Jolt-b: recursion friendly Jolt with basefold commitment

Hang Su [*] Qi Yang [†] Zhenfei Zhang [‡]

July 11, 2024

Abstract

The authors of Jolt [AST24] pioneered a unique method for creating zero-knowledge virtual machines, known as the lookup singularity. This technique extensively uses lookup tables to create virtual machine circuits. Despite Jolt's performance being twice as efficient as the previous state-of-the-art[1], there is potential for further enhancement.

The initial release of Jolt uses Spartan [Set20] and Hyrax [WTs+18] as their backend, leading to two constraints. First, Hyrax employs Pedersen commitment to build inner product arguments, which requires elliptic curve operations. Second, the verification of a Hyrax commitment takes square root time $O(\sqrt{N})$ relative to the circuit size N. This makes the recursive verification of a Jolt proof impractical, as the verification circuit would need to execute all the Hyrax verification logic in-circuit. A later version of Jolt includes Zeromorph [KT23] and HyperKZG as their commitment backend, making the system recursion-friendly, as now the recursive verifier only needs to perform $O(\log(N))$ operations, but at the expense of a need for a trusted setup.

Our scheme, Jolt-b, addresses these issues by transitioning to the extension field of the Goldilocks and using the Basefold commitment scheme [ZCF23], which has an $O(\log^2 N)$ verifier time. This scheme mirrors the modifications of Plonky2 over the original Plonk [GWC19]: it transitions from EC fields to the Goldilocks field; it replaces the EC-based commitment scheme with an encoding-based commitment scheme.

We implemented Jolt-b, along with an optimized version of the Basefold scheme. Our benchmarks show that at a cost of 2.47× slowdown for the prover, we achieve recursion friendliness for the original Jolt. In comparison with other recursion-friendly Jolt variants, our scheme is 1.24× and 1.52× faster in prover time than the Zeromorph and HyperKZG variants of Jolt, respectively.

1 Introduction

Zero-knowledge virtual machines are one of the most novel technologies we have seen in the blockchain space in recent years. At a high level, it is verifiable computation at a super scale. It allows a prover to prove a potentially large amount of computation, with minimal computation or communication requirement from the verifier. A typical example is the so-called zero-knowledge Ethereum Virtual Machines (zk-EVMs) where a prover can generate a succinct proof, attesting the correct execution of a large set of onchain transactions. Such a proof is so small that it can be verified on chain via a smart contract. The verification is also automatic and does not involve any interaction. This technology has been utilized in numerous projects, including Polygon, zkSync, and Scroll.

Designing efficient zkVM protocols is still an on-going problem. Until last year, it was considered almost impractical, with the only plausible candidate being RISC-0, which builds a zkVM for the RISC-V instruction set via the plonk proving system [GWC19]. The past year has seen many breakthrough works which shed light on practical zkVMs. The SP1 project combines RISC-0's toolchain with the Valida VM and Plonky3

[*]Cysic, Inc. Email: hangsu.crypto@gmail.com
[†]Cysic, Inc. Email: qiyang649@gmail.com
[‡]Ethereum Foundation. Email: zhenfei.zhang@ethereum.org

[1]https://a16zcrypto.com/posts/article/building-jolt/

prover, achieving a 3x improvement on top of RISC-0. Jolt [AST24], the topic of this paper, took a completely different route, which they called lookup singularity, and achieved a 2x improvement over SP1. On yet another orthogonal direction, ceno [LZZ+24] proposed a new method for zkVM design via vertical segmentation based on basic blocks, and their memory consistency checks are even less than the number of memory accesses in the original program.

Taking a closer look at the Jolt instantiation [AST24], it uses the Spartan prover [Set20], which is instantiated by an Interactive Oracle Proof (IOP) for R1CS, and a Hyrax polynomial commitment scheme (PCS) [WTs+18] for multilinear polynomials. Underneath the hood, Hyrax operates over cyclic groups, practically instantiated with elliptic curves. It builds on top of the Pedersen commitment [Ped92]. It runs in linear time for the prover, and square root time for the verifier, in terms of the size of the circuit.

1.1 Our contribution

This paper explores `Jolt` utilizing an alternative commitment scheme, referred to as Basefold. We name our variant `Jolt-b`. Our work is based on a simple observation that Hyrax is recursion unfriendly; However, by employing a variant of the Basefold, we can transition to smaller fields. This not only results in a more efficient IOP but also enhances recursion-friendliness.

We do not claim any academic credit beyond this observation.

Looking back, we've observed that `Plonky2` [Pol21] enhances plonk [GWC19] in a similar manner. The remainder of our contribution is rooted in the engineering effort required to assemble all the components and validate the aforementioned observation. Our contributions are as follows:

- We implemented Goldilocks and its extension fields operations using the Halo2 trait.
- We developed a new Poseidon hash function instantiation, which hashes 8 Goldilocks base field elements into 4. We've named it the *Octopos* hash. It's worth noting that a similar design has been employed in `Plonky2` [Pol21] and `Plonky3` [Pol23].
- We built a new Basefold library from the ground up. We introduced a few modifications compared to the reference Basefold implementation [Had24].
  - Primarily, we utilized a Goldilocks field for the Basefold commitment scheme. This necessitated a new hash function with a different fan-in, which we implemented as the Octopos hash function.
  - Secondly, we employed Reed-Solomon code as the underlying field is now FFT-friendly. Compared to the reference implementation, though the asymptotic complexity remains the same, the concrete performance improves significantly.
  - Lastly, we implemented a batch opening method, ensuring that the amortized cost of the Basefold opening and verification remains almost constant regardless of the number of openings. This final step is vital as without it, the verifier would need to verify a linear number of commitments, making it impossible to construct an efficient recursive verifier.

Recursion friendliness Recursion friendliness is a crucial attribute for zkVMs. Often, the zkVM must prove a circuit of immense size, which is impractical to prove in one go. In these instances, the zkVM proves the circuit recursively. It breaks the program into subprograms, proves each separately, and then uses a recursive proof to confirm the correctness of all subprogram proofs.

Recursion also finds application in on-chain verification. Frequently, the verifier is a smart contract on a blockchain with a gas limit. The proof must be verified within this limit. A recursive proof significantly reduces the circuit size and, consequently, the gas cost of verification.

Generally, a zkVM requires two properties to be recursion friendly: First, the proof system's verification algorithm must be succinct, i.e., it should run in sublinear time relative to the circuit size. Second, it should

	Jolt [AST24]			This work
	Hyrax	Zeromorph	HyperKZG	
No trusted setup	●	○	○	●
Recursion friendly	○	●	●	●

Table 1: Comparison of this work to the instantiations of Jolt from various PCS in recursive friendliness and whether it has a transparent setup.

use as few non-native field arithmetic operations as possible, as these are generally 30 to 50$\times$ more expensive than native field arithmetic operations, when expressed in circuits.

As previously mentioned, our solution is more recursion-friendly than the original Jolt, as its verification algorithm runs in $O(\log^2 N)$, compared to $O(\sqrt{N})$ in the original Jolt. We are arguably more friendly than the Zeromorph and HyperKZG variants of Jolt, as we only have a single field and do not require non-native field arithmetics. In contrast, Zeromorph and HyperKZG requires pairing friendly curves, and non-native field arithmetics are essential even when cyclic curves are used.

It's worth noting that if one were to build an on-chain verifier, our scheme would still require non-native field arithmetics to convert from the Goldilocks field to the scalar field of the BN254 curve.

Software stack Our software stack builds as follows:

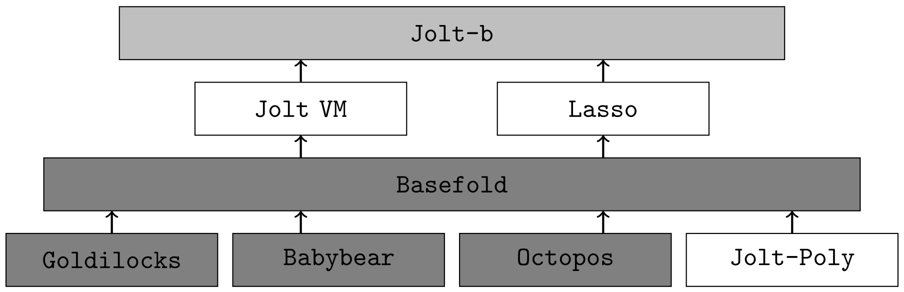

Figure 1: Stack of libraries comprising Jolt-b. The components in gray we implemented ourselves. The components in lightgray we built on top of existing solutions.

Performance We highlight the performance of our solution. A more detailed breakdown of the costs are presented in Section 4. At a high level, our scheme is 2.47$\times$ slower than the original Jolt in the prover time. Our scheme is 1.24$\times$ and 1.52$\times$ faster in prover time than Jolt from Zeromorph and HyperKZG, respectively. Specifically, due to the use of smaller fields, our IOP component is over 2.8$\times$ faster than the ones use in Jolt with all three variants.

1.2 Future work

We have identified several avenues for further improvement.

- Adaptive lookup singularity. Jolt-b reuses most of the circuit instantiations from Jolt. The majority of the circuits in Jolt-b are inherently written with lookups in a passive manner: when it can be expressed in lookups, it is. It has been debated whether such an approach leads to the best performance; or whether one can opportunistically use lookups when a lookup-based circuit is indeed more efficient than a set of native R1CS gates.

- Improved concrete query complexity. In STIR [ACFY24], Arnon et al. applied a new technique to recursively improve the rate of the tested Reed-Solomon code. This leads to concretely smaller query complexity, and concretely smaller proof size. We believe applying such technique in Basefold proximity test could bring similar improvement in proof size, and decrease the recursive verifier circuit size.
- Improved proximity gaps for interleaved linear codes. Both `Jolt` and `Jolt-b` extensively use batch proving for multilinear polynomials. For batch opening in Basefold, the prover runs the proximity test over an interleaved codeword, namely interleaved proximity test, has been studied by previous works [BCI+20, AHIV23, DP24], Improvement in proximity gaps for affine space increases soundness in each query of the interleaved proximity test, which reduces the recursive verifier circuit size eventually.
- Integration with Ceno. In [LZZ+24], the authors proposed a new paradigm to design zkVMs. They segment the VM's opcode list into basic blocks. They demonstrate that a GKR prover can take advantage of duplicated basic blocks; and the memory checks within each basic block are free for their design. [LZZ+24] is orthogonal to `Jolt-b` in that we can utilize `Jolt-b`'s lookup singularity for opcode circuits within each basic block. Such a system seems to be the superior of the two schemes.

On tower of power-of-2 fields Another direction for improving Jolt, as pointed out by [Dra24], is to use tower of power-of-2 fields, known as Binius [AMPS24]. While Binius appears promising on paper, we believe

- Binius is not the right approach to build a zkVM;
- Binius can be very beneficial for building pre-compiles alongside `Jolt-b`.

The entire value proposition of Binius is that one can use the smallest power-of-two field for the witnesses, and therefore only pay as much as one needs. A typical example is that one can now implement hash functions such as Sha2 or Keccak over a binary field. The catch is that to implement operations over u16 or u32 structures, one will need to either move up the tower or use lookups. This requires careful consideration for the interpreter to decide which field to work on; and also adds additional complexity to proof recursion. In the end, a Keccak circuit, written with Binius, still requires at least two orders of magnitude more gates, compared with a Poseidon circuit.

In summary, we believe Binius is most powerful when a binary field is essential. This makes it a perfect candidate for the Keccak or Sha2 precompile for `Jolt-b`.

2 Preliminaries

We write λ (oftentimes implicitly) to denote the security parameter. For a positive integer $n \in \mathbb{N}$, we write $[n]$ to denote the set $\{1, \ldots, n\}$. We write $\{x_i\}_{i \in [n]}$ to denote the ordered multi-set of values $x_1, \ldots, x_n$. We will typically use bold lowercase letters (e.g., $\mathbf{v}, \mathbf{w}$) to denote vectors and bold uppercase letters (e.g., $\mathbf{A}, \mathbf{B}$) to denote matrices. For a vector $\mathbf{v} \in \mathbb{Z}_p^n$, we will use non-boldface letters to refer to its components; namely, we write $\mathbf{v} = (v_1, \ldots, v_n)$. For a finite set S, we write $x \xleftarrow{\text{R}} S$ to denote that x is sampled uniformly from S. For a distribution $\mathcal{D}$, we write $x \leftarrow \mathcal{D}$ to denote that x is sampled from $\mathcal{D}$.

We say that a function f is negligible in λ if $f(\lambda) = o(1/\lambda^c)$ for all $c \in \mathbb{N}$; we denote this $f(\lambda) = \mathsf{negl}(\lambda)$. We write $\mathsf{poly}(\lambda)$ to denote a function bounded by a fixed polynomial in λ. We say an event happens with negligible probability if the probability that the event occurs is negligible, and that it happens with overwhelming probability if its complement occurs with negligible probability.

We also recall the Schwartz-Zippel lemma [Sch80, Zip79]:

Lemma 2.1 (Schwartz-Zippel [Sch80, Zip79]). *Let $f \in \mathbb{F}[x_1, \ldots, x_n]$ be a multivariate polynomial of total degree at most d over $\mathbb{F}$, not identically zero. Then for any set $S \subseteq \mathbb{F}$,*

$$\Pr\left[f(\alpha_1, \ldots, \alpha_n) = 0 \mid \alpha_1, \ldots, \alpha_n \xleftarrow{\text{R}} S\right] \leq \frac{d}{|S|}.$$

2.1 Coding Notations

We generally adopt the notation from Ligero [AHIV23] and Basefold [ZCF23].

Definition 2.2 (Hamming distance)**.** The Hamming distance between two vectors $\mathbf{u}, \mathbf{v} \in \Sigma^n$, where Σ is a finite alphabet, is defined as the number of positions at which the two vectors differ.

Definition 2.3 (Code)**.** An error-correcting code C of length n over a finite alphabet Σ is a subset of Σ^n. The elements of C are called the codewords in C.

Definition 2.4 (Linear code)**.** A linear error-correcting code with dimension k, block length n, alphabet $\Sigma = \mathbb{F}$ is an injective mapping from $\mathbb{F}^k$ to a linear subspace $C \subseteq \mathbb{F}^n$. C is generated by a generator matrix $\mathbf{G} \in \mathbb{F}^{k \times n}$, such that $C = \mathsf{rowspan}(\mathbf{G})$, and an encoding of a vector $\mathbf{v} \in \mathbb{F}^k$ is $\mathbf{v}^\mathsf{T}\mathbf{G}$. We denote the rate of the code $\rho = k/n$. The minimum Hamming distance of a code, denoted by $d(C)$, is the minimum Hamming distance between two distinct codewords in C. We denote $d(\mathbf{v}, C)$ the distance between $\mathbf{v} \in \Sigma^n$ and C by $d(\mathbf{v}, C) = \min_{\mathbf{u} \in C} d(\mathbf{u}, \mathbf{v})$. If C has a minimum distance $d \in [0, n]$, we say C is an $[n, k, d]$ linear code.

Definition 2.5 (Maximum Distance Separable Code)**.** Let C be an $[n, k, d]$ code. Then C is Maximum Distance Separable (MDS) if $d = n - k + 1$.

Definition 2.6 (Reed-Solomon code)**.** For positive integers n, k, alphabet $\Sigma = \mathbb{F}$, and a vector $\boldsymbol{\eta} \in \mathbb{F}^n$ of distinct elements, the code $\mathsf{RS}_{\mathbb{F},n,k,\boldsymbol{\eta}}$ is the $[n, k, n-k+1]$ MDS linear code over $\mathbb{F}$ that consists of all n-tuples $(p(\eta_1), \ldots, p(\eta_n))$ where p is a polynomial over $\mathbb{F}$ that $\deg(p) < k$.

Definition 2.7 (Interleaved code)**.** Let C be an $[n, k, d]$ linear code over $\mathbb{F}$. We let C^m denote the $[n, mk, d]$ (interleaved) code with alphabet $\Sigma = \mathbb{F}^m$, such that each row of codeword $\mathbf{U} \in \mathbb{F}^{m \times n}$ is a codeword in C.

2.2 Multilinear Extensions

An ℓ-variate polynomial $p : \mathbb{F}^\ell \to \mathbb{F}$ is multilinear if p's individual degree is at most one, namely the degree of each variable in p is at most one. Let $f : \{0,1\}^\ell \to \mathbb{F}$ be any function mapping the ℓ-dimension Boolean hypercube to a field $\mathbb{F}$. A polynomial $g : \mathbb{F}^\ell \to \mathbb{F}$ is said to extend f if g agrees with f at any point over $\{0,1\}^\ell$. For any $f : \{0,1\}^\ell \to \mathbb{F}$, there is a unique multilinear polynomial $\tilde{f} : \mathbb{F}^\ell \to \mathbb{F}$ that extends f. The polynomial $\tilde{f}$ is referred to as the multilinear extension (MLE) of f.

A particular multilinear extension that arises frequently is $\tilde{\mathsf{eq}}$, which is the MLE of the function $\mathsf{eq} : \{0,1\}^s \times \{0,1\}^s \to \mathbb{F}$ defined as follows:

$$\mathsf{eq}(\mathbf{x}, \mathbf{e}) = \begin{cases} 1 & \mathbf{x} = \mathbf{e} \\ 0 & \text{otherwise} \end{cases}.$$

An explicit expression for $\tilde{\mathsf{eq}}$ is

$$\tilde{\mathsf{eq}}(\mathbf{x}, \mathbf{e}) = \prod_{i=1}^{s} \left(x_i \cdot e_i + (1 - x_i) \cdot (1 - e_i) \right). \tag{2.1}$$

The right hand side of Eq. (2.1) is multilinear, and that if evaluated at any input $(\mathbf{x}, \mathbf{e}) \in \{0,1\}^s \times \{0,1\}^s$, it outputs 1 if and only if $\mathbf{x} = \mathbf{e}$, and 0 otherwise. Hence, $\tilde{\mathsf{eq}}$ is the unique multilinear polynomial extending eq, and the evaluation of $\tilde{\mathsf{eq}}(\mathbf{r}_1, \mathbf{r}_2)$ at any input $(\mathbf{r}_1, \mathbf{r}_2) \in \mathbb{F}^s \times \mathbb{F}^s$ can be computed in $O(s)$ time.

Multilinear extensions of vectors. Given a vector $\mathbf{v} \in \mathbb{F}^m$, we can view $\mathbf{v}$ as a function $v : \{0,1\}^{\log m} \to \mathbb{F}$ that maps its $(\log m)$-bits input $(i_1, \ldots, i_{\log m})$ as the binary representation of an integer $i \in [0, m-1]$. We write $\tilde{v}$ to denote the multilinear polynomial extending $v : \{0,1\}^{\log m} \to \mathbb{F}$.

Lagrange interpolation. We recall an explicit expression for the MLE of any function defined over $\{0,1\}^\ell$.

Lemma 2.8 ([Set20, Tha22]). *Let $f : \{0,1\}^\ell \to \mathbb{F}$ be any function. Then polynomial $\tilde{f}$ extends f:*

$$\tilde{f}(\mathbf{x}) = \sum_{\mathbf{w}\in\{0,1\}^\ell} f(\mathbf{w}) \cdot \chi_{\mathbf{w}}(\mathbf{x}),$$

where for any $\mathbf{w} \in \{0,1\}^\ell$, $\chi_{\mathbf{w}}(\mathbf{x}) = \tilde{\mathsf{eq}}(\mathbf{x},\mathbf{w})$.

The polynomials $\{\chi_{\mathbf{w}}\}_{\mathbf{w}\in\{0,1\}^\ell}$ are called the Lagrange basis polynomials for ℓ-variate multilinear polynomials. The evaluations $\{\tilde{f}(\mathbf{w})\}_{\mathbf{w}\in\{0,1\}^\ell}$ are called the coefficients of $\tilde{f}$ in the Lagrange basis.

2.3 Polynomial Commitment Scheme

The Polynomial Commitment Scheme (PCS) is a cryptographic primitive that enables a prover to commit to a polynomial f over a field $\mathbb{F}$. Given a point $\mathbf{z}$ and an evaluation y, the prover can subsequently create a proof that it knows the committed multilinear polynomial f satisfies $f(\mathbf{z}) = y$. We now give a formal definition of a polynomial commitment scheme for multilinear polynomials.

Definition 2.9 (Polynomial Commitment [BFS20, GLS$^+$23]). A multilinear polynomial commitment scheme over a field $\mathbb{F}$ is a tuple $\Pi_{\mathsf{PCS}} = (\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Eval}, \mathsf{Verify})$ of efficient algorithms:

- $\mathsf{Setup}(1^\lambda, \ell) \to \mathsf{pp}$: On input the security parameter λ and the number of variables in a polynomial $\ell \in \mathbb{N}$, the setup algorithm outputs public parameters pp.
- $\mathsf{Commit}(\mathsf{pp}, f) \to C$: On input the public parameters pp and a ℓ-variate multilinear polynomial f, the commit algorithm outputs a public commitment C.
- $\mathsf{Open}(\mathsf{pp}, C, f) \to \{0,1\}$: On input the public parameters pp, the commitment C, and the ℓ-variate multilinear polynomial f, the open algorithm outputs a bit $b \in \{0,1\}$.
- $\mathsf{Eval}(\mathsf{pp}, C, \mathbf{z}; f) \to \pi$: On input the public parameters pp, the commitment C, the evaluation point $\mathbf{z}$, and the multilinear polynomial f, the evaluation algorithm outputs an evaluation proof π.
- $\mathsf{Verify}(\mathsf{pp}, C, \mathbf{z}, y, \pi) \to \{0,1\}$: On input the public parameters pp, the commitment C, the point $\mathbf{z}$, the purported evaluation y, and the evaluation proof π, the verify algorithm outputs a bit $b \in \{0,1\}$.

A polynomial commitment scheme Π_{PCS} is **correct** if an honest prover can always convince a verifier of a correct evaluation. Specifically, if the prover is honest, then for all multilinear polynomials f and all points $\mathbf{z}$,

$$\Pr\left[\mathsf{Verify}(\mathsf{pp}, C, \mathbf{z}, y, \pi) = 1 \;:\; \begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \ell) \\ C \leftarrow \mathsf{Commit}(\mathsf{pp}, f) \\ (y, \pi) \leftarrow \mathsf{Eval}(\mathsf{pp}, C, \mathbf{z}; f) \end{array}\right] = 1.$$

A polynomial commitment scheme Π_{PCS} is **binding** if an efficient adversary cannot produce a commitment C that can be opened to two distinct polynomials f_0, f_1. More formally, for all efficient adversaries $\mathcal{A}$,

$$\Pr\left[b_0 = b_1 \neq 0 \wedge f_0 \neq f_1 \;:\; \begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \ell) \\ (C, f_0, f_1) \leftarrow \mathcal{A}(1^\lambda, \mathsf{pp}) \\ b_0 \leftarrow \mathsf{Open}(\mathsf{pp}, C, f_0) \\ b_1 \leftarrow \mathsf{Open}(\mathsf{pp}, C, f_1) \end{array}\right] \leq \mathsf{negl}(\lambda).$$

A polynomial commitment scheme Π_{PCS} is **knowledge sound** with knowledge error ε if for all stateful efficient malicious provers $\mathcal{P}^*$, there exists an extractor $\mathcal{E}$ running in expected polynomial time such that

$$\Pr\left[b_0 = b_1 \neq 0 \wedge f(\mathbf{z}) = y \;:\; \begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \ell) \\ (C, \mathbf{z}, y, \pi) \leftarrow \mathcal{P}^*(\mathsf{pp}) \\ b_0 \leftarrow \mathsf{Verify}(\mathsf{pp}, C, \mathbf{z}, y, \pi) \\ f \leftarrow \mathcal{E}^{\mathcal{P}^*}(\mathsf{pp}, C, \mathbf{z}, y, \pi) \\ b_1 \leftarrow \mathsf{Open}(\mathsf{pp}, C, f) \end{array}\right] \leq \varepsilon(\lambda).$$

In certain cases, the PCS may necessitate a Structured Reference String (SRS) to be generated in advance by a trusted party, such as the KZG commitment scheme [KZG10]. This procedure is referred to as a trusted setup, typically facilitated through various multi-party computation protocols, and introduces additional constraints to the zero-knowledge proof system.

In practice, it is often necessary to commit to a set of polynomials and prove evaluations of multiple points simultaneously. In such scenarios, it is beneficial to batch the commitments and proofs, making the Commit and Eval algorithms potentially more efficient than computing them individually. It is also important to note that since the verifier must validate all the proofs. As a result, it is crucial for the verifier being able to batch verify all the proofs, with a sublinear cost to the number of proofs. The batching property significantly influences the size of the verification circuit and, consequently, the recursion friendliness of the entire zero-knowledge proof system.

As we will explore in subsequent sections of this paper, the selection of a PCS can profoundly influence the efficiency of the zero-knowledge proof system. We will review some of the PCSs relevant to our work in the following subsections.

2.3.1 Hyrax PCS

`Jolt` is constructed on the Hyrax PCS [WTs+18]. This scheme commits to an n-variate multilinear extension $f(\mathbf{x})$ over $\mathbb{F}$. It provides security under the discrete log assumption in the random oracle model and does not necessitate a trusted setup or a pairing friendly curve, unlike other prevalent PCSs such as the KZG [KZG10]. Asymptotically, Hyrax operates in linear time for the prover and square root time for the verifier, in relation to the size of the circuit. In practice, the prover is especially efficient when the witnesses are sparse.

Considering the VM structure of `Jolt`, we observed that the majority of the witnesses (over 80%) are zeros, while a substantial portion of the non-zero elements have low norms. This attribute significantly enhances the efficiency of the prover time in the Hyrax PCS, as it commits directly to the witnesses. Internally, Hyrax employs the Pedersen commitment [Ped92] to commit to the column vectors of a matrix $\mathbf{M}$, which comprises all the witness elements of size $N = 2^n$. The commitment process essentially involves fixed-base multi-scalar multiplications with the witnesses serving as the scalars, making it the fastest PCS for the prover.

On the other hand, the number of Pedersen commitments is therefore $O(\sqrt{N})$, each corresponding to a column of $\mathbf{M}$. The verifier is tasked with checking the correctness of each Pedersen commitment, which results in an $O(\sqrt{N})$ proof size as well as verification time. This posed a significant challenge for recursive verification, as the recursive verifier would be required to execute all the Hyrax verification logic in-circuit.

2.3.2 Zeromorph and HyperKZG PCS

The Zeromorph and HyperKZG PCSs are developed to overcome the limitations of proof size in Hyrax.

Zeromorph [KT23] comprises a two phase strategy: a 'Zeromorph transformation', which transmutes a multilinear polynomial into a univariate polynomial, and a 'Zeromorph commitment', which commits to the univariate polynomial. The latter can be instantiated with either elliptic curve-based solutions (i.e., KZG) or encoding-based solutions (i.e., FRI).

The Zeromorph transformation is a generic transformation, contingent on the univariate PCS supporting a degree check for the committed polynomial. With this support, the transformation can be implemented by opening all the univariate polynomials at the same random point. If the univariate scheme is not homomorphic, each univariate will require a separate opening. As an example of this specific process, i.e., the Zeromorph transformation with FRI batching, is referred to as ZeromorphFRI [ZCF23].

In practice, homomorphism (e.g., KZG) is used to consolidate all the univariate commitments into a single commitment, followed by a single opening for this combined commitment. This again assumes a degree check for each commitment, which are batched in Zeromorph. This is the version that is implemented in `Jolt`.

The HyperKZG scheme is akin to Zeromorph in that it also commits to a univariate polynomial. The primary difference lies in the transformation, where HyperKZG employs techniques from the tensor-product protocol introduced in Gemini [BCHO22].

It's important to highlight that both Zeromorph and HyperKZG necessitate a trusted setup, marking a considerable disadvantage in comparison to Hyrax (and our proposed solution). The verification process involves pairing checks, which may also be expensive for recursion.

3 Our modified Basefold scheme

3.1 Overview

We instantiated `Jolt-b` by replacing Hyrax with Basefold [ZCF23], which is a PCS for multilinear polynomials. Asymptotically, the Basefold scheme commits in $O(N \log N)$ time, proves in $O(N)$ time, verifies in $O(\log^2 N)$ time, and has an $O(\log^2 N)$ proof size, all with respect to the polynomial size.

More specifically, the Basefold scheme is built upon an Interactive Oracle Proof of Proximity (IOPP) for a family of foldable linear codes with an $O(N \log N)$ encoding time, which generalizes Reed-Solomon codes. The committing phase commits to a multilinear polynomial f by committing to the encoding of f with a Merkle tree, while the evaluation phase interleaves ℓ-rounds of Basefold IOPP with the classic sumcheck protocol [LFKN90]. When instantiated with Reed-Solomon code, this yields a multilinear PCS from FRI, which explains away the commit time, prover time, verifier time, and proof size.

3.2 Our instantiations

In this section, we describe our overall Basefold instantiations. We begin by describing the methodology for setting the parameters to instantiate the multilinear PCS, which includes the prime-order field modulus, the codeword rate, the generator matrices, and the Merkle tree configurations. We then describe a batching optimization built upon Basefold to prove evaluations of multiple polynomials at a same point, which improves the concrete efficiency of the resulting construction.

Basefold commitment scheme parameter selection. In the following description, we let ℓ denote the variate of the multilinear polynomial, $N = 2^\ell$ denote the size of ℓ-dimensional boolean hypercube, p denote the prime field modulus, g_i denote the primitive 2^i-th root of unity of $\mathbb{F}_p$, ρ denote the codeword rate, $\mathbf{G}_0$ denote the generator matrix for a linear code that is Maximum Distance Separable (MDS), and $\{\mathbf{T}_i\}_{i\in[0,n-1]}$ denote the diagonal matrices over $\mathbb{F}^*$. We choose the parameters as follows:

- For our Basefold instantiation from Reed-Solomon code, the codeword alphabet $\mathbb{F}$ is chosen so that $\mathbb{F}^*$ has a sufficiently large power-of-two subgroup. In our specific instantiation, we choose the 64-bit prime field with modulus $p = 2^{64} - 2^{32} + 1$, commonly referred to as the Goldilocks field [Pol21].

  In this case, we replace the original Basefold encoding algorithm [Had24] with FFT, which takes advantage of the "FFT-friendliness" property of $\mathbb{F}$.

- Though we prove the evaluations of multilinear polynomials over Goldilocks extension field of degree 2, we commit polynomials over Goldilocks base field, which accelerates the codeword generation and uses less hashes for Merkle trees.

- Compared to the Basefold reference implementation [Had24], where $\mathrm{diag}(\mathbf{T}_i)$ in each round is derived from a seeded pseudorandom function (PRF), our instantiations from Reed-Solomon code can express the derivation of $\mathrm{diag}(\mathbf{T}_i)$ compactly in constraint systems, which alleviate a large recursion circuit that was required to verify PRF in circuit.
- The choice of codeword rate ρ derives from the discussion in Plonky2 [Pol21]: namely, we choose ρ to be $1/8$ to optimize the prover time.
- We used a fixed arity of 8 for leaves when instantiating our Merkle tree to commit to the codeword, namely, each hash function takes in 8 Goldilocks base field elements as input, and outputs 256 bits to the subsequent hash functions for internal nodes. We choose the arity being 8 to optimize the prover time. This is similar to the approach used in previous implementations of proof systems from FRI [Pol21].
- We provide two distinct instantiation of the "Octopos" tree from Poseidon and SHA2. The Poseidon instantiation aims for recursion friendliness, where recursion circuit has a smaller size, while the SHA2 instantiation aims for better prover time, incurring a slightly larger resulting recursion circuit size.

Batched evaluation and verification. Crucial to the concrete efficiency of the proof system is a batched version of multilinear PCS, as without it the verifier will need to verify each of the polynomial, resulting in a linear overhead in the number of polynomials. The curve-based PCSs can be converted into batched versions by taking advantage of the linear homomorphism intrinsic in the algebraic structure. However, for encoding-based PCSs instantiated from Merkle trees, such linear homomorphism is unavailable. We introduce a batched Basefold protocol to minimize the use of Merkle tree, which has been a bottleneck for concrete efficiency.

In the following description, we let m denote the batch size, $\{f_i\}_{i\in[m]}$ denote the ℓ-variate multilinear polynomials, $\mathbf{z} \in \mathbb{F}^\ell$ denote the evaluation point, $\{y_i\}_{i\in[m]}$ denote the claimed evaluations.

Our batched evaluation protocol adapts from the batched FRI protocol [BCI+20], where we batch m polynomials into one with random linear combinations. The resulting prover runtime is $O(m \cdot N)$, while there is only one invocation to the Basefold evaluation algorithm. We present the interactive version of the batched evaluation protocol in Fig. 2.

Public input: oracles $\{\pi_{f_i} := \mathsf{Enc}_n(\mathbf{f}_i)\}_{i\in[k]}$, point $\mathbf{z} \in \mathbb{F}^\ell$, claimed evaluations $\{y_i\}_{i\in[m]}$.
Prover witness: the polynomials $\{f_i\}_{i\in[m]}$ with coefficients $\{\mathbf{f}_i\}_{i\in[m]}$.

1. $\mathcal{V}$ sends $\mathcal{P}$ a random challenge $\mathbf{t} \overset{\mathrm{R}}{\leftarrow} \mathbb{F}^{m-1}$.
2. Determine the sum $s := t_1 + \sum_{i\in[m-1]} t_i \cdot y_i$.
3. Let $\tilde{g}$ be the low degree extension of $\{f_i\}_{i\in[m]}$ evaluated at $\mathbf{t} \in \mathbb{F}^{m-1}$, where

$$\tilde{g}(\mathbf{b}) := f_1(\mathbf{b}) + \sum_{i\in[m-1]} f_{i+1}(\mathbf{b}) \cdot t_i.$$

4. Let π_g be the virtual oracle for $\tilde{g}$ constructed from π_{f_i}, where

$$\pi_g := \pi_{f_1} + \sum_{i\in[m-1]} \pi_{f_{i+1}} \cdot t_i.$$

5. $\mathcal{P}$ proves to $\mathcal{V}$ the statement $\tilde{g}(\mathbf{z}) = s$ with π_g through the vanilla Basefold evaluation protocol.

Figure 2: The batched evaluation algorithm for the Basefold PCS

Since the batched evaluation protocol in Fig. 2 is a public-coin interactive proof, we can apply Fiat-Shamir heuristic to transform it into a non-interactive proof. The batched evaluation protocol is complete from the completeness of the sumcheck PIOP and the completeness of the Basefold PCS. The knowledge soundness relies on Theorem 8.3 by Ben-Sasson et al. [BCI⁺20], which upper bounds the soundness error for FRI IOPP.

Remark 3.1. In an earlier realization for Basefold batched evaluation, we were inspired by a simplification for the batched opening protocol introduced in HyperPlonk [CBBZ23], where all the opening points are the same for all polynomials. The key idea is to merge m polynomial evaluation proofs into one via low-degree extension, such that each ℓ-variate multilinear polynomial f_i is indexed by $\log m$ bits over a $(\log m)$-dimensional boolean hypercube. Since f_i is multilinear, it suffices to constrain that

$$c_i := \left(\sum_{\mathbf{x} \in \{0,1\}^\ell} f_i(\mathbf{x}) \cdot \tilde{\mathsf{eq}}(\mathbf{x}, \mathbf{z}) - y_i \right) = 0. \tag{3.1}$$

Observe that Eq. (3.1) holds for all $i \in [m]$, if and only if the low degree extension

$$\sum_{i \in [m]} c_i \cdot \tilde{\mathsf{eq}}(\mathbf{r}, \langle i \rangle)$$

is identically zero, where $\langle i \rangle$ is $(\log m)$-bit representation of $i - 1$. This relation can be exactly proven by the "ZeroCheck PIOP" introduced in Spartan [Set20] and HyperPlonk [CBBZ23], where it suffices to check that for a random challenge $\mathbf{t} \xleftarrow{R} \mathbb{F}^{\log m}$,

$$\sum_{i \in [m]} \tilde{\mathsf{eq}}(\langle i \rangle, \mathbf{t}) \cdot \left(\sum_{\mathbf{x} \in \{0,1\}^\ell} f_i(\mathbf{x}) \cdot \tilde{\mathsf{eq}}(\mathbf{x}, \mathbf{z}) - y_i \right) = 0.$$

The resulting prover time is $O(m \cdot N)$, while our algorithm uses the same number of hashes as the one in proving a single evaluation. Since Basefold code is a linear code, we construct a virtual oracle through linear combination of the input oracles. The rest follows from the vanilla Basefold proving algorithm. We present the interactive version of the batched evaluation protocol from random tensor products in Fig. 3.

Public input: oracles $\{\pi_{f_i} := \mathsf{Enc}_n(\mathbf{f}_i)\}_{i \in [k]}$, point $\mathbf{z} \in \mathbb{F}^\ell$, claimed evaluations $\{y_i\}_{i \in [m]}$.
Prover witness: the polynomials $\{f_i\}_{i \in [m]}$ with coefficients $\{\mathbf{f}_i\}_{i \in [m]}$.

1. $\mathcal{V}$ sends $\mathcal{P}$ a random challenge $\mathbf{t} \xleftarrow{R} \mathbb{F}^{\log m}$.
2. Determine the sum $s := \sum_{i \in [m]} \tilde{\mathsf{eq}}(\langle i \rangle, \mathbf{t}) \cdot y_i$.
3. Let $\tilde{g}$ be the low degree extension of $\{f_i\}_{i \in [m]}$ evaluated at $\mathbf{t} \in \mathbb{F}^{\log m}$, where

$$\tilde{g}(\mathbf{b}) := \sum_{i \in [m]} f_i(\mathbf{b}) \cdot \tilde{\mathsf{eq}}(\langle i \rangle, \mathbf{t}).$$

4. Let π_g be the virtual oracle for $\tilde{g}$ constructed from π_{f_i}, where

$$\pi_g := \sum_{i \in [m]} \pi_{f_i} \cdot \tilde{\mathsf{eq}}(\langle i \rangle, \mathbf{t}).$$

5. $\mathcal{P}$ proves to $\mathcal{V}$ the statement $\tilde{g}(\mathbf{z}) = s$ with π_g through the vanilla Basefold evaluation protocol.

Figure 3: The batched evaluation algorithm from random tensor products for the Basefold PCS

Instead of applying random linear combinations over interleaved codewords as in Fig. 2, the linear combination coefficients in Fig. 3 are derived from $\otimes_{i\in[\log m]}[1-t_i, t_i]$, and thus the randomness complexity is $O(\log m)$ rather than $O(m)$. Similar to Fig. 2, the previous construction Fig. 3 is also complete by the completeness of Sumcheck PIOP and Basefold PCS. However, for knowledge soundness, the state-of-the-art tensor proximity gap is a third of the code's distance, by Diamond et al. [DP24].

Theorem 3.2 (Proximity test for interleaved linear codes [DP24])**.** *Fix an arbitrary $[n, k, d]$ linear code C with alphabet $\Sigma = \mathbb{F}_q$ and a proximity parameter $0 < e \le d/3$. If $\mathbf{U} \in \mathbb{F}_q^{m\times n}$ satisfies*

$$\Pr\left[d\left(\bigotimes_{i\in[\log m]}[1-r_i, r_i]\cdot \mathbf{U}, C\right) \le e\right] > 2\cdot \log m \cdot \frac{e+1}{q},$$

then $d(\mathbf{U}, C^m) \le e$.

A restrained proximity gap for interleaved proximity test leads to less soundness gained from each query, and thus a higher concrete query complexity for a same conjectured target security bit. We prefer Fig. 2 over this one, for we prefer a smaller recursive circuit size, because of a higher soundness from each query.

4 Implementation and Evaluation

We implemented and benchmarked `Jolt-b`. We divided our benchmark into several components:

- First, we compare the raw IOP component. We expect a performance improvement due to the use of a smaller field.
- Next, we compare the PCS component. We expect the PCS complexity for the prover to be much worse for the following reasons:
  - We use Basefold, an encoding-based PCS. It is not homomorphic and therefore naturally requires k commitments to commit to k polynomials.
  - The polynomials in `Jolt` are sparse, which suits elliptic curve-based commitments well. Most of the scalars are zeroes or small field elements for multi-scalar-multiplications. In contrast, encoding-based PCS generally cannot take advantage of sparseness. The elements are no longer small or sparse as soon as they are converted to their low degree extensions or hashed.
- We then present the total cost for the prover.
- We chose to use Basefold because of its verification performance. A rough estimation of the recursive prover's circuit size is sufficient. We provide three estimations: the estimated cost for the original `Jolt`, the cost for `Jolt-b` with the reference implementation of basefold [Had24], and the cost for `Jolt-b` with our batching technique.

We run all of our experiments on an Thinkpad X1 Extreme Gen4 running Linux 6.9.5. The machine has 16 CPUs (Intel i7-11850H at 4.80 GHz) and 64 GB of RAM. We show the overall benchmarks comparing runtime performances in Table 2.

Comparisons Due to the use Lasso, we observed that the majority of the witnesses (over 80%) are zeroes, while a large portion of the non-zero elements are small field elements. The sparseness of witnesses makes the prover time of the Hyrax PCS very efficient, as they commit directly on top of the witnesses; making them the fastest PCS for the prover.

In contrast, with Basefold, we need to commit to the codeword of the witnesses. The encoding process, i.e., compute the low degree extensions via FFT, is already as expensive as the entire prover time of Hyrax.

			Jolt [AST24]			This work	
			Hyrax	Zeromorph	HyperKZG	Poseidon	SHA2
SHA3	Prover	IOP	1.007s	1.064s	1.045s	354.6ms	356.7ms
		PCS.Commit	146.8ms	177.3ms	181.8ms	9.704s	2.049s
		PCS.Eval	350.6ms	1.388s	1.450s	2.017s	1.309s
		Total	1.504s	2.63s	2.677s	12.076s	3.715s
	Verifier Time		138.7ms	62.4ms	63.2ms	294ms	14.5ms
	Circuit Size (gates)		8.7×10^{10}	2.2×10^9	2.2×10^9	4.8×10^7	7.2×10^9
	Proof Size		433.73kb	253.12kb	263.92kb	29.5mb	
Fibonacci	Prover	IOP	299.6ms	328.1ms	319.8ms	75.9ms	79.9ms
		PCS.Commit	9.7ms	23.9ms	18.8ms	3.034s	635.7ms
		PCS.Eval	42.7ms	408.5ms	488.3ms	760.3ms	415.4ms
		Total	351.9ms	760.6ms	826.9ms	3.870s	1.131s
	Verifier Time		35ms	57ms	58.5ms	219ms	12.1ms
	Circuit Size (gates)		2.8×10^{10}	2.3×10^9	2.3×10^9	3.2×10^7	4.7×10^9
	Proof Size		874.37kb	342.56kb	356.92kb	19.1mb	

Table 2: Concrete performance comparison of `Jolt-b` to the instantiations of `Jolt` from various PCSs. It is important to recognize that the substantial proof size of `Jolt-b` originates from its use of Basefold. Specifically, HyperPlonk when combined with Basefold [ZCF23] results in a proof size over 6 MB, marking an increase of 800-fold compared to HyperPlonk with KZG.

The reminder of the PCS time, i.e., building the Merkle tree, can be configured with different hash functions, such as SHA2 or Poseidon, targeting different trade-offs between prover time and recursion friendliness. In the end, our SHA2 version of the scheme is 1.85× slower than the Hyrax PCS, while the Poseidon version is 6.44× slower.

On the other hand, to get a sense of how much computation the prover need to prove the computation in the tests, we ran the tests over single-threaded settings together with the default multi-threaded settings. The runtime comparisons are shown in Table 3, where we believe there are still space for improvement for `Jolt-b` in multi-threaded settings.

Estimation of the Recursive Prover's Circuit Size In this section, we provide an estimation of the recursive prover's circuit size, utilizing the state-of-the-art circuit from Halo2. The general rule of thumb with Halo2 is to count the number of witnesses. Please note that these estimations are approximate and may

		Jolt [AST24]			This work	
		Hyrax	Zeromorph	HyperKZG	Poseidon	SHA2
SHA3	Multi-Threaded Prover	1.504s	2.63s	2.677s	12.076s	3.715s
	Single-Threaded Prover	6.796s	11.434s	14.091s	71.511s	9.422s
Fibonacci	Multi-Threaded Prover	351.9ms	760.6ms	826.9ms	3.870s	1.131s
	Single-Threaded Prover	1.464s	3.165s	3.875s	31.742s	2.625s

Table 3: Concrete performance comparison in the prover times of our work to the prover times of `Jolt` under multi-threaded setting and single-threaded setting.

not be entirely accurate. They are intended to provide a sense of the scale of the circuit size.

When represented in Halo2's circuit:

- A group operation requires over 2×10^6 witnesses.
- A Poseidon hash function requires slightly over 100 witnesses.
- A Sha2 hash function requires around 1.5×10^4 witnesses.

For the recursive circuit of Sha3 in `Jolt` and `Jolt-b`, the requirements for the verifier are as follows:

- With `Jolt` and Hyrax, the verifier needs to perform 4.4×10^4 group operations, with a total of 8.8×10^{10} witnesses.
- With `Jolt` and Zeromorph, the verifier needs to perform 1.2×10^3 group operations, with a total of 2.4×10^9 witnesses, assuming the final pairings are deferred.
- With `Jolt` and HyperKZG, the verifier needs to perform 1.1×10^3 group operations, with a total of 2.2×10^9 witnesses, assuming the final pairings are deferred.
- With `Jolt-b`, the verifier needs to verify 4.8×10^5 hashes in total.
  - If the hash is instantiated with Poseidon, the total cost equals to 4.8×10^7 witnesses.
  - If the hash is instantiated with Sha2, the total cost equals to 7.2×10^9 witnesses.

We remark that the recursion circuit for `Jolt-b` with Poseidon is 2 to 3 orders of magnitude smaller than that of the original `Jolt`. It's worth noting that without the batching technique, the verifier would need to verify each of the proofs individually, requiring approximately 78.2 billion and 1.1 trillion witnesses for Poseidon and Sha2, respectively.

Acknowledgments

We thank Binyi Chen and Xiong Fan for helpful comments and pointers.

References

[ACFY24] Gal Arnon, Alessandro Chiesa, Giacomo Fenzi, and Eylon Yogev. STIR: Reed–solomon proximity testing with fewer queries. Cryptology ePrint Archive, Paper 2024/390, 2024. https://eprint.iacr.org/2024/390.

[AHIV23] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: lightweight sublinear arguments without a trusted setup. *DCC*, 91(11):3379–3424, 2023.

[AMPS24] Tomer Ashur, Mohammad Mahzoun, Jim Posen, and Danilo Sijacic. Vision mark-32: ZK-friendly hash function over binary tower fields. Cryptology ePrint Archive, Paper 2024/633, 2024. https://eprint.iacr.org/2024/633.

[AST24] Arasu Arun, Srinath Setty, and Justin Thaler. Jolt: Snarks for virtual machines via lookups. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 3–33. Springer, 2024.

[BCHO22] Jonathan Bootle, Alessandro Chiesa, Yuncong Hu, and Michele Orrù. Gemini: Elastic SNARKs for diverse environments. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part II*, volume 13276 of *LNCS*, pages 427–457. Springer, Heidelberg, May / June 2022.

[BCI$^+$20] Eli Ben-Sasson, Dan Carmon, Yuval Ishai, Swastik Kopparty, and Shubhangi Saraf. Proximity gaps for reed-solomon codes. In *61st FOCS*, pages 900–909. IEEE Computer Society Press, November 2020.

[BFS20] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. Transparent SNARKs from DARK compilers. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 677–706. Springer, Heidelberg, May 2020.

[CBBZ23] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. HyperPlonk: Plonk with linear-time prover and high-degree custom gates. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part II*, volume 14005 of *LNCS*, pages 499–530. Springer, Heidelberg, April 2023.

[DP24] Benjamin E. Diamond and Jim Posen. Proximity testing with logarithmic randomness. *IACR Communications in Cryptology*, 1(1), 2024.

[Dra24] Justin Drake. SNARK proving ASICs. ZK Summit 11, 2024. https://www.youtube.com/watch?v=URCH2d1cdyg.

[GLS$^+$23] Alexander Golovnev, Jonathan Lee, Srinath T. V. Setty, Justin Thaler, and Riad S. Wahby. Brakedown: Linear-time and field-agnostic SNARKs for R1CS. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part II*, volume 14082 of *LNCS*, pages 193–226. Springer, Heidelberg, August 2023.

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

[Had24] Hadas Zeilberger. Basefold reference implementation. https://github.com/hadasz/plonkish_basefold, 2024.

[KT23] Tohru Kohrita and Patrick Towa. Zeromorph: Zero-knowledge multilinear-evaluation proofs from homomorphic univariate commitments. Cryptology ePrint Archive, Paper 2023/917, 2023. https://eprint.iacr.org/2023/917.

[KZG10] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 177–194. Springer, Heidelberg, December 2010.

[LFKN90] Carsten Lund, Lance Fortnow, Howard J. Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. In *31st FOCS*, pages 2–10. IEEE Computer Society Press, October 1990.

[LZZ$^+$24] Tianyi Liu, Zhenfei Zhang, Yuncong Zhang, Wenqing Hu, and Ye Zhang. Ceno: Non-uniform, segment and parallel zero-knowledge virtual machine. Cryptology ePrint Archive, Paper 2024/387, 2024. https://eprint.iacr.org/2024/387.

[Ped92] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 129–140. Springer, Heidelberg, August 1992.

[Pol21] Polygon Zero. Plonky2. https://github.com/0xPolygonZero/plonky2, 2021.

[Pol23] Polygon Zero. Plonky3. https://github.com/Plonky3/Plonky3, 2023.

[Sch80] Jacob T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4), 1980.

[Set20] Srinath Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part III*, volume 12172 of *LNCS*, pages 704–737. Springer, Heidelberg, August 2020.

[Tha22] Justin Thaler. Proofs, arguments, and zero-knowledge. *Foundation and Trends in Privacy and Security*, 4(2–4):117–600, 2022.

[WTs+18] Riad S. Wahby, Ioanna Tzialla, abhi shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zkSNARKs without trusted setup. In *2018 IEEE Symposium on Security and Privacy*, pages 926–943. IEEE Computer Society Press, May 2018.

[ZCF23] Hadas Zeilberger, Binyi Chen, and Ben Fisch. Basefold: Efficient field-agnostic polynomial commitment schemes from foldable codes. Cryptology ePrint Archive, Paper 2023/1705, 2023. https://eprint.iacr.org/2023/1705.

[Zip79] Richard Zippel. Probabilistic algorithms for sparse polynomials. In *EUROSAM*, 1979.