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. Since the proof size may not still be sufficiently succinct, we apply different techniques such as self-recursion, standard recursion, and proof aggregations.
The security of different components and steps will be discussed in separate papers as we advance on the final design of the Linea prover.

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

1 Introduction

Polynomial Commitments A polynomial commitment [37] 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 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 [32], Plonk [6], Halo [21], Halo2 [26], Marlin [24], Spartan [42], Virgo [49], Brakedown [30], Orion [47], Libra [46], Aurora [15], Fractal [23], Sonic [39], Nova [35], and Lasso [43] to cite a fraction of the existing works.

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 [32]). 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 [11], Cairo [28], Polygon-zkEVM [45], RISC [48], ScrollTech [1]. However, building a system capable of proving arbitrary executions of the Ethereum Virtual Machine is no easy task. To give an idea, the zk-EVM of Consensys [11] 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.

Interactive Oracle Proofs 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 [13]. IOP protocols can be transformed into concrete secure argument systems using a Merkle tree. Later works have introduced several variants of IOPs such as polynomial-IOPs or tensor-IOPs, where the prover can perform polynomial evaluation queries [6] or tensor queries [20]. 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.

Reed-Solomon Codes and Its Decoding Regimes Generally speaking, the Reed-Solomon encoding receives the evaluations of a function over k points, considers them as the coefficients (or evaluations) of a polynomial $P(X)$, and then outputs the evaluation of such a $P(X)$ over a fixed set D (usually the set of roots of unity over the finite field $\mathbb{F}_q$). The output is called a codeword of size $|D|$.

Considering the relative Hamming distance as the measure, a decoding algorithm receives the vector w over D and outputs codewords close to w. For the Reed-Solomon code, one considers the unique decoding regime and list decoding regime. In unique decoding, the radius of the ball around w (relative Hamming distance) is small and there is only one codeword that can be that close to w, while in the list decoding regime the radius is larger and there are many codewords that fall in the ball around w.

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. [18] specifies how to instantiate proof-carrying data through recursion using a pairing-friendly cycle of elliptic curves. The works od Halo [21], Halo2 [26] and Nova [35] present several techniques to implement PCD or IVC using a (possibly non-pairing-friendly) cycle of elliptic curves. In [12] the authors present a recursion technique that specifically targets recursion over the protocol of GKR [29] and more generally any interactive protocol whose Fiat-Shamir transform involves hashing long string in the first round.

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 [14] 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 [11].

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 [6, 19, 26, 28, 27]. As the original goal of our work is to build a succinct proof system for the zk-EVM specified in [11], 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 [5] into an BPC scheme inspired by the works of Brakedown [30], batch-FRI [17], and RedShift [34].

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 can bring a trade-off of efficiency and soundness-error. Particularly, if the field is large enough, the efficiency gain comparing 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 [38] 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 [38]. 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 [41]. 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 erasure code (whose encoding algorithm requires FFT). Orion and Brakedown [47, 30] 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 easier 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 *self-recursion* technique; it works by arithmetizing the Vortex proof and returning it back to the Wizard protocol.

The self-recursion reduces the size of the proof to its 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[32] or Plonk[6] 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 [30] and Orion [47], 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 [30] and Orion [47]. 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 [38]. 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 [32].

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,

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

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 [6].

[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.

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 [47]). *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 interaction would 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).

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 ([34]) but extended to a batched setting. Our presentation closely follows the formalization of [19, 6]. 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 $k, n = poly(\lambda)$.*
- *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 $k, n = \text{poly}(\lambda)$, 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 $(P_{\text{PC}}, V_{\text{PC}})$ 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 $k = \text{poly}(\lambda)$, 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 Overview of compilation

In this section, we present the set of techniques we use for proving the execution of a zk-EVM. Namely, the zk-EVM of Linea[11] is formalized in a high-level constraint language, and we translate it into a concrete proof system producing proofs that are verifiable on the Ethereum network. As outlined in Fig. 1, we organize this transpilation process around four major axis: the Wizard-IOP model, the Arcane compiler (including UniEval), the Vortex commitment scheme, and a self-recursion technique.

4 Wizard IOP

The prover $\mathcal{P}$ of an IOP protocol[14] 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 [39], [24], [6] 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 [20] 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 over a given field and the verifier is allowed to perform queries chosen from a *wide class*. As we explain later in this section, these queries can involve several polynomials 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 vector 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

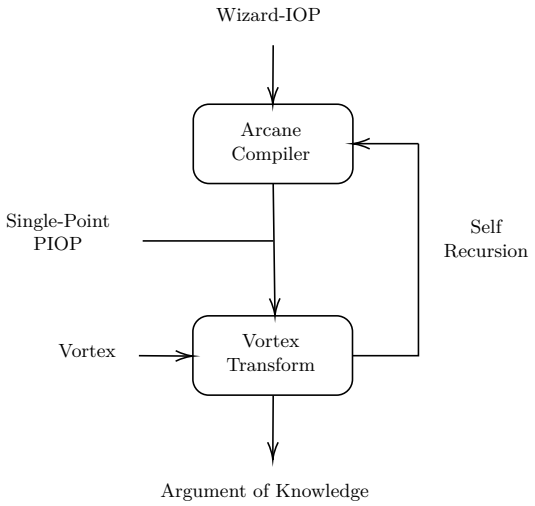

Fig. 1. Global structure of the prover

specifying protocols with a top-down approach. We start from an abstract protocol, and we 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 [11] involves hundreds of polynomials and thousands of constraints. It would be unthinkable 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 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. Assume 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 to forget any part.

4.1 Available queries

In the following, we list and describe the queries available to $\mathcal{V}$.

Range Check Let B be a bound known beforehand. The query is made over a vector 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 Given two lists of vectors (seen as matrices) S and T, we check that all rows in S should be included among the rows of T, ignoring multiplicity. We denote the inclusion query as, "Inclusion" : $S \subset T$.

Fixed Permutation Check Given two lists of vectors (seen as matrices) and any (imposed) fixed permutation σ, we check that the i^{th} row in S must equal the row at index $\sigma(i)$ in T. If that is the case, the oracle returns 1, otherwise 0. Matrices S and T are expected to have the same number of rows. We denote a fixed permutation check as, "FixedPermutation" : $S \sim_\sigma T$.

Permutation Check Given two lists of vectors (seen as matrices) 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 are expected to have the

same number of rows (note that in fixed permutation queries, σ is imposed. Here we just want to prove that a permutation σ exists). We denote a permutation query as, "Permutation": $S \sim T$.

Scalar-Product Given two vectors a and b, as well as a scalar c, the oracle returns 1 if and only if $\langle a|b\rangle = c$. For the scalar-product check $\langle a|b\rangle = c$, we use the notation, "ScalarProduct" : $\langle a|b\rangle = c$.

Local Constraint The verifier queries several vectors at potentially different points. The oracle returns the values for the queried positions and the verifier expects that these values satisfy a specific relation (this relation represents the local constraint).

As an example, let u, v be two vectors 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 vectors $v_1, \cdots, v_k$ of the same size n. 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 it is more convenient.

A global constraint is always defined over a domain of the same size as the polynomials involved (the one with the maximum size), Namely, for polynomials of degree d, the global constraint should be satisfied over the domain $X^d - 1$. Using this convention, we may not explicitly mention the domain for a global constraint.

Univariate Evaluations (UniEval) For a vector v_i 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 a random point (random but possibly related to other steps of the underlying protocol) 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.

4.2 Abstract references

Abstract references are a useful way to refer to vectors that are directly derived from pre-existing committed vectors. These operators can be combined with one another and can be used as the object of a query. For instance, one might send a range check query for a subsample of a committed vector v rather than on the entirety of the positions of v. Note that abstract references are neither queries nor are they committed vectors but they can be seen as a way to make queries about committed vectors more expressive.

Subsampling The procedure is given access to a vector v of size n, and as inputs an offset i and a sampling period k such that $k|n$ and $i < k$. The object returned is a vector of size n/k obtained by taking all the elements v_{jk+i} for all $j < n/k$. We use the notation $\mathsf{Subsample}(v; i, k)$ to denote the subsampling from v with offset i and period k.

Interleaving Given access to k committed vectors $v_1, \cdots, v_k$, we return a reference to the vector obtained by interleaving them (e.g., for the vector $a = (a_0, \ldots, a_n)$ and b of the same size, the interleaving is $(a_0, b_0, a_1, b_1, \ldots, a_n, b_n)$). We use the notation $\mathsf{Interleaving}(a, b)$ to designate the obtained vector.

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 $\mathsf{Shift}(v; k)$ to refer to the resulting vector.

Repeating Returns a k fold repetition of the input vector v.

5 The Arcane Compiler, Polynomial IOP from Wizard IOP

The Arcane compiler transforms Wizard IOPs into Polynomial IOPs. Arcane is organized as a sequence of compilation steps, each of them responsible for a small transformation. A transformation can be either a small optimization or a reduction technique that transforms an "abstract" query into "simpler" queries. Applying these compilations steps one after the other produces step after step a Wizard-IOP that uses fewer types of queries.

To provide a more tangible idea, Arcane starts by removing the range checks and converting them into inclusion checks. Then, in their turn, the inclusion checks are converted into local, global constraints and permutation checks and so on. In the end, Arcane outputs a polynomial-IOP where the verifier performs one (univariate evaluation) query on each message, all at the same point. Hence, we call the resulting protocol a single-query Polynomial-IOP. This section discusses the compilation steps of the Arcane compiler in sequential order. To give a brief overview, the steps happen in the following order:

1. reduction of the range checks
2. reduction of the inclusion checks
3. reduction of the fixed-permutation checks
4. reduction of the permutation checks
5. reduction of the scalar-product checks
6. merging of the global constraints
7. reduction of the abstract references
8. single-point univariate queries from multiple univariate queries and local constraints

The techniques we present are essentially borrowed from previous works [26], [6] and [28].

5.1 Reduction of the Range Checks

Although a number of more optimized techniques for range-checks are known, we opt for the simplest possible one in our settings. During a preprocessing phase, we send oracle access to a vector $b = (0, 1, 2, \cdots, B-1)$ 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 Reduction of the Inclusion Checks

The technique we present is borrowed from the work of Halo2 [26] and is given in Fig. 2. Let $\{R_i\}_{i\in[m]}$ and $\{I_i\}_{i\in[m]}$ be two sets of columns such that all have the same size and I_i is included in the corresponding reference column R_i. As a convention, we use $v(X)$ to designate the polynomial which is encoding the associated vector v in Lagrange basis. For example, by $R_i(X)$ we mean the polynomial encoding of R_i obtained by interpolating the entries of R_i on a domain of m-roots of unity.

As one can see, the above construction converts an inclusion constraint to permutation, local and global queries.

5.3 Reduction of the Fixed-Permutation Checks

The technique we present is inspired by the work of [26] and [6]. 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 $B' = (B\|s')$.

Inclusion($\{I_i, R_i\}_{i\in[m]}$)

1. if $m > 1$:
   - Verifier samples $r \leftarrow \mathbb{F}$ and sends it to the oracle.
   - Prover and Oracle set $R'(X) = \sum_i r^i R_i(X)$ and $I'(X) = \sum_i r^i I_i(X)$

   else : They set $R'(X) = R_1(X)$ and $I(X) = I_1(X)$
2. Prover sends two polynomials $R^*(x)$ and $I^*(X)$ to the oracle defined as follows:
   - "Permutation" : $\{I^*, R^*\}$ is a permutation of $\{I', R'\}$
   - "Local" : $I^*[0] = R^*[0]$
   - "Global" : $(I^*(\omega X) - I(X))(R^*(\omega X) - I^*(\omega X)) = 0 \quad (*)$

Fig. 2. Reduction of the Inclusion Check.

5.4 Reduction of the Permutation Checks

Here we use the polynomial notation to denote what would be understood as vectors in the Wizard-IOP framework. Let P_1 and P_2 be polynomials that are computed as an interpolation of the same vectors up to a permutation. Namely, P_1 and P_2 are interpolations of $\mathbf{v}_1$ and $\mathbf{v}_2$, which are allegedly permutations of one another (the vectors are assumed to be of the same length l). The technique we present is borrowed from a series of works including [28], [6], [26] originating from the work of [10]. The intuition behind the protocol is as follows: a polynomial $P_1(X)$ is the permutation of $P_2(X)$ if and only if the grand-product associated with the first polynomial as $\prod_{i\in[l]}(X + v_{1,i})$ and the one from the second polynomial i.e., $\prod_{i\in[l]}(X + v_{2,i})$ are equal at a random point $X = \alpha$. Or equivalently

$$Z(\alpha) := \frac{\prod_{i\in[l]}(\alpha + v_{1,i})}{\prod_{i\in[l]}(\alpha + v_{2,i})} = 1$$

The pseudocode is given in Fig. 3 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 checks 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 oracle
   - Prover and Oracle set $A'(X) = \sum_i r^i A_i(X)$ and $B'(X) = \sum_i r^i B_i(X)$

   else : they set $A'(X) = A_1(X)$ and $B(X) = B_1(X)$
2. Prover sends $Z(X)$, the unique polynomial such that
   - "Local" : $Z(1) = 1$
   - "Global" : $Z(\omega X)(B'(X) + \alpha) = Z(X)(A'(X) + \alpha) \quad (*)$

Fig. 3. Reduction of a Permutation Check.

As one can see, the above construction converts permutation constraints into local and global constraints.

5.5 Reduction of the Scalar-Product Checks

As a reminder, Scalar-Product queries allow the verifier to query the scalar product of two committed polynomials (seen in Lagrange basis). We describe a technique to efficiently reduce a batch of scalar product queries into local and global constraints. This technique is derived from the univariate sumcheck described

in [15]. Let $a(X) = \sum_{i<n_H} a_i L_{\omega^i}(X)$ and $b(X) = \sum_{i<n_H} b_i L_{\omega^i}(X)$ be two polynomials of degree $n_H = |H|$. We also introduce:

$$p(X) = a(X)b(X) \bmod X^{n_H} - 1 = \sum_{i<n_H} p_i X^i$$

then we have that $\sum_{i<n_H} a(\omega^i)b(\omega^i) = \sum_{i<n_H} a_i b_i = n_H p_0 = n_H p(0)$ (due to the relation $\sum_i \omega_i^k = 0$ for $k \neq 0 \mod n_H$).

This naturally gives us a technique for compiling at once a batch of k scalar-product queries on $(a_1(X), \cdots a_k(X))$ and $(b_1(X), \cdots b_k(X))$ into global and local constraints. In Fig. 4, the reader can assume that the verifier already has oracle access to $(a_1(X), \cdots a_k(X))$ and $(b_1(X), \cdots b_k(X))$ and alleged scalar-product value $c_\bullet$ for each pair $(a_\bullet(X), b_\bullet(X))$ from the prover.

$\mathsf{ScalarProduct}(a_1, \ldots, a_k; b_1, \ldots, b_k; c_1, \ldots, c_k)$

1. the prover sends the polynomials $a_j(X)$ and $b_j(X)$ to the oracle.
2. The verifier sends a random challenge $r \leftarrow\$ \mathbb{F}$
3. The prover computes $P(X) = \sum_{j<k} r^j a_i(X) b_i(X) \mod X^n - 1$. Then, she sends oracle access to $P(X)$ to the verifier.
4. "Local" : sends query for $P(0)$ and expects $n_H \sum_{j<k} r^j c_j$
5. "Global" : $P(X) - \sum_{j<k} r^j a_j(X) b_j(X) = 0$

Fig. 4. Reduction of the Scalar-Product Check

5.6 Merging the Global Constraints

This simple compiler step essentially captures all the global constraints of the input Wizard-IOP. From then on, the compiler will group these queries into buckets according to the size of the associated domain. Coming back to the compiler description, once all queries have been grouped in buckets, the compiler generates a single global query per bucket by taking a random linear combination of the queries. The main objective of this step is to reduce the overhead of the next query.

5.7 Reduction of the Global Constraints

We present a standard technique from the work of Plonk [6]. Let $v_1, \cdots, v_k$ be k vectors of $\mathbb{F}^n$ 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,

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

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

5.7.1 Global Constraint Over Subsampled Vectors We may encounter the case where one of the vectors subject to a global constraint query, say, $v_\bullet$ is *subsampled* from an oracle-given vector w. In this case, we apply a variant of the above-described procedure. Let us assume $v_\bullet = \mathsf{Subsample}(i, p, w)$ where i, f, w are respectively the offset, the period and the original subsampled vector. We know that $p = |w|/n$ because the global constraint requires its "inputs" to be of size n. If we set $w' = \mathsf{Shift}(w, i)$ (cyclic-shift w by i), then we have that $v_\bullet = \mathsf{Subsample}(0, f, w')$. Now, using the fact that the polynomial encoding of $w'(X)$ agrees with $v_\bullet(X)$ over the n-th roots of unity, we simply use it instead of $v_\bullet(X)$ in the above-described procedure. As a result, the polynomial $Q(X)$ has degree $> (d-1)n$ (because $w'(X)$ has a larger degree than $v_\bullet(X)$). Thus, a drawback of this technique is that it increases the oracle complexity.

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

1. The prover computes and sends oracle access to $Q(X)$ computed as follows,

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

2. The verifier samples a random coin $\alpha \leftarrow\!\!\$\ \mathbb{F}$
3. The verifier makes the following query
   – "Univariate" : $v_1(\alpha) \ldots v_k(\alpha), Q(\alpha)$
   And checks $C(v_1(\alpha) \ldots) \stackrel{?}{=} (\alpha^n - 1)Q(\alpha)$

Fig. 5. Reduction of the Global Constraints

5.8 Reduction of the Abstract References

From this point on, the partially compiled Wizard-IOP only uses local constraints or univariate queries, possibly involving abstract references. We now discuss how to "eliminate" these abstract references from the protocol. For local constraints, it is quite straightforward. Since a local constraint involves opening a vector at a specific point agreed in an offline phase, we may simply shift the fixed opening position accordingly.

On the other hand, it remains to discuss how to convert univariate queries on abstract references into univariate queries "directly" on oracle-given polynomials (shown by P here). In the following, we summarize the possible conversions in a list of equivalence. Since abstract references can be composed with each other, the implicit conversion procedure must be repeated recursively.

$$\begin{aligned}
\mathsf{CyclicShift}(P, k)(x) = y &\iff P(\omega^k x) = y \\
\mathsf{Repeat}(P, k)(x) = y &\iff P(x^k) = y \\
\mathsf{Interleave}(P_1, \cdots, P_k)(x) = y &\iff \sum_{i \in [k]} P(\omega^{-i} x) Z_{n,nk}(\omega^{-i} x)
\end{aligned}$$

In the latter, n is the degree of each polynomial and $Z_{k,nk}(X)$ is the domain-selector (defined in Section 2.2). Note that if the domain for $\mathsf{Repeat}(P, k)$ is Ω_{nk}, then the domain for $P(X)$ is Ω_n, the subgroup of Ω_{nk}.

Regarding the "Subsampling", the verifier could in theory build the polynomial associated with the subsampling via Lagrange interpolation, but this requires many queries to the original polynomial. Instead, we play with the form of global constraints and follow the procedure of Section 5.7.1. Therefore, in the current state of this work, there is a small restriction: subsampling can only be used "at the top". Namely, "subsampling" may only be used at the "top" and cannot be used in univariate queries directly. When we have global constraints over a subsampled vector, we use the variant mentioned above (Section 5.7.1 by changing the domain of the global constraint.

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 PIOP to a PIOP where the oracle-given polynomials are all queried at a single random point. We will call such a PIOP scheme a UniEval PIOP, 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 PIOP 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.

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. 6 we present our batching protocol for the relation Eq. (1). The protocol is inspired by the batching approach presented in [19].

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

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. 6. Multi-point to single-point reduction procedure.

6.2 Compiler: PIOP to UniEval PIOP

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

– For any PIOP, define its associated protocol PIOP′ as follows; we let all the queries in PIOP 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 PIOP′ is the same as PIOP where the prover also plays the role of the oracle by itself.

– By the end of an execution of PIOP′, we get the trace of the polynomial queries issued during PIOP′; 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. 6, 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 PIOP′, 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 $\epsilon_{\text{UniEval}}$, $\epsilon_{\text{PIOP}'}$ and ϵ_{MPSP} be, respectively, the soundness-error of protocols UniEval-PIOP, protocol PIOP′ and $\text{MPSP}(R_i, S_i; P_i(X))_i$. Then, we have, $\epsilon_{\text{UniEval}} \leq \epsilon_{\text{PIOP}'} + \epsilon_{\text{MPSP}}$.

7 Vortex, List Polynomial Commitment

Vortex is a variant of the commitment scheme proposed in Orion [47] and Brakedown [30], and it relies on MIMC [4], also a lattice-based hash which we describe in Section 7.1, and an erasure-code. 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.

For a matrix of size $m \cdot n = N$, the Vortex commitments 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 [38] 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. 7

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 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

[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. 7. Description of the lattice-based hash

small values of n and b, other techniques such as Tom-Cook are known to be efficient as well. In Appendix A we recap the security analysis of this hash function and give concrete parameters for a target bits 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 $\forall i < m, w_i \cdot l = y_i$ if W is a batch of polynomials, for $l := (1, x, x^2, \cdots, x^k)$. The verifier then sends the random scalar β, and the prover responds with u claimed to be $u := \mathcal{B}^\top W$, if W is polynomial, where $\mathcal{B} = (1, \beta, \beta^2, \ldots, \beta^{m-1})$. 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 $\mathcal{B}^\top s_i \stackrel{?}{=} u'_{q_i}$
   - the hash of s_i is correct and consistent with h_{q_i}.
   - **Evaluation Check:** the relation $u(x) \stackrel{?}{=} \mathcal{B}^\top \cdot y$ where the vector u is considered as the coefficient of polynomial $u(x)$

   The first check (the random combination over random columns), is used for checking the proximity of a batch in [5]. Fig. 8 sums up the above.

[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'.

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

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)

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

Fig. 8. Vortex Polynomial commitment

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 the construction of the Merkle tree needs to be modeled as a random oracle for the scheme to retain extractability.

Security of Vortex. Vortex satisfies the knowledge-soundness of (batched) list polynomial commitment given in Definition 12.

Vortex 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 provides us with a way to switch between the definitions of batched polynomial commitments to a version that only commits to one polynomial.

8 AOK from PIOP and Vortex

In [22, 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 [34], 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 a 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 [34, 16].

Slightly more formal, let (P_O, V_O) be a PIOP for the relation $\mathcal{R}$ that is transformed to a 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.

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 $|L|r/|\mathbb{F}|$.

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 hash values over the chosen columns are computed correctly
- The scalar-product of chosen columns and the vector $\mathcal{B}$ are computed correctly.
- The encoding $\mathsf{encode}_{\mathcal{L}}(u)$ is correctly computed as u'.
- The opened columns are the right ones.

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 again 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 while, 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 erasure-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.

The dedicated PIOP protocols for self-recursion would be discussed in future works as we advance on the final design.

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.

Shorter Proof Size Optionally, it is possible to further compress the proof by a final standard recursion (e.g., PLONK or Groth16 over the output of our scheme) for non-interactive proof systems. At a high level, we wrap the verifier's computations inside an arithmetic circuit. Since the self-recursed protocol is a public-coin protocol, we compile it into a non-interactive protocol using the Fiat-Shamir transform. The random oracle is instantiated using a SNARK-friendly hash function, such as Poseidon or RC-Concrete [31, 9]. The underlying field of the arithmetization can differ from the underlying field of the self-recursed protocol. Doing so comes with a multiplicative overhead in the size of the arithmetic circuit. Fortunately, the prior self-recursion strategy already ensures that the proof to verify is already somewhat small. As a result, we get a very short proof with a better prover time. We leave for future work the details of the concrete SNARK scheme that we may use and of how we implement the verifier in the circuit.

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 et al. "Estimate all the {LWE, NTRU} schemes!" In: *IACR Cryptol. ePrint Arch.* 2018.
[4] Martin R. Albrecht et al. "MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity". In: *ASIACRYPT*. Vol. 10031. LNCS. 2016, pp. 191–219.
[5] 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).
[6] 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.
[7] Shi Bai et al. "Improved Combinatorial Algorithms for the Inhomogeneous Short Integer Solution Problem". In: *J. Cryptol.* (2019).
[8] Shi Bai et al. "Improved Combinatorial Algorithms for the Inhomogeneous Short Integer Solution Problem". In: *J. Cryptol.* (2019).
[9] Mario Barbara et al. *Reinforced Concrete: Fast Hash Function for Zero Knowledge Proofs and Verifiable Computation*. Cryptology ePrint Archive, Report 2021/1038. `https://ia.cr/2021/1038`. 2021.
[10] Stephanie Bayer and Jens Groth. "Efficient Zero-Knowledge Argument for Correctness of a Shuffle". In: *EUROCRYPT*. 2012.
[11] Olivier Bégassat et al. *A ZK-EVM specification*. 2022.
[12] 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`.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

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

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

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

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

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

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

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

[29] Shafi Goldwasser, Yael Kalai, and Guy Rothblum. "Delegating Computation: Interactive Proofs for Muggles". In: *ACM STOC*. ACM, 2008, pp. 113–122.

[30] Alexander Golovnev et al. "Brakedown: Linear-time and post-quantum SNARKs for R1CS". In: *IACR Cryptol. ePrint Arch.* 2021 (2021), p. 1043.

[31] Lorenzo Grassi et al. "Starkad and Poseidon: New Hash Functions for Zero Knowledge Proof Systems". In: *IACR Cryptol. ePrint Arch.* (2019), p. 458.

[32] Jens Groth. "On the Size of Pairing-Based Non-interactive Arguments". In: *Advances in Cryptology - EUROCRYPT*. Vol. 9666. LNCS. Springer, 2016, pp. 305–326.

[33] Nick Howgrave-Graham and Antoine Joux. "New Generic Algorithms for Hard Knapsacks". In: *Advances in Cryptology – EUROCRYPT 2010*. 2010, pp. 235–256.

[34] Assimakis Kattis, Konstantin Panarin, and Alexander Vlasov. "RedShift: Transparent SNARKs from List Polynomial Commitment IOPs". In: *IACR Cryptol. ePrint Arch.* (2019), p. 1400.

[35] 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.
[36] Jianwei Li and Phong Q. Nguyen. "A Complete Analysis of the BKZ Lattice Reduction Algorithm". In: *IACR Cryptol. ePrint Arch.* 2020 (2020), p. 1237.
[37] Benoît Libert, Somindu C. Ramanna, and Moti Yung. *Functional Commitment Schemes: From Polynomial Commitments to Pairing-Based Accumulators from Simple Assumptions*. 2016.
[38] 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.
[39] 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.
[40] Daniele Micciancio and Oded Regev. *Class on lattice-based cryptography*. 2008.
[41] 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.
[42] Srinath Setty. *Spartan: Efficient and general-purpose zkSNARKs without trusted setup*. CRYPTO. 2020.
[43] 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.
[44] 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.
[45] Polygon Hermez Team. *Scalable payments. Decentralised by design, open for everyone*. https://hermez.io.
[46] 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.
[47] Tiancheng Xie, Yupeng Zhang, and Dawn Xiaodong Song. "Orion: Zero Knowledge Proof with Linear Prover Time". In: *IACR Cryptol. ePrint Arch.* 2022.
[48] Risc Zero. https://github.com/risc0/risc0. 2022.
[49] Jiaheng Zhang et al. *Transparent Polynomial Delegation and Its Applications to Zero Knowledge Proof*. 2020 IEEE Symposium on Security and Privacy (SP). 2019.

A 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 [38]. 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.

A.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 [25] 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.

A.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.

A.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 $q, n, m' = nm, b$.

A.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 $m_0 < m'$ 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,

$$b_2 = \delta^{m_0} q^{n/m_0} \wedge b\sqrt{m_0} < q \tag{3}$$

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.

There are two strategies to choose b_2.

– **Pessimistic** Pick b_2 to be the radius of the smallest ball (not centered at 0) that contains $[0; b[^{m_0}$. In that case, from the Minkowski bound,

$$b_2 = \sqrt{m_0}\frac{b}{2}$$

.

– **Heuristic** Pick b_2 to be the radius of a ball whose volume equals b^m. This gives us

$$b_2 = b\frac{\Gamma(m_0/2+1)^{1/m_0}}{\sqrt{\pi}}$$

For our parameters, we pick the heuristic approach.

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 you 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 [3]. 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 [44], 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.

$$\log t_{\mathsf{oracle}} = 0.292k + 16.4 \tag{4}$$

In a recent work, [36] gives a refined estimation of the overall runtime of BKZ2.0 (number of calls to the oracle) alongside a lower-bound of the achieved root-Hermite factor. In [36], they give a lower bound for the L2 norm of the first vector of the output basis, but we worked out the root-Hermite factor. We present their result in the two equations below. ρ gives the total number of calls to the oracle and the second expression is a lower-bound on the obtained δ.

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

$$\log\delta = \frac{1}{2m_0(k-1)}(m_0 - 1 + \frac{k(k-2)}{m_0})\log\gamma_k \tag{5}$$

γ_k is a mathematical constant: the k-th Hermite constant. We do not have a closer formula for it. It is related to the density achieved by optimal sphere packing in dimension k. LN20 [36] uses it because they wanted an upper-bound in the running time of an SIS instance for all existing lattices with given dimension k. In practice, we use random lattice instances, and we instead use estimations of the density of a random lattice instance. Thus, we use a term obtained using the Gaussian heuristic instead (as it is advised by the authors of LN20) and this will give us Eq. (6):

$$\log \delta = \frac{1}{4m_0(k-1)} \left(m_0 - 1 + \frac{k(k-2)}{m_0} \right) \left(\log \frac{k}{2\pi e} + \frac{1}{k} \log \pi k \right) \tag{6}$$

One should note that Eq. (6) is *only* asymptotically correct. Thus, we will only use it for $k > 36$. This is to avoid inaccuracies from using values out of the range. This value was obtained from an experiment where we increased k and m_0 with $k = m_0$. The values of $\log \delta$ we obtained were growing for $k < 36$ (which led to the decision to discard them) and decreasing for $k > 36$. In practice, we have only retained parameters-values for which $k > 200$ thus the latter is not a concern here.

A.5 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.

A.5.1 Camion-Patarin and Wagner attacks The course [40] describes the basic version of these attacks and gives an easy procedure to determine their efficiency. The attack is also known as CPW. In [7], 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 [7] 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 [40]. From [40], 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 [40]. 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*.

A.5.2 On the HGJ and BCJ refinements Howgrave-Graham and Joux introduced these techniques in 2010, [33]. 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.

Definition 18 (Density of a SIS instance). *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.*

A.5.3 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).

A.5.4 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 19 (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.

A.6 Concrete parameters

Based on the above analysis, we have run a parameter selection. The table below 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 $X^n + 1$.

$log_2(q)$	$log_2(\beta)$	n	BKZ attack	CPW attack
64	2	32	182.17	144.0
64	4	64	147.31	305.57
64	6	128	166.13	598.14
64	10	256	149.93	1272.31
64	16	512	136.4	2741.67
64	22	1024	160.7	5967.82
254	2	7	157.7	259.03
254	4	16	146.1	270.0
254	6	32	164.73	637.0
254	10	64	148.63	1262.46
254	16	128	135.18	2720.33
254	24	256	133.28	5921.27
254	32	512	164.03	13013.8