Vortex : Building A Lattice-based SNARK scheme with Transparent Setup

Alexandre Belling and Azam Soleimanian

Consensys R&D
{firstname.lastname}@consensys.net

Abstract. We present the first transparent and plausibly post-quantum SNARK relying on the Ring Short Integer Solution problem (Ring-SIS), a well-known assumption from lattice-based cryptography. At its core, our proof system relies on a new linear-commitment scheme named *Vortex* which is inspired from the work of Orion and Brakedown. Vortex uses a hash function based on Ring-SIS derived from "SWIFFT" (Lyubashevsky et al., FSE08). We take advantage of the linear structure of this particular hash function to craft an efficient self-recursion technique. Although Vortex proofs have $O(\sqrt{n})$ size in the witness size, we show how our self-recursion technique can be used to build a SNARK scheme based on Vortex. The resulting SNARK works over any field with reasonably large 2-adicity (also known as *FFT-friendly fields*). Moreover, we introduce Wizard-IOP, an extension of the concept of polynomial-IOP. Working with Wizard-IOP rather than separate polynomial-IOPs provides us with a strong tool for handling a wide class of queries, needed for proving the correct executions of the complex state machines (e.g., zk-EVM as our use-case) efficiently and conveniently.

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

1 Introduction

Functional Commitment Functional commitment [42] is a cryptographic primitive in which a prover may commit to a function f and later prove evaluation of f at any point. In this work, we mainly focus on two types of functional commitments: linear commitments where f is restricted to be a linear function and polynomial commitments where f is a polynomial. An interesting observation is that polynomial commitment is a special form of linear commitment, since $P(x)$ can be written as the scalar-product of two vectors $(p_0, \ldots, p_{d-1})$ and $(1, x, \ldots, x^{d-1})$, for $P(X) = \sum_i p_i x^i$.

Succinct Non-Interactive Argument of Knowledge (SNARK) 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 with respect to a public input x. In particular, the verifier needs less time to verify the proof, generated by 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 [36], Plonk [5], Halo [22], Halo2 [30], Marlin [28], Spartan [46], Virgo [55], LegoSnark [24], Hyrax [50], HyperPlonk [26], Brakedown [34], Orion [53], Libra [52], Aurora [16], Fractal [27], Sonic [44] 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, 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 be verified in a few milliseconds on any platform (Groth16 [36]). For these reasons, zk-VMs have important applications in blockchain scalability and blockchain interoperability. This is also the reason why this area of research has recently seen tremendous activity in research and development: Consensys-zkEVM [12], Cairo [32], Polygon-zkEVM [48], RISC [54], 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 [12] 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 Proof Interactive Oracle Proof (IOP) is a family of abstract ideal protocols in which the verifier is not required to read the prover's messages in full. Instead, the verifier has oracle access to the prover's messages and may probabilistically query them at any positions [14]. IOP protocols can be transformed into concrete secure argument systems using a Merkle-Tree. Later works have introduced several variants of IOP such as polynomial-IOP or tensor-IOP, where the prover can perform polynomial evaluation queries [5] or tensor queries [19]. Similarly, these protocols can be converted into concrete argument systems (including SNARK) using functional extractable commitments. This type of approach for building argument systems has proven to be fruitful and has now become a standard.

Recursion is a technique that consists in 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. [17] specifies how to instantiate proof-carrying data through recursion using a pairing-friendly cycle of elliptic curves. The following papers such as Halo [22], Halo2 [30] and Nova [40] present several techniques to implement PCD or IVC using a (possibly non-pairing-friendly) cycle of elliptic curves. The works of Fractal [27] and Plonky2 [49] also explore how to implement recursion for SNARKs based on hash functions. In [13] the authors presented a recursion technique that specifically targets recursion over the protocol of GKR [33] and more generally any interactive protocol whose Fiat-Shamir transform involves hashing long string in the first round.

1.1 Our Contributions and Techniques

Wizard-IOP In Section 4, we present the Wizard-IOP framework. It can be viewed as an extension of the notion of (polynomial-)IOP [15] (supporting more complex queries). In this framework, the prover is allowed to send oracle-access to multiple vectors across several rounds of interactions and the verifier may perform queries from a wide class. To give an idea, the verifier may send queries evaluating scalar-products of committed vectors or polynomial evaluations. He may also send queries involving cyclic-shift 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-IOP, Wizard-IOP offers a higher-level framework for designing protocols. This makes Wizard-IOP more 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 [12].

Arcane, a compilation framework Thereafter, Section 5 introduces the Arcane Compiler, a tool that allows transforming any secure protocol specified in the Wizard-IOP model into one that is secure in the polynomial-IOP model and in which the verifier only queries a single opening point for all polynomials. The techniques we use to build Arcane derive from known modular polynomial-IOPs from past works such as Plonk, Halo2 or Cairo [5, 18, 30, 32, 31]. As the original goal of our work is to build a succinct proof system for the zk-EVM specified in [12], this compiler approach has numerous benefits. An important one is that it allows specifying and implementing batching and optimization techniques that would be a lot 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 Arcane is that it yields a single-point evaluation PIOP, allowing us to use the output of Arcane alongside a non-homomorphic polynomial commitment (i.e., Vortex) to create an efficient argument system.

Vortex, a lattice-based Batchable Linear Commitment (BLC) In Section 6, we present Vortex. Vortex is a variant of the linear commitment scheme presented in the work of Brakedown and Orion [34, 53]. It relies on a hash function based on the Ring-SIS assumption (referred to as Ring-SIShash throughout this work) and a systematic erasure code. The first instance of Ring-SIS-based hash functions was introduced in [43]. It is a SNARK-friendly hash function with 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 $\mathcal{R} = \mathbb{Z}_q(X)/X^d + 1$.

As mentioned earlier a linear commitment may allow a prover to open the scalar product of a committed vector by a public vector chosen by the verifier. A Batchable Linear Commitment (BLC) allows the same type of opening for a batch of committed vectors with the same public vector. Equivalently, we may say that a BLC allows one to commit to a matrix M and open the matrix-vector product of M by a public vector. 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 [53, 34] achieve linear-time prover 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. We believe that the reliance on Ring-SIS hash functions is an important novelty of our work. This family of hash functions has the benefit of being at the same time SNARK-friendly, comparatively as fast as standard hash functions and, most of all, they rely on very well-studied cryptographic assumptions [2]. On the last point, Pedersen Hashes [37] and the Sinsemillia hash function [30] also rely on well-studied assumptions for collision resistance. Ring-SIShash compares favorably to them both in native execution time and “SNARK-friendliness”. However, (as for Sinsemillia and Pedersen hash) the function that we use is not suitable to be used as a Random Oracle or a PRF [1].

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, being a SNARK requires polylogarithmic proof-size and non-interactivity. To work around this problem, we use a self-recursion technique that recursively shrinks the size of the proof to the square root at each step. 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 using a suitably chosen hash function. Thereafter, the obtained SNARK can optionally be compressed further to $O(1)$ using existing proof systems such as Groth16[36] or Plonk[5] whose concrete proof size and verification time are tiny. 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

[1] for instance, the hash of zero is zero

say that self-recursion compresses the proof loosely but fast, while recursion with pairing-based SNARKs compresses the proof tightly but slowly.

1.2 Related Work

In [51, 4, 8] the authors presented vector and linear commitments based on lattice assumptions that can be combined with an IOP to build SNARK. But these works need trusted setups, resulting in a SNARK with a trusted-setup. Moreover, they use new (falsifiable) assumptions, all new variants of SIS assumption.

The work of [25] presents a functional commitment from the standard SIS assumption. For linear commitment and polynomial commitment, the size of the proof is linear w.r.t the length of the associated vectors. They also consider *selective-binding* where the adversary must name the input on which it will attempt to break binding before seeing the public parameters. This can be lifted to the more realistic notion of adaptive binding (via complexity leveraging), up to a loose reduction by a factor of the size of the input [25].

The work of [21] presents a commitment based on techniques inspired by the work of Bulletproof [23] (to prove the correctness of scalar-product) that imposes overheads on the choice of the SIS instance due to the “slack” (that is, the ratio between the original committed vector and the extracted one [21]). This construction can be extended to our linear commitment (by applying Bulletproof twice: once over the commitment, once over the scalar product), but as mentioned, with the cost of larger parameters.

The work [10] presented a sublinear argument system (which may not be considered SNARK based on the definition of succinctness) based on lattices. Furthermore, their scheme also has “slack”.

To the best of our knowledge, our work is the first to present a transparent IOP-based SNARK relying on well-studied assumptions from lattice-based cryptography.

In Orion [53], the authors adapt the linear-commitment of Brakedown [34] that has a large proof size and then built a proof composition technique to reduce the proof size of their linear-commitment from square root to polylogarithm size. They also use optimized erasure codes to work with any field (including non-FFT-friendly fields). There are two main steps in the proof. First, the prover sends a commitment to a vector u'. Then, she runs a second generic SNARK to prove that she knows a witness W (batch of vectors) that is consistent with u' at some random positions Q. The generic SNARK should prove the encoding and scalar product used during the protocol. They use a SNARK with quasi-linear prover-time and polylogarithmic proof size in the length of vectors (number of rows in W). Since the generic SNARK that they use is itself a code-based scheme possibly using a different code, the author named this technique *code-switching*.

Similarly to Orion, we decrease the size of the proof, but with a different and specific (rather than generic) technique. We benefit from the linear form of Ring-SIShash to provide a proof for correct computation of the hashes and the scalar products that occur during the protocol verification. This arises new queries that are already supported by Wizard-IOP that again uses our linear commitment scheme with possibly different parameters. Hence the name "self-recursion" rather than "recursion". Compared to the *code-switching* of Orion [53], our self-recursion technique not only allows using codes with a different rate but it also allows using a different lattice for the hashes (lattice-switching).

1.3 Overview of Vortex and its Self-Recursion

Vortex As for [53] and [34], the Vortex construction is rather simple and can be succinctly described. Assume that $\mathcal{P}$ and $\mathcal{V}$ are, respectively, the prover and the verifier. First, we elaborate on the commitment procedure. The prover commits to a matrix W of size n as follows. $\mathcal{P}$ starts by encoding the rows of the matrix using a systematic erasure code to obtain a new matrix W'. The prover then hashes each column of W' and sends them to the verifier as a commitment. Thereafter, the verifier wishes to know if $u = l \cdot W$ for some vector l drawn at random. The verifier encodes u to u' using the erasure code and randomly queries the openings of t columns of W'. For the opened columns, the verifier checks two things: (1) whether the alleged columns openings are consistent with the corresponding hash values from the commitment and (2) whether the scalar-product of l and the chosen columns are consistent with u'.

The above description is the same as the linear commitment in Brakedown [34] and Orion [53]. However, the first difference is that we explicitly use Ring-SIShash for the columns of W'. SIS hash can be seen as a variant of the SWIFFT hash function [43]. 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 its 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 In order to convert Vortex-transform of (P)IOP protocols into SNARKs, 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 on its turn again be 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 which can be compiled into a SNARK using the Fiat-Shamir transform. Our self-recursion technique relies heavily on the fact that Vortex's verifier uses Ring-SIShash 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. Among the various techniques that we use, we highlight the *Horner Protocol* a dedicated Wizard-IOP for proving that a vector of polynomials whose coefficients are packed into a committed vector evaluates into another committed vector. This is a protocol specific to the SIS setting (polynomials in the SIS), that can prove the correct opening of many SIS polynomials in a batch without outputting the openings themselves. More precisely, as an observation, as we use the openings of SIS polynomials internally, the prover does not require sending the openings. Proving knowledge of their correct computations is enough.

2 Preliminaries

Here we define the syntax of our main building blocks; SNARK and Polynomial Commitment.

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. We show the set of true statements by $\mathcal{L}_\mathcal{R} = \{x \ : \ \exists\ w \quad \mathcal{R}(x; w) = 1\}$. The definitions in this section are mainly borrowed from [36].

Definition 1 (*non-interactive* Argument 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 $\mathcal{R} \leftarrow \mathcal{R}_\lambda$, it 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 and for $\mathcal{R}(x; w) = 1$ 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:*

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

The advantage of the adversary in the knowledge-soundness game (the probability on the left side) is called knowledge-error.[2]

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.

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

[2] Although, we only use the notion of knowledge-soundness throughout this work. The reader should be aware, a more general notion exists, *witness-extended emulation.* Where the extractor also outputs an (indistinguishable) transcript of the protocol.

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 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 commitment

We conveniently adapt the definition of polynomial commitment given by [5, 18] (to its non-interactive version) to match the formalism of the present document. Formally, a polynomial commitment is a tuple of p.p.t. algorithms (Setup, Commit, Prove, Verify) where,

- $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, t)$ generates the public parameters pp suitable to commit to polynomials of degree $< t$. It is to be done by a trusted authority.
- $C \leftarrow \mathsf{Commit}(\mathsf{pp}, P(X))$ outputs a commitment C to a polynomial $P(X)$ of degree at most t using pp.
- $(x, y, \pi_x) \leftarrow \mathsf{Prove}(\mathsf{pp}, P(X), x)$ outputs (x, y, π_x) where π_x is a proof for the evaluation of $y = P(x)$.
- $0/1 \leftarrow \mathsf{Verify}(\mathsf{pp}, C, x, y, \pi_x)$ verifies that $y = P(x)$ is the correct evaluation of the polynomial committed in C.

Here we define the correctness and the security of a polynomial commitment scheme.

Definition 7 (Correctness). *We say that a polynomial commitment scheme has (perfect) completeness if for all $P(X), t, \lambda, x$,*

$$\Pr\left[1 \leftarrow \mathsf{Verify}(\mathsf{pp}, C, y, \pi) \ : \begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, t) \\ C \leftarrow \mathsf{Commit}(\mathsf{pp}, P(X)) \\ \pi \leftarrow \mathsf{Prove}(\mathsf{pp}, P(X), x) \\ y = P(x) \end{array}\right] = 1$$

Definition 8 (Secure polynomial commitment [18]). *A polynomial commitment (Setup, Commit, Prove, Verify) is secure if it is knowledge-sound w.r.t the relation,*

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

2.4 IOP and Polynomial-IOP

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 the messages are polynomials and the verifier has oracle access to the evaluation of polynomials on the queried points.

2.5 Erasure code

Definition 9. *Let Σ be the alphabet. A* ***code*** *$\mathcal{C}$ of size k over Σ is a collection of words (codewords) of k element from Σ. Let $h : \Sigma^k \times \Sigma^k \to \mathbb{N}$ be the hamming distance defined over Σ^k. A code has* ***minimal distance*** *d if for all distinct pair of codewords w_i, w_j we have that $h(w_i, w_j) \geq d$.*

Definition 10. *Let $\mathbb{F}$ be a field and the alphabet. A code $\mathcal{C}$ is* ***linear*** *if linear combinations of codewords are also codewords. More precisely, if it is a vector subspace of $\mathbb{F}^k$.*

Definition 11. *An* ***erasure correction code*** *is a tuple $(\mathcal{C}, \mathsf{Encode}_{\mathcal{C}}, \mathsf{Decode}_{\mathcal{C}})$. Where $\mathcal{C}$ is a code of size k over an alphabet Σ,* Encode *is an algorithm that defines a function $\Sigma^n \to \mathcal{C}$ where $n < k$. We call Σ^n the message space.* Decode *is an algorithm that can correct up to d erasures from a codeword and recovers the original message word. The fraction k/n is known as the rate of the code.*

Definition 12 ((Systematic) Reed-Solomon code). *is a family of erasure codes with $O(nlogn)$ encoding time (for messages of length n) where the encoding is as follows. Let v be the message and $|v| = n$. It builds the polynomial interpolation to the entries of v over $(\omega_i)_{i=0}^{n-1}$ i.e., $v(X) = \sum_i v_i \mathcal{L}_i(X)$. Then, it evaluates this polynomial over $k - n$ more points. Namely, $v(\omega_j)$ for $j = n, \ldots, k-1$. We call the vector $\mathsf{Checksum}_v = (v(\omega_j))_{j=n}^{n'-1}$ as the checksum. The codeword associated with v is $(v||\mathsf{Checksum}_v)$.*

2.6 A General Security Proof for Sub-Protocols

Apart from the security of Vortex that follows naturally from the proof in Brakedown and Orion [34, 53], but based on the collision resistance of Ring-SISHash, 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*

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

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.

Lemma 2 (Knowledge-Soundness of PIOP). *A PIOP is knowledge-sound if it can reduce the claim equivalently to Global constraints and if the size of the finite field is large enough (to have negligible probability $d/|\mathbb{F}|$ in 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 AOK we have,

Lemma 3 (knowledge-soundness of AOK). *If the PIOP and the polynomial commitment are knowledge-sound, then the AOK is knowledge-sound as well.*

Security Analysis of Sub-Protocols.

Initially, all our sub-protocols are presented in the PIOP framework, where the claim is (equivalently) converted to local and global constraints over the polynomials (and over a large domain). From there by the Schwarts-Zippel Lemma, the constraints are satisfied if and only if the constraints are satisfied over a single random point. This guarantees that the PIOP is knowledge-sound. Then, one uses a polynomial commitment to convert PIOP to AOK, this is what converts the sub-protocols in the ideal PIOP model to a concrete AOK which can be used in our construction. The resulting sub-protocols are knowledge-sound thanks to the fact that polynomial commitment and PIOP are knowledge-sound.

Putting everything together, all the sub-protocols are knowledge-sound thanks to the Schwarts-Zippel lemma (and equivalence of claim with constraints) and knowledge-soundness of polynomial commitment.

3 Overview of compilation

We present here the set of techniques we use for proving the execution of a zk-EVM. Namely, the zk-EVM of [12] 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 4 major axis: the Wizard-IOP model, the Arcane compiler, the Vortex commitment scheme and a self-recursion technique.

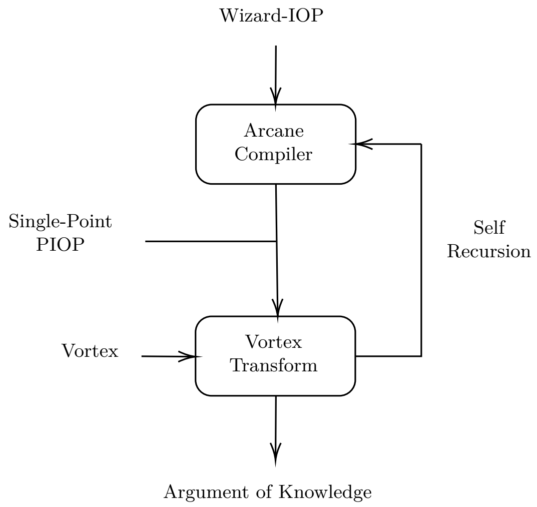

Fig. 1. Global structure of the prover

4 Wizard IOP

As mentioned, in an IOP protocol[15] a prover $\mathcal{P}$ sends oracle access to (possibly large) messages to a verifier $\mathcal{V}$. The verifier then can send certain kinds of queries (from a small family) to the oracle. Several variants of IOP exist in the literature. In particular, polynomial-IOP [44], [28], [5] is a model where all prover messages are viewed as polynomials and the verifier may query evaluation of these polynomials at random points of his choice. More recent works study tensor-IOP [19] 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-IOP is a model pushing this perspective on IOP a bit further. Here, the prover may send 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 reference" 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 lets us specify ever more complex protocols in the simplest possible way, while intermediate protocol design techniques (such as proving a lookup relation or a permutation relation) are treated as automatable compilation steps. Subsequently, instead of mentally building modular protocols from the bottom up using the notion of univariate queries as atoms of a more complex system, the framework of Wizard-IOP allows specifying protocols with a top-down approach. We start from 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 automating optimizations and batching techniques. Most of all, the zk-EVM arithmetization specified in [12] involves hundreds of polynomials and thousands of constraints. It would be unthinkable to unfold all the sub-protocols and optimization techniques that we would need to present a concrete polynomial-IOP for this arithmetization. But since their description is written in a formalism closely matching the Wizard-IOP model, we can directly transpile their arithmetization into a Wizard-IOP. Besides, thinking in terms of compilation steps rather than sub-protocols helps to maintain our work. Assume a new (purely hypothetical) batching technique for "range-check" is discovered and improves the prover's runtime by a factor of 2, then we can simply add it in the compiler and this will echo 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

We hereby list and describe the queries available to $\mathcal{V}$.

Range check Given a bound B known beforehand. The query is made over a vector v, the oracle responds 1 if and only if all entries v_i of v satisfy $0 \leq v_i < B$. We shall denote the range checks as, "Range" : $v < B$.

Inclusion check Given two lists of vectors (seen as matrices) S and T 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 have that the row i in S must equal the row $\sigma(i)$ in T. If so, the oracle returns 1, else 0. Thus, 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 σ exists, whatever it is). We denote a permutation query as, "Permutation": $S \sim T$.

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

Local Constraint The oracle returns the values for the queried positions and the verifier expects that these values satisfy a specific relation. 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 vector $v_1, \cdots, v_k$ of the same size n. The oracle return 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 one. 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 engaged polynomials (the one with the maximum size), Namely, for polynomials of degree d, the global constraint should be satisfied over the domain $X^d - 1$. By this fact, 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 $v(X)$ evaluates 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. 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 for instance. Note that abstract references are neither queries nor they are committed vectors but they can be seen as a way to make queries about committed vectors more expressive.

Subsampling Given access to a vector v of size n, an offset i and a sampling period k such that $k|n$ and $i < k$. Returns the 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 $v_1, \cdots, v_k$, returns 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). Returns a cyclically shifted version of v by k elements. We may use the notation $\mathsf{Shift}(v; k)$ to refer to the obtained vector.

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

5 Arcane Compiler, Polynomial IOP from Wizard IOP

The Arcane compiler transforms Wizard IOP into a Polynomial IOP. Arcane is organized as a sequence of compilation steps, each of them carrying a small transformation with them. A transformation can be either a small optimization or a reduction technique that transforms an “abstract” query into more “simple” queries. Applying these compilations steps one after the other produces step after step Wizard-IOP that uses fewer types of queries. To give a more tangible idea, Arcane starts by removing the range check 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 [30], [5] and [32].

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 [30]. 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 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.

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

1. if $m > 1$ :
     – Verifier samples $r \leftarrow \mathbb{F}$ and sends 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 polynomial $R^*(x)$ and $I^*(X)$ to the oracle defined in a way that
     – "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. Inclusion Check.

As one can see, the above construction converts 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 [30] and [5]. 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 in 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 if suited and since the polynomial encoding of s is $s(X) = X$ there is implicitly no need to send it to the oracle. The compiler then replaces every fixed permutation query on A and B by a permutation query on $A' = (A\|s)$ and $B' = (B\|s')$.

5.4 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 interpolation to the same vectors up to a permutation. Namely, P_1 and P_2 respectively interpolate $\mathbf{v}_1$ and $\mathbf{v}_2$ which are allegedly permutations of one another and of length l. The technique we present is borrowed from a series of works including [32], [5], [30] originating from the work of [11]. 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 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" : $Z(\omega X)(B'(X) + \alpha 1) = Z(X)(A'(X) + \alpha)$   (*)
```

Fig. 3. 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 reduction allows 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 [16]. 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

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

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. 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 overheads of the next query.

5.7 Reduction of the Global constraints

We present a standard technique that we owe to the work of Plonk [5]. 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.

$\mathsf{ReduceGlobalConstraint}(C, v_1, \cdots, v_k)$

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" : $v_1(\alpha), \cdots, v_k(\alpha), Q(\alpha)$

   And checks $C(v_1(\alpha), \cdots) \overset{?}{=} (\alpha^n - 1)Q(\alpha)$

Fig. 5. Global Constraints

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.

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 on offline, 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 needs many queries to the original polynomial. Instead, we play with the form of global constraints and follow 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.

5.9 Single-point Univariate queries from multiple Univariate queries and Local constraints

In the last step, the verifier makes only univariate queries, either at arbitrary random points (univariate evaluation queries) or at fixed points (local constraints). The goal of this step is to reduce into a protocol where all the oracle-given polynomials are all queried at a single point. To achieve this, we leverage an idea of [18] that allows batching polynomial opening in the context of the KZG polynomial commitment scheme [39]. We adapt this method for polynomial-IOPs as follows.

We assume a set of opened points S, a set of n polynomials $\{i \in [n] : P_i(X)\}$ each of degree d_i. Where $P_i(X)$ is queried in a set of evaluation point $S_i \subset S$. For $x \in S$, let $L_{x,S}(X)$ be the Lagrange polynomial corresponding with the point $x \in S$. Finally, let $R_i(X) = \sum_{x\in S_i} \hat{y}_{x,i} L_{x,S_i}(X)$ be the polynomial mapping the opening point of S_i to their alleged evaluations $\hat{y}_{x,i}$. More precisely, $P_i(x) = \hat{y}_{x,i}$ and $R_i(X)$ agrees with $P(X)$ over S_i. The compilation routine we describe in what follows results from the observation that,

$$\forall i \in [n] : (P_i(X) - R_i(X)) \prod_{x \notin S_i} (X - x) \text{ is divided by } \prod_{x \in S} (X - x)$$

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

1. The verifier samples $\alpha \leftarrow\$ \mathbb{F}$
2. The prover computes and sends oracle-access to

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

3. The verifier samples $x \leftarrow\$ \mathbb{F}$ and queries
   - "Univariate": $P_1(x), \cdots P_n(x), Q(x)$

   Finally, he checks

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

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

6 Vortex, lattice-based linear commitment

Vortex is a variant of the commitment scheme proposed in Orion [53] and Brakedown [34], and it relies on a lattice-based hash, which we describe in Section 6.1, and an erasure-code. In this work, we use the systematic version of Reed-Solomon which has encoding time $O(N \log N)$, where N is the size of the codeword. Vortex allows performing a batched argument for scalar-product opening for multiple committed vectors at once by the same public vector. Vortex is described in Section 6.2. Vortex commitment and opening arguments have size $O(\sqrt{N})$ and the arguments have verification time $O(\sqrt{N})$. in Section 8 we present our *self-recursion* technique to achieve succinctness.

6.1 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 [43] 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

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

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

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

The collision resistance and the preimage both directly derive from the Ring-SIS and the Ring-ISIS[3] 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 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.

6.2 Description of Vortex

In this subsection, we expand on the details of Vortex. We will first assume two integers n and m. Vortex allows committing to n vectors $w_i \in \mathbb{F}^m$ in a single commitment and opening them simultaneously for scalar products with a common public vector.

[3] Inhomogeneous SIS

Let $\mathcal{H}$ be a lattice-based hash function (Section 6.1) parametrized to be able to hash vectors of size (at least) m. We also use $\mathcal{L}$ a *systematic* erasure-code with block-size n and codeword-size $n' > n$. We denote its distance by d, its rate by $\rho = n/n'$ and we name its encoding algorithm $\mathsf{encode}_{\mathcal{L}}$. One key difference with Orion [53] and Brakedown [34] here is that we additionally require the encoding function to be systematic. This means that 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" aside of the original block. In practice, we set $\mathcal{L}$ to be the systematic version of the Reed-Solomon code in Lagrange basis. It has $O(n \log n)$ encoding time and benefits from Maximal Distance Separability.

Vortex is a functional commitment scheme that allows efficient simultaneous opening of multiple scalar-product of committed vectors and the same public (and random) vector. The protocol consists in 4 algorithms : Setup, Commit, $\mathsf{ProveBatchOpening}$ and $\mathsf{VerifyBatchOpening}$. An important difference here is that contrary to how the polynomial commitments of Brakedown [34] and Orion [53] are laid out, Vortex does not have *proximity-check* separate from the $\mathsf{BatchOpening}$. This design choice follows from an observation made in [34] (initially proved in [20]) that when the query is random (and we will only need random queries), we can *merge* those two phases. As a result, we obtain a halving in the proof size and verification time. Jumping a bit ahead, we are going to use Vortex alongside (single-point) PIOP to build an AOK, and for this Vortex processes the queries coming from (single-point) PIOP, and these queries are random.

1. Setup is a transparent offline phase run by both the prover and the verifier. During this phase, they perform some precomputations involving sampling the parameters of the Ring-SIS instance and the erasure code that are used to save some time in the other steps, as part of the *public parameters*.
2. The Commit algorithm: Let W, be the matrix whose column i is w_i. Thus, W has m rows and n columns. The prover encodes each row of W (noted $W[j]$) using the encoding function and obtains W' (which has n' columns). Observe that, since the encoding procedure $\mathsf{encode}_{\mathcal{L}}$ is systematic, we have that all columns W are also columns of W'. Without loss of generality, we will assume that W corresponds to $(w'_1 \cdots w'_n)$, the n first columns of W'. The prover then computes the hash of the columns. The collection $H = h_1 \cdots h_{n'}$ forms the *commitment*.
3. The batch-opening phase is an interactive protocol where the prover runs the $\mathsf{ProveBatchOpening}$ algorithm and the verifier runs the $\mathsf{VerifyBatchOpening}$. At the beginning of this phase, the prover holds a set of committed witnesses (and checksums) $w_1, \cdots, w'_n$ and the verifier holds the *final commitment* as input. Both also hold $l \in \mathbb{F}^m$ and $u \in \mathbb{F}^n$, the statement, as input. The prover's goal is to convince the verifier that $\forall i < n, \langle w_i | l \rangle = u_i$. The verifier samples t columns $q_1, \cdots q_t$ ($q_i \leq n'$) uniformly at random, and the prover responds with $(s_1 \cdots s_t) = (w'_{q_1}, \cdots w'_{q_t})$. Then, the verifier computes "the folding" u' the encoding of u and performs the following checks for all

opened columns : (1) the scalar-product $\langle s_i | l \rangle \overset{?}{=} u'_{q_i}$ and (2) the hash of s_i is consistent with h_{q_i}.

Fig. 10 sums up the above,

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

1. Setup an instance of Ring-SIS, 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 $\mathsf{encode}_{\mathcal{L}}$ if any
4. Collect all the computed parameters in pp and return it

Fig. 8. Vortex setup

$\mathsf{Commit}(\mathsf{pp}, W) \rightarrow (h_1 \cdots h_{n'})$

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

Fig. 9. Vortex commitment

$\mathsf{Opening}$ with statement (l, u)

$\mathsf{ProveOpening}(\mathsf{pp}, W', l, u)$		$\mathsf{VerifyOpening}(\mathsf{pp}, H, l, u)$
		$\mathbf{q} \leftarrow\$ [n']^t$
	$\leftarrow \mathbf{q}$	
	$(s_1, \cdots, s_t) \rightarrow$	
		$u' \leftarrow \mathsf{encode}_{\mathcal{L}}(u)$
		for $0 < i \leq t$:
		$\quad \langle s_i | l \rangle \overset{?}{=} u'_{q_i}$
		$\quad \mathsf{Hash}(s_i) \overset{?}{=} h_{q_i}$

Fig. 10. Vortex opening

The security of Vortex (Definition 8) follows similarly to the security proof of Brakedown [34].

7 Vortex Transform: From PIOP to Argument of Knowledge

In this section, we describe the Polynomial Commitment scheme based on Vortex that allows transforming the (single-point univariate) PIOP output by the Arcane compiler into a concrete Argument of Knowledge (AOK) in the random-oracle model.

7.1 Vortex Polynomial Commitment

Here we show how to build a polynomial commitment from Vortex. To commit and open a polynomial P (with degree less that m) via Vortex, it is enough for the public vector l to be of the form $l = (1, x, x^2, \ldots, x^m)$, while the coefficients of P are embedded in a column of W. The prover $\mathcal{P}$ may send a polynomial P whose degree is larger than the number of rows in W. In this case, it can be folded in several chunks $P(X) = P_0(X) + X^m P_1(X) + \cdots$ and inserts each $P_i(X)$ to W, so we shall call the resulting vectors u, u' (from Vortex) “foldings” of polynomials. The verifier can then recombine the evaluations of $P_i(X)$ to obtain the evaluation of $P(X)$. The columns to be committed to in Vortex are obtained by taking the coefficients of each polynomial $P_i(X)$.

8 Self-Recursion of Vortex

We explain how to delegate the verification of the verifier with knowledge soundness. We benefit from the linear structure of our lattice-based hash function to construct a self-recursion protocol for Vortex. This allows us to build a SNARK over any FFT-friendly field.

As Vortex proofs are big (albeit sublinear) : $O(\sqrt{N})$, to get 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 it has oracle access to the folding u' and the hash values). 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 public vector l are computed correctly.
- the encoding $\mathsf{encode}_{\mathcal{L}}(u)$ is correctly computed as u'.
- The opened columns are the right ones.

Concretely, *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 foldings of the Vortex). 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 the Vortex Transform of Section 6 and we can reiterate this process by reusing different Ring-SIS instances and different erasure 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 self-recursion.

SNARK from Argument of Knowledge Considering the AOK presented in Section 7 After multiple steps of self-recursion of 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 recursion 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 [35, 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 the details of the concrete SNARK scheme that we may use and of how we implement the verifier in the circuit. From then on, the present section focuses exclusively on the self-recursion technique.

For the organization of the section: first, in Section 8.1, we present the *Horner protocol*, a commit-and-open protocol specific to our SIS setting, allowing us to commit to a batch of polynomials in constant size. This protocol is mainly used during the Ring-SIS-hash opening. Then in Section 8.2 we provide all the steps and testing protocols needed for the self-recursion.

8.1 Preliminaries for the Self-Recursion: Horner Commitment

In the following sections, we commonly employ a specific technique - *Horner commitments* - for batching the polynomial evaluations that are specific to our SIS setting (Note that in Vortex, we may encounter two types of polynomials: the ones that Vortex may get as input to commit to and the ones that are relevant for SIS). *Horner Commitments* allows one to prove the correct openings without giving the evaluations/openings themselves. This is an important feature since we use this protocol internally (before UniEval queries) and thus can reduce the size of the hash input for the Fiat-Shamir transform, and therefore the proof size [4]. To prove the openings of the SIS polynomials are correct (without knowing the openings), we propose a batching construction based on the Horner approach. Note that a more trivial way is to commit to each polynomial and then open it at the given point α, this requires many individual commitments and openings, which would impose a huge overhead on the communication (particularly, we use the Horner trick over columns from the Vortex that are already long, populated with many SIS polynomials). In the following, the Horner protocol receives m polynomials P_i of degree d and a point α and proves the alleged evaluations of all $P_i(\alpha)$. For a polynomial $P(X)$, we denote its Horner form at point α by $P_\alpha^{\mathsf{Horner}}(X)$(defined in the protocol).

$\mathsf{Horner}(\{P_i\}_{i\in[m]}$,$\alpha)$

- Prover and Oracle set $P(X)$ as the polynomial interpolating to the concatenations of P_i (here P_i is the vector of coefficient of $P_i(X)$).
- Prover and Oracle set $P_\alpha^{\mathsf{Horner}}(X)$ (the Horner-form of $P(X)$) as follows:

$$\forall\, 0 \le i < m, j < d : P_\alpha^{\mathsf{Horner}}(\omega^{id+j}) = \begin{cases} P(\omega^{id+j}) & \text{if } j = d-1 \\ P(\omega^{id+j}) + \alpha P_\alpha^{\mathsf{Horner}}(\omega^{id+j+1}) & \text{else} \end{cases}$$

- the verifier checks that "Global": $P_\alpha^{\mathsf{Horner}}(X) - P(X) - \alpha P_\alpha^{\mathsf{Horner}}(X\omega)(1 - Z_{d-1,md}(X)) = 0$ where $Z(X)_{d-1,md}$ is the domain-selector.

Fig. 11. Honer Commit and Open.

As one can see, the openings $P_i(\alpha)$ are subsampling of $P_\alpha^{\mathsf{Horner}}$ with offset 0, and period $d-1$.

Remark 1. Compared to the approach of committing and opening individually m polynomials, here the verifier only needs to check the global constraint over the domain Ω_{md} (that is, a single UniEval query). This is thanks to the fact that as we use the values $P_i(\alpha)$ internally, we are not forced to output them directly.

8.2 Recursion Steps and Testing Protocols

Now we are ready to present the testing protocols needed for the self-recursion.

[4] This also matters when a different outer-layer SNARK is used

Recursion for public-coin protocol. We describes the subprotocols in a way that each also includes the simulations of queries between the Vortex and the subprotocol. We assume that $(\mathcal{P}, \mathcal{V})$ are the prover and verifier of Vortex. Our goal is to instantiate $(\mathcal{P}', \mathcal{V}')$, prover and verifier of a Wizard-IOP protocol associated with self-recursion from which an extractor can recover a transcript of $(\mathcal{P}, \mathcal{V})$. In our settings, $\mathcal{P}'$ sees the messages from both $\mathcal{P}$ and $\mathcal{V}$, and tries to convince $\mathcal{V}'$ that she saw an accepting transcript.

Note that $(\mathcal{P}, \mathcal{V})$ is a public-coin interactive protocol in which $\mathcal{V}$ generates random challenges. One difficulty of our setting is that we need to convince $\mathcal{V}'$ that the challenges were generated correctly. For that matter, we establish a direct connection between $\mathcal{V}'$ and $\mathcal{V}$: whenever $\mathcal{V}$ wants to generate a random coin, he instead delegates the generation to $\mathcal{V}'$.

8.2.1 Testing systematic Reed-Solomon code-membership In the original Vortex, the verifier $\mathcal{V}$ has plain access to u and computes u' by itself. Doing so enforces two things: u is indeed the original message of u' and that u' is a member of the code (correct Reed-Solomon encoding of a vector). Here, the prover $\mathcal{P}'$ instead sends oracle-access to u'. The verifier $\mathcal{V}'$ can learn u from u' by looking at the n first coordinates, but he still needs to verify that u' is a valid codeword, namely u' is associated with a polynomial of degree n. Therefore, we present an approach for u' being a valid codeword.

$\mathcal{P}'$ forwards to $\mathcal{V}'$ a polynomial $u'(X)$ whose evaluations over roots of unity are the alleged evaluations($\langle l|W'\rangle = u'$).Note that Reed-Solomon codes have efficient probabilistic error detection procedures, we propose a simple Wizard-IOP procedure that allows testing membership of a Reed-Solomon code in which the verifier is only required to evaluate $u'(X)$ at a random point. In the following $u'(X) = \sum u'_i L_i(X) = \sum \bar{u}_i X^i$ for some values $\bar{u}_i$ and $\hat{u}(X) = \sum \bar{u}_i L_i(X)$. Here and in the rest of this section, by scalar-product of two polynomials, we are implicitly referring to the scalar-product of the vectors corresponding with their evaluations. This is a reasonable convention by the description of scalar-product protocol in Fig. 4.

1. $\mathcal{P}'$ computes $\hat{u}(X)$, the polynomial that interpolates the coefficients of $u'(X)$. Allegedly, the $n' - n$ last evaluations of $\hat{u}(X)$ are zeros, since, equivalently, u' is alleged to be part of the code $\mathcal{L}$ (and therefore should be of degree n). $\mathcal{P}'$ sends oracle-access to $u'(X)$ and $\hat{u}(X)$ to $\mathcal{V}'$.
2. $\mathcal{V}'$ queries, "Scalar-Product": $\langle \hat{u}(X)|\hat{Z}_u(X)\rangle = 0$, where $\hat{Z}_u(X)$ is a polynomial that vanishes on its n first entries and is 1 between n and n'.The verifier $\mathcal{V}'$ then generates a random coin r_u.
3. $\mathcal{P}'$ sends oracle-access to a polynomial $P_u(X)$ of degree n' such that $\forall k < n' : P_u(\omega^k) = r_u^k$
4. $\mathcal{V}'$ respond with the following queries: "Scalar-product" : $\langle P_u(X)|\hat{u}(X)\rangle = y_u$ and "Evaluation query" : $u'(r) = y'_u$. $\mathcal{V}'$ then checks that $y_u \stackrel{?}{=} y'_u$
5. finally $\mathcal{V}'$ checkes the well-forming of $P(X)$ via $P_u(\omega X) = r_u P_u(X)$ and $P_u(1) = 1$.

The objective of (2) is to prove that $\hat{u}$ encodes a polynomial whose last $n' - n$ coefficients are 0 and the objective of (4) is to prove that $\hat{u}$ and u' are the same polynomials encoded differently.

8.2.2 Evaluation of long polynomials with oracle-access to the folding

As mentioned in Section 7.1, a long[5] polynomial $P_i(X)$ can be represented via several columns included in W. Thus, assume that each of n_p columns of W (starting from the first column) is associated with a polynomial P_i. Then we have $P_i(x) = \sum_{j=0}^{n_p} u'_j x^{mj} = y_i$ where m is the number of rows in W and u' is the folding from Vortex.

Unlike the original Vortex that the verifier has plain-access to the folding u' and can build y_i, here $\mathcal{V}'$ has plain access only to y_i and the prover $\mathcal{P}'$ should convince the verifier that these values are correctly computed from the vector u'. The protocol between $\mathcal{P}'$ and $\mathcal{V}'$ is described in Fig. 12.

$\mathsf{OracleEval}(\{u'_j, y_i\}_{j\in[n'],i\in[m]},x)$

Prover's input: $\{u'_j, y_i\}_{j\in[n'],i\in[m]},x$
Verifier's input: $\{y_i\}_{j\in[n']},x$

- Prover and Oracle set $U(X)$ as the polynomial interpolating to u'_i.
- Verifier sends the randomness $\beta \leftarrow \mathbb{Z}_q$.
- Prover and Oracle set $U_\beta(X)$ as follows:

$$: U_\beta(\omega^{in_p+j}) = \begin{cases} \beta^i \cdot x^{mj} & \forall i < m, j < n_p \\ 0 & \text{if } \; n_p i + k > mn_p \end{cases}$$

- Verifier:
  - Checks that U_β is well-formed:
    "Global": $U_\beta(\omega X) - U_\beta(X)x^m + (U_\beta(X) - I_\beta(X))Z_{n_p,mn_p}$ where $I_\beta(X) = \sum \beta^i \mathcal{L}_{\omega^{in_p}}(X)$.
  - sends a query "scalar-product" : $\langle U(X)|U_\beta(X)\rangle = \sum \beta^i y_i$.

Fig. 12. Evaluation with oracle-access to the folding.

8.2.3 Checking the Ring-SIS hashes evaluations

Here we aim for proving that (1) Ring-SIS hashing of the chosen columns are correct and (2) the bound for Ring-SIS hash is respected. To prove (1), we first move from claims over $\mathcal{R}$ to the claims over $\mathbb{F}[X]$, and then from the claims over $\mathbb{F}[X]$ to the claims over $\mathbb{F}$ via our Horner Protocol. For (2), we use a simple range check.

First, let us clarify the setting. $\mathcal{V}'$ has oracle access to $H(X)$ the polynomial encoding commitment and plain access to $\mathbf{q}$ the random choice of columns to be opened.

[5] We use the word "long" rather than "high degree", since we see polynomials as vectors inside Vortex

And the hash computation over the chosen columns $S_\bullet$ is as

$$\sum_{i<m} A_i(X) S_{\bullet,i}(X) = h(X) \mod \mathcal{R}$$

.

For convenience, we reuse the notation of Section 6.1, $L_\bullet = (L_{\bullet,j})_{j<m_s}$, to denote the embedding in $\mathcal{R} = \frac{\mathbb{F}[X]}{X^d+1}$ of the limb expansion of a column $\bullet$ and $h(X)$ the hash encoded as a ring element of $\mathcal{R}$. In the current phase, for each opened columns $\bullet$, $\mathcal{V}'$ will be provided oracle-access to a polynomial $S_\bullet(X)$ encoding the coefficients of $L_{\bullet,j}(X) = \sum_{k<d} L_{\bullet,j,k} X^k$ as follows, $S_\bullet(\omega^{jd+k}) = L_{\bullet,j,k}$ i.e., $S_\bullet(X) = \sum_j L_{\bullet,j,k} \mathcal{L}_{jd+k}(X)$.

The prover should also provide oracle-access to the polynomial $A(X)$ that is obtained by interpolating the vector A, where A is the concatenation of the coefficients of the polynomials $A_i(X) = \sum_j a_{i,j} X^j$ constituting the hashing key of Ring-SIShash defined in Section 6.1. Thus, $\forall i < m_{\mathsf{sis}}, j < d, A(\omega^{id+j}) = a_{i,j}$. Here, m_{sis} denotes the number of polynomials in the Ring-SIShash, and d is the degree of the Ring-SIS modulus polynomials.
Particularly, $\mathcal{P}'$ will attempt to convince $\mathcal{V}'$ that $S_i(X)$, associated with chosen columns, encodes values within the range specified by the instance of Ring-SIS hash function. In addition to this, $\mathcal{V}'$ also needs to be convinced that the values encoded in each $S_i(X)$ are consistent with the hashes encoded in the polynomial $H(X)$ (again for the chosen columns), that would be followed from the fact that,

$$\sum_{i<m} A_i(X) S_{\bullet,i}(X) = h(X) + (X^d + 1) h_{\mathsf{leftover}}(X) \tag{1}$$

for some polynomial $h_{\mathsf{leftover}}(X)$. This allows moving from ring $\mathcal{R}$ to $\mathbb{F}[X]$. For convenience, we refer to $h_{\mathsf{leftover}}(X)$ as the 'leftover of the hash'.

The general idea here is to evaluate all the polynomials involved in Eq. (1) at point α, namely, moving from $\mathbb{F}[X]$ to $\mathbb{F}$. This converts the discussion over polynomials to discussion over field elements, which is easy to deal with. For this, we use the Horner protocol presented in Section 8.1.

1. $\mathcal{P}'$ sends,
   - oracle access to $H(X)$, $A(X)$, and to $S_{i_k}(X)$ for $i_k \in \mathbf{q}$ i.e., on the chosen columns.
   - oracle access to a polynomial $H_{\mathsf{leftover}}(X)$ of degree $d|\mathbf{q}| = dt$ encoding the coefficients of the leftovers h_{leftover} of the selected column hashes.
2. the verifier sends a query to check the coefficient of $S_{i_k}(X)$ for $i_k \in \mathbf{q}$ are within range as required by the Ring-SIS instance.
3. the verifier $\mathcal{V}'$ chooses a random value α, the prover and the verifier run the Horner protocol for $H(X)$ (defined before) where the Horner form is $H_\alpha^{\mathsf{Horner}}(X)$. The verifier defines $H'_\alpha(X)$ as the subsampling of $H_\alpha^{\mathsf{Horner}}(X)$ with period d and offset 0. This provides access to the values $h(\alpha)$ for the chosen columns (consistent with $H(X)$). They do the same for $H_{\mathsf{leftover}}(X)$, and for $A(x)$. Then define $A'_\alpha(X)$ and $H'_{\mathsf{leftover},\alpha}(X)$ as the subsampling of $A_\alpha^{\mathsf{Horner}}(X)$ and $H^{\mathsf{Horner}}_{\mathsf{leftover},\alpha}(X)$ (res.) with respect to the period d and offset 0.

4. They also need to run the Horner protocol on S_{i_k} for $i_k \in \mathbf{q}$. They can do this with one more layer of batching via random combination. More precisely, the verifier sends a randomness β and they run the Horner protocol on $S(X) = \sum \beta^k S_{i_k}(X)$. Again, let $S'_\alpha(X)$ be defined as the subsampling of $S^{\mathsf{Horner}}_\alpha(X)$. The prover also provides oracle access to $R_\beta(X)$ the polynomial encoding the successive powers of β.
5. $\mathcal{V}'$ sends the following queries to check the correct computation of Ring-SISHash.
   - "Scalar-product" : $y_s \leftarrow \langle H'_\alpha(X)|R_\beta(X)\rangle$ (over t-roots of unity only).
   - "Scalar-product" : $y_t \leftarrow \langle H'_{\mathsf{leftover},\alpha}(X)|R_\beta(X)\rangle$ (over the t-roots of unity only)
   - "Scalar-product" : $y_h \leftarrow \langle A'_\alpha(X)|S'_\alpha(X)\rangle$ (over the m_{sis} roots of unity only)
   - And finally check that $y_h \overset{?}{=} y_s + (\alpha^d + 1)y_h$
   - "Global" : $(X-1)(R_\beta(\omega X) - \beta R_\beta(X)) = 0$ (for well-forming of $R_\beta(X)$)

Here we use the trick from the section *Section* 5.8, to convert these global constraints over subsamplings to the global constraint over the original (Horner-form) polynomials (note that the verifier already has oracle-access to the Horner-form polynomials, by the description of Horner protocol).

8.2.4 Testing the scalar-products Let $l = (1, x, x^2, \cdots)$ be the vector from Vortex statement. The prover (of Vortex) $\mathcal{P}$ will output $s_1, \cdots, s_n$ and $\mathcal{P}'$ needs to convince $\mathcal{V}'$ that the scalar products $\langle s_i|l\rangle = \bar{u}$ hold for all $i \in [t]$. In our setting, $\mathcal{V}'$ has oracle-access to $\bar{u}(X)$(the polynomial encoding u' at the chosen columns) and plain access to l. The challenge is that he does not have access to s_i directly. Instead, $\mathcal{P}'$ sends oracle access to polynomials encoding the limb decomposition of the s_i (i.e., $S_i(X)$ from Section 8.2.3). Therefore, we need a workaround to allow the verifier $\mathcal{V}'$ to test the correctness of the scalar-product. The prover $\mathcal{P}'$ and the verifier $\mathcal{V}'$ proceed as follows:

1. The prover provides the oracle access to $u'(X)$, and $\bar{u}(X)$ encoding the entries of the folding u' on the chosen columns.
2. $\mathcal{P}'$ sees each element $s_{\bullet,j}$ belonging to $\mathbb{Z}_q$ and represent it in base b (where b is the bound for Ring-SIShash). We denote by $S_\bullet(X)$, the polynomial encoding the concatenation base-b representation of $s_{\bullet,j}$.
3. $\mathcal{P}'$ constructs a polynomial $l_b(X)$. Let $n_b = \log q/\log b$ be the number of limbs per field element needed by the lattice-based hash function described in Section 6.1 and let m_r be the depth of a Vortex column. $l_b(X)$ is constructed as the unique polynomial satisfying,
$$\forall i < m_r, j < n_b,\ l_b(\omega^{in_b+j}) = l_i b^j$$
Note that by this representation we expect that
$$\langle S_i(X)|l_b(X)\rangle = \sum_j x^j(\text{base-b representation of } s_{i,j}) = \sum_j x^j s_{i,j}$$
.

4. $\mathcal{V}'$ sends the following queries:
   - "Global" : $(l_b(X) - l(X))Z_b(X) + (l_b(\omega X) - bl_b(X))(n_b - Z_b(X)) = 0$ (to check the well-forming of $l_b(X)$).
   - "Scalar-product" (for all $i_k \in \mathbf{q}$) $\langle S_{i_k}(X) | l_b(X) \rangle \overset{?}{=} \bar{u}_k$
   - "Inclusion" : $(q(X), \bar{u}(X)) \subset (I_{n'}(X), u'(X))$ where $q(X)$ is encoding of $\mathbf{q}$, and $I_{n'}(\omega^i) = i$ for $i < n'$ that is committed in an offline phase.

8.2.5 Complexity analysis. We begin by reminding the readers of the parameters in play. Let n_{poly} be an integer denoting the number of polynomials the prover wishes to commit to and let $d_1, \cdots, d_{n_{\mathsf{poly}}}$ be their respective domain size. When the prover commits to these polynomials, he packs them in a matrix of m_r rows and n columns and then the matrix is row-encoded into a matrix of $n' > n$ columns using a systematic Reed-Solomon code. We say that $\rho = \frac{n'}{n}$ is the expansion factor of the code. The verifier is sent the hash of those columns obtained using the ring-SIS hash function of Section 6.1. In its internal work, the hash function splits all field elements into n_b limbs shorter than b (where b is a parameter of the ring-SIS instance in use) and outputs a hash consisting of d field elements (also a parameter of the ring-SIS instance). The verifier asks the prover to open t columns.

We assume $(\mathcal{P}, \mathcal{V})$ to be original prover and verifier of a protocol compiled using Vortex. Let $N = \sum_{i \in n_{\mathsf{poly}}} d_i = nm_r$ denote the size of the matrix $\mathcal{P}$ wishes to commit to and open. We denote the runtime complexity of $\mathcal{P}$ by $\mathbf{P}^{(0)}(N)$, the runtime complexity of $\mathcal{V}$ by $\mathbf{V}^{(0)}(N)$ and the communication complexity by $\mathbf{C}^{(0)}(N)$. Moreover, we have that

- $\mathbf{P}^{(0)}(N) = \Omega(N)$
- $\mathbf{V}^{(0)}(N) = v(N) + O(\sqrt{N})$ with $v(N) = o(\sqrt{N})$
- $\mathbf{C}^{(0)}(N) = c(N) + O(\sqrt{N})$ with $c(N) = o(\sqrt{N})$

After one step of recursion, the resulting new prover $\mathcal{P}^{(1)}$ is tasked to commit and open a matrix whose size is proportional to a Vortex proof: $N' = O(\sqrt{N})$ if the parameters are suitably chosen. This incurs an overhead of $O(\sqrt{N}\log N)$.On its end, the verifier $\mathcal{V}^{(1)}$ ought to perform a constant number of additional checks, but verify the opening of a smaller matrix. This translates into an increase of the verification time and the communication complexity by small constant overheads v_1 and respectively c_1 but also a smaller dominating term $O(n^{\frac{1}{4}})$ instead of $O(\sqrt{N})$. We denote the runtime complexity of $\mathcal{P}^{(1)}$ by $\mathbf{P}^{(1)}(N)$, the runtime complexity of $\mathcal{V}^{(1)}$ by $\mathbf{V}^{(1)}(N)$ and the updated communication complexity by $\mathbf{C}^{(1)}(N)$. To sum up the above, after one step of recursion the updated complexity are the following:

- $\mathbf{P}^{(1)}(N) = \Omega(N) + o(N)$
- $\mathbf{V}^{(1)}(N) = v(N) + v_i + O(N^{\frac{1}{4}})$

- $\mathbf{C}^{(1)}(N) = c(N) + c_i + O(N^{\frac{1}{4}})$

And after, $f = \log\log N$ steps the term $O(N^{\frac{1}{2^f+1}})$ is reduced to a constant and we have:

- $\mathbf{P}^{(f)}(N) = \Omega(N)$
- $\mathbf{V}^{(f)}(N) = v(N) + \max v_i \log\log N$
- $\mathbf{C}^{(f)}(N) = c(N) + \max c_i \log\log N$

Whether and how these asymptotic bounds can be improved by carefully choosing different lattice instances and erasure codes remains an open question. As general guidance, the first instance of Vortex should use a large-degree-large-bound SIS instance and an erasure code with a small expansion factor. However, these choices of parameters tend to increase N' and we need to preserve $N' = o(N)$. The recursion steps coming thereafter should optimize for the verification time and communication complexity.

Note that with each recursion we need a key for the hash function and the key of recursion i is committed by the key of recursion $i+1$. To guarantee the use of the proper keys, the prover and the verifier compute the recursive commitments to the keys in an offline phase (i.e., preprocessing). As a result, the verifier key has size $o(\log N)$ and the proving key complexity has size $O(N)$.

Acknowledgement

We would like to thank Zhenfei Zhang for pointing out an issue with some of the parameter sets of SIS problem in the initial version of this paper. We are also grateful to Nicolas Liochon and Olivier Bégassat 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. “Lattice-Based SNARKs: Publicly Verifiable, Preprocessing, and Recursively Composable - (Extended Abstract)”. In: *Advances in Cryptology - CRYPTO 2022.* Vol. 13508. LNCS. Springer, 2022, pp. 102–132. DOI: 10.1007/978-3-031-15979-4_4. URL: https://doi.org/10.1007/978-3-031-15979-4%5C_4.

[5] 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.

[6] Shi Bai et al. "Improved Combinatorial Algorithms for the Inhomogeneous Short Integer Solution Problem". In: *J. Cryptol.* (2019).

[7] Shi Bai et al. "Improved Combinatorial Algorithms for the Inhomogeneous Short Integer Solution Problem". In: *J. Cryptol.* (2019).

[8] David Balbás et al. "Functional Commitments for Circuits from Falsifiable Assumptions". In: *IACR Cryptol. ePrint Arch.* (2022), p. 1365. URL: `https://eprint.iacr.org/2022/1365`.

[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] Carsten Baum et al. "Sub-linear Lattice-Based Zero-Knowledge Arguments for Arithmetic Circuits". In: *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part II.* Ed. by Hovav Shacham and Alexandra Boldyreva. Vol. 10992. Lecture Notes in Computer Science. Springer, 2018, pp. 669–699. DOI: `10.1007/978-3-319-96881-0\_23`. URL: `https://doi.org/10.1007/978-3-319-96881-0%5C_23`.

[11] Stephanie Bayer and Jens Groth. "Efficient Zero-Knowledge Argument for Correctness of a Shuffle". In: *EUROCRYPT.* 2012.

[12] Olivier Bégassat et al. *A ZK-EVM specification.* 2022.

[13] 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`.

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

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

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

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

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

[19] 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.

[20] Jonathan Bootle, Alessandro Chiesa, and Jens Groth. "Linear-Time Arguments with Sublinear Verification from Tensor Codes". In: *Theory of Cryptography - 18th International Conference, TCC 2020, Durham, NC, USA,*

November 16-19, 2020, Proceedings, Part II. Vol. 12551. Lecture Notes in Computer Science. Springer, 2020, pp. 19–46. DOI: 10.1007/978-3-030-64378-2_2. URL: https://doi.org/10.1007/978-3-030-64378-2%5C_2.

[21] Jonathan Bootle et al. "A Non-PCP Approach to Succinct Quantum-Safe Zero-Knowledge". In: *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part II.* Ed. by Daniele Micciancio and Thomas Ristenpart. Vol. 12171. Lecture Notes in Computer Science. Springer, 2020, pp. 441–469. DOI: 10.1007/978-3-030-56880-1_16. URL: https://doi.org/10.1007/978-3-030-56880-1%5C_16.

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

[23] Benedikt Bünz et al. "Bulletproofs: Short Proofs for Confidential Transactions and More". In: *2018 IEEE Symposium on Security and Privacy, SP 2018.* IEEE Computer Society, 2018, pp. 315–334. DOI: 10.1109/SP.2018.00020. URL: https://doi.org/10.1109/SP.2018.00020.

[24] Matteo Campanelli, Dario Fiore, and Anaïs Querol. "LegoSNARK: Modular Design and Composition of Succinct Zero-Knowledge Proofs". In: *ACM SIGSAC.* Nov. 2019, pp. 2075–2092.

[25] Leo de Castro and Chris Peikert. "Functional Commitments for All Functions, with Transparent Setup". In: *IACR Cryptol. ePrint Arch.* (2022), p. 1368. URL: https://eprint.iacr.org/2022/1368.

[26] Binyi Chen et al. "HyperPlonk: Plonk with Linear-Time Prover and High-Degree Custom Gates". In: *IACR Cryptol. ePrint Arch.* 2022 (2022), p. 1355.

[27] 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.

[28] 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.

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

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

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

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

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

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

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

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

[37] Daira Hopwood et al. *Zcash protocol specification: Version 2022.04.26 Technical report, Zerocoin Electric Coin Company.* `https://github.com/zcash/zips/blob/main/protocol/protocol.pdf`. 2022.

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

[39] Aniket Kate, Gregory Zaverucha, and Ian Goldberg. "Constant-Size Commitments to Polynomials and Their Applications". In: *Advances in Cryptology - ASIACRYPT*. Vol. 6477. LNCS. Springer, 2010, pp. 177–194.

[40] 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.

[41] Jianwei Li and Phong Q. Nguyen. "A Complete Analysis of the BKZ Lattice Reduction Algorithm". In: *IACR Cryptol. ePrint Arch.* 2020 (2020), p. 1237.

[42] Benoît Libert, Somindu C. Ramanna, and Moti Yung. *Functional Commitment Schemes: From Polynomial Commitments to Pairing-Based Accumulators from Simple Assumptions.* 2016.

[43] 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`.

[44] 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.

[45] Daniele Micciancio and Oded Regev. *Class on lattice-based cryptography.* 2008.

[46] Srinath Setty. *Spartan: Efficient and general-purpose zkSNARKs without trusted setup.* CRYPTO. 2020.

[47] 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.

[48] Polygon Hermez Team. *Scalable payments. Decentralised by design, open for everyone.* `https://hermez.io`.

[49] Polygon Zero Team. *PLONKY2 : Fast Recursive Argument with Plonk and FRI*. `https://github.com/mir-protocol/plonky2/blob/main/plonky2/plonky2.pdf`. Draft : version 2022. 2022.

[50] Riad Wahby et al. "Doubly-Efficient zkSNARKs Without Trusted Setup". In: *SP*. IEEE Computer Society, 2018, pp. 926–943.

[51] Hoeteck Wee and David J. Wu. "Succinct Vector, Polynomial, and Functional Commitments from Lattices". In: *IACR Cryptol. ePrint Arch.* (2022). URL: `https://eprint.iacr.org/2022/1515`.

[52] 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.

[53] Tiancheng Xie, Yupeng Zhang, and Dawn Xiaodong Song. "Orion: Zero Knowledge Proof with Linear Prover Time". In: *IACR Cryptol. ePrint Arch.* 2022.

[54] Risc Zero. `https://github.com/risc0/risc0`. 2022.

[55] 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 6.1, 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 [43]. 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 $Ax = t$*

Foremost, from a few observations on SIS. One can see, that

- SIS (and ISIS) cannot become harder by increasing m. That's because an attacker can always restrict the search space to $m' < m$ by arbitrarily forcing some entries of x to zero.

- It can only become 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's because it's equivalent to restricting the search space.
- $b >= q$ makes the problem trivial, as it can be solved by Gaussian elimination in polynomial-time.

Remark 2. The work of [29] 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).

From then on, 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 $b < q$. 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 "$m_{SIS} = nm_{RSIS}$". Working with ring-SIS has several practical benefits compared to SIS: the space taken to represent A is n time 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 6.1 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 of resistance coincide with the I-SIS 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 per 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 exactly to finding a short-vector in the kernel lattice.

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

The first thing, one should have in mind is that we are always free to pick $m_0 < m'$ if it pleases us to do so. 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,

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

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. So, 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 [47], 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.

$$\log t_{\mathsf{oracle}} = 0.292k + 16.4 \tag{2}$$

In a recent work, [41] 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 [41], 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.

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

$$\log \delta = \frac{1}{2m_0(k-1)}(m_0 - 1 + \frac{k(k-2)}{m_0}) \log \gamma_k \tag{3}$$

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 [41] 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. (4):

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

One should note that Eq. (4) 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 is a nonsense) 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 [45] describes the basic version of these attacks and gives an easy procedure to determine their efficiency. The attack is also known as CPW. In [6], 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 they used careful manual-tuning of its parameters.

As in [6] 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 $\mathbf{n} \leq \mathbf{50}$, 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 $\mathbf{n} > \mathbf{50}$, then we apply the simplified analysis given in [45]. From [45], the cost This will give us 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 tuning 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 [45]. 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, [38]. 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 [7], 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 $\psi : s(X) \rightarrow Xs(X)$. 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 [7] 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. [7] 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 as memory intensive as they, typically as much 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