Linea Prover Documentation

Linea, Prover Team

Consensys, Linea

Abstract. Rollup technology today promises long-term solutions to the scalability of the blockchain. Among a thriving ecosystem, Consensys has launched the Linea zkEVM Rollup network for Ethereum. At a high level, the Ethereum blockchain can be seen as a state machine and its state transition can be arithmetized carefully. Linea's prover protocol uses this arithmetization, along with transactions on layer two in order to compute a cryptographic proof that the state transition is performed correctly.
The proof is then sent over to the Ethereum layer, where the smart contract (verifier contract) on Ethereum checks the proof and accepts the state transition if the proof is valid. The interaction between layer two and Ethereum is costly, which imposes substantial limitations on the proof size. Therefore, Linea's prover aims to compress the proof via cryptographic tools such as list polynomial commitments (LPCs), polynomial interactive oracle proofs (PIOPs), and Succinct Non-Interactive Arguments of Knowledge (SNARKs).
We introduce Wizard-IOP, a cryptographic tool for handling a wide class of queries (such as range checks, scalar products, permutations checks, etc.) needed to ensure the correctness of the executions of the state machines efficiently and conveniently. Another cryptographic tool is the Arcane compiler, which outputs standard PIOPs and is employed by Wizard-IOP to make different queries homogeneous. After applying Arcane, all the queries constitute evaluation queries over the polynomials. We then apply the Unique Evaluation compiler (UniEval), which receives the output of the Arcane and provides us with a PIOP that requires only a single evaluation check.
At this point, we employ Vortex, a list polynomial commitment (LPC) scheme to convert the resulting PIOP into an argument of knowledge. The argument of knowledge is then made succinct by applying different techniques such as self-recursion, standard recursion, and proof aggregations.

Keywords: Linea, zkEVM, SNARK, Ring-SIS, Self-Recursion, Arcane, Wizard-IOP, Range Checks, Lookup Proofs, Permutation Proofs.

1 Introduction

The advent of blockchain technology and its underlying cryptographic mechanisms have ushered in a new era of secure and transparent digital transactions. Among these, the Ethereum Virtual Machine (EVM) stands out as a cornerstone for executing smart contracts on the Ethereum blockchain. As part of this ecosystem, Linea launched a zk-EVM, a scalable EVM execution environment leveraging cryptographic techniques to achieve higher transaction processing capability than the main Ethereum network. This paper delves into the cryptographic applications that ensure the integrity of Linea, specifically focusing on advancements in zero-knowledge proofs and their applications in zk-Virtual Machines (zk-VMs).

zk-VMs and zk-EVMs In a state machine, a transition is the process of moving from an old state to a new state by reading a series of inputs and performing sets of opcodes which are a limited and low-level set of instructions. Ethereum is, in essence, a transaction-based state machine, where the state contains all account addresses and their mapped account states. The Ethereum Virtual Machine (EVM) is the mechanism responsible for performing the transitions as a succession of opcodes. zk-VMs (zk-Virtual Machines) and, more specifically, the zk-EVM (Ethereum Virtual Machine) are complex and powerful cryptographic systems that allow one party to generate proofs assessing the correct execution of a Virtual Machine using a SNARK scheme. The proofs can be as short as a few hundred bytes and can be verified in a few milliseconds on any platform (Groth16 [33]). For these reasons, zk-VMs have important applications in blockchain scalability and interoperability. This is also the reason why this area of research has recently seen tremendous activity

in research and development: Linea [10], Cairo [30], Polygon-zkEVM [47], RISC0 [50] or Scroll [1]. However, building a system capable of proving arbitrary executions of the Ethereum Virtual Machine is no easy task. To give an idea, Linea's arithmetization [10] models execution traces of the Ethereum Virtual Machine using hundreds of polynomials and thousands of arithmetic constraints of various types. In this setting, the total witness size for proving the execution of a regular block consists of hundreds of millions of field elements.

Background and Motivation

Cryptographic primitives such as Polynomial Commitments and Succinct Non-Interactive Arguments of Knowledge (SNARKs) have been pivotal in enabling blockchain scalability and privacy. These tools can significantly reduce the computational overhead on the blockchain network. Our research is motivated by the challenge of proving the correctness of arbitrary execution traces of the EVM as specified in [10] in a manner that is both computationally efficient and verifiable.

Polynomial Commitments A polynomial commitment [38] is a cryptographic primitive in which a prover commits to a polynomial $P(X)$ and later proves the evaluation of $P(X)$ at a given point x.

List Polynomial Commitments (LPCs) An LPC is a polynomial commitment with a relaxed security requirement: the commitment is not associated with a single polynomial but rather with a list of polynomials, where the prover can open the commitment to any polynomial from the list. Thus the commitment is not binding to one polynomial but to a list of polynomials.

Succinct Non-Interactive Arguments of Knowledge (SNARKs) Given a binary relation $\mathcal{R}(x, w)$, SNARKs allow proving knowledge of a witness w such that the relation $\mathcal{R}$ (usually drawn from a large family) is satisfied for a public input x. In particular, the verifier needs less time to verify the proof, generated by a SNARK, rather than to re-do all the computations. In the last few years, an ever-growing number of SNARK constructions have emerged, including Groth16 [33], Plonk [7], Halo [22], Halo2 [28], Marlin [25], Spartan [44], Virgo [51], Brakedown [32], Orion [49], Libra [48], Aurora [16], Fractal [24], Sonic [40], Nova [37], and Lasso [45] to cite a fraction of the existing works.

Interactive Oracle Proofs (IOPs) Interactive Oracle Proofs (IOPs) are a family of abstract ideal protocols in which the verifier is not required to read the prover's messages in full. Instead, the verifier has oracle access to the prover's messages and may probabilistically query them at any positions [14]. IOP protocols can be transformed into concrete secure argument systems using Merkle trees thanks to the [15]. Later works have introduced several variants of IOPs such as polynomial-IOPs or tensor-IOPs, where the prover can perform polynomial evaluation queries [7] or tensor queries [21]. Similarly, these protocols can be converted into concrete argument systems (including SNARKs) using functional extractable commitments. This type of approach for building argument systems has led to an extensive line of works and has now become a standard.

Recursion is a technique that consists of verifying a publicly verifiable non-interactive proof inside another argument system. This technique can be used for building incrementally verifiable computation (IVC), proof-carrying data (PCD), proof aggregation, or further compression of proof size. [19] specifies how to instantiate proof-carrying data through recursion using a pairing-friendly cycle of elliptic curves. The works of Halo [22], Halo2 [28] and Nova [37] present several techniques to implement PCD or IVC using a (possibly non-pairing-friendly) cycle of elliptic curves. In [11] the authors present a recursion technique that specifically targets recursion over the protocol of GKR [31] and more generally any interactive protocol whose Fiat-Shamir transform involves hashing long string in the first round.

Contributions

We introduce a novel framework that leverages the Vortex List Polynomial Commitments (LPCs) [13] to instantiate a SNARK specifically tailored for the execution traces of the EVM. Our approach addresses the scalability challenges posed by existing zk-VMs and offers a more efficient mechanism for generating and verifying execution proofs. Furthermore, we present an innovative self-recursion technique and explore its application in optimizing proof verification, contributing to the broader field of incrementally verifiable computation (IVC) and post-quantum SNARKs.

In synthesizing these cryptographic techniques, our research not only advances the state of the art in zk-VMs but also opens new avenues for secure and efficient blockchain computations.

1.1 Our Contributions and Techniques

Arithmetization is a complex step that converts the state transition to some mathematical structure. In Linea's system, the structure of the arithmetization is a set of columns of fixed length. The correct state transition is then verified by sending specific queries on these columns. The queries are usually from a wide class: range checks, permutation checks, scalar-product checks, etc. Since working with different queries can be prone to mistakes and more effort, we first homogenize the queries. For this purpose, we employ our Wizard-IOP, which receives the columns and different types of queries, and uses the Arcane compiler to provide us with a set of columns and just *one* type of query. The previous columns (input of Arcane) are technically a subset of a new set of columns (output of Arcane). The columns are treated as either the coefficients or evaluations of corresponding polynomials and the homogenized query is the evaluation of these polynomials.

Wizard-IOP In Section 4, we present the Wizard-IOP framework. It can be viewed as an extension of the notion of (polynomial-)IOP [15] supporting more complex queries. In this framework, the prover is allowed to send oracle-access to multiple vectors across several rounds of interactions and the verifier may perform queries from a wide class. To give an idea, the verifier may send queries evaluating scalar-products of committed vectors or polynomial evaluations. It may also send queries involving cyclic-shifts of committed vectors or queries asserting that two vectors are permutations of each other.

Wizard-IOP allows designing protocols in a way that contrasts with the usual polynomial-IOP techniques. Compared to polynomial-IOPs, Wizard-IOP offers a higher-level framework for designing protocols. This makes Wizard-IOP suitable for designing protocols that would otherwise be more complex using solely the framework of polynomial-IOP. Most of all, the fact that Wizard-IOP supports queries with this level of abstraction makes it seamlessly compatible with the work of the zk-EVM specification of [10].

Arcane and UniEval compiler Thereafter, Section 5 introduces the Arcane Compiler, a tool that allows transforming any secure protocol specified in the Wizard-IOP model into one secure in the polynomial-IOP model. The UniEval compiler then turns this PIOP into a PIOP where the verifier queries the oracle only on a single opening point for all polynomials. The techniques we use to build Arcane are derived from known modular polynomial-IOPs from past works such as Plonk, Halo2, or Cairo [7, 20, 28, 30, 29]. As the original goal of our work is to build a succinct proof system for the zk-EVM specified in [10], this compiler approach has numerous benefits. An important one is that it allows specifying and implementing batching and optimization techniques that would be significantly more complex otherwise. While the sub-protocols we employ are not new, the succession of steps it follows is endemic to our work. The main feature of our compilation steps (Arcane and UniEval) is that it yields a single-point evaluation PIOP, allowing us to use the output of the compiler alongside a non-homomorphic polynomial commitment (i.e., Vortex) to create an efficient argument system.

Vortex, a Batchable Polynomial Commitment (BPC) A polynomial commitment allows a prover to open the committed polynomial over a given point. A Batchable Polynomial Commitment (BPC) allows the same type of opening for a batch of committed polynomials on the same point. In Section 7, we present Vortex, an adaptation of Ligero [6] into a BPC scheme inspired by the works of Brakedown [32], batch-FRI [18], and RedShift [36].

Similarly to Brakedown, our BPC does not rely on the FRI protocol and it has a proximity check and an evaluation check where the proximity check is indeed the Ligero test. The main difference from Brakedown is the security regime we are dealing with. Based on encoding schemes, one can imagine two security regimes: the unique decoding regime that is the counterpart for the standard binding and the list decoding regime leading to a relaxed binding property where the commitment can be opened to a fixed list.

Working in the list decoding regime requires a new design. Indeed, the evaluation protocol of Vortex is different from the one in Brakedown, where we combine the proximity check and evaluation check as the evaluation protocol. More precisely, in Brakedown, the proximity check can be run independently of the evaluation point, while in Vortex the proximity check is run after seeing the evaluation point. Working in the list decoding regime also brings trade-offs between efficiency and soundness-error. In particular, if the field is large enough, the efficiency gain compared to the loss in the soundness-error becomes of practical interest. We show that a polynomial commitment scheme in the list decoding regime (Vortex LPC) is sufficient for the compilation of PIOP to an argument of knowledge (AoK).

From the instantiation point of view, for hashing the columns, our Vortex scheme relies on a hash function based on the Ring-SIS assumption [39] where we also apply an MIMC hash over the output of the SIS-hash. The first instance of Ring-SIS-based hash functions was introduced in [39]. It is a SNARK-friendly hash function with a linear structure defined over the ring of polynomials of degree less than d, as $H_a(s) = \sum a_i(x)s_i(x) \in \mathcal{R}$ for $\mathcal{R} = \mathbb{Z}_q(X)/X^d + 1$. Another advantage of using such a hash function is the possibility of using lookup arguments if the hash computation is not computed on the verifier side. To encode the rows, we use the (systematic) Reed-Solomon encoding [43]. Vortex commitments have size $O(\sqrt{|M|})$, prover time $O(|M| \log |M|)$ and verification time $O(\sqrt{|M|})$. The reason our proving time is not linear is due to the use of the Reed-Solomon error-correcting codes (whose encoding algorithm requires FFT). Orion and Brakedown [49, 32] achieve linear-time prover algorithms thanks to dedicated and optimized linear-time encodable erasure codes. Although we believe our techniques could be adapted to their erasure codes, we motivate our choice with the fact that Reed-Solomon codes are fast enough for our needs and nicer to work with for recursion. We leave this as an area of optimization to be explored in later versions of this work.

SNARK via Self-Recursion Since Vortex is interactive and has verifier complexity and proof size $O(\sqrt{n})$, using the above compilation technique does not yield immediately a SNARK. Indeed, obtaining a SNARK requires polylogarithmic proof size and non-interactivity. To work around this problem, we use a technique named *self-recursion*. It works by re-arithmetizing the Vortex verifier in the Wizard-IOP framework. The self-recursion reduces the size of the proof by a square root every time it is applied. After $O(\log \log n)$ steps of recursion, we obtain a protocol with $O(\log \log n)$ proof size and verification time. The proof can then be made non-interactive in the random oracle model (ROM) using a suitably chosen hash function. Thereafter, the resulting SNARK can optionally be compressed further to $O(1)$ using existing proof systems such as Groth16[33] or Plonk[7] whose concrete proof sizes are small and verification times are efficient. The advantage of combining self-recursion together with simple recursion is that it greatly reduces the prover time compared to going for a simple recursion with Groth16 or Plonk. One might say that self-recursion compresses the proof loosely but fast, while recursion with pairing-based SNARKs compresses the proof tightly but slowly.

1.2 Overview of Vortex and its Self-Recursion

Vortex Similarly to Brakedown [32] and Orion [49], the Vortex construction is simple and can be succinctly described. Assume that $\mathcal{P}$ and $\mathcal{V}$ are the prover and the verifier. First, we elaborate on the commitment procedure. The prover commits to a matrix W in two steps *row-encoding* and *column-hashing*. $\mathcal{P}$ starts by encoding the rows of the matrix using a Reed-Solomon code to obtain a new matrix W' (namely, row-encoding). The prover then hashes each column of W' and sends them to the verifier as its commitment (namely, column hashing).

The protocol is then followed by two other phases, *proximity check* and *evaluation check*. In the proximity check, the prover sends a vector u, then the prover and the verifier apply the Ligero proximity test over the committed matrix and the encoding of vector u (the encoding is called u'). The Ligero test proves that if the random linear combination of rows is close to codeword u' then all the rows are close to a codeword.

Setting the distance to the unique decoding radius, there is only one polynomial close to the function embedded in the matrix. Finally, the evaluation check guarantees that the evaluation of the polynomial (obtained from the proximity check and unique decoding radius) over a given point x is correct.

The above description is the same as the polynomial commitment in Brakedown [32] and Orion [49]. The main difference between Vortex and Brakedown is the evaluation protocol, where we combine the Proximity check and evaluation check as the evaluation protocol. More precisely, in Brakedown, since we are in the unique decoding regime, the proximity check can be run independently of the evaluation point, while in Vortex the proximity check is run after seeing the evaluation point.

For the instantiation of the hash used on the columns, we use Ring-SIS hashing on the columns of W', and we then apply MIMC over the SIS hash of the columns. Finally, a Merkle tree (based on MIMC) is used to achieve a constant-size commitment. SIS hashing can be seen as a variant of the SWIFFT hash function [39]. Its internal machinery is summed up in the following. Let $v \in \mathbb{F}^m$ be a vector to hash, and $\mathcal{R}$ be a polynomial ring. First, the bits of v are rearranged in a vector v_b of limbs of $\log b$ bits each (b is a parameter of the hash function). In turn, v_b is embedded in a vector of polynomials $\mathbf{w} = (w_1, w_2, \cdots, w_m) \in \mathcal{R}^m$ such that each entry of v_b corresponds to a coefficient in $\mathbf{w}$ in order. Given a randomly sampled public hashing key $\mathbf{A} = (A_0, A_1, \cdots, A_m) \in \mathcal{R}^m$, the digest h_v is obtained as the coefficients of the polynomial

$$h_v(X) = \sum_i A_i(X) w_i(X)$$

.

Self-Recursion Vortex itself is transformed into a PIOP, and in order to convert this PIOP into a SNARK, we develop a technique that we call self-recursion. At a very high level, we design a Wizard-IOP for verifying Vortex proofs. This Wizard-IOP can be once again compiled through the Arcane compiler and Vortex. As a result, we obtain a shorter proof at the cost of a small overhead on the prover time. This operation can be repeated, and after $O(\log \log n)$ iterations, we obtain a short interactive proof that can be compiled into a SNARK using the Fiat-Shamir transform. Our self-recursion technique relies heavily on the fact that the Vortex verifier uses the Ring-SIS hash for hashing the columns and the Reed-Solomon code to encode the alleged evaluations u. Indeed, these two operations are amenable to cheap arithmetization and probabilistic tests (due to their linear structures). Thus, they allow a very efficient recursion procedure.

2 Preliminaries

Here we define the syntax of our main building blocks; SNARKs and polynomial commitment schemes (PCS).

2.1 Argument of Knowledge

We define $\mathcal{R}_\lambda$ to be a relation generator (i.e., $\mathcal{R} \leftarrow \mathcal{R}_\lambda$) such that $\mathcal{R}$ is a polynomial time decidable binary relation. For $\mathcal{R}(x, w)$, we call x as the statement and w as the witness. The set of true statements is denoted by $\mathcal{L}_\mathcal{R} = \{x : \exists\ w \text{ s.t. } \mathcal{R}(x, w) = 1\}$. The definitions in this section are mainly borrowed from [33].

Definition 1 (*Non-Interactive* Arguments for $\mathcal{R}_\lambda$). *A Non-Interactive Argument for $\mathcal{R}_\lambda$ is a tuple of three p.p.t. algorithms* $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ *defined as follows,*

- $\sigma \leftarrow \mathsf{Setup}(\mathcal{R})$*: on input $\mathcal{R} \leftarrow \mathcal{R}_\lambda$ generates a reference string σ. All the other algorithms implicitly receive the relation $\mathcal{R}$.*
- $\pi \leftarrow \mathsf{Prove}(\sigma, x, w)$*: it receives the reference string σ, statement x and witness w. If $\mathcal{R}(x, w) = 1$ it outputs a proof π.*
- $1/0 \leftarrow \mathsf{Verify}(\sigma, x, \pi)$*: it receives the reference string σ, the statement x and the proof π and returns* 0 *(reject) or* 1 *(accept).*

Definition 2 (Completeness). *Completeness says that given a true statement $x \in \mathcal{L}_\mathcal{R}$, the prover can convince the honest verifier; for all $\lambda \in \mathbb{N},\ \mathcal{R} \in \mathcal{R}_\lambda, x \in \mathcal{L}_\mathcal{R}$:*

$$\Pr\left[1 = \mathsf{Verify}(\sigma, x, \pi) :\ \sigma \leftarrow \mathsf{Setup}(\mathcal{R}),\ \pi \leftarrow \mathsf{Prove}(\sigma, x, w)\right] = 1$$

Definition 3 (Soundness). *An argument of knowledge is sound if it is not feasible to convince the verifier of a wrong statement. More formally, for any non-uniform p.p.t. adversary $\mathcal{A}$ we have,*

$$\Pr[1 = \mathsf{Verify}(\sigma, x, \pi)\ \wedge\ x \notin \mathcal{L}_\mathcal{R} :\ \mathcal{R} \leftarrow \mathcal{R}_\lambda,\ \sigma \leftarrow \mathsf{Setup}(\mathcal{R}),\ (x, \pi) \leftarrow \mathcal{A}(\sigma)] \approx 0$$

Definition 4 (Knowledge-Soundness). *Knowledge-soundness strengthens the notion of soundness by adding an extractor that can compute a witness from a given valid proof. The extractor gets full access to the adversary's state, including any random coins. Formally, for any non-uniform p.p.t adversary $\mathcal{A}$ there exists a non-uniform (expected polynomial time) extractor $\mathcal{X}_\mathcal{A}$ such that:*

$$\Pr\left[1 = \mathsf{Verify}(\sigma, x, \pi)\ \wedge \mathcal{R}(x; w) = 0 :\ \begin{array}{c}\mathcal{R} \leftarrow \mathcal{R}_\lambda,\ \sigma \leftarrow \mathsf{Setup}(\mathcal{R}),\\ ((x, \pi), w) \leftarrow (\mathcal{A} \parallel \mathcal{X}_\mathcal{A})(\sigma)\end{array}\right] \approx 0$$

The advantage of the adversary in the knowledge-soundness game (the probability on the left side) is called knowledge-error.[1]

Compared to a non-interactive argument of knowledge, a succinct non-interactive argument of knowledge, or SNARKs, adds a requirement of succinctness. In short and informally, the proof and the verifier time must be small compared with the witness of the relation being proven. We adopt a broad notion of succinctness by only requiring the polylogarithmic proof size and verifier runtime in the witness size.

Definition 5 (Succinctness, SNARK). *A non-interactive argument system $\mathcal{X}$ for a relation $\mathcal{R}_\lambda$ is **succinct** if the size of the proof π produced by the prover and the run-time of the verifier is $O(\mathsf{polylog}|w|)$, for all relations $\mathcal{R}$ drawn from $\mathcal{R}_\lambda$. A non-interactive argument system with this property is called* SNARK*.*

2.2 Roots of unity and Lagrange polynomials

Let $\mathbb{F}_q$ be a finite field of prime order q. We call the roots of the polynomials $Z_k(X) = X^k - 1$ the k-th roots of unity. Together, they form a multiplicative subgroup Ω_k of $\mathbb{F}_q^*$, provided that $k|q-1$. We say that $Z_k(X) = X^k - 1$ is the vanishing polynomial of Ω_k.

We assume k is a power of 2, for each subgroup $\Omega_{k'}$ of Ω_k (thus, $k'|k$), we have $\omega' = \omega^{k/k'}$ where ω and ω' are the generator of Ω_k and $\Omega_{k'}$ (res.).

For any subgroup Ω_k, the collection of polynomials given by $(\mathcal{L}_{\omega,\Omega_k}(X))_{\omega\in\Omega_k}$ forms the Lagrange basis for polynomials of degree $k-1$ where,

$$\forall \omega \in \Omega_k : \mathcal{L}_{\omega,\Omega_k}(X) = \frac{\omega(X^k - 1)}{k(X - \omega)}$$

[1] Although we only use the notion of knowledge-soundness throughout this work, a more general notion exists: *witness-extended emulation* where the extractor outputs an (indistinguishable) transcript of the protocol.

Let $v = (v_1, \cdots, v_k)$ be a vector of $\mathbb{F}^k$. We call $v(X)$ the polynomial encoding v and we will often implicitly refer to a vector and its polynomial encoding with the same notation.

$$v(X) = \sum_{i\in[k]} v_i \mathcal{L}_{\omega^i,\Omega_k}(X) = \frac{X^k - 1}{k} \sum_{i\in[k]} v_i \cdot \frac{\omega^i}{X - \omega^i}$$

When k is implicit, we use ω, Ω and L_ω instead of ω_k, Ω_k or L_{ω,Ω_k} for convenience in our notations.

Definition 6 (Domain Selector). *We define the (sub)domain-selector as the polynomial $Z_{n,kn}(X)$ that is 1 over the subgroup Ω_n of Ω_{nk}, and zero everywhere else. Namely, we have $Z_{n,kn}(X) = \sum_{j=0}^{n-1} \mathcal{L}_{\omega^{kj},\Omega_{kn}}(X)$ and ω (res. ω^k) being the generator of Ω_{nk} (res. Ω_n).*

2.3 Polynomial Commitments

Definition 7. *A polynomial commitment is a tuple of p.p.t. algorithms* (Setup, Commit, Open) *where,*

- $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, t)$ *generates the public parameters* pp *suitable to commit to polynomials of degree $< k$.*
- $C \leftarrow \mathsf{Commit}(\mathsf{pp}, P(X))$ *outputs a commitment C to a polynomial $P(X)$ of degree at most k using* pp.
- $1/0 \leftarrow \mathsf{Open}(\mathsf{pp}, C, x, y; P(X))$ *is a (public-coin) protocol between the prover and the verifier where the prover aims to prove the relation;*

$$\mathcal{R} = \{(x, y, C; P(X)) : \ P(x) = y, C = \mathsf{Commit}(\mathsf{pp}, P(X))\}$$

In this protocol, the prover's input is $(P(X), x, y, C, \mathsf{pp})$ and the verifier's input is (x, y, C, pp). The output of the protocol is 1 if the verifier accepts the proof and 0 otherwise.

We use the definition of the correctness and the knowledge-soundness from [7].

2.4 IOPs and Polynomial-IOPs

An interactive oracle proof (IOP) for a relation $\mathcal{R}(x, w)$ is an interactive proof in which the verifier is not required to read the prover's messages in their entirety; rather, the verifier has oracle access to the prover's messages, and may probabilistically query them. In polynomial IOP (PIOP) the messages are polynomials and the verifier has oracle access to the evaluation of polynomials on the queried points.

2.5 Reed-Solomon Codes

Definition 8 (Linear Code [49]). *A linear error-correcting code with message length k and codeword length n with $k < n$ is a linear subspace $C \subset \mathbb{F}^n$, such that there exists an injective mapping from message to codeword $EC : \mathbb{F}^k \to C$ which is called the encoder of the code. Any linear combination of codewords is also a codeword. The rate of the code is defined as $\rho := k/n$. The distance between two codewords u, v is the number of coordinates on which they differ, denoted as the Hamming distance $\Delta(u, v)$. The relative (or fractional Hamming distance) is defined as $\delta(u, v) = \Delta(u, v)/n$. The minimum distance is $d := \min_{u,v} \Delta(u, v)$.*

Definition 9 (Reed-Solomon Code). *Consider positive integers n, k, a finite field $\mathbb{F}$, and a set $D \subseteq \mathbb{F}^*$ with $|D| = n$ (the set D will be referred to as the domain). The Reed-Solomon code over $\mathbb{F}$ with domain D and the message space of size k is defined as:*

$$\mathsf{RS}[\mathbb{F}, D, k] := \{p(x)|_{x\in D} : p(X) \in \mathbb{F}[X], \deg(p) \leq k\},$$

By $p(x)|_{x\in D}$, we denote the set of evaluations of p over the set D and $n = |D|$ is called the codeword size. For $v \in D$ and $p \in \mathsf{RS}[\mathbb{F}, D, k]$, we will also use the notation $p|_v$ to refer to $p(v)$.

By $F_{<n}[X]$, we denote the set of polynomials of degree less than or equal to k, i.e.

$$\mathbb{F}_{<k} := \{p(X) \in \mathbb{F}[X] : \deg(p) \leq k\},$$

Distance to a Reed-Solomon Code Consider arbitrary $f \in \mathbb{F}^{|D|}$. The distance of f from the set $V = \mathsf{RS}[\mathbb{F}, D, k]$ is defined as $\Delta(f, V) := \min_{v \in V} \Delta(f, v)$ (and similarly for relative distance).

2.5.1 Reed-Solomon Codes over Roots of Unity In this work, we choose the domain set $D = \Omega_n$ as the set of n^{th} roots of unity. Consider a fixed generator ω of Ω_k. Then $D = \{\omega^i\}_{i=0}^{n-1}$ and we will associate polynomial evaluations $p(x)|_D$, called codeword space, with vectors $(p(\omega^0), p(\omega^1) \dots p(\omega^{n-1}))$, ordered by the natural ordering induced by the exponents of generator ω.

2.6 A General Security Proof for Sub-Protocols

Apart from the security of Vortex that would be discussed in a separate work, all the sub-protocols that we use (particularly the one for the self-recursion) are secure following the same reasoning. This reasoning heavily depends on Schwartz-Zippel Lemma.

Lemma 1 (Schwartz-Zippel Lemma). *Let $P(X)$ be a non-zero polynomial of degree d over a field $\mathbb{F}$. Let S be a finite subset of $\mathbb{F}$ and let r be selected at random from S. Then*

$$\Pr_{r \in \mathbb{F}}[P(r) = 0] \leq d/|\mathbb{F}|$$

Throughout the paper, we always represent the sub-protocols in the PIOP framework. This would allow us to argue their security in a general manner.

PIOP and its Knowledge-Soundness. The PIOP is knowledge-sound if there exists a probabilistic polynomial time algorithm E (called the extractor) which interacts with the prover on a statement x and it has the capability: to run the prover for a specified number of steps, inspect its state and rewind it repeatedly to a previous state. If the prover interactions cause the verifier to accept, the extractor is able to recover a witness w such that $R(x, w) = 1$.

Remark Generally, in many other protocols (and the ones we employ) the PIOP relation is reduced to global constraints which are evaluated at random points. The resulting protocol is secure if the global constraints are satisfied over the random points and if the size of the finite field is large enough (to have negligible probability $d/|\mathbb{F}|$ in the Schwartz-Zippel Lemma, $|\mathbb{F}|$ should be large). Note that the final protocol is made non-interactive using the final Fiat-Shamir transform and its soundness is not statistical but computational.

It is well-known that a PIOP can be transformed into a concrete AOK by replacing the oracle with a polynomial commitment. For such a resulting protocol, we have:

Lemma 2 (Knowledge-Soundness of AOK). *If the* PIOP *and the polynomial commitment are knowledge-sound, then the* AOK *is knowledge-sound.*

Putting everything together, all the sub-protocols can be proven to be knowledge-sound through this general approach: first the reduction of relations to some global constraints, then the Schwartz-Zippel lemma is applied to guarantee that the constraints are satisfied, and finally the oracle of the PIOP is replaced by a polynomial commitment.

2.7 List Polynomial Commitment

We now present the syntax and security of the list polynomial commitment. The definitions here follow the ones from Redshift ([36]) but extended to a batched setting. Our presentation closely follows the formalization of [20, 7]. We considered batched openings of multiple polynomials. One difference is that we only consider openings of all these polynomials at the same evaluation point.

The list polynomial commitment has a relaxed binding property, each commitment corresponding to a list of polynomials that is determined by a distance parameter. The commitment can be opened to any of the polynomials belonging to the list. Moreover, the polynomials in the list will jointly agree on the same agreement set.

Definition 10 ((Batched) List Polynomial Commitment). *A list polynomial commitment scheme is a triplet* $(\mathsf{Setup}, \mathsf{Commit}, \mathsf{OpenEval})$ *that is defined w.r.t. a linear code, distance parameter* θ *and domain* D*. It satisfies:*

- $\mathsf{Setup}(1^\lambda, k)$ *generates public parameters* pp *(a structured reference string) suitable to commit to polynomials of degree* $< k$*. Implicitly, the parameters for encoding are included in* pp*.*
- $\mathsf{Commit}(\mathsf{pp}, f_1(X) \dots f_n(X))$ *outputs a commitment* C *to functions* $f_1(X) \dots f_n(X) \in \mathbb{F}[X]$
- $\mathsf{OpenEval}$ *is an IOP between a prover* P_{PC} *and a verifier* V_{PC}*, where the prover is given* n *functions* $f_1(X) \dots f_n(X) \in \mathbb{F}[X]$ *and attempts to convince the verifier of the following relation:*

$$\begin{aligned} \exists A \subset D \text{ s.t } |A| \geq (1-\theta) \cdot |D| \text{ and } \exists (P_1 \dots P_n) \in (\mathbb{F}^{<k}[X])^n \text{ s.t.} \\ \big(P_i(x) = y_i \ \wedge \ f_i(a) = P_i(a)|_{a \in A} \text{ for all } i \in [n]\big) \wedge \\ \wedge \ C = \mathsf{Commit}(\mathsf{pp}, f_1 \dots f_n) \end{aligned}$$

where both parties receive the following:

- *security parameter* λ*, degree bound* k *and batch size* n*, such that* $k, n = \mathit{poly}(\lambda)$*.*
- *The public parameters* pp*, where* $\mathsf{pp} = \mathsf{Setup}(1^\lambda, k)$*.*
- *An evaluation point* x *and alleged openings* $y = (y_1 \dots y_n)$*.*
- *Alleged commitment* C *for functions* $f_1(X) \dots f_n(X)$*.*

In addition, the verifier receives oracle access to evaluations of f_i *over* D*.*

Definition 11 (Completeness of a List Polynomial Commitment Scheme). *We say that a polynomial commitment scheme has (perfect) completeness if for any security parameter* λ*, any integers* $k, n = \mathrm{poly}(\lambda)$*, any polynomials* $P_1(X) \dots P_n(X) \in \mathbb{F}_{<k}[X]$*, arbitrary evaluation point* x *and alleged opening* y*, if* $C = \mathsf{Commit}(\mathsf{pp}, P_1(X) \dots P_n(X))$ *and* $P_i(x) = y_i$ *for all* $i \in [n]$ *then an interaction of* $(P_{\mathrm{PC}}, V_{\mathrm{PC}})$ *where* P_{PC} *runs on the aforementioned parameters will result in the verifier accepting with probability one.*

Definition 12 (Knowledge Soundness in the Random Oracle Model). *There must exist a* PPT *extractor* E *such that for every* PPT *adversary* $\mathcal{A}$ *and arbitrary degree* $k = \mathrm{poly}(\lambda)$*, the probability that* $\mathcal{A}$ *wins the following game is negligible, where the probability is taken over the coins of* Setup*,* $\mathcal{A}$ *and* V_{PC}*. Moreover, the extractor has access to the random oracle queries of* $\mathcal{A}$*:*

- $\mathcal{A}$ *receives degree* k *and* $\mathsf{pp} = \mathsf{Setup}(1^\lambda, k)$*.* $\mathcal{A}$ *outputs* C*.*
- E *receives the commitment* C *and inspects the random oracle queries made by* A *in the previous step and recovers* $f_1(X) \dots f_n(X) \in [X]$*.*
- E *applies the efficient list-decoding algorithm on all* f_i *simultaneously to obtain list* L*, defined as:*

$$L = \left\{ (P_1(X), \dots, P_n(X)) \in (\mathbb{F}^{<k}[X])^n \text{ s.t. } \begin{array}{c} \exists A \subset D, \text{ s.t. } |A| \geq |D| \cdot (1-\theta) \\ \text{and } f_i(a) = P_i(a)|_{a \in A} \end{array} \right\}$$

- $\mathcal{A}$ *outputs an evaluation point* x *and claimed openings* $y := (y_i)_i$*.*
- $\mathcal{A}$ *interacts with the* V_{PC} *verifier of the* $\mathsf{OpenEval}$ *algorithm. The inputs of* $\mathcal{A}$ *for this subprotocol are* C*,* x *and* y*.*
- *The extractor may check consistency and output a set* S *of witnesses, where* $S \subseteq L$*.*
- $\mathcal{A}$ *succeeds if* V_{PC} *accepts and there exists no tuple* $(P_1(X) \dots P_n(X)) \in L$ *such that* $P_i(x) = y_i$ *for all* $i \in [n]$*.*

3 Linea's Proof System Overview

This section outlines the structure of the prover's stack of Linea and provides a high-level description of its subcomponents. Linea's proof system is designed to ensure the integrity and efficiency of transactions within the Ethereum Virtual Machine (EVM) through a multi-faceted approach involving several specialized circuits and proof systems. Each circuit plays a critical role in validating different aspects of transaction execution and data compression, culminating in an aggregated proof. Below, we detail the components of the proof systems and their respective functionalities. The current document does not detail the compression proof and aggregation proofs beyond the current section and is essentially focused around the execution proof system.

3.1 Execution proof

The execution proof is pivotal in validating the correct execution of transactions within the EVM. It employs a proof structure that integrates the work Vortex [13] and [12] and Plonk [7]. It represents the largest part of this work and is comprehensively described in Section 5, Section 6 Section 7, Section 8 and Section 9). The finality of this process is a Plonk proof based on the BLS12-377 curve. As an outline, the execution proof is generated from a set of execution traces satisfying the constraints of $alex : ref$ and from a set of EVM state accumulator proofs (see Appendix C) justifying the state-root transition allegedly from a state-diff indicated in the traces. The execution proofs statement includes:

- knowledge of EVM traces satisfying the arithmetization constraints
- correctness of the state-accumulator traces and their consistency with the EVM traces.
- the correctness of the execution of the precompiles, Keccak and Secp256k1 ECDSA verifications.
- the consistency of the public inputs of the proof with the above (state root hash transition, encoded transactions, timestamps, bridge-messages

This is accomplished by following a multi-step cryptographic compilation process

1. A dedicated protocol in the Wizard-IOP model described in Section 4 is constructed specifically for the relation of the execution proof.
2. The Wizard-IOP protocol is compiled down to a SNARK following the steps described in Section 5, Section 6 Section 7, Section 8 and Section 9. This is achieved by repeating a cycle of self-recursion compilation.
3. The resulting SNARK is recursively verified within a Plonk proof to prepare it for the aggregation step

3.2 Compression proof

Complementary to the execution proof, the Compression Circuit focuses on verifying the effective compression of data streams, which constitute the inputs for the EVM execution circuit. This verification is crucial for ensuring that the compressed data, once submitted on the Ethereum blockchain, can be accurately decompressed to reveal the essential inputs for the execution circuit's validation process. The Compression Circuit utilizes the Plonk proof system also based on the BLS12-377 curve, to generate proofs of correct data compression.

3.3 Finalization (Aggregation) proof

At the core of Linea's proof system lies the Finalization (or Aggregation) Circuit, which serves as the linchpin for recursively verifying proofs generated by N execution circuits and M compression circuit instances. This circuit embodies the primary assertion of Linea's prover system and is the only circuit subjected to external verification. It leverages a composite proof system that combines several Plonk circuits on the BW6, BLS12-377, and BN254 curves. This strategic use of 2-chained curves—BLS12-377 and BW6—enables efficient recursion of proofs. The final proof is formulated on the BN254 curve, chosen for its efficient verifiability on Ethereum, facilitated by the availability of precompiles. The aggregation proof is also responsible for "connecting" the public inputs of all proofs and assessing their consistency the public inputs of the final aggregation proof.

4 Wizard IOP

The prover $\mathcal{P}$ of an IOP protocol[15] provides oracle access to (possibly large) messages to a verifier $\mathcal{V}$. The verifier can then send certain kinds of queries (from a small family) to the oracle. Several variants of IOP exist in the literature. In particular, polynomial-IOPs [40], [25], [7] specify a model in which all prover messages are viewed as polynomials and the verifier may make queries to evaluations of these polynomials at random points of the verifier's choice. More recent works study tensor-IOP [21] protocols in which the verifier is granted the right to query scalar-products of the prover's messages (seen as vectors over a field) by random vectors with the restriction that these vectors must have a tensor structure.

Wizard-IOPs specify a model that extends this perspective on IOPs. The prover sends oracle access to vectors (columns) over a given field with a prespecified size and the verifier is allowed to perform queries chosen from a *wide class*: inclusion, permutation, inner-product, global-constraints. The terminology column is also used to denote prover messages or offline precomputed vectors of values. As we explain later in this section, queries can involve several columns or "abstract references" to them. We elaborate on the notion of "abstract references" later, but to give an initial idea: taking the "cyclic shift" of a column v would be considered an "abstract reference". The backbone idea behind Wizard-IOP is that it allows us to specify ever more complex protocols in the simplest possible way while intermediate protocol design techniques (such as proving a lookup relation or a permutation relation) are treated as automatable compilation steps. Subsequently, instead of mentally building modular protocols from the bottom up using the notion of univariate queries as atoms of a more complex system, the framework of Wizard-IOP allows specifying protocols with a top-down approach. We start from an abstract protocol and work out an optimized polynomial-IOP throughout the steps of the Arcane compiler Section 5. While this simplifies protocol specification and security analysis, it also allows to automate optimizations and batching techniques. The zk-EVM arithmetization specified in [10] involves hundreds of columns and thousands of constraints. It would be extremely tedious to manually unfold all the sub-protocols and optimization techniques required in order to present a concrete polynomial-IOP for this arithmetization. However, since their description is written in a formalism closely matching the Wizard-IOP model, we can almost directly transpile their arithmetization into a Wizard-IOP. Another advantage to reasoning in terms of compilation steps rather than sub-protocols is that it facilitates maintenance processes. Assuming that a new (purely hypothetical) batching technique for "range-check" is discovered and improves the prover's runtime by a factor of 2, then we could simply add it to the compiler and this will propagate on every Wizard-specified protocol. Similarly, if a vulnerability is found in one of the techniques, fixing a compiler step will fix all protocols using it without any risk of forgetting any part.

4.1 Random coins

The Wizard-IOP framework allows declaring random coins as part of the protocol. As in public-coin protocol, random coins are random messages sent by the verifier to the prover. Registered coins can then be used in the following parts of the protocol. The framework offers two types of coins: (1) **random field**: field elements on a field of size approximately 256 bits (2) **random integer vector** a list of integers of bounded size, usually representing random positions in a vector.

4.2 Columns

Columns are the cornerstone of the Wizard-IOP framework. They symbolically represent a vector of values used as part of the protocol in a broad sense. They can be used to denote portions of the witness, messages sent to the verifier, portions of the proving or verifying key or intermediate committed values. Columns are declared as part of the protocol definition along with their sizes (restricted to be a power of two) and their visibility (e.g. whether the column is meant to be sent to the verifier, or to be part of the witness).

The framework allows the following column's visibilities:

- **Ignored**: this marks the column as "already compiled" and being disregardable by the compiler. They still exist in the protocol but stay purely internal to the prover. An example is: when "Committed" columns are eventually committed to, the compiler sets the "Ignored" tag on the column to indicate there is no need to commit a second time to it.
- **Committed**: this marks the column as being sent to the oracle. Implicitly, this indicates the intent to commit to this column during the compilation phase and to keep its content as part of the witness or an intermediate value visible only by the prover.
- **Proof**: this marks the column as being sent to the verifier directly
- **Precomputed**: this indicates that the column's assignment is (1) internal to the prover, (2) destined to be committed to, (3) known at compile time.
- **Verifying-key**: the marks the column's assignment as visible by the verifier and known at compile time.
- **Verifier-defined**: this indicates that the column is entirely constructible by the verifier from other values that the verifier has at its disposal. For instance, a column constructed by stacking the multiple of several random coins would bear this tag.

4.2.1 Abstract references Abstract references are a useful way to refer to vectors directly derived from pre-existing committed columns. These operators can be combined and used as the object of a query. For instance, one might declare a global constraint involving a cyclically shifted version of a committed column v rather than on the entirety of the positions of v. Importantly, abstract references are neither committed columns nor predicates about columns but can be seen as a way to make queries about committed columns more expressive. Abstract references do not have a status on their own as they may involve several queries having a different status in the protocol.

Expression: defined as a polynomial expression involving columns of the same size and possibly scalar values.

Cyclic shifting: Given a vector v and an integer k (possibly negative), we return a cyclically-shifted version of v by k elements. We may use the notation $v \ll k$ to refer to the resulting vector.

4.2.2 Verifier defined columns Verifier-defined columns are columns that are constructed from other values known from the verifier. They are typically tiny or have a very sparse structure.

- **Periodic sampling**: A periodic sampling column is a column repeating a sequence of the form $1, 0, 0, 0, \ldots$ where the pattern size is a power of two.
- **Position indicatrice**: A column that contains zeroes in every position except for a prespecified one.
- **Constant column**: A column whose position are all equal to a prespecified constant
- **Integer vector coin**: A column built by taking the values of a **random integers vector** coin, padded with a constant value.
- **Stacked values**: A column built by stacking values known by the verifier (including columns) padded with a constant.

4.3 Available queries

In the following, we list and describe the queries available to the verifier $\mathcal{V}$ in the wizard-IOP framework.

Range check Let B be a constant bound. The query is made over a column v, and the oracle responds with 1 if and only if all the entries v_i of v satisfy $0 \leq v_i < B$. We denote the range checks as, "Range" : $v < B$.

Inclusion check (lookup) Given two lists of columns (or tables) S and T, we check that all rows in S should be included among the rows of T, ignoring multiplicities. We denote the inclusion query as, "Inclusion" : $S \subset T$.

Inclusion queries also support additional features:

– **Fragmented tables**: The table T can be supplied in the form of several tables with the same number of columns. In this case, the query is understood as holding for the union of the two tables.
– **Conditional inclusion**: In addition to S and T, the user can provide a query filter F. The inclusion constraints will be considered as "void" in positions where the filter F is zero.

Fixed permutation check Given two lists of columns (seen as tables) and any (imposed) fixed permutation σ, the oracle checks if the i^{th} row in S must equal the row at index $\sigma(i)$ in T for all rows i of S. If and only if that is the case, the oracle returns 1, otherwise 0. The tables S and T must have the same number of rows. We denote a fixed permutation check as, "FixedPermutation" : $S \sim_\sigma T$.

Fixed-permutation queries also support optional features:

– **Fragmented tables**: The table T and S can be supplied in the form of several tables with the same number of columns. In this case, the query is understood as holding for the union of the two tables.

Permutation check Given two tables (e.g. two lists of tables) S and T, all rows in S should be included among the rows of T (and vice-versa), accounting for multiplicities. Thus, S and T must have the same number of rows. Note that in fixed permutation queries, σ is imposed. Here the oracle accepts if a permutation σ exists. We denote a permutation query as, "Permutation": $S \sim T$.

Permutation queries also support optional features:

– **Fragmented tables**: The table T and S can be supplied in the form of several tables with the same number of columns. In this case, the query is understood as holding for the union of the two tables.

Inner product query Given two columns A and B, as well as a scalar c, the oracle returns 1 if and only if $\langle a|b\rangle = c$. We use the notation "InnerProduct" : $\langle a|b\rangle = c$ to denote the declaration of a query within a protocol or sub-protocol.

Local constraint The verifier evaluates an arithmetic equality involving possibly random coins, particular positions of columns or the result of other queries and returns 1 if and only if the equality is satisfied. The position at which columns are pointed to must be fixed as part of the protocol definition.

For example, let u, v be two columns to which we have oracle access. We may send the local constraint query "Local": $u[0] - 2v[1] == 0$ to ask the oracle if the first entry of u equals the double of the second entry of v. We may conveniently express local constraints over polynomials (rather than vectors) over fixed points.

Global constraint Given a k-variate arithmetic expression $\mathcal{C}$ whose (total) degree should be reasonably low and a list of k columns $v_1, \cdots, v_k$ of the same size n and possibly other scalar values arising in the protocol (random coins, query response, ...). The oracle returns 1 if and only if for all i, $\mathcal{C}(v_{1,i}, \cdots, v_{k,i}) = 0$.

For instance, the global constraint "Global": $\mathsf{Shift}(u, 1) - u = 0$ asserts that "all" the entries of u are equal to the next consecutive entry of u. Thus, this constraint asserts that all entries of u are equal. Again, we may express a Global constraint based on polynomials (rather than vectors) when convenient.

Global constraints also support the following optional feature:

– **Bound cancellation**: unless specifically stated otherwise, Global constraints are cancelled implicitly at positions where the cyclic shifts wraps around. In the above, example the Global constraint would be ineffective on the last row. This behaviour is motivated by the fact that this is what is intended most of the time.

Local position opening: Given a column A and a position k indexing a row of the column A, the oracle returns $y = A_k$

Univariate evaluations (UniEval) For a column v of size n, let the polynomial $v(X)$ evaluate to v_i on a subgroup of n-roots of unity. The oracle returns a univariate evaluation of $v(X)$ over an unspecified point (possibly a random point) chosen by the verifier. For convenience, we will usually talk about one univariate query for multiple polynomials to let the compiler know these are queried at the same point. Note that they are not natively supported in the initial protocol definition. The reason is that (1) Lagrange basis evaluations are not nicely compatible with how the columns are segmented in the splitting compiler in Section 5 (2) the use cases we know of for this are niche (although far from irrelevance). Nonetheless, it is still possible to emulate Lagrange basis evaluation in the Wizard-IOP framework without using the Univariate query. We refer the reader to section Appendix A.2 for an example.

5 The Arcane Compiler: Polynomial-IOP from Wizard-IOP

The Arcane compiler is a cryptographic compiler that converts Wizard Interactive Oracle Proofs (IOPs) into Polynomial IOPs through a series of compilation steps. Each step is designed to perform either minor optimizations or apply reduction techniques to simplify the protocol by reducing the variety of queries it uses. The process begins with substituting the range checks in favour of inclusion checks, which are then further simplified into local and global constraints. The process continues substituting queries for others until it ultimately yields a Polynomial IOP, where the only remaining queries are of type "univariate". The compilation is performed in such a way that the protocol retains its functionality throughout the compilation phases only suffering minor polynomial statistical soundness losses.

The Arcane compiler also addresses the challenge of working with columns of heterogeneous sizes and outputs a P-IOP whose oracle-message (committed columns) are all of the same size. This is achieved thanks to two complementary new techniques: "column splitting" and "column sticking". The former works by segmenting large columns into smaller ones preserving the sequentiality of the cells within the segments. The latter performs the opposite operation, by interleaving small columns into larger ones. An important benefit of this approach is its compatibility with statistical optimizations. Indeed, it turns out the arithmetization of [10] uses columns as CPU registers and prepends the "active" rows (i.e. corresponding to actual EVM execution traces) with padding values so that the resulting columns all have their target sizes. This results in constant column assignment for the corresponding segments and these can be tracked at runtime and lead to practical optimizations for the prover.

The organization of the section follows the compilation sub-routines in order:

- Section 5.1 specifies the compilation rules to simplify the range-checks.
- Section 5.2 details the above-mentioned column-splitting phase.
- Section 5.4 explores our techniques for compiling the inclusion queries using methods largely inspired from [34].
- Section 5.5 and Section 5.6 discuss the reduction of the fixed-permutations and the permutation constraints, highlighting techniques inspired from [9] and [7].
- Section 5.7 details the compilation of the inner-product queries, presenting en-route an efficient batch argument for inner-products.
- Section 5.8 discusses the aforementioned "column sticking" compilation phase.
- Section 5.11 expands on the techniques used of the compilation of the global constraints, initially presented in [7]
- Section 5.12 concludes the Arcane compilation ironing out all the remaining artefacts into proper univariate queries.

5.1 Reduction of the range checks

We opt for a simple method for compiling range checks. During a preprocessing phase, we send oracle access to a column $b = (0, 1, 2, \cdots, B-1)$ (e.g. we tag it as "precomputed")for each bound B appearing in the input protocol. Then, all range-checks, "Range" $v < B$, are converted into inclusion checks, assessing if all entries of v are entries of b regardless of the positions or multiplicity.

5.2 Splitting the large columns

The **splitting** compilation step specifically addresses a challenge that we encounter while working with the EVM arithmetization: we are working with columns of heterogeneous sizes and the polynomial commitment Vortex expects a regular matrix to commit to. The **splitting** step is set with a target column length and its role is to cut all the columns whose length exceeds the target length. To illustrate, say we have a column A of size 2^{20} and we have a target size of 2^{15}. The splitting step cuts A in 32 consecutive column segments $A_1, \dots A_{32}$ each representing a sequence of 2^{15} consecutive values of A and then replaces all occurrences of A in the original protocol by occurrences of $A_{1\dots32}$, namely in all queries objecting to A.

We detail the replacement procedure for each type of query and abstract references:

- **Inclusion check**: When A is part of the "including" side of the relation, the query is replaced by an equivalent one using the fragmented table feature over the segments $A_{1\dots32}$. When A is on the "included" side of the query, then the query is replaced by multiple queries, one for each segment.
- **Permutation and fixed-permutation checks**: the query is replaced by an equivalent one using the fragmented table feature over the segments $A_{1\dots32}$
- **Inner product checks**: the query is replaced by one equivalent query for each segment. The verifier checks that the sum of the alleged values for each segmented query matches the alleged value for the original one.
- **Local openings and local constraints**: The query is replaced by an equivalent one pointing to the relevant segment storing the requested position on the original column.
- **Global constraints**: This is the tricky bit. Naively replacing the constraint by a sequence of equivalent ones on each segment is unfortunately not sufficient. When the original constraints refer to a cyclically shifted version of an original column, applying the same shifting over the segment would cause boundary issues. To remediate, we cancel the constraint on the segments boundaries and add a "stitching" local constraint to cover the cancelled positions.

Similarly, another set of technicalities arises when considering queries referencing columns of type expression and that these expressions refer to cyclically shifted columns. In this case, the "segmented" version of the shifted column overlaps two segments of the non-shifted segment. The compiler addresses to this by reconstructing an equivalent segment for the shifted segment by stitching together cyclic-shifting of the segments of the underlying column. The stitching operation is employed by using the **position indicatrice** columns, noted I_k for position k.

Let A be an original column of size N, split in l segments of size n: $A_1, \dots A_l$ and let us consider the reconstruction of the l segments of $A \ll m$ (noted B_i) where m is positive (for negative m substitute m by $N + m$). Set $q = m//n, r = m \mod n$ and set $I_{[r+1,n)} = \sum_{p\in[r+1,n)} I_p$. B_i is constructed as

$$\begin{aligned} B_i &= (A_{i+q} \ll r) + I_{[r+1,n)} \left((A_{i+q+1} \ll r) - (A_{i+q} \ll r)\right) \\ &= (A_{i+q+1} \ll r) + I_{[0,r)} \left((A_{i+q} \ll r) - (A_{i+q+1} \ll r)\right) \end{aligned}$$

depending on which one is cheaper to represent.

5.2.1 Example 1: Lookup Let A and B be two columns of size 8 and 4 respectively and let a query Q_L : ‘Ìnclusion” : $A \subset B$. Consider running the splitting on the resulting input Wizard with a splitting size of 4.

$$A = (a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7)$$
$$B = (b_0, b_1, b_2, b_3)$$

The compiler, keeps B as is since it already has the target size. A is replaced by two sub-columns (A_1, A_2) of size 4 such that $A = A_1 \| A_2$. Q_L is also split into two sub-queries $Q_{L,1}$ and $Q_{L,2}$.

$$A_1 = (a_0, a_1, a_2, a_3)$$
$$A_2 = (a_4, a_5, a_6, a_7)$$
$$Q_{L,1} = \text{“Inclusion”} : A_1 \subset B$$
$$Q_{L,2} = \text{“Inclusion”} : A_2 \subset B$$

5.3 Example 2: Global constraint with shifting

Let A be a column of size 8 and let Q_G be a global constraint involving A:

$$A = (a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7)$$
$$Q_G = \text{“Global”} : A + 1 = A \ll 1$$

As in the previous example, A is split into two sub-columns A_1 and A_2. The query Q_G is also decomposed in two sub-queries $Q_{G,1}, Q_{G,2}$. Observe, that the same query as Q_G applied over A_1 and A_2 does not work as it dismisses the constraint between a_3 and a_4 and introduces erroneous constraints between a_3, a_0 and a_7, a_4. Therefore, we construct faithful images of the two sub-columns of $A \ll 1$: $(A \ll 1)_1, (A \ll 1)_2$ and the corresponding sub-queries as follows:

We have $I_{\{3\}} = (0, 0, 0, 1)$ and A_1 and A_2 as above. Then,

$$\begin{aligned}(A \ll 1)_1 &= (A_1 \ll 1)(1 - I_{\{3\}}) + (A_2 <<\ll 1) I_{\{3\}} \\ &= (a_1, a_2, a_3, a_0 \times 0) + (a_5 \times 0, a_6 \times 0, a_7 \times 0, a_4) \\ &= (a_1, a_2, a_3, a_4) \\ (A \ll 1)_2 &= (A_2 \ll 1)(1 - I_{\{3\}}) + (A_1 <<\ll 1) I_{\{3\}} \\ &= (a_5, a_6, a_7, a_0)\end{aligned}$$

$$Q_{G,1} = \text{“Global”} : A_1 + 1 = (A \ll 1)_1$$
$$Q_{G,2} = \text{“Global”} : A_2 + 1 = (A \ll 1)_2$$

5.4 Reduction of the inclusion checks

Our present technique is borrowed from [34] (a.k.a. univariate log-derivative lookups). Let $S = (S_0, \ldots, S_{k-1})$ and $T = (T_0, \ldots, T_{k-1})$ be two tables and assume a query $S \subset T$ is being declared. Mathematically speaking, the techniques relies on the equivalence that $S \subset T$ if and only if $\exists M \in \mathbb{F}^{|T|}$ such the equality of rational function $\sum \frac{1}{S_i+X} = \sum \frac{M_i}{T_i+X}$ holds. Here, the sums are spanning respectively on the length of S and T. To test the functional equality, the verifier can make a probabilistic check. Namely, he can test it on a random point of $\mathbb{F}$ instead of testing it for all $X \in \mathbb{F}$. The soundness of the technique follows from a variant of the Schwartz-Zippel lemma holding for rational functions. We begin by describing the protocol in case S and T consist of only a single column in Fig. 1 and explain step-by-step how the protocol is expanded and optimized to support every feature.

Inclusion(S, T)

1. The prover commits to a column M with an equal number of rows as T. Which contains the occurrence count of the entries of T in M.
2. The verifier responds with a random coin field element γ.
3. The prover commits to Σ_S a polynomial interpolating the vector of partial sums (Σ_S) such that $(\Sigma_S)[i] = (\Sigma_S)[i-1] + \frac{1}{S_i+\gamma}$ and $(\Sigma_S)[0] = \frac{1}{S_0+\gamma}$.
4. The prover commits to Σ_T, interpolating the vector of partial sums (Σ_T) such that $(\Sigma_T)[i] = (\Sigma_T)[i-1] + \frac{M_i}{T_i+\gamma}$ and $(\Sigma_T)[0] = \frac{M_0}{T_0+\gamma}$.
5. The verifier uses a local opening query to open the last positions of Σ_S and Σ_T and checks that:
$$(\Sigma_S)[|S|-1] == (\Sigma_T)[|T|-1]$$
6. "Local" constraint :
$$(\Sigma_S)[0] = \frac{1}{S_0+\gamma}$$
7. "Local" constraint :
$$(\Sigma_T)[0] = \frac{M_0}{T_0+\gamma}$$
8. "Global" constraint :
$$((\Sigma_S)[i] - (\Sigma_S)[i-1])\,(S_i+\gamma) = 1$$
9. 'Global' query :
$$((\Sigma_T)[i] - (\Sigma_T)[i-1])\,(T_i+\gamma) = M_i$$

Fig. 1. Compilation of a vanilla inclusion query

When the table has multiple columns, we replace S and T by a random linear combination of the columns of S and T by powers of a distinct random coin field element α to reduce to an equivalent single-column case. This substitution only occurs in the construction of $\Sigma_{\{S,T\}}$ and the queries, beside that α is the only addition to the protocol.

Batching queries over the same table Sometimes we want to check the inclusion of multiple columns (or table) $S_0, S_1, S_2, S_3, \cdots S_{n-1}$ within the same column (or table). In that case, we can run a protocol variant that is more efficient than running the lookup technique many times in parallel. The protocol is described in Fig. 2.

Conditional inclusion: We extend the inclusion functionality to subsets of S and T, where the lookup function receives as additional arguments two boolean vectors A, B. Consider an ordering on sets $S = (s_1 \ldots s_n)$, and $T = (t_1 \ldots t_m)$, and define subsets $S_A = (s_i | A_i \text{ is true})$, $T_B = (t_i | B_i \text{ is true})$ and we want to

Inclusion($\{S_0, S_1, \ldots\}, T$)

1. The prover commits to a single M, which is the vector of multiplicities of the elements of T in all $S_0, S_1, S_2, \cdots, S_{n-1}$.
2. The verifier responds with a random coin field element γ.
3. The prover commits to $\Sigma_{S,k}$ for all S_k independently as in the original protocol.
4. The prover commits to a single Σ_T, interpolating the vector of partial sums (Σ_T) such that

$$(\Sigma_T)[i] = (\Sigma_T)[i-1] + \frac{M_i}{T_i + \gamma}$$

$$(\Sigma_T)[0] = \frac{M_0}{T_0 + \gamma}$$

5. The verifier follows uses a local opening query to open the last positions of each $\Sigma_{S,k}$ and Σ_T and checks that:

$$(\Sigma_S)[|S| - 1] == (\Sigma_T)[|T| - 1]$$

6. "Local constraint": for all k

$$(\Sigma_{S,k})[0] = \frac{1}{S_{k,0} + \gamma}$$

7. "Local" constraint :

$$(\Sigma_T)[0] = \frac{M_0}{T_0 + \gamma}$$

8. "Global" constraint, for all k:

$$((\Sigma_{S,k})[i] - (\Sigma_{S,k})[i-1])\,(S_k[i] + \gamma) = 1$$

9. 'Global' query :

$$((\Sigma_T)[i] - (\Sigma_T)[i-1])\,(T_i + \gamma) = M_i$$

Fig. 2. Multiple queries over the same column batched over the same T

test that the S_A is included in T_B. This approach avoids extra commitments to the filtered versions of the columns. Filters B on T can be dealt with in an efficient, elegant and minimal manner. We modify the T and S tables as follows: we append the filter B column as a column to the T table (for simplicity, imagine it is the last column of the modified T table. After appending B to table T, we append a corresponding column containing only values of 1 to the S-table (this is done using "verifier-defined" constant column equal to "1", which avoids introducing an extra commitment). It can be easily checked that the conditional inclusion holds if and only if an ordinary inclusion holds for the modified tables. Filters on S cannot be dealt with as above. What we opt for instead is to modify our formulas to take into account the A filter, in a way described in Fig. 3.

Fragmenting the column T We extend the inclusion query to allow Wizard protocol to specify an inclusion query where the table T is symbolically represented as the union of the rows of smaller tables $T_0, T_1, \ldots, T_n$. In this case, the compiler handles in an analogous way of handling multiple table ways S for a single table T. Namely, the compiler requires the prover to commit to multiple tables M and Σ_T, one for each table T_i. Each table $\Sigma_{T,i}$ is individually constrained as Σ_T in the vanilla protocol. At the end the verifier compares the sum of the final value of every of these tables instead of just the final value of the vanilla column Σ_T.

5.4.1 Security analysis sketch We initially sketch our analysis for the multicolumn case and a conditional filter on S and then highlight how the analysis can be extended to reflect the soundness of the batched and fragmented cases. We note t as the number of rows in T and s as the number of rows in S and then n denotes the number of columns. As highlighted earlier in the section, the predicate of the inclusion query is equivalent to an equality of rational multivariate functions:

Inclusion(S, A, T)

1. The prover commits to M, which is the vector of multiplicities of the elements of T in S_A (S_A is set S filtered on A).
2. The verifier responds with a random coin γ
3. The prover commits to Σ_S, a polynomial interpolating the vector of partial sums (Σ_S) such that

$$\Big((\Sigma_S)[i] - (\Sigma_S)[i-1]\Big)\Big(S_i + \gamma\Big) = A_i$$

$$(\Sigma_S)[0] = \frac{A_0}{S_0 + \gamma}$$

.
4. The prover commits to Σ_T, interpolating the vector of partial sums (Σ_T) such that

$$(\Sigma_T)[i] = (\Sigma_T)[i-1] + \frac{M_i}{T_i + \gamma}$$

$$(\Sigma_T)[0] = \frac{M_0}{T_0 + \gamma}$$

.
5. The verifier checks that

$$(\Sigma_S)[|S| - 1] == (\Sigma_T)[|T| - 1]$$

. He obtains the values of $(\Sigma_S)[|S| - 1]$ and $(\Sigma_T)[|T| - 1]$ via two local opening queries.
6. "Local" query :

$$(\Sigma_S)[0] = \frac{A_0}{S_0 + \gamma}$$

7. "Local" query :

$$(\Sigma_T)[0] = \frac{M_0}{T_0 + \gamma}$$

8. "Global" query :

$$((\Sigma_S)[i] - (\Sigma_S)[i-1])\,(S_i + \gamma) = A_i$$

9. The verier makes a "Global" query :

$$((\Sigma_T)[i] - (\Sigma_T)[i-1])\,(T_i + \gamma) = M_i$$

Fig. 3. Compiling a conditional inclusion query

$$\sum_{i \in [t]} \frac{M_i}{X + \sum_{j \in [n]} T_{i,j} Y^j} = \sum_{i \in [s]} \frac{A_i}{X + \sum_{j \in [n]} S_{i,j} Y^j}$$

It is worth noting that the S-batching case or the T-fragmentation case are equivalent to splitting respectively the right and the left hand of the equality. Thus, to analyze these cases, it suffices to consider the equivalent case where the S and T tables are unified in a single table. Coming back to our soundness analysis, observe that the described protocol perfectly simulates the evaluation of the above functional equality at random points (γ, α). Therefore, a malicious prover attacking soundness can only succeed if she comes up with an assignment such that the two rational functions are unequal but coincide on the random evaluation points (γ, α). The Schwartz-Zippel applied to rational function gives that the protocol has statistical soundness error $O(\frac{(t+s)n}{\mathbb{F}})$

5.4.2 Optimization When many lookups are declared over the same table, a possible optimization is to "group" multiple Σ_S into the same column. This reduces the number of committed columns through the compilation step at the expense of increasing the degree of the generated global constraints.

5.5 Reduction of the fixed-permutation checks

The technique we present is inspired by the work of [28] and [7]. Let n, m be integers and let σ be a permutation of $[n]$ and $A = \{A_i\}_{i\in[m]}$ and $B = \{B_i\}_{i\in[m]}$ such that B is obtained by permuting the rows of A according to σ. As σ is known beforehand, we give oracle-access to a signature of σ in an offline phase. This signature consists of two vectors $s = (1, \omega, \cdots, \omega^{n-1})$ and $s' = (\omega^{\sigma(1)-1}, \cdots, \omega^{\sigma(n-1)-1})$. Naturally, the same s and s' can be reused for different queries and since the polynomial encoding of s is $s(X) = X$ there is implicitly no need to send it to the oracle. The compiler then replaces every fixed permutation query on A and B by a permutation query on $A' = (A\|s)$ and $B' = (B\|s')$.

5.6 Reduction of the permutation checks

Let A and B be two columns of equal length l. The technique we present is borrowed from a series of works including [30], [7], [28] originating from the work of [9]. The intuition behind the protocol is as follows: a column A is a permutation of B if and only if the polynomials defined as $\prod_{i\in[l]}(X + A_i)$ and $\prod_{i\in[l]}(X + B_i)$ are equal. Our sub-protocol simulates an equality check between these two at a random point $X = \alpha$. Or equivalently

$$Z(\alpha) = \prod_{i\in[l]} \frac{\alpha + A_i}{\alpha + B_i} = 1$$

This approach can be extended to the case where A and B consist of multiple columns by substituting A and B in the above polynomial equality by linear combinations of the columns for each table by powers of a random independent coin r. Fig. 4 gives the pseudo-code of the Arcane compilation step for permutations. where the permutation function receives two sets of vectors $\{A_i\}_{i\in[m]}$ and $\{B_i\}_{i\in[m]}$ and highlights how Arcane converts a permutation check into local and global constraints.

Permutation$(\{A_i, B_i\}_{i\in[m]})$

1. if $m > 1$:
   - Verifier samples $r \leftarrow \mathbb{F}$ and sends it to the prover
   - The prover internally set $A' = \sum_i r^i A_i$ and $B' = \sum_i r^i B_i$

   else : they set $A' = A_0$ and $B = B_0$
2. Prover the column Z, defined as:

$$Z_i = \prod_{j\leq i} \frac{A' + \alpha}{B' + \alpha}$$

3. "Local" query : $Z_0 = 1$
4. "Global" query: $(Z >> 1)(B' + \alpha) = Z(A' + \alpha)$ (not cancelled on n-1)

Fig. 4. Reduction of a Permutation Check.

Fragmented table: The case where either the table A or B or both are fragmented is handled by splitting the rational product in several sub-products. A consequence is that the sub-protocol uses several sub-products. For optimization, we distinguish the following cases:

- There are fragments in A and B that have the same size, A', B': in this case, it is interesting to group them in a single product

$$\prod_{k\in[l]} \frac{A'_k + \alpha}{B'_k + \alpha}$$

.

– There are fragments on either side of the permutation A' or B' whose size does not match any fragment on the other side: in this case, they are individually assigned a product

$$\prod_{k\in[l]} A'_k + \alpha \quad \text{or} \quad \prod_{k\in[l]} \frac{1}{B'_k + \alpha}$$

In the sub-protocol description of Fig. 5, we assume that the mapping of the fragments $\{A_i\}, \{B_i\}$ to their respective product are clear from context and we also assume, for simplicity that the tables have a single column (the multi-column case is handled is in the vanilla case).

Permutation$(\{A_i, B_i\}_{i\in[m]})$

For each product, indexed by v and their respective fragments A' and B' or $\emptyset$ if it does not apply

1. Prover sends Z, the unique polynomial such that:

$$Z_v = \prod_{i\leq k} \frac{A'_i + \alpha}{B'_i + \alpha} \quad \text{or} \quad \prod_{k\in[l]} A'_k + \alpha \quad \text{or} \quad \prod_{k\in[l]} \frac{1}{B'_k + \alpha}$$

2. "Local" constraint: $Z_{v,0} = 1$
3. "Global" constraint: $(Z_v >> 1)(B' + \alpha) = Z_v(A' + \alpha)$
4. "Local opening": Z_v at the last position

Finally The verifier takes the final value of each Z_v and checks that their product equals 1.

Fig. 5. Reduction of a Permutation Check.

5.6.1 Security analysis sketch Let us start with the single-column case. A malicious prover may pass the verification if and only if $\prod_{i\in[l]}(X + v_{1,i}) \neq \prod_{i\in[l]}(X + v_{2,i})$ and the equality holds for $X = \alpha$. Indeed, the functional equality between the two products is equivalent to the predicate stated by the permutation query. Following this, we have that the above sub-protocol has statistical soundness error $\frac{n}{|\mathbb{F}|}$ by application of the Schartz-Zippel lemma. For the multi-column case, the predicate of the permutation query is equivalent to the following bivariate polynomials equality

$$\prod_{i\in[l]}(X + \sum_{j<\mathsf{ncol}} v_{1,i}Y^j) = \prod_{i\in[l]}(X + \sum_{j<\mathsf{ncol}} v_{1,i}Y^j)$$

From that point, applying the Schwartz-Zippel lemma gives us statistical soundness with error $O(\frac{l(\mathsf{ncol})}{\mathbb{F}})$. The same analysis can be applied to the more complex fragmented case by observing that the sub-protocol simulates the same evaluation as above for the union of all the tables on both sides.

5.6.2 Optimizations As for the reduction of the inclusion checks, in the fragmented case, it is possible to pack the construction of multiple grand products into a single column Z if they relate to columns of the same size. This optimization trades commitment complexity for an increase in the degree of the generated global constraints.

5.7 Reduction of the inner-product queries

As a reminder, inner-product queries allow the verifier to query the inner product of two committed columns A and B. We first explain how the inner-product reduction phase handles a single query concretely and then

explain how to extend it to handle multiple queries. In the outlines, the prover constructs a third column S storing the cumulative sum of the row-wise products of A and B. The verifier then queries the last entry of the S and expects it to equal the alleged value of the inner product as claimed by the prover.

InnerProduct$(A; B; c)$

1. The prover sends the columns A and B to the oracle and claims c.
2. The prover computes S such that $S_k = \sum_{j \leq k} A_j B_j$ and sends it to the oracle.
3. "Local": $S_0 = A_0 B_0$
4. "Global": $S_i - S_{i-1} = A_i B_i$ (except for $i = 0$)
5. "LocalOpening": the final position of S, c'
6. The verifier checks that $c == c'$

Fig. 6. Reduction of the inner product Check

Batching This technique can be extended to support multiple queries $(A_0, B_0, c_0), (A_1, B_1, c_1), \ldots, (A_n, B_n, c_n)$. To support this feature, we equip the protocol with a "batching" random coin r and we construct S as $S_k = \sum_{i \in [n]} \sum_{j \leq k} r^i A_{i,j} B_{i,j}$. The queries created by the compilation step are equivalently modified by swapping $A_i B_i$ terms by $\sum_i r^i A_{i,j} B_{i,j}$. Finally, the verifier compares c' with $\sum_i r^i c_i$.

5.7.1 Security analysis sketch In the single-query case, soundness and completeness are perfect as S is fully constrained to trace the inner product's step-by-step computation. In the multi-query case, S is perfectly constrained to compute the linear combination of the inner product of a and b. This has two consequences: (1) This justifies perfect completeness (2) The only way for a malicious prover to be successful with a false statement $(c''_0, c''_1, \ldots, c''_n)$ is that $\sum_{i \in [n]} r^i (c''_i - A.B)$ cancels. By the Schwartz-Zippel lemma, this can happen with probability at most $\epsilon = \frac{1}{|\mathbb{F}|}$. This justifies ϵ-statistical soundness.

5.8 Sticking the small columns together

The **sticking** compiler performs the opposite of the **splitting** step. Namely, it applies to all the columns whose size is smaller than some prespecified target size. These columns are regrouped into larger columns having the target size by interleaving. To illustrate, columns $A = (a_0, a_1, a_2, \ldots a_{n-1})$ and $B = (b_0, b_1, \ldots, b_{n-1})$, assuming they have half of the target size, would be regrouped into a column $L = (a_0, b_0, a_1, b_1, \ldots, a_{n-1}, b_{n-1})$. This compilation stage is reached much later than the **splitting** step. This is motivated by the fact that (1) this limits the overheads of the inclusion and the permutation stages (2) this simplifies the current compilation stage since it can assume that the incoming protocol does not use inclusion, range, permutation, inner-product or fixed-permutation queries as they are already compiled out at this stage. The sticking compilation stage processes the impacted queries as follows:

- **Local constraint and local opening**: the queries are replaced by equivalent ones pointing to the corresponding positions in the regrouped columns.
- **Global constraints**: the query is replaced by an equivalent one retaining the same expression but whose variables are replaced by the corresponding regrouped column, shifted by the appropriate offset and multiplied by a periodic sampling column.

The sticker also sets a lower bound on the columns it is working with. All columns smaller than this limit are sent directly to the verifier and the constraints applying to them are manually checked by the verifier. Passed this point, we may assume that all the column the protocol commits to are of the same size.

5.8.1 Example 1: With a global constraint Assume a Wizard-IOP protocol consisting of two columns $A = (a_1, a_2, a_3, a_4)$ and $B = (b_1, b_2, b_3, b_4)$ and say we apply the sticker compiler with a target size of 8. And assume a global constraint $Q_G =$ "Inclusion" : $A^2 = B$.

The "sticking" compiler step, replaces A and B by L and Q_G by Q'_G as

$$\begin{aligned} L &= (a_1, b_1, a_2, b_2, a_3, b_3, a_4, b_4) \\ I_1 &= (1, 0, 1, 0, 1, 0, 1, 0) \\ I_2 &= (0, 1, 0, 1, 0, 1, 0, 1) \end{aligned}$$

$$Q'_G = \text{"Global"} : (I_0 L)^2 = (I_1 L \ll 1)$$

5.9 Example 2: With a local constraint

Assume a Wizard-IOP protocol consisting of two columns A and B as above and a local constraint stating that $A[0] = B[0]$. Then L is constructed identically as above and the constraints Q_L is replaced by Q'_L as:

$$Q'_L = \text{"Local"} : L[0]^2 = L[1]$$

5.10 Example 3: With a larger target size

Let again reuse a Wizard protocol declaring A and B as above and this time assume we apply the "sticking" compiler with a target size of 16. In this case, A and B are not large enough together to fill a column. In this situation, we use virtual "zero columns" as fillings and construct

$$L = (a_1, b1, 0, 0, a_2, b2, 0, 0, a_3, b3, 0, 0, a_4, b4, 0, 0)$$

5.11 Reduction of the global constraints

We present a standard technique from the work of Plonk [7]. Let $v_1, \cdots, v_k$ be k columns and a k-variate arithmetic circuit $C(X_1, \cdots, X_k)$ of degree d. We denote by $v_\bullet(X)$ the polynomials encoding $v_\bullet$ in Lagrange basis. We have that the global constraint is satisfied if and only if there exists a polynomial $Q(X)$ of degree $(d-1)n$ such that,

$$C(v_1(X) \cdots v_k(X)) = (X^n - 1)Q(X)$$

Following this observation, the Arcane compiler runs the following procedure separately for each global query.

5.11.1 Optimizations For efficiency, the compilation of the global constraint also involves several optimization routines which we outline below:

1. The columns of types "expression" are swallowed into the global constraints using them as input variables.
2. To limit the degree of the expression, an optimization heuristically identifies frequent high-degree terms and assigns them to an intermediate column. This trades the degree of the constraints for commitment complexity.
3. The queries are grouped in buckets by degree and for each bucket, we replace them with a merged constraint which is satisfied if a random linear combination of the initial constraints vanishes.
4. The expressions are symbolically reduced using a symbolic expression simplifier (which performs several factorization routines)

GlobalConstraint$(C, v_1, \cdots, v_k)$

1. The prover computes and commits to Q computed as,
$$Q = \frac{C(v_1(X) \cdots v_k(X)}{X^n - 1}$$
2. The verifier samples a random coin $\alpha \leftarrow\!\!\$\ \mathbb{F}$
3. "Univariate" query : $v_1(\alpha) \ldots v_k(\alpha), Q(\alpha)$
4. The verifier checks
$$C(v_1(\alpha) \ldots) \stackrel{?}{=} (\alpha^n - 1)Q(\alpha)$$

Fig. 7. Reduction of the Global Constraints

5.11.2 Security analysis sketch We start by analyzing the compilation of a single constraint and then extend the analysis to cover the merging operation that we carry when dealing with several constraints. First of all, note that since the existence of the quotient such that

$$C(v_1(X) \cdots v_k(X)) = (X^n - 1)Q(X)$$

is satisfied is mathematically equivalent to the predicate of a global constraint, we have a that a malicious prover can only succeed if the equation does not hold as equality between polynomials but does hold on α. From the Schwartz-Zippel lemma, it follows that the protocol has statistical soundness with error at most $\frac{d}{|\mathbb{F}|}$). In the merged case, we also have to account for the possibility that the merged constraint is satisfied whereas some of the sub-constraints are not satisfied. This can happen with probability at most $\frac{n}{|\mathbb{F}|}$ where n is the number of constraints. Thus, in this case, the statistical soundness error is at most $\frac{n}{|\mathbb{F}|} + \frac{d}{|\mathbb{F}|}$

5.12 Reduction to a polynomial IOP

From this point on, the partially compiled Wizard-IOP only uses local constraints, local opening and univariate queries, possibly involving cyclically shifted columns. We now discuss on the last steps of the Arcane compiler to reduce to a polynomial-IOP, eliminating the references to shifted columns.

5.12.1 Local constraints and local openings are compiled out by converting them into equivalent univariate queries by interpreting the committed columns as polynomials expressed in Lagrange basis as we did during the global constraint reduction step. Accounting for the shifting is quite straightforward as it suffices to offset the opened point with the relevant number of positions.

5.12.2 Univariate queries over cyclically-shifted columns : it remains to discuss how to convert them into univariate queries "directly" on oracle-given polynomials (shown by P here). The conversion is performed following this observation:

$$(P \ll k)(x) = y \iff P(\omega^k x) = y$$

6 UniEval Compiler: from PIOP to UniEval PIOP

Let $\mathcal{P}$ be a PIOP protocol, where for $i \in [n], j \in S_i$, the verifier queries a polynomial P_i over a point x_j.

The aim of the compiler, presented here, is to reduce the initial polynomial-IOP to a polynomial-IOP where the oracle-given polynomials are all queried at a single random point. We will call such a polynomial-IOP scheme a UniEval polynomial-IOP, and the single query is denoted "Grail query". For any evaluation $P_i(x)$ where x is not the Grail query, the verifier gets $P_i(x)$ directly from the prover. In this model, replacing the oracle with a polynomial commitment scheme requires a proof of the evaluation for all the polynomials

at the same point i.e., over the Grail query. The gained advantage is that batching at the polynomial commitment level is now more straightforward as all the polynomials are queried on the same evaluation point. Indeed, due to this compiler, batching over different points is done at the polynomial-IOP level. At the polynomial commitment level, we only need batching over the same point. To build our compiler, we first present a batching technique of multiple polynomials over multiple points. We then use this protocol to compile any PIOP into a UniEval PIOP. This section briefly outlines the UniEval compiler presented in [13], we refer to this document for a more comprehensive description and security analysis.

6.1 Multiple-Point to Single-Point Reduction

We assume a set of points T and a set of n polynomials $\{i \in [n] : P_i(X)\}$, each of degree $d_i \leq d$. Each $P_i(X)$ is queried on a set of evaluation points $S_i \subset T$. Define $R_i(X)$ as the alleged evaluations of $P_i(X)$ over the set S_i, namely, $R_i(X)$ agrees with purported $P_i(X)$ over S_i (and $R_i(X)$ is of degree $|S_i|$). The aim is to present a protocol for the relation;

$$R := \{(S_i, R_i(X); P_i(X))_i \quad \forall i \quad P_i(X)|_{S_i} = R_i(X)|_{S_i}\} \tag{1}$$

Claim. The relation R holds if and only if:

$$\forall i \in [n] : (P_i(X) - R_i(X)) \prod_{x \in T \setminus S_i} (X - x) \text{ is divided by } \prod_{x \in T} (X - x). \tag{2}$$

Knowing this fact, in Fig. 8 we present our batching protocol for the relation Eq. (1). The protocol is inspired by the batching approach presented in [20].

$\mathsf{MPSP}(S_1, \cdots S_n, R_1, \cdots, R_n; P_1, \cdots, P_n)$

1. the prover sends oracle access to P_i.
2. The verifier samples $\alpha \leftarrow\!\$\, \mathbb{F}$.
3. The prover computes and sends oracle-access to:

$$Q(X) = \sum_{i \in [n]} \alpha^i \frac{P_i(X) - R_i(X)}{\prod_{x \in S_i} (X - x)}$$

4. The verifier samples $z \leftarrow\!\$\, \mathbb{F}$ and queries $P_1(z), \cdots P_n(z), Q(z)$.
5. Finally, the verifier checks that: relation in 3 is satisfied for $X = z$ i.e.,

$$Q(z) \prod_{x' \in T} (z - x') = \sum_{i \in [n]} \left(\alpha^i (P_i(z) - R_i(z)) \prod_{x'' \notin S_i} (z - x'') \right)$$

Fig. 8. Multi-point to single-point reduction procedure.

6.2 Compiler: P-IOP to UniEval P-IOP

We are now ready to compile a PIOP to its UniEval version.

– For any P-IOP, define its associated protocol P-IOP$'$ as follows; we let all the queries in P-IOP be sent directly to the prover, and let the prover respond to these queries (the prover replies with alleged values for the evaluations, without providing a proof at this stage, as that would be handled later in the protocol). Indeed P-IOP$'$ is the same as P-IOP where the prover also plays the role of the oracle by itself.

– By the end of an execution of P-IOP$'$, we get the trace of the polynomial queries issued during P-IOP$'$; the set of polynomials P_i, the points S_i, and the alleged evaluations of $P_i(X)$ over S_i which we denote by $R_i(X)$ (prover's responses).
– Now, we consider our multi-point to the single-point protocol in Fig. 8, for the statement (R_i, S_i) and the witness $P_i(X)$ from the trace. Call this protocol $\text{MPSP}(R_i, S_i; P_i(X))_i$.

The compiler first runs P-IOP$'$, get the trace, and then runs $\text{MPSP}(R_i, S_i; P_i(X))_i$. The resulting PIOP is what we call UniEval-PIOP, denoted by UniEval-PIOP.

Knowledge-Soundness. Let $\epsilon_{\text{UniEval}}$, $\epsilon_{\text{P-IOP}'}$ and ϵ_{MPSP} be, respectively, the soundness-error of protocols UniEval-P-IOP, protocol P-IOP$'$ and $\text{MPSP}(R_i, S_i; P_i(X))_i$. Then, we have, $\epsilon_{\text{UniEval}} \leq \epsilon_{\text{P-IOP}'} + \epsilon_{\text{MPSP}}$.

7 Vortex: a List Polynomial Commitment

This section briefly presents the Vortex List Polynomial Commitment scheme introduced in [13] and how we instantiate it as part of Linea using a lattice-based hash (ring-SIS) hash function. Vortex is a variant of the commitment scheme proposed in Orion [49] and Brakedown [32], and it relies on a lattice-based hash which we describe in Section 7.1, and an erasure-code and the MiMC hash function [5].

In this work, we use the systematic version of the Reed-Solomon code which has encoding time $O(N \log N)$, where N is the size of the codeword. Vortex allows to perform a batched argument of multiple committed polynomials evaluated over the same given point x. One of the main differences here is the way we treat not just one polynomial but a batch of polynomials. In Breakdown and Orion, they assume a large degree polynomial and fold it into a matrix, while here we assume each row of the matrix is a separate polynomial. This is beneficial for our use case (zkEVM) where we have to deal with many polynomials at once. Another difference is, that we discuss the security not in the unique decoding regime, but in the list decoding regime. This point helps us to improve the efficiency of the scheme but brings some challenges regarding the security proof and for the PIOP transformation into AoK (PIP) through the Vortex commitment which we will address later. Vortex is described in Section 7.2. Vortex additionally comes with a batching argument allowing to open simultaneously multiple matrices at the same point.

For a matrix of size $m \cdot n = N$, the Vortex commitment and opening arguments have size $O(\sqrt{N})$. Moreover, the opening arguments have verification time $O(\sqrt{N})$. In Section 9, we present our *self-recursion* technique to achieve succinctness.

7.1 Lattice-Based Hash

The lattice-based hash function we present relies on the Ring-SIS assumption to achieve collision resistance. The design of our hash function is essentially the same as the SWIFFT [39] hash function. The only concrete difference is that the design of SWIFFT restricts the input set of the Ring-SIS inputs to be $\{0, 1\}$ while our hash function accepts an input set of the form $[0, 2^n - 1]$ (for small n).

Let q be a prime, $\mathbb{F}_q$ be the finite-field, b a power of two such that $b < q$ and d, m two positive integers such that d is a power of 2 and $m > \frac{\log q}{\log b}$. We consider the ring $\mathcal{R} = \frac{\mathbb{F}_q[X]}{X^d+1}$ of polynomials whose coefficients lie in $\mathbb{F}_q$ modulo $X^d + 1$. To instantiate the hash function, we need first to go through a transparent setup phase where a Ring-SIS key is sampled. We set $N = md\frac{\log b}{\log q}$. A description of the procedure is given in Fig. 9

Collision and preimage resistance are derived from the Ring-SIS and the Ring-ISIS[2] problems respective to the instances (q, m, b).

If $(q-1)|(n+1)$, the scalar product of $L \cdot A$ may be computed with the following procedure. Let $\bar{\omega} \in \mathbb{F}_q$ such that $\bar{\omega}^n = -1$. Note that $\{\bar{\omega}^{2i+1}\}$ forms a coset of the n-th roots of unity that all vanishes under $X^n + 1$. We can efficiently compute the evaluations of L_i and precompute the one for A_i using the Cooley-Tuckey

[2] Inhomogeneous SIS

$\mathsf{Setup}(q, m, d, b) \to \mathsf{pp}$

1. $\mathsf{A} = (A_i)_{i<m} \leftarrow\!\!\$\ \mathcal{R}^m$
2. return $\mathsf{pp} = \mathsf{A}$

$\mathsf{Hash}(\mathsf{x} \in \mathbb{F}_q^N)$

1. Encode each element of x in $\log q / \log b$ limbs l_i, such that $\|l_i\| < b$ for all i.
2. Arrange the limbs l_i as coefficients of polynomials to obtain a vector $L = (L_i)_{i<m} \in \mathcal{R}^m$
3. Compute the scalar product $h = A \cdot L$ (requiring polynomial multiplication in $\mathcal{R}$)
4. return h by returning its coefficients

Fig. 9. Description of the lattice-based hash

algorithm (also known as FFT, or NTT in the literature). In this basis, the multiplication of polynomials coincides with the Hadamard (entry-wise) product, and we can get h directly in evaluation before switching back to coefficient basis in the end. Overall, the complexity of the hashing procedure is $O(mn \log n)$. For small values of n and b, other techniques such as Tom-Cook are known to be efficient as well. In Appendix B we recap the security analysis of this hash function and give concrete parameters for a target level of security.

7.2 Description of Vortex

In this subsection, we expand on the details of Vortex. We will first assume two integers m and k, denoting the number of rows and columns.

Let $\mathcal{H}$ be our hash function (Section 7.1) parameterized to be able to hash vectors of size (at least) m. We also use a *systematic*[3] Reed-Solomon $\mathcal{L}$ with message size k and codeword-size $n > k$. We denote its encoding algorithm $\mathsf{encode}_{\mathcal{L}}$. Vortex consists of three algorithms: Setup, Commit, and OpenEval.

1. Setup is a transparent offline phase run by both the prover and verifier. During this phase, they perform precomputations involving sampling the parameters for the hash and the encoding scheme used as the *public parameters*.
2. The Commit algorithm: Let W, be the matrix whose i^{th} row is $w_i \in \mathbb{F}^k$. Thus, W has m rows and k columns. The prover encodes each row of W (noted by w_i) using the encoding function and obtains W' (which has n columns).[4] The prover then computes the hash of the columns. The value $H = h_1, \cdots, h_n$ forms the *commitment*.
3. The batch-opening phase or OpenEval is an interactive protocol where the prover runs the ProveOpening algorithm and the verifier runs the VerifyOpening. At the beginning of this phase, the prover holds W, W' and the verifier holds the *final commitment* as input. Both hold the statement x, y. The prover's goal is to convince the verifier that for $\forall i < m, w_i(x) = y_i$, if W is a batch of polynomials and their evaluations stand in Lagrange basis. The verifier then sends the random scalar α, and the prover responds with u claimed to be $u := \boldsymbol{\alpha}^\top W$, if W is polynomial, where $\boldsymbol{\alpha} = (1, \alpha, \alpha^2, \ldots, \alpha^{m-1})$. Then, the verifier samples t columns $q_1, \cdots q_t$ ($q_i \leq n$) uniformly at random, and the prover responds with $(s_1 \cdots s_t)$ chosen columns of W'. The verifier computes u' as the Reed-Solomon encoding of u and performs the following checks for all opened columns:
   - **Proximity Check:** the scalar-product $\boldsymbol{\alpha}^\top s_i \stackrel{?}{=} u'_{q_i}$
   - the hash of s_i is correct and consistent with h_{q_i}.
   - **Evaluation Check:** the relation $u(x) \stackrel{?}{=} \boldsymbol{\alpha}^\top \cdot y$ where the vector u is considered as the coefficient of polynomial $u(x)$

[3] This means the original block should be a sub-vector of the corresponding codeword. By "checksum", we refer to the part of a codeword, that is added beside the original block.

[4] Observe that, since the encoding procedure $\mathsf{encode}_{\mathcal{L}}$ is systematic, we have that all columns W are also columns of W'.

The first check (the random combination over random columns), is used for checking the proximity of a batch in [6]. Fig. 10 sums up the above.

$\mathsf{Setup}(n, m, \mathcal{L}, \lambda) \to \mathsf{pp}$

1. Setup an instance of hash, Hash, corresponding to the security level λ
2. Choose t (the number of columns that should be opened later) to reach the security level λ
3. Runs pre-computations relative to $\mathsf{encode}_{\mathcal{L}}$ (e.g., finding $D \subset \mathbb{F}_q$ and relevant parameters for the security level λ)
4. Collect all the computed parameters in pp and return it.

$\mathsf{Commit}(\mathsf{pp}, W) \to (h_1 \cdots h_n)$

1. Encode each row of W and obtain W'
2. Hash each column of W' to obtain $(h_1 \cdots h_n)$
3. Return $(h_1 \cdots h_n)$

$\mathsf{OpenEval}$ with statement (l, y)

$\mathsf{ProveOpening}(\mathsf{pp}, W', x, y)$		$\mathsf{VerifyOpening}(\mathsf{pp}, H, x, y)$
	$\leftarrow \alpha$	
$u = \boldsymbol{\alpha}^\top W$		
	$u \rightarrow$	
		$\mathbf{q} \leftarrow\!\!\$\ [n]^t$
	$\leftarrow \mathbf{q}$	
	$(s_1, \cdots, s_t) \rightarrow$	
		$u' \leftarrow \mathsf{encode}_{\mathcal{L}}(u)$
		for $0 < i \leq t$:
$W' = \mathsf{encode}_{\mathcal{L}}(W)$		
Denote the columns of W' as $(s_1 \cdots s_{n'})$		
		$\langle s_i \vert \boldsymbol{\alpha} \rangle \stackrel{?}{=} u'_{q_i}$
		$\mathsf{Hash}(s_i) \stackrel{?}{=} h_{q_i}$
		$u(x) \stackrel{?}{=} \boldsymbol{\alpha} \cdot y.$

Fig. 10. Vortex Polynomial commitment

Security of Vortex. The security of Vortex is thoroughly analyzed in [13], here we restate the results of this work. Vortex satisfies the knowledge-soundness of (batched) list polynomial commitment given in Definition 12 in the Random Oracle Model and has a straight line extractor. Writing $F(x) = \mathsf{MiMC}(\mathsf{SIS}(x))$ we heuristically model F as a random oracle and use it to hash the columns. The intuition is that MiMC, being itself modellable as a random oracle, breaks down all the linear properties of the SIS hash.

List Polynomial Commitment for Long Polynomials Here we show how to build a polynomial commitment from Vortex. The prover $\mathcal{P}$ can send a polynomial P whose degree is larger than the number of columns in W. The polynomial can be folded in several chunks $P(X) = P_0(X) + X^n P_1(X) + \cdots$. Each one of the chunks $P_i(X)$ is then inserted into W as an entire row. To commit and open the polynomials $P(X)$ via Vortex, set W as above. The verifier can then recombine the $P_i(X)$ evaluations to obtain the $P(X)$ evaluation. This allows us to switch between the definitions of batched polynomial commitments to a version that only commits to one polynomial. This fact is provided as a side note and is not used in Linea's

prover since the Arcane compiler is already responsible for splitting polynomials. Besides, our work stands for polynomials in Lagrange basis.

7.3 Constant-size commitment

As a simple optimization over the commitment size, we apply a SNARK-friendly hash function (e.g., MiMC hash or Poseidon) over each h_i and then compute a Merkle tree over the results. This is particularly useful for compiling PIOP to AoK via Vortex since the Vortex commitment phase will not be offline anymore and will be part of the proof. It is important to note, however, that the hash function used in constructing the Merkle tree needs to be modelled as a random oracle for the scheme to retain extractability.

7.4 Batch opening for several batches of Polynomial

As shown above, Vortex allows committing to several polynomials (forming a matrix) at once. This can be extended to a "multi-batch" opening protocol in which the prover argues the evaluation of multiple batches (which may have been committed to at different moments) simultaneously at the same evaluation point. In this batched protocol, the prover sends only a single linear combination of all the rows of all the matrices as opposed to one per committed matrix in the non-batched setting. Additionally, during the column opening phase, the verifier queries the same columns position for all the matrices. This is the version of the protocol that we actually use to instantiate an Argument of Knowledge.

8 AOK from PIOP and Vortex

This section briefly discusses the compiler described in [13] and compares it with related works. For a in-depth description of the compiler and its analysis, we refer to [13].

In [23, appendix E], combining a knowledge-sound polynomial commitment with knowledge-sound PIOP results in a knowledge-sound argument system. This can not be applied directly to our setting. Particularly, since we are working with polynomial commitments in the list decoding regime (LPC), the knowledge-soundness of Vortex is not defined w.r.t a standard relation for a PC scheme. In KVP22 [36], they show that Batch-FRI in the list-decoding regime (as an LPC) can be combined with PLONK-PIOP resulting in an argument system. Here we generalize their result and show that any PIOP can be combined with an LPC. There is some evidence that shows that such a transformation can still be possible for special PIOP and with the cost of losing a factor $|L|$ of the soundness of PIOP [36, 17]. Slightly more formally, let (P_O, V_O) be a PIOP for the relation $\mathcal{R}$ that is transformed to an AoK (P, V) via a list polynomial commitment (P_c, V_c). Then it is conjectured that the soundness of AoK follows from the following,

$$|L|.\epsilon_{\text{PIOP}} + \epsilon_{\text{LPC}} \approx O(|L|.k^c/|\mathbb{F}| + \epsilon_{\text{LPC}})$$

where k is the degree of polynomials involved in PIOP and $|L|$ is the maximum size of the list associated with LPC. If the size of the field is big compared to $|L|$ working in the list decoding regime can be beneficial. Here we give our proof intuition asserting the above conjecture is true. The proof is detailed in [13].

Note that for LPC, a cheating prover can leverage the list decoding property to commit to a list and later decide which one to evaluate. This means for each round, the prover may use a different agreement set (or visually a different list index), this would increase the soundness loss to $|L|^r/|\mathbb{F}|$. We force the prover to use the same agreement set over all the rounds by applying the evaluation protocol of Vortex over the concatenation of all the matrices from different rounds. This means that while for each round the commitment is applied over the relevant matrix, the opening is applied over a bigger matrix which is the concatenation of all the matrices. By this technique, we succeed in keeping the soundness loss to $|L|r/|\mathbb{F}|$.

8.1 Compiler sketch

As a reminder, the compiler starts from a UniEval P-IOP which is structured as follows. At every round and possibly once offline, the prover sends a batch of polynomials to the oracle and short messages to the prover and the protocol concludes on a Grail-query where all polynomials are queried at the same point. For simplicity, we will account for the offline precomputed polynomials as being part of the round -1 and suppose that all the polynomials involved in the protocol have the same size (this is ensured by the Arcane compiler already). Here, we use the variant of Vortex that uses Merkle roots of the ring-SIS hashes as commitment. For each of these batches W_h of m_h polynomials, the prover computes using Commit sends a Vortex commitment's Merkle root Root_h. Fig. 11 expands on how our compiler simulates the oracle. Note that all randomnesses used in the protocol are still managed as interactive messages between the prover and the verifier of $\mathsf{UniEval}$ and the compilation does not touch these.

$\mathsf{AoKCompiler}(\mathsf{UniEval})$

$\mathsf{AoKSetup}(\lambda, \mathcal{L}, n)$

---

1. $\mathsf{pp}_{\mathsf{UniEval}} \leftarrow \mathsf{UniEvalSetup}(\lambda)$
2. Set m as the size of the largest batch of polynomials found in $\mathsf{UniEval}$'s description.
3. $\mathsf{pp}_{\mathsf{Vortex}} \leftarrow \mathsf{VortexSetup}(\lambda, \mathcal{L}, m, n)$
4. Set W_{-1} as the matrix constructed by taking all the precomputed polynomials and commits to it to obtain Root_{-1}
5. $\mathsf{pp}_{\mathsf{AoK}} \leftarrow (\mathsf{pp}_{\mathsf{UniEval}}, \mathsf{pp}_{\mathsf{Vortex}}, \mathsf{Root}_{-1})$
6. **return** $\mathsf{pp}_{\mathsf{AoK}}$

$\mathsf{CommitmentPhase}(\mathsf{pp}_{\mathsf{AoK}}, W_h)$

---

1. Prover computes $\mathsf{Root}_h \leftarrow \mathsf{VortexCommit}(\mathsf{pp}_{\mathsf{Vortex}}, W_h)$ and sends it to the verifier

$\mathsf{OpeningPhase}(\mathsf{pp}_{\mathsf{AoK}}, \mathsf{Root}_{-1,\ldots,n_{\mathsf{round}}}, x, y)$

---

1. The prover and the verifier engage in the batch opening protocol of Vortex for all the rounds at once.

Fig. 11. From UniEval P-IOP to AoK using Vortex LPC

9 Self-Recursion of Vortex

As Vortex proofs are large, to obtain a SNARK, we compress the proof via a self-recursion technique where instead of opening the chosen columns $(s_1, \ldots, s_t)$ and sending them to the verifier, the prover computes the hashes and the scalar-products itself (while the verifier has oracle access to u' and the hash outputs). It sends proofs for the following facts:

- The encoding $\mathsf{encode}_{\mathcal{L}}(u)$ is correctly computed as u': in our self-recursion protocol we work directly with the Reed-Solomon encoded column. It remains to show that u' is a Reed-Solomon codeword and this is addressed in Section 9.1.2
- The checks $u(x) \stackrel{?}{=} \boldsymbol{\alpha}^T y$ is addressed in Section 9.1.2, it leverages the fact that $u'(x) = u(x)$ in our settings.
- The opened columns are consistent with u' as $\boldsymbol{\alpha}^T s'_i = u'_{q_i}$: this is addressed in Section 9.4.
- The opened column hashes are included in the commitment trees at the requested positions. This includes the correctness of the SIS hashing operation from the verifier. This is addressed in Section 9.2 and is the

most involved part of the section. And additionally, the verifier is required to verify Merkle proofs to assess the membership of the computed SIS hashes in the trees. This later part is addressed in Section 9.3.

For each of the above relation, a dedicated PIOP protocol is designed. Concretely, the process of *self-recursion* transforms Vortex into a Wizard-IOP in which the prover sends oracle access to the relevant messages instead of sending them to the verifier directly (including the columns, all hash values and vector u of the Vortex commitment).

The verifier is then tasked to perform a few queries so that he can convince himself that the prover's messages add up to an accepting transcript. The resulting protocol can then be recompiled using the Arcane compiler (developed in Section 5) and Vortex Section 7 and we can reiterate this process by reusing different instances of Ring-SIS and Reed-Solomon codes. This technique allows us to play with the tradeoff that we have when choosing the Ring-SIS parameters and the erasure code. Typically, Ring-SIS instances that use a large modulus degree compress poorly but are very fast to run. On the other hand, Ring-SIS instances with a small modulus degree compress very well but are slower to run. This creates a trade-off between the prover time on one side and the verifier time and proof size on the other. The self-recursion strategy allows us to use Ring-SIS instances with a large degree for the initial steps and progressively reduce the degree. Similarly, we can use an error-correcting code with a large rate (and small relative distance) at the beginning and progressively decrease the rate as we loop into more applications of the self-recursion process.

SNARKs from Arguments of Knowledge Consider the AOK presented in Section 8. After applying multiple steps of self-recursion on Vortex, the proof achieves succinctness, and it is possible to finalize it into a SNARK using the Fiat-Shamir transform. Note that each of the cycles of self-recursion occurs in the interactive world, thus the Fiat-Shamir transform is applied only once at the end. The final SNARK has a proof size and verifier time of $O(\log\log(N))$ and linear prover time and the final Vortex proof stands for a witness of size $O(1)$. While the asymptotic performances are good, the concrete size of the final proof does not allow it to be directly verified on Ethereum. To remediate, we recursively composite it with the Plonk [7] proof system. As an optimization, we use MiMC hashes [5] instead of the ring-SIS based hash function that we present in Section 7.1. The motivation is that, while MiMC's runtime is slower than the lattice-based hash, the cost of emulating the computation within a Plonk circuit can be made drastically cheaper thanks to the optimizations of [12] which uses the GKR [31] proof system to reduce the cost of proving the hashes. This optimization is also ported to the Fiat-Shamir hashes in the circuit.

9.1 Protocol description

In the following, we assume an initial interactive protocol output by the Vortex compiler using Merkle-tree hashing for shorter proofs and thus, with the following structure: n_{round} rounds of interactions initiated by the prover where he sends a Merkle-root and possibly several queries result and the verifier responds with a sequence of random coins and the protocol concludes on a Vortex proof opening. For simplicity, we make a small technical assumption that will be helpful for the self-recursion process: the number of committed polynomials per round is a multiple of $K = \frac{n \log_2 \beta}{\log_2(|\mathbb{F}|)}$, when $K > 1$ where n is the output-size of the SIS instance, $\log_2 \beta$ the bit size of each limb and $\mathbb{F}$ the underlying prime field. This can be ensured by adding zero polynomials at every round until this is reached. For reference, we use the following variable notations:

- n_{col}: the number of columns in the committed (and unencoded matrix). This matches the size of the polynomials we initially committed to using Vortex.
- n_{row}: the total number of rows, all rounds included and including the precomputation rows.
- n_{rounds}: the number of interaction rounds before the Vortex opening starts in the input protocol
- t: the number of opened column during the Vortex opening

9.1.1 Casting the Vortex proof into Wizard column As outlined in the introduction of this section, the initial step of the self-recursion is to convert the Vortex proof messages into committed Wizard columns. Since the individual commitments are already short, they remain prover messages but they will be used as part of the protocol. The same goes for the alleged evaluations y, they remain visible to the verifier but are used within the self-recursion protocol under the hood of a "verifier-defined" column which we name Y. We list the columns below:

- Root_h: columns of size 1, tagged as "proof messages"
- Y: a "verifier-defined" column storing all the alleged evaluations of the Vortex opening.
- U_α: the random linear combination of the rows of the committed matrices, directly in encoded form.
- Q: a column storing the indices of the opened columns in the protocol as sampled by the verifier. This column is "verifier-defined".
- $\mathsf{Preimage}_i$: the opened columns of the matrices stacked round by round and limb-decomposed. Namely, $\mathsf{Preimage}_0$ contains concatenated values of the first opened columns for all committed round matrices.
- MerkleProofs: the Merkle-proofs used to justify the membership of the MiMC hashes of the SIS hashes of the opened columns.

9.1.2 The linear combination of the rows As part of the opening phase of Vortex, the prover sends U_α, storing the linear combination of the rows of the committed matrix (encoded) by powers of a random coin α (β in the Vortex section). And the verifier is willing to ensure that:

1. $U_\alpha(x) = \mathcal{A}^T y$ where $\mathcal{A} = (1, \alpha^1, \alpha^2, \ldots, \alpha^{n_{\text{row}}-1})$ (noted $\mathcal{B}$ in the Vortex description)
2. U_α is a Reed-Solomon codeword

The first check can be carried by calling the sub-protocols presented in Appendix A.1 and Appendix A.2 and the Reed-Solomon membership can be taken by a probabilistic argument. Reed-Solomon membership is equivalent to proving low-degreeness of a polynomial described a U_α in Lagrange basis. As part of this argument, the prover exhibits a column U_{coeff} storing the coefficient of the polynomial represented by U_α in Lagrange form. Note that the size of U_{coeff} is shorter than the size of U_α. The prover and the verifier then engage in an argument to check that U_{coeff} and U_α evaluate to the same value at a random point. Fig. 12 details this sub-protocol.

$\mathsf{ReedSolomon}(\mathsf{U}_\alpha)$

1. Prover commits to U_{coeff}
2. Verifier samples and sends a random coin r
3. The prover responds with an alleged evaluation point $y = U_{\mathsf{coeff}}(r) = U_\alpha(r)$ where the first evaluation is in monomial basis and the second one in Lagrange basis.
4. The prover and the verifier engage in $\mathsf{MonomialEval}(U_{\mathsf{coeff}}, r, y)$ as in Appendix A.1
5. The prover and the verifier engage in $\mathsf{LagrangeEval}(U_\alpha, r, y)$ as in Appendix A.2.

Fig. 12. Reed-Solomon membership check

9.2 SIS hashing of the columns

As part of the column opening phase, the verifier is tasked to computes the $t \cdot n_{\mathsf{round}}$ SIS hashes, one for each of the opened columns as sent by the verifier. We present an outline of how the sub-protocol emulating this part is constructed. To clarify our settings, recall that each column $\mathsf{Preimage}_i$ stacks the limbs of the opened column q_i for all the rounds $0 \ldots n_{\mathsf{round}} - 1$. The protocol unfolds as follows:

1. All $\mathsf{Preimage}_{0\ldots t-1}$ are range-checked. Note that, at this point, the only thing that remains to be proved is the linear part of verifying SIS hashes.

2. We declare the precomputed columns $A_0, \ldots A_h, \ldots A_{n_{\mathsf{round}}-1}$, the SIS key shards. They are constructed by taking A the SIS commitment key of Vortex and shifting it so that the first coefficient of A lines up with the first row of $\mathsf{Preimages}_\bullet$ corresponding to round h and all rows of A_h corresponding to rows that are outside of round h in $\mathsf{Preimages}_\bullet$ are zeroized.

3. The prover initiates by sending a committed columns $\mathsf{ConcatenatedDhQ}$ storing the concatenation of the t SIS hashes of the t opened column for every round $h < n_{\mathsf{round}}$, with the following ordering ordered as $\mathsf{ConcatenatedDhQ}_{it+j} = d_{i,j}$ where $d_{i,j}$ denotes the SIS hash for round i and the opened column j.

4. From this point, the prover and the verifier engage in a batching argument. The verifier 2 random coins that we name r_{merge} and r_{collapse}. The philosophy of this step is that r_{collapse} is used to combine items across the t linearly opened columns together, r_{merge} to linearly combine items across the n_{round}. [5]

5. The prover and the verifier engage in a folding protocol of Appendix A.4 over $\mathsf{ConcatenatedDhQ}$, a first time over r_{merge} to obtain a column $\mathsf{Dh}_{\mathsf{merge}}$ and a second time over r_{collapse} to finally obtain $\mathsf{Dh}_{\mathsf{merge,collapse}}$ representing a linear combination of all the individual SIS digests. It is tagged as a "proof" message.s

6. The prover and the verifier set A_{merge} and $\mathsf{Preimage}_{\mathsf{collapse}}$ as aliases for the linear combinations of the columns A_h by powers of r_{merge} and of the columns $\mathsf{Preimage}_i$ by r_{collapse}

7. At this point, we can reinterpret the columns A_{merge} and $\mathsf{Preimage}_{\mathsf{collapse}}$ as vectors of ring elements $\mathbf{A} = (A_0(X), A_1(X), \ldots)$ and $\mathbf{P} = (P_0(X), P_1(X), \ldots)$ in the SIS ring $\mathcal{R}$, expressed in coefficient form and concatenated within their respective columns. Similarly, $\mathsf{Dh}_{\mathsf{merge,collapse}}$ can be viewed as a ring element $\mathbf{D}$ in $\mathcal{R}$. The rest of the protocol aims at proving that $\mathbf{D} = \mathbf{A}^T\mathbf{P}$ where the inner-product is taken modulo $X^d + 1$. We start from the following observation: let $P(X)$ be a polynomial of degree $< 2d$ and let $Q(X) = P(X) \mod X^d + 1$ and $R(X) = P(X) \mod X^d - 1$, then we have:

$$2P(X) = (X^d - 1)Q(X) + (X^d + 1)R(X)$$

   The intent will be to test this equality at a random point. Concretely, the prover sends a column E_{dual} as "proof", containing the coefficient of the polynomial $\mathbf{A}^T\mathbf{P} \mod X^d - 1$

8. The verifier samples a coin r_{fold} and engages with the prover in the folding protocol of Appendix A.4 for both A_{merge} and $\mathsf{Preimage}_{\mathsf{collapse}}$ to obtain $A_{\mathsf{merge,fold}}$ and $\mathsf{Preimage}_{\mathsf{collapse,fold}}$. The verifier then queries the inner product of these two folded polynomials to obtain p.

9. Finally, the verifier checks that

$$2p \stackrel{?}{=} (r_{\mathsf{fold}}^d - 1)\mathsf{Dh}_{\mathsf{merge,collapse}}(r_{\mathsf{fold}}) + (r_{\mathsf{fold}}^d + 1)E_{\mathsf{dual}}(r_{\mathsf{fold}})$$

9.3 Inclusion of the SIS hashes in the Merkle roots

This section briefly outlines how the sub-protocol responsible for hashing the SIS hashes into Merkle leaves and proving inclusion in the commitments Merkle roots. We start from the columns Q storing the indices of the opened positions, Rooth each storing the roots of all the opened columns. We also reuse the column $\mathsf{ConcatenatedDhQ}$ from the SIS hashing phase, containing the concatenated SIS hashes computed by the verifier.

[5] In fact, in our implementation we use $r_{\mathsf{merge}} = r_{\mathsf{collapse}}^t$

1. We first invoke the sub-protocol for Appendix A.7 in order to instantiate a column Leaves storing the hashes of the SIS hashes.
2. Then, we construct the columns $\mathsf{MerkleRoots}$ and $\mathsf{MerklePositions}$ using two fixed-permutation query from a verifier-defined column constructed from Root_h and from Q. These will be used to construct the statement of the Merkle module Appendix A.8
3. We conclude the sub-protocol by invoking the Merkle module using the $\mathsf{MerkleProofs}$ column.

9.4 Consistency of the opened columns with $U_{\boldsymbol{\alpha}}$

This section briefly outlines how the sub-protocol emulating the check $\boldsymbol{\alpha}^T s_i \stackrel{?}{=} u'_{q_i}$. Our starting point is the columns $\mathsf{Preimages}_i$ storing the values of s_i in limb-decomposed form and the column U_α. The verifier also has access to Q, "verifier-defined" storing the opened positions of the column opening phase.

1. The first step of the sub-protocol aims at deriving a column $U_{\alpha,q}$ from U_α and Q, storing the entries of U_α at the positions indexed by Q. This is implemented using an inclusion query and a precomputed table I storing all the integers up to n_{encoded} in increasing order $(0, 1, 2, 3, \cdots, n_{\mathsf{encoded}} - 1)$. The verifier queries
$$\text{"Inclusion"}: \quad (Q, U_{\alpha,q}) \subset (I, U_\alpha)$$
2. Then, it remains to ensure that the linear combination of $\mathsf{Preimages}_i$ (recomposed in field elements) with $\boldsymbol{\alpha}$ matches the entries $U_{\alpha,i}$. To this end, we use the coin r_{collapse} and the column $\mathsf{Preimages}_{\mathsf{collapse}}$ and we use the sub-protocols in Appendix A.1 and Appendix A.3 to check:
$$\mathsf{MonomialEval}(U_{\alpha,q}, r_{\mathsf{collapse}}) = \mathsf{BivariateEval}(\mathsf{Preimages}_{\mathsf{collapse}}, 2, \alpha)$$

Acknowledgements

We are grateful to the Linea Prover team, zkEVM, Gnark team, and Nicolas Liochon for their feedback and useful discussions on the paper.

References

[1] *A native zkEVM Layer 2 Solution for Ethereum.* URL: `https://scroll.io/`.
[2] Miklós Ajtai. "Generating Hard Instances of Lattice Problems". In: *Electron. Colloquium Comput. Complex.* TR96 (1996).
[3] Martin R. Albrecht, Rachel Player, and Sam Scott. "On the concrete hardness of Learning with Errors". In: *Journal of Mathematical Cryptology* 9 (2015), pp. 169–203.
[4] Martin R. Albrecht et al. "Estimate all the {LWE, NTRU} schemes!" In: *IACR Cryptol. ePrint Arch.* 2018.
[5] Martin R. Albrecht et al. "MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity". In: *ASIACRYPT*. Vol. 10031. LNCS. 2016, pp. 191–219.
[6] Scott Ames et al. "Ligero: Lightweight Sublinear Arguments Without a Trusted Setup". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017).
[7] Zachary Ariel Gabizon, J. Williamson, and Oana Ciobotaru. "PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge". In: *IACR Cryptol. ePrint Arch.* (2019), p. 953.
[8] Shi Bai et al. "Improved Combinatorial Algorithms for the Inhomogeneous Short Integer Solution Problem". In: *J. Cryptol.* (2019).
[9] Stephanie Bayer and Jens Groth. "Efficient Zero-Knowledge Argument for Correctness of a Shuffle". In: *EUROCRYPT*. 2012.
[10] Olivier Bégassat et al. *A ZK-EVM specification.* 2022.

[11] Alexandre Belling, Azam Soleimanian, and Olivier Bégassat. "Recursion over Public-Coin Interactive Proof Systems; Faster Hash Verification". In: *IACR Cryptol. ePrint Arch.* (2022), p. 1072. URL: https://eprint.iacr.org/2022/1072.

[12] Alexandre Belling, Azam Soleimanian, and Olivier Bégassat. *Recursion over Public-Coin Interactive Proof Systems; Faster Hash Verification*. Cryptology ePrint Archive, Paper 2022/1072. 2022. URL: https://eprint.iacr.org/2022/1072.

[13] Alexandre Belling, Azam Soleimanian, and Bogdan Ursu. *Vortex: A List Polynomial Commitment and its Application to Arguments of Knowledge*. Cryptology ePrint Archive, Paper 2024/185. https://eprint.iacr.org/2024/185. 2024. URL: https://eprint.iacr.org/2024/185.

[14] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. "Interactive Oracle Proofs". In: *Theory of Cryptography - 14th International Conference, TCC 2016-B, Beijing, China, October 31 - November 3, 2016, Proceedings, Part II*. Ed. by Martin Hirt and Adam D. Smith. Vol. 9986. Lecture Notes in Computer Science. 2016, pp. 31–60. DOI: 10.1007/978-3-662-53644-5_2. URL: https://doi.org/10.1007/978-3-662-53644-5%5C_2.

[15] Eli Ben-sasson, Alessandro Chiesa, and Nicholas Spooner. "Interactive Oracle Proofs". In: *Theory of Cryptography TCC 2016-B*. Vol. 9986. LNCS. 2016, pp. 31–60.

[16] Eli Ben-Sasson et al. "Aurora: Transparent Succinct Arguments for R1CS". In: *IACR Cryptol. ePrint Arch.* 2018.

[17] Eli Ben-Sasson et al. "DEEP-FRI: Sampling Outside the Box Improves Soundness". In: *11th Innovations in Theoretical Computer Science Conference, ITCS 2020, January 12-14, 2020, Seattle, Washington, USA*. Vol. 151. LIPIcs. 2020, 5:1–5:32.

[18] Eli Ben-Sasson et al. "Proximity Gaps for Reed-Solomon Codes". In: *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*. Ed. by Sandy Irani. IEEE, 2020, pp. 900–909. DOI: 10.1109/FOCS46700.2020.00088.

[19] Eli Ben-sasson et al. "Scalable Zero Knowledge Via Cycles of Elliptic Curves". In: *Algorithmica* 79 (Oct. 2016), pp. 1–59.

[20] Dan Boneh et al. "Efficient polynomial commitment schemes for multiple points and polynomials". In: *IACR Cryptol. ePrint Arch.* (2020), p. 81.

[21] Jonathan Bootle, Alessandro Chiesa, and Jens Groth. "Linear-Time Arguments with Sublinear Verification from Tensor Codes". In: *IACR Cryptol. ePrint Arch.* 2020 (2020), p. 1426.

[22] Sean Bowe, Jack Grigg, and Daira Hopwood. *Recursive Proof Composition without a Trusted Setup*. Cryptology ePrint Archive, Report 2019/1021. 2019.

[23] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. "Transparent SNARKs from DARK Compilers". In: *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part I*. Ed. by Anne Canteaut and Yuval Ishai. Vol. 12105. Lecture Notes in Computer Science. Springer, 2020, pp. 677–706.

[24] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. "Fractal: Post-quantum and Transparent Recursive Proofs from Holography". In: *LNSC*. Vol. 12105. May 2020, pp. 769–793.

[25] Alessandro Chiesa et al. "Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS". In: *Advances in Cryptology EUROCRYPT*. Vol. 12105. LNCS. Springer, 2020, pp. 738–768.

[26] Ronald Cramer, Léo Ducas, and Benjamin Wesolowski. "Short Stickelberger Class Relations and Application to Ideal-SVP". In: *EUROCRYPT*. 2017.

[27] Léo Ducas et al. "CRYSTALS-Dilithium Algorithm Specifications and Supporting Documentation". In: 2017. URL: https://api.semanticscholar.org/CorpusID:198994007.

[28] the Electric Coin Company. *The Halo 2 book*. URL: https://zcash.github.io/halo2/.

[29] Ariel Gabizon and Zachary J. Williamson. "Plookup: A simplified polynomial protocol for lookup tables". In: *IACR Cryptol. ePrint Arch.* (2020).

[30] Lior Goldberg, Shahar Papini, and Michael Riabzev. "Cairo - a Turing-complete STARK-friendly CPU architecture". In: *IACR Cryptol. ePrint Arch.* 2021 (2021), p. 1063.

[31] Shafi Goldwasser, Yael Kalai, and Guy Rothblum. "Delegating Computation: Interactive Proofs for Muggles". In: *ACM STOC*. ACM, 2008, pp. 113–122.
[32] Alexander Golovnev et al. "Brakedown: Linear-time and post-quantum SNARKs for R1CS". In: *IACR Cryptol. ePrint Arch.* 2021 (2021), p. 1043.
[33] Jens Groth. "On the Size of Pairing-Based Non-interactive Arguments". In: *Advances in Cryptology - EUROCRYPT*. Vol. 9666. LNCS. Springer, 2016, pp. 305–326.
[34] Ulrich Haböck. *Multivariate lookups based on logarithmic derivatives*. Cryptology ePrint Archive, Paper 2022/1530. https://eprint.iacr.org/2022/1530. 2022. URL: https://eprint.iacr.org/2022/1530.
[35] Nick Howgrave-Graham and Antoine Joux. "New Generic Algorithms for Hard Knapsacks". In: *Advances in Cryptology – EUROCRYPT 2010*. 2010, pp. 235–256.
[36] Assimakis Kattis, Konstantin Panarin, and Alexander Vlasov. "RedShift: Transparent SNARKs from List Polynomial Commitment IOPs". In: *IACR Cryptol. ePrint Arch.* (2019), p. 1400.
[37] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. "Nova: Recursive Zero-Knowledge Arguments from Folding Schemes". In: *Advances in Cryptology – CRYPTO 2022*. Ed. by Yevgeniy Dodis and Thomas Shrimpton. Cham: Springer Nature Switzerland, 2022, pp. 359–388.
[38] Benoît Libert, Somindu C. Ramanna, and Moti Yung. *Functional Commitment Schemes: From Polynomial Commitments to Pairing-Based Accumulators from Simple Assumptions*. 2016.
[39] Vadim Lyubashevsky et al. "SWIFFT: A Modest Proposal for FFT Hashing". In: *Fast Software Encryption, 15th International Workshop, FSE 2008*. Vol. 5086. Lecture Notes in Computer Science. Springer, 2008, pp. 54–72. DOI: 10.1007/978-3-540-71039-4_4. URL: https://doi.org/10.1007/978-3-540-71039-4%5C_4.
[40] Mary Maller et al. "Sonic: Zero-Knowledge SNARKs from Linear-Size Universal and Updatable Structured Reference Strings". In: *ACM SIGSAC -CCS*. ACM, 2019, pp. 2111–2128.
[41] Daniele Micciancio and Oded Regev. *Class on lattice-based cryptography*. 2008.
[42] Priyanka Mukhopadhyay. "Improved algorithms for the Shortest Vector Problem and the Closest Vector Problem in the infinity norm". In: Dec. 2018. DOI: 10.4230/LIPIcs.ISAAC.2018.35.
[43] I. S. Reed and G. Solomon. "Polynomial Codes Over Certain Finite Fields". In: *Journal of the Society for Industrial and Applied Mathematics* 8.2 (1960), pp. 300–304. DOI: 10.1137/0108018. eprint: https://doi.org/10.1137/0108018. URL: https://doi.org/10.1137/0108018.
[44] Srinath Setty. *Spartan: Efficient and general-purpose zkSNARKs without trusted setup*. CRYPTO. 2020.
[45] Srinath T. V. Setty, Justin Thaler, and Riad S. Wahby. "Unlocking the lookup singularity with Lasso". In: *IACR Cryptol. ePrint Arch.* (2023), p. 1216. URL: https://eprint.iacr.org/2023/1216.
[46] Zedong Sun, Chunxiang Gu, and Yonghui Zheng. "A Review of Sieve Algorithms in Solving the Shortest Lattice Vector Problem". In: *IEEE Access* 8 (2020), pp. 190475–190486.
[47] Polygon Hermez Team. *Scalable payments. Decentralised by design, open for everyone*. https://hermez.io.
[48] Tiacheng Xie et al. "Libra: Succinct Zero-Knowledge Proofs with Optimal Prover Computation". In: *Advances in Cryptology - CRYPTO*. Vol. 11694. LNCS. Springer, 2019, pp. 733–764.
[49] Tiancheng Xie, Yupeng Zhang, and Dawn Xiaodong Song. "Orion: Zero Knowledge Proof with Linear Prover Time". In: *IACR Cryptol. ePrint Arch.* 2022.
[50] Risc Zero. https://github.com/risc0/risc0. 2022.
[51] Jiaheng Zhang et al. *Transparent Polynomial Delegation and Its Applications to Zero Knowledge Proof*. 2020 IEEE Symposium on Security and Privacy (SP). 2019.

A Useful wizard subprotocols

In this section, we present a set of utility Wizard protocols with simple functionality which can be seen as common sub-routines that we use as part of *Section* 9.

A.1 Monomial basis evaluation protocol

In this sub-protocol, we assume a column P of size n and scalar values r and y, visible to the verifier, whose assignment value is unspecified during the protocol definition (r is possibly a random coin). The goal of the Monomial basis sub-protocol is to construct a convincing argument for the statement:

$$P(r) = \sum_{i<n} P_i r^i = y$$

where $P(X)$ is the polynomial whose coefficients are the entries of P. The Horner evaluation algorithm inspires the sub-protocol we present. The protocol uses an accumulator column H retracing the computation steps of the algorithm and constructed as:

$$H_i = \sum_{i \leq j < n} P_j r^{j-i}$$

Fig. 13 describes the protocol:

MonomialEval(P, a, y)

1. The prover constructs and commits to the column H
2. "Local constraint": $H_f = P_f$ where f indicates that the last value of the column is used.
3. "Global" $H = x(H \ll 1) + P$
4. The verifier checks that $H_0 = y$ and gets H_0 via a local opening query

Fig. 13. Monomial evaluation sub-protocol

A.2 Lagrange basis evaluation protocol

In this sub-protocol, we assume a column P of size n and a verifier with access to a scalar value r and y whose assignment value is unspecified during the protocol definition (r is possibly a random coin). a must not be a root of unity. The goal of the Lagrange basis sub-protocol is to construct a convincing argument for the statement:

$$P(r) = \frac{1}{n} \sum_{i<n} \frac{P_i}{\frac{a}{\omega^i} - 1} = y$$

where $P(X)$ is the Lagrange interpolation polynomial of P over the usual roots of unity domain without resorting to using a univariate evaluation query. This addresses a limitation of the Arcane compiler as univariate queries cannot exist at the beginning of the compilation. As an outline, the sub-protocol requires the prover to commit to an accumulating polynomial I retracing all the steps of computing $P(a)$ with the barycentric formula. Let ω be a generator of the groups of roots of unity. The sub-protocol is detailed in Fig. 14. In the sub-protocol, we introduce, the intermediary alias $\Delta = I - (I \ll -1)$. We have that

$$\forall i \geq 1 : \Delta_i = \frac{P_i}{\frac{a}{\omega^i} - 1}$$

$$\omega^i = \frac{a\Delta_i}{\Delta_i + P_i}$$

If we divide the equation by itself substituting $i - 1$, we eliminate the dependency with w^i

$$\omega = \frac{\Delta_i(\Delta i - 1 + P_{i-1})}{\Delta_{i-1}(\Delta_i + P_i)}$$

We can now rearrange the term to obtain a polynomial relation as below. This us a relation that we can use for a global constraint.

Interpolation(P, a, y)

1. The prover constructs and commits to the column I such that
$$\forall i < n : I_i = \sum_{0 \leq j \leq i} \frac{P_j}{\frac{a}{\omega^j} - 1}$$
2. "Local constraint": $I_0(a-1) = P_0$
3. "Local constraint": $(a-\omega)(I_1 - I_1) = \omega P_1$
4. "Global constraint":
$$(\omega - 1)\Delta(\Delta \ll -1) + \omega(\Delta \ll -1)P - \Delta(P \ll -1) = 0$$
The constraint is cancelled at points 0 and 1, implicitly.
5. The verifier checks that the last value of I equals the alleged value y using a local opening query.

Fig. 14. Lagrange evaluation sub-protocol

A.3 Evaluate in coefficient form a 2-variable polynomial

Consider a column C to the coefficients of a 2-variate polynomial. The protocol that we present here aims at proving evaluation of C at a point (x, y), where the polynomial has degree $k-1$ in x and degree $l-1$ in y. The structure of the sub-protocol is analogous to the univariate coefficient evaluation one: it is also inspired from the Horner algorithm and uses an accumulating column H such that

$$\forall 0 \leq i < n-1 : H_i = P_i + H_{i+1}(Z_{l,n,i+1}x + (1 - Z_{l,n,i+1})yx^{-k}) \tag{3}$$

Where $Z_{l,n}$ is a "periodic sampling" column satisfying

$$\begin{aligned} i = 0 \mod l &\iff Z_{l,n,i} = 1 \\ i \neq 0 \mod l &\iff Z_{l,n,i} = 0 \end{aligned}$$

The subprotocol is described in Fig. 15

BivariateEval(P, a, y)

1. The prover computes and commits to H
2. Global Constraint:
$$H = P + (H \ll 1)((Z_{l,n} \ll 1)x + (1 - (Z_{l,n} \ll 1)yx^{-k})$$
3. Local Constraint: $H_f = P_f$
4. Local Opening: H_0 and the verifier checks $H_0 \stackrel{?}{=} y$

Fig. 15. Monomial bivariate evaluation sub-protocol

A.4 Folding by one variable

By "folding", we do not mean the popular folding technique introduced in Nova [37]. Consider P a column seen as a bivariate polynomial in monomial form in X and Y. We aim to obtain the partial polynomial

$P(x, Y) = P_x(Y)$ for some value x. To this end we design a dedicated protocol reusing the univariate and bivariate monomial evaluation protocols:

Folding(P, P_x)

1. The verifier samples a random coin r_y.
2. The prover proves that $P(x, r_y) = P_x(r_y)$ using the bivariate evaluation technique on the left and the monomial one on the right.

Fig. 16. Monomial bivariate evaluation sub-protocol

A.5 Periodically subsampling a column

This section presents a sub-protocol that can be used to assert that a column S of size n and obtained by periodically subsampling a larger column L of size rn with offset k. Namely, the statement of the protocol is

$$\forall i \in [n] : S_i = L_{ri+k}$$

The technique used by this protocol is analogue to the folding protocol: the protocol uses a random coin ρ and simulates the probabilistic check:

$$\sum_{i \in [n]} S_i \rho^i \stackrel{?}{=} \sum_{i \in [rn]} \mathbb{I}_{i|r} L_i \rho^{i/r}$$

where $\mathbb{I}_{i|r}$ is the indicatrice function for $i|r$. The left hand evaluation is done using the monomial evaluation protocol and the second is evaluated using an analogue of the bivariate evaluation protocol which we present in Fig. 17

Subsampling(L, ρ)

1. The prover constructs and commits to the column H
2. "Local constraint":
$$H_f = P_f I$$
where f indicates that the last value of the column is used and I is a periodic sampling column indicating with a 1 the positions in H to be used.
3. "Global"
$$H = I\,[x(H \ll 1) + P] + [1 - I)](H \ll 1)$$
4. The verifier checks that $H_0 = y$ and gets H_0 via a local opening query

Fig. 17. Subsampling sub-protocol

A.6 MiMC compression function

This section details a sub-protocol for computing the MiMC [5] block compression function. We start by providing a reminder of the MiMC compression function algorithm in Fig. 20. σ_0 denotes the initial state of the compression function and b denotes the block. α is a small integer coprime with $|\mathbb{F}| - 1$ and N is the number of rounds of the compression function and c_i is the round constant of the round i. The compression function uses the Miyaguchi-Preneel construction.

MiMCCompression(σ_0, b)

1. $\sigma \leftarrow \sigma_0$
2. **for** round $i < N$
3. $\quad \sigma \leftarrow (\sigma + b + c_i)^\alpha$
4. **end for**
5. $\sigma \leftarrow 2\sigma_0 + b$

Fig. 18. MiMC compression function with the Miyaguchi-Preneel construction

The sub-protocol we present Fig. 20 in the following justifies a statement of the form $S_1 = \mathsf{MiMCCompression}(S_0, B)$ where S_1, S_0 and B are columns of the same size and the statement is to be understood as "row-by-row". The protocol works by creating intermediate columns for each round of the protocol and global constraints justifying the correctness of their assignment so that the overall degree of all the constraints stays reasonably low. To illustrate, we highlight the protocol using the scalar field of **BLS-12-377** in our settings. In this context, $\alpha = 17$ and $N = 62$ and the protocol declares two columns (A_i, B_i) for each round i.

MiMCCompression(S_0, b, S_i)

1. $S \leftarrow S_0$
2. **for** round $i < N - 1$
3. $\quad$ The prover commits to and constraints (using global constraints) (A_i, B_i) such that

$$A_i = (S + c_i + b)^4$$

$$B_i = (S + c_i + b)A_i^4$$

4. $\quad S \leftarrow B_i$
5. **end for**
6. The prover commits to A_f such that:

$$A_f = (S + c_i + b)^4$$

7. "Global constraint":

$$S_1 = (S + c_i + b)A_i^4 + 2S_0 + B$$

Fig. 19. Wizard sub-protocol for the MiMC compression function

A.7 Linear hashing

This section presents a sub-protocol to justify the correctness of hashing n small vectors of the same size l. Here, we assume l and n to be powers of two. The statement involves two columns L and H such that $|L| = l|H|$ and asserts that H is constructed from L as follows:

LinearHashing(L, H)

1. **for** $i \in [n]$
2. $\quad \sigma \leftarrow 0$
3. $\quad$ **for** $j \in [l]$
4. $\quad\quad \sigma \leftarrow \mathsf{MiMCCompress}(\sigma, L_{il+j})$
5. $\quad$ **end for**
6. $\quad H_i \stackrel{?}{=} \sigma$
7. **end for**

Fig. 20. Linear hashing statement

The sub-protocol works in two steps, the first steps operates on L to justify the construction of a column S storing the intermediate steps of the compression function and the second steps uses the sub-sampling protocol to checks that the final values of the hashes are the right ones. Thus, here we detail only the first part. The prover commits to two extra columns: O_σ and N_σ storing respectively, the old state and the new state of the permutation function. Then, the prover and the verifier engage in the MiMC block compression sub-protocol to ensure that O_σ and N_σ are consistent at every row and add an extra global constraint to ensure that the compression function calls are in sequence and that the state is reset at every new hash. Here, I_r is a periodic sampling column

$$O_\sigma = I_r(N_\sigma \ll -1)$$

When the hashes don't all have the same size , it is possible to extend the above technique by replacing I_r either by a column provided by the prover whose construction correctness should be enforced externally or using a precomputed column. This allows handling the cases where n and l are not known in advance also.

A.8 Merkle-tree membership

This section outlines our techniques to verify binary Merkle-trees, detailing the structure and constraints involved in the verification process. The technique has also two steps: (1) creating a module responsible for computing the Merkle proofs and (2) linking the assigment of the computation module into a "result" module which contains the statement. The second step is made using a subsampling argument between these two modules.

A.8.1 The result module The result module contains the following columns totalling the statement of the protocol:

- Leaf: storing the leaves whose membership is proved
- Position: storing the position of the leaves in the tree
- Root: storing the root of each tree, allegedly storing the corresponding leaf at the corresponding position
- IsActive: stores boolean flags indicating whether the current row of the module actually requires a Merkle proof verification. This is used to cancel constraints in the computation module to allow padding.

A.8.2 The computation module The table in Figure 21 represents how we model Merkle-tree verification, where each proof uses a segment of d rows. In each segment, the Merkle root is recomputed bottom-up. The zero column is implicitly not really committed to (it is a constant). This can be confusing, so we say "bottom" to mean the last row of a segment, which corresponds to the first hash of the Merkle-proof. For convenience, we use the following two expressions to construct the constraints:

IsInactive	NewProof	IsEndOfProof	Root	Curr	Proof	PosBit	PosAcc	*Zero*	Left	IntermState	Right	NodeHash
0	0	1	R_0	$H_{d-1,0}$	$\pi_{d-1,0}$	$b_{d-1,0}=0$	$p_{d-1,0}=b_{d-1,0}$	0	$H_{d-1,0}$	$I_{d-1,0}$	$\pi_{d-1,0}$	R_0
0	0	0	R_0	$H_{d-2,0}$	$\pi_{d-2,0}$	$b_{d-2,0}=1$	$p_{d-2,0}=b_{d-2,0}+2p_{d-1,0}$	0	$\pi_{d-2,0}$	$I_{d-2,0}$	$H_{d-2,0}$	$H_{d-1,0}$
...	...	...	...	...	...	...	...	0	...	...	...	...
0	1	0	R_0	L_0	$\pi_{0,0}$	$b_{0,0}=1$	$p_{d-2,0}=b_{0,0}+2p_{1,0}$	0	$\pi_{0,0}$	$I_{0,0}$	L_0	$H_{1,0}$
0	0	1	R_1	$H_{d-1,1}$	$\pi_{d-1,1}$	$b_{d-1,1}=1$	$p_{d-1,1}=b_{d-1,1}$	0	$\pi_{d-1,1}$	$I_{d-1,1}$	$H_{d-1,1}$	R_0
0	0	0	R_1	$H_{d-2,1}$	$\pi_{d-2,1}$	$b_{d-2,1}=1$	$p_{d-2,1}=b_{d-2,1}+2p_{d-1,1}$	0	$\pi_{d-2,1}$	$I_{d-2,1}$	$H_{d-2,1}$	$H_{d-1,1}$
...	...	...	...	...	...	...	...	0	...	...	...	...
0	1	0	R_1	L_1	$\pi_{0,1}$	$b_{0,1}=0$	$p_{d-2,1}=b_{0,1}+2p_{1,1}$	0	L_1	$I_{0,1}$	$\pi_{0,1}$	$H_{1,1}$
...	...	...	...	...	...	...	...	0	...	...	...	...
0	1	0	R_n	L_n	$\pi_{0,n}$	$b_{0,n}=0$	$p_{d-2,n}=b_{0,n}+2p_{1,n}$	0	L_n	$I_{0,n}$	$\pi_{0,n}$	$H_{n,n}$
1	0	0	0	0	0	0	0	0	0	I_{dead}	0	H_{dead}
...	...	...	...	...	...	...	...	0	...	...	...	...
1	0	0	0	0	0	0	0	0	0	I_{dead}	0	H_{dead}

Fig. 21. Modeling Merkle-tree verification. Each proof uses a segment of d rows.

$$\begin{aligned}\text{NotNewProof} &= 1 - \text{NewProof}\\ \text{IsActive} &= 1 - \text{IsInactive}\\ \text{NotEndOfProof} &= 1 - \text{EndOfProof}\end{aligned}$$

These are not materialized by columns but are used as subexpressions in the constraints for clarity. We now move on to list all the constraints that are enforced by the module:

1. **Constance of Root over a segment** Root is constant within a segment and it must be inactive when the "IsInactive" flag is set.

   "Global constraint":

   $$(\text{Root}[i] - \text{IsActive}[i]\text{Root}[i+1])\text{NotNewProof}[i] == 0$$

2. **Consistency of Root and Nodehash**: At the end of each segment, the root must be equal to NodeHash.

   "Global constraint":

   $$\text{EndOfProof}[i](\text{Root}[i] - \text{NodeHash}[i]) == 0$$

3. **Booleanity of PosBit**: This enforces PosBit to be boolean and zero if the inactive flag is set.

   "Global constraint":

   $$\text{PosBit}[i] = \text{IsActive}[i]\text{PosBit}[i]^2$$

4. **PosAcc packs PosBit in an integer**: this is aimed at reconstructing the leaf's position in the tree from its binary representation PosBit.PosAcc progressively computes the position of the opened leaf from the bits. It must be zero when the inactive flag is set.

   "Global constraint":

   $$\text{PosAcc}[i] = \text{IsActive}[i](\text{PosBit}[i] + 2\text{NotEndOfProof}[i]\text{PosAcc}[i+1])$$

5. **Left and Right assignment**: the flag PosBit decides which one of Proof or Curr is mapped to Left or Right.

   "Global constraints":

   $$\begin{aligned}\text{Left}[i] &= \text{PosBit}[i]\text{Proof}[i] - (1 - \text{PosBit}[i])\text{Curr}[i]\\ \text{Right}[i] &= \text{PosBit}[i]\text{Curr}[i] - (1 - \text{PosBit}[i])\text{Proof}[i]\end{aligned}$$

6. **Within a chunk, use the previous node hash as current node**: when the inactive flag is set, this also enforces that Curr is zero.

   "Global Constraint":

$$\text{NotNewProof}[i](\text{Curr}[i] - \text{IsActive}[i]\text{NodeHash}[i+1])$$

7. **Correctness of the MiMC computations**: this is done by using the MiMC compression function subprotocol:

$$\mathsf{MiMC}(\text{Zero}, \text{Left}, \text{Interm})$$
$$\mathsf{MiMC}(\text{Interm}, \text{Right}, \text{NodeHash})$$

8. **Zero padding of the proof**: this enforces that the proof is padded wherever the inactive flag is set: "Global constraint":

$$\text{Proof}[i] = \text{IsActive}[i]\text{Proof}[i]$$

B Selecting ring-SIS instances

In Section 7.1, we specify a generalized version of the SWIFFT hash function. In the current section, we provide an overview of the existing attacks and their costs. As for SWIFFT, our hash function is directly an instantiation of ring-SIS. The hash function, or rather, the family of hash functions we analyze hashes into prime fields and support several norm bounds instead of $\{0, 1\}$ for Ajtai [2] and SWIFFT [39]. The instances that we analyze span over a large range of parameters, and this requires evaluating both *lattice reduction attacks* and *combinatorial attacks*. Finally, the scope of this work is restricted to the classical setting.

B.1 The Short-Integer-Solution and Its "Ring" Variant

Let $m > n$ be integers, q a prime and $b < q$.

Definition 13 (Short-Integer-Solution Problem (SIS)). *Given random $A \in \mathbb{Z}_q^{n \times m}$, find x such that $Ax = 0_n \wedge \|x\|_\infty < b$*

Definition 14 (Inhomogeneous-SIS (ISIS)). *Given random $A \in \mathbb{Z}_q^{n \times m}$ and $t \in \mathbb{Z}_q^n$, find x such that $Ax = t$*

We start with a few observations on SIS:

- SIS (and ISIS) cannot become harder by increasing m. That is because an attacker can always restrict the search space to $m' < m$ by arbitrarily forcing some entries of x to zero.
- The problem only becomes harder as we increase n, this corresponds to adding more constraints on what can be a valid x.
- It can only become harder as we restrict to smaller b. That is because it is equivalent to restricting the search space.
- $b \geq q$ makes the problem trivial, as it can be solved by Gaussian elimination in polynomial-time.

Remark 1. The work of [26] uncovered an efficient procedure for solving γ-ideal-SVP in polynomial time, a problem closely related to ring-SIS. We argue that they do not apply to the scope of our analysis. Indeed,

- They are in the quantum setting
- The approximation factor they apply the attack on is exponential. This is not what we typically use for cryptographic applications

– Ring-SIS is not exactly an ideal lattice problem (it is therefore not currently known if an efficient reduction from ring-SIS to Ideal-SVP actually exists).

Now we define the ring version of the SIS problem.

Definition 15 (The Ring-(Inhomogeneous)SIS Problem). *Given $A \in \mathcal{R}^m$ drawn randomly, following the uniform distribution (for its coefficients) and $b < q$, the ring-ISIS problem is to find $x \in \mathcal{R}^m$, non-zero, such that $\|x\|_\infty < b \wedge Ax = 0_\mathcal{R}$.*

The ring-(I)SIS assumptions can be seen as special cases of SIS where A is drawn from a restricted set of matrices representing the polynomial multiplication module $X^n + 1$. One should note that m means different things in our definitions of SIS and ring-SIS. For clarity "$m_{\text{SIS}} = nm_{\text{RSIS}}$". Working with ring-SIS has several practical benefits compared to SIS: the space taken to represent A is n times smaller, and the product Ax can be computed much faster using FFT algorithms in $nm \log n$ instead of mn^2.

B.2 Security properties

We require our hash function (as specified in Section 7.1 to have preimage resistance and collision resistance.

Definition 16 (Preimage Resistance). *Given y, find x such that $\mathsf{H}(x) = y$*

The definition of preimage resistance coincides with the InhomogenousSIS problem. We attack it by solving SIS $(y, A) \cdot (1, x) = 0$. This is equivalently as hard as solving SIS with input size m.

Definition 17 (Collision Resistance). *Find x, x' such that $\mathsf{H}(x) = \mathsf{H}(x')$*

An attack against collision-resistance is obtained by breaking SIS for the matrix $(A\| - A)$, under the constraint that a solution $s = (s_1\|s_2)^T$ satisfies $s_1 \neq s_2$. This is equivalent to multiplying m by 2. From that, we can deduce the fact that collisions are easier to find than preimages. Thus, in the following, we will restrict our attention to attacks for finding collisions.

B.3 Overview of the cryptanalysis report

To estimate the hardness of ring-SIS instances, we consider two classes of attacks: combinatorial and lattice reductions. In practice, no attack is known to work significantly better on ring-SIS rather than an equivalent SIS instance. Additionally, in practice, the security of our hash function is bottlenecked by attacks on collision resistance. Thus, we will only consider the *equivalent* (not-ring)-SIS instance with parameters $q, n, m' = nm, b$.

B.4 Lattice Reduction Techniques (BKZ2.0)

Foremost, we note that solving an SIS instance is the same as finding a short vector in the kernel lattice.

$$\mathcal{L} = \Lambda^\perp(A) = \{z \in \mathbb{Z}_q^{m'} : Az = 0\}$$

We are always free to pick $m_0 < m' - n$ if that is convenient. The best-known algorithm to do so is BKZ2.0, a generalization of the seminal LLL algorithm. This algorithm works by repeatedly calling an SVP oracle which optimally reduces lattices to smaller dimension $k < m_0$. The BKZ algorithm will output, with overwhelming probability, a vector of size $b_2 = \|v\|_2 = \delta^{m_0}\mathsf{vol}(\mathcal{L})^{1/m_0}$ and thus we need to set,

$$b_2 = \delta^{m_0} q^{n/m_0} \wedge b\sqrt{m_0} < q \tag{4}$$

The second term comes from the fact that if m_0 is too big, then the smallest L2-ball containing the L_∞ ball of SIS candidate contains the whole space. This does not necessarily mean the instance is broken, but it

means our estimations are irrelevant. Therefore, we will reject those cases. We recall that for random lattices, kernels $\mathsf{vol}(\mathcal{L}) = q^n$ with overwhelming probability.

Since the efficiency of BKZ has been widely studied in the L_2 norm settings but not in the L_∞ setting, we need a strategy to convert the L_∞ instances into equivalent ones over L_2. Let $\mathcal{B}_2(r)$ and $\mathcal{B}_\infty(r)$ denotes respectively the L_2 and L_∞ balls of radius r centered at the origin. Then, from the Minkowski bound we have that

$$\mathcal{B}_2(r) \subset \mathcal{B}_\infty(r) \subset \mathcal{B}_2(\sqrt{m_0}r)$$

To derive a cost estimate for BZK in the L_∞ setting, we try values of b_2 in the range $[\frac{b}{2}; \sqrt{m_0}\frac{b}{2}]$ estimate their security as we will explain the following of the section, and then account for a probability that the short L_2 is small enough in the L_∞ norm to be a solution of the target SIS instance. We retain the estimate of the value of b_2 that minimizes $\frac{C}{P}$ where C is the runtime of BKZ and P is the success probability. We estimate P by modelling the vector output by BKZ as being normally distributed (each coordinates independently) with a standard deviation of $\frac{b_2}{\sqrt{m_0}}$ as suggested in [27].

Here, we have two free parameters: m_0 and δ. δ is what we call the root Hermite factor. It can be interpreted as the "output quality" that we can expect from BKZ. For the most part, it depends on the BKZ block size k (and also a little on m_0).

A comprehensive choice of the oracle, along with a model for their runtime can be found in the work of [4]. All oracles and models come with different tradeoffs. The most efficient ones (in runtime) are sieve ones, while enumeration ones require smaller space. Finally, based on the work of [46], we take that LD Sieve is the fastest sieve algorithm. This gives us the following heuristic runtime formula (in CPU cycles) for a single call to the SVP oracle.

$$\log t_{\mathsf{oracle}} = 0.292k + 16.4 \tag{5}$$

[3] gives the following cost estimates of the overall runtime of BKZ2 (number of calls to the oracle) and an estimation of the root-Hermite factor achieved by BKZ.

$$\rho = \frac{{m_0}^3}{k^2} \log m_0$$

$$\log_2 \delta = \frac{1}{2(k-1)} \left(\log_2(\frac{k}{2\pi e}) + \frac{\log_2 \pi k}{k} \right) \tag{6}$$

One should note that Eq. (6) is *only* asymptotically correct and it returns imprecise results. For this reason, our analysis rules out small values of k by setting a lower bound on the examined values. In practice, this is not a problem because the arbitrary lower bound that we set does not correspond to instances that are not hard enough for cryptographic purposes.

B.5 Direct SVP attacks

Although BKZ has been shown to achieve superior results than direct SVP attacks we also consider the latter. We consider two attacks, one using the sieving algorithm over the L_2 norm (as in the BKZ case), accounting for success probability. And another one derived from the work of [42] directly targeting L_∞ SIS.

B.6 Combinatorial Attack

In addition to lattice reduction techniques, an important class of attacks for SIS and ISIS stems from the field of attacks against the subset-sum problem.

B.6.1 Camion-Patarin and Wagner attacks The course [41] describes the basic version of these attacks and gives an easy procedure to determine their efficiency. The attack is also known as CPW. In [8], the authors present several improved methods over the former method, and they achieve a 10-bit reduction on SWIFFT. Those improvements have been obtained by generalizing the initial attack for which they used careful manual tuning of its parameters.

As in [8] suggests, once we have found the optimal list-tree depth k, we can reduce the value of m to the smallest value that verifies

$$\frac{2^k}{k+1} < \frac{m \log(b)}{n \log q}$$

We remind the reader that we are looking for collisions in the input space $x \in [0; b[^m$ which differs from $\|x\|_\infty < b$. This explains why our formula uses b in place of $2b-1$ as it can be sometimes found in the literature. The above attack can, in fact, be generalized to a setting where the output space is split in k chunks of size $l_1, l_2, \ldots, l_k$ such that $\sum_i l_i = n$. By tweaking the size of each l_i we can optimize the attack.

Methodology We will consider two cases:

- If $\mathbf{n \leq 50}$, we exhaustively try every possible combination of l_i such that $\sum_i l_i = n$ for $k < \log_2 m$. And we simulate the attack by counting all operations. To reduce the cost of the exhaustive search, we restrict the search space to $l_i \leq l_{i+1}$.
- If $\mathbf{n > 50}$, then we apply the simplified analysis given in [41]. From [41], the cost this will give us is an overly pessimistic result, but in practice, these SIS instances are better attacked using lattice reduction techniques. Thus, this fact is without consequence on our estimations.

In our estimation, for values of n (i.e., the dimension of the output space), we considered a refinement of the technique to account for the fact that different tunings are possible (splitting the output space in "non-equals" chunks). We exhaustively search the best set of parameters when $n < 50$. Otherwise, the exhaustive search of parameters is too computationally heavy, and we fall back to the method of [41]. This is without consequence for our estimations. Indeed, in practice, for our choices of q, we observe that SIS instances with $n < 50$ are typically bottlenecked by the BKZ attack—for our choices of q—in practice.

To estimate the cost of the attack:

- In the *basic case*, we use the formula
- In the *exhaustive case*, we simply count all operations. We assume the running time of merging two lists is linear in the size of the resulting merged list. We consider that the running time of *creating the initial leaves lists* is roughly equal to *enumerating all possibilities*.

B.6.2 On the HGJ and BCJ refinements Howgrave-Graham and Joux introduced these techniques in 2010, [35]. This class of attacks is somewhat similar to CPW, in the sense that it relies on recursively splitting the initial problem and merging the partial solutions. As an outline, the difference there is that it relies on splitting the problem in "weight" rather than in space.

These techniques have proven to be more effective when the problem has a low-density of solutions, while CPW is more effective for higher-density instances. In our case, we seek to pick instances of SIS which maximize the "compression ratio" and hence the density. Typically, our instances have densities that are above the range of effectiveness of these attacks. Thus, we do not consider them in this work.

B.6.3 On Optimizations for Ring-SIS In [8], the authors present a technique to reduce the cost of the attack when the set of *acceptable* input polynomials is preserved by multiplication by the transformation $\psi : s(X) \to Xs(X)$. This is the case when either the ring modulus is $X^n - 1$ or the input space has sign symmetry (meaning $\mathcal{B} = -\mathcal{B}$) and the modulus is $X^n + 1$. We stress that neither is our case, and we recall that we use the modulus $X^n + 1$ with $\mathcal{B} = [0; b[$. It is however possible to reduce to a case where this technique

is applicable nonetheless. Let $\mathbf{1}_m = (1, 1, 1, \ldots),$, instead of directly trying to find s such that $As = 0$ we seek $s' \in \mathcal{B}' =' \mathcal{B} - \frac{b-1}{2}$ such that $A(s' + \frac{b-1}{2}(1, 1, 1, \cdots)) = 0$. If b is even (our case), then the solution space for s' has sign symmetry. We note that although $\mathcal{B}'$ is not a set of short integers, this does not affect the runtime of CPW. We do not expand on the technical details of the techniques. At a high-level, these techniques decrease the size of each list by a factor of $2n$, where n is the degree of the ring-modulus. Thus, it achieves a speed-up of $2n$.

However, as the work of [8] points out, this optimization is incompatible with the following one, based on the Hermite Normal Form (HNF).

B.6.4 Optimization using the Hermite Normal Form (HNF) The Hermite Normal Form of a matrix is an equivalent representation of the (I)-SIS problem. If $A = (A_0 \| A_1 \| \ldots \| A_{n-1})$ is the SIS matrix, then we call $H = (I \| A_0^{-1} A_1 \| A_0^{-1} A_1 \| \cdots) = (I \| A')$ its normal form. The (I)SIS can then be equivalently rephrased as, what we call, the *approximate* (I)SIS problem.

Definition 18 (Approximate (I)SIS problem). *Find $s, e \in \mathcal{B}$, such that $Ax + e = R$, where $R = 0$ in the homogeneous case.*

Based on this, we can adapt the CPW algorithm to turn it into an attack for the approximate (I)SIS problem. [8] expands on the details of the algorithm.

Some notes on the costs estimates We note that both estimates are missing some hidden costs,

- The attacks we consider are typically as memory intensive as they cost in terms of computation.
- We do not account for the evaluation costs of each partial candidate solution. This would in practice add a few bits of security.
- The storage of each candidate is not "1". On top of impacting the memory complexity (which we chose not to account for anyway), it has an impact on the costs of the memory accesses as well.

For these reasons, we believe the costs are somewhat over-pessimistic. Nonetheless, we prefer to go with the initial approach and leave it as a future task to evaluate the concrete cost in CPU cycles of these attacks.

B.7 Concrete parameters

Based on the above analysis, we have run a parameter selection. The table in Fig. 22 gives a set of parameters for the ring-SIS instance. Here, q denotes the order of the underlying prime field, b is the bound of the SIS instance, and n is the degree of the ring modulus $X^n + 1$. For the instance, B1 the non-ring version of SIS is considered. The reason the attack costs are higher than the claimed security level has two explanations. First, we are willing to take conservative parameters for our SIS instances as our work is, to the best of our knowledge, the first to take SIS-based hash function in production. The second reason is that we revisited are our BKZ estimator and found that the our estimation $\delta = f(k, m_0)$ was returning values that were lower than what was used in [4] and thus giving very pessimistic results. The results we present Fig. 22 that we present are consistent with their estimates. We will reconsider these parameters in the future. We stress that the q we are using are prime numbers

C Linea's state representation

This section briefly articulates how the state of Linea is Merkle-ized. The solution differs from the Keccak-based Patricia Merkle-Tree that Ethereum uses. The motivations are multiple: our use cases require a more prover-efficient solution as using Ethereum's solution would necessitate in the 10000's of Keccak per transaction block and this would dominate the prover costs. The second point is that a PMT-based solution involves more control flow logic making it more complex to arithmetize in practice.

	q	b	n	SVP (L_2)	SVP (L_∞)	BKZ attack	CPW attack
A1	$\approx 2^{64}$	2^2	32	315.41	634.88	371.53	144.0
A2	$\approx 2^{64}$	2^4	64	315.41	634.88	358.29	305.57
A3	$\approx 2^{64}$	2^6	128	415.27	872.96	415.0	598.14
A4	$\approx 2^{64}$	2^{10}	256	494.99	1111.04	380.99	1272.31
A5	$\approx 2^{64}$	2^{16}	512	614.42	1269.76	344.85	2741.67
A6	$\approx 2^{64}$	2^{22}	1024	886.27	1904.64	401.92	5967.82
B1	$\approx 2^{254}$	2^2	7	275.99	551.18	324.59	259.03
B2	$\approx 2^{254}$	2^4	16	313.07	634.88	355.56	270.0
B3	$\approx 2^{254}$	2^6	32	412.06	853.12	411.68	637.0
B4	$\approx 2^{254}$	2^{10}	64	491.19	1031.68	377.78	1262.46
B5	$\approx 2^{254}$	2^{16}	128	609.74	1269.76	341.83	2720.33
B6	$\approx 2^{254}$	2^{24}	256	807.72	1745.92	333.14	5921.27
B7	$\approx 2^{254}$	2^{32}	512	1203.09	2539.52	405.68	13013.8

Fig. 22. Lattice parameters targetting 128 bits of security

Our solution employs Sparse Merkle Trees (SMTs) at its foundation and uses them as a random access memory where memory words are addressed by integers. The SMT is parametrized in such a way that it minimizes the number of required levels and has 40 levels. Our approach enhances the SMT to support key-value addressing by building a structure inspired by sorted doubly linked lists at the leaf level where each leaf references its immediate neighbours in the key set. This allows us to instantiate a key-value addressed state accumulator which can be addressed either be the hash of an Ethereum address (for the world state) or by a 32-byte value for the storage of each contract.

C.1 Structure of the tree

Here, we present how to derive the value (and the implicit structure) of the non-empty leaves of the sparse Merkle tree. We set the (x) leaves as the hash of the following structure:

- **HKey**: the hash of the key
- **Value**: the value we wish to store
- **Prev**: the position of the leaf whose associated **HKey** is immediately below the current node's **HKey**
- **Next**: the position of the leaf whose associated **HKey** is immediately above the current node's **HKey**

We refer to this structure as the opening of a leaf. Empty leaves have (cryptographically speaking) no opening since it is infeasible to find a preimage for 0 (which we use for empty leaves in our SMT).

C.2 Initialization of the tree

To initialize our accumulator, we insert two utility leaves in the tree. These leaves do not contain functional data by themselves. They are only here for convenience and practicality. Insert two nodes (that we will call the "head" and the "tail" by convention). By "position", we mean the index of the sparse Merkle-tree leaves. In our construction, we set up a doubly linked list and "head" and "tail" represent nodes that are respectively always at the beginning and always at the end of the list. But they are not associated with any particular "key" in the set. That's why their $HKey$ field is not the hash of something. It also ensures that these entries' values are never accessed via the accumulator protocol.

At position 0 (head):

- **HKey** 0 (and not Hash(0))
- **Value** 0

– **Prev** 0 (it points to itself as an invariant)
– **Next** 1 (points to the tail at the beginning, but this will be updated as we add entries in the set)

At position 1 (tail):

– **HKey** p-1 where p is the size of the BLS12-377 scalar field
– **Value** 0
– **Prev** 0 (points to the head at the beginning, but this will be updated as we add entries in the set)
– **Next** 1 (it points to itself as an invariant)

C.3 Tracking the position of the empty leaves

Without going into the details here, the state manager picks an empty leaf to write over when we insert into an empty storage address. However, we have a requirement that the position in which we insert new leaves must be deterministic: any person who looks at the transaction history should be able to reconstitute the Merkle-tree identically (e.g., with the same leaves at the same positions).

If the state manager were given wiggle room to insert into any position of the SMT, then he would be able to perform an attack where he uses a random node for insertion and never discloses it. The accumulator state update would still be correct, but it would be infeasible to reconstruct an identical SMT since the exact positions where the insertion occurs are ultimately private.

Countermeasure :

As a countermeasure, we introduce the following mechanism: we force the state-manager to insert to the left. Namely, the state-manager never reuses a position twice even if the corresponding node has been deleted. While it may seem wasteful to not reuse erased leaves, we will never run out of leaves to write over. Under plausible hypothesis of storage usage, we estimate it would take over hundreds of years before we run out of leaves with this solution.

That being said, it should be verifiable externally without maintaining more than a root hash. For that purpose, we extend the SMT with an extra level on the top whose children are the "SMT root hash" and position of the next free node: NextFreeNode. Note that the depth of the tree is now 41.

Every time we make an insertion, we fetch the 'NextFreeNode' value from the Merkle tree and increase it by 1. All updates to this field are justified by a Merkle-Proof (even though it contains only a single element) as any other leaves. With this modification in mind, when we refer to the root of the SMT, we mean the root of the upmost node and not the root of the sub-SMT root.

C.4 Operations on Cryptographic Accumulator

The cryptographic accumulator, based on Sparse Merkle Trees (SMTs), facilitates operations such as insertions, deletions, and updates with verifiable proofs. This section outlines the procedures for these operations and their verification mechanisms.

Insertion Operation To insert an entry (k, v) into the map, the process involves identifying the two surrounding keys, $HKey-$ and $HKey+$, which are the largest and smallest keys, respectively, such that $HKey- < \text{hash}(k) < HKey+$. The positions of these keys within the tree are denoted as $i-$ and $i+$. The next free node, $i = \text{nextFreeNode}$, is determined by taking the value stored near the root of the tree. This value is incremented at the end of the procedure.

Procedure:

1. Obtain Merkle proofs for leaves at positions $i+$, $i-$, and i, along with the opening nodes for $i+$ and $i-$ ($N+$ and $N-$).
2. Verify that $N+.Prev = i-$ and $N-.Next = i+$ to ensure no intermediate key exists between $HKey-$ and $HKey+$.
3. Update the SMT by setting $N-.Next = i$, inserting the new value of $\text{hash}(N-)$, and similarly updating $N+$ and the new node N.
4. Increment nextFreeNode and derive the new root hash from the Merkle proof.

Output: The prover generates three Merkle proofs and the positions $i+$, $i-$, and i, along with the modified openings $N-$ and $N+$.

Verification: The verifier checks the consistency of $N-$ and $N+$ with their Merkle proofs and the current root hash.

Read Zero Operation The "Read Zero" operation is designed to demonstrate to an external verifier that a specific key k is not present within the map. This operation is similar in nature to the insertion process, and thus, a concise explanation is provided here:

Procedure:

1. Identify the two surrounding keys, $HKey-$ and $HKey+$, which are the largest and smallest keys, respectively, such that $HKey- < \text{hash}(k) < HKey+$. Denote their positions within the tree as $i-$ and $i+$.
2. Retrieve and send the openings $N-$ and $N+$, along with their respective Merkle proofs, to demonstrate the non-inclusion of key k between these two keys.

Verification: The external verifier conducts the following checks to validate the claim that key k is not part of the map:

1. Verify the consistency of the Merkle proofs with the alleged openings $N-$ and $N+$ against the current root hash of the Sparse Merkle Tree (SMT).
2. Confirm that $N + .Prev = i-$ and $N - .Next = i+$, and that $HKey- < \text{hash}(k) < HKey+$, ensuring there is no intermediate key $HKey'$ that could exist between $HKey-$ and $HKey+$.

This operation effectively proves the non-existence of a key within the map by leveraging the structure and verifiability of the Sparse Merkle Tree.

Deletion Operation Deleting a key from the map is analogous to removing an item from a doubly linked list, essentially reversing the insertion operation.

Procedure:

1. Identify the positions $i+$, $i-$, and i for the keys $k+$, $k-$, and the key to be deleted.
2. Update $N-$ to point to $N+$ instead of N and similarly update $N+$ to point to $N-$, effectively removing N from the list.
3. Replace the leaf $\text{hash}(N)$ by 0 and update the rest of the SMT.

Output: The prover provides three Merkle proofs and the positions $i+$, $i-$, and i, along with the original openings $N-$, N, and $N+$.

Verification: The verifier checks the consistency of the updated nodes with their Merkle proofs and the current root hash.

Update Operation Updating a value in the map follows a similar procedure to a normal Merkle-tree update.

Procedure:

1. Locate the position i and the node N for the key to be updated.
2. Obtain the Merkle proof for N.
3. Update the value field of N and derive the new root hash using the Merkle proof.

Verification: The verifier checks the Merkle proof for N against the current root hash and verifies the updated value.

Read Operation Reading a key from the map involves a standard Merkle proof verification to confirm the presence or absence of the key.

Procedure:

1. Find the position i and the node N for the key.
2. Obtain and return the Merkle proof for N.

Verification: The verifier checks the Merkle proof for N against the current root hash to validate the read operation.

C.4.1 Hash Functions and Sparse Merkle Tree Specifications The MiMC hash function, operating over the scalar field of the BLS12-377 curve, is selected for its efficiency and security properties. The SMT is characterized by a binary arity and a depth of 40, ensuring adequate capacity and collision resistance for our applications. The implementation adheres to the MiMC construction with Miyaguchi-Preneel scheme, utilizing 62 rounds of the cipher with an S-box exponent of $e = 17$, in non-Feistel mode. The construction omits the XOR operations, replacing them with field additions within the BLS12-377 scalar field.