big((\chi - 1)R'_{x,y} + \kappa_0K_0(\chi)R''_{x,y}\big) \\ c &= \kappa_1^3 + \kappa_2 + \kappa_2^2 \\ d &= -\kappa_1^3R_{x,y} - \kappa_2R'_{x,y} - \kappa_2^2R''_{x,y} - \kappa_1V_{x,y} - \kappa_1^4A_{pub} \end{aligned}$$

On-chain computations:

To construct the aggregated polynomial, the following 2 $\mathbb{G}_1$ and 4 $\mathbb{F}_p$ elements are computed on-chain: A_{pub}, $[F]_1$, $[G]_1$, $t_n(\chi)$, $t_{s_{max}}(\zeta)$, $t_{m_l}(\chi)$

- $[F]_1 := [B]_1 + \theta_0[s^{(0)}(x,y)]_1 + \theta_1[s^{(1)}(x,y)]_1 + \theta_2[1]_1$
- $[G]_1 := [B]_1 + \theta_0[s^{(2)}(x,y)]_1 + \theta_1[y]_1 + \theta_2[1]_1$
- $t_n(\chi) := \chi^n - 1$ where n is hardcoded

- $t_{s_{max}}(\zeta) := \zeta^{s_{max}} - 1$ where s_{max} is hardcoded
- $t_{m_I}(\chi) := \chi^{m_I} - 1$ where m_I is hardcoded
- $K_0(\chi) := \frac{\chi^{ml}-1}{m_I(\chi-1)}$
- $A_{pub} := \sum_{j=0}^{l-1} a_j M_j(\chi)$

$\{M_i\}_{i=0}^{l-1} \subset \mathbb{F}[X]$ is defined as a Lagrange basis polynomial such that $M_i(\omega_n^i) = 1$ and $M_k(\omega_n^i) = 0$ for every $i \neq k$ The general form of the Lagrange basis polynomial $M_j(X)$ is:

$$M_j(X) := \prod_{\substack{0 \leq m < l \\ m \neq j}} \frac{X - \omega_n^m}{\omega_n^j - \omega_n^m} \tag{10}$$

For roots of unity, this simplifies to the following:

$$M_j(X) := \frac{X^n - 1}{n(X - \omega_n^j)} \tag{11}$$

Therefore:

$$A_{pub} := \sum_{j=0}^{l-1} a_j \left(\frac{X^n - 1}{n(X - \omega_n^j)} \right) \tag{12}$$

ω_n is defined by the trusted-setup. Please refer the the MPC Ceremony protocol [11] for a detailed explanation of the process.

Using this optimization (barycentric interpolation instead of FFT), we avoid looping over modular exponentiations m times.

5 Analysis

The optimization strategies implemented in the Tokamak zkEVM verifier yield substantial improvements in gas consumption across multiple computational dimensions. Our analysis quantifies these gains by examining two primary optimization vectors: the reduction of elliptic curve operations through polynomial aggregation and scalar factorization, and the replacement of computationally intensive FFT operations with barycentric interpolation for sparse polynomial evaluation.

5.1 $\mathbb{G}_1$ exponentiations

This optimization reduced the required $\mathbb{G}_1$ exponentiations by 9, resulting in significant gas savings of 108,000 units.

	initial version	optimized version
number of G1 operations	76	65
number of G1MSMs	40	31
G1MSMs gas cost	480,000 $(40 \times 12,000)$	372,000 $(31 \times 12,000)$
number of G1ADDs	36	34
G1ADDs gas cost	13,500 (36×375)	12,750 (34×375)
number of pairings	1 $(k = 10)$	1 $(k = 10)$
pairings gas cost	$363,700(10 \times 32,600 + 37,700)$	$363,700(10 \times 32,600 + 37,700)$
Total EC op gas costs	**857,200**	**748,450**

5.2 Barycentric interpolation

In order to compute the binding polynomial evaluated at χ, we used the barycentric interpolation instead of FFT. This optimization is particularly effective for sparse polynomials, where only a small fraction of the evaluation points are non-zero.

Mathematical Foundation Given a polynomial $P(X)$ represented by its evaluations at the n-th roots of unity where $P(\omega^i) = a_i$ for $i = 0, 1, ..., n-1$ and ω is a primitive n-th root of unity, the polynomial can be evaluated at any point χ using the barycentric interpolation formula:

$$P(\chi) = \begin{cases} a_k & \text{if } \chi = \omega^k \text{ for some } k \\ \frac{\chi^n - 1}{n} \cdot \sum_{i \in \mathcal{I}} \frac{a_i \cdot \omega^i}{\chi - \omega^i} & \text{otherwise} \end{cases} \tag{13}$$

Where $\mathcal{I}$ represents the set of indices with non-zero values. Unlike traditional implementations that hardcode specific indices, our modular approach dynamically detects $\mathcal{I}$ at runtime, adapting to different circuit configurations.

Efficiency Analysis The traditional approach using Inverse Fast Fourier Transform (IFFT) followed by polynomial evaluation has a complexity of $O(n \log n)$ operations, where each operation involves modular exponentiation. For $n = 128$, this results in approximately 2,300 modular exponentiations.

In contrast, barycentric interpolation for sparse data requires only $O(|\mathcal{I}|)$ operations, where $|\mathcal{I}|$ is the number of non-zero values. The gas cost breakdown is as follows:

Operation	IFFT Method	Barycentric (Sparse)	Barycentric (Dense)
Modular exponentiations	∼2,300	∼35	∼200
CALLDATALOAD operations	128	$\|\mathcal{I}\|$	128
Total gas cost	180,000-200,000	8,000-15,000	50,000-80,000
Gas reduction (sparse case)	-	**92-95%**	**60-75%**

Modular Implementation Our implementation employs a dynamic approach that adapts to different sparsity patterns:

1. **Dynamic Non-Zero Detection**: The function iterates through all n evaluation points, identifying non-zero values at runtime rather than relying on hardcoded indices.
2. **Efficient Power Computation**: For computing ω^i, we use:
   - Precomputed powers for common values ($\omega^1, \omega^2, \omega^4, ..., \omega^{64}$)
   - Repeated multiplication for small exponents ($i < 16$)
   - Binary exponentiation for larger exponents
3. **Two-Pass Optimization**: For maximum efficiency with unknown sparsity:
   - First pass: Identify and store non-zero indices
   - Second pass: Process only the stored non-zero values
4. **Special Case Handling**: When $\chi = \omega^k$ for some k, the function directly returns a_k, avoiding unnecessary computation.

Adaptive Performance The modular implementation automatically adapts to the sparsity of the input:

- **Sparse inputs** (e.g., 16 non-zero out of 128): 10,000-15,000 gas
- **Medium density** (e.g., 64 non-zero values): 30,000-50,000 gas
- **Dense inputs** (most values non-zero): 50,000-80,000 gas

This approach ensures optimal performance regardless of the circuit configuration, eliminating the need for circuit-specific optimizations while maintaining a 60-95% gas reduction compared to the IFFT-based method.

6 Conclusion

This paper details the Tokamak zkEVM verifier's innovative on-chain mechanism for validating zkProofs without witness knowledge. The verification contract achieves this through specialized interactions with newly deployed precompiles handling BLS12-381 curve operations. We further analyze the system's gas optimization strategy, which strategically aggregates polynomials to bypass computationally expensive multiple $\mathbb{G}_1$ exponentiations, demonstrating significant efficiency gains. By highlighting these components, we provide a comprehensive overview of the design of the verifier and the efficiency improvements for scalable zk-proof verification on Ethereum. For more details on the implementation described below, please refer to the official repository [7].

References

1. Pinocchio: Nearly practical verifiable computation, 2013. `https://ieeexplore.ieee.org/document/6547113`.
2. Jens Groth. On the size of pairing-based non-interactive arguments. In *Proceedings, Part II, of the 35th Annual International Conference on Advances in Cryptology — EUROCRYPT 2016 - Volume 9666*, page 305–326, Berlin, Heidelberg, 2016. Springer-Verlag.
3. Bls signatures, 2022. `https://www.ietf.org/archive/id/draft-irtf-cfrg-bls-signature-05.html`.
4. Alex Vlasov, Kelly Olson, Alex Stokes, and Antonio Sanso. EIP-2537: Precompile for BLS12-381 curve operations [DRAFT]. `https://eips.ethereum.org/EIPS/eip-2537`, February 2020. Ethereum Improvement Proposals, no. 2537. [Online serial].
5. Pectra, 2025. `https://ethereum.org/en/roadmap/pectra/`.
6. Jehyuk Jang and Jamie Judd. An efficient SNARK for field-programmable and RAM circuits. Cryptology ePrint Archive, Paper 2024/507, 2024. `https://eprint.iacr.org/2024/507`.
7. Jason Hwang Jehyuk Jang and Mehdi Beriane. Tokamak zkevm official repository, 2025. `https://github.com/tokamak-network/Tokamak-zk-EVM/tree/main`.
8. Plonk unrolled for ethereum, 2020. `https://github.com/matter-labs/solidity_plonk_verifier/raw/recursive/bellman_vk_codegen_recursive/RecursivePlonkUnrolledForEthereum.pdf`.
9. An introduction to the polygon zkevm proving mechanism using pil-start, 2024. `https://github.com/0xPolygonHermez/zkevm-techdocs/blob/main/knowledge-layer/specs/PDFs/estark.pdf`.
10. Vitalik buterin. Starks, part 3: Into the weeds, 2018. `http://vitalik.eth.limo/general/2018/07/21/starks_part_3.html`.
11. Muhammed Ali Bingol. Multi-party Setup Ceremony for Generating Tokamak zk-SNARK Parameters. `https://eprint.iacr.org/2024/1671.pdf`, October 2024.

A EC addition, multiplication and pairing examples

– Example of a G1ADD operation:

```
{
    "Input": "0000000000000000000000000000000017f1d3a73197d7942695638c4fa9ac0fc3688c4f9774b905a14e
    ↪  3a3f171bac586c55e83ff97a1aeffb3af00adb22c6bb0000000000000000000000000000000008b3f481e3aaa0
    ↪  f1a09e30ed741d8ae4fcf5e095d5d00af600db18cb2c04b3edd03cc744a2888ae40caa232946c5e7e100000000
    ↪  0000000000000000000000112b98340eee2777cc3c14163dea3ec97977ac3dc5c70da32e6e87578f44912e90
    ↪  2ccef9efe28d4a78b8999dfbca9426000000000000000000000000000000000186b28d92356c4dfec4b5201ad09
    ↪  9dbdede3781f8998ddf929b4cd7756192185ca7b8f4ef7088f813270ac3d48868a21",
    "Name": "bls_g1add_g1+p1",
    "Expected": "0000000000000000000000000000000a40300ce2dec9888b60690e9a41d3004fda4886854573974
    ↪  fab73b046d3147ba5b7a5bde85279ffede1b45b3918d82d0000000000000000000000000000000006d3d887e9f
    ↪  53b9ec4eb6cedf5607226754b07c01ace7834f57f3e7315faefb739e59018e22c492006190fba4a870025",
    "Gas": 375,
},
```

– Example of a G1MSM operation input/output which is performing a scalar multiplication of a g1 point by 2 (g1 + g1 = 2 * g1):

```
{
    "Input": "0000000000000000000000000000000017f1d3a73197d7942695638c4fa9ac0fc3688c4f9774b905a14e
    ↪  3a3f171bac586c55e83ff97a1aeffb3af00adb22c6bb0000000000000000000000000000000008b3f481e3aaa0
    ↪  f1a09e30ed741d8ae4fcf5e095d5d00af600db18cb2c04b3edd03cc744a2888ae40caa232946c5e7e100000000
    ↪  0000000000000000000000000000000000000000000000000000000000000002",
    "Name": "bls_g1msm_(g1+g1=2*g1)",
    "Expected": "0000000000000000000000000000000000572cbea904d67468808c8eb50a9450c9721db30912801254
    ↪  3902d0ac358a62ae28f75bb8f1c7c42c39a8c5529bf0f4e0000000000000000000000000000000000166a9d8cabc
    ↪  673a322fda673779d8e3822ba3ecb8670e461f73bb9021d5fd76a4c56d9d4cd16bd1bba86881979749d28",
    "Gas": 12000
},
```

– Example a pairing equation execution:

$$e(g1, g2) * e(p1, g2) * e(p1 + g1, -g2) = 1 \tag{14}$$

```
{
{
    "Input": "0000000000000000000000000000000017f1d3a73197d7942695638c4fa9ac0fc3688c4f9774b905a14e
    ↪  3a3f171bac586c55e83ff97a1aeffb3af00adb22c6bb0000000000000000000000000000000008b3f481e3aaa0
    ↪  f1a09e30ed741d8ae4fcf5e095d5d00af600db18cb2c04b3edd03cc744a2888ae40caa232946c5e7e100000000
    ↪  0000000000000000000000000024aa2b2f08f0a91260805272dc51051c6e47ad4fa403b02b4510b647ae3d1770b
    ↪  ac0326a805bbefd48056c8c121bdb8000000000000000000000000000000013e02b6052719f607dacd3a08827
    ↪  4f65596bd0d09920b61ab5da61bbdc7f5049334cf11213945d57e5ac7d055d042b7e0000000000000000000000
    ↪  00000000000ce5d527727d6e118cc9cdc6da2e351aadfd9baa8cbdd3a76d429a695160d12c923ac9cc3baca289
    ↪  e193548608b828010000000000000000000000000000000000606c4a02ea734cc32acd2b02bc28b99cb3e287e85
    ↪  a763af267492ab572e99ab3f370d275cec1da1aaa9075ff05f79be0000000000000000000000000000000112b
    ↪  98340eee2777cc3c14163dea3ec97977ac3dc5c70da32e6e87578f44912e902ccef9efe28d4a78b8999dfbca94
    ↪  260000000000000000000000000000000186b28d92356c4dfec4b5201ad099dbdede3781f8998ddf929b4cd77
    ↪  56192185ca7b8f4ef7088f813270ac3d48868a210000000000000000000000000000000024aa2b2f08f0a9126
    ↪  0805272dc51051c6e47ad4fa403b02b4510b647ae3d1770bac0326a805bbefd48056c8c121bdb8000000000000
    ↪  00000000000000000013e02b6052719f607dacd3a088274f65596bd0d09920b61ab5da61bbdc7f5049334cf1
    ↪  1213945d57e5ac7d055d042b7e000000000000000000000000000000ce5d527727d6e118cc9cdc6da2e351a
    ↪  adfd9baa8cbdd3a76d429a695160d12c923ac9cc3baca289e193548608b828010000000000000000000000000000
    ↪  0000000606c4a02ea734cc32acd2b02bc28b99cb3e287e85a763af267492ab572e99ab3f370d275cec1da1aaa9
    ↪  075ff05f79be0000000000000000000000000000a40300ce2dec9888b60690e9a41d3004fda4886854573
    ↪  974fab73b046d3147ba5b7a5bde85279ffede1b45b3918d82d0000000000000000000000000000006d3d887
    ↪  e9f53b9ec4eb6cedf5607226754b07c01ace7834f57f3e7315faefb739e59018e22c492006190fba4a87002500
    ↪  0000000000000000000000000024aa2b2f08f0a91260805272dc51051c6e47ad4fa403b02b4510b647ae3
    ↪  d1770bac0326a805bbefd48056c8c121bdb8000000000000000000000000000013e02b6052719f607dacd3
    ↪  a088274f65596bd0d09920b61ab5da61bbdc7f5049334cf11213945d57e5ac7d055d042b7e0000000000000000
    ↪  0000000000000000d1b3cc2c7027888be51d9ef691d77bcb679afda66c73f17f9ee3837a55024f78c71363275
    ↪  a75d75d86bab79f74782aa0000000000000000000000000013fa4d4a0ad8b1ce186ed5061789213d9939
    ↪  23066dddaf1040bc3ff59f825c78df74f2d75467e25e0f55f8a00fa030ed",
    "Name": "bls_pairing_bilinearity_e(G1,G2)*e(P1,G2)*e(P1+G1,-G2)=1",
    "Expected": "0000000000000000000000000000000000000000000000000000000000000001",
    "Gas": 135500,
},
```

B Challenge Computation

Algorithm 4 Transcript Initialization

function TRANSCRIPTINIT($[U]_1, [V]_1, [W]_1, [Q_{A,X}]_1, [Q_{A,Y}]_1, [B]_1, [R]_1, [Q_{C,X}]_1, [Q_{C,Y}]_1, V_{x,y}, R_{y,z}, R'_{y,z}, R''_{y,z}$)
  $(u_0, u_1, u_2, u_3) \leftarrow \text{Parse}([U]_1)$ ▷ where $(u_0, u_1, u_2, u_3) \in (\mathbb{Z}_{256})^4$
  $(v_0, v_1, v_2, v_3) \leftarrow \text{Parse}([V]_1)$ ▷ where $(v_0, v_1, v_2, v_3) \in (\mathbb{Z}_{256})^4$
  $(w_0, w_1, w_2, w_3) \leftarrow \text{Parse}([W]_1)$ ▷ where $(w_0, w_1, w_2, w_3) \in (\mathbb{Z}_{256})^4$
  $(qax_0, qax_1, qax_2, qax_3) \leftarrow \text{Parse}([Q_{AX}]_1)$ ▷ where $(qax_0, qax_1, qax_2, qax_3) \in (\mathbb{Z}_{256})^4$
  $(qay_0, qay_1, qay_2, qay_3) \leftarrow \text{Parse}([Q_{AY}]_1)$ ▷ where $(qay_0, qay_1, qay_2, qay_3) \in (\mathbb{Z}_{256})^4$
  $(b_0, b_1, b_2, b_3) \leftarrow \text{Parse}([B]_1)$ ▷ where $(b_0, b_1, b_2, b_3) \in (\mathbb{Z}_{256})^4$
  $(r_0, r_1, r_2, r_3) \leftarrow \text{Parse}([R]_1)$ ▷ where $(r_0, r_1, r_2, r_3) \in (\mathbb{Z}_{256})^4$
  $(qcx_0, qcx_1, qcx_2, qacx_3) \leftarrow \text{Parse}([Q_{CX}]_1)$ ▷ where $(qcx_0, qcx_1, qcx_2, qacx_3) \in (\mathbb{Z}_{256})^4$
  $(qcy_0, qcy_1, qcy_2, qcy_3) \leftarrow \text{Parse}([Q_{CY}]_1)$ ▷ where $(qcy_0, qcy_1, qcy_2, qcy_3) \in (\mathbb{Z}_{256})^4$
  $T_1 \leftarrow \text{Update}_T(u_0, T_0)$
  $T_2 \leftarrow \text{Update}_T(u_1, T_1)$
  $T_3 \leftarrow \text{Update}_T(u_2, T_2)$
  $T_4 \leftarrow \text{Update}_T(u_3, T_3)$
  $T_5 \leftarrow \text{Update}_T(v_0, T_4)$
  $T_6 \leftarrow \text{Update}_T(v_1, T_5)$
  $T_7 \leftarrow \text{Update}_T(v_2, T_6)$
  $T_8 \leftarrow \text{Update}_T(v_3, T_7)$
  $T_9 \leftarrow \text{Update}_T(w_0, T_8)$
  $T_{10} \leftarrow \text{Update}_T(w_1, T_9)$
  $T_{11} \leftarrow \text{Update}_T(w_2, T_{10})$
  $T_{12} \leftarrow \text{Update}_T(w_3, T_{11})$
  $T_{13} \leftarrow \text{Update}_T(qax_0, T_{12})$
  $T_{14} \leftarrow \text{Update}_T(qax_1, T_{13})$
  $T_{15} \leftarrow \text{Update}_T(qax_2, T_{14})$
  $T_{16} \leftarrow \text{Update}_T(qax_3, T_{15})$
  $T_{17} \leftarrow \text{Update}_T(qay_0, T_{16})$
  $T_{18} \leftarrow \text{Update}_T(qay_1, T_{17})$
  $T_{19} \leftarrow \text{Update}_T(qay_2, T_{18})$
  $T_{20} \leftarrow \text{Update}_T(qay_3, T_{19})$
  $\theta_0 \leftarrow \text{GetChallenge}(0, T_{20})$.
  $\theta_1 \leftarrow \text{GetChallenge}(1, T_{20})$.
  $\theta_2 \leftarrow \text{GetChallenge}(2, T_{20})$.
  $T_{21} \leftarrow \text{Update}_T(r_0, T_{20})$
  $T_{22} \leftarrow \text{Update}_T(r_1, T_{21})$
  $T_{23} \leftarrow \text{Update}_T(r_2, T_{22})$
  $T_{24} \leftarrow \text{Update}_T(r_3, T_{23})$
  $\kappa_0 := \text{GetChallenge}(3, T_{24})$.
  $T_{25} \leftarrow \text{Update}_T(qcx_0, T_{24})$
  $T_{26} \leftarrow \text{Update}_T(qcx_1, T_{25})$
  $T_{27} \leftarrow \text{Update}_T(qcx_2, T_{26})$
  $T_{28} \leftarrow \text{Update}_T(qcx_3, T_{27})$
  $T_{29} \leftarrow \text{Update}_T(qcy_0, T_{28})$
  $T_{30} \leftarrow \text{Update}_T(qcy_1, T_{29})$
  $T_{31} \leftarrow \text{Update}_T(qcy_2, T_{30})$
  $T_{32} \leftarrow \text{Update}_T(qcy_3, T_{31})$
  $\chi := \text{GetChallenge}(4, T_{32})$.
  $\zeta := \text{GetChallenge}(5, T_{32})$.
  $T_{33} \leftarrow \text{Update}_T(V_{x,y}, T_{32})$
  $T_{34} \leftarrow \text{Update}_T(R_{y,z}, T_{33})$
  $T_{35} \leftarrow \text{Update}_T(R'_{y,z}, T_{34})$
  $T_{36} \leftarrow \text{Update}_T(R''_{y,z}, T_{35})$
  $\kappa_1 := \text{GetChallenge}(4, T_{36})$.
  $\kappa_2 := \text{GetChallenge}(5, T_{36})$.
  return $(\theta_0, \theta_1, \theta_2, \kappa_0, \chi, \zeta, \kappa 1, \kappa 2)$
end function