

A faint, semi-transparent background image of the London skyline at night, showing the Palace of Westminster, Big Ben, and the River Thames with a bridge.

Cryptography Research Update

January 2023



Dear Geometers,

We're delighted to present the company's research snapshot for 2022, comprising some of the cryptography papers we think have made the most impact on our industry this year.

The research in this booklet is broken into three sections:

1. Geometry Team
2. Geometry PortCos
3. Wider Industry

Let's dive into some of the important themes we saw this year.

HyperPlonk

One of the most dramatic announcements this year was made at zkSummit 8 in Berlin, where the Espresso Systems team presented HyperPlonk. This new proof system has potentially significant consequences for the state of ZK Proofs:

- **Elimination of FFTs:** FFTs (or more generally NTTs) start to dominate prover costs for large circuits, so for computations that are not easily parallelised (i.e. those that can't easily be broken into smaller circuits), HyperPlonk could be a very significant breakthrough, and we will watch implementations closely in 2023.
- **High-degree Custom Gates:** HyperPlonk additionally allows for higher degree custom gates, enabling deeper optimisations of important primitives, such as elliptic curve operations or algebraic hashes. Using these, though, will remove the ability to use HyperPlonk as a drop-in replacement in an existing PLONK implementation.

However, there are limitations — recent research has shown NTTs can be accelerated efficiently in hardware, and that sumchecks, which are a core component in NTT-less proving systems such as HyperPlonk, might not be as hardware friendly.

We foresee some tension between low-level optimisations and concrete constants continuing into 2023.

The Year of Lookups

Which brings us to the topic of lookups.

The major research theme of zero knowledge research this year has been the progress of lookups. These arguments introduce special-purpose gates / constraints to zero knowledge proving systems, allowing systems architects to reduce down the number of gates in their circuits, and therefore to make order-of-magnitude advances in prover performance.

Five Fast Lookup Proofs in 2022

We have had five successive breakthroughs in lookup performance this year, each an improvement on the last. Here's a snapshot of their performance, focussing just on prover time.

Using the notation:

- n = Size of Witness Vector
- t = Size of Lookup Table

In 2022 alone, we have seen advances in prover times as follows:

- **Caulk** | $\mathcal{O}(n^2 + n \log(t))$ | ~ Quadratic
 - **Operations:** In the field (cheap) and the small group (semi expensive)
- **Caulk+** | $\mathcal{O}(n^2)$ | Quadratic
 - **Operations:** In the field (cheap) and the small group (semi expensive)
- **Flookup** | $\mathcal{O}(n \log^2(n))$
 - **Operations:** In the field, with $\mathcal{O}(n)$ operations in the small group (semi expensive) and the big group (more expensive)
- **Baloo** | $\mathcal{O}(n \log^2(n))$
 - **Operations:** Same magnitude as Flookup, but is homomorphic (commitments add together sensibly)
- **cq** | $\mathcal{O}(n \log(n))$
 - **Operations:** In the field, with $\mathcal{O}(n)$ operations in the small group (semi expensive, but only linear in number)

What does this mean?

We expect this research to give rise to the unlocking of various technologies in 2023 including:

1. Performant zkEVM

- Needs fast lookups for hash computations
- Our portfolio company **Scroll** will be an early beneficiary of this

2. More Expressive VMs

- New L2 infrastructures can start to play with larger word sizes in boolean paradigms
- Companies running more classical VMs, such as **RISC Zero**, may benefit from this relief

3. Range Checks

- Wide range of applications that will benefit from more performant range checks
- This includes identity systems and verifiable inference models ("zkML")
- Other applications leveraging big-integer arithmetic and fixed-width integers will also benefit, such as signature verification on non-matching curves and proof composition

Outlook for 2023

With the markets in a state of disintegration, and the visible surface of the industry badly tarnished yet again, it is heartening to see exceptional researchers continuing to make extraordinary breakthroughs that will reduce the cost of Web3 execution, and enable a new wave of applications that could not previously afford to exist on public networks.

We look forward to improvements in developer friendliness, safety and expressivity of ZK tooling. We're also excited to see other areas of cryptography and distributed systems, such as MPC and FHE, getting into the hands of developers.

With our best wishes for 2023,

Tom & Kobi

A Gentle Introduction to Lookups

Tom Walton-Pocock

Here's a gentler introduction for those less familiar with the particulars of ZKP schemes, to help add in a bit of context for why lookup arguments are so vital to the mission of fast state machines on Ethereum.

Arithmetic Circuits

In 2020, a new technique known as Plookup entered the nascent PLONK universe, allowing acceleration of computations that were previously expensive in SNARKs.

Recall that PLONK is a zero knowledge proof algorithm created by Gabizon and Williamson at Aztec in 2019, that creates small proofs that someone (the 'prover') has computed an "arithmetic circuit" (a very bare bones representation of a computer programme — or, as we call programmes that run on blockchains, a smart contract).

Remember also that the paradigm of ZKP-based scaling is to put a huge extra burden onto a centralised agent (the proof / block constructor) in exchange for taking an enormous computational burden away from the blockchain network. This enables the blockchain network to remain decentralised (enable low-performance

Arithmetic circuits are very simple to describe — they are addition (+) and multiplication (x) gates, connected together to form the desired programme:

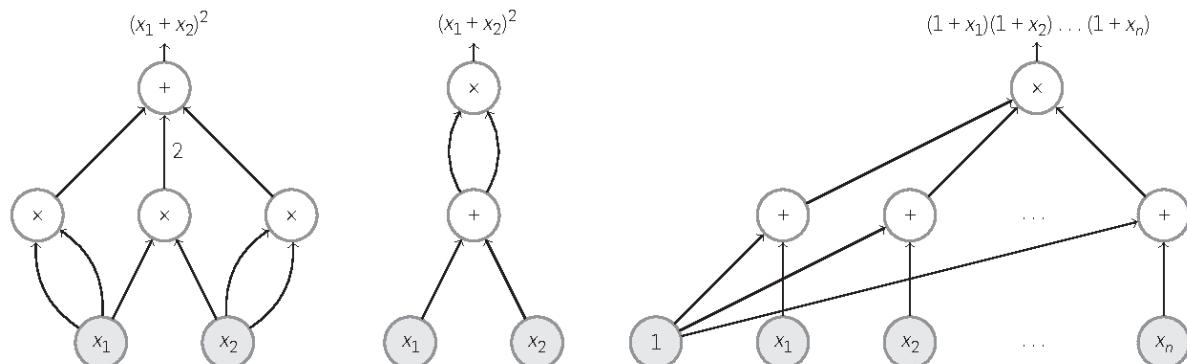


Figure: Examples of simple arithmetic circuits, from Communications of the ACM, Volume 60, Issue

EVM Limitations

When used in a ZKP, the numbers travelling through these programmes live inside a huge prime modulus.

Unfortunately, when Gav Wood and Vitalik Buterin were constructing the Ethereum state machine in 2014, Ethereum's future dependence on ZK Proofs was not visible. So they made some understandable choices, such the inclusion of 256-bit integers (because those would nicely hold the output of hashes) which they might not have done had SNARKs been more performant.

Furthermore, Ethereum state and transaction trees are built from the Keccak-256 hash, which like most hashes is full of boolean operations (i.e. does operations on bits like a classical computer).

But bits and booleans are not at all compatible with prime-modulus world.

So, translating between these two worlds — the world of bits, and the world of numbers in a prime modulus, hugely blows up the number of gates required to do an Ethereum transaction.

In other words, building zkEVM in basic PLONK is prohibitively expensive, creating very high proving times.

Enter Lookups

In 2020, Plookup entered the fray — a technique from the authors of PLONK, inspired by the lookup tables from the semiconductor industry, a type of 'super gate' to be created for specific, recurring computations.

One computation that occurs again and again in doing Ethereum transactions is the Keccak-256 hash function mentioned above. Instead of writing that operation fully in addition and multiplication gates (it would take gates to do so), important units of the computation can be shipped in as 'lookup gates'.

The trick is — instead of doing the full boolean computation, just check that the inputs (which are numbers in a prime modulus) and the outputs (which are also numbers in a prime modulus) are consistent with the operation you care about. A good example of that computation is the XOR operation.

Table of Contents

PART 1 | Geometry Research

An ECDSA Nullifier Scheme for Unique Pseudonymity within Zero Knowledge Proofs

Mental Poker in the Age of SNARKs, Parts 1 & 2

Groth16 Malleability

Hashing to the secp256k1 Elliptic Curve

Optimized BLS multisignatures on EVM

PART 2 | Portfolio Company Research

Ingonyama | A Mathematical Theory of Danksharding

Ingonyama | Fast Modular Multiplication

Ingonyama | SoK: Hash functions in Zero Knowledge Proofs

Ingonyama | PipeMSM: Hardware Acceleration for Multi-Scalar Multiplication

Ingonyama | Goldilocks NTT

Ingonyama | NTT Mini: Exploring Winograd's Heuristic for Faster NTT

RISC Zero | The RISC Zero Proof System: A Step-By-Step Description

RISC Zero | RISC Zero zkVM: Scalable, Transparent Arguments of RISC-V Integrity

Scroll | zkEVM

Slush | A Proposal for Fractal Scaling

Socket | Unity is Strength: A Formalization of Cross-Domain Maximal Extractable Value

Socket | Assessing Blockchain Bridges

PART 3 | Wider Industry Research

Caulk, Lookup arguments in sublinear time

Caulk+, Table-independent lookup arguments

Flookup, Fractional decomposition-based lookups in quasi-linear time

Baloo, Nearly optimal lookups

cq, Cached quotients for fast lookups

HyperPlonk, Plonk with linear-time prover and high-degree custom gates



Part 1 | Geometry Research



An ECDSA Nullifier Scheme for Unique Pseudonymity within Zero Knowledge Proofs

Aayush Gupta, Kobi Gurkan



An ECDSA Nullifier Scheme for Unique Pseudonymity within Zero Knowledge Proofs

by

Aayush Gupta*, Kobi Gurkan**

Abstract

ZK-SNARKs (Zero Knowledge Succinct Noninteractive ARguments of Knowledge) are one of the most promising new applied cryptography tools: proofs allow anyone to prove a property about some data, without revealing that data. Largely spurred by the adoption of cryptographic primitives in blockchain systems, ZK-SNARKs are rapidly becoming computationally practical in real-world settings, shown by i.e. tornado.cash and rollups. These have enabled ideation for new identity applications based on anonymous proof-of-ownership. One of the primary technologies that would enable the jump from existing apps to such systems is the development of deterministic nullifiers.

Nullifiers are used as a public commitment to a specific anonymous account, to forbid actions like double spending, or allow a consistent identity between anonymous actions. We identify a new deterministic signature algorithm that both uniquely identifies the keypair, and keeps the account identity secret. In this work, we will define the full DDH-VRF construction, and prove uniqueness, secrecy, and existential unforgeability. We will also demonstrate a proof of concept of the nullifier.

Thesis Supervisor: Vinod Vaikuntanathan (for Aayush's thesis)

Title: Professor

* = 0xPARC, MIT

** = Geometry Research

Acknowledgments

Thanks to Lakshman Sankar for guidance, especially for helping answer questions on security and validating the direction on the day-to-day, and helping me stay accountable via meetings and check-ins.

Thanks to Kobi Gurkan for helping me think through the nitty-gritty of the proofs, especially weekend calls to explain the details of the papers, pointing to useful papers and resources, and basically coming up with the research direction.

Thanks to Wei Jie Koh for helping to write a formal specification of my Rust code to help production adoption.

Thanks to Vivek Bhupatiraju for helping me whiteboard details of the trickier proofs.

Thanks to Remco and Kobi for helping put together the full picture of the quantum insecurity of the algorithm.

Thanks to Nalin, Adhyayan, and Brian Gu for helping to come up with the original Stealthdrop construction that motivated this direction, and for brainstorming for days on possible solutions to address this problem.

Thanks to Justin Glibert for insightful discussion and piercing questions for each topic I considered along the way.

Thanks to Jordi Baylina and Blaine Bublitz for support with circom and snarkjs toolstacks.

Thanks to Prof. Yael Kalai for helping to review this paper.

Thanks to Prof. Dan Boneh and Riad Wahby for helping to validate the original research direction and suggesting some great changes early.

Thanks to Prof. Vaikuntanathan for approving my proposal and thesis, and helping validate the research direction.

Contents

1	Introduction	7
1.1	Contributions	9
1.2	Applications	10
1.3	Related Work	11
1.4	Roadmap	13
2	Preliminaries	15
2.1	Hashing	15
2.2	Signatures	16
2.2.1	ECDSA	16
2.2.2	Pairing Friendly Signatures	16
2.2.3	Other Signatures	16
2.3	Ethereum	17
2.3.1	Constraints	17
2.4	ZK SNARKs	18
3	Nullifier Scheme	19
3.1	Construction	19
3.2	Proofs	21
3.2.1	Definitions	21
3.2.2	Proof of Uniqueness	22
3.2.3	Proof of Existential Unforgeability	23
3.2.4	Proof of Secrecy	24

4 Discussion	27
4.1 Performance	27
4.2 Morality and Mitigations	27
4.3 The Interactivity-Quantum Secrecy Tradeoff	28
4.4 Future Work	29
5 Conclusion	31

Chapter 1

Introduction

Due to the widespread adoption of the Ethereum Virtual Machine (EVM) by various cryptography research organizations and cryptocurrency platforms, ECDSA signatures and keypairs are now commonplace. As a result, porting existing primitives to the pairing-unfriendly secp256k1 elliptic curve is desirable for widespread research adoption.

Spurred by the adoption of cryptocurrency, the ability to reveal specific parts of your identity in lieu of the whole is an interesting new primitive [9]. Signatures via ECDSA are nondeterministic, meaning a signer can produce an arbitrary number of signatures for a message. However, they do provide non-repudiation of signed data, and enable applications such as semi-anonymous message boards.

For instance, one can prove via a zero knowledge proof of knowledge (ZKP) that their account satisfies some property i.e. they are an NFT holder or protocol user, without revealing who they are. Set membership proofs are a typical usecase of zk-snarks for privacy [8], and we can imagine a zk proof of the form "I can prove that I own the private key [via generating a valid signature] for some public key that is a leaf of the merkle tree comprised of all set members, with this public merkle root." For such applications, a person merely needs to prove the existence of at least one valid signature per message to be sure that such a message is legitimate [19]. However, such applications have the advantage that there is no uniqueness constraint on the provers: that is, the same wallet proving itself as a member multiple times is

intended behavior. However, many applications require a maximum of one action per user, especially protocols that desire sybil resistance like claiming an airdrop.

For a concrete example, a zero knowledge airdrop [26] requires that an anonymous claimer can produce some unique public identifier of a claim to a single address on an airdrop, that does not reveal the value of that address[35]. One can imagine a "claimer" can send a zk-proof of knowledge of merkle path to some public merkle root, along with a proof of private key ownership (whether signature verification or something else). A public nullifier signal ensures they cannot claim again in the future. This unique public identifier is coined a "nullifier" because the address nullifies its ability to perform the action again. For a specific nullifier function, a hash of the public key would be ideal but can easily be brute forced due to the finite number of on-chain addresses, so alternate solutions are needed. One can imagine that a signature would be an ideal nullifier; however, most signature algorithms (and all presently deployed ones on Ethereum) have 2^{256} valid signatures for the same message and public key, so you can imagine someone can "prove" membership an effectively infinite number of times and have a different "nullifier" each time, defeating the purpose of a unique nullifier.

One can then imagine that $\text{hash}(\text{message}, \text{secret key})$ is a decent nullifier, where each app has (usually) one canonical message – there's no way to reverse engineer the secret key, it's lightweight and computable in a hardware wallet, and it is unique for each account and application. However, we can't verify it without access to the secret key itself. For security reasons, we want a way to be able to do all of these computations without a user having to insert their private key anywhere, especially not copy paste it as plaintext. For anything that does need a private key, we want computation to be very lightweight so we can run it on a hardware wallet as well if needed: the complex elliptic curve pairing functions required to prove ZK SNARKs are not feasible to compute in current memory-constrained hardware wallets, and because ZK proof systems evolve so rapidly, we don't want wallets to commit to a specific system this early. However, this simple construction provides the key insight that we use to motivate our nullifier scheme.

If we want to forbid actions like double spending or double claiming, we need these nullifiers to be unique per account. Because ECDSA signatures are nondeterministic, signatures don't suffice; we need a new deterministic function, verifiable with only the public key. We want the nullifier to be non-interactive, to uniquely identify the keypair yet keep the account identity secret. Overall, the key insight is that such nullifiers can be used as a public commitment to a specific anonymous account.

1.1 Contributions

We implement and verify a method for a non-interactive nullifier scheme. We also provide an explanatory blog post with intuitions for layfolk.

The exact definition of a successful nullifier is as follows: we want a deterministic nullifier function $N(sk, pk)$, for an ECDSA keypair (sk, pk) , to have the following properties, for some application-specific message m , which can also be considered to encapsulate information about any common reference strings. Note that $proof$ can be any proof transcript at all, including an adversarially chosen one. ϵ is any negligible function. Note that these definitions may have more inputs as well, such as common reference inputs. We also assume that no address has public key 0 for ease of notation.

1. **Correctness.** For all pk_i , and verification function Ver , $Ver(N(m, sk_i, pk_i), proof_j, m, pk_i) = pk_i$, for all such $proof_j$, and $P[V(x, x \neq N(m, sk_i, pk_i) \forall i, proof) = 0] > 1 - \epsilon$.
Note that Ver does not take any private input, thus it must be possible to verify the signature scheme with only public information.
2. **Uniqueness.** Any keyholder cannot generate two nullifiers. Formally, $V[x, proof_x] = pk_1, V[y, proof_y] = pk_1$) implies $x = y$. This implies the function must be deterministic.
3. **Secrecy.** Also called unpredictability/hiding [18]. An adversary \mathcal{A} with pk_i , a succinct list of all valid public keys, as well as pk_j , a list of all public keys approved by the set inclusion algorithm, cannot distinguish two keys

given the nullifier, the only public ZK proof output. For $pk_1, pk_2 \in \{pk_j\}$,
 $P(\mathcal{A}[N(sk_1, pk_1), N(sk_2, pk_2), \{pk_j\}] = (pk_1, pk_2)) < 0.5 + \epsilon$.

Note these definitions are almost identical to VUFs [18], but all current implementations of VUFs are not compatible with the secp256k1 curve. Note that it will usually be desirable to keep the public key private using this nullifier, so we intend the verification step to occur within a zk-proof. Our definitions can be achieved via DDH-VRFs that incorporate ZK-SNARK verifiers instead of classical sigma protocols [6], since proving public key set membership is much easier in ZK-SNARKS.

1.2 Applications

Standardizing such a nullifier scheme into at least one wallet would make any applications that rely on unique, anonymous identities practical. These include airdrops, persistent identities on message boards, and uniqueness claims for proof of ownership.

ZK airdrops are achieved via publishing a set accumulator (i.e. a Merkle tree) of allowed users and having users submit a proof to claim an airdrop from an unlinked account. Eligible people would prove proof of private key ownership via signature, along with a proof that the corresponding public key is in the Merkle tree by providing a Merkle path, and in the same proof verify their nullifier as a check for double-claiming.

Message boards can have a persistent anonymous identity, meaning someone can post under the same nullifier multiple times, and everyone knows that they are the same person, but not who that person is. Users would simply upload a nullifier where the message corresponds to the unique ID of the thread, along with an ECDSA signature from their public key in the same ZK-SNARK of the message contents and timestamp where the signature verification is kept private in the SNARK.

We can also build a more secure version of [tornado.cash](#), where instead of leaking the nullifier string in plaintext on the frontend from a server via the website, the nullifier can simply be this ECDSA nullifier corresponding to some message in a standard format, like “[tornado.cash](#) note 5 for 10 eth”.

We think that wallets that adopt this standard first will hold the key for their users being able to interact with the next generation of ZK applications first. We hope this standard becomes as commonplace as ECDSA signing, within every secure enclave.

1.3 Related Work

This specific problem has only recently become of particular interest due to the aforementioned zero knowledge applications reaching production. Nullifiers have used since zcash [4] and tornado.cash [2], but we have a non-interactivity requirement that they may not. Without a non-interactivity requirement, a user could simply hash a random string and use another hash of that preimage as their nullifier from then on: tornado.cash uses a similar scheme based on random strings. However, we want to be able to publish a nullifier without any such interaction, so it is much harder to build off of their constructions.

It is also not possible to create an algebraic nullifier function using only one group and its algebraic operations, as proven in [1]. However, using tools such as MPC or pairings may allow us to map between multiple fields/groups and thus not be solely algebraic. In addition, treating a hash function as a pseudorandom oracle also breaks algebraic constraints, as the output of a hash function cannot be expressed as a linear combination of its inputs.

There is also research on fair ZK, in which a unique witness can be made to have a deterministic ZK proof, and thus a hash can be used as a nullifier [23]. However, this is not presently useful, as we still need a deterministic nullifier to prove we know the private key. Note that our construction is also similar to a unique ring signature over ECDSA, but where each public key, for each set, emits a unique signature that doesn't identify them [29].

One proposed solution is using MPC to shuffle an encrypted secret, first proposed by Riad Wahby. This solution would allow us to have deterministic nullifiers with guarantees up to a group instead of an individual; users would simply add their

nullifiers to an anonymity set, which would be shuffled in such a way that we can detect if someone in the group is interacting twice with the same application, but we can't pinpoint which one. This would work by first having a coordinator publish encrypted secrets, and future participants "shuffle" which messages are encrypted by which addresses, leading to a 1-of-n trust guarantee (i.e. at least one shuffler must be honest for it to be unlinkable). This primarily works with RSA-style encryptions and does not seem practical for our use case, due to the interactivity required to generate a valid RSA key pair ahead of time. Verifiable random shuffles also depend on the existence of a homomorphic encryption function, which the standardized Metamask ed25519 encryption algorithm (*X25519_XSalsa20_Poly1305* is standard, with a MAC and with keypairs derived from ECDSA private keys) is not. Note this is impractical for our setting, as ECDSA has no associated encryption/decryption function (except for ECIES, which relies on a Diffie-Hellman like setup).

There is additionally another body of work on verifiable unpredictable functions (VUFs), which has the same zero knowledge and security guarantees as the solution we are looking for [18]. The output of some VUF would be used to generate either encrypted randomness for a user that only they can decrypt, or as the basis for a deterministic signature. However, research is scant especially compared to VRFs (verifiable random functions). The algorithm outlined in the VUF paper [18] is promising but requires a pairing friendly curve to verify, similar to BLS. These are usually groups where decisional Diffie-Hellman (DDH) is easy, but computational Diffie-Hellman (CDH) is hard. However, ECDSA does not operate on the same elliptic curve, and DDH is hard so it is pairing unfriendly. However, exploring other pairing based schemes could be promising in the future.

This leaves one final avenue for us to proceed on. We want a deterministic function of a user's secret key, that can be verified with only their public key, eventually done inside a ZK-SNARK to keep the public key itself anonymous behind a set membership check. We choose to implement and standardize such a deterministic signature scheme, specifically building atop a concrete DDH-VRF scheme [6], such as EC-VRF [15] as originally described in a paper about NSEC5 in DNSSEC [27].

1.4 Roadmap

We first provide definitions of all the relevant existing cryptographic primitives needed to understand the nullifier construction.

We then present the nullifier construction of the BLS-like [5] signature scheme. The construction is based on section 3.2 of [10] and BLS, and is effectively fixed Goh-Jarecki signatures where we substitute r inside the hash with pk instead to make it deterministic [14]. We prove all the relevant nullifier requirements.

Finally, we present moral considerations and present benchmarks, as well as describe future work.

Chapter 2

Preliminaries

In this section, we will introduce the hashing primitives, signature algorithms, ZK-SNARK systems, and internet infrastructure systems that our construction relies on.

2.1 Hashing

Our nullifier algorithm relies on several hashing algorithms, including SHA256, SHA512, and the IETF RFC standard hash to curve algorithm for secp256k1 QUUX-V01-CS02-with-secp256k1_XMD:SHA-256_SSWU_RO_ (we will abbreviate this last algorithm as hash2curve henceforth, for brevity) [12].

There are many SNARK-friendly hash functions, like MiMC, Poseidon, or Pederson. However, we didn't use those in our initial proof of concept and paper, because we want to guarantee maximum compatibility and probability of adoption by wallets. If a prominent wallet provider indicates a willingness to use one of these hash functions, we believe it is advantageous to switch our scheme to use it.

Note that our construction also relies on a hash to curve algorithm, because a hash multiplied by the generator would break the existential unforgeability of any signature scheme dependent on it [33].

We assume these hash algorithms are collision resistant and deriving a preimage is hard given the hashed value, and breaking each requires 2^n queries for some n.

2.2 Signatures

2.2.1 ECDSA

ECDSA is the signature protocol used by Bitcoin, Ethereum and most blockchain systems [25, 7], due to both Schnorr’s copyright and ECDSA’s relatively smaller key size, especially when compared to RSA. Most RSA keys are 2,048 bits, but the much shorter 256-bit ECDSA key provides roughly equal security to a 3,248 bit RSA key [28].

ECDSA uses the secp256k1 curve, meaning all the points are on $y^2 = x^3 + 7$ [20]. Because almost all existing blockchain and public key infrastructure uses this curve for non-deterministic signatures, we are interested in a nullifier construction for this class of curves specifically.

2.2.2 Pairing Friendly Signatures

BLS is a deterministic proof over pairing-based curves. One notable thing about secp256k1 is that it is not pairing friendly. One clear reason for this (of many) is that decisional Diffie Hellman and computational Diffie Hellman are both hard in secp256k1; but, in pairing friendly curves such as the ones used in BLS, decisional Diffie Hellman is easy, while computational Diffie Hellman is still hard [5]. Note that this means that a simple deterministic signature scheme (which would give us our ideal nullifier for free) such as BLS will not work with ECDSA, although if we use its form as an inspiration [5].

2.2.3 Other Signatures

Note that RSA signatures are deterministic, which means that those signatures could directly be proven in the ZK-SNARK – we wouldn’t need a bespoke signature scheme.

Verifiable Unpredictable Functions, or VUFs, are another pairing-curve based construction that allows specific construction of nullifier-style identities [18] and are impractical for ECDSA largely due to the reliance on the pairing check.

2.3 Ethereum

Ethereum is a data available public ledger that can act as a decentralized verification layer for our contracts and SNARKs. There are several types of wallets, including secure enclave enabled hardware wallets, software wallets that run on the OS level, and browser wallets that can query any secure enclaves required and directly interact with smart contracts via web interfaces. We want to isolate simple operations that leak the secret key to the secure enclave, isolate the resulting signals that leak anonymity within the secure wallet application or a client-side-only web app, and only reveal the ZK-SNARKed public data to the world. To ensure data availability and censorship resistance, ideally this data is hosted via a credibly neutral blockchain layer that can act as a 3rd party verifier, trusted via decentralization.

2.3.1 Constraints

We want a nullifier algorithm to be practical for both in-browser wallets such as Metamask and hardware wallets such as Ledger and Trezor. However, especially hardware wallet environments are considerably resource-constrained: the inexpensive secure enclave chips can only compute limited functions, and the smallest ones have a total of 320 kB of flash memory for all applications [34]. Thus, it will take impractical amounts of time (and is currently infeasible with that amount of memory) to compute a function as complex as a SNARK proof.

This constraint is the reason that we cannot generate a simple ZK-SNARK proof of knowledge of a private key that generates a valid public key, with the anonymity set as the only public input to the circuit. Even as memory limits increased, we do not want to commit secure enclaves to a specific proof system or proof, which would block applications from adding bespoke checks to the proof or upgrading the proving system. Thus, we opt to separate the calculation of signals in the enclave, from the verification of the signals in a ZK SNARK outside the enclave.

Note that we prefer to use a ZK-SNARK instead of a non-interactive Sigma protocol for ZK, because we want to additionally allow applications to easily add checks

inside the zero knowledge proof – for instance public key set membership in a Merkle tree, or an proof of a path in the Ethereum trie of on-chain state at some point for that public key.

2.4 ZK SNARKs

There are several generic proving systems, including Groth16 which uses R1CS arithmeticization [3], PLONK which uses AIR [13], and Nova which uses relaxed R1CS [21]. We write our proof of concept in circom, which uses r1cs and groth16, although we expect to easily be able to swap out the groth16 proof for a Nova proof to boost performance. Note that we desire to use a statistical zero knowledge argument system over a computational one, to ensure that data remains zero knowledge after 256 qbit quantum supremacy (although the nullifier breaks past zero knowledge, which we comment on later).

We use a ZK-SNARK for easy composability with other algorithms, like a set inclusion algorithm for the public key, or any other existing circuit, along with the ease of verification of the succinct proof on-chain.

Chapter 3

Nullifier Scheme

3.1 Construction

This is a description of how the different entities of the secure enclave chip, browser wallet, client-side-only zk prover, and blockchain play their part in calculating the signatures. We assume the secure enclave is the only entity with access to the secret key sk and secure randomness r , and can be either a hardware enclave or a software enclave. We use exponential notation instead of multiplicative notation so a reader can apply their intuitions about the discrete log problem. In practice, hash_2 is a standard multi-value hash function like SHA256, and hash refers to a hash-to-curve function.

The enclave calculates and outputs publicly (to become a public input into the ZK SNARK):

$$\text{nullifier} = \text{hash}[m, pk]^{sk}$$

$$s = r + sk * c$$

The enclave also calculates and outputs this privately to the client (to become a private input into the ZK SNARK, as they de-anonymize the user):

$$\begin{aligned}
c &= \text{hash}_2(g, g^{sk}, \text{hash}[m, pk], \text{hash}[m, pk]^{sk}, g^r, \text{hash}[m, pk]^r) \\
pk &= g^{sk} \\
g^r &\quad [\text{optional output}] \\
\text{hash}[m, pk]^r &\quad [\text{optional output}]
\end{aligned}$$

The optional inputs are included here to increase prover efficiency via precomputation, but they can be calculated from the other signals. The ZK SNARK, which is generated on a client-side only software that the prover runs, proves:

$$\begin{aligned}
&\leftarrow \text{nullifier}, s, c, pk, g, m \\
c &== \text{hash}_2\left(g, pk, \text{hash}[m, pk], \text{nullifier}, \frac{g^s}{pk^c}, \frac{\text{hash}[m, pk]^s}{\text{nullifier}^c}\right) \\
pk &\in \text{set}
\end{aligned}$$

Note that divisions as written are calculated in practice as subtractions, and that under the random oracle model for hashing, this check is implicitly proving the following (which you can easily verify completeness for by substituting $s = r + sk * c$).

$$\begin{aligned}
\frac{g^s}{pk^c} &= g^s / (g^{sk})^c = g^r \\
\frac{\text{hash}[m, pk]^s}{\text{nullifier}^c} &= \text{hash}[m, pk]^s / (\text{hash}[m, pk]^{sk}) = \text{hash}[m, pk]^r
\end{aligned}$$

We expect that users will include other application-specific checks as well in the same ZK SNARK, such as set inclusion checks for the stated public key (this can be done with a Merkle proof for a public Merkle root, for instance). For this circuit specific ZK SNARK, a verifier with the corresponding prover code will be posted on a Turing-complete public blockchain ledger, to allow anyone to both verify that the proof is accurate, and provide proofs that, upon verification, allow access to some specific on-chain action.

We use a ZK-SNARK instead of a simple ZK interactive protocol with Fiat-Shamir to allow application creators to easily extend the existing proof to fit their use cases.

3.2 Proofs

We prove security in the random oracle model. Specifically, we prove uniqueness (each public key can only generate one nullifier), secrecy (one cannot learn the public key from the public signals), and existential unforgeability (given proof signals, one cannot forge another proof or learn the secret key). Previous work has proved uniqueness, pseudorandomness, and collision resistance of the original signature scheme without the zero knowledge component [27].

3.2.1 Definitions

We define EUF via definitions similar to the BLS paper [5] and Goh-Jarecki's EDL paper [14], but for the secp256k1 curve.

Definition 3.1 (Group Definition) *If the group satisfies τ, t', ϵ' , such that it τ is the ratio of timesteps needed to break DDH vs do the group operation (time = 1), and that an adversary can break CDH with probability ϵ' in t' timesteps. This group has order p .*

We assume for the implementation that this elliptic curve is secp256k1.

Definition 3.2 (Advantage of existentially forging) *This is the probability that an adversary can produce a nullifier and valid zk proof that validates for a known message m . Because this is an app specific, app-chosen parameter, we assume the adversary can choose this message m . We assume the hash function treated as random oracle, and we account for hash collision failure probabilities in our calculation.*

Definition 3.3 (Scheme security) *If an adversary in time t , with q_h queries to hash function, and q_s queries to sig oracle, can achieve an advantage of existentially forgery at most ϵ , then it is (t, q_h, q_s, ϵ) secure.*

Definition 3.4 (Security against Existential Forgery) *No forger can (t, q_h, q_s, ϵ) break a nullifier.*

Theorem 3.5 (Security Bound) *We will prove that $t \leq t' - 2c_A(\lg p)(q_H + q_S)$ and $\epsilon \geq 2e \cdot q_S \epsilon'$, and c_A is a small constant between 1 and 2. Here e is the base of the natural logarithm.*

Theorem 3.6 (Computational Diffie Hellman Security) *A probabilistic algorithm \mathcal{A} is said to (t, ϵ) -break CDH in a group $G_{g,p}$ if on input (g, p, q) and random query (g^a, g^b) , and after running in at most t steps, \mathcal{A} computes g^{ab} with ϵ probability. We say that group $G_{g,p}$ is a (t, ϵ) -CDH group if no algorithm (t, ϵ) -breaks CDH in $G_{g,p}$.*

Note that these definitions are almost identical to BLS's signature security proof [5], and CDH follows because it is merely a reduction of discrete log [24].

Note that there is additionally a proof of set inclusion property: that one can a valid proof iff the public key is in the allowed set. Note that if we prove the uniqueness, secrecy, and existential unforgeability of the signature scheme, then this set inclusion proof immediately follows from the accumulator scheme check being correct, so we omit the formal proof.

3.2.2 Proof of Uniqueness

We prove that an adversarial secret key holder cannot generate more than one valid nullifier. Formally adversary \mathcal{A} breaks uniqueness if:

$$Pr \left(\begin{array}{l} Gen(secp256k1) \rightarrow sk, pk \\ sk, pk, g, m \rightarrow nullifier, s, c \\ c = \text{hash}_2 \left(g, pk, \text{hash}[m, pk], nullifier, \frac{g^s}{pk^c}, \frac{\text{hash}[m, pk]^s}{nullifier^c} \right) \\ \mathcal{A} \leftarrow sk, pk, g, m \\ \mathcal{A} \rightarrow nullifier_{\mathcal{A}}, s_{\mathcal{A}}, c_{\mathcal{A}} \\ nullifier_{\mathcal{A}} \neq nullifier \\ c_{\mathcal{A}} = \text{hash}_2 \left(g, pk, \text{hash}[m, pk], nullifier_{\mathcal{A}}, \frac{g^{s_{\mathcal{A}}}}{pk^{c_{\mathcal{A}}}}, \frac{\text{hash}[m, pk]^{s_{\mathcal{A}}}}{nullifier_{\mathcal{A}}^{c_{\mathcal{A}}}} \right) \end{array} \right) > negl$$

where negl is the negligible function. We can assume all inputs to hash_2 are well formed because we can add those constraints to the ZK-SNARK. We show that in the random oracle model, this probability is negligible.

Proof. Imagine the adversary makes q_h queries to the hash function. For each query, they have to commit to some $(pk, \text{nullifier}_{\mathcal{A}}, \frac{g^{s_{\mathcal{A}}}}{pk^{c_{\mathcal{A}}}}, \frac{\text{hash}[m, pk]^{s_{\mathcal{A}}}}{\text{nullifier}_{\mathcal{A}}^{c_{\mathcal{A}}}})$ ahead of time to find $c_{\mathcal{A}}$. We can assume $pk = g^{sk}$ and $\text{nullifier}_{\mathcal{A}} = \text{hash}[m, pk]^{r'}$ for some r' , since $\text{hash}[m, pk]$ is a generator of a cyclic prime group. Then the adversary has committed to $(g^{sk}, \text{hash}[m, pk]^{r'}, g^{s_{\mathcal{A}}-sk \cdot c_{\mathcal{A}}}, \text{hash}[m, pk]^{s_{\mathcal{A}}-r' \cdot c_{\mathcal{A}}})$ ahead of time. Since they have committed to the exponents in the last two terms, the adversary has committed to an arbitrary $(sk - r')c_{\mathcal{A}}$ before finding out $c_{\mathcal{A}}$. This means that either $sk = r'$ and the discrete logs match so $\text{nullifier}_{\mathcal{A}} = \text{nullifier}$, or else $c_{\mathcal{A}}$ is 0, which happens with negligible probability $1/2^k$.

3.2.3 Proof of Existential Unforgeability

We prove that an adversary \mathcal{A} with a (t, q_h, q_s, ϵ) advantage on breaking existential unforgeability for the signature scheme, can then (τ, ϵ) break CDH [11] in the random oracle model. Note that this proof is essential from the perspective of the secure enclave, to ensure that the data it produces does not leak information about the secret key to ZK-SNARK generator, which might be inside a wallet.

Proof. Imagine Alice wants to break CDH of (g, g^a, g^b) , and Bob is the adversary who can generate a valid $(\text{hash}[m, pk], \text{nullifier}, s, c, m)$ tuple for a previously unseen message m after q_s queries of public and private outputs of the signature oracle (previous $(\text{hash}[m, pk], \text{nullifier}, s, c, m)$ tuples). To do this, Alice generates a random bit $b_i \sim \text{Bernoulli}(p)$ for the \mathcal{A} 's i th query of the signature oracle. This bit will control whether Alice predicts that Bob will try to break CDH on this round or not, and thus Alice will tailor their responses accordingly.

If $b_i = 1$, when Bob queries pk , Alice returns g^b . When Bob queries $\text{hash}[m, pk]$, Alice as the random oracle returns the value g^a . If Bob chooses to break the signature scheme this round and responds, because of the uniqueness proof, they return the nullifier $\text{hash}[m, pk]^{sk} = g^{ab}$ with probability ϵ . Alice can then use this value to break

Diffie Hellman. If Bob queries Alice for the nullifier, this scheme fails.

If $b_i = 0$, when Bob queries pk , Alice returns g^{r_i} , where r_i is uniformly random over the order of the prime field. When Bob queries $\text{hash}[m, pk]$, Alice as the random oracle returns the value g^{q_i} , where q_i is similarly uniformly randomly over the order of the prime field. If Bob queries Alice for the nullifier, Alice returns $g^{r_i \cdot q_i}$. If Bob tries to break the signature scheme by returning a tuple, this scheme fails because Alice gets no information.

Note that Alice needs to correctly guess all of the b_i 's correctly to successfully answer Bob's first q_h queries as well as the final breaking query. Thus, similar to BLS, the probability of breaking the scheme is $p^{q_h}(1 - p)\epsilon$. Maximizing p via derivative shows the max probability is attained at $p = \frac{q_h}{1+q_h}$, putting success probability at $\left(\frac{q_h^{q_h}}{(1+q_h)^{1+q_h}}\right)\epsilon$. For large q_h , using the identity $\lim_{x \rightarrow \infty}(1 - \frac{1}{x})^x = e^{-1}$ gives that the probability is around $\frac{\epsilon}{e \cdot q_h}$ where e is the natural logarithm, and the number of steps is $t' = t + 2c_A(lgp)(q_H + q_S)$, where q_H is the number of hash function queries, q_S is the number of signature queries, lgp is the approximate complexity of an exponentiation in a group of size p and dominates algorithm time per hash/signature, and c is an overhead constant that in practice is probably between 1 and 2.

Thus, if G is a (τ, t', ϵ') -CDH group, then there can exist no algorithm that (t, q_H, q_S, ϵ) -breaks our nullifier signature scheme with $t \leq t' - 2c_A(lgp)(q_H + q_S)$ and $\epsilon \geq 2e \cdot q_S \epsilon'$, and c_A is a small constant between 1 and 2. Note that this proof is almost the same as the BLS proof [5].

3.2.4 Proof of Secrecy

We prove that given only public signals, an adversary who is not the prover cannot learn anything about the original user's identity. That is, adversary \mathcal{A} breaks secrecy if they can distinguish which public key was used to generate the nullifier. Formally,

$$Pr \left(\begin{array}{l} Gen(secp256k1) \rightarrow sk, pk \\ sk, pk, g, m \rightarrow nullifier, s, c \\ c = \text{hash}_2 \left(g, pk, \text{hash}[m, pk], nullifier, \frac{g^s}{pk^c}, \frac{\text{hash}[m, pk]^s}{nullifier^c} \right) \\ \mathcal{A} \leftarrow nullifier, pk, pk_2, g, m \\ \mathcal{A} \rightarrow pk \end{array} \right) > 0.5 + negl$$

where $negl$ is the negligible function, and the adversary is effectively trying to tell if pk or arbitrary, valid pk_2 corresponds to the nullifier. We show that in the random oracle model, we can reduce the security of this to DDH.

Proof. Specifically, imagine an adversary Alice could in fact determine the correct public key with probability $0.5 + \epsilon$, $\epsilon > negl$, using public signals. Then, imagine that Bob wanted to distinguish DDH; i.e. distinguish which of (g, g^a, g^b, g^x) , (g, g^a, g^b, g^y) , for $x \neq ab, y = ab$, is the valid Diffie-Hellman tuple with epsilon advantage. Then Bob can randomly shuffle the tuples and designate the first one as g^x , then pass in $(nullifier = g^x, pk = g^a, pk_2 = r, g, m)$, where $pk_2 \neq pk$ is randomly generated. When Alice queries the hash oracle for $\text{hash}[m, pk]$, Bob can respond with g^a . When Alice queries pk , Bob can respond with g^b . Then, if Alice has an ϵ advantage, they can distinguish $g^a b$ in $1/2$ of the shuffles, so Bob has an $\epsilon/2$ advantage in solving decisional Diffie Hellman.

Chapter 4

Discussion

4.1 Performance

Our proof of concept for the nullifier uses a library called RustCrypto oriented towards constant time for all inputs, making timing-based side-channel attacks much harder [22]. We hope that ports to other languages prioritize this as well. Our code is OSS on GitHub *. It takes 0.35 seconds to compile, run, and verify without a ZK-SNARK.

The gas efficiency of our code is dominated by the Groth16 pairing check right now, but we expect that to be decreased when we switch to Nova.

4.2 Morality and Mitigations

We believe that as cryptographers, we have a moral imperative to release our work responsibly, and consider possible harms from releasing our research. We also deem it necessary to build in warnings and mitigations into such primitives before it is too late [31].

We hope that all uses of the nullifier will come with appropriate disclosures about the loss of anonymity upon ECDSA-breaking quantum supremacy.

*<https://github.com/zk-nullifier-sig/zk-nullifier-sig>

4.3 The Interactivity-Quantum Secrecy Tradeoff

Note that in the far future, once 256 qbit quantum computers can break ECDSA keypair security via breaking discrete log [32], most Ethereum keypairs will be broken, but funds will be safe. People can merely sign messages committing to new quantum-resistant keypairs (or just higher bit keypairs on similar algorithms), and the canonical chain can fork to make such keypairs valid. ZK-SNARKs become forgeable as their soundness guarantees are broken [36], for instance via a similar discrete log calculating toxic waste. However, they still guarantee, for statistically zero knowledge systems such as Groth16, that secret data in past proofs still cannot ever be revealed. In the best case, the blockchain’s guarantees should all continue to be upheld.

However, if people rely on any type of deterministic nullifier like our construction, their anonymity is immediately broken: someone can merely derive the secret keys for the whole anonymity set, calculate all the nullifiers, and see which ones match. This problem will exist for any deterministic nullifier algorithm on ECDSA, since revealing the secret key reveals the only source of “randomness” that guarantees anonymity in a deterministic protocol.

If people want to keep post-quantum secrecy of data, you have to give up at least one of our properties: the easiest one is probably non-interactivity. For example, for the zero knowledge airdrop, each account in the anonymity set publicly signs a commitment to a new semaphore id commitment (effectively address pk publishes $\text{hash}[\text{randomness} \mid \text{external nullifier} \mid \text{pk}]$) [17]. Then to claim, they reveal their external nullifier and ZK prove it came from one of the semaphore ids in the anonymity set. This considerably shrinks the anonymity set to everyone who has opted into a semaphore commitment prior to that account claiming, meaning the first claimer can potentially have anonymity set of size 1. The loss of noninteractivity also means some constructions such as the tornado cash improvement via our present nullifier construction is impossible (see use cases). However, since hashes (as far as we currently know) are still hard with quantum computers, it’s unlikely that people will be able to ever de-anonymize you.

We hope that people will choose the appropriate algorithm for their chosen point on the interactivity-quantum secrecy tradeoff for their application, and hope that including this information helps folks make the right choice for themselves. Folks prioritizing shorter-term secrecy, like DAO voting or short-term confessions of the young, might prioritize this document’s nullifier construction, but whistleblowers or journalists might want to consider the semaphore construction instead. We also note that since it will take around 2321 signal qubits (not accounting for noise) to solve elliptic curve cryptography in 256 bit prime fields [30], we expect it to take several decades for this to become feasible – estimates range from 2050 to 2100, to never due to theoretical noise limitations.

4.4 Future Work

We expect to release a blog post to explain the algorithm intuitively. Finally, we hope that folks will use our open source repository on GitHub (so far with a Rust proof of concept, Javascript to come)[†] to re-implement the scheme in a variety of formats and languages, especially ones compatible with different wallets. Although we considered publishing an IETF RFC for additional legitimacy, the existing RFC for the signature scheme alone (section 5 of [16]) seems sufficient for now.

[†]<https://github.com/zk-nullifier-sig/zk-nullifier-sig/>

Chapter 5

Conclusion

Having identified nullifiers as a key piece of cryptographic infrastructure limiting the use cases for ZK-SNARKs, in this work we explored the future of pseudonymous identity systems via nullifiers.

We demonstrated a construction of a new signature scheme with a public, deterministic nullifier component, and additional private signals that help verify the nullifier without the secret key. The nullifier both uniquely identifies the keypair and keeps the account identity secret. We proved uniqueness, secrecy, and existential unforgeability, and demonstrated a performant proof-of-concept.

We informally explained the quantum secrecy and interactivity tradeoff for nullifiers, and described possible mitigations for moral hazards introduced by this paper.

Bibliography

- [1] Masayuki Abe, Jan Camenisch, Rafael Dowsley, and Maria Dubovitskaya. On the impossibility of structure-preserving deterministic primitives. *J. Cryptology*, 32(1):239–264, January 2019.
- [2] Roman Storm Alexey Pertsev, Roman Semenov. Tornado.cash privacy solution. *tornado.cash*, 2019.
- [3] Karim Baghery, Zaira Pindado, and Carla Ràfols. Simulation extractable versions of groth’s zk-snark revisited. In *International Conference on Cryptology and Network Security*, pages 453–461. Springer, 2020.
- [4] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*. IEEE, May 2014.
- [5] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing, 2001.
- [6] Dan Boneh and Victor Shoup. *A Graduate Course in Applied Cryptography*. <https://toc.cryptobook.us/>.
- [7] Vitalik Buterin. Ethereum whitepaper, 2014.
- [8] Vitalik Buterin. Some ways to use zk-snarks for privacy, Jun 2022.
- [9] cabal.xyz team. cabal.xyz, 2022.
- [10] Alex Davidson, Ian Goldberg, Nick Sullivan, George Tankersley, and Filippo Valsorda. Privacy pass: Bypassing internet challenges anonymously. *Proc. Priv. Enhancing Technol.*, 2018(3):164–180, June 2018.
- [11] Whitfield Diffie and Martin E Hellman. Multiuser cryptographic techniques. In *Proceedings of the June 7-10, 1976, national computer conference and exposition*, pages 109–112, 1976.
- [12] A. Faz-Hernandez, R.S. Wahby, S. Scott, N. Sullivan, and C.A. Wood. Hashing to elliptic curves ietf rfc, Jun 2022.

- [13] Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *Cryptology ePrint Archive*, 2019.
- [14] Eu-Jin Goh and Stanisław Jarecki. A signature scheme as secure as the diffie-hellman problem. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 401–415. Springer, 2003.
- [15] Sharon Goldberg, Dimitrios Papadopoulos, and Jan Včelák. Verifiable Random Functions (VRFs). Internet-Draft draft-goldbe-vrf-01, Internet Engineering Task Force, June 2017. Work in Progress.
- [16] Sharon Goldberg, Leonid Reyzin, Dimitrios Papadopoulos, and Jan Včelák. Verifiable Random Functions (VRFs). Internet-Draft draft-irtf-cfrg-vrf-15, Internet Engineering Task Force, August 2022. Work in Progress.
- [17] Kobi Gurkan, Koh Wei Jie, and Barry Whitehat. Community proposal: Semaphore: Zero-knowledge signaling on ethereum. *Accessed: Jul, 1:2021*, 2020.
- [18] Kobi Gurkan, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. Aggregatable distributed key generation. In *Lecture Notes in Computer Science, Lecture notes in computer science*, pages 147–176. Springer International Publishing, Cham, 2021.
- [19] heyanon team. heyanon, 2022.
- [20] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International journal of information security*, 1(1):36–63, 2001.
- [21] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. *Cryptology ePrint Archive*, 2021.
- [22] Nate Lawson. Side-channel attacks on cryptographic software. *IEEE Security & Privacy*, 7(6):65–68, 2009.
- [23] Matt Lepinski, Silvio Micali, and Abhi Shelat. Fair-Zero knowledge. In *Theory of Cryptography, Lecture notes in computer science*, pages 245–263. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.
- [24] Ueli M Maurer and Stefan Wolf. The relationship between breaking the diffie-hellman protocol and computing discrete logarithms. *SIAM Journal on Computing*, 28(5):1689–1721, 1999.
- [25] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.
- [26] Adhyyan Sekhsaria Nalin Bhardwaj, Aayush Gupta. Stealthdrop, 2022.

- [27] Dimitrios Papadopoulos, Duane Wessels, Shumon Huque, Moni Naor, Jan Včelák, Leonid Reyzin, and Sharon Goldberg. Making nsec5 practical for dnssec. *Cryptology ePrint Archive*, 2017.
- [28] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.
- [29] Ronald L Rivest, Adi Shamir, and Yael Tauman. How to leak a secret. In *International conference on the theory and application of cryptology and information security*, pages 552–565. Springer, 2001.
- [30] Martin Roetteler, Michael Naehrig, Krysta M Svore, and Kristin Lauter. Quantum resource estimates for computing elliptic curve discrete logarithms. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 241–270. Springer, 2017.
- [31] Phillip Rogaway. The moral character of cryptographic work. *Cryptology ePrint Archive*, 2015.
- [32] Meryem Cherkaoui Semmouni, Abderrahmane Nitaj, and Mostafa Belkasmi. Bitcoin security with post quantum cryptography. In *International Conference on Networked Systems*, pages 281–288. Springer, 2019.
- [33] Mehdi Tibouchi. A note on hashing to bn curves. *SCIS. IEICE*, 40, 2012.
- [34] Sergei Volotikin. Software attacks on hardware wallets. *Black Hat USA*, 2018.
- [35] Riad S Wahby, Dan Boneh, Christopher Jeffrey, and Joseph Poon. An airdrop that preserves recipient privacy. In *Financial Cryptography and Data Security*, Lecture notes in computer science, pages 444–463. Springer International Publishing, Cham, 2020.
- [36] Yu Yu and Xiang Xie. Privacy-preserving computation in the post-quantum era. *National Science Review*, 8(9):nwab115, 2021.

Mental Poker in the Age of SNARKs (Part 1)

Nicolas Mohnblatt



Mental Poker in the Age of SNARKs - Part 1

Written by Nicolas Mohnblatt. Work produced in collaboration with Andrija Novakovic and Kobi Gurkan for Geometry and MatchboxDAO.

Is it possible to play a classic card game without physical cards, without trusting the participants and without a trusted third party?

This problem was first formulated and given the name “Mental Poker” in 1979 by none other than Rivest, Shamir and Adleman. Below is an extract from the original paper [1] in which they explain the notion of *Mental Poker*.

I. What does it mean to play "Mental Poker"?

The game of “Mental Poker” is played just like ordinary poker (see “Hoyle”[2]) except that there are no cards: *all communications between the players must be accomplished using messages*. It may perhaps make the ground rules clearer if we imagine two players, Bob and Alice, who want to play poker over the telephone. Since it is impossible to send playing cards over a phone line, the entire game (including the deal) must be realized using only spoken (or digitally transmitted) messages between the two players.

We assume that neither player is above cheating. “Having an ace up one’s sleeve” might be easy if the aces don’t really exist! A fair method of playing Mental Poker should preclude any sort of cheating.

Since then, many solutions have been published but none were deemed practical. These protocols incurred computation and communication costs that were too large for a game to be played in real-time. Today with the advent of decentralised ledgers, zero-knowledge rollups and efficient SNARKs, we believe that it is time to revisit *Mental Poker* once again.

In this first of two articles we will cover existing solutions and our improvements towards making such protocols practical. We look at *Mental Poker* from a high level with an emphasis on understanding the cryptographic primitives at work and how they interact. In Part 2 we will open Pandora’s box and closely inspect the mathematics and cryptography.

We implement the protocol described here in a Rust library as part of the collaboration between Geometry and [MatchboxDAO](#) (see [source code](#)). Our naive implementation for a 52-card deck allows a player to produce a proof of correct shuffle (the most expensive operation) in just over 50ms on a standard laptop, with verification requiring less than 1ms. Note however that a game is only guaranteed to be fair after all players have taken turns performing this shuffle operation.

Problem Overview

The main challenges in creating a protocol for *Mental Poker* are:

1. hide the card values from all players
2. reveal a card to a specific player or group of players
3. ensure that the cards are dealt correctly, with fair randomness

Some additional desirable properties include the ability to keep some information secret after the game ends - effectively allowing for bluff strategies - and the possibility to add or remove players to a game.

To these requirements we must add performance considerations. How much data do we need to store for each card? How many messages does each player need to send/receive? What kind of computations do players need to run?

Geometry revisits Mental Poker

Related Work

The most noteworthy solutions in the literature are *A Toolbox for Mental Card Games* (Schindelhauer, 1998) [2], its implementation in a C library (Stamer, 2005) [3] and a revised approach presented in *Mental Poker Revisited* (Barnett and Smart, 2003) [4]. While these protocols work on paper they were deemed impractical due to large computation or communication costs. Other approaches focus on the game of poker specifically (as opposed to general card games) [5], introduce a partly trusted party [5] or require trusted execution environments [6].

We choose to focus on the protocol by Barnett and Smart for multiple reasons. Firstly, the security of their protocol relies on the well known Decisional Diffie-Hellman

assumption. Secondly, the protocol is built on top of clear and logical abstractions which allow for easy “plug-and-play” improvements. Finally the computation and communication costs of their protocol are dominated by zero-knowledge proofs: an area in which we have seen great performance improvements since the paper was published in 2003.

The Improved Barnett-Smart Protocol

The Barnett-Smart protocol relies on two cryptographic abstractions. Operations on cards are performed using a set of *verifiable l -out-of- l threshold masking functions*. Quite a mouthful - we will explain these in the following section. The second abstraction allows to perform operations on a deck and is known as a *zero-knowledge proof of a correct shuffle*. Also a mouthful but a less cryptic one.

The masking functions described in the original paper make use of a threshold variant of the well-known El Gamal encryption scheme. This scheme can be instantiated over elliptic curves to provide high security while maintaining a compact representation for cards (each card is represented by two elliptic curve points). On the other hand, the shuffle proof presented by Barnett and Smart did not withstand the test of time. Today much cheaper and quicker schemes are available. **Our improvement lies in replacing the original zero-knowledge proof by a zero-knowledge argument** - more on this distinction later.

Verifiable l -out-of- l Threshold Masking Functions

Verifiable l -out-of- l threshold masking functions (VTMFs) were introduced by Barnett and Smart to abstract away from specific implementation details. We will only focus on what they describe as a “discrete-log VTMF”. First let us break down the name into digestible terms:

- **masking**: these functions allow us to “mask” or hide a piece of data. In this sense, encryption is a form of masking.
- **verifiable**: an observer should have the ability to check that a masking function was applied correctly, even if the function requires some secret information to be ran.
- **l -out-of- l threshold**: some of these functions will require a given threshold of participants to be ran. In this case the threshold is set to the number of

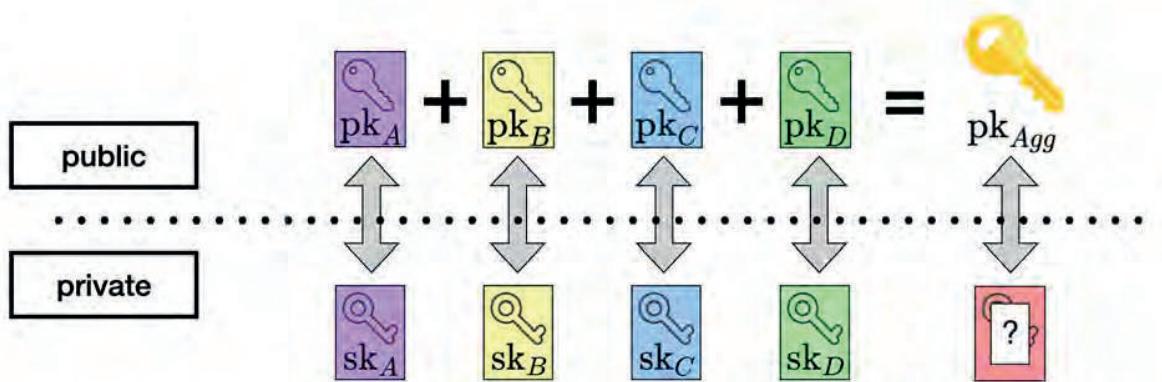
participants (l -out-of- l), effectively requiring that *all players* collaborate.

Combining these three properties, VTMFs will allow us to hide card values (masking) while guaranteeing that no individual player is cheating (verifiable) nor can a coalition of players cheat (l -out-of- l threshold). The functions are: **key generation**, **mask**, **remask** and **unmask**. We cover each of these individually.

Key Generation

Each player is expected to run a key generation algorithm which outputs a secret key and a public key. For player A , we denote the secret key sk_A and the public key pk_A . Each player publishes their public key along with a zero-knowledge proof that they know the corresponding secret key. This proof is necessary to prevent denial-of-service attacks and rogue key attacks.

Once each player has completed the above process, an *aggregate public key*, pk_{Agg} , can be computed. While there exists in theory an aggregate secret key that corresponds to pk_{Agg} , this secret key will remain unknown to *all parties* as long as at least one player keeps their secret key hidden. The figure below illustrates this key aggregation process.



An illustration of a 4-player key aggregation. Importantly, no one can derive the secret key that corresponds to the aggregate public key.

Mask

A *mask* operation is an encryption. A plaintext card is processed along with the aggregate public key and some randomness to produce a ciphertext: the masked card. To recover the plaintext card, one must decrypt (*unmask*) the masked card. As we will see, this process can only be performed by a coalition of all players.

A good mental model is to picture a masked card as an opaque box with a set of padlocks on it. Each padlock can be opened by one of the players' secret key. Notice however that each ciphertext is unique: this is akin to having a label on the box. The figure below illustrates this mental model.



A mask operation can be understood as placing a card in an opaque box with padlocks for each player and a label.

Masking alone does not guarantee a fair game. Indeed the player who performed the mask operation will be able to maintain a mapping from card value to box label. In the above example the ace of clubs maps to the label "A". To break this mapping, we introduce the *remask* operation.

Remask

A *remask* operation takes a masked card (ciphertext) and produces a new masked card for the same underlying card value (plaintext). This operation does not require to decrypt the ciphertext and therefore can be performed by any player. To follow our box analogy, a remask operation just replaces the label on a box.

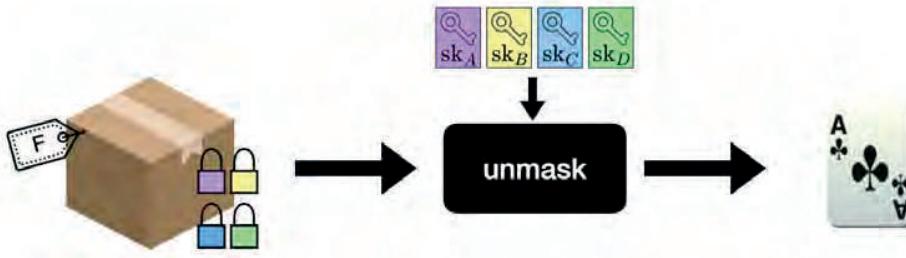


A remask operation can be understood as re-labelling a box (masked card).

Unmask

The *unmask* operation is a threshold decryption. Each player must use their secret key to contribute to the decryption. Crucially, this group decryption can be performed in stages. This allows to give out cards to specific players. For example players A , B and C can perform their part of the unmasking process such that the resulting ciphertext can be decrypted by player D alone. Player D can now decrypt the card in private and will be the only one to know its value.

To complete our box analogy, unmasking allows each player to remove their padlock from a box.

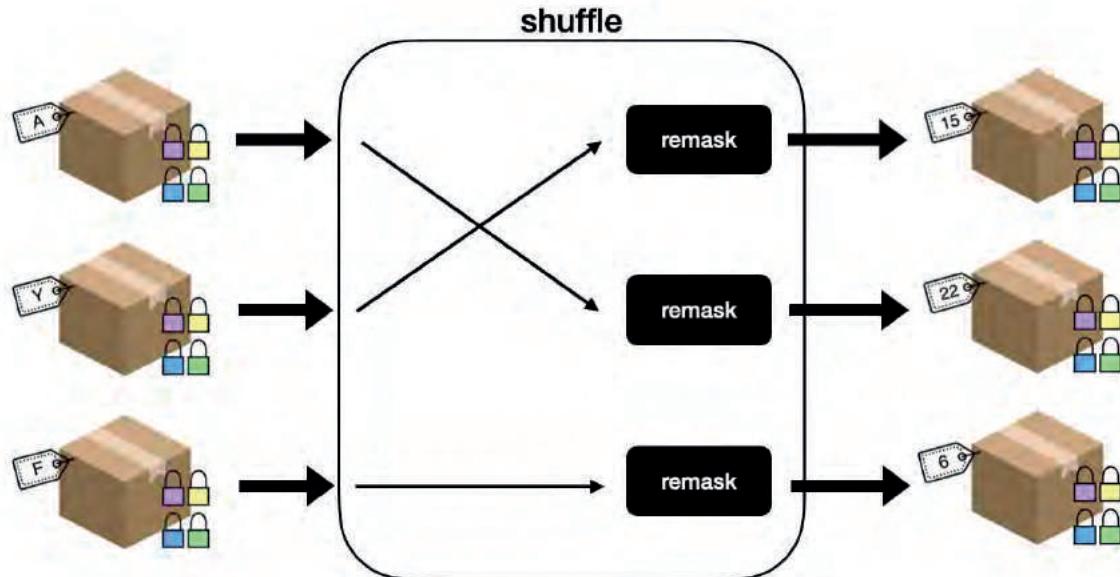


An unmask operation requires every player to use their secret key to remove their padlock.

Zero-knowledge Argument of a Correct Shuffle

In this model, a shuffle is obtained by changing the order of a deck of masked cards and remasking them. Indeed recall that each masked card is uniquely identifiable, as opposed to a physical card that has been flipped. Simply changing the order of masked

cards would not hide any information. Performing an additional remask allows to truly “lose” the cards in the deck.



An illustration of a shuffle: we apply a permutation to the input masked cards and remask them.

The original protocol from Barnett and Smart produces a zero-knowledge proof to attest that this operation was computed correctly. This proof is expensive to compute as it requires the prover to perform a large number of shuffles, and therefore a large number of *remask* operations. Instead we replace the proof by an *argument of knowledge*.

An argument is cheaper to compute but yields slightly weaker security. While a valid proof can never be forged by a malicious prover, a valid argument can be forged by a sufficiently powerful adversary. Therefore, an argument is only meaningful under the assumption that the adversary is *computationally bounded*. Notice however that other cryptographic primitives such as public key encryption also share this property.

Today, there exists an incredible variety of arguments of knowledge that could be used to prove a correct shuffle: universal SNARKs (e.g. Marlin, PlonK), circuit-specific SNARKs (Groth16) and inner-product arguments (Bulletproofs, Bayer-Groth) to list a few. All of these are valid choices. We differentiate them based on the operations they require, any potential setups, the size of an argument, and running time for the prover and verifier.

The setting for *Mental Poker* is very different to the usual SNARK setting. First notice that we are dealing with a small number of participants that will all be online at the same time. In this case, the infamous trusted setup becomes a respectable option. Secondly, our input sizes are relatively small. A classic deck of cards is composed of 52 cards; other games rarely use more than a few hundred. In these ranges, computations that run in $\mathcal{O}(\sqrt{n})$ steps and $\mathcal{O}(\log(n))$ steps are relatively similar. The same reasoning applies for proof sizes.

To build our proof-of-concept code, we chose to prioritise arguments with no special requirements for the underlying curve (no need for pairings or FFTs) and with simple implementations. Based on these criteria we used the argument presented by Bayer and Groth in *Efficient Zero-Knowledge Argument for Correctness of a Shuffle* (2014) [7]. We stress however that any of the options mentioned above would yield a functioning and efficient protocol.

An example round of Texas Hold'Em

The table below presents an example round of Texas Hold'Em, showing on the left the actions that players would take in the classic (physical) version of the game, and on the right the digital equivalents.

Setup

	Texas Hold'Em	Mental Texas Hold'Em
1.	Gather a group of players around a table	Players each run the key generation algorithm, publish their public key and prove in zk that they own the corresponding secret key. A master public key is computed.
2.	Bring a deck of 52 standard cards.	Publicly encode each card value to a card (plaintext). Mask all 52 cards with a public randomness value and publish all proofs.

Round n

	Texas Hold'Em	Mental Texas Hold'Em
3.	Dealer shuffles the deck	All players take turns running a shuffle and provide a zero-knowledge proof

4.	Give out the cards: top card goes to the first player, second card goes to second, etc...	Take part in the unmask process to allow players to see their cards: all players but the first publish a will partially unmask the first card, all players but the second partially unmask the second card, etc...
5.	Betting using physical chips	Betting using digital chips
6.	Discard and reveal cards using the game rules	Following the order of the deck, ignore discarded cards. To open a card, all players collaborate to perform a full unmask of the chosen card
7.	Repeat 5 and 6 as mandated by the rules	Repeat 5 and 6 as mandated by the rules
8.	Final betting. Open hands and declare a winner	Final betting. Each player can complete the unmask process for their private cards. Compare hand values and declare a winner.

Benefits of On-chain Coordination

One final aspect in which we can improve the existing protocols is to make use of decentralised ledgers and smart contracts running on top of them. The Barnett-Smart protocol already requires “a broadcast channel between all players”. A smart contract can be seen as an extension of such a broadcast channel where the channel itself can perform some computation.

In this context, we can delegate all the proof verifications to the smart contract: rather than having every player run a verifier for every proof, we only need one verification of each proof. The code for the smart contract is public and can therefore be audited by all players if they wish so. A particularly distrustful player can still choose to verify the proofs themselves.

Furthermore, a multi-round betting game like poker benefits greatly from the payment security offered by decentralised ledgers. Research such as [this paper](#) by Kumaresan *et al.* consider incentive and slashing mechanisms to force players to behave honestly.

To be continued...

In this first article we have covered the Barnett-Smart protocol for Mental Poker as well as our improvement to make the protocol practical. In the [Part 2](#) we show how and why

the cryptography works.

References

1. A. Shamir, R. Rivest, and L. Adleman, "Mental Poker", Technical Memo LCS/TM-125, Massachusetts Institute of Technology, April 1979. <https://apps.dtic.mil/dtic/tr/fulltext/u2/a066331.pdf>
2. Schindelhauer, C. A Toolbox for Mental Card Games. Tech. Rep. of Medizinische Universitat Lubeck.
3. Stamer, H. Efficient Electronic Gambling: An Extended Implementation of the Toolbox for Mental Card Games. WEWoRC 2005, LN P-74, 1-12, 2005
4. Barnett, Adam, and Nigel P. Smart. "Mental poker revisited." In *IMA International Conference on Cryptography and Coding*, pp. 370-383. Springer, Berlin, Heidelberg, 2003.
5. Golle, P. Dealing Cards in Poker Games. In Proceedings of the International Conference on Information Technology: Coding and Computing, (2005)
6. Castellà-Roca, J., Domingo-Ferrer, J., Sebé, F. (2006). A Smart Card-Based Mental Poker System. In: Domingo-Ferrer, J., Posegga, J., Schreckling, D. (eds) Smart Card Research and Advanced Applications. CARDIS 2006. Lecture Notes in Computer Science, vol 3928. Springer, Berlin, Heidelberg.
https://doi.org/10.1007/11733447_4
7. Bayer, Stephanie, and Jens Groth. "Efficient zero-knowledge argument for correctness of a shuffle." In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 263-280. Springer, Berlin, Heidelberg, 2012

Mental Poker in the Age of SNARKs (Part 2)

Nicolas Mohnblatt



Mental Poker in the Age of SNARKs - Part 2

Written by Nicolas Mohnblatt. Work produced in collaboration with Andrija Novakovic and Kobi Gurkan for Geometry and MatchboxDAO.

In Part 1 of our Mental Poker series we highlighted the challenges of decentralised card games and presented the improved Barnett-Smart protocol as a practical solution. In this second part we take a deep dive into the cryptography that enables this protocol.

We will first inspect an instantiation of the verifiable l -out-of- l threshold masking functions (VTMFs) using El Gamal encryption [1]. We will also show how repeated applications of these functions achieve the necessary properties for a fair card game. In the second section we cover the zero-knowledge argument for a correct shuffle as presented in “Efficient Zero-knowledge Argument for Correctness of a Shuffle” [2].

Discrete-log VTMF [1] (Mental Poker Revisited)

A set of VTMFs is composed of 4 functions: **key generation**, **mask**, **remask** and **unmask**. These can be instantiated using a threshold variant of the El Gamal encryption scheme. Verifiability is achieved by computing zero-knowledge proofs that encryption and decryption are performed correctly.



Notation:

We assume that players have agreed to use a finite abelian group \mathbb{G} of prime order q . We denote G the generator of \mathbb{G} and use additive notation (" $+$ ") for the group operation.

We assume the existence of a public mapping \mathcal{M} from card values to elements of \mathbb{G} ; this could be a hash function for example. Given a card value, all players can compute $M = \mathcal{M}(\text{card})$ with $M \in \mathbb{G}$.

The $\xleftarrow{\$}$ operator denotes an element chosen uniformly at random from a set, e.g. $X \xleftarrow{\$} \mathbb{G}$

In this section we will consider a group of l players which we denote with the subscript $i \in \{1, 2, \dots, l\}$

Key Generation

Player i generates a private key $x_i \xleftarrow{\$} \mathbb{Z}_q$. The corresponding public key H_i is computed as $H_i = x_i G$ and published.

To prevent rogue key attacks, we require that players prove in zero knowledge that they know the private key corresponding to the key they publish. In this discrete logarithm setting, such a proof is easily obtained using the [Schnorr identification scheme](#).

Once all players have published their keys and proofs, an aggregate public key H can be computed as:

$$H = \sum_{i=1}^l H_i = \sum_{i=1}^l x_i G \tag{1}$$

Mask

The mask function, denoted \mathcal{E}_H when using the public key H , is in fact the El Gamal encryption of a card $M \in \mathbb{G}$. The resulting ciphertext $\mathbf{C} \in \mathbb{G} \times \mathbb{G}$ is a masked card.

$$\mathbf{C} = \mathcal{E}_H(M, r) = \begin{pmatrix} C_a \\ C_b \end{pmatrix} = \begin{pmatrix} rG \\ M + rH \end{pmatrix} \quad (2)$$

Here r , known as the *masking factor*, is an element of \mathbb{Z}_q chosen uniformly at random by the player performing the mask operation. Notice that given a masked card and its corresponding masking factor, the plaintext card can be recovered without a secret key by computing $C_b - rH$. This would spell disaster for our scheme. As a consequence, masking factors must remain secret to protect a masked card's hidden value.

Remask

The remask function, denoted \mathcal{E}_H' , allows to re-randomise a masked card $\mathbf{C} = \begin{pmatrix} C_a \\ C_b \end{pmatrix}$. The player performing the remask operation chooses a random element $r' \xleftarrow{\$} \mathbb{Z}_q$ and computes:

$$\mathbf{C}' = \mathcal{E}_H'(\mathbf{C}, r') = \mathbf{C} + \begin{pmatrix} r'G \\ r'H \end{pmatrix} = \begin{pmatrix} C_a + r'G \\ C_b + r'H \end{pmatrix} \quad (3)$$

Example: using Mask and Remask to hide a card from all players

Let's take a three player game as an example with Alice, Bob and Charlie respectively players 1, 2 and 3. Alice computes \mathbf{C}_1 , the masking of card M using the masking factor α . Bob then computes \mathbf{C}_2 by remasking \mathbf{C}_1 using the masking factor β . Finally Charlie computes \mathbf{C}_3 by remasking \mathbf{C}_2 with the masking factor γ . Let us quickly run through the math to see what is actually happening:

$$\mathbf{C}_1 = \mathcal{E}_H(M, \alpha) = \begin{pmatrix} \alpha G \\ M + \alpha H \end{pmatrix} \quad (4)$$

$$\mathbf{C}_2 = \mathcal{E}_H'(\mathbf{C}_1, \beta) = \mathbf{C}_1 + \begin{pmatrix} \beta G \\ \beta H \end{pmatrix} = \begin{pmatrix} (\alpha + \beta)G \\ M + (\alpha + \beta)H \end{pmatrix} \quad (5)$$

$$\mathbf{C}_3 = \mathcal{E}_H'(\mathbf{C}_2, \gamma) = \mathbf{C}_2 + \begin{pmatrix} \gamma G \\ \gamma H \end{pmatrix} = \begin{pmatrix} (\alpha + \beta + \gamma)G \\ M + (\alpha + \beta + \gamma)H \end{pmatrix} \quad (6)$$

After every player has processed the card we are left with a masked card under the aggregate masking factor $\alpha + \beta + \gamma$: indeed notice that $\mathbf{C}_3 = \mathcal{E}_H(M, (\alpha + \beta + \gamma))$. Recovering the original card M would require all players to share the masking factor they used or to participate in decryption.

Unmask

How can we untie the knot that was created above? We make use of the fact that a masked card (ciphertext) is composed of two group elements and each of these have had the same masking factors applied to them. Given a masked card $\mathbf{C} = \begin{pmatrix} C_a \\ C_b \end{pmatrix}$ the i -th player can compute the value $D_i = x_i C_a$ using their private key. After each player publishes their value, the unmask operation can be performed as:

$$M = C_b - \sum_{i=1}^l D_i \tag{7}$$

For those who want to see why this works:

$$C_b - \sum_{i=1}^l D_i = C_b - \sum_{i=1}^l x_i C_a \tag{8}$$

$$= (M + rH) - \sum_{i=1}^l x_i (rG) \tag{9}$$

$$= M + rH - r \sum_{i=1}^l H_i \tag{10}$$

$$= M \tag{11}$$

Verifiability

Verifiability for the operations above (bar the key generation) is obtained by generating **Chaum-Pedersen proofs of discrete log equality**. In this classic Σ -protocol, a prover produces two values αG and βH and proves in zero knowledge that they know α and that $\alpha = \beta$. A mask operation is verified using a Chaum-Pedersen proof for the values C_a and $C_b - M$ with $\alpha = r$, remask using a Chaum-Pedersen proof for $C'_a - C_a$ and

$C'_b - C_b$ with $\alpha = r'$ and finally decrypt using a Chaum-Pedersen proof for G and C_a using $\alpha = x_i$.

Zero-Knowledge Argument for Correctness of a Shuffle [2] (Bayer-Groth)



Notation:

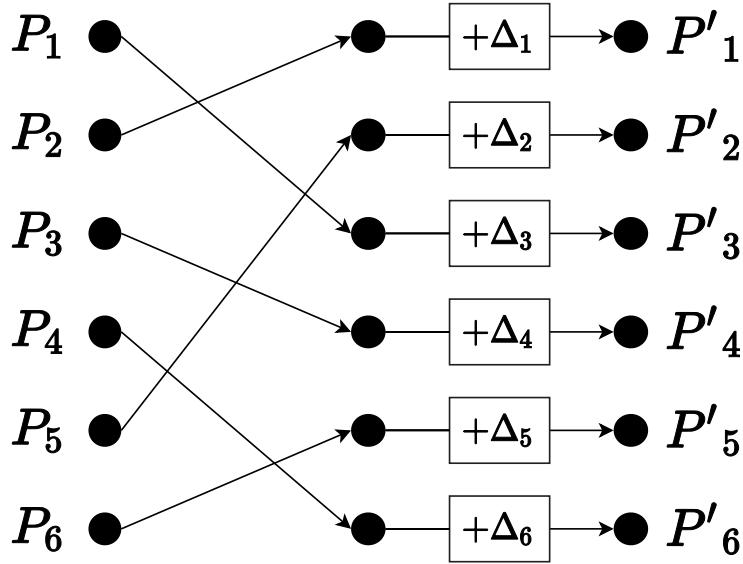
As we now consider actions on a deck of cards by a single player, we use subscripts to denote an item's position in an array (rather than indicating a player as seen in the previous section).

Preliminaries

To understand the argument of a correct shuffle of ciphertexts, we will first look at shuffling a set of N group elements P_1, P_2, \dots, P_N , each a member of \mathbb{G} . Each element P_i is given a new index and some blinding $\Delta_i \xleftarrow{\$} \mathbb{G}$ is added. This blinding operation is the same as applying remask to one of the two elements in a masked card. We can express each point at the output of the shuffle P'_i as:

$$P'_i = P_{\pi(i)} + \Delta_i \quad (12)$$

where π is a secret permutation. Below is a visual representation of the process for 6 elements. Can you write out the permutation that satisfies [Equation \(12\)](#)? (Answer in the image's caption)



An example of a 6 point shuffle using the permutation π defined as $\pi(1) = 2$, $\pi(2) = 5$, $\pi(3) = 1$, $\pi(4) = 3$, $\pi(5) = 6$, $\pi(6) = 4$. The permutation is read “backwards” on the diagram since equation 1 is indexed with respect to the output.

Argument Overview

In order to produce an efficient argument, we first express a correct shuffle in the form of a polynomial equality. Such equalities can be checked with an overwhelming degree of certainty by evaluating each polynomial at a randomly chosen point in the domain (see [Schwartz-Zippel lemma](#)). One of the polynomials will be computed by the verifier using public data, the other by the prover using secret data. Since the prover cannot be trusted, we must devise additional arguments to attest that the prover polynomial is computed correctly. We consider each of these steps separately in the sections below.

Shuffle as a Polynomial Equality

The core of the argument is the final verifier equation which we show below. For a fixed random challenge $z_c \xleftarrow{\$} \mathbb{Z}_q \setminus \{0, 1\}$, the verifier checks that:

$$\sum_{i=1}^N z_c^i P_i = \sum_{i=1}^N z_c^{\pi(i)} (P'_i - \Delta_i) \quad (13)$$

The **left-hand side** of this equation uses only public data (the shuffle's inputs) and our random scalar; it can therefore be computed by the **verifier**. The **right-hand side** however can only be evaluated by the **prover** as it requires knowledge of the secret permutation π and blinding elements Δ_i .

The first thing we should notice is that these expressions are very similar to the evaluation of two polynomials at the input z_c . The underlying polynomials can be made apparent by rewriting our elements P_i , P'_i and Δ_i as the product of some scalar and a generator. For all i in $\{1, 2, \dots, N\}$, let

- $s_i \in \mathbb{Z}_q$ such that $P_i = s_i G$
- $s'_i \in \mathbb{Z}_q$ such that $P'_i = s'_i G$
- $\delta_i \in \mathbb{Z}_q$ such that $\Delta_i = \delta_i G$

Substituting the above into [Equation \(13\)](#) and factoring G out of the sum we can rewrite the verification equation as:

$$\left(\sum_{i=1}^N z_c^i s_i \right) G = \left(\sum_{i=1}^N z_c^{\pi(i)} (s'_i - \delta_i) \right) G \quad (14)$$

Consider the polynomials f and g described below and notice that the above expressions correspond to these polynomials evaluated at z_c and multiplied by our group generator:

$$f(z) = \sum_{i=1}^N z^i s_i \quad \text{and} \quad g(z) = \sum_{i=1}^N z^{\pi(i)} (s'_i - \delta_i) \quad (15)$$

Given that sums are commutative, we can rewrite f as $f(z) = \sum_{i=1}^N z^{\pi(i)} s_{\pi(i)}$: we simply change the order of the terms in the sum. This now makes it apparent that for values of z that exclude 0 and 1:

$$f(z) = g(z) \iff \sum_{i=1}^N z^{\pi(i)} s_{\pi(i)} = \sum_{i=1}^N z^{\pi(i)} (s'_i - \delta_i) \quad (16)$$

$$\iff \forall i \in \{1, 2, \dots, N\}, s_{\pi(i)} = s'_i - \delta_i \quad (17)$$

$$\iff \forall i \in \{1, 2, \dots, N\}, P_{\pi(i)} = P'_i - \Delta_i \quad (18)$$

Or put into words, **the polynomials f and g are identical if and only if the shuffle is valid.**

Checking the Polynomial Equality

To check the polynomial equality, we evaluate both polynomials at a point $z_c \xleftarrow{\$} \mathbb{Z}_q \setminus \{0, 1\}$ and check that $f(z_c) = g(z_c)$. While this does not guarantee that the polynomials are identical, it allows us to assert it with probability $1 - \frac{N}{|\mathbb{Z}_q|}$ (see [Schwartz-Zippel lemma](#)), where $|\mathbb{Z}_q| >> N$.

Notice also that we do not have access to the values $s_{\pi(i)}$, s'_i and δ_i but instead can only use $P_{\pi(i)}$, P'_i and Δ_i . Therefore we must evaluate the polynomials “in the exponent” as made explicit by [Equation \(14\)](#).

Ensuring the Prover Behaves Honestly

As mentioned previously, the right-hand side of the verification equation, $\sum_{i=1}^N z_c^{\pi(i)}(P'_i - \Delta_i)$, must be computed by the untrusted prover without revealing any private information. We reconcile both these requirements with commitments and zero-knowledge arguments.

First the prover will commit to a permutation and will show in zero knowledge that they permuted powers of the challenge z_c according to the committed permutation (permutation argument). After running such an argument, the verifier will hold a commitment $C_{z,\pi}$ which binds the prover to the vector $[z_c^{\pi(1)}, z_c^{\pi(2)}, \dots, z_c^{\pi(N)}]$. This vector can now be used in an inner-product argument to produce a provable evaluation of the expected sum: $\sum_{i=1}^N z_c^{\pi(i)}(P'_i - \Delta_i)$.

These arguments are not trivial and probably require their own articles to present a solid intuitive and mathematical understanding. A potential extension of the series? Who knows... In the meantime, a bit of ZK trivia: both the inner-product argument and permutation argument presented in “Efficient Zero-knowledge Argument for Correctness of a Shuffle” [2] were later adapted and improved to respectively form part of Bulletproofs and PlonK.

From Shuffling Group Elements to Shuffling Cards

Recall that all the above only concerns shuffling a single group element rather than a masked card (a ciphertext composed of two group elements). We can however extend

the above reasoning to masked cards in either of two ways:

1. Run the argument for elements twice and request that the prover use the same committed permutation for both sets of group elements. This is naturally feasible with the existing argument structure as the prover is already required to commit to a permutation and argue that it was used correctly.
2. Consider each masked card as an element of the group $\mathbb{H} = \mathbb{G} \times \mathbb{G}$ and define the group operation for \mathbb{H} to remain consistent with how we add ciphertexts in our remask formula.

Implementation

All the protocols described in this article were implemented in an [open-source Rust library](#) by Andrija Novakovic, Kobi Gurkan and yours truly as part of the collaboration between Geometry and [MatchboxDAO](#). The implementation relies on the [arkworks](#) ecosystem and is built to be highly modular. Indeed the arguments can be replaced in a “plug-and-play” manner while the protocols are implemented for any generic curve. As you might have noticed, these protocols do not require special operations such as pairings or FFTs and can therefore be run on any elliptic curve.

We benchmarked the library on a [STARK-friendly curve](#) (used by StarkWare), running on consumer hardware. A shuffle argument for 52 cards, complete with the remasking of the cards, takes about 50ms. Verification of a shuffle can be done in less than 1ms. Note however that a game is only guaranteed to be fair after all players have taken turns performing this shuffle operation.

Conclusion

In this article we have reviewed the principal results of *Mental Poker Revisited* [1] and *Efficient Zero-knowledge Argument for Correctness of a Shuffle* [2]. While we have not covered all the mechanics of the shuffle argument, we have presented the reader with a global overview of the protocol’s steps and justification as to why the protocol correctly verifies a shuffle.

Combining both results yields the efficient protocol for card games that we described in Part 1 of our Mental Poker series. Our Rust implementation of the protocol in

collaboration with [MatchboxDAO](#) is available [here](#); decentralised games will hopefully be coming soon!

References

1. Barnett, Adam, and Nigel P. Smart. "Mental poker revisited." In *IMA International Conference on Cryptography and Coding*, pp. 370-383. Springer, Berlin, Heidelberg, 2003.
2. Bayer, Stephanie, and Jens Groth. "Efficient Zero-knowledge Argument for Correctness of a Shuffle." In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 263-280. Springer, Berlin, Heidelberg, 2012

Groth16 Malleability

Andrija Novakovic



Groth16 malleability

Andrija Novakovic, Kobi Gurkan

Table of Contents

- [Groth16 malleability](#)
 - [Table of Contents](#)
 - [Prologue](#)
 - [Background](#)
 - [R1CS and QAP](#)
 - [Attacks](#)
 - [Groth16](#)
 - [Attack 1](#)
 - [Attack 2](#)
 - [Solution in practice](#)
 - [Attack implementation](#)
 - [Conclusion](#)
 - [Acknowledgements](#)

Prologue

Our story begins in the realms of the Groth16 SNARKs, where variables roam free and their freedom only limited by quadratic constraints. It is of great importance that once the prover weaves the witness into a proof, the public inputs can not be changed. The elder from your village notices you writing a new constraint system, and asks with a grave voice: "Child, have you remembered the artificial square constraint?"

```
// Dummy square to prevent tampering signalHash.  
signal signalHashSquared;  
signalHashSquared <= signalHash * signalHash;
```

A popular urban legend in cryptosecurity is that there's a magic spell to defend Groth16 proofs from attack - just add into your circuit an artificial square constraint. This mathematical amulet takes the form of a public input that's not used anywhere in the circuit. Its purpose: to attach a piece of data to the proof.

Let's explore the origin of this somewhat superstitious belief. We will additionally cover:

- the types of attacks from which Groth16 is at risk, and
- why, in practice, these attacks are prevented in popular Groth16 implementations

Background

Groth16 is proven to have the property of *knowledge soundness*. In short, this means that given a valid proof for a set of public inputs (the *instance*), there exists a corresponding set of private inputs (the *witness*) such that the circuit (the *relation*) is satisfied. The relation is, more precisely, described as a *constraint system*, a set of equations that have to be satisfied in order for the proof to be valid.

In some situations, like in [Semaphore](https://github.com/appliedzkp/seaphore) (<https://github.com/appliedzkp/seaphore>), there is a need to include a public input that is not involved in any of the logic constraints, we just want it attached to the proof. In Semaphore's case, one reason is to preserve anonymity of the proof generator - since in Ethereum every transaction has to be broadcast from an address with some ETH in it to pay transaction fees, this could reveal the identity of the proof broadcaster. This is why there are *relayers* who broadcast in their behalf and receive some fee. To prevent front-running, the proof generator attaches the address of the relayer to the proof.

So, what happens when a public input is not involved in any of the logic constraints? Knowledge soundness just says I can find a valid witness. A valid witness for a proof and instance with a public input that doesn't participate is still valid when that public input is changed!

That means that the Groth16 construction by itself does not guarantee that the instance can't change, even if I don't know the witness and just hold the proof. I can change the proof to be for a different instance. I can even change the proof itself and it will still be valid.

[Baghery et al.](https://eprint.iacr.org/2020/811) (<https://eprint.iacr.org/2020/811>) examined this topic in more details, and have deduced that if some important conditions on the constraint system are added, then Groth16 is *weak simulation-extractable*, meaning that while the proof itself can change (in their description, re-randomized), the instance will remain fixed - which is exactly the property we'd want in Semaphore. There are other constructions, both in that paper and different proving systems (like GM17), that provide *simulation extractability*, where even the proof cannot change.

The conditions from Baghery et al. are the following.

Theorem 1. Assume that $\{u_i(x)\}_{i=0}^l$ are linearly independent and $\text{Span}\{u_i(x)\}_{i=0}^l \cap \text{Span}\{u_i(x)\}_{i=l+1}^m = \emptyset$. Then Groth16 achieves weak white-box SE against algebraic adversaries under the $(2n-1, n-1)$ -dlog assumption.

This is somewhat obscure and unclear, and one of the goals of this post is to provide intuition around what these conditions mean and what happens when they're missing.

We believe that some conversations around this dangerous property of the bare Groth16 constructions has led to the belief that the "fake square constraint" is needed, and we've since seen it in a few implementations.

What we've also seen is that practically all implementations of Groth16 add implicit extra constraints that immediately satisfy the conditions in Baghery et al. and therefore no explicit "fake square constraint" is needed to ensure a public input won't change.

R1CS and QAP

We recap what R1CS is and how it's converted to Quadratic Arithmetic Programs (QAP). We focus on the representations, since this is what's important to understand the attacks. If you already know this, feel free to skip to the next part.

The low-level descriptions of constraints in Groth16 is *R1CS*, a Rank-1 Constraint System, where each constraint takes the following form.

$$\left(\sum_{i=0}^m z_i u_{i,j} \right) \left(\sum_{i=0}^m z_i v_{i,j} \right) = \sum_{i=0}^m z_i w_{i,j}$$

Terminology:

- z_i - the i -th variable. By convention we set $z_0 = 1$. There are m other variables.
- $u_{i,j}$ - the coefficient of variable i in constraint j in the left input.
- $v_{i,j}$ - the coefficient of variable i in constraint j in the right input.
- $w_{i,j}$ - the coefficient of variable i in constraint j in the output.

For example, if I want the variables z_1, z_2, z_3 to satisfy $(z_1 + 2z_2)(z_1) = z_3$, then I'd have a constraint of the form:

$$(z_0 \cdot 0 + z_1 \cdot 1 + z_2 \cdot 2 + z_3 \cdot 0)(z_0 \cdot 0 + z_1 \cdot 1 + z_2 \cdot 0 + z_3 \cdot 0) = (z_0 \cdot 0 + z_1 \cdot 0 + z_2 \cdot 0 + z_3 \cdot 1)$$

Assuming arbitrarily it's constraint number 0, this means:

$$u_{0,0} = 0, u_{1,0} = 1, u_{2,0} = 2, u_{3,0} = 0$$

$$v_{1,0} = 0, v_{1,0} = 1, v_{2,0} = 0, v_{3,0} = 0$$

$$w_{1,0} = 0, w_{1,0} = 0, w_{2,0} = 0, w_{3,0} = 1$$

Taking it further, we can arrange each set of constraints $u_{i,j}, v_{i,j}, w_{i,j}$ in matrices. For example, assume we that we also require $(z_2 + z_3)(z_1) = z_1 + z_2$, which would be constraint number 1, the matrices would look like:

$$A = \begin{pmatrix} 0 & 1 & 2 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

$$B = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

$$C = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

and arranging the variables in a column vector $z = \begin{pmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{pmatrix}$, satisfying the entire constraint system would mean checking:

$$Az \circ Bz = Cz$$

where Az, Bz, Cz mean a matrix-vector product and $x \circ y$ means an element-wise product.

Taking it even further, we can take each column in the matrix and compress it into a polynomial over some domain $\{\gamma_0, \gamma_1\}$. For instance, taking A and transforming each column into a polynomial would give us the vector $(u_0(x), u_1(x), u_2(x), u_3(x))$, such that $u_i(\gamma_j) = A_{j,i}$.

The parameters of the QAP are the polynomials $u_i(x), v_i(x), w_i(x)$, and checking that the entire constraint system is satisfied means checking that $A(x)B(x) = C(x)$, where $A(x) = \sum_{i=0}^m z_i u_i(x)$ and similarly for $B(x)$ and $C(x)$.

This is done by defining another polynomial $t(x) = (x - \gamma_0)(x - \gamma_1)$ and checking whether there exists a polynomial $H(x)$ such that

$$\frac{A(x)B(x) - C(x)}{t(x)} = H(x)$$

this works because if the constraints are satisfied at each γ_j then both the numerator and denominator are multiples of $(x - \gamma_j)$ allowing those terms to cancel out before they evaluate to 0. On the other hand, if the constraint is not satisfied then the numerator will be non-zero and the denominator will be zero, which would result in an undefined result in theory, and in practice the division will just not result in a polynomial $H(x)$.

Attacks

To understand the different attacks that can be done, we briefly show how the Groth16 Setup, Prove and Verify operations look like. We focus on the mechanics rather than the motivation, since that will be sufficient to attack the system.

We assume you're familiar with pairing groups, their general structure and their properties. If not, [the post by Vitalik](https://medium.com/@VitalikButerin/exploring-elliptic-curve-pairings-c73c1864e627) (<https://medium.com/@VitalikButerin/exploring-elliptic-curve-pairings-c73c1864e627>) is a good introduction.

Groth16

Notation

$[e]_i$ - means $e \cdot G_i$, where G_i is a generator of the pairing group \mathbb{G}_i .

Setup

$$(\sigma, \tau) = \text{Setup}(R); \tau = (\alpha, \beta, \gamma, \delta, x)$$

$$\sigma_1 = \left(\alpha, \beta, \delta, \{x^i\}_{i=0}^{n-1}, \left\{ \frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\gamma} \right\}_{i=0}^l, \left\{ \frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\delta} \right\}_{i=l+1}^m, \left\{ \frac{x^i t(x)}{\delta} \right\}_{i=0}^{n-2} \right)$$

$$\sigma_2 = (\beta, \gamma, \delta, \{x^i\}_{i=0}^{n-1})$$

The elements in τ are random values and each of the elements e in σ_1 and σ_2 are assumed to be $[e]_i$.

Notice that σ_1 contains elements that are divided by γ and elements that are divided by δ . The l elements divided by γ are the public inputs and the ones divided by δ are the private inputs.

Note - these parameters are usually generated for production use-cases by using an MPC protocol, such as MMORPG (<https://eprint.iacr.org/2017/1050.pdf>).

Prove

$$\pi \leftarrow \text{Prove}(R, \sigma, z_1, \dots, z_m); \pi = ([A]_1, [C]_1, [B]_2)$$

$$A = \alpha + \sum_{i=0}^m z_i u_i(x) + r\delta$$

$$B = \beta + \sum_{i=0}^m z_i v_i(x) + s\delta$$

$$C = \frac{\sum_{i=l+1}^m z_i (\beta u_i(x) + \alpha v_i(x) + w_i(x)) + h(x)t(x)}{\delta} + As + Br - rs\delta$$

$$h(x) = \frac{A(x)B(x)-C(x)}{t(x)}$$

Verify

$$0/1 \leftarrow \text{Vfy}(R, \sigma, a_1, \dots, a_l, \pi)$$

check that:

$$[A]_1 \cdot [B]_2 = [\alpha]_1 \cdot [\beta]_2 + \sum_{i=0}^l z_i \left[\frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\gamma} \right]_1 \cdot [\gamma]_2 + [C]_1 \cdot [\delta]_2$$

Notice that the verifier directly uses the public inputs in the verification equation.

Attack 1

The attacker is given a proof and a set of public inputs such that verification equation passes. Since the attacker doesn't know any private inputs we will concentrate on the verification algorithm.

Assume that we have a public input z_i that doesn't participate in any constraint. That means that $u_i(\gamma_j) = v_i(\gamma_j) = w_i(\gamma_j) = 0$ for all j . That in turn means that for all x , $u(x) = v(x) = w(x) = 0$, the zero polynomial. This further means that the element $\left[\frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\gamma} \right]_1$ in σ_1 is just... 0!

As a consequence, it doesn't matter which value we put in z_i , the verification equation will trivially ignore it since it is multiplied by 0.

This is scary, because our goal was to tie a public input to the proof in a way that it cannot change, as we've seen is needed in Semaphore. This kind of worry led to introducing the "fake square constraint", which while it is always satisfiable and doesn't add any interesting constraint

to the system (since we can always find a value s such that $a * a = s$), because it wasn't clear to implementers that, in practice, it's already taken care of.

Attack 2

We start with the same setup - The attacker is given a proof and a set of public inputs such that verification equation passes.

Specifically, recalling the conditions from Baghery et al.

Theorem 1. Assume that $\{u_i(x)\}_{i=0}^l$ are linearly independent and $\text{Span}\{u_i(x)\}_{i=0}^l \cap \text{Span}\{u_i(x)\}_{i=l+1}^m = \emptyset$. Then Groth16 achieves weak white-box SE against algebraic adversaries under the $(2n - 1, n - 1)$ -dlog assumption.

Let's explore this statement in depth and understand how the second condition of a non-empty span intersection can lead to an attack. After that we introduce the common solution for both attacks.

Let the games begin.

Without loss of generality fix some $k \in [l + 1..m]$ and $j \in [0..l]$ such that the polynomials $u_k(x)$, $v_k(x)$ and $w_k(x)$ are linearly dependent with the polynomials $u_j(x)$, $v_j(x)$ and $w_j(x)$ respectively.

This means, at the very least, that $\text{span}(\{u_i(x)\}_{i=0}^l) \cap \text{span}(\{u_i(x)\}_{i=1}^m) \neq \emptyset$, which is the condition we were examining.

This can happen when, for example, a private variable's coefficient in a constraint is the same multiple of the public variable's coefficient in each of the left input, right input and output. For example, if z_1 is a public input and z_8 is a private input, we'd have:

$$(z_1 - 2 * z_8)(2z_1 - 4 * z_8) = 3z_1 - 6z_8$$

In this case, $\beta * u_k(x) + \alpha * v_k(x) + w_k(x)$ and $\beta * u_j(x) + \alpha * v_j(x) + w_j(x)$ are also linearly dependent. Let's define

$$W_k(x) = \beta * u_k(x) + \alpha * v_k(x) + w_k(x)$$

$$PI_j(x) = \beta * u_j(x) + \alpha * v_j(x) + w_j(x)$$

such that for some linear factor l

$$l * W_k(x) = PI_j(x)$$

Note that this "non empty span" relation could be much more complex, including multiple witnesses and public polynomials, we just keep it simple so it's easier to read.

From now on forget about circuits, R1CS, etc. Focus only on the pure mathematical relationship between data and how we can construct a malleable proof with what we have.

In order to see this attack more clearly we start by rewriting the verification equation:

$$[A]_1 \cdot [B]_2 = [\alpha]_1 \cdot [\beta]_2 + \sum_{\substack{i=0 \\ i \neq j}}^l z_i \left[\frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\gamma} \right]_1 \cdot [\gamma]_2 + z_j \left[\frac{PI_j(x)}{\gamma} \right]_1 \cdot [\gamma]_2 + [C]_1 \cdot [\delta]_2$$

This is still same equation with just $PI_j(x)$ isolated.

Before we start with the concrete attack, let's take a look at the result of the pairing operation when treating a pairing intuitively as a multiplication (we introduce $C_{paired}(x)$ just to keep notation and formulas compact):

$$Result = \alpha * \beta + \sum_{i=0}^l z_i * (\beta u_i(x) + \alpha v_i(x) + w_i(x)) + C_{paired}(x)$$

At this point it might not be clear how a linear dependency relation can be exploited, but notice one important thing - after the pairing, the unknown denominator γ is canceled out. We will need this idea later.

Again, without loss of generality, let's substitute z_j with z'_j and see how it affects the final result.
 $Result' = Result + z'_j * PI_j(x) - z_j * PI_j(x) = Result + (z'_j - z_j) * PI_j(x).$

So, in order to create malleable proof we need to somehow subtract $(z'_j - z_j) * PI_j(x)$ from the final verification equation.

And finally $W_k(x) = l * PI_j(x)$ comes into play, with one small issue:

$$\left[\frac{W_k(x)}{\delta} \right]_1 = \left\{ \frac{\beta u_k(x) + \alpha v_k(x) + w_k(x)}{\delta} \right\}$$

is publicly available in σ_1 but it's divided by the unknown value δ . Good news, we have already solved the same issue with $PI_j(x)$ and γ - we will use help of the pairing to get rid of it. The only term that can help us to get rid of δ is C , thus let's modify C so that it carries $(z'_j - z_j) * PI_j(x)$.

$$C' = C + l * (z_j - z'_j) * \left[\frac{W_k(x)}{\delta} \right]_1$$

Now we have:

$$\begin{aligned} [A]_1 \cdot [B]_2 &= [\alpha]_1 \cdot [\beta]_2 + \sum_{\substack{i=0 \\ i \neq j}}^l z_i \left[\frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\gamma} \right]_1 \cdot [\gamma]_2 + z'_j \left[\frac{PI_j(x)}{\gamma} \right]_1 \cdot [\gamma]_2 + C'(x) \cdot [\delta]_2 \\ &= [\alpha]_1 \cdot [\beta]_2 + \sum_{\substack{i=0 \\ i \neq j}}^l z_i \left[\frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\gamma} \right]_1 \cdot [\gamma]_2 + z'_j \left[\frac{PI_j(x)}{\gamma} \right]_1 \cdot [\gamma]_2 + C(x) \cdot [\delta]_2 + l * (z_j - z'_j) * \left[\dots \right]_1 \end{aligned}$$

After pairing:

$$\begin{aligned} &= \alpha * \beta + \sum_{\substack{i=0 \\ i \neq j}}^l z_i * (\beta u_i(x) + \alpha v_i(x) + w_i(x)) + z'_j * PI_j(x) + C_{paired}(x) + l * (z_j - z'_j) * W_k(x) \\ &= \alpha * \beta + \sum_{\substack{i=0 \\ i \neq j}}^l z_i * (\beta u_i(x) + \alpha v_i(x) + w_i(x)) + z'_j * PI_j(x) + C_{paired}(x) + (z_j - z'_j) * PI_j(x) \\ &= \alpha * \beta + \sum_{\substack{i=0 \\ i \neq j}}^l z_i * (\beta u_i(x) + \alpha v_i(x) + w_i(x)) + z'_j * PI_j(x) + C_{paired}(x) + z_j * PI_j(x) - z'_j * PI_j(x) \\ &= \alpha * \beta + \sum_{\substack{i=0 \\ i \neq j}}^l z_i * (\beta u_i(x) + \alpha v_i(x) + w_i(x)) + C_{paired}(x) + z_j * PI_j(x) \\ &= \alpha * \beta + \sum_{i=0}^l z_i * (\beta u_i(x) + \alpha v_i(x) + w_i(x)) + C_{paired}(x) \end{aligned}$$

Which is exactly the same *Result* we started with! This means that attacker was able to change the public input and forge a valid proof for the new statement.

Solution in practice

There is just one, at first glance magical constraint, that will solve both issues.

$$z_i * 0 = 0$$

It's not intuitive at all, since $z_i * 0 = 0$ is always satisfied no matter what value we assign to z_i . But keep in mind the idea that we don't care about the actual values at all, just about the polynomials. This constraint solves our problem, since we have $u_i(\gamma_j) = 1$ in this constraint!

1. With $u_i(\gamma_j) = 1$, $u_i(x), v_i(x), w_i(x)$ are not all zero polynomials which also leads to $\left[\frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\gamma} \right]_1$ not being zero, and in the verification algorithm changing z_i to anything but the value the prover supplied will result in the verification equation check failing.
- It may not be clear at first look, but when $u_i(x), v_i(x), w_i(x)$ are the zero polynomials, we have linear dependency between $u_i(x)$ for $i \in 0, \dots, l$.

2. Going back to the condition from Baghery et al., and assuming we have this condition satisfied for each z_i in constraint j , we can now safely say that $\{u_i(x)\}_{i=0}^l$ are linearly independent, since any linear combination $\sum_{i=0}^l c_i u_i(x)$ will be $c_i u_i(\gamma_j)$, and since $u_i(\gamma_j) = 1$, it follows that all $c_i = 0$. This is true for both conditions from Baghery et al.

As promised, we point to where the popular implementations of snarkjs and arkworks introduce these extra constraints. Confusingly, the constraints are introduced implicitly during the setup rather than when building the constraints system, and so you won't see them when examining constraints before performing the setup.

Circuit setup generation in snarkjs:

[\(https://github.com/iden3/snarkjs/blob/2c5e04e3cc1a8da73252ba3c2262309d49a15c0b/src/zkey_new.js#L291\).](https://github.com/iden3/snarkjs/blob/2c5e04e3cc1a8da73252ba3c2262309d49a15c0b/src/zkey_new.js#L291)

Prover in arkworks: [\(https://github.com/arkworks-rs/groth16/blob/765817f77a6e14964c6f264d565b18676b11bd59/src/r1cs_to_qap.rs#L176\).](https://github.com/arkworks-rs/groth16/blob/765817f77a6e14964c6f264d565b18676b11bd59/src/r1cs_to_qap.rs#L176)

Attack implementation

We provide the implementation of the attack with small snarkjs tweaks which was the actual modification done to enable our puzzle:

[\(https://github.com/geometryresearch/groth16-malleability_\(https://github.com/geometryresearch/groth16-malleability\).](https://github.com/geometryresearch/groth16-malleability_(https://github.com/geometryresearch/groth16-malleability).)

[\(https://github.com/geometryresearch/snarkjs/commit/353405187586e990f669e16735f54efa490e4944#diff-97ab8679b549ee4348bb7108b50a8ea0d8a8dae36678a0074143bd919a787f2\).](https://github.com/geometryresearch/snarkjs/commit/353405187586e990f669e16735f54efa490e4944#diff-97ab8679b549ee4348bb7108b50a8ea0d8a8dae36678a0074143bd919a787f2)

Conclusion

This is just one out of the many examples showing how cryptography can be tricky, why you need to understand the proving schemes you rely on, and how superstitions can arise around how to make them secure.

It also highlights how implementers can sometimes use bits of knowledge coming from outside of a specific paper to make implementations secure, leaving this knowledge somewhat undocumented.

Lastly, recall the "fake square constraint" from the beginning of this post:

```
// Dummy square to prevent tampering signalHash.
signal signalHashSquared;
signalHashSquared <= signalHash * signalHash;
```

If you think again, it achieves exactly the same effect as $z_i * 0 = 0$, as it also enforces that $\left[\frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\gamma} \right]_1$ is not zero.

If you haven't yet, we encourage you to try your hands at the puzzle and see whether you can apply what you've learnt here in practice!

Acknowledgements

We thank Nicolas Mohnblatt, Wei Jie Koh and Tom Walton-Pocock for their diligent review and helpful comments.

We thank Tom Waite for his thorough review and finding interesting attack vectors in first versions of the contract.

tags: **Groth16 Cryptography ZeroKnowledge**

What do you think?

0 Responses



0 Comments

Login ▾

G

Start the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS

Name

Share

Best **Newest** **Oldest**

Be the first to comment.

[Subscribe](#)

[Privacy](#)

[Do Not Sell My Data](#)

Hashing to the secp256k1 Elliptic Curve

Koh Wei Jie



Geometry Study Club: Hashing to the secp256k1 Elliptic Curve

Koh Wei Jie October 2022

Many cryptographic protocols, such as aggregatable distributed key generation and BLS signature schemes, require hash-to-curve algorithms, which deterministically convert an arbitrary bytestring into a point on an elliptic curve. Such algorithms are not trivial because they must not only produce valid curve points, but also do so securely and efficiently.

In this post, I will summarise the state of the art of hash-to-curve functions with an emphasis on their application to the secp256k1 elliptic curve, as well as some of the security considerations and performance optimisations behind hash-to-curve algorithms in general.

Background

At Geometry, we worked with Aayush and Lakshman of 0xPARC and Personae Labs on a signature scheme that produces unique deterministic nullifiers. The scheme is meant to be compatible with existing Ethereum wallets as it works with keypairs on the secp256k1 curve. To produce a signature, the signer must derive an elliptic curve point that is the output of a hash-to-curve function over a message and their public key.

To maximise the chances that this signature scheme sees widespread adoption and standardisation, we opted for a hash-to-curve ciphersuite that is itself part of a set of functions undergoing a process of standardisation: `secp256k1_XMD:SHA-256_SSWU_R0_`. It is defined in *Hashing to Elliptic Curves*, a draft standard submitted to the Internet Research Task Force (IRTF). One should note that at the time of writing, this document is relatively new and has not yet been formally made an IETF standard.

In a future post, I will elaborate on a circom implementation of this hash-to-curve function which allows it to be used in zk-SNARK circuits.

About *Hashing to Elliptic Curves*

Hashing to Elliptic Curves (henceforth referred to as the draft standard) provides specific algorithms for hashing arbitrary bytestrings to elliptic curves. It defines specific ciphersuites - a collection of constants and algorithms - for various elliptic curves, such as secp256k1, P-256, and BLS12-381. While it only provides ciphersuites for this limited set of elliptic curves, it also provides general guidance for how to construct secure ciphersuites for other curves if a reader so desires.

The `hash_to_curve` algorithm

For the `secp256k1_XMD:SHA-256_SSWU_R0_` ciphersuite, the draft specification defines the `hash_to_curve` function as such:

1. Given an input message `msg` of arbitrary length, use `hash_to_field` to deterministically produce two field elements `u[0]` and `u[1]`.
2. Compute `Q0 = map_to_curve(u[0])`.
3. Compute `Q1 = map_to_curve(u[1])`.
4. Compute `R = Q0 + Q1` using point addition.
5. Clear the cofactor of `R` and return the result.

`hash_to_field`

`hash_to_field` performs `expand_message_xmd`, which hashes `msg` and outputs a 96-byte array. The output `u[0]` is the first 48 bytes, and `u[1]` is the last 48 bytes. I will elaborate on `expand_message_xmd` in a separate section; for now, simply assume that it emits a uniformly random hash of the input.

`map_to_curve`

`map_to_curve` uses a constant-time method that always finds a point on an elliptic curve given a field element. It comes in various flavours, depending on the type of the curve and its parameters.

The draft specification prescribes the Elligator 2 method for Montgomery curves, and the twisted Edwards Elligator 2 method for twisted Edwards curves. For Weierstrass curves like `secp256k1`, there are three options:

1. The Shallue-van de Woestijne (SW) method, which works with any elliptic curve, but is the slowest option.
2. The Simplified Shallue-van de Woestijne-Ulas (SSWU) method, which applies to curves defined as $y^2 = x^3 + Ax + B$ where $A \neq 0$ and $B \neq 0$.
3. The Simplified SWU method for curves where AB equals 0. This is the most efficient of these three methods.

For our purposes, the Simplified SWU method where AB equals 0 is relevant because the `secp256k1` curve is defined with $A = 0$ and $B = 7$, so $AB = 0$.

`clear_cofactor`

This function converts a point on a curve into a point in the prime order subgroup. To do so is simply to multiply the point with a constant h . In the case of the `secp256k1` curve, nothing has to be done as h for this curve equals 1.

Security considerations

Any hash-to-curve function or implementation should exhibit certain security properties to preserve the security of the cryptographic protocol in which it is adopted. Otherwise, a weak hash-to-curve function can expose the protocol to attacks. As the draft standard points out, a hash-to-curve function must not only be collision-resistant, but should also not reveal the discrete logarithm of the output point.

To reason about the security of a cryptographic protocol, cryptographers write proofs based on assumptions about its components. These theoretical assumptions can be described as *models*. The authors of the draft standard argue that their methods are secure in what is known as the *random oracle model* (ROM). As such, protocols which rely on such hash-to-curve functions can, if designed correctly, also be proven to be secure in the ROM.

The Random Oracle Model

The authors of the draft standard argue that their hash-to-curve algorithms are secure in the ROM. These functions achieve this in two ways: by ensuring that the algorithms convert input messages via hash functions into field elements that are uniformly random, and by ensuring that the output curve point of the hash-to-curve algorithm is statistically uniform.

First of all, the hash-to-field procedure relies on a method that expands a bytestring into a uniformly random bytestring. There are two variants of this method: `expand_message_xof` and `expand_message_xmd`.

`expand_message_xof` relies on a hash function which natively provides an output of a desired length. Since this hash function (which the recommend to be in the SHAKE XOF family) is provably secure under the ROM, it is easy to show the hash-to-field function is also secure under the ROM.

`expand_message_xmd`, however, relies on a hash function with a fixed-length output. While some hash functions are indifferentiable from a random oracle, or are sponge-based hash functions whose inner function is provably a random permutation or random transformation, and therefore allow one to argue that `expand_message_xmd` is indifferentiable from a random oracle, there are some hash functions for which this is not the case: Merkle–Damgård hash functions.

The draft standard cites Coron et al in CDMP05 who argue that the strengthened Merkle–Damgård transformation, employed in hash functions like SHA, does not meet the following security property:

the arbitrary length hash function H must behave as a random oracle when the fixed-length building block is viewed as a random oracle or an ideal block-cipher.

To allow `expand_message_xmd` to be provably secure in the ROM *and* employ Merkle–Damgård hash functions like SHA256, the authors use a construction in CDMP05 named HMAC^f (p12). A rough description of this construction (which skips a suffix-padding step for simplicity) is as follows:

1. Let k be the number of input bytes that a Merkle–Damgård hash function MD^f requires and let m be the message.
2. Let m_0 be a byte array of k zeros.
3. Let $y' = \text{MD}^f([m_0||m])$ where $||$ denotes concatenation.
4. Return $\text{MD}^f([y'])$.

Recall that length-extension attacks work when given a hash value $h(m_a)$, an attacker is able to trivially instantiate the internal state of the underlying hash function with $h([m_a])$ in order to derive $h([m_a||m_b])$, *even if the attacker does not know the value of m_a* . This violates a crucial property of the random oracle model – that is, it should not be the case that the hash of one message (m_b) can be determined from that of another (m_a).

The HMAC^f construction does not suffer from this issue. An attacker who wishes to derive $\text{HMAC}^f([m||x])$ only knowing y' will be unlikely to succeed. Consider what an attacker may attempt:

1. Obtain $a = \text{HMAC}^f([m])$.
 - Note that a is equivalent to $\text{MD}^f(\text{MD}^f([m_0||m]))$.
2. Knowing only MD^f and y' , the attacker wants to derive $b = \text{HMAC}^f([m||x])$.
 - Note that b is equivalent to $\text{MD}^f(\text{MD}^f([m_0||m||x]))$.
3. The attacker needs $\text{MD}^f([m_0||m])$ to equal m_0 so that she can initialise the internal state of MD^f with m_0 and then proceed with the rest of MD^f on x to obtain $\text{MD}^f([m_0||m||x])$.
4. It is, however, highly unlikely that $\text{MD}^f([m_0||m])$ equals m_0 .

Therefore, $\text{HMAC}^f([m])$ does not suffer from the length extension attack, and is therefore more easily proven to be secure under the ROM.

As such, `expand_message_xmd` implements this construction, commonly known as a hash-based message authentication code (HMAC). For reference, note that its implementation follows the following pattern. For hash-to-curve ciphersuites that use SHA256 and require a 96-byte output, `expand_message_xmd` is roughly defined as such:

1. Let `Z_pad` be a byte array of 64 zeros (since $k = 64$ for SHA256). This is analogous to m_0 as described above.
2. Construct `msg_prime` as the concatenation of `Z_pad`, the message, and some constants, such as `DST_prime`.
3. Compute `b_0 = SHA256(msg_prime)`. This is analogous to y' as described above.
4. Compute `b_1 = SHA256(b_0 || 1 || DST_prime)`.
5. Compute `b_2 = SHA256(strxor(b_0, b1) || 2 || DST_prime)`.
6. Compute `b_3 = SHA256(strxor(b_0, b2) || 3 || DST_prime)`.
7. Return `b_1 || b2 || b3`.

As explained above, prepending `Z_pad` in step 2 allows prevents length extension attacks which could allow an attacker to more easily influence the values `b_1`, `b_2`, and `b_3`.

Domain separation

One may notice the `DST_prime` constant in the `expand_message_xmd` algorithm above. It encodes a bytestring that represents a domain separation tag (DST), followed by the DST length as a single byte. Each

ciphersuite has a unique DST, such as QUUX-V01-CS02-with-secp256k1_XMD:SHA-256_SSWU_RO_ for the secp256k1_XMD:SHA-256_SSWU_RO_ suite.

The reason for including a unique `DST_prime` in `expand_message_xmd` is to ensure that its output is specific to one and only one ciphersuite. In effect, each ciphersuite can be understood to operate with an independent random oracle, which is an important security assumption for protocols proven secure under the random oracle model.

Output uniformity

In the draft standard, each elliptic curve comes with two ciphersuite: one that performs hash-to-curve, and another which performs encode-to-curve. The difference between the two is that the output of the former function is more statistically uniform than that of the latter, but the latter is more efficient. The authors state that a safe default is to use hash-to-curve, unless one is sure that nonuniformity is acceptable in one's cryptographic protocol.

The reason that hash-to-curve differs lies in its implementation. The `hash_to_curve` function derives two different elliptic curve points from the input message (using the `map_to_curve` function) and adds them together. In contrast, the `encode_to_curve` function only calls `map_to_curve` once. As the output of `map_to_curve` is nonuniform, it follows that the output of `encode_to_curve` is nonuniform. Since `hash_to_curve` adds two nonuniform points, however, the output is uniform. The authors cite Brier et al which proves that this is the case.

Constant-time implementation

An important security property of the hash-to-curve ciphersuites in the draft standard is that they can be computed *in constant time*. This property is critical if the message to be hashed must be secret.

To have secure constant-time hash-to-curve functions is a step forward because much prior work on hash-to-curve algorithms use a try-and-increment approach, where an input message is converted into a field element, tested if it is a valid x-coordinate in the curve, and if not, it is incremented and the test is repeated.

The drawbacks of such try-and-increment methods are twofold. First, they must be performed a set number of times (n) even if one finds a valid point in $n-x$ tries. This is to thwart side-channel timing attacks. Secondly, there is always a possibility that n is insufficient and so one has to try again with a new message. The added complexity required to handle these situations can lead to a greater chance of security vulnerabilities in a protocol, as seen in the Dragonblood vulnerabilities discovered in implementations of the WPA3 and EAP-pwd protocols.

Conclusion

In this post, I outlined various security and performance considerations made by the authors of *Hashing to Elliptic Curves*. I believe that a well-designed and secure set of hash-to-curve ciphersuites will greatly benefit the cryptography landscape as such functions are at the core of many important protocols. To that end, more constructive scrutiny on this draft standard is necessary.

Optimized BLS multisignatures on EVM

Kobi Gurkan



Optimized BLS multisignatures on EVM

BLS signatures are attractive in their homomorphic properties. In multisignatures, multiple signers are able to collaborate non-interactively to produce a small signature attesting to the approval of multiple signers on the same message. Similarly, in aggregate signatures the same is achieved, on different messages, with higher verification cost.

The post is concerned with how to make verification of multisignatures efficient.

BLS signatures

The signature scheme requires a pairing friendly curve and to be able to efficiently perform operations on the base field \mathbb{F}_q and the two source groups \mathbb{G}_1 and \mathbb{G}_2 . We also note \mathbb{F}_r as the scalar field of the curve.

[EIP-196](#) introduces addition and scalar multiplication operations on \mathbb{G}_1 of the *BN254* curve, [EIP-197](#) introduces the check that a product of pairings is equal to one on *BN254*. Subsequently, [EIP-1108](#) reduces the costs of these operations. Finally, we have low-level support for modular arithmetic in the EVM in the form of the `addmod` and `mulmod` operations to perform operations on the base field \mathbb{F}_q of *BN254*.

Recapping an EVM-suitable version of BLS signatures, we have:

- Generator $G \in \mathbb{G}_2$
- Secret key $sk \in \mathbb{F}_r$
- Public key $pk \in \mathbb{F}_r$
- Message $m \in \mathbb{B}^*$, a string of bytes
- Hash to curve algorithm $H : \mathbb{B}^* \rightarrow \mathbb{G}_1$
- Signature $\sigma \in \mathbb{G}_1$

Verification is done as follows.



Generally, roles can be reversed and signatures can be in \mathbb{G}_2 . This version is EVM-suitable for a few reasons. Immediate reasons include that $-G$ can be precomputed and only a product of pairings is used. Most importantly, though, is the fact that $H(m)$ is relatively efficiently computable in the case of *BN254* and signatures being in \mathbb{G}_1 .

Hash to curve

The hash to curve process generally includes hashing to possible x coordinates (base field elements), taking square roots (to produce y using the curve equation) and, after obtaining a curve point, multiplying by a cofactor which takes the obtained point from a larger group of points to the specific prime group of points that is \mathbb{G}_1 . It's particularly efficient in *BN254* and hashing to \mathbb{G}_1 since the cofactor is 1 - meaning that every point you obtain through hashing is directly in \mathbb{G}_1 .

As shown in <https://ethresear.ch/t/bls-signatures-in-solidity/7919> and the implementation in [Hubble](#), it's possible to implement either try-and-increment and [Foque-Tibouchi](#) hashing efficiently. In cases where there's an intermediate aggregator that's suitable, it's possible to optimize further by providing hints to the expensive *sqrt* operation through [mapPointWithHelp](#).

Doing the same on \mathbb{G}_2 will be expensive, as the cofactor in \mathbb{G}_2 is around 254 bits and multiplying by a 256-bit scalar can cost around 4 million gas, as shown in [solidity-BN256G2](#). It's possible to significantly optimize this using some tricks that create a shortcut in the scalar multiplication (using [endomorphisms](#)), but the resulting gas cost will still be quite high, likely not less than a million.

Multisignatures

A dilemma appears when moving to a multisignature case. This situation appears, for example, when we have a validator set that collectively needs to approve a message.

Public key aggregation, if taking proper care during registration to obtain proofs-of-possession from participants, can be as simple as adding all the public keys together. Similarly, the signatures are aggregated using simple addition as well.

We know that having signatures in \mathbb{G}_1 is efficient, because then the hash to curve is to \mathbb{G}_1 . The public keys would then be on \mathbb{G}_2 . Since there's no precompile for additions on \mathbb{G}_2 , we have to resort to implementing it on the EVM, which results in around 30k gas for each addition (according to [solidity-BN256G2](#)), as opposed to 500 in \mathbb{G}_1 .

Even if we were to target for the happy case where usually most signers participate and only one or two are missing, and so we would start with a fully aggregated public key of all the signers and then subtract, removing those would still be expensive if signatures are frequent enough and even more expensive if there are more.

Helped aggregation

Since we have cheap addition in \mathbb{G}_1 , maybe we could use that?

We could ask participants, when registering, to submit two public keys corresponding to their secret key sk :

- $pk_2 = sk \cdot G \in \mathbb{G}_2$, same as the original public key.
- $pk_1 = sk \cdot H \in \mathbb{G}_1$, where H is a generator in \mathbb{G}_1 . This is a new addition.
- We check a BLS signature $e(-\sigma, G) * e(H_2(pk_2), pk_2) = 1$, where H_2 is a hash to curve algorithm that is different than H . For example, it can be almost the same as H , but with a different domain.
- We check $e(pk_1, G) * e(-H, pk_2) = 1$ during registration. This ensures pk_1 and pk_2 have the same secret key, due to the B-KEA assumption.

Read Appendix 1 for more details why both checks are needed. The BLS signature check is a proof-of-possession, as described in The Power of Proofs-of-Possession: Securing Multiparty Signatures against Rogue-Key Attacks.

Now, whenever a multisignature is submitted, including a signature and a bitmap of included participants, the submitter further includes the aggregated public key composed of pk_2 variants on \mathbb{G}_2 , denoted apk - e.g. $apk = pk_{2,Alice} + pk_{2,Bob} + pk_{2,Charlie}$. The contract itself performs the aggregation through adding the pk_1 variants of the public keys, resulting in some P_1 - e.g. $P_1 = pk_{1,Alice} + pk_{1,Bob} + pk_{1,Charlie}$. The contract then checks $e(P_1, G) * e(-H, apk) = 1$.

This results in the following (pseudo) interface.

```
function verifySignature(bool[] signersBitmap, G2Element apk, G1Element signature)
```

where `apk` is computed off-chain as the sum of all the pk_2 variants that correspond to active bits in `signersBitmap`. The function then takes the fully aggregated public_key, i.e. the sum of all `pk_1` variants, from contract storage and subtracts the `pk_1` variants that correspond to inactive bits. Assuming most signers participate in each signature,

this results in better gas usage than adding all the `pk_1` variants from scratch each time.

This allows efficiently aggregating keys in \mathbb{G}_1 and verifying an off-chain computed aggregated key on \mathbb{G}_2 , such that each change in the aggregated public key results in a marginal point addition gas cost.

Putting it all together

We can avoid introducing extra pairing checks, namely checking $e(pk_1, G) * e(-H, pk_2) = 1$, by bundling it with an existing check in the BLS verification equation. First, since we have two individual pairing equations to check, we have to separate them by a delineation factor α , which needs to be random. We can derive it by cheaply hashing the prover-controlled inputs to the verification equation, namely (σ, m, P_1, apk) , and then interpreting the result as a number.

Then, we have the following.

$$e(\sigma, -G) * e(H(m), apk) * (e(P_1, G) * e(-H, apk))^{\alpha} = 1$$

We will know each of the equations is 1 if the overall result is 1 because of the Schwartz–Zippel lemma.

Utilizing bilinearity, we bring the α to the \mathbb{G}_1 component.

▪

We then combine terms, removing the need for extra pairings.

▪

Proof of concept notes and code in the form of a Sage notebook and remix contracts available in the following gist:

<https://gist.github.com/kobigurk/257c1783ddf556e330f31ed57febc1d9>, resulting in around 16k gas overhead for hashing into alpha and the two scalar multiplication. Might be possible to optimize.

Conclusion

Through introducing an extra element and check during registration of participants, we show a method that results in efficient BLS multisignature verification, where each change in the aggregated public key results in only a small gas marginal cost, and a fixed cost of cheap hashing a few elements and two scalar multiplications, each of which costs 6k gas.

Acknowledgements

Thanks to Asa Oines from Hyperlane for motivating the problem and to Nico Mohnblatt for discussions that led to this idea.

Thanks to Mary Maller for pointing out that B-KEA in itself is not enough for a “knowledge of secret key” proof. Having it alone would open the multisignature protocol to rogue key attacks.

Thanks to Jonathan Wang and Yi Sun for finding a glaring soundness error in a previous version of the protocol.

Thanks to Wei Jie Koh, Daniel Benarroch, Chih-Cheng Liang, Antonio Sanso and Nicolas Gailly for helpful comments.

Appendix 1 - B-KEA is malleable

We show that when only B-KEA is used, rogue-key attacks are possible.

Assume we have Alice with keys $A_1 = aH$ and $A_2 = aG$ satisfying $e(A_1, G) * e(-H, A_2) = 1$.

Bob can then choose $B_1 = X_1 - A_1$, $B_2 = X_2 - A_2$, where $X_1 = xH$, $X_2 = xG$. Bob can submit B_1 and B_2 without knowing the secret key and still pass the $e(B_1, G) * e(-H, B_2) = 1$ check.

Now, if Bob can influence the bitmap that goes into the verification function (let's say, by convincing the aggregator) and claim that both he and Alice signed, and he provides a BLS signature with x , the multisignature verification will pass successfully, since $A_1 + B_1 = A_1 + X_1 - A_1 = X_1$.

Appendix 2 - Batching in registration

Similar to how we batched the verification equations in the signature verification, we can batch the registration checks. It's not as critical since this is a one-time operation, but we

present it here for completeness.

Given the following equations.

•

Derive a delination factor α by hashing $(\sigma, m, pk1, pk2)$ and combine the two equations.

$$e(-\sigma, G) * e(H_2(pk_2), pk_2) * (e(pk_1, G) * e(-H, pk_2))^\alpha = 1$$

Grouping elements, we get the following.

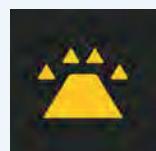
•

Part 2 | Portfolio Company Research



A Mathematical Theory of Danksharding

Yuval Domb
Ingonyama



A Mathematical Theory of Danksharding

Yuval Domb
yuval@ingonyama.com

Abstract

Danksharding is the new sharding design proposed for the Ethereum 2.0 blockchain, which introduces significant simplifications compared to previous designs. In Danksharding, the Beacon block is a periodic data structure, constructed by a Builder, whose primary concern is to enable Validators to verify the correctness and availability of its data, using constant-time sampling and verification. This report provides a mathematical analysis of the Beacon block construction process, including new structural and processing improvements and alternatives.

1 Introduction

Danksharding is central to Ethereum's *rollup-centric* roadmap [1]. The idea is to provide space for large blobs of data, which are verifiable and available, without attempting to interpret them. The blobs data space is expected to be used by layer-2 rollup protocols, supporting high throughput transactions. The main innovation introduced by Danksharding is the *merged-fee-market*, which is enabled by enforcing of a single proposer per slot. Danksharding introduces separation of Proposers and Builders, to avoid high system requirements on Validators. The Builders bid for the right to choose the contents of the slot, while the Proposer selects the highest bidder. Only block Builders need to process the entire block, while Validators and users can download and verify parts or all of the data very efficiently through *data-availability-sampling* [2].

This report is focused on the Beacon block building process. Bearing very high computational complexity, optimization of the block building process is invaluable. This, in our opinion, is best done via thorough understanding of the mathematical models supporting the process. The block building process is broken down to five stages: data organization, coefficient extraction, data interpolation, KZG commitments, and KZG proofs. The processing per stage is described in detail, attempting to be as self-contained as possible. The reader is encouraged to obtain some basic understanding of Discrete-Fourier-Transform (DFT) [3] prior to reading the report.

One outcome of the report is reinforcement of the notion that the basic computationally dominant primitives, required by Danksharding, are highly correlated with ones required for Zero-Knowledge-Proofs. Amongst those primitives are Multi-Scalar-Multiplications (MSM), Number-Theoretic-Transforms (NTT), a new class of NTT - NTT over Elliptic-Curve (EC) group-element vectors (ECNTT), and various lower-lever finite-field arithmetic primitives, such as modular-multiplier, EC-adder etc.

The report structure follows the chronological order of the processing stages, presenting required mathematical tools as needed.

2 Preliminaries

All references to EC, in this report, refer to BLS12_381 [4]. We are mainly concerned with the EC Abelian-group \mathbb{G}_1 and the EC scalar-field \mathbb{F}_r . Both are of size $r < 2^{256}$ with

$$r = 0x73eda753299d7d483339d80809a1d80553bda402fffef5bfefffff00000001$$

Note that 2^{32} divides the size of the multiplicative-group in \mathbb{F}_r . Field-elements and group-elements refer hereafter to elements from \mathbb{F}_r and \mathbb{G}_1 , respectively. A field-element is conveniently described using a 32 bytes unsigned integer smaller than r . Field-addition and field-multiplication refer hereafter to operations in \mathbb{F}_r . Group-addition and scalar-multiplication refer hereafter to addition in \mathbb{G}_1 and multiplication of a group-element in \mathbb{G}_1 by a field-element in \mathbb{F}_r , respectively. We use s , \mathbf{v} , and \mathbf{M} to denote field-element scalars, vectors, and matrices, respectively. We use \mathcal{G} and \mathbf{G} to denote group-element scalars and vectors, respectively.

3 Data Organization

The input data consists of $n = 256$ shard-blobs. Each shard-blob is a vector of $m = 4096$ field-elements referred-to as symbols. The data symbols are organized in an $n \times m$ input matrix

$$\mathbf{D}_{n \times m}^{\text{in}} = \begin{pmatrix} d(0, 0) & d(0, 1) & \dots & d(0, m-1) \\ d(1, 0) & d(1, 1) & \dots & d(1, m-1) \\ \dots & \dots & \dots & \dots \\ d(n-1, 0) & d(n-1, 1) & \dots & d(n-1, m-1) \end{pmatrix} \quad (1)$$

where each row is one of the shard-blob vectors [5].

In order to enable interpolation and polynomial-commitment to the data, we will proceed to treat the data symbols as polynomial evaluations. To that end, the data symbols can be treated as evaluations of a 2D-polynomial, as suggested by the Ethereum foundation. We suggest a simpler, in our opinion, way to satisfy the necessary requirements by treating each data symbol as an evaluation of two 1D-polynomials, corresponding to the row and column of the symbol. This interlaced coding method is typically referred to as a product-code [6]. The proceeding sections analyze both alternatives and their consequences.

As will be presented shortly, DFT enables efficient data processing so long as the data is accessed in cosets. Let us define a data-coset as a batch of data symbols whose domain forms a coset of a multiplicative subgroup. Using appropriately sized roots-of-unity (multiplicative) groups for the data domain allows forming these cosets. Let us thus associate each domain location in the input matrix with a field-element pair (u_μ, w_η) , where $\mu \in [0, n-1]$, $\eta \in [0, m-1]$ correspond to the row and column indexes, respectively. The row field-element is defined as $u_\mu \equiv u^{\text{rbo}(\mu)}$, where u is a $2n$ 'th root-of-unity such that $u^{2n} = 1$. The column field-element is defined as $w_\eta \equiv w^{\text{rbo}(\eta)}$, where w is a $2m$ 'th root-of-unity such that $w^{2m} = 1$. The function $\text{rbo}(\cdot)$ outputs the reversed-bit-order of its input. Examine, for instance, the column domain. The complete domain $\{u^0, u^1, u^2, \dots, u^{2n-1}\}$ forms a group, and appropriately selected subsets such as $\{u^0, u^2, u^4, \dots, u^{2n-2}\}$ form subgroups. A coset is defined as a subgroup translation such as $\{u^1, u^3, u^5, \dots, u^{2n-1}\}$. Using reverse-bit-order ordering rather than natural-ordering allows accessing cosets in block (consecutive) rather

than interleaved manner. This block-wise access will be used with the reverse-bit-ordering assumption implicitly, henceforth.

4 Coefficients Extraction

The size of the input data matrix \mathbf{D}^{in} is $n \times m$. The row (column) domain of the input data is the first n (m) locations in the reverse-bit-ordered, root-of-unity group formed by u (w). As such it forms the domain subgroup corresponding to the n 'th (m 'th) root-of-unity. Taking the data symbols to be evaluations of a 2D-polynomial or 1D-product-polynomials with row degree $n-1$ and column degree $m-1$ uniquely defines the polynomials' coefficients.

4.1 2D Coefficients Extraction

The 2D-polynomial representing the input data can be expressed as

$$d(x, y) = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} \hat{c}[i, j] x^i y^j \quad (2)$$

where x and y correspond to the previously mentioned rows and columns, respectively. The input data evaluations (1), treated as $d(\mu, \eta) = d(u_\mu, w_\eta)$ form a fully-constrained linear system for which the polynomial coefficients can be extracted.

Plugging an evaluation from (1) into (2) results in the following:

$$d(u_\mu, w_\eta) = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} \hat{c}[i, j] u_\mu^i w_\eta^j \quad (3)$$

Defining a scaled version of the coefficients as

$$c[i, j] = \sqrt{nm} \cdot \hat{c}[i, j] \quad (4)$$

leads to the following normalized DFT relationship between polynomial coefficients $c[i, j]$ and evaluations $d(u_\mu, w_\eta)$

$$d(u_\mu, w_\eta) = \frac{1}{\sqrt{nm}} \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} c[i, j] u_\mu^i w_\eta^j \quad (5)$$

$$= \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} \left(\frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} c[i, j] u_\mu^i \right) w_\eta^j \quad (6)$$

$$= \mathcal{F}_m \left\{ \mathcal{F}_n \{c[i, j]\}_{i=0}^{n-1} \right\}_{j=0}^{m-1} |_{(\mu, \eta)} \quad (7)$$

$$= \mathcal{F}_n \left\{ \mathcal{F}_m \{c[i, j]\}_{j=0}^{m-1} \right\}_{i=0}^{n-1} |_{(\mu, \eta)} \quad (8)$$

where \mathcal{F}_m can be represented by right-multiplication by

$$\mathbf{W} = \frac{1}{\sqrt{m}} \begin{pmatrix} 1 & 1 & \dots & 1 \\ 1 & w_1 & \dots & w_1^{m-1} \\ \cdot & \cdot & \dots & \cdot \\ 1 & w_{m-1} & \dots & w_{m-1}^{m-1} \end{pmatrix} \quad (9)$$

and \mathcal{F}_n can be represented by left-multiplication by

$$\mathbf{U} = \frac{1}{\sqrt{n}} \begin{pmatrix} 1 & 1 & \dots & 1 \\ 1 & u_1 & \dots & u_{n-1} \\ \cdot & \cdot & \dots & \cdot \\ 1 & u_1^{n-1} & \dots & u_{n-1}^{n-1} \end{pmatrix} \quad (10)$$

The input data matrix can thus be extracted from the 2D-polynomial coefficients as

$$\mathbf{D}^{\text{in}} = \mathbf{UCW} \quad (11)$$

where

$$\mathbf{C}_{n \times m} = \begin{pmatrix} c[0, 0] & c[0, 1] & \dots & c[0, m-1] \\ c[1, 0] & c[1, 1] & \dots & c[1, m-1] \\ \dots & \dots & \dots & \dots \\ c[n-1, 0] & c[n-1, 1] & \dots & c[n-1, m-1] \end{pmatrix} \quad (12)$$

For the remainder of the document, let us denote the forward DFT relation, going from the coefficient-domain (time-domain) to the evaluation-domain (frequency-domain) as the Number Theoretic Transform (NTT) and its inverse the Inverse NTT (INTT).

The claim that \mathbf{W} and \mathbf{U} are NTT matrices is not completely trivial and requires some clarification, due to the utilized reverse-bit-ordering. For this, with no loss of generality, let us examine \mathbf{W} . Denote by $\widetilde{\mathbf{W}}$ the natural-ordered NTT matrix

$$\widetilde{\mathbf{W}} = \frac{1}{\sqrt{m}} \begin{pmatrix} 1 & 1 & \dots & 1 \\ 1 & w & \dots & w^{m-1} \\ \cdot & \cdot & \dots & \cdot \\ 1 & w^{m-1} & \dots & w^{(m-1)^2} \end{pmatrix} \quad (13)$$

and its corresponding INTT matrix

$$\widetilde{\mathbf{W}}^{-1} = \widetilde{\mathbf{W}}^H = \frac{1}{\sqrt{m}} \begin{pmatrix} 1 & 1 & \dots & 1 \\ 1 & w^{-1} & \dots & w^{-(m-1)} \\ \cdot & \cdot & \dots & \cdot \\ 1 & w^{-(m-1)} & \dots & w^{-(m-1)^2} \end{pmatrix} \quad (14)$$

where H denotes the conjugate transpose operator¹. It's easy to verify that $\widetilde{\mathbf{W}}$ is unitary since $|\widetilde{\mathbf{W}}| = 1$ and $\widetilde{\mathbf{W}}^H \widetilde{\mathbf{W}} = \mathbf{I}$ where \mathbf{I} is the identity matrix, hence it is a valid NTT. The reverse-bit-ordering can be defined as a permutation matrix \mathbf{P} therefore

$$\mathbf{W} = \mathbf{P} \widetilde{\mathbf{W}} \quad (15)$$

and thus \mathbf{W} is also unitary and a valid NTT.

Extraction of the 2D-polynomial coefficients, based on (11), can thus be described as

$$\mathbf{C} = \mathbf{U}^H \mathbf{D}^{\text{in}} \mathbf{W}^H \quad (16)$$

Note that (16) is equivalent to performing INTT over the rows and then INTT over the columns of the result, or visa-versa.

¹Conjugate for finite field-elements is defined in the usual way as w^* s.t. $ww^* = w^*w = |w|^2$.

4.2 1D Coefficients Extraction

The 1D row and column polynomials representing the input data $d(u_\mu, w_\eta)$ can be depicted as

$$d_\mu^{\text{row}}(w_\eta) = \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} c_\mu^{\text{row}}[j] w_\eta^j \quad (17)$$

$$d_\eta^{\text{col}}(u_\mu) = \frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} c_\eta^{\text{col}}[i] u_\mu^i \quad (18)$$

where $d_\mu^{\text{row}}(w_\eta) = d_\eta^{\text{col}}(u_\mu) = d(u_\mu, w_\eta)$. The total number of product-code polynomials is $n + m$. The input data matrix can thus be extracted from the 1D-polynomials as

$$\mathbf{D}^{\text{in}} = \mathbf{C}^{\text{rows}} \mathbf{W} = \mathbf{U} \mathbf{C}^{\text{cols}} \quad (19)$$

which immediately leads to

$$\mathbf{C}^{\text{rows}} = \mathbf{D}^{\text{in}} \mathbf{W}^H \quad (20)$$

$$\mathbf{C}^{\text{cols}} = \mathbf{U}^H \mathbf{D}^{\text{in}} \quad (21)$$

where the rows of \mathbf{C}^{rows} are the n row product-codes, and the columns of \mathbf{C}^{cols} are the m column product-codes. Note that the size of the 1D coefficient representation is double the size of the 2D representation.

5 Data Interpolation

The full (interpolated) data matrix can be described as the concatenation of four sub-matrices as follows:

$$\mathbf{D} = \begin{pmatrix} \mathbf{D}_{n \times m}^{\text{in}} & \mathbf{D}_{n \times m}^{\text{rows}} \\ \mathbf{D}_{n \times m}^{\text{cols}} & \mathbf{D}_{n \times m}^{\text{both}} \end{pmatrix} \quad (22)$$

5.1 2D Data Interpolation

The 2D-interpolation is achieved by

$$\mathbf{D} = \mathcal{F}_{2n} \left\{ \mathcal{F}_{2m} \{c[i, j]\}_{j=0}^{2m-1} \right\}_{i=0}^{2n-1} \quad (23)$$

where $c[i, j] \equiv 0$, $\forall i \geq n, j \geq m$, \mathcal{F}_{2m} is a length $2m$ row-wise NTT, and \mathcal{F}_{2n} is a length $2n$ column-wise NTT.

A more efficient way to achieve this is by noting that the interpolated evaluations are a time-shifted version of the original evaluations which translates to frequency-modulation in the coefficient-domain. This immediately translates to the following scheme

$$\mathbf{D}^{\text{rows}} = \mathbf{U} \mathbf{C} \Lambda_w \mathbf{W} \quad (24)$$

$$\mathbf{D}^{\text{cols}} = \mathbf{U} \Lambda_u \mathbf{C} \mathbf{W} \quad (25)$$

$$\mathbf{D}^{\text{both}} = \mathbf{U} \Lambda_u \mathbf{C} \Lambda_w \mathbf{W} \quad (26)$$

where $\Lambda_w = \text{diag}([1 w w^2 \dots w^{m-1}])$ and $\Lambda_u = \text{diag}([1 u u^2 \dots u^{n-1}])$ are the frequency-modulation matrices.

5.2 1D Data Interpolation

Similar arguments translates to the following scheme for the 1D case

$$\mathbf{D}^{\text{rows}} = \mathbf{C}^{\text{rows}} \Lambda_w \mathbf{W} \quad (27)$$

$$\mathbf{D}^{\text{cols}} = \mathbf{U} \Lambda_u \mathbf{C}^{\text{cols}} \quad (28)$$

$$\mathbf{D}^{\text{both}} = \mathbf{U} \Lambda_u \mathbf{U}^H \mathbf{D}^{\text{rows}} = \mathbf{D}^{\text{cols}} \mathbf{W}^H \Lambda_w \mathbf{W} \quad (29)$$

6 KZG Commitments

From this stage and on, each row of the interpolated data matrix \mathbf{D} is treated as evaluations of a 1D-polynomial of degree $m - 1$, as was done previously for the 1D case, namely $d_\mu(x)$. This allows us to reuse the row-polynomials \mathbf{C}^{rows} from (20). For the 2D case, one can calculate an equivalent as $\mathbf{C}^{\text{rows}} = \mathbf{U}\mathbf{C}$. This conclusion is easily reached from simple comparison of (11) and (19).

Each row-polynomial is committed to individually. The commitment scheme is polynomial-KZG [7] with a single predetermined setup of length m

$$[1], [s], \dots, [s^{m-2}], [s^{m-1}] \quad (30)$$

where $[s^k] \equiv s^k \cdot \mathcal{G}$ for some secret field-element s , and generator group-element \mathcal{G} . In general, we use the form $[f(s)]$ to denote the evaluation of a polynomial $f(x) = \sum_i f_i x^i$ at group-element $[s]$, i.e. $[f(s)] = \sum_i f_i [s^i]$. A commitment to row $\mu \in [0, 2n - 1]$ is an evaluation of its polynomial $d_\mu(x)$ at $[s]$, i.e. $\mathcal{K}_\mu = [d_\mu(s)]$. Commitments for rows $[0, n - 1]$ are calculated according to

$$\mathcal{K}_{0:n-1} = \mathbf{C}^{\text{rows}}[\mathbf{s}] \quad (31)$$

where $[\mathbf{s}] \equiv [[1], [s], \dots, [s^{m-2}], [s^{m-1}]]^T$ and T is the transpose operator. Note that the above is equivalent to n, m -long Multi-Scalar Multiplications (MSM). Commitments for rows $[n, 2n - 1]$ can be interpolated following (29) as

$$\mathcal{K}_{n:2n-1} = \mathbf{U} \Lambda_u \mathbf{U}^H \mathcal{K}_{0:n-1} \quad (32)$$

The implied NTTs in (32) are performed over EC group-element vectors and are termed ECNTT². The interpolation is valid due to linearity. More specifically, linearity applies since \mathbb{F}_r and \mathbb{G}_1 are a linear bijection, i.e. $[s_0 + s_1] = [s_0] + [s_1]$ and $|\mathbb{F}_r| = |\mathbb{G}_1|$.

7 KZG Proofs

The proofs are Multi-reveal KZG proofs (MKZG) following the scheme from [8]. Each proof is constructed for a sample of $l = 16$ data symbols from \mathbf{D} . Each sample corresponds to polynomial evaluations over a coset of the subgroup $[1, \psi, \psi^2, \dots, \psi^{l-1}]$, where ψ is an

²Note that optimization of ECNTT is different from NTT since field-multiplications are traded for scalar-multiplications whose computational complexity is typically proportional to $\log_2(s)$ where s is the scalar.

l 'th root-of-unity and each sample is taken from a distinct row $\mu \in [0, 2n - 1]$. Each sample proof is constructed to prove that

$$\begin{pmatrix} d_\mu(\phi^k) \\ d_\mu(\phi^k\psi) \\ \vdots \\ d_\mu(\phi^k\psi^{l-1}) \end{pmatrix} = \begin{pmatrix} z_0 \\ z_1 \\ \vdots \\ z_{l-1} \end{pmatrix} \quad (33)$$

where ϕ is a $\frac{2m}{l}$ 'th root-of-unity, and $k \in [0, \frac{2m}{l} - 1]$. To this end, we compute the quotient polynomial

$$q_{\mu,k}(x) = \frac{d_\mu(x) - r_{\mu,k}(x)}{x^l - \phi^{kl}} \quad (34)$$

where $r_{\mu,k}(x) = d_\mu(x) \bmod (x^l - \phi^{kl})$ is the remainder polynomial which is the unique polynomial of degree $l - 1$ that obeys (34) (see [9] for a direct construction method), and $(x^l - \phi^{kl}) = (x - \phi^k)(x - \phi^k\psi) \cdots (x - \phi^k\psi^{l-1})$ by definition of a roots-of-unity group [10]. The MKZG proof is simply an evaluation of the quotient polynomial at $[s]$, i.e. $\Pi_{\mu,k} = [q_{\mu,k}(s)]$. Verification is achieved using

$$e(\mathcal{K}_\mu - [r_{\mu,k}(s)], [1]) = e(\Pi_{\mu,k}, [s^l] - [\phi^{kl}]) \quad (35)$$

where $e(\cdot, \cdot)$ is a paring operator.

Equation (34) can be rewritten as

$$q_{\mu,k}(x) = \frac{d_\mu(x) - r_{\mu,k}(x) + \phi^{kl}q_{\mu,k}(x)}{x^l} \quad (36)$$

$$= \left\lfloor \frac{d_\mu(x) + \phi^{kl}q_{\mu,k}(x)}{x^l} \right\rfloor \quad (37)$$

$$= \left\lfloor \frac{d_\mu(x)}{x^l} \right\rfloor + \left\lfloor \frac{d_\mu(x)}{x^{2l}} \right\rfloor \phi^{kl} + \left\lfloor \frac{d_\mu(x)}{x^{3l}} \right\rfloor \phi^{2kl} + \cdots + \left\lfloor \frac{d_\mu(x)}{x^{(\frac{m}{l}-1)l}} \right\rfloor \phi^{(\frac{m}{l}-2)kl} \quad (38)$$

where the floor operator signifies truncated division, i.e. terms with negative exponents x^i , $\forall i < 0$ are discarded. This can be presented more compactly in vector form as

$$q_{\mu,k}(x) = \boldsymbol{\phi}_k^T \cdot \mathbf{d}_\mu(x) \quad (39)$$

$$\boldsymbol{\phi}_k^T = \left[1, \phi^{kl}, \phi^{2kl}, \dots, \phi^{(\frac{m}{l}-2)kl}, \phi^{(\frac{m}{l}-1)kl} \right] \quad (40)$$

$$\mathbf{d}_\mu(x) = \left[\left\lfloor \frac{d_\mu(x)}{x^l} \right\rfloor, \left\lfloor \frac{d_\mu(x)}{x^{2l}} \right\rfloor, \left\lfloor \frac{d_\mu(x)}{x^{3l}} \right\rfloor, \dots, \left\lfloor \frac{d_\mu(x)}{x^{(\frac{m}{l}-1)l}} \right\rfloor, 0 \right]^T \quad (41)$$

where we extend both vectors to length $\frac{m}{l}$, adding one redundant element for convenient NTT sizing. The last two required operations are described in the following subsections.

7.1 Calculating $[\mathbf{d}_\mu(s)]$

Evaluating (41) at $[s]$ (i.e. $[\mathbf{d}_\mu(s)]$) can be done as follows:

$$[\mathbf{d}_\mu(s)] = \left[\sum_{j=tl}^{m-1} c_\mu^{\text{row}}[j][s^{j-tl}] \right]_{t \in [1, \frac{m}{l}]} \quad (42)$$

where $\mathbf{c}_\mu^{\text{row}}$ is row- μ of \mathbf{C}^{rows} . One method by which this can be achieved is by performing $(\frac{m}{l} - 1)$ MSMs of varying sizes (note that the last element in the vector does not need to be calculated and is by definition the EC point-at-infinity).

Another method is as follows. We start by noting that equation (42) is an l -downsample of the output of a linear convolution. The linear convolution can be achieved by element-wise multiplication in the evaluations-domain. The coefficient-domain sequences ($\mathbf{c}_\mu^{\text{row}}$ and $[\mathbf{s}]$) must be zero padded prior to the NTT in order that the cyclic-convolution nature of the NTT does not alias the desired linear-convolution onto itself. Zero-padding to length $2m$ is sufficient. The frequency-domain of the coefficients is already available as row μ of \mathbf{D} . Since the setup does not change, its interpolated frequency-domain \mathbf{S} can be pre-calculated as

$$\mathbf{S}_{0:m-1} = \overleftarrow{[\mathbf{s}]}^T \mathbf{W} \quad (43)$$

$$\mathbf{S}_{m:2m-1} = \overleftarrow{[\mathbf{s}]}^T \Lambda_w \mathbf{W} \quad (44)$$

where $\overleftarrow{[\mathbf{s}]}$ is the reversed-order vector of $[\mathbf{s}]$. This is required since the r.h.s. of (42) is technically a correlation and not a convolution. The two frequency-domain vectors are element-wise multiplied and l -fold aliased as

$$[\delta_\mu(s)] = \left[\frac{1}{l} \sum_{j=0}^{l-1} d_\mu^{\text{row}} \left[j \frac{m}{l} + t \right] \mathbf{S} \left[j \frac{m}{l} + t \right] \right]_{t \in [0, \frac{m}{l} - 1]} \quad (45)$$

where the l -fold aliasing is the evaluation-domain equivalent of the l -downsampling in the coefficient-domain. Finally, $[\mathbf{d}_\mu(s)]$ is the $\frac{m}{l}$ -size INTT of $[\delta_\mu(s)]$.

7.2 Calculating Row Proofs $[\mathbf{q}_\mu(s)]$

Going back to (39), note that all row- μ proofs

$$[\mathbf{q}_\mu(s)] \equiv \left[[q_{\mu,0}(s)], [q_{\mu,1}(s)], \dots, [q_{\mu,\frac{2m}{l}-1}(s)] \right] \quad (46)$$

can be calculated by a $\frac{2m}{l}$ -size NTT, since ϕ_k^T is the NTT vector for frequency ϕ^k .

Interpolation tricks, similar to the ones used before, can be utilized to calculate only the first $\frac{m}{l}$ proofs directly and interpolate the rest.

Extending the proofs for all rows can be done directly by repeating this process for all $\mu \in [0, 2n - 1]$ or again by calculating the first n and interpolating the rest.

Acknowledgement

The author would like to thank Karthik Inbasekar, Dmytro Tymokhanov, and Omer Shلومוט for helpful discussions.

References

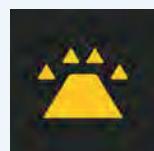
- [1] Vitalik Buterin. Proto-danksharding faq. [https://notes.ethereum.org/@vbuterin/proto_danksharding_faq#:~:text=is%20just%20data\).-,What%](https://notes.ethereum.org/@vbuterin/proto_danksharding_faq#:~:text=is%20just%20data).-,What%)

20is%20proto%2Ddanksharding%20(aka., yet%20actually%20implementing%
20any%20sharding.

- [2] Vitalik Buterin. An explanation of the sharding + das proposal. https://hackmd.io/@vbuterin/sharding_proposal.
- [3] Discrete fourier transform. https://en.wikipedia.org/wiki/Discrete_Fourier_transform.
- [4] Ben Edgington. Bls12-381 for the rest of us. <https://hackmd.io/@benjaminion/bls12-381>.
- [5] Dankrad Feist. Data availability encoding. https://notes.ethereum.org/@dankrad/danksharding_encoding.
- [6] E. R. Berlekamp. *Algebraic coding theory*. Aegean Park Press, Laguna Hills, CA, USA, 1984.
- [7] Dankrad Feist. Kzg polynomial commitments. <https://dankradfeist.de/ethereum/2020/06/16/kate-polynomial-commitments.html>, 2020.
- [8] Dankrad Feist and Dmitry Khovratovich. Fast amortized kate proofs. https://github.com/khovratovich/Kate/blob/master/Kate_amortized.pdf, 2020.
- [9] Vitalik Buterin. Quadratic arithmetic programs: from zero to hero. <https://medium.com/@VitalikButerin/quadratic-arithmetic-programs-from-zero-to-hero-f6d558cea649>, 2016.
- [10] Ray Li. Roots of unity. <http://theory.stanford.edu/~rayyli/static/contest/lectures/Ray%20Li%20rootsofunity.pdf>, 2021.

Fast Modular Multiplication

Yuval Domb
Ingonyama



Fast Modular Multiplication

Yuval Domb
yuval@ingonyama.com

July 25, 2022 (version 1.0)

Abstract

Modular multiplication is arguably the most computationally-dominant arithmetic primitive in any cryptographic system. This note presents an efficient, hardware-friendly algorithm that, to the best of the author's knowledge, outperforms existing algorithms to date.

1 Introduction

The standard modulo-prime multiplication problem in \mathbb{F}_s can be cast as

$$r = a \cdot b \mod s \tag{1}$$

where $a, b, r \in \mathbb{F}_s$, s is prime, and the standard \mathbb{Z} -algebra is utilized. Equivalently this can be written as

$$a \cdot b = l \cdot s + r \tag{2}$$

with $l \in \mathbb{Z}$ such that $0 \leq r < s$.

The purpose of this note is to provide an efficient, hardware-friendly method for fast computation of (1).

Assume that all variables are represented by d -radix digits, and denote by n the required number of digits used to represent any element in \mathbb{F}_s , thus

$$n = \lceil \log_d s \rceil \tag{3}$$

For simplicity, let us set $d = 2$ and use bits in place of digits, for the remainder of the note.

Finally, although this note is primarily concerned with modulo-prime multiplication, its results can be generalized to any case of $a \mod s$ where $a < s^2$ for any s , prime or not.

2 Contribution

The main contribution of this note is to show how Barrett's Reduction [1], together with good parameter selection and a simple bounding technique, can be used to approximate the quotient l up to a small constant error independent of n , for any n . Surprisingly, the resulting reduction algorithm closely resembles Montgomery's Modular-Multiplication algorithm [2], without the coordinate translation requirement. This bounding technique can be used to further lower the calculation complexity of special cases of interest, not presented here, at the price of increased constant error.

3 Reduction Scheme

3.1 Assume l is approximately known

Assume that l is approximately known and denote by \hat{l} its approximation, such that

$$l - \lambda \leq \hat{l} \leq l \quad (4)$$

where $\lambda = O(1)$ is a known constant.

Starting initially with $\lambda = 0$, it is clear that

$$ab[2n - 1 : 0] - \hat{ls}[2n - 1 : 0] = r[n - 1 : 0] \quad (5)$$

where the brackets denote bit locations and sizes. Note that $\lambda = 0$ means that we know apriori that the remainder is at most n bits in length, so that all remaining most-significant (ms) bits in the rhs of (5) must be zero.

Utilizing simple bit manipulation, this can be cast as the following long addition

$$\begin{array}{r} \overline{\hat{ls}[2n - 1]} \dots \overline{\hat{ls}[n]} \overline{\hat{ls}[n - 1]} \dots \overline{\hat{ls}[0]} \\ + ab[2n - 1] \dots ab[n] ab[n - 1] \dots ab[0] \\ \hline 0 \dots 0 r[n - 1] \dots r[0] \end{array}$$

where the over-line denotes the bit-inversion operator and the 1-bit at the top-right is treated as an initial carry bit. The important insight is that only $ab[n - 1 : 0]$ and $\hat{ls}[n - 1 : 0]$ are necessary in order to complete this calculation, which immediately saves approximately half of the calculations. Note that the resulting adder is a *fixed width adder*, (i.e. $n + n \rightarrow n$). This means that any overflow ms bits must be ignored. An equivalent alternative to the above is a fixed-width subtractor ($n - n \rightarrow n$), where the result is treated as an unsigned integer.

Let us denote the type of multiplier required to generate the above products as an $n \times n \rightarrow n_{\text{lsb}}$ multiplier, where n_{lsb} refers to the n least-significant bits of the full product. This means that the products $a \cdot b$ and $\hat{l} \cdot s$ can be generated using $n \times n \rightarrow n_{\text{lsb}}$ multipliers. In addition, if s is constant, $\hat{l} \cdot s$ can be generated using a constant $n \times n \rightarrow n_{\text{lsb}}$ multiplier.

Finally when $\lambda \neq 0$

$$ab - \hat{ls} = r + \lambda s \quad (6)$$

and the number of bits required to represent the rhs of (5) is

$$\lceil \log_2(r + \lambda s) \rceil \leq n + \left\lceil \log_2 \frac{r + \lambda s}{s} \right\rceil \leq n + \lceil \log_2(1 + \lambda) \rceil \quad (7)$$

so if $\lambda = 1$ the total number of additional bits required would be 1.

3.2 Use Barrett's Reduction to approximate l

Barrett's modular reduction approximates l as follows

$$l = \left\lfloor \frac{ab}{s} \right\rfloor = \lim_{k \rightarrow \infty} \frac{ab \cdot m(k)}{2^{k+n}} \quad (8)$$

where

$$m(k) = \left\lfloor \frac{2^{k+n}}{s} \right\rfloor < 2^{k+1} \quad (9)$$

is a function of the k , representing the maximal, $k + 1$ bits, lower-bound approximator for (8). For finite k , the approximation error is

$$e(k) \equiv \frac{1}{s} - \frac{m(k)}{2^{k+n}} < 2^{-(k+n)} \quad (10)$$

where the upper-bound can be derived by examining the maximal difference between the binary representation of the left and right terms. This immediately leads to the approximation error on $l(k)$

$$e(l, k) \equiv \frac{ab}{s} - \frac{ab \cdot m(k)}{2^{k+n}} < 2^{2n} \cdot 2^{-(k+n)} = 2^{n-k} \quad (11)$$

Thus if $k \geq n$ the approximation error is at most 1.

3.3 Choose the parameters and bound the error

Choosing $k = n$ (i.e. $m(n) < 2^{n+1}$) leads to the following approximation on l

$$\hat{l}_0 = \left\lfloor \frac{abm}{2^{2n}} \right\rfloor \quad (12)$$

$$e(\hat{l}_0) < 1 \quad (13)$$

where the multiplication is $n \times n \times (n + 1) \rightarrow (n + 1)_{\text{msb}}$, and the approximation error obeys (11).

Let us instead perform the above multiplication in two stages. Initially, assume that $ab[2n - 1 : 0]$ is available and perform the multiplication as follows

$$\frac{abm}{2^{2n}} = \frac{ab[2n - 1 : n] \cdot m}{2^n} + \frac{ab[n - 1 : 0] \cdot m}{2^{2n}} \quad (14)$$

$$< \frac{ab[2n - 1 : n] \cdot m}{2^n} + 2 \quad (15)$$

where the right-most term is trivially upper-bounded by 2. This immediately leads to the following approximation on l

$$\hat{l}_1 = \left\lfloor \left\lfloor \frac{ab}{2^n} \right\rfloor \cdot \frac{m}{2^n} \right\rfloor \quad (16)$$

$$e(\hat{l}_1) < 3 \quad (17)$$

where the inner multiplication is $n \times n \rightarrow n_{\text{msb}}$, the outer (constant) multiplication is $n \times (n + 1) \rightarrow (n + 1)_{\text{msb}}$, and the approximation error is upper-bounded by the sum of (13) and the right-most term of (15). Note that since $m(n)$ is typically very close to 2^n and n is typically large, no additional bits need to be added (i.e. $n + 1$ bits suffice) to represent the constant error (17). Nonetheless, this overflow corner-case must be examined and ruled out per a given setup.

3.4 Putting it all together

Below is a block diagram of the hardware-optimized modular multiplier. The diagram assumes that s and m are known constants, and uses the \hat{l}_1 approximation for l .

Note that the left-most multiplication module is independent of the reduction logic, allowing the remainder of the circuit to be generalized beyond multiplication reductions.

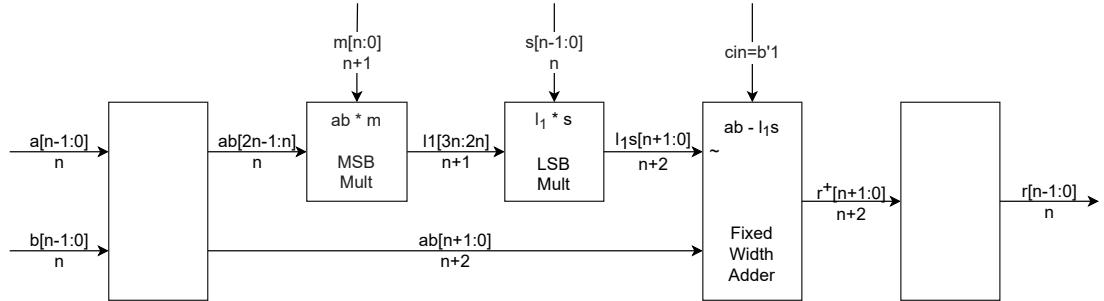


Figure 1: Hardware-optimized modular multiplier

3.5 Examples

3.5.1 Example for n=16

```

— params —
n = 16
s[n-1:0] = 65521
m[n:0] = 65551
a[n-1:0] = 64111
b[n-1:0] = 11195

— direct calc —
l[n:0] = 10954
r[n-1:0] = 5611
5611 = 64111*11195 - 10954*65521

— a*b full mult —
ab[2n-1:0] = 717722645
ab[2n-1:n] = 10951
ab[n+1:0] = 234517

— ab*m msb mult —
l1[3n:2n] = 10953
e(l1) = 1

— l1*s lsb mult —
l1s[n+1:0] = 163385

— ab-l1s fixed width adder —
r+[n+1:0] = 71132

```

```
— subtract redundant s —
r_hat = 5611
```

3.5.2 Example for n=32

```
— params —
n = 32
s[n-1:0] = 4294967291
m[n:0] = 4294967301
a[n-1:0] = 1152833672
b[n-1:0] = 2546222476

— direct calc —
l[n:0] = 683444321
r[n-1:0] = 2821307461
2821307461 = 1152833672*2546222476 - 683444321*4294967291

— a*b full mult —
ab[2n-1:0] = 2935371006736011872
ab[2n-1:n] = 683444320
ab[n+1:0] = 3699053152

— ab*m msb mult —
l1[3n:2n] = 683444320
e(l1) = 1

— l1*s lsb mult —
l1s[n+1:0] = 13762647584

— ab-l1s fixed width adder —
r+[n+1:0] = 7116274752

— subtract redundant s —
r_hat = 2821307461
```

3.5.3 An exotic example

If $s = 65717$ and $a = 65535, b = 65631$, we get that the real value of l is $\lfloor \frac{ab}{s} \rfloor = 65449$. On the other hand, our approximation gives $\hat{l}_1 = 65446$, and so the error $e(\hat{l}_1)$ is 3 in this case. However, primes s for which such examples exist are sparse and for most primes, the largest possible error will not exceed 2.

References

- [1] Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In *Advances in Cryptology — CRYPTO' 86*, pages 311–323, 1987.
- [2] Peter L. Montgomery. Modular multiplication without trial division. In *Mathematics of Computation*, volume 44, pages 519–521, 1985.

SoK: Hash functions in Zero Knowledge Proofs

Karthik Inbasekar
Ingonyama



SoK: Hash functions in Zero Knowledge Proofs

Karthik Inbasekar

karthik@ingonyama.com

Abstract: In this article we present a systematization of knowledge (SoK) on the usage of hash functions in Zero Knowledge Proofs (ZKP's). Since ZKP's operate on finite fields, traditional time-tested hash functions like SHA256 are unsuitable due to overhead in proving and verification of large number of bitwise operations. ZK-friendly hash functions are optimized for modular arithmetic in a finite field which reduces the circuit complexity significantly. This has led to widespread deployment of these hash functions in production-level systems. That said, the ZK-friendly hash functions are relatively newer than the traditional hash functions, and are yet to undergo the test of time. Our main goal is to survey such ZK-friendly hash functions, and highlight their general use cases in ZKP's.

Contents

1	Introduction	1
2	Hashes in ZKP: A gentle introduction	3
2.1	Universal properties	3
2.2	Permutation function	5
3	Use cases	8
3.1	Merkle trees and commitments	8
3.2	Fiat-Shamir	9
3.3	Poseidon: Filecoin	10
3.4	Rescue: ethSTARK	12
4	Overview of ZK-friendly hashes in other applications	15
5	Closing Words and a Disclaimer	16

1 Introduction

A Zero Knowledge Proof (ZKP) is a protocol between a Prover and a verifier, which allows a prover to convince a verifier that a certain statement is true without revealing anything more than the truth of the statement itself. The statement takes the form of a NP relation $\mathcal{R}(x, w) : y = F(x, w)$ where x is a public input and w is a private input. The ZKP ensures that a verifier will be convinced by the proof of a honest prover who knows w , and learns nothing about w itself. While simultaneously ensuring that, a malicious prover who does not know w will be unable to convince the verifier with very high probability.

The NP relation is typically expressed as an arithmetic circuit, and there are several arithmetization programs available such as Rank One Constraint System (R1CS) [1, 2],

Plonk Arithmetization [3] and AIR (Algebraic Intermediate Representation) arithmetization [4]. Arithmetic gates form the atomic structure of the circuits. Depending on the arithmetization, the spectrum of gates design in a circuit goes from simple addition or multiplication gates to a custom gate capable of performing compound operations. The input and output values of the gates are referred to as wire values. In ZKP, these wire values are elements in a finite field \mathbb{F} (usually a large prime) and thus the fundamental computational primitive is modular arithmetic over a finite field.

Hash functions are versatile cryptographic primitives that are crucial in the design and functionality of many ZKP's. Many well known hash functions such as SHA2, SHA3, BLAKE are designed such that hashes can be computed blazingly fast on traditional CPUs. This is one of the reasons why these hash functions contain a lot of bitwise operations. While these are collision resistant and known to be secure, they are unsuitable for ZKP. This is because the prover will have to compute a lot of bitwise operations in a circuit¹ such as (1.1), and the verifier will have to check every bit with the solution, since the statement is only satisfied if all the computed bits are agreed upon (See for instance the SHA256 example in [5]). Thus, representing bit-wise operations of large bit-size numbers leads to large circuit sizes, which in turn leads to large polynomial degrees in the problem arithmetization. For instance the Discrete Fourier Transform complexity increases with the polynomial degree and directly affects the execution time of the prover. We caution the reader that the ZK space is moving fast, and this picture might very well change with the new developments in ZK that are based on lookup tables [6–8]. So far there are no production ready systems that efficiently employ lookup tables. For the purpose of this article we consider ZK-friendly hash functions that are natively defined over finite fields, have preimage resistance (one way property) and collision resistance. Below we summarize some of the use cases of hash functions in ZKP's.

- Cryptographic hash functions naturally appear in ZKP as the NP relation. The circuit could be made of a sequence of hash functions

$$H(\dots H(H(w_0, w_1), w_2), w_3) \dots, w_{r-1}) = y \quad (1.1)$$

where the w_i 's are private inputs of the prover while H and y are known and agreed upon by the prover and the verifier. The prover will claim knowledge of the preimage w_i , execute the circuit, and provides the verifier a cryptographic proof for computational integrity of the execution. A well known example is to provide proof of membership of a leaf in a Merkle tree. Generally, the hash function is represented as a circuit over a (large) prime field, and thus proving the knowledge of the preimage is computationally intensive. The main challenge is to find efficient representations of hash function in the circuit, to minimize arithmetic complexity such that it results in low degree polynomials and minimizes the number of constraints. (see for example [9]). These considerations lead to the search for “arithmetization oriented” hash functions where the efficiency efforts are primarily focused on optimization of algebraic

¹Intuitively, the reason why bitwise operations are inefficient with arithmetic circuits is because a single AND or XOR operation would need one multiplication or addition gate. On the other hand, the same multiplication or addition gate is capable of multiplying or adding two very large numbers.

complexity as described in R1CS/AIR/Plonk constraints for ZKPs, minimizing the prover’s computation. Thus unlike traditional hashes, the ZK friendly hashes are less focused on minimizing execution time of the hash operation itself.

- In addition, Hash functions are used as a building block of the cryptographic commitments based on Merkle trees, where the leaves of the Merkle tree are elements of some vector that one wants to commit to. The main computation in this case is a sequence of hashes, built in the form of a tree, with the root of the tree being the commitment to original data (see §3.1).
- Furthermore, Hash functions are also used as in the conventional way, e.g. while signing transactions or for authenticated encryption of fixed length messages or to generate pseudo-random field elements based on a seed. These predefined fixed-length sequence of inputs allow hash functions to instantiate a random oracle for Fiat-Shamir transform, removing interactivity between the prover and verifier (see §3.2).

Some of the finite field friendly hash functions that are used in the ZK space are listed in table 1. Our main goal in this article is to provide a survey of the ZK-friendly hash functions, and give some intuition about their use cases in ZKPs. In particular, we will defer the security and efficiency analysis of the hashes involved to a future article, since it deserves a more thorough technical treatment. The article is organized as follows. In §2 we start off with some universal properties of hash functions specific to ZK §2.1. In §2.2 we review the sponge construction and permutation which we believe are universal building blocks for ZK friendly hashes. In §3 we discuss some common use cases, including Merkle trees §3.1, and Fiat-Shamir heuristic for non-interactive protocols §3.2. Finally in §3.3, §3.4 we discuss two robust sponge constructions based on the HADES [28] and MARVELlous [29] designs, namely Poseidon hash [10] in filecoin [30] and Rescue hash [19] in ethSTARK [4]. Finally, in §4 we conclude with a high level overview of hash functions used in some of the Polygon ecosystems: Plonky2 [16], zkEVM [18] and Miden VM [20].

2 Hashes in ZKP: A gentle introduction

2.1 Universal properties

Generally speaking, hash functions in ZKPs operate on finite field elements. We observe the following characteristics in ZK-friendly hash functions:

- Natively defined to operate on elements in a finite field \mathbb{F} .
- Equipped with only addition, multiplication operations and potentially look-ups.
- Low arithmetic complexity (constraints, polynomial degrees)
- If the application requires b bit security, then the hash function must also be at least b bits secure. Furthermore, the usual properties of hash functions continue to hold.

Hash function	Application	Use case
Poseidon [10]	Filecoin [11]	Vanilla Hash computation Octinary/Binary Merkle Tree computation Circuit: R1CS arithmetization
	Mina (Kimchi)* [12]	Turboplonk arithmetization, Fiat-Shamir
	Scroll tech [13, 14]	Plonkish Arithmetization (Halo2) Fiat-Shamir (Aggregator circuit)
Poseidon377	Penumbra* [15]	-
Poseidon-Goldilocks	Plonky2 [16]	Turboplonk arithmetization Merkle commitment, FRI-IOP [17]
	Polygon-zkEVM [18]	AIR arithmetization, FRI-IOP
Rescue [4], Rescue-Prime [19]	ethSTARK [4]	AIR arithmetization Merkle commitment, FRI-IOP
	Polygon MidenVM [20]	Vanilla hash computation Merkle tree, AIR arithmetization
Sinsemilla [21]	ZCash* [22]	Merkle Commitments, Lookup tables
Pedersen hash		Commitments, Merkle tree
Anemoi [23]	-	Optimized for Merkle trees (Anemoi-Jive)
Reinforced concrete [24]	-	Specialized for Look up tables
MIMC [25]	-	-
Grendel [26]		
Griffin [27]		

Table 1. Finite field friendly Hash functions in ZK space. The * refers to usage in sub-systems not in production at the time of writing of this paper.

- **One way property:** Also called pre-image resistance, roughly defined such that given the output of a Hash $h = H(x)$ it should be computationally infeasible to invert the function and determine the pre-image x .
- **Collision resistance:** It should be computationally infeasible to find two different x 's that generate the same h .

All the hash functions listed in table 1 are based on the sponge construction [31]. Although the subject of hashes is by itself very rich and diverse, the specific requirements of ZKP's that we discussed in the introduction seems most suited for Substitution Permutation Networks (SPN's) based on the sponge construction. The sponge is an iterative construction that takes in arbitrary input/output sizes but applies a fixed length permutation f . The functional form of f can vary across hashes. However, typically f consists of some linear layers that include element-wise addition and matrix multiplication, and non-linear layers

that consists of power maps. Without going into too much detail, the power maps are designed to increase the degree of the elements, while the linear layer spreads them uniformly across \mathbb{F} .

The sponge permutation has two parameters: the rate r and capacity c , that set the security level to be at least:

$$S = \log_2(\sqrt{p}) \min(r, c) \quad (2.1)$$

for preimage resistance and collision resistance, where p is the prime modulus. The instance of a sponge is specified by the triplet (t, p, S) . Consider a tuple of field elements $t = r + c \in \mathbb{F}_p^{r+c}$, where r is the rate and c the capacity, while p is the prime modulus. The sponge permutation takes a tuple of size t , applies a series of fixed length $|t|$ permutations and outputs a digest r elements in \mathbb{F}^r . The rate r affects the throughput, and the capacity should be chosen such that (2.1) is met for the application in question. In fig 1, we quote the general definition of a sponge. In general the input message could be much larger than the permutation width $|t|$, in fig 1 the input message $M = [m_1||m_2||m_3||\dots]$ is padded such that $|m_i| = r$. Then after each permutation, a part of the message M i.e m_i is “absorbed” into the permutation function. The initial state is chosen to be $I = [0^r||0^c]$. The sponge has two phases, absorb and squeeze.

- In the absorb phase, the f function is applied on the input $t = [m_i \oplus r_i || c_i]$. The permute block f outputs t elements which undergo further permutation after addition with the next message: $[m_{i+1} \oplus r_{i+1} || c_{i+1}]$, and so on. Once all m_i are exhausted, the sponge design switches to squeeze phase.
- The output at the end of the absorb phase has already reached the uniform distribution. In the squeeze phase, the user can determine the number of output blocks, and for each block the first r elements are outputed as the digest, followed by another permutation f until the number of output blocks is reached.

Note that the sponge method is quite like a symmetric cipher and can be used to encode arbitrary large messages as well. For ZKP’s however, we do not worry about encoding large messages, but rather the input size M is known a priori, and the hash instance is either part of an arithmetic circuit for a NP statement such as (1.1), a standalone computation or a Merkle tree. In the following section we discuss the permutation function used in some of the hash functions keeping the above statement in mind.

2.2 Permutation function

Given an input state $\text{state} \in \mathbb{F}^t$, the permutation f is a bijective map $\mathbb{F}^t \rightarrow \mathbb{F}^t$. The permutation function (the hash instance), in general proceeds over several rounds R in a sequence of operations. The requirement of R is largely fixed by satisfying security inequalities and vary across implementations.

1. **Instantiation:** We instantiate the hash by a parameter triple (t, p, S) which sets the prime field, the security level, and the size of the internal state $|t|$.
 - **The prime field modulus p :** in bit size $n = \log_2(p)$

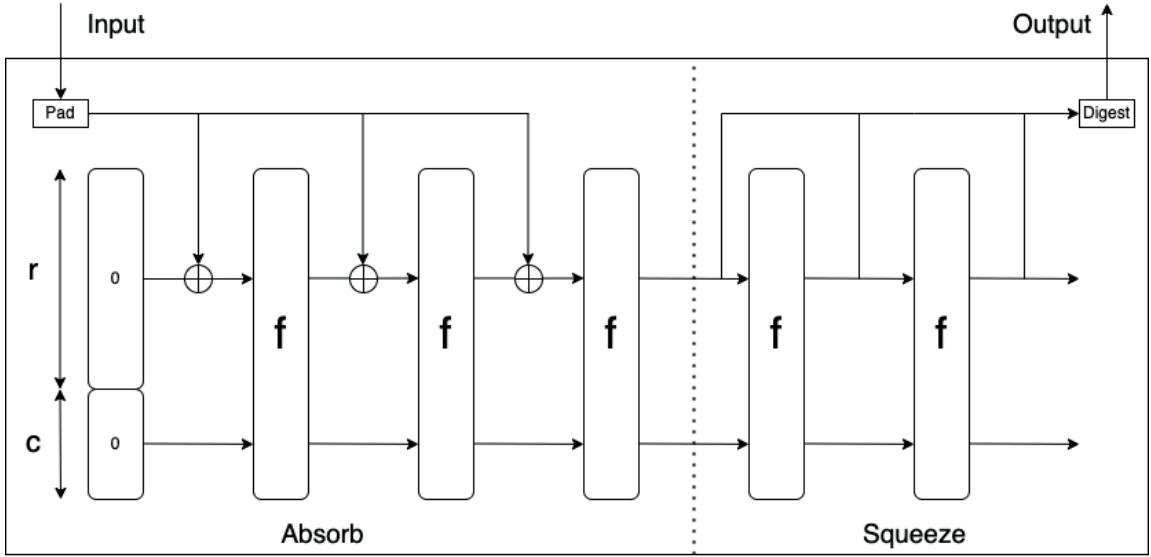


Figure 1. Sponge construction [32]

- **Security level S :**
- **The width t :** is calculated using the formula

$$t = \text{len}(\text{input}) + \text{len}(\text{output}) = \text{len}(\text{input}) + \left\lceil \frac{2S}{\log_2(p)} \right\rceil \quad (2.2)$$

As an e.g. for a hash with 128 bit security, and a 256 bit prime field, the digest size is 1.

2. **Preparatory:** From the instantiation (t, p, S) all additional parameters required for the hash are computed. We will represent all intermediate states of the hash by the variable

$$\text{state}[i], \forall i = 0, 1, \dots, t - 1$$

3. **Computation:** The computations consists of sequential operations referred to as the round function. At the end of all the rounds R the hash function outputs the digest element(s).

In general the round function consists of the following building blocks.

1. **Non-linear layer (S-box):** The S-box is chosen as the power map

$$\pi_0 : x \rightarrow x^\alpha \quad (2.3)$$

$$\pi_1 : x \rightarrow x^{-1} \quad (2.4)$$

where $\alpha \geq 3$ is usually chosen as the smallest integer that guarantees invertibility and provides non-linearity. The S box exponent α is the smallest possible integer that satisfies

$$\gcd(\alpha, p - 1) = 1 \quad (2.5)$$

where p is the prime modulus in a prime field \mathbb{F}_p . If no such α is found satisfying (2.5), then one can use π_1 . Depending on the design of the hash sometimes either or both of (2.3), (2.4) are used. The main reason to include the power map, is to increase the degree such that the security requirements are met.

2. **Addition of Round Constants:** This is simply t field additions of the state with that round's constant RoundConstants_j .

$$\text{state} = \text{state} \vec{\oplus} \text{RoundConstants}_j \quad (2.6)$$

The round constants play the role of the public key, and usually new values are injected into each round. Given the triple (t, p, S) these keys are generated usually in a deterministic way and known a-priori the hash computation itself.

3. **Linear layer:** The linear layer is a matrix multiplication of the state by an MDS (Maximum Distance Separable) matrix whose goal is to spread the uniform randomness properties across the entire state.

$$\text{state} = \text{state} \times \mathcal{M} \quad (2.7)$$

The MDS matrix also depends only on the triple (t, p, S) and is precomputable prior to the hash invocation.

Below we try to give a flavor of the definitions above for two hash functions permutations designs, one is based on HADES [28] and the other is based on MARVELlous [29]. The main difference between the two designs is in the non-linear layer as explained below (also see fig 2)

- In the HADES design [28], the S-boxes are unevenly distributed across several rounds. There are partial rounds where the S-box is only applied to some of the elements in t , and full rounds where the S-box is applied to all elements in t . The structure of the rounds are such that the partial rounds are sandwiched in one batch in-between two batches of full rounds. The external rounds protect against statistical attacks and the internal rounds raise the degree of the permutations. An example of this design are Poseidon/Reinforced concrete [10, 24].
- In the MARVELlous design [29], the non-linear layer uses the power map π_1 (2.3) in even rounds and π_2 (2.4) in odd rounds. Since π_2 is an inverse map, if π_1 is of low degree then π_2 is in general a very high degree map (and vice-versa) due to modular inversion. This increases the computational complexity of the hash, but provides better security. An example of this design is the Rescue/Rescue-prime [4, 19].

We wish to stress that despite several new hash functions such as Anemoi [23], Grendel [26] and Griffin [27], the basic operations are some variations of the HADES or MARVELlous templates. Note that reinforced concrete [24] and Sinsemilla [21] are the first of the ZK-friendly hashes optimized for lookup tables. In section 3, we describe some of the general use cases of hashes in ZK, followed by a review of Poseidon and Rescue from the HADES and MARVELlous families respectively.

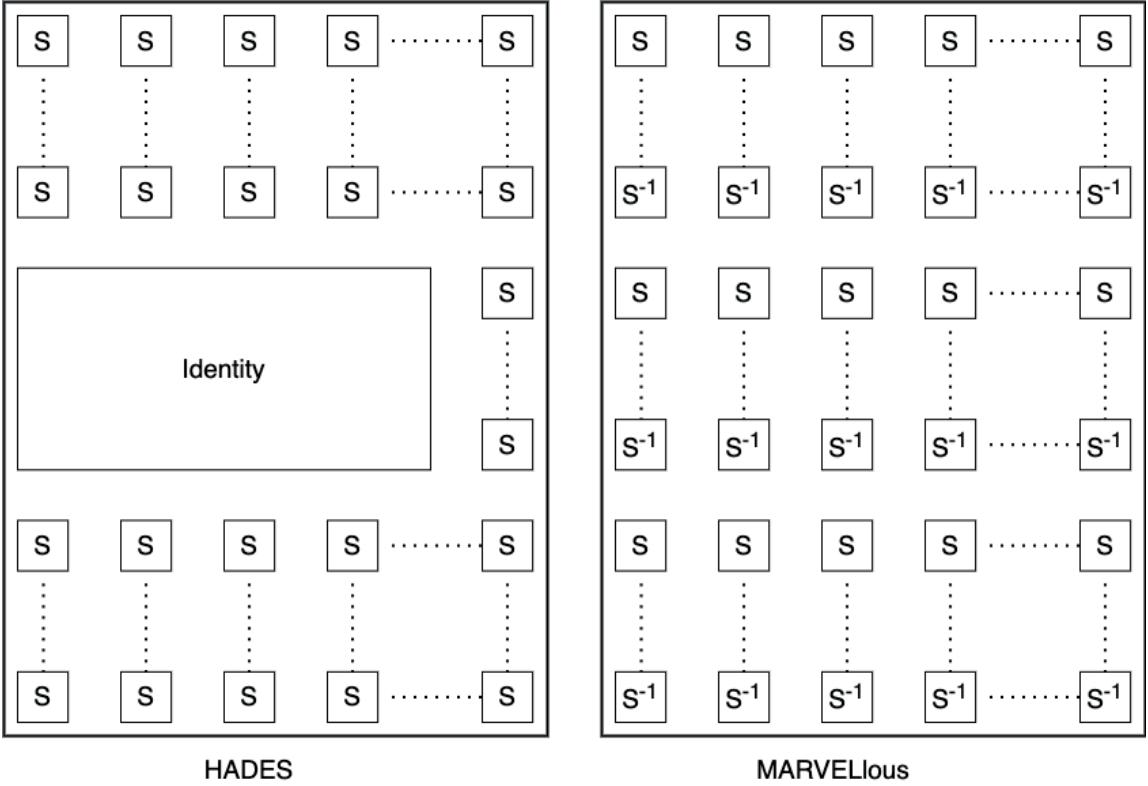


Figure 2. HADES vs MARVELlous design: The rows on each represent the states, S represents the map π_1 (2.3) and S^{-1} represents π_2 (2.4). In HADES there are uneven distribution of S-boxes. The partial rounds are sandwiched between full rounds. In each partial round, the S-box is applied to one element in t , and identity for the remaining $t - 1$ elements. For the MARVELlous design, the S-box π_1 and the inverse S-box π_2 are applied in every round.

3 Use cases

In this section we discuss some straightforward usages of hashes in ZK space, such as merkle commitments §3.1 and Fiat-Shamir §3.2 transformations and more specific applications of Poseidon §3.3 and Rescue §3.4 where arithmetization of the hash plays a more significant role.

3.1 Merkle trees and commitments

A Merkle tree is a data structure of nodes, in a layered tree form. See for instance fig 3 for a binary merkle tree. The nodes in the bottom most layer are called the leaf nodes H_i , which in general contain hashes of data. Each non leaf parent node is a hash of its child nodes. The one way property and collision resistance of the hash function guarantees that the sequence of such hashes into a unique root thus making it a binding commitment scheme for any vector of nodes. Each leaf element in the tree has a unique path that specifies its route and siblings on the way to the root. For e.g. in fig 3 the leaf T_2 has the path element

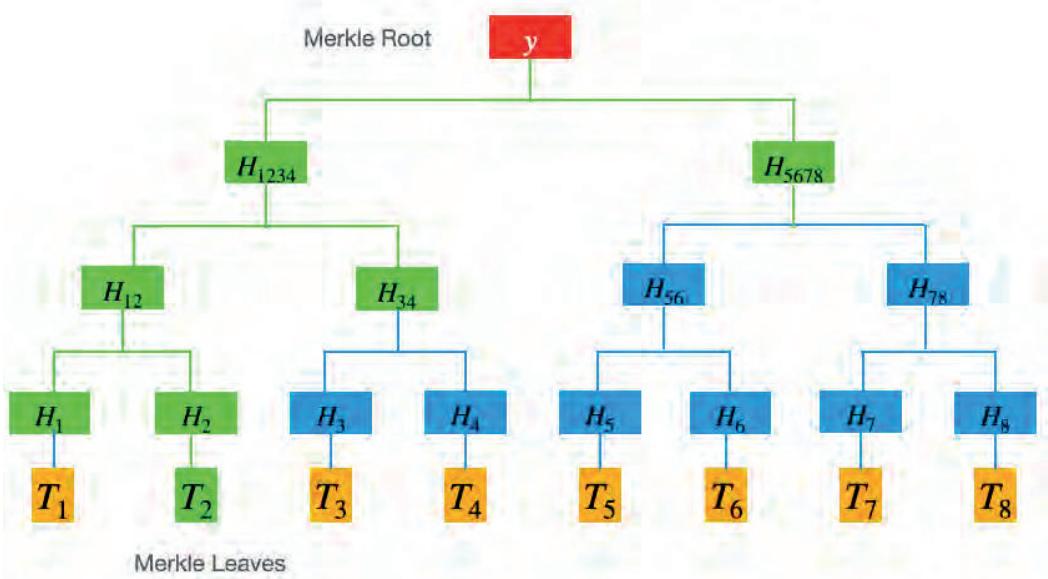


Figure 3. A binary Merkle tree: $H_{ij} = H(H_i, H_j)$, the blocks in green are the path elements corresponding to proof of membership of T_2 in the tree. The root y is a public commitment.

(L, L, R) from y and has the siblings $\Pi : (H_{1234}, H_{5678}, H_{12}, H_{34}, H_1, H_2)$.². As one can see in fig 3, the tree has $n_{nodes} = 8$, the depth of the tree is $3 = \log_2(n_{nodes})$. In order to prove that T_2 is an element in the leaf of the tree with y a public commitment, the proof should consist of $3 * 2 = 6$ elements.

In a more general case, where instead of a binary tree, if we have a t -nary tree, it would have a depth $d = \log_t(n_{nodes})$ and the total proof size for membership in the leaf would amount to $d * t$ field elements.

3.2 Fiat-Shamir

The Fiat–Shamir heuristic allows one to take an interactive argument of knowledge and uses random oracle properties to convert it into a non-interactive argument of knowledge. As the reader would have guessed, the hash function here plays the role of the random oracle and creates a digital signature of the interaction. Consider the following sequence of interactions (based on [33])

1. The prover and verifier agree on a public data $x \in \mathbb{F}^{|x|}$
2. Prover adds witness, does a prescribed computation and sends proof elements $\Pi_a \in \mathbb{F}^{|t|-|x|}$
3. Verifier responds with challenge $\alpha \leftarrow \mathbb{F}$
4. Prover includes α in a prescribed computation and responds with proof element $\Pi_b \in \mathbb{F}^{|t|-|x|}$

²L: left, R: Right

5. Verifier responds with challenges $\beta \leftarrow \mathbb{F}$
6. Prover includes α, β in a prescribed computation and responds with final proof Π_c

The protocol is rendered non-interactive by replacing the interaction steps 3, 5 with a random oracle hash $c \leftarrow H(a_0, a_1, \dots, a_{t-1})$ as follows.

1. The prover and verifier agree on a public data $x \in \mathbb{F}^x$
2. Prover adds witness, does a prescribed computation and computes proof $\Pi_a \in \mathbb{F}^{|t|-|x|}$
3. Prover computes $\alpha \in \mathbb{F}$, by $\alpha \leftarrow H(x, \Pi_a)$
4. Prover includes α in a prescribed computation and generates proof element $\Pi_b \in \mathbb{F}^{|t|-|x|}$
5. Prover computes $\beta \in \mathbb{F}$ by $\beta \leftarrow H(x, \Pi_a, \Pi_b)$ ³
6. Prover includes α, β in a prescribed computation and responds with final proof Π_c

If both the prover and the verifier agree on the random oracle generator, i.e the hash function and how it is computed a priori, then the protocol becomes non-interactive. Note that the form of the hash does not just include the values sent by the prover to the verifier in the interactive protocol, but also includes the public value. This is necessary to prevent the "frozen heart" vulnerability that allows a malicious prover to forge proofs [34]. Almost every ZKP implementation uses the Fiat-Shamir Heuristic, and although it is not necessarily a computational bottleneck, the hashes used in this part need to be secure.

3.3 Poseidon: Filecoin

In this section, we describe Poseidon [10] as it is used in the Filecoin project [30]. Following our discussion of the universal properties of hashes in ZKP §2.1 and more specifically the sponge construction in §2.2, we define a poseidon instantiation as specification of the triple (t, p, S) where p is the prime modulus of the scalar field. Filecoin uses the elliptic curve BLS12-381 [35]. The parameters of the scalar field in the group are

$$\begin{aligned} \text{bit size} &= \log_2(p) \sim 256 \text{ bits} \\ \text{Security} : S &= 128\text{bits} \\ \text{digest size} &= \left\lceil \frac{2S}{\log_2(p)} \right\rceil = 1 \end{aligned} \tag{3.1}$$

The computation consists of two types of rounds of sequential operations referred to as the round function.

- R_F Full rounds represented split into two $R_f = R_F/2$ rounds.
- Partial rounds R_p which operates between the two R_f rounds.

³It is really important to include Π_a in this follow up challenge, this is necessary for proof non-malleability via challenge generation. We thank Suyash Bagad for pointing this out.

At the end of all the rounds $R_f + R_P + R_f$ the Poseidon hash function outputs the digest `state[1]`. Any round function of Poseidon permutation consists of the following three components.

1. Add Round Constants, denoted by $ARC(\cdot)$. Every round j begins with t field additions of the state with that round's constant RoundConstants_j .

$$\text{state} = \text{state} \vec{\oplus} \text{RoundConstants}_j \quad (3.2)$$

2. The non-linear layer: *S-box function* is defined as

$$S : \mathbb{F}_p \rightarrow \mathbb{F}_p$$

$$S(x) = x^5 \quad (3.3)$$

If the round corresponds to a full round, the S -box is applied on each element of the state, else only on the first element

$$\text{state} = \begin{cases} \text{state}[i]^5 |_{i=0,1,\dots,t-1} & \text{Full round} \\ \text{state}[0]^5 & \text{Partial round} \end{cases} \quad (3.4)$$

The total number of S boxes is given by $N_{S\text{-}box} = tR_F + R_p$ and is optimized while satisfying the security inequalities (see Poseidon paper [10]).

3. MDS layer denoted by $\mathcal{M}_{t \times t}(\cdot)$: This operation is the linear layer of Poseidon construction. It consists of the matrix multiplication of the **state** by a $t \times t$ MDS matrix.

$$\text{state} = \text{state} \times \mathcal{M} \quad (3.5)$$

For more details, on the MDS matrix and round constant generation we refer to our github implementation in Python [36]. The Poseidon instantiations used in filecoin are summarized in table 2. Let us briefly discuss how the Poseidon function is used in filecoin. The relevant

Instantiation	t	input	(R_F, R_P)	Arity	A	M
Poseidon ₁₁ ($\mathbb{F}_p^{[12]}$) $\rightarrow \mathbb{F}_p^{[1]}$	12	$[2^{11} - 1, t_1, t_2, \dots, t_{11}]$	(8, 57)	-	2463	2922
Poseidon ₈ ($\mathbb{F}_p^{[9]}$) $\rightarrow \mathbb{F}_p^{[1]}$	9	$[2^8 - 1, t_1, t_2, \dots, t_8]$	(8, 56)	8	1617	2004
Poseidon ₃ ($\mathbb{F}_p^{[3]}$) $\rightarrow \mathbb{F}_p^{[1]}$	3	$[2^2 - 1, t_1, t_2]$	(8, 56)	-	352	592

Table 2. Poseidon Instantiations in Filecoin: In the table A refers to the number of modular additions and M refers to the number of modular multiplications per hash invocation.

process is known as PC2, and we refer to the documentation [30] for further details. What is relevant for this discussion is that, the data structure in *PC2* of filecoin consists of 11×2^{30} array of 256 bit field elements. Poseidon_{11} is used to hash all the columns into digests that create the leaves of a Merkle Tree. Following which Poseidon_8 is employed to create an Octinary Merkle tree of depth $\log_8(2^{30}) = 10$. In addition another instance of Poseidon_8 is

used on the last layer of the data structure to create another Octinary Merkle tree. Finally the digests of the two Octinary Merkle trees are hashed using Poseidon_3 and the digests are committed publicly. The total number of Poseidon hashes is roughly $2^{30} + (2^{30} - 1)/7 * 2 + 1$ taking into account the column hashes and the Merkle trees. It takes about 6 – 7 minutes using a *RTX3090* GPU to just run the hash execution part of the protocol. Thus this is a rather unique case, where the execution of the hashes themselves take time. This is perhaps already evident in table 2 where we find that there are substantial modular addition and multiplication operations on 256 bit field elements.

In order to provide a proof of integrity of the computation, random nodes are selected in the beginning of the data structure and the prover is required to provide a valid path up to the public commitment. The R1CS arithmetization of the Poseidon hash function creates gates with "free additions" built in due to large fan in support of the R1CS system. In table 3 we list the sizes of the constraints in R1CS arithmetization for the knowledge of preimage problem in the case of Poseidon_3 and Poseidon_{12} , and knowledge of membership problem in the case of the Merkle tree instances Poseidon_8 . The circuit sizes for the entire

Instantiation	Proof of	R1CS constraints
$\text{Poseidon}_3(\mathbb{F}_p^{[3]}) \rightarrow \mathbb{F}_p^{[1]}$	Preimage	240
$\text{Poseidon}_8(\mathbb{F}_p^{[9]}) \rightarrow \mathbb{F}_p^{[1]}$	Merkle Tree	3416
$\text{Poseidon}_{11}(\mathbb{F}_p^{[12]}) \rightarrow \mathbb{F}_p^{[1]}$	Preimage	459

Table 3. Constraint sizes in R1CS for Filecoin Poseidon applications (128 bit security) evaluated at a vector size 2^{24} [27]

Filecoin project and the arithmetization leads to polynomials of sizes up to 2^{26} , and it takes roughly 7 minutes to generate a Groth16 Proof using a *RTX3090* GPU. Thus in this use case, both the evaluation of the hashes and the generation of the ZKP are compute bottlenecks.

3.4 Rescue: ethSTARK

In this section, we describe Rescue/Rescue-prime [19] as it is used in the ethSTARK project [4]. The Rescue/Rescue-prime hash is based on the MARVELlous design [29] discussed earlier. Unlike the filecoin project, ethSTARK is based on the Zk-STARKS and operates on a small 61 bit prime field

$$\begin{aligned} \text{bit size} &= \log_2(p = 2^{61} + 20 * 2^{32} + 1) \sim 61 \text{ bits} \\ \text{Security : } S &= 128\text{bits} \\ \text{digest size} &= \left\lceil \frac{2S}{\log_2(p)} \right\rceil = 4 \end{aligned} \tag{3.6}$$

The initial state consists of field elements $[\mathbb{F}^4 || \mathbb{F}^4 || 0^4]$ and the round consists of two batches, with each batch differing in the non-linear layer (S-box). Besides the S box is now applied to all elements in all rounds.

1. Batch1: The first non linear layer begins with a cube root permutation

$$\pi_1 : x^{1/3} \equiv x^{\frac{2p-1}{3}} \quad (3.7)$$

in this case the equivalence in the RHS above is due to the fact that $3 \nmid p - 1$. This is followed by a MDS matrix multiplication

$$\text{state}_{12} = \text{state}_{12} \times \mathcal{M}_{12 \times 12} \quad (3.8)$$

and Round constant addition

$$\text{state} = \text{state} \oplus \vec{K}_r \quad (3.9)$$

to generate an Intermediate state **Inter** which is used for measuring AIR constraints later on.

2. Batch2: consists of a non linear layer that cubes the state

$$\pi_2 : x^3 \quad (3.10)$$

followed by a MDS matrix multiplication

$$\text{state}_{12} = \text{state}_{12} \times \mathcal{M}_{12 \times 12} \quad (3.11)$$

and Round constant addition

$$\text{state} = \text{state} \oplus \vec{K}_r \quad (3.12)$$

to end the round.

The sequence of operations is summarized in fig 4. For 128 bit security the Rescue security inequalities require about 10 rounds till the desired goal of uniformity is achieved [4].

The circuit for the ethSTARK is essentially that the prover knows a sequence of witness $w = \{w_0, w_1, \dots, w_n\}$ where $w_i \in \mathbb{F}^4$ such that (1.1) is satisfied. The hash chain length is $n = |w| - 1 = 3 * 2^i$ for $i \in [10, 18]$. Given the length of the hash chain, the prover runs the chain by providing the witnesses and recording the intermediate states **Inter** in an Algebraic Execution Trace (AET) as shown in fig 5. In the fig $g \in \mathbb{F}_p^\times$ (the multiplicative group of size T), and each column is interpreted as polynomial evaluations $(g^r, f_j(g^r))$ of degree $< |T|$. As we mentioned earlier each invocation of the hash takes about 10 rounds and generates 10 of the **Inter** AET rows, and one must add the initial and final state as well. The AET together with the constraints on the various rows and columns form the AIR arithmetization of the computation.

For technical reason the ethSTARK runs invocations in multiples of 3, thus the AET is of dimension $2^{i+5} \times 12$. The execution of these hashes and run times are relatively fast and are not a major compute bottleneck, presumably because of the small bit prime fields. The main compute bottleneck in the STARK case comes because the AET needs to be converted into a polynomial using iNTT (Inverse Number Theoretic Transform)⁴, after that they are

⁴Inverse Discrete Fourier Transform in a finite field

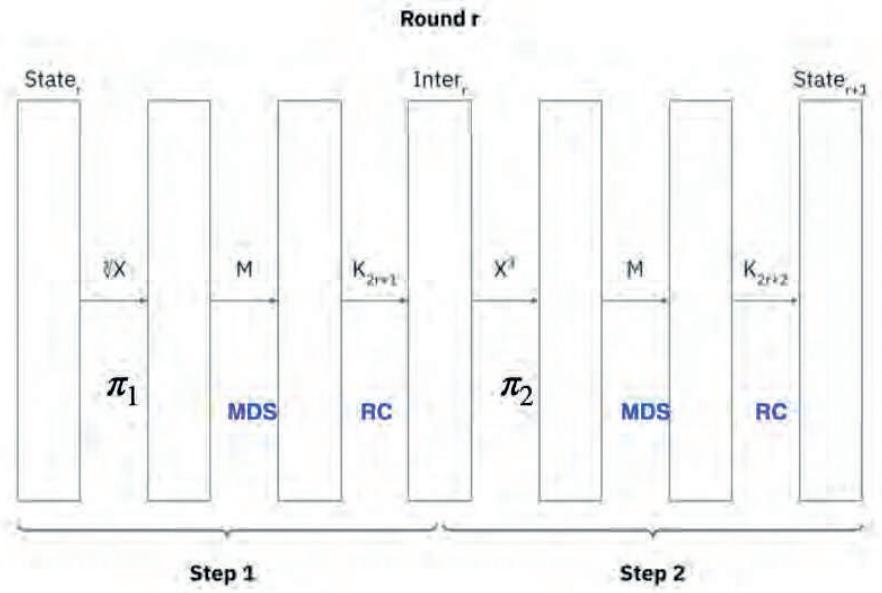


Figure 4. Round function in Rescue

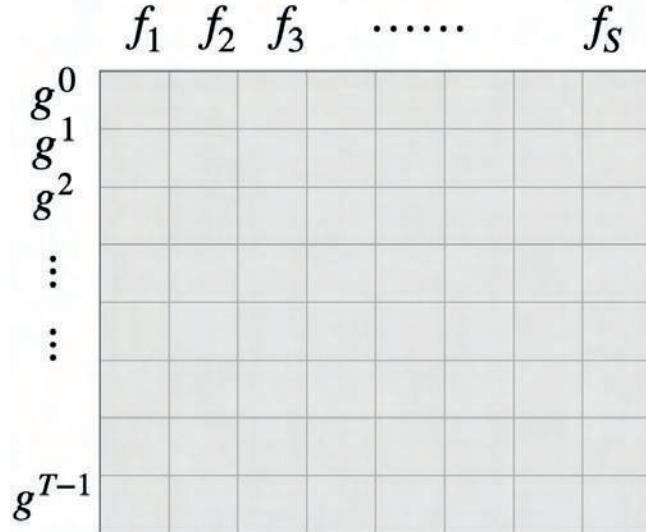


Figure 5. Algebraic execution trace : AET

evaluated in an extended domain which includes a blow up factor $\beta \in [4, 8, 16]$ resulting in polynomial degrees up to $\beta * 2^{i+5}$ which can be very large. The NTT/iNTT takes about 80% of the computational time in the ethSTARK system. Furthermore, the proof system uses FRI-IOP protocol which requires the prover to Merkle commit polynomials of a given degree N in $\log_2 N$ steps, where in each step the degree is decimated by $1/2$. While we discussed the Merkle commitment §3.1 earlier and observed that it is inefficient,

in ethSTARK it consumes up to about 20% of the prover time and is not negligible. This is example of a ZK friendly hash in a STARK system, where both NTT/iNTT and Merkle commitments/FRI are computational bottlenecks.

4 Overview of ZK-friendly hashes in other applications

In this section, we review some of the other use cases not covered in detail in this article. In table 1 we mentioned the ZK applications from Polygon: zkEVM [18], Plonky2 [16] and MidenVM [20]. All these systems use the same prime field \mathbb{F}_p , where $p = 2^{64} - 2^{32} + 1$ corresponds to the Goldilocks prime⁵. In this section, we review some of the use cases of Zk-friendly hashes in these ecosystems.

In the case of Plonky2 [16], the Poseidon instantiation is called Poseidon-Goldilocks, the parameters are ($t = 12, p, S = 128$) and the hash operates with the round parameters $(R_F, R_P) = (8, 22)$. Notice that the number of rounds is far less than the filecoin case discussed in table 2. However, in the non-linear layer, the S-box is $\pi_0 : x^7$ which is larger than the $\pi_0 : x^5$ used in the Filecoin instantiation. In our trial runs, 40% of the prover time is spent on the FRI-IOP in generating Merkle trees, with the remaining time spent in NTT/iNTT on permutation constraints in the Plonk [3] prover system. In using the Plonk prover, one has a freedom to arithmetize the hash function using many basic gates, or using custom gates for more complex functions. Using more basic gates, increases the number of rows in the execution trace and adds to prover complexity. While adding more complex custom gates increases the number of columns in the execution trace, and makes the constraints more complex, thereby increasing verifier time. Plonky2 uses a single custom gate to evaluate an entire instance of Poseidon, this results in execution traces with large number of columns, but fast prover times of about 300–350 milliseconds. Since the proving technique used in Plonky2 is FRI, it results in long proof sizes, and Plonky2 uses Proof recursion to compress the proof sizes to about 43 kilobytes. The trade off between row vs column ratio in the Plonk execution trace, is an optimization unavailable in the R1CS or AIR arithmetization programs and is a topic of active research.

In Polygon zkEVM [18], the instantiation of the Poseidon hash function is identical to that used in Plonky2 [16] described in the paragraph above. The zkprover is a component of the EVM that has a state machine (VM) dedicated to Poseidon. The zkprover's state machines execute programs, and prove that the programs were correctly executed. The Poseidon State Machine (SM) records internal permutations (within the rounds of Poseidon) as state transitions and stores them in the form of rows of a lookup table. It further builds a polynomial constraint system for the state transitions, based on the Poseidon internal permutations. Furthermore, it expresses the state transitions as polynomials and commits to these polynomials using a Polynomial commitment scheme. Since the execution trace is now used as a lookup table, the zkprover uses Plookup [6] to verify if the committed polynomials satisfy the necessary constraints and produce correct trace entries in the lookup

⁵A Goldilocks prime has the structure $p = \phi^2 - \phi + 1$, where $\phi = 2^n$, In the context of the current discussion $\Phi = 2^{32}$. The main advantage of using the Goldilocks prime appears to be fast Karatsuba multiplications [37]

table. The zkprover has a STARK recursion component that operates STARK proofs using FRI-IOP. In the first step, it generates a STARK proof per execution of Poseidon VM. Following which, it batches a fixed number of zk-STARK proofs each corresponding to different executions of the Poseidon VM and produces a STARK proof per batch. Then it bundles a fixed number of batches of proofs and proceeds recursively to produce a single STARK proof, that proves all the other STARK proves that it represents as well as the VM executions represented by the proofs. Finally, the zkEVM compresses the STARK proof using a SNARK by employing the rapidsnark library [38]. This is yet another fascinating ecosystem where ZK-friendly hashes play a significant role.

Next we move on to Miden VM [20] which produces a STARK proof for any program executed by the system. The execution proof can be verified without knowing what the program is. Miden VM operates on the same Goldilocks prime field of plonky2 and zkEVM discussed above, and uses AIR arithmetization. The VM has a dedicated hash component to accelerate the computations of the ZK-friendly hash Rescue-prime [19]. More specifically, the "hash chiplet" in Miden VM is a processor that can execute instructions set (similar to hardware instruction sets for processors) to perform atomic operations in a hash, such as

- A single permutation of Rescue-prime (for e.g. a single round function in fig 4), and outputs the state at the end of a round.
- Execution of a simple $2 \rightarrow 1$ hash, i.e. $H(w_0, w_1) \rightarrow w_3$ where $w_i \in \mathbb{F}^4$ and H is the Rescue-prime hash function.
- Perform a linear hash of n elements in the sponge mode. The hash chiplet sets the rate of the permutation function to 8 and capacity to 4, and absorbs the n elements as described in fig 1. In the squeeze phase it outputs 4 field elements.
- Verification of path in a merkle tree (see fig 3) and to update root in the tree.

In all cases, the VM generates execution trace for AIR depending on the choice of internal states to define the constraint. The STARK proof is generated using the FRI-IOP protocol.

5 Closing Words and a Disclaimer

Our discussion leads us to a general picture that there is much to be explored in lowering the computational overhead of ZK-friendly hash functions. While security is paramount in the definition of hashes in general, low computation complexity is vital for scaling of any ZKP application. Since computational complexity scales with security in general, it is vital that the design of ZK-friendly hash functions add yet another component to their toolbox: Hardware friendliness.

Disclaimer: The ZK space is a fast moving target. While we try to keep this SoK up-to-date, note that the validity of some of the data presented here may change faster.

Acknowledgments

We would like to thank Suyash Bagad and Omer Shlomovits for a thorough review of the draft. We would also like to thank Yuval Domb, Oren Margalit, Ekaterina Semenova, and Guy Weissenberg for comments and suggestions. We thank Robin Salen, Izzy Meckler and Ye Zhang for references.

References

- [1] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, “Quadratic span programs and succinct nizks without pcps.” Cryptology ePrint Archive, Paper 2012/215, 2012.
<https://eprint.iacr.org/2012/215>.
- [2] V. Buterin, “Quadratic arithmetic programs from zero to hero.” <https://medium.com/@VitalikButerin/quadratic-arithmetic-programs-from-zero-to-hero-f6d558cea649>.
- [3] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, “Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge.” Cryptology ePrint Archive, Report 2019/953, 2019. <https://ia.cr/2019/953>.
- [4] StarkWare, “ethstark documentation.” Cryptology ePrint Archive, Paper 2021/582, 2021.
<https://eprint.iacr.org/2021/582>.
- [5] DECENTRIQ, “Proving sha256 preimage knowledge using zokrates.”
<https://blog.decentraliq.com/proving-hash-pre-image-zksnarks-zokrates/>.
- [6] A. Gabizon and Z. J. Williamson, “plookup, a simplified polynomial protocol for lookup tables.” Cryptology ePrint Archive, Paper 2020/315, 2020.
<https://eprint.iacr.org/2020/315>.
- [7] A. Zapico, V. Buterin, D. Khovratovich, M. Maller, A. Nitulescu, and M. Simkin, “Caulk: Lookup arguments in sublinear time.” Cryptology ePrint Archive, Paper 2022/621, 2022.
<https://eprint.iacr.org/2022/621>.
- [8] J. Posen and A. A. Kattis, “Caulk+: Table-independent lookup arguments.” Cryptology ePrint Archive, Paper 2022/957, 2022. <https://eprint.iacr.org/2022/957>.
- [9] ZPrize, “Accelerating poseidon hash function.” https://assets.website-files.com/625a083eef681031e135cc99/6305a4714b50ef347bcd5ee3_poseidon.pdf.
- [10] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schafneiger, “Poseidon: A new hash function for zero-knowledge proof systems.” Cryptology ePrint Archive, Paper 2019/458, 2019. <https://eprint.iacr.org/2019/458>.
- [11] Filecoin, “Neptune.” <https://github.com/filecoin-project/neptune>.
- [12] Mina, “Kimchi.” <https://o1-labs.github.io/proof-systems/specs/kimchi.html#poseidon-hash-function>.
- [13] Scroll-Tech, “Aggregator.” <https://github.com/scroll-tech/halo2-snark-aggregator/blob/main/halo2-snark-aggregator-api/src/hash/poseidon.rs>.
- [14] Scroll-Tech, “Mpt circuit.”
<https://github.com/scroll-tech/mpt-circuit/blob/master/spec/mpt-proof.md>.
- [15] Penumbra, “Poseidon377.”
<https://protocol.penumbra.zone/main/crypto/poseidon/paramgen.html>.

- [16] Polygon-Zero, “Plonky2.” <https://github.com/mir-protocol/plonky2>.
- [17] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, “Fast reed-solomon interactive oracle proofs of proximity.” <https://eccc.weizmann.ac.il/report/2017/134/>.
- [18] Polygon-Hermez, “Polygon zkvm.” <https://github.com/Oxpolygonhermez>.
- [19] A. Szepieniec, T. Ashur, and S. Dhooghe, “Rescue-prime: a standard specification (sok).” Cryptology ePrint Archive, Paper 2020/1143, 2020. <https://eprint.iacr.org/2020/1143>.
- [20] Polygon, “Miden vm.” <https://maticnetwork.github.io/miden/design/chiplets/hasher.html>.
- [21] ZCash, “Sinsemilla hash.” <https://zcash.github.io/halo2/design/gadgets/sinsemilla.html>.
- [22] D. Hopwood, S. Bowe, T. Hornby, and N. Wilcox, “Zcash protocol specification.” <https://zips.z.cash/protocol/protocol.pdf>.
- [23] C. Bouvier, P. Briaud, P. Chaidos, L. Perrin, and V. Velichkov, “Anemoi: Exploiting the link between arithmetization-orientation and ccz-equivalence.” Cryptology ePrint Archive, Paper 2022/840, 2022. <https://eprint.iacr.org/2022/840>.
- [24] L. Grassi, D. Khovratovich, R. Lüftnegger, C. Rechberger, M. Schafnecker, and R. Walch, “Reinforced concrete: A fast hash function for verifiable computation.” Cryptology ePrint Archive, Paper 2021/1038, 2021. <https://eprint.iacr.org/2021/1038>.
- [25] M. Albrecht, L. Grassi, C. Rechberger, A. Roy, and T. Tiessen, “Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity.” Cryptology ePrint Archive, Paper 2016/492, 2016. <https://eprint.iacr.org/2016/492>.
- [26] A. Szepieniec, “On the use of the legendre symbol in symmetric cipher design.” Cryptology ePrint Archive, Paper 2021/984, 2021. <https://eprint.iacr.org/2021/984>.
- [27] L. Grassi, Y. Hao, C. Rechberger, M. Schafnecker, R. Walch, and Q. Wang, “A new feistel approach meets fluid-spns: Griffin for zero-knowledge applications.” Cryptology ePrint Archive, Paper 2022/403, 2022. <https://eprint.iacr.org/2022/403>.
- [28] L. Grassi, R. Lüftnegger, C. Rechberger, D. Rotaru, and M. Schafnecker, “On a generalization of substitution-permutation networks: The hades design strategy.” Cryptology ePrint Archive, Paper 2019/1107, 2019. <https://eprint.iacr.org/2019/1107>.
- [29] T. Ashur and S. Dhooghe, “Marvellous: a stark-friendly family of cryptographic primitives.” Cryptology ePrint Archive, Paper 2018/1098, 2018. <https://eprint.iacr.org/2018/1098>.
- [30] Filecoin, “The filecoin project.” <https://github.com/filecoin-project>.
- [31] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, *On the indifferentiability of the sponge construction*, in *Advances in Cryptology – EUROCRYPT 2008* (N. Smart, ed.), (Berlin, Heidelberg), pp. 181–197, Springer Berlin Heidelberg, 2008.
- [32] G. Bertoni, J. Daemen, S. Hoffert, M. Peeters, G. V. Assche, and R. V. Keer, “Keccak hash.” https://keccak.team/sponge_duplex.html.
- [33] D. Khovratovich, J. Aumasson, and P. Quine, “Safe (sponge api for field elements) – a toolbox for zk hash applications.” https://hackmd.io/bHgsH6mMStCVibM_wYvb2w?view.
- [34] J. Miller, “Serving up zkps.” [https://blog.trailofbits.com/2021/02/19/serving-upzero-knowledge-proofs/](https://blog.trailofbits.com/2021/02/19/serving-up-zero-knowledge-proofs/).

- [35] Arkworks, “Arkworks bls12-381.”
https://docs.rs/ark-bls12-381/latest/ark_bls12_381/.
- [36] E. Semenova, “Poseidon - python.” <https://github.com/ingonyama-zk/poseidon-hash>.
- [37] J. Cook, “Goldilocks prime.”
<https://www.johndcook.com/blog/2019/05/12/ed448-goldilocks/>.
- [38] Polygon, “Rapid snark.” <https://github.com/iden3/rapidsnark>.

PipeMSM: Hardware Acceleration for Multi-Scalar Multiplication

Charles. F. Xavier,
Ingonyama



PipeMSM: Hardware Acceleration for Multi-Scalar Multiplication

Charles. F. Xavier
charlie@ingonyama.com

Abstract—Multi-Scalar Multiplication (MSM) is a fundamental computational problem. Interest in this problem was recently prompted by its application to ZK-SNARKs, where it often turns out to be the main computational bottleneck.

In this paper we set forth a pipelined design for computing MSM. Our design is based on a novel algorithmic approach and hardware-specific optimizations. At the core, we rely on a modular multiplication technique which we deem to be of independent interest.

We implemented and tested our design on FPGA. We highlight the promise of optimized hardware over state-of-the-art GPU-based MSM solver in terms of speed and energy expenditure.

Index Terms—Zero-Knowledge, Hardware acceleration, Multi-Scalar Multiplication (MSM), FPGA

I. INTRODUCTION

The multi-product problem is defined as follows: given a sequence of elements $(t_0, t_1, \dots, t_{n-1})$, compute the following sequence of products using the minimal number of multiplications possible:

$$y_i = \prod_{j=0}^{n-1} t_j^{x_{i,j}}; \forall i = 0, 1, \dots, k-1 \quad (1)$$

Here $x_{i,j} \in \mathbb{Z}_2 : \{0, 1\}$ are elements of a matrix $X_{k \times n}$. When $x_{i,j} \in \mathbb{Z}_r, r \geq 2$ then the sequence y_i computes various exponents of the elements t_j and (1) is known as a multi-exponentiation problem. Four decades ago, Pippenger provided an asymptotically optimal algorithm for both of these problems [1].

Some of the problems that can be reduced to the multi-product or multi-exponentiation are evaluating sparse multivariate polynomials and computing matrix-vector products. The logarithmic version of the same multi-exponentiation problem can be equivalently stated as a Multi-Scalar Multiplication (MSM)

$$\tilde{y}_i = \sum_{j=0}^{n-1} x_{i,j} \tilde{t}_j. \quad (2)$$

One interesting application of the MSM problem is in cryptography, and in particular ZK-SNARKs. In this instance of the problem, we are interested in computing a single product y , where $\tilde{t}_j = G_j$ are points of an Elliptic Curve (EC) group \mathcal{G} over a prime order field \mathbb{F}_q and x_j are scalars s.t. $x_j \in \mathbb{F}_{|\mathcal{G}|}$. MSM as defined above is represented throughout this paper as

$$\text{MSM}(x, G) = \sum_{j=0}^{n-1} x_j G_j \quad (3)$$

Note that at the end of the computation (3) the left-hand side is a single EC point from \mathcal{G} .

A. MSM dominates ZK prover computation

A Zero Knowledge Proof (ZKP) is a protocol between two parties, a prover, and a verifier. The prover aims to convince the verifier that a statement is true with high probability, without revealing any additional information. Consider an NP relation $\mathcal{R}(x, w)$ with a corresponding language $\mathcal{L}(\mathcal{R})$, where x is a public input known to both the prover and the verifier and the "witness" w is known only to the prover. For example, think of a boolean circuit with some assignments known to both parties (x) and the other ones known only to the prover (w).

A proof π must possess several properties. At the minimum we require completeness, soundness, and zero-knowledge for which formal definitions can be found in [2]. ZK-SNARKs (Zero-Knowledge Succinct Non-interactive Arguments of Knowledge) are a subtype of ZK proofs with several additional properties like small proof size and fast verification. This allows ZK-SNARKs to serve as a foundation for highly efficient verifiable computation. The trade-off is in the prover's complexity which becomes a computational bottleneck. See [3] for several references on the subject at various levels, and [4] for a practical technical introduction to the inner workings of ZK-SNARKs.

We use the term *arithmetization* to define the process of "codifying" a relation \mathcal{R} in a set of arithmetic constraints over a finite field \mathbb{F} . It is a crucial step, aimed at describing the problem in a unified arithmetic language, enabling the prover to then apply information-theoretic and cryptographic tools for generating the proof. Examples of popular formats in which the "codified" relations are represented are QAP [5], [6] and R1CS.

A necessary step in any ZK-SNARK protocol is efficiently evaluating polynomials at points requested by the verifier. A key cryptographic primitive for an interactive protocol that proves/verifies polynomial relations is a Polynomial Commitment (PC) scheme. PC refers to a computationally binding commitment to a polynomial P , using which the committer can prove the evaluation $P(k)$ at any point $k \in \mathbb{F}$ without revealing P at other points (hiding).

In table I we survey some of the popular commitment schemes used in ZKPs.

We see that for example, Plonk [10] and Marlin [11] use the pairing-based KZG10 [7] scheme. Bulletproofs [12] use

Scheme	KZG10 [7]	IPA [8]	DARK [9]
Cryptographic primitive	Pairing	Discrete Log	Group of unknown order
Ingredients	$\mathcal{G}/\mathbb{F}_q : G_1 \in \mathbb{G}_1, G_2 \in \mathbb{G}_2, G_T \in \mathcal{G}_T$ Pairing: $e(\mathcal{G}_1, \mathcal{G}_2) \rightarrow \mathcal{G}_T$ $SRS = (\tau^0 G_1, \tau^1 G_1, \dots, \tau^{n-1} G_1)$	$\mathcal{G}/\mathbb{F}_q : G_i \in \mathcal{G}$ $\text{random}(G_i) \leftarrow \mathcal{G}$ $q \in \mathbb{Z}_p; q \sim \mathcal{O}(p \log_2 n)$	$ \mathcal{G} $ unknown $\text{random}(G_i) \leftarrow \mathcal{G}$ $q \in \mathbb{Z}_p; q \sim \mathcal{O}(p \log_2 n)$
Trusted setup	Yes	No	depends on \mathcal{G}
Comm(P) $P(X) = \sum_{i=0}^{n-1} a_i X^i$	$P(\tau)G_1 = \sum_{i=0}^{n-1} a_i SRS_i$	$\langle a, G \rangle = \sum_{i=0}^{n-1} a_i G_i$	$\sum_{i=0}^{n-1} a_i Q_i$ $Q = (q^0 G, q^1 G, q^2 G, \dots, q^{n-1} G)$
Computational primitive	MSM	MSM*	MSM*
ZKP's	Plonk [10], Marlin [11]	BulletProofs [12]	Supersonic [9]

TABLE I
CRYPTOGRAPHIC PRIMITIVES TO COMPUTATIONAL PRIMITIVES

inner product arguments [8] that rely on the discrete logarithm assumption. Groups of unknown order play a central role in DARK (Diophantine Argument of Knowledge) [9], used by Supersonic. All of the above protocols rely on MSM to compute polynomial commitments. In table I the MSM* notation is used to highlight that G_i in IPA and multiples of G in DARK are not predetermined and computed during proof generation. On the other hand, in KZG10 the bases $\tau^i \cdot G_1$ are precomputed before proving.

One last classic example of a popular protocol that also requires the prover to compute several MSMs is Groth16 [13].

To sum up, table I highlights the strong presence of MSM in polynomial commitments. While there is sufficient motivation to consider MSM as a standalone problem, even purely from a theoretical standpoint, it also turns out to be a key computational problem that needs to be tackled in order to accelerate ZK-SNARK computation.

In table II, we see that some of the most popular protocols spend a significant fraction of time computing MSMs. While

Protocol	No. of MSM (Prover)	Prover's time %
Groth16 [13]	4 in \mathcal{G}_1 , 1 in \mathcal{G}_2	70 – 80%
Marlin [11] + Lunar [14]	11 in \mathcal{G}_1	70 – 80%
Plonk [10]	9 in \mathcal{G}_1	85 – 90%

TABLE II
MSM DOMINATION IN ZKP

the exact percentage of time can vary depending on the implementation and circuit size, in general, MSM is the main computational bottleneck for ZK-SNARK-based proof systems.

In this paper we break down the acceleration of MSM (3) into three sections as follows

- Efficient modular multiplication §II
- Elliptic curve point addition §III
- Multi-Scalar Multiplication §IV

We start with modular multiplication, a fundamental primitive for all finite field arithmetic. There exist efficient modular reduction methods for primes of a special form. However for the general case, which we consider, our modular reduction method needs to work with arbitrary primes. Several such

methods can be found in the literature: Barrett reduction [15], Montgomery reduction [16], and lookup table-based methods [17]. In §II we present an optimized variant of the modular multiplication method based on Barrett reduction.

Moving on to the higher level, in §III we discuss the elliptic curve addition based on complete formulae which have been overlooked in the literature, and point out their efficiency in data sampling and parallelism.

Finally, in §IV we discuss our algorithmic optimizations for the efficient parallel implementation of the MSM solver.

Having all the theoretical tools in hand, we implemented our design on Xilinx U55C FPGA. In §V, we demonstrate how our implementation compares to state-of-the-art GPU-based MSM solver.

We conclude with further potential algorithmic improvements in §VI.

B. Related Work

Sppark [18] is a new library developed by Supranational that provides GPU implementation for primitives used in ZKP, among them MSM. We compare our implementation to Sppark in section V.

The best existing FPGA implementation for SNARK are projects led by Ben Devlin for Zcash [19] and the Ethereum Foundation [20]. The projects either implement a limited version of MSM with a naive approach, resulting in poor performance, or skip MSM altogether.

PipeZK [21] is a recent work that provides end-to-end pipelined design for ZK-SNARK. However, PipeZK chose ASIC as its target hardware, skipping over FPGAs. ASIC is at best years away from being accessible while FPGAs are available today with supply time similar to GPUs or even instantly in the cloud [22]. The measurements in PipeZK are therefore done via simulation of ASIC without a physical silicon chip. Our measurements on the other hand are done on physical hardware.

II. EFFICIENT MODULAR MULTIPLICATION

The standard modular multiplication problem in \mathbb{F}_q is formulated as

$$r = a \cdot b \mod q \quad (4)$$

where $a, b, r \in \mathbb{F}_q$, q is a prime. Equivalently this can be written as

$$a \cdot b = l \cdot q + r \quad (5)$$

with $l \in \mathbb{Z}$ such that $0 \leq r < q$. In this section we describe an efficient, hardware-friendly method for computing (4). For the rest of this paper assume that all variables are represented in binary, and denote by n the number of digits used to represent any element in \mathbb{F}_q , i.e. $n = \lceil \log_2 q \rceil$. We begin by recounting the details of the Barrett reduction method.

A. Barrett reduction

Our strategy is to assume that l is approximately known and we denote by \hat{l} the corresponding approximation

$$l - \lambda \leq \hat{l} \leq l \quad (6)$$

where $\lambda \geq 0$ is a known constant.

1) Case 1: $\lambda = 0$: It is clear by definition (5) that

$$ab[2n-1 : 0] - \hat{l}q[2n-1 : 0] = r[n-1 : 0] \quad (7)$$

where the brackets denote bit sizes. Writing out bits explicitly, (7) is rewritten as:

$$\begin{array}{cccccc} ab[2n-1] & \dots & ab[n] & ab[n-1] & \dots & ab[0] \\ - \hat{l}q[2n-1] & \dots & \hat{l}q[n] & \hat{l}q[n-1] & \dots & \hat{l}q[0] \\ \hline 0 & \dots & 0 & r[n-1] & \dots & r[0] \end{array}$$

Note that only $ab[n-1 : 0]$ and $\hat{l}q[n-1 : 0]$ are necessary in order to execute the computation. Also note that the result is always a positive n -bit number, so any carry to the $n+1$ 'st bit is ignored.

2) Case 2: $\lambda > 0$: In this case

$$ab - \hat{l}q = r + \lambda q \quad (8)$$

the number of bits required to represent the right-hand side of the above is

$$\lceil \log_2(r + \lambda q) \rceil \leq n + \left\lceil \log_2 \frac{r + \lambda q}{q} \right\rceil \leq n + \lceil \log_2(1 + \lambda) \rceil \quad (9)$$

so instead of the lowest n bits in (7) we must take $n + \lceil \log_2(1 + \lambda) \rceil$. For example, if $\lambda = 3$, the total number of additional bits required would be 2.

3) Approximating l : The only remaining part is to compute the approximation \hat{l} :

$$l = \left\lfloor \frac{ab}{q} \right\rfloor = \lim_{k \rightarrow \infty} \frac{ab \cdot m(k)}{2^{k+n}} \quad (10)$$

where

$$m(k) = \left\lfloor \frac{2^{k+n}}{q} \right\rfloor < 2^{k+1} \quad (11)$$

For a finite k , the approximation error of $1/q$ is

$$e(k) \equiv \frac{1}{q} - \frac{m(k)}{2^{k+n}} < 2^{-(k+n)} \quad (12)$$

where the upper bound can be derived by examining the maximal difference between the binary representation of the left

and right terms. This immediately leads to the approximation error on $l(k)$

$$e(l, k) \equiv \frac{ab}{q} - \frac{ab \cdot m(k)}{2^{k+n}} < 2^{2n} \cdot 2^{-(k+n)} = 2^{n-k} \quad (13)$$

Thus if $k \geq n$ the error is at most 1. Choosing $k = n$ (i.e. $m(n) < 2^{n+1}$) leads to the following approximation on l

$$\hat{l}_0 = \left\lfloor \frac{abm}{2^{2n}} \right\rfloor, e(\hat{l}_0) < 1 \quad (14)$$

where the approximation error obeys (13).

Let us instead perform the above multiplication in two stages. Initially assume that $ab[2n-1 : 0]$ is available and perform the multiplication as follows

$$\frac{abm}{2^{2n}} = \frac{ab[2n-1 : n] \cdot m}{2^n} + \frac{ab[n-1 : 0] \cdot m}{2^{2n}} \quad (15)$$

$$< \frac{ab[2n-1 : n] \cdot m}{2^n} + 2 \quad (16)$$

This immediately leads to the following approximation on l

$$\hat{l}_1 = \left\lfloor \left\lfloor \frac{ab}{2^n} \right\rfloor \cdot \frac{m}{2^n} \right\rfloor, e(\hat{l}_1) < 3 \quad (17)$$

where the inner multiplication is n -by- n -bit, the outer multiplication is n -by- $(n+1)$ -bit, and the approximation error is upper bounded by the sum of (14) and the right-most term of (16).

In fig 1 we present a block diagram of the modular multiplier. The diagram uses the \hat{l}_1 approximation for l given in (17). Note that although multiplication $\left\lfloor \frac{ab}{2^n} \right\rfloor \cdot m$ is n -by- $n+1$ bit, the result is less than $q2^n$ so it's $2n$ -bit long and so \hat{l}_1 is n -bit long.

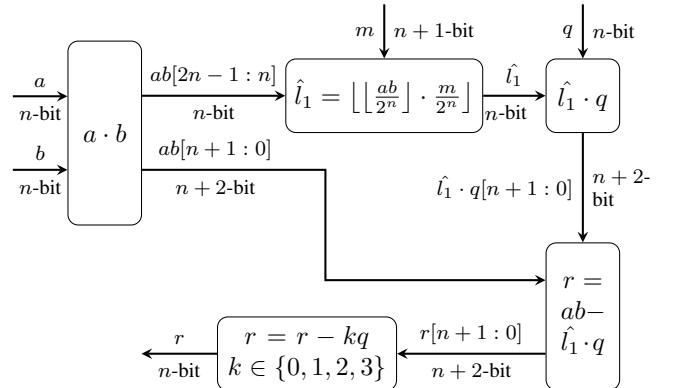


Fig. 1. Optimized Barrett modular multiplier

B. Optimizing integer multiplications in Barrett algorithm

In general, two large integers x, y can be efficiently multiplied using the Karatsuba algorithm (see e.g., [23]). Writing $x = x_1 2^k + x_0$ and $y = y_1 2^k + y_0$, one reduces the computation of the product $x \cdot y$ to 3 narrower products as shown below

$$x \cdot y = 2^{2k} x_1 y_1 + x_0 y_0 + 2^k ((x_1 + x_0)(y_1 + y_0) - x_1 y_1 - x_0 y_0) \quad (18)$$

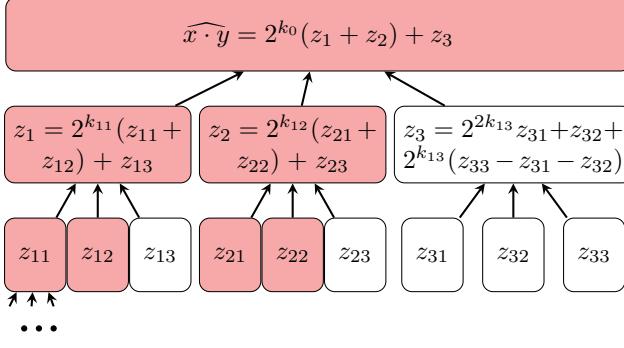


Fig. 2. LSB multiplication

To multiply two large integers, (18) can be iterated before reaching multiplications that are narrow enough to be computed by the hardware directly. However, note that in fig 1, we do not always need the whole result of integer multiplications. In particular, we are only interested in the most significant n bits of $ab[2n - 1 : n] \cdot m$ and the least significant $n + 2$ bits of $l_1 \cdot q$.

We first demonstrate how to compute $n + \sigma$ lowest bits of the product of two n -bit numbers $x \cdot y$ (in the case of algorithm 1, $\sigma = 2$). As before, we write $x = x_1 2^k + x_0$ and $y = y_1 2^k + y_0$ and expand $x \cdot y = x_1 y_1 2^{2k} + 2^k(x_1 y_0 + x_0 y_1) + x_0 y_0$. Note that, as long as $2k \geq n + \sigma$, lowest $n + \sigma$ bits of the product $x \cdot y$ are correctly computed by

$$\widehat{x \cdot y} = 2^k(x_1 y_0 + x_0 y_1) + x_0 y_0 \quad (19)$$

Moreover, we only need the lower $n + \sigma - k$ bits of $x_1 y_0$ and $x_0 y_1$ to be correct. This leads us to a recursive algorithm as shown in fig 2, where the white boxes refer to regular Karatsuba iterations (18) and the pink boxes refer to the simplified iterations (19). To ensure the correctness of the algorithm, we need to make sure that at each simplified iteration (19), k_i satisfies $n + \sigma \leq 2k_i + k_{i-1} + k_{i-2} + \dots + k_0$.

A more difficult problem is to compute n highest bits of the $2n$ -bit product of two numbers $x \cdot y$. Following our earlier approach, we write $x = x_1 2^k + x_0$ and $y = y_1 2^k + y_0$. Since $x \cdot y = x_1 y_1 2^{2k} + 2^k(x_1 y_0 + x_0 y_1) + x_0 y_0$ we can approximate $x \cdot y$ as

$$\widehat{x \cdot y} = x_1 y_1 2^{2k} + 2^k(x_1 y_0 + x_0 y_1) \quad (20)$$

As in the previous case, we build a recursive algorithm as shown in fig. 3. Iterations that use formula (20) are pink while the Karatsuba iterations (18) are white. Assuming that $x_1 \cdot y_0$ and $x_0 \cdot y_1$ are also computed approximately, we get that the error of the approximation (20) is $\Delta(x \cdot y) = x \cdot y - \widehat{x \cdot y} = x_0 y_0 + 2^k(\Delta(x_1 \cdot y_0) + \Delta(x_0 \cdot y_1)) \in [0; 2^{2k} + 2^k(\Delta(x_0 \cdot y_1) + \Delta(x_1 \cdot y_0))]$. Using this, we can bound the error $\Delta(x \cdot y)$ of the recursive algorithm in fig. 3 as follows

$$\begin{aligned} 0 \leq \Delta(x \cdot y) &< 2^{2k_0} + 2^{k_0}(\Delta(z_1) + \Delta(z_2)) < \\ &2^{2k_0} + 2^{k_0}(2^{2k_{12}} + 2^{2k_{13}} + 2^{k_{12}}(\Delta(z_{22}) + \Delta(z_{23})) + \\ &2^{k_{13}}(\Delta(z_{32}) + \Delta(z_{33}))) < \dots = \Delta^{\{k_{ij}\}} \end{aligned} \quad (21)$$

Here $\Delta^{\{k_{ij}\}}$ is the upper bound on the error $\Delta(x \cdot y)$ which depends on every k_{ij} . This means that when we use

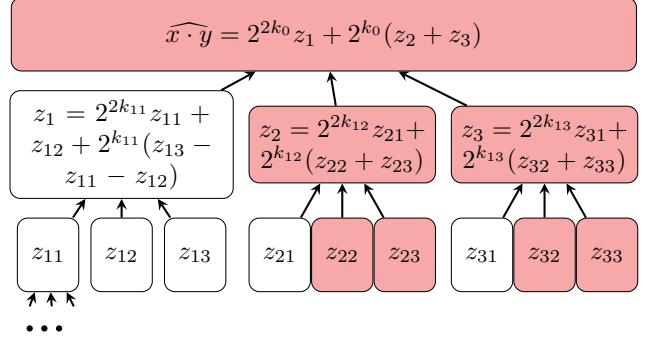


Fig. 3. MSB multiplication

the algorithm in fig. 3 in formula (17), \hat{l}_1 will be computed with an additional error of $\left\lceil \frac{\Delta^{\{k_{ij}\}}}{2^n} \right\rceil$. Thus the total error is upper bounded by

$$e(\hat{l}_1) < 3 + \left\lceil \frac{\Delta^{\{k_{ij}\}}}{2^n} \right\rceil \quad (22)$$

C. Optimizing FPGA resources for integer multipliers

Two out of three multipliers in figure 1 are multiplications by constants (m and q). This fact can be used to perform extra optimizations at the hardware level. Since our target hardware for this work is FPGA, we now present FPGA specific analysis. The technique can be replicated for other hardware with a different set of constraints.

The main basic logic blocks of FPGAs are DSP blocks and Look-Up Tables (LUTs). A straightforward way to implement integer multiplication at the leaves of the Karatsuba tree is by using DSP blocks. This is the most efficient way to multiply two unknown numbers. However, because the constants are known in advance, the cost can be reduced by implementing some of the multiplications using LUTs instead.

The exact resource optimization depends on the capacities of DSP and LUT of a specific FPGA. The general cost function for multiplication by a constant A can be defined recursively as:

$$C_t(A, \{k_{ij}\}) = \begin{cases} 1 & LUTs(A) > t \text{ and } bits(A) \leq DSP_size, \\ LUTs(A)/t & LUTs(A) \leq t, \\ C_t(A_0, \{k_{ij}\}) + C_t(A_1, \{k_{ij}\}) + C_t(A_2, \{k_{ij}\}) & \text{else} \end{cases} \quad (23)$$

Here A_0, A_1, A_2 are the three Karatsuba terms which depend on k_{ij} and on the Karatsuba equation that is used (either (18), (19) or (20)). $LUTs(A)$ is a function that returns the number of LUTs required to implement the multiplication by a constant A . $bits(A)$ returns the bit-length of A . DSP_size is the maximum bit-length of the input to a DSP block. This recursive cost function is linear in the number of LUTs and has a hard threshold t that can be interpreted as the number of LUTs, above which it's better to use a DSP block instead.

The Karatsuba tree enables some flexibility in the choice of $\{k_{ij}\}$. Once we have this search space and a cost to be minimized, we can define and solve the optimization problems for q (the LSB multiplier on fig. 2) and m (the MSB multiplier on fig. 3):

$$\begin{aligned} & \underset{\{k_{ij}\}}{\text{minimize}} \quad C_t(q, \{k_{ij}\}) \\ & \text{subject to} \quad n + \sigma_l \leq 2k_i + k_{i-1} + k_{i-2} + \dots + k_0 \\ & \underset{\{k_{ij}\}}{\text{minimize}} \quad C_t(m, \{k_{ij}\}) \\ & \text{subject to} \quad n - \sigma_m \geq 2k_i + k_{i-1} + k_{i-2} + \dots + k_0 \end{aligned}$$

The problems are solved by iterating on the splitting options $\{k_{ij}\}$ using dynamic programming. In the end, each product with cost 1 is implemented as a DSP block and if the cost is lower, LUTs are used. The threshold t is chosen such that the total number of DSP blocks is equal to the available amount. σ_m is chosen such that the error bound $\Delta^{\{k_{ij}\}}$ in (21) stays small enough. σ_l is found using (9) after the resulting error bound is calculated from (22). See appendix A for further details concerning LUTs and a demonstration of the technique for a specific FPGA.

III. ELLIPTIC CURVE ADDITION: COMPLETE FORMULAE

Many elliptic curve addition formulae are not complete, meaning they break for certain special cases, like adding two equal points or adding a point at infinity. To avoid handling these exceptions, we work with complete formulae that were originally introduced by [24]. Despite them being available for a decade or so now, they are not employed in practice since they have a larger number of modular operations as compared to other EC addition formulae in different coordinate systems [25].

Unfortunately, many of the known EC addition formulae in Jacobian or affine coordinates that are used in practice lead to high latency. In this paper, we show that the complete formulae [24] in homogeneous projective coordinates lead to a low-latency implementation in hardware. We follow the presentation of the formulae discussed in [26] (algorithm 7) for prime order curves. Although our techniques can be applied to many different curves, in this paper we consider a Barreto-Lynn-Scott type pairing-friendly curve BLS12-377 [27]–[29]. The equation for \mathbb{G}_1 of BLS12-377 in Weierstrass form is given by

$$y^2 = x^3 + 1 \quad (24)$$

where (x, y) are the affine coordinates of a point. The base field \mathbb{F}_q is given by a 377-bit prime q^1 and the scalar field \mathbb{F}_p is given by a 253-bit prime p . Exact values of p and q can be found in [27].

We rewrite (24) in homogeneous projective coordinates by substituting $(x, y) \rightarrow (X/Z, Y/Z)$ and the curve takes the form

$$Y^2 Z = X^3 + Z^3 \quad (25)$$

¹meaning that every number and every equation in this section is modulo q

The complete formulae for adding points $P : (X_1, Y_1, Z_1)$ and $Q : (X_2, Y_2, Z_2)$ are given by

$$\begin{aligned} X_3 &= (X_1 Y_2 + X_2 Y_1)(Y_1 Y_2 - 3Z_1 Z_2) \\ &\quad - 3(Y_1 Z_2 + Y_2 Z_1)(X_1 Z_2 + X_2 Z_1) \\ Y_3 &= (Y_1 Y_2 + 3Z_1 Z_2)(Y_1 Y_2 - 3Z_1 Z_2) \\ &\quad + 9X_1 X_2 (X_1 Z_2 + X_2 Z_1) \\ Z_3 &= (Y_1 Z_2 + Y_2 Z_1)(Y_1 Y_2 + 3Z_1 Z_2) \\ &\quad + 3X_1 X_2 (X_1 Y_2 + X_2 Y_1) \end{aligned} \quad (26)$$

Naively, formulae (26) appear to require 15 modular multiplications, 13 additions, and 4 multiplications by 3. Figure 4 shows how the same computation can be done using just 12 multiplications, 17 additions, and 3 multiplications by 3. Due to its repetitive nature and homogeneity in the degree of coordinates, these formulae are highly parallelizable and hardware-friendly.

The circuit in fig. 4 is implemented in a pipeline and optimized for low latency. It takes in as inputs points P and Q and outputs coordinates (X_3, Y_3, Z_3) of the result $P + Q$.

IV. MULTI-SCALAR MULTIPLICATION (MSM)

This section is independent of the specific curve or field in question. Henceforth, we will provide a general description. Let \mathcal{G} be an elliptic curve group of prime order p . Let $G = [G_0, G_1, \dots, G_{N-1}] \in \mathcal{G}^N$ and $x = [x_0, x_1, \dots, x_{N-1}] \in \mathbb{F}_p^N$ be N -element vectors of elliptic curve points and scalars, respectively. As mentioned in the introduction, MSM is a problem of computing

$$\begin{aligned} MSM(x, G) &= \sum_{n=0}^{N-1} x_n \cdot G_n = \\ &x_0 \cdot G_0 + x_1 \cdot G_1 + \dots + x_{N-1} \cdot G_{N-1} \end{aligned} \quad (27)$$

Note that plus signs here refer to the elliptic curve addition. The scalar multiplication $x_n \cdot G_n$ refers to adding G_n to itself x_n times. Denote by

$$b = \lceil \log_2 p \rceil \quad (28)$$

the maximum number of bits in x .

As we discussed in the introduction §I, state-of-the-art algorithm to solve the MSM problem (27) is known as the Pippenger algorithm [1], and its variant that is widely used in the ZK space is called the bucket method [30] (see "Overlap in the Pippenger approach" in section 4). We describe it and our proposed improvements in the following section. There, we treat EC additions (PADD) and doublings (PDBL) as atomic operations, and considering our usage of complete formulae (26), we assume the costs of PADD and PDBL to be equal.

A. The bucket method

Let us partition each x_n from equation (27) into K parts such that each partition consists of c bits and $K = \lceil \frac{b}{c} \rceil$.

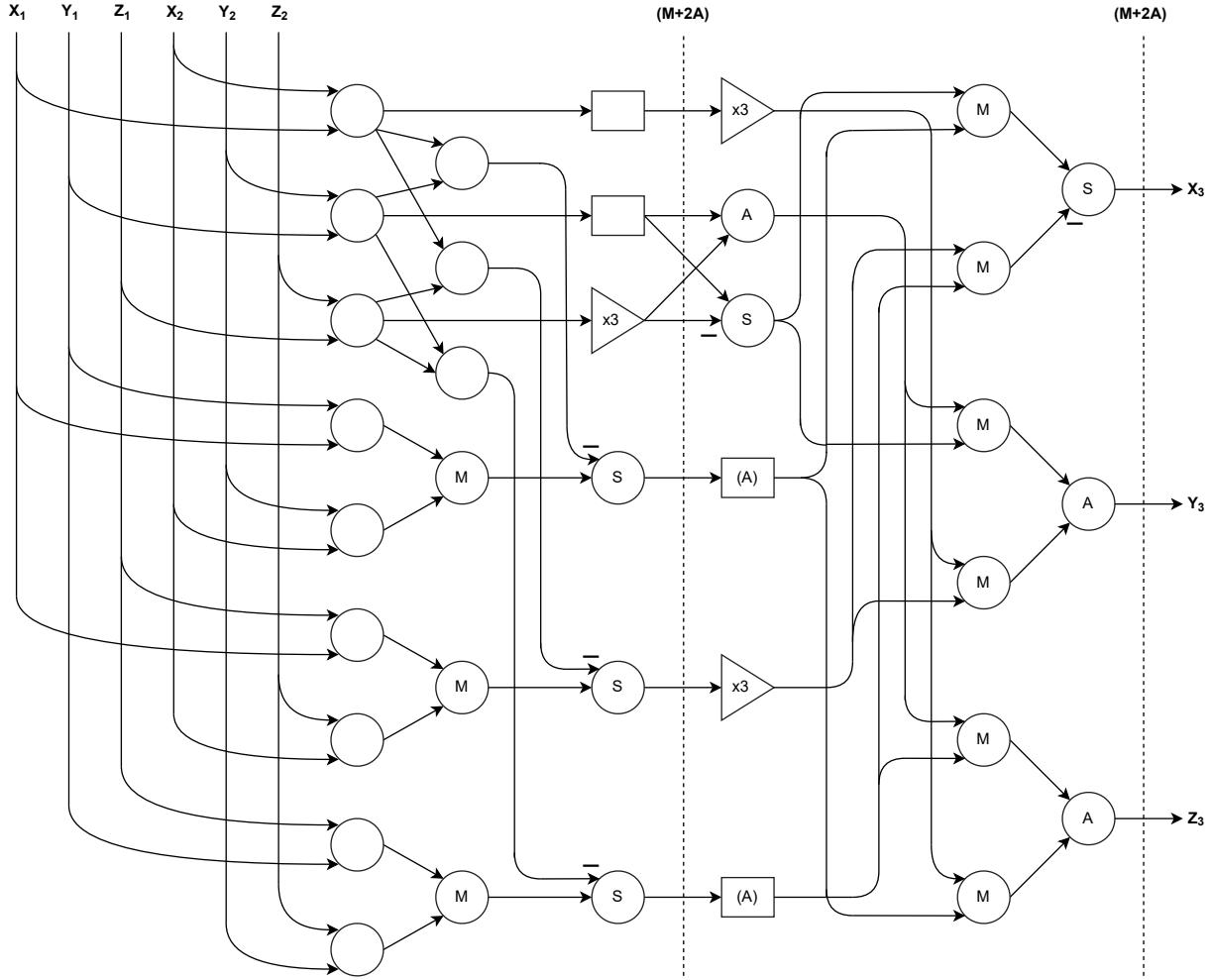


Fig. 4. Low latency elliptic curve addition module. $M, A/S$ refer to Multiplier and Adder/Subtractor, respectively. Triangles are multipliers by 3 and rectangles show the delays that occur due to sampling of intermediate values.

Denoting by $x_n^{[k]}$ the k th partition of x_n , equation (27) can be rewritten as

$$G = \sum_{n=0}^{N-1} \sum_{k=0}^{K-1} 2^{kc} x_n^{[k]} G_n = \sum_{k=0}^{K-1} 2^{kc} \sum_{n=0}^{N-1} x_n^{[k]} G_n = \sum_{k=0}^{K-1} 2^{kc} G^{[k]} \quad (29)$$

where $G^{[k]} = \sum_{n=0}^{N-1} x_n^{[k]} G_n$ is the result of computing multi-scalar-multiplication with c -bit scalars $\{x_n^{[k]}\}$. Clearly, if we computed $G^{[k]}$ for each k , then the last step in (29) can be completed using Horner's rule as

$$G = 2^c (\dots (2^c (2^c G^{[K-1]} + G^{[K-2]}) + G^{[K-3]}) \dots) + G^{[0]} \quad (30)$$

Thus we reduced the MSM problem to computing the sums $G^{[k]}$. The total number of operations required in (30) can be broken down as follows:

- 1) Computing K partial sum $G^{[k]}$
- 2) $K - 1$ PADDs attributed to the summations
- 3) $c(K - 1)$ PDDBLs attributed to the 2^c multiplications

We now proceed to describe an efficient algorithm to calculate $G^{[k]}$. The algorithm involves iterating over G_n , placing each one in a "bucket" corresponding to its coefficient $x_n^{[k]}$. Clearly, the number of non-zero buckets is $2^c - 1$. Per step, each bucket $l \in [1, 2^c - 1]$ acts as an accumulator, adding an assigned G_n to its current sum $B_l^{[k]}$. Buckets initiate at zero. For each k , the number of PADDs to compute all $B_l^{[k]}$ is $N + 1 - 2^c$. The partial sum $G^{[k]}$ is calculated as

$$G^{[k]} = \sum_{n=0}^{N-1} x_n^{[k]} \cdot G_n = \sum_{l=1}^{2^c-1} l \cdot B_l^{[k]} \quad (31)$$

which can be computed efficiently as the sum of the following recursive series

$$S_l^{[k]} = \sum_{i=1}^l B_i^{[k]} = S_{l-1}^{[k]} + B_l^{[k]} \quad (32)$$

thus

$$G^{[k]} = \sum_{l=1}^{2^c-1} S_l^{[k]} \quad (33)$$

We present the bucket method in algorithm 1. For the convenience of presentation, we partition the bucket method algorithm into three distinct parts, referred to as Loop 1, 2, and 3 hereafter.

Algorithm 1 Basic bucket method

```

1: Set  $B_l^{[k]} = 0$ ,  $\forall k \in [0, K - 1]$ ,  $\forall l \in [1, 2^c - 1]$  ▷ The 1st loop
2:
3: for  $n = 0, 1, \dots, N - 1$  do ▷ N inputs
4:   for  $k = 0, 1, \dots, K - 1$  do ▷ K partial sums
5:     Set  $l = x_n^{[k]}$  ▷ Continue (skip) if  $l = 0$ 
6:      $B_l^{[k]} \leftarrow B_l^{[k]} + G_n$  ▷ Partial bucket sums  $B_l^{[k]}$ 
7:   end for
8: end for
9:
10: Set  $S^{[k]} = 0, G^{[k]} = 0$  ▷ The 2nd loop
11: for  $l = 2^c - 1, 2^c - 2, \dots, 1$  do ▷ Loop over buckets
12:   for  $k = 0, 1, \dots, K - 1$  do ▷ K partial sums
13:      $S^{[k]} \leftarrow S^{[k]} + B_l^{[k]}$  ▷ Partial sums  $G^{[k]}$ 
14:      $G^{[k]} \leftarrow G^{[k]} + S^{[k]}$ 
15:   end for
16: end for ▷ The 3rd loop
17:
18: Set  $G = 0$  ▷ K partial sums
19: for  $k = K - 1, K - 2, \dots, 0$  do ▷ Horner's rule
20:    $g \leftarrow 2^c G + G^{[k]}$ 
21: end for

```

Cost of algorithm 1: One way to define the cost of 1 is to count the number of required EC operations. In figure 5 we plot this number against the parameter c and show the optimal values of c for various input sizes N .

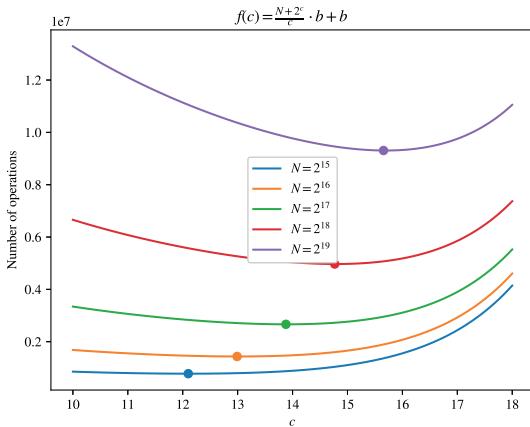


Fig. 5. The number of EC operations vs c , assuming $b = 253$ bits for various input sizes N . Dots represent the values of c that minimize $f(c)$.

A different way to define the cost of an algorithm is the expected span in clock cycles it requires to run once from

start to completion. We'll use this cost function throughout the text. To that end, consider a fully pipelined PADD/PDBL kernel with a total delay of d clock cycles. This means that the kernel handles inputs for one EC operation every clock cycle and the result is returned after d clock cycles.

Loop 1 runs $(N - 2^c)K$ times with the delay of 1 between two consecutive iterations. The cost of Loop 1 is thus $(N - 2^c)K = \frac{(N-2^c)b}{c}$. In Loop 2, we need to perform $2^{c+1}K$ EC operations, each taking d clock cycles and $2K$ of them can be performed in parallel at a time (assuming $2K$ is less than d which is true in practice). The cost of Loop 2 is thus $\frac{2^{c+1}Kd}{2K} = d2^c$. Loop 3 contains $(K - 1)(c + 1)$ serial EC operations, taking around db clock cycles. Summing the costs of all loops provides the following cost function

$$f(c) = (N - 2^c)bc^{-1} + d(2^c + b) \quad (34)$$

B. An improvement to Loop 2 - segmentation of buckets

While the first loop in algorithm 1 can be parallelized, the second loop consists of K highly sequential computations, and only $2K$ parallel additions can be done at a time. To solve this issue, we break the second loop into M segments. Thus, instead of computing (31) we compute a segmented version of the buckets

$$\begin{aligned} G^{[k]} = & \left(1 \cdot B_1^{[k]} + 2 \cdot B_2^{[k]} + \dots + U \cdot B_U^{[k]} \right) \\ & + \left((U + 1) \cdot B_{U+1}^{[k]} + (U + 2) \cdot B_{U+2}^{[k]} + \dots + 2U \cdot B_{2U}^{[k]} \right) \\ & + \dots \\ & + \left((U(M - 1) + 1) \cdot B_{U(M-1)+1}^{[k]} + \right. \\ & \left. (U(M - 1) + 2) \cdot B_{U(M-1)+2}^{[k]} + \dots + (2^c - 1) \cdot B_{2^c-1}^{[k]} \right) \end{aligned} \quad (35)$$

where $U = \lceil \frac{2^c - 1}{M} \rceil$. We compute each of M segment sums separately, which can be done in parallel using the algorithm 2. Then the results are aggregated to obtain $G^{[k]}$.

Trade-offs: In the basic version 1, the computation of each $G^{[k]}$ includes $2 \cdot (2^c - 2)$ PADD. At every moment in time, 2 of these can be performed in parallel. In the segmented version, $2 \cdot (2^c - 2 - M)$ PADD are performed; $2M$ can be performed in parallel, and we suppose that M is large enough to fully occupy the pipelined adder. $\log_2(U)$ more PDBL² and $3M$ PADD need to be performed to get the final result $G^{[k]}$. Thus, the segmented version requires $K(M + \log_2(U))$ more EC operations while making most of the second loop more parallelizable. We get the total cost in clock cycles as

$$f(c) = (N + 2^c)bc^{-1} + d(2M + K + b) + 2K(2^c - 2 - M) \quad (36)$$

C. Simulating the bucket method

In this section, we discuss the performance of the segmented bucket method presented in algorithm 2. As mentioned above,

²Here we assume that U is a power of two so that computing the scalar multiplication by U only takes $\log_2(U)$ PDBL.

Algorithm 2 Buckets segmentation for the 2nd loop

```

1: ▷ The 2nd loop
2: Set  $S^{[k,m]} = 0, G^{[k,m]} = 0$  ▷ 2.0
3: Def  $U \equiv \frac{2^c}{M}$  ▷ Assume that  $M$  is a power of two
4: for  $u = 1, 2, \dots, U$  do ▷  $U$  segment buckets
5:   for  $m = 0, 1, \dots, M - 1$  do ▷  $M$  segments
6:     Set  $l = U(M - m) - u$  ▷ Skip if  $l = 0$ 
7:     for  $k = 0, 1, \dots, K - 1$  do ▷  $K$  partial sums
8:        $G^{[k,m]} \leftarrow G^{[k,m]} + S^{[k,m]}$ 
9:        $S^{[k,m]} \leftarrow S^{[k,m]} + B_l^{[k]}$ 
10:    end for
11:   end for
12: end for
13:
14: Set  $S^{[k]} = 0$  ▷ 2.1
15: Def  $S^{[k,-1]} \equiv 0$ 
16: for  $m = 0, 1, \dots, M - 2$  do ▷  $M - 1$  segment
17:   for  $k = 0, 1, \dots, K - 1$  do ▷  $K$  partial sums
18:      $S^{[k,m]} \leftarrow S^{[k,m]} + S^{[k,m-1]}$ 
19:      $S^{[k]} \leftarrow S^{[k]} + S^{[k,m]}$ 
20:   end for
21: end for
22:
23: Def  $v \equiv \log_2 U$  ▷ 2.2
24: for  $v$  cycles do
25:   for  $k = 0, 1, \dots, K - 1$  do ▷  $M$  segments
26:      $S^{[k]} \leftarrow S^{[k]} + S^{[k]}$  ▷  $K$  partial sums
27:   end for
28: end for
29:
30: Set  $G^{[k]} = 0$  ▷ 2.3
31: for  $m = 0, 1, \dots, M - 1$  do ▷  $M$  segments
32:   for  $k = 0, 1, \dots, K - 1$  do ▷  $K$  partial sums
33:      $G^{[k]} \leftarrow G^{[k]} + G^{[k,m]}$ 
34:   end for
35: end for
36:
37: ▷ 2.4
38: for  $k = 0, 1, \dots, K - 1$  do ▷ Loop over  $K$  partial sums
39:    $G^{[k]} \leftarrow G^{[k]} + S^{[k]}$ 
40: end for

```

we work with the BLS12-377 curve [27], [28]. So the scalars $x_i \in \mathbb{F}_p$ in (27) are 253 bit integers, while the base field elements $G_i \in \mathbb{F}_q$ are 377 bit integers.

We assume that both scalars and points are variable inputs. In fig. 6, we present a high level design that executes the algorithm 2 in three phases

- 1) Bucket accumulation
- 2) Segments sum
- 3) Final accumulation

For the simulations in this section, we choose the number of segments in Loop 2 to be $M = 8$. The parameter c is varied. In our example in fig. 6, $c = 9$ so the design includes 29 bucket accumulators that are divided into 8 segments, each of them enclosing 64 buckets.

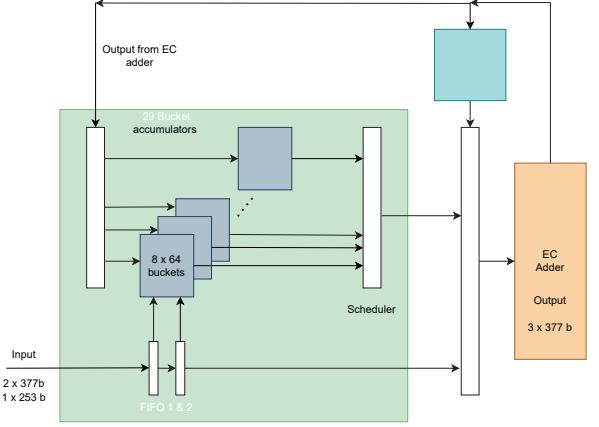


Fig. 6. Bucket method module for $c = 9$.

The EC adder module is a hardware unit that executes the addition of projective coordinates as described in §III. For the purposes of this section the EC adder module is a black box that works for d clock cycles before returning the result. However, the EC adder can receive a new task every clock.

Once the input enters the bucket accumulator module (the green box in fig. 6), the point is distributed to the buckets, defined by its scalar. If the bucket is empty then the input is written to it. If the bucket already contains a point then the accumulator sends an addition task to the EC adder with the current value and the new input. The result is sent back to the appropriate bucket. The bucket accumulator handles the output from the EC adder in higher priority than new input data. In Loop 2, the points in each of the segments are scheduled to be summed by the EC adder module. Thus, the output of the EC adder is sent back to the segments until all the data in the segments is exhausted, following which the data is sent to the final accumulator at the end of Loop 2. The final accumulator accumulates the results of all the segmented sums and uses the EC adder to perform Loop 3. While the final accumulator functions, the bucket accumulators are free to handle the next task.

Our operating design for the EC adder discussed in §III gives a delay of $d = 115$ clocks. Using a simulator written in Python and our design for input sizes $N = 2^{15}, 2^{17}$ and find that the optimum run time is obtained for $c = 12, 14$ respectively in tight agreement with our theoretical prediction plotted in fig 5. Our simulated results for $N = 2^{15}$ and $N = 2^{17}$ are plotted in fig 7.

Furthermore, it is possible to squeeze in more than one EC adder in our design. To get a sense of how efficient that would be, we plot the rate of MSM computation as a function of c in fig. 8. For optimal c , we see that the time for computing an MSM almost halves with the introduction of the second EC adder. Our simulations also indicate that the idle rate for two EC adders stays insignificant (see fig 9) and can be neglected.

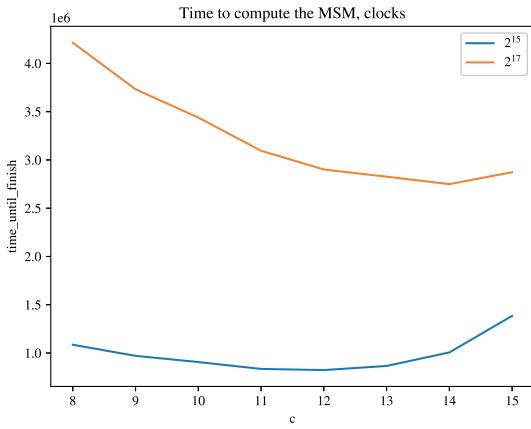


Fig. 7. We have plotted the bucket module run to finish vs c for a input vector of size $N = 2^{15}$, $N = 2^{17}$ and we find that the optimal run time is obtained when $c = 12$ and $c = 14$ respectively.

V. IMPLEMENTATION

In our current implementation, we use the design depicted in fig. 6. We choose parameters $c = 12$, $M = 8$, where M is the number of segments from algorithm 2. We use a single EC adder module, its delay is equal to 115 clock cycles. The clock speed is set to 125MHz.

To compare our implementation with the state-of-the-art GPU implementation, Sppark [18], we measure the delay and peak power draw of both. For our implementation we used Xilinx Alveo U55C, and for Sppark we used the Nvidia RTX3090 family, chosen as they outperformed all other GPU cards we tested: A100, V100, RTX5000, RTX6000.

Our benchmark on the two systems is done by taking two equally-sized vectors of scalars and EC points as inputs. It is worth noting that one likely scenario is for the EC points vectors to be precomputed in memory which reduces significantly the total running time³. Additionally we compute energy consumption per MSM which measures the monetary costs of computation. The performance is measured on MSMs of sizes 2^{15} to 2^{20} . The results, which agree with our simulations (see subsection IV-C), are summarized in table III.

As can be seen, our implementation is significantly better in terms of the peak power consumption and competitive in terms of delay, especially for smaller sizes. Also note that the computation time for Sppark grows slower than one might expect. This is due to the under-utilization of resources by the GPU for smaller-sized MSMs, as evident from the growing power consumption.

For future work we suggest three directions for improving our implementation:

- 1) *Increase clock frequency*: For our proof of concept we used clock frequency of 125MHz. With some engineering effort we believe it can be increased up to 500MHz. This immediately results in 4x speed improvement.

³Measuring running time at the host includes time to write to the hardware. Since the hardware part runs fast, we are sensitive to how the host-hardware interface is implemented and how much data goes through.

- 2) *Add EC adders*: As seen on figures 8 and 9, adding one more EC adder, doubles the performance.
- 3) *Further algorithmic performance improvements*: See section VI.

VI. IMPROVEMENT PROPOSALS

In this section, we present potential improvements we have not yet implemented in hardware.

A. Signed scalars

Here we describe a method that utilizes the cheap negation in elliptic curve groups. In affine coordinates $P : (x, y)$ we can write

$$P \rightarrow -P \equiv (x, y) \rightarrow (x, -y) \quad (37)$$

Starting from the definition of the basic bucket method (29) we note that $x_n^{[k]} \in [1, 2^c - 1]$. To make use of the property (37), we need negative scalars, so we aim to shift the scalar set: $x_n^{[k]} \in [-2^{c-1}, 2^{c-1} - 1]$. A simple way to do this is to parse through $x_n^{[0]}, x_n^{[1]}, \dots, x_n^{[K-1]}$ and, if $x_n^{[k]}$ is larger than $2^{c-1} - 1$, subtract 2^c and increment the $k + 1$ 'st scalar by 1.

If the highest $K - 1$ 'st digit is greater than 2^{c-1} then there is a potential overflow. In the case of BLS12-377 [27] 253-bit scalars will have some extra bits to offset the overflow provided c does not divide 253 (so, $c \neq 11$ and $c \neq 23$). The algorithm for shifting between the representations is given in algorithm 3 and can be run on the fly in Loop 1 of algorithm 1.

Algorithm 3 Conversion of c -bit scalars to signed scalars

```

1: input:  $x_n^{[k]} \in [0, 2^c - 1] \forall k \in 0, 1, \dots, K - 1$ 
2: for  $k = 0, 1, \dots, K - 1$  do
3:   if  $x_n^{[k]} \geq 2^{c-1}$  then
4:      $\tilde{x}_n^{[k]} \leftarrow x_n^{[k]} - 2^c$                                  $\triangleright$  Subtract  $2^c$ 
5:      $\tilde{x}_n^{[k+1]} \leftarrow 1 + x_n^{[k+1]}$        $\triangleright$  Add 1 to the next scalar
6:   end if
7: end for
8: return  $\tilde{x}_n^{[k]} \in [-2^{c-1}, 2^{c-1} - 1] \forall k \in 0, 1, \dots, K - 1$ 

```

In Loop 1, we include a check for the sign of the scalar, which leads to a decrease in the number of buckets: from 2^c to 2^{c-1} . The rest of the algorithm can proceed as before including the segmentation improvement 2. The general idea is illustrated in algorithm 4. The new cost is

$$f(c) = (N + 2^{c-1}) \frac{b}{c} + d \left(\log_2 \left(\left\lceil \frac{2^{c-1} - 1}{M} \right\rceil \right) + 2M + K \right) \quad (38)$$

This allows a free (memory-wise) increase $c \rightarrow c + 1$, decreasing the cost of the Loop 1 by a factor of $\frac{c+1}{c}$.

Finally, we mention that negation is just one example of the so-called "endomorphisms" which can be cheaply computed on curves of the BLS family. For an overview of other such endomorphisms and difficulties in applying them to the bucket method, see [31].

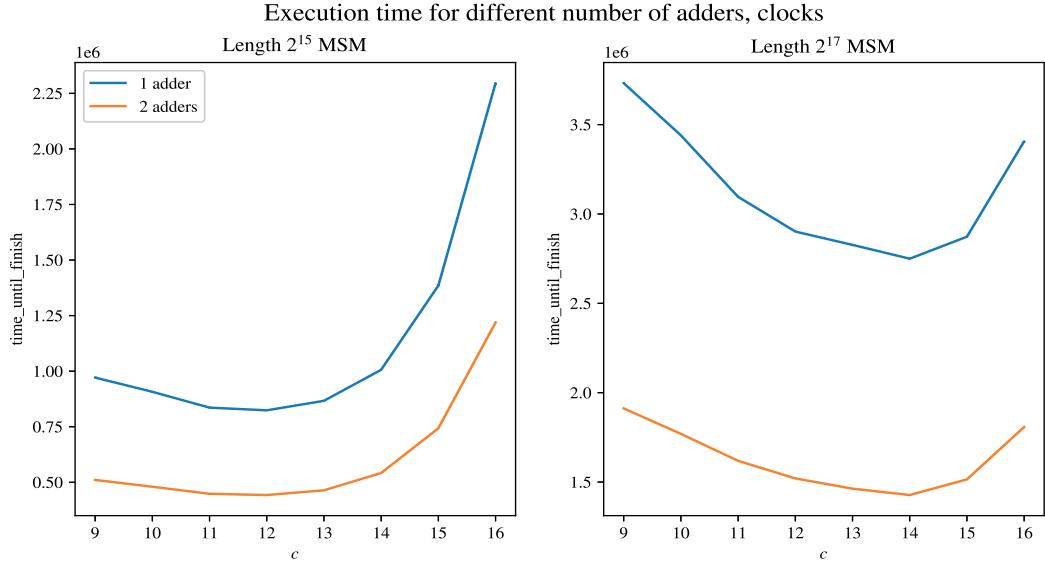


Fig. 8. Execution times for the sizes $N = 2^{15}$ and $N = 2^{17}$ respectively for various c . We note that the rate almost doubles when using 2 EC adders.

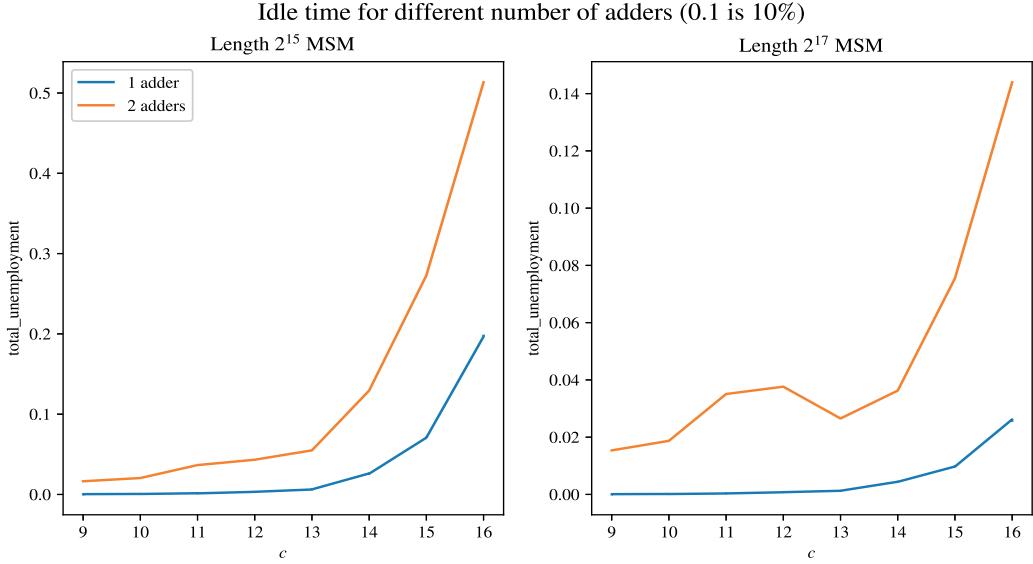


Fig. 9. Idle rates of EC adders for $N = 2^{15}$ and $N = 2^{17}$ respectively for various c . We note that the idle rate increases with the increase in the number of EC adders

B. Batched bucket method

This is a proposed improvement to the first loop of algorithm 1, which also requires a change in the way we do EC addition. Namely, we switch to representing points in affine coordinates. Since the affine addition includes field inversion, we first discuss how to offset the high computational costs of inversions using the so-called Montgomery trick.

1) *Montgomery trick:* Suppose that we want to invert $m > 1$ numbers: a_1, a_2, \dots, a_m modulo q . We can avoid performing many computationally costly inversions by using the so-called Montgomery trick. It is summarized in algorithm 5. It is easy to see that algorithm 5 performs $3(m - 1)$ multiplications and 1 inversion instead of m inversions. Since

in most cases modular multiplication is more than 3 times faster than inversion, the Montgomery trick is a very useful maneuver, provided that several field elements can be inverted "in parallel". For instance, it does not work if the input to one inversion depends on the other inversion being performed first.

The classical Montgomery trick as defined above is serial. It is, however, possible to parallelize the Montgomery trick, as shown in algorithm 6.

Analyzing this algorithm, we see that the first loop requires $L(R - 1)$ multiplications. The inner Montgomery trick used to invert the final partial product in each batch takes $3(L - 1)$ multiplications. Finally, the second loop takes $2L(R - 1)$

Implementation	Hardware	Length of MSM	Time, ms.	Peak power, watts	Energy per MSM, joules
This work	U55C	2^{15}	9.8	18.4	0.174
		2^{16}	17.6	29.1	0.414
		2^{17}	35.9	33.6	0.976
		2^{18}	68.8	34.8	2.15
		2^{19}	136.6	34.8	4.50
		2^{20}	273.0	34.9	9.23
Sppark [18]	RTX 3090	2^{15}	9.1	173	1.44
		2^{16}	10.6	210	2.15
		2^{17}	14.6	287	3.34
		2^{18}	21.4	282	5.39
		2^{19}	33.7	384	8.99
		2^{20}	53.9	465	14.5

TABLE III
THE COMPARISON OF OUR FPGA IMPLEMENTATION AND SPPARK

Algorithm 4 The bucket method with signed scalars

```

1: Set  $B_m^{[k]} = 0$ ,  $\forall k \in [0, K - 1]$ ,  $\forall m \in [1, 2^{c-1}]$                                 ▷ The 1st loop
2: for  $n = 0, 1, \dots, N - 1$  do                                              ▷  $N$  inputs
3:   for  $k = 0, 1, \dots, K - 1$  do                                              ▷  $K$  partial sums
4:     Set  $l = x_n^{[k]}$                                                         ▷ Convert to signed scalars
5:     if  $l \geq 0$  then
6:        $B_l^{[k]} \leftarrow B_l^{[k]} + G_n$  ▷ Partial bucket sums  $B_l^{[k]}$ 
7:     else
8:        $B_{|l|}^{[k]} \leftarrow B_{|l|}^{[k]} - G_n$  ▷ Partial bucket sums  $B_{|l|}^{[k]}$ 
9:     end if
10:    end for
11:   end for
12: end for
13:
14: Set  $S^{[k]} = 0, G^{[k]} = 0$                                               ▷ The 2nd loop
15: for  $l = 2^{c-1}, 2^{c-1} - 1, \dots, 1$  do                                         ▷ Loop over  $2^{c-1}$  buckets
16:   for  $k = 0, 1, \dots, K - 1$  do                                              ▷  $K$  partial sums
17:      $S^{[k]} \leftarrow S^{[k]} + B_l^{[k]}$ 
18:      $G^{[k]} \leftarrow G^{[k]} + S^{[k]}$                                               ▷ partial sums  $G^{[k]}$ 
19:   end for
20: end for
21:
22: Set  $G = 0$                                                                ▷ The 3rd loop
23: for  $k = K - 1, K - 2, \dots, 0$  do                                         ▷  $K$  partial sums
24:    $G \leftarrow 2^c G + G^{[k]}$                                               ▷ Horner's rule
25: end for

```

multiplications. In total, we need $3(LR - 1) = 3(m - 1)$ multiplications, which is the same number as for the regular Montgomery trick. Thus, we parallelized the Montgomery trick with no additional computational cost.

The cost of the parallelized version is additional memory. Note that the lists t_i , as well as data for the inner Montgomery trick, need to be stored. This amounts to storing $m + L$ field elements, L more than the non-parallel version (algorithm 5).

In general, one can recursively execute the parallel Montgomery trick with more than one layer, in a tree-like structure.

Algorithm 5 The Montgomery Trick

```

1: input:  $a_i, \forall i = 1, 2, \dots, m$ 
2:  $t \leftarrow [a_1, \dots]$                                          ▷ Compute partial products
3:    $a_1 \cdot a_2 \pmod{q}$ ,
4:    $a_1 \cdot a_2 \cdot a_3 \pmod{q}$ ,
5:   :
6:    $a_1 \cdot a_2 \cdot a_3 \dots a_{m-2} \pmod{q}$ ,
7:    $a_1 \cdot a_2 \cdot a_3 \dots a_{m-2} \cdot a_{m-1} \pmod{q}$ ]
8:  $A \leftarrow (t[m-2] \cdot a_m)^{-1} \pmod{q}$           ▷ List  $t$  is 0-based
9: for  $j \in \{m, m-1, \dots, 3, 2\}$  do
10:    $a_j^{-1} \leftarrow A \cdot t[j-2] \pmod{q}$ 
11:    $A \leftarrow A \cdot a_j \pmod{q}$ 
12: end for
13:  $a_1^{-1} \leftarrow A$ .

```

Algorithm 6 Parallelized Montgomery Trick

```

1: input:  $a_i, \forall i = 1, 2, \dots, m$ 
2: for  $i \in \{0, 1, \dots, L - 1\}$  do:
3:    $t_i \leftarrow [a_{iR+1}, \dots]$  ▷ Parallel partial products in batches
4:    $a_{iR+1} \cdot a_{iR+2} \pmod{q}$ ,
5:   :
6:    $a_{iR+1} \cdot a_{iR+2} \dots a_{(i+1)R-2} \pmod{q}$ ,
7:    $a_{iR+1} \cdot a_{iR+2} \dots a_{(i+1)R-2} \cdot a_{(i+1)R-1} \pmod{q}$ ]
8:    $A_i \leftarrow t_i[R-2] \cdot a_{(i+1)R} \pmod{q}$ 
9: end for
10:  ▷ Invert  $\{A_i\}$  in-place using regular Montgomery trick
11:   $A_0, A_1, \dots, A_{L-1} \leftarrow \text{Montgomery}(A_0, A_1, \dots, A_{L-1})$ 
12:  for  $i \in \{0, 1, \dots, L - 1\}$  do:
13:    for  $j \in \{R, R-1, \dots, 3, 2\}$  do:
14:       $a_{iR+j}^{-1} \leftarrow A_i \cdot t_i[j-2] \pmod{q}$ 
15:       $A_i \leftarrow A_i \cdot a_{iR+j} \pmod{q}$ 
16:    end for
17:     $a_{iR+1}^{-1} \leftarrow A_i$ 
18:  end for

```

In any case, the number of multiplications remains the same and the memory footprint in the extreme case of a binary tree will be $2m$, or 2 times larger than for the non-parallel version.

To use the Montgomery trick, we will abstract away the details of its implementation. Instead, we'll think of it as a stack of field elements that, after being populated, can return the inverses of these elements in the reverse order. In pseudocode, we denote by $.push(a)$ a method that pushes a into the stack; this corresponds to the loop at lines 2-9 of algorithm 6. Method $.invert()$ computes the inner Montgomery trick at line 12. Finally, $.iter()$ successively iterates over the inverses of inputs in reverse order and stops when the data structure is empty; this happens at lines 13-19 in 6.

2) *Batched bucket method:* We adopt the notation from the basic bucket method discussed in algorithm 1. The buckets $B_l^{[k]}$ now store a single elliptic curve point in affine coordinates. We assume that the buckets are also equipped with a method $.pop()$ that returns the contents and empties the bucket. We also maintain a stack t of capacity t_{max} that will hold pairs of affine EC points and bucket indices. When the stack reaches its maximum capacity, the inner Montgomery trick is called. The stack is initialized empty and has standard methods like $.len()$, $.push()$, and $.pop()$. By M we denote the Montgomery trick stack discussed earlier.

Algorithm 7 Batched version of the Loop 1 of the bucket method

```

1: for  $n = 0, 1, 2, \dots, N - 1$  do            $\triangleright$  Receive  $n$ -th scalar
2:   for  $k = 0, 1, 2, \dots, K - 1$  do        $\triangleright$  Working with  $G^{[k]}$ 
3:      $l \leftarrow x_n \pmod{2^c}$                    $\triangleright$  Bucket index
4:
5:     if  $l > 0$  then                       $\triangleright$  Ignore bucket 0
6:        $\triangleright$  Batch invert only when stack  $t$  is full
7:       if  $\text{length}(t) = t_{max}$  then
8:          $M.\text{invert}()$                     $\triangleright$  Inner Montgomery
9:         for  $d$  in  $M.\text{iter}()$  do
10:           $(A, B, \hat{k}, \hat{l}) \leftarrow t.pop()$ 
11:           $B_{\hat{l}}^{[\hat{k}]} \leftarrow \text{ECadd}(A, B, d)$   $\triangleright$  Algorithm 8
12:        end for
13:      end if
14:
15:      if  $B_l^{[k]} = \text{NULL}$  then           $\triangleright$  Empty bucket
16:         $B_l^{[k]} \leftarrow G_n$               $\triangleright$  Write  $G_n$  into the bucket
17:      else                            $\triangleright$  Non-empty bucket
18:         $G \leftarrow B_l^{[k]}.pop()$          $\triangleright$  Pop a point
19:         $M.push(x_G - x_{G_n})$ 
20:         $t.push((G, G_n, k, l))$ 
21:      end if
22:    end if
23:
24:     $x_n \leftarrow \lfloor \frac{x_n}{2^c} \rfloor$        $\triangleright$  Removing  $c$  last bits from  $x_n$ 
25:  end for
26: end for

```

The EC addition referred to in line 11 of algorithm 7 is done using the standard affine formulae. The only difference is that the denominators are precomputed (see algorithm 8).

Algorithm 8 Affine addition of elliptic curve points

```

1: input:  $G_1, G_2, d$ 
2:  $(x_1, y_1) \leftarrow G_1$             $\triangleright$  Coordinates of the point  $G_1$ 
3:  $(x_2, y_2) \leftarrow G_2$             $\triangleright$  Coordinates of the point  $G_2$ 
4:  $\lambda \leftarrow (y_1 - y_2) \cdot d \pmod{q}$ 
5:  $x_3 \leftarrow \lambda^2 - x_1 - x_2 \pmod{q}$ 
6:  $y_3 \leftarrow \lambda(x_1 - x_3) - y_1 \pmod{q}$ 
7: return  $(x_3, y_3)$ 

```

REFERENCES

- [1] N. Pippenger, “On the evaluation of powers and related problems,” in *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*, 1976, pp. 258–263.
- [2] J. Thaler, “Proofs arguments and zero knowledge,” <https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.pdf>.
- [3] T. Ingonyama, “Ingopedia,” <https://github.com/ingonyama-zk/ingopedia/blob/main/README.md>.
- [4] M. Petkus, “Why and how zk-snark works,” 2019. [Online]. Available: <https://arxiv.org/abs/1906.07221>
- [5] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, “Quadratic span programs and succinct nizks without pcp,” Cryptology ePrint Archive, Paper 2012/215, 2012, <https://eprint.iacr.org/2012/215>. [Online]. Available: <https://eprint.iacr.org/2012/215>
- [6] V. Buterin, “Quadratic arithmetic programs from zero to hero,” <https://medium.com/@VitalikButerin/quadratic-arithmetic-programs-from-zero-to-hero-f6d558cea649>.
- [7] A. Kate, G. M. Zaverucha, and I. Goldberg, “Constant-size commitments to polynomials and their applications,” in *Advances in Cryptology - ASIACRYPT 2010*, M. Abe, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 177–194.
- [8] J. Bootle, A. Cerulli, P. Chaidos, J. Groth, and C. Petit, “Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting,” Cryptology ePrint Archive, Paper 2016/263, 2016, <https://eprint.iacr.org/2016/263>. [Online]. Available: <https://eprint.iacr.org/2016/263>
- [9] B. Bünz, B. Fisch, and A. Szepieniec, “Transparent snarks from dark compilers,” Cryptology ePrint Archive, Paper 2019/1229, 2019, <https://eprint.iacr.org/2019/1229>. [Online]. Available: <https://eprint.iacr.org/2019/1229>
- [10] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, “Plonk: Permutations over lagrange bases for oecumenical noninteractive arguments of knowledge,” Cryptology ePrint Archive, Paper 2019/953, 2019, <https://eprint.iacr.org/2019/953>. [Online]. Available: <https://eprint.iacr.org/2019/953>
- [11] A. Chiesa, Y. Hu, M. Maller, P. Mishra, P. Vesely, and N. Ward, “Marlin: Preprocessing zksnarks with universal and updatable srs,” Cryptology ePrint Archive, Paper 2019/1047, 2019, <https://eprint.iacr.org/2019/1047>. [Online]. Available: <https://eprint.iacr.org/2019/1047>
- [12] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, “Bulletproofs: Short proofs for confidential transactions and more,” Cryptology ePrint Archive, Paper 2017/1066, 2017, <https://eprint.iacr.org/2017/1066>. [Online]. Available: <https://eprint.iacr.org/2017/1066>
- [13] J. Groth, “On the size of pairing-based non-interactive arguments,” Cryptology ePrint Archive, Paper 2016/260, 2016, <https://eprint.iacr.org/2016/260>. [Online]. Available: <https://eprint.iacr.org/2016/260>
- [14] M. Campanelli, A. Faonio, D. Fiore, A. Querol, and H. Rodríguez, “Lunar: a toolbox for more efficient universal and updatable zksnarks and commit-and-prove extensions,” Cryptology ePrint Archive, Report 2020/1069, 2020, <https://ia.cr/2020/1069>.
- [15] P. Barrett, “Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor,” in *Advances in Cryptology — CRYPTO’ 86*, A. M. Odlyzko, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 311–323.
- [16] P. L. Montgomery, “Modular multiplication without trial division,” Mathematics of Computation 44, no. 170 (1985): 519–21., <https://doi.org/10.2307/2007970>.
- [17] Z. Cao, R. Wei, and X. Lin, “A fast modular reduction method,” Cryptology ePrint Archive, Paper 2014/040, 2014, <https://eprint.iacr.org/2014/040>. [Online]. Available: <https://eprint.iacr.org/2014/040>
- [18] Supranational, “Sppark: Zero-knowledge template library,” <https://github.com/supranational/sppark>.
- [19] “Zcash fpga,” <https://github.com/ZcashFoundation/zcash-fpga>.

- [20] “Fpga snark prover targeting the bn128 curve,” https://github.com/bsdevlin/fpga_snark_prover.
- [21] Y. Zhang, S. Wang, X. Zhang, J. Dong, X. Mao, F. Long, C. Wang, D. Zhou, M. Gao, and G. Sun, “Pipezk: Accelerating zero-knowledge proof with a pipelined architecture,” in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 416–428.
- [22] “Fpga accelerator development and deployment in the cloud,” <https://aws.amazon.com/ec2/instance-types/t1>.
- [23] D. J. Bernstein, “Multidigit multiplication for mathematician,” <http://cr.yp.to/papers/m3.pdf>.
- [24] W. Bosma and H. Lenstra, “Complete systems of two addition laws for elliptic curves,” *Journal of Number Theory*, vol. 53, no. 2, pp. 229–240, 1995. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0022314X85710888>
- [25] D. J. Bernstein and T. Lange. [Online]. Available: <http://www.hyperelliptic.org/EFD/>
- [26] J. Renes, C. Costello, and L. Batina, “Complete addition formulas for prime order elliptic curves,” Cryptology ePrint Archive, Paper 2015/1060, 2015, <https://eprint.iacr.org/2015/1060>. [Online]. Available: <https://eprint.iacr.org/2015/1060>
- [27] Arkworks, “Arkworks library for zksnarks,” <https://docs.rs/ark-blis12-377>.
- [28] S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, and H. Wu, “Zexe: Enabling decentralized private computation,” Cryptology ePrint Archive, Paper 2018/962, 2018, <https://eprint.iacr.org/2018/962>. [Online]. Available: <https://eprint.iacr.org/2018/962>
- [29] P. S. L. M. Barreto, B. Lynn, and M. Scott, “Constructing elliptic curves with prescribed embedding degrees,” in *Security in Communication Networks*, S. Cimato, G. Persiano, and C. Galdi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 257–267.
- [30] D. J. Bernstein, J. Doumen, T. Lange, and J.-J. Oosterwijk, “Faster batch forgery identification,” Cryptology ePrint Archive, Report 2012/549, 2012, <https://ia.cr/2012/549>.
- [31] G. Gutoski, zK hack : Youtube. [Online]. Available: https://www.youtube.com/watch?v=Bl5mQA7UL2I&ab_channel=ZeroKnowledge

APPENDIX

OPTIMIZATIONS IN MODULAR MULTIPLICATION FOR U55C

Here, we focus on optimizations for the hardware of our choice — Xilinx U55C FPGA.

Recall that FPGAs contain DSP blocks that can multiply 18-bit and 27-bit numbers. Using the Karatsuba tree, a 377-bit by 377-bit multiplication can be done with 162 DSP blocks.

As mentioned in section III, each EC adder is composed of 12 modular multipliers which are in turn composed of 3 377-bit by 377-bit integer multipliers. Implementing this naively, we will need 5832 DSP blocks for one EC adder or 11664 for two. However, Our FPGA barely have enough DSP blocks even for one EC adder. Therefore, replacing some of the DSP blocks with LUTs is necessary for building a working pipelined EC adder.

As an example of multiplying by constant, consider $A = 10001110000$, then $A \cdot B$ is:

$$A \cdot B = B \ll 10 + B \ll 6 + B \ll 5 + B \ll 4$$

This can be improved using the Canonical Signed Digit (CSD) representation. The constant A can be represented as a combination of a positive part and a negative part: $A = 1001000000 - 10000$.

Now the computation contains only 3 terms:

$$A \cdot B = B \ll 10 + B \ll 7 - B \ll 5$$

The average fraction of non-zero bits in the CSD representation is one-third as compared to one-half in the binary

representation. The costs of addition and subtraction are the same LUT-wise and therefore we prefer the CSD representation.

All of the above leads to the conclusion that the DSP blocks that should be replaced are the ones performing multiplication by constants with low Hamming weight in the CSD representation. The basic LUT can implement any $5 \rightarrow 2$ logical function. Therefore addition/subtraction combinations of 2 or 3 bits are implemented with a single LUT. When we have 4 or 5 bits (this can happen only when the computation includes at least 4 or 5 terms), 2 LUTs are required. This logic can be extended to operations with larger bit sizes. This implies that the cost in LUTs of a multiplication depends not only on the Hamming weight of the constant but also on the non-zero bit distribution. Here are some examples. If we take a variable B with a bit-size of 10 bits, and different constants A :

1. $A = 10000000100$ translates into $B \ll 10 + B \ll 2$. The cost is 2 LUTs because there are 2 terms with an overlap of 2. The LUTs are for computing $b_0 + b_8$ and $b_1 + b_9$ as seen in fig. 10 (The carry is taken care of by a Carry-8 module).

2. $A = 10001000000$ translates into $B \ll 10 + B \ll 6$. The cost is 6 LUTs because there are 2 terms with an overlap of 6 as shown in fig. 11.

$$\begin{aligned} &b_9b_8b_7b_6b_5b_4b_3b_2b_1b_0 \\ &+b_9b_8b_7b_6b_5b_4b_3b_2b_1b_0 \end{aligned}$$

Fig. 10. Addition with an overlap of 2.

$$\begin{aligned} &b_9b_8b_7b_6b_5b_4b_3b_2b_1b_0 \\ &+b_9b_8b_7b_6b_5b_4b_3b_2b_1b_0 \end{aligned}$$

Fig. 11. Addition with an overlap of 6.

The same Karatsuba tree that we use for 377-bit numbers can actually be used for numbers up to 416 bits. This gives us some flexibility in the choice of $\{k_{ij}\}$. After computing optimal values of $\{k_{ij}\}$ by solving the optimization problems that are defined in II-C, we obtain the results depicted in tables IV and V.

MSB multiplication by m	# of DSP	# of LUT
No optimizations	162	9.5K
No optimizations, DSP replaced by LUTs	35	19.5K
Optimized k_{ij} , no MSB optimization	35	16K
Both optimizations	35	13.5K

TABLE IV
THE RESULTS OF OPTIMIZATIONS IN FIG. 3 AND IN II-C

After implementing all optimizations, we have that $\Delta^{\{k_{ij}\}} \approx 1.994 \cdot 2^{377}$. Refining error bounds using specific

LSB multiplication by q	# of DSP	# of LUT
No optimizations	162	9.5K
No optimizations, DSP replaced by LUTs	35	19K
Optimized k_{ij} , no LSB optimization	35	15.5K
Both optimizations	35	10K

TABLE V
THE RESULTS OF OPTIMIZATIONS IN FIG. 2 AND IN II-C

values of m and q , the maximal error $e(\hat{l}_1)$ from equation (22) can be shown to be 4. This means that the only modification that needs to be applied to the algorithm in fig. 1 — potentially subtracting up to $4q$ (and not $3q$) in the last step.

Goldilocks NTT

Tomer Solberg
Ingonyama



Goldilocks NTT Trick

Tomer Solberg
tomer@ingonyama.com

Abstract

Polynomial arithmetic is at the heart of modern Zero Knowledge Proving (ZKP) systems. The Number Theoretic Transform (NTT) is a crucial tool in facilitating efficient computational complexity over large polynomials encountered in ZKPs. NTTs are dominated by the number of field multiplications.

In this short note we focus on the specific case of NTT in the 64 bit Goldilocks field. Following the observation that 2^{48} is a root of unity, expressing the first few roots of unity in terms of powers of 2, allows modular multiplications to be replaced with simple bit-shift operations, resulting in a significant cost saving for NTT.

Update: This trick is well known in the FHE (Fully Homomorphic Encryption community) [1, 2, 3, 4]

1 Introduction

NTT is a core math primitive found in many of today’s deployed ZKPs. For example, in STARK arithmetization, the trace produced by the prover is interpolated into a suitable univariate polynomial form for building a polynomial Interactive Oracle Proof (IOP). The transformation between the coefficients form and evaluation form of the polynomial is done efficiently via an NTT.

In any ZKP, as the size of the vector becomes larger $n > 2^{20}$ the quasi-linear behavior of the NTT is expected to dominate [5]. Noting this potential problem as the sizes of circuits grow, the ZKP industry has taken various directions to solve this computational bottleneck.

One direction is to avoid NTTs entirely: for instance, the approach of Hyperplonk [6], which proposes multivariate interpolations on a Boolean hypercube instead of univariate interpolations on roots of unity. While this direction is interesting, it replaces the NTT with another computation that dominates its sumcheck protocol [7] involving MultiLinear Extension (MLE).

An alternative direction involves special primes which possess unique algebraic properties. One such example is the Goldilocks prime introduced by Mike Hamburg [8] based on the prime modulus:

$$q = \phi^2 - \phi - 1 \tag{1}$$

This prime was originally introduced in the context of elliptic curve Ed446-Goldilocks with $\phi = 2^{224}$, since $224 = 32 \cdot 7 = 28 \cdot 8 = 56 \cdot 4$ and thus it supports fast arithmetic in radix 2^{28} or radix 2^{32} suited for 32 bit machines and radix 2^{56} suited for 64 bit machines. The main advantage of the Goldilocks prime is an ultra-fast Karatsuba multiplication. If we

represent field elements as $a + b\phi$ for a, b of size 32 bit, we get:

$$\begin{aligned}(a + b\phi) \cdot (c + d\phi) &= ac + (ad + bc)\phi + bd\phi^2 \\ &= (ac + bd) + ((a + b)(c + d) - ac)\phi \mod q\end{aligned}\quad (2)$$

The final step of the multiplication is taking the modulo q reduction of the result. This is also extremely computationally cheap in the context of Goldilocks primes, given that $\phi = 2^k$ is some power of 2. Note the following

$$\begin{aligned}2^{2k} &\equiv 2^k - 1 \mod q \\ 2^{3k} &\equiv -1 \mod q\end{aligned}\quad (3)$$

where the second property follows from the factorization

$$(2^{3k} + 1) = (2^k + 1)(2^{2k} - 2^k + 1) \equiv 0 \mod q. \quad (4)$$

Then, if a number x which is represented by $4k$ bits (say, a product of two $2k$ bit numbers from the field), we can easily reduce it modulo q , by splitting it into the $2k$ least significant bits x_{LSB} , the k intermediary significant bits x_{ISB} , and the k most significant bits x_{MSB} , and reduce it in the following manner

$$\begin{aligned}x &\equiv x_{LSB} + 2^{2k}x_{ISB} + 2^{3k}x_{MSB} \mod q \\ &\equiv x_{LSB} + (2^k - 1)x_{ISB} - x_{MSB} \mod q\end{aligned}\quad (5)$$

The final step is to add or subtract q if an underflow or overflow occurs. Therefore, the complexity of the modular multiplication in the Goldilocks case is equivalent to the standard Karatsuba integer multiplication, plus a small number of extra additions.

For 64 bits, the Goldilocks prime [9]

$$p = 2^{64} - 2^{32} + 1 \quad (6)$$

has seen extensive usage in the ZKP industry already. Several existing ZKP systems such as Polygon ZKEVM [10], Polygon Miden VM [11], and Plonky2 [12] have STARK-based prover components that exploit efficient computations modulo p .

ethSTARK [13] uses a slightly different prime $p' = 2^{61} + 20 \cdot 2^{32} + 1$ and the NTT dominates about 80% of the prover time. If this prime were replaced with the Goldilocks prime, the net performance of ethSTARK would see a significant performance boost, due to size and efficient modular reduction with the Goldilocks field. While much of the tricks discussed above are known in the industry already, in this article we discuss another less known fact (in the ZK community) which makes higher radix NTTs in the Goldilocks field faster than ever before.

Update: We were informed by Michiel Van Beirendonck that @CosicBe used this trick to get the 3rd place in the Zprize event. The repository is not open sourced yet.

For the rest of this note and w.l.o.g, let us focus on the 64-bit Goldilocks field mentioned above. Namely, \mathbb{F}_p with $p = 2^{64} - 2^{32} + 1$.

2 Goldilocks Imaginary Unit and Its Roots

A useful property of \mathbb{F}_p is that the imaginary unit $j = \sqrt{-1}$ is a power of 2. This is true for any field \mathbb{F}_q with $q = 2^{2k} - 2^k + 1$ where k is even, where

$$2^{3k} \equiv -1 \pmod{q} \quad (7)$$

implies $j = \sqrt{-1} = 2^{3k/2}$, and specifically for \mathbb{F}_p we get $j = 2^{48}$. If $k/2$ can be divided further by factors of 2, other Roots of Unity (RoU) can also be expressed in powers of 2. Defining the N -th root of unity of the Goldilocks field \mathbb{F}_p as ω_N , we can write

$$\begin{aligned} \omega_N^{N/4} &= j = 2^{48} \\ \omega_N^{N/8} &= 2^{24} \\ \omega_N^{N/16} &= 2^{12} \\ &\vdots \\ \omega_N^{N/64} &= 2^3. \end{aligned} \quad (8)$$

Expressing RoUs in terms of powers of two is useful in the context of NTT's, as multiplications can simply be replaced with bit shifts of the samples, followed by reduction modulo p , which in the Goldilocks case as we have seen is a matter of simple additions.

3 Efficient NTT with Goldilocks' Imaginary Unit

NTT computation is generally performed using the Cooley-Tukey (CT) algorithm, which is a recursive algorithm that typically uses basic building blocks referred to as radix- 2^k blocks. Sticking to NTT lengths that are powers of 2 for simplicity, say $N = 2^\ell$, allows using recursive stages of radix- 2^k blocks, assuming k divides ℓ . In this case, the number of stages is k/ℓ and each stage contains $2^{k-\ell}$ such radix- 2^k blocks.

Without getting into exact details, the calculation of an NTT of size N requires multiplications by powers of the N -th RoU and additions. In the CT algorithm, this translates to using powers of the k -th RoU inside the radix- 2^k blocks and powers of the N -th RoU outside the blocks. The multiplications outside the blocks are also known as multiplications by twiddle factors. The total number of multiplications required per each radix- 2^k block with its associated twiddles is $k2^{k-1}$ of which $2^k - 1$ are the twiddle factors. Roughly speaking, the above can be used to calculate the ratio of twiddle factors for the overall length N NTT.

For example, a radix-4 block contains internally a single multiplication by $\omega_N^{N/4} = j$. This is a useful fact for computing FFTs, as multiplication by j of a complex number is computationally trivial. Similarly, using a radix-4 architecture for computing NTT over the Goldilocks field can save 1/4 of the multiplications, by replacing the internal multiplication with a shift and addition, as shown previously. This situation improves further by considering higher radices. Radix-8 blocks contain a total of 12 multiplications, of which 5 are internal multiplications by powers of $\omega_N^{N/8} = 2^{24}$. Replacing these reduces the number of multiplications by a factor of 7/12. See Diagram 1 for a visualization of a radix-8 NTT block. This can be further utilized all the way up to radix-64 as is shown in Table 1.

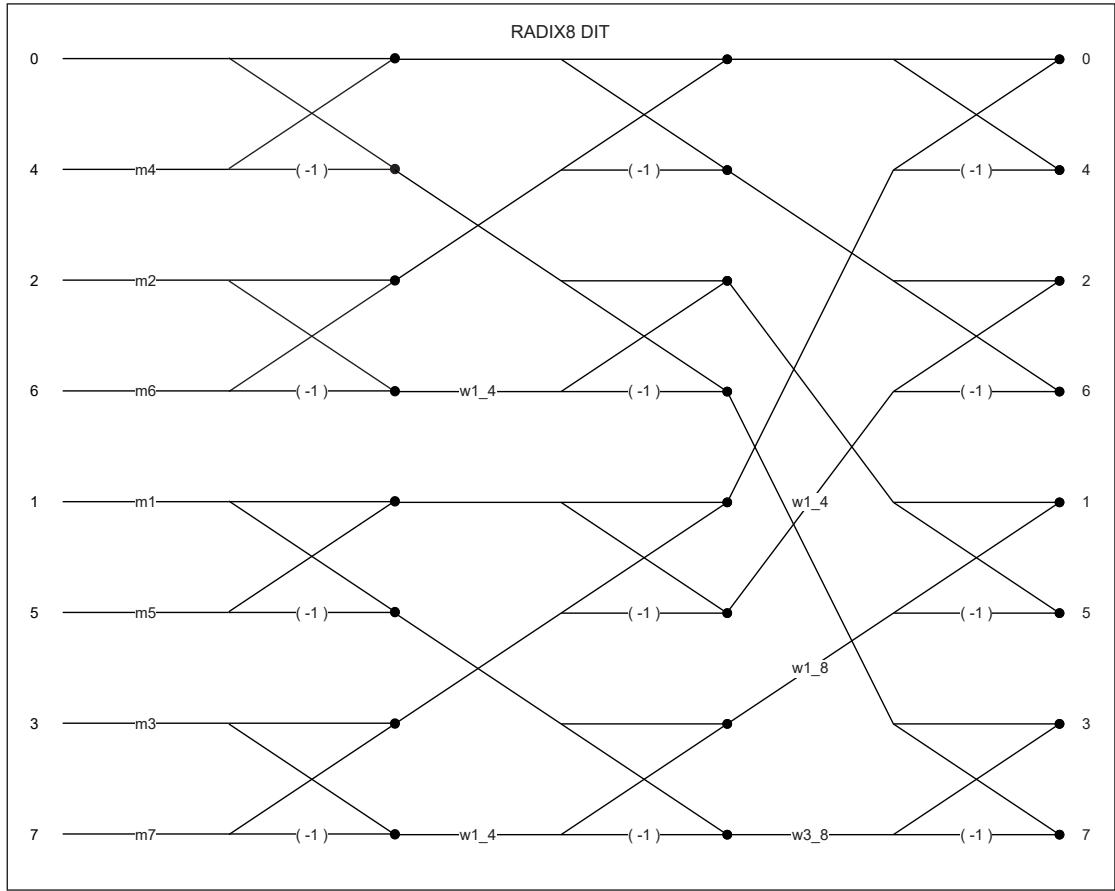


Figure 1: Radix-8 NTT Block

4 Acknowledgements

The author would like to thank Yuval Domb, Karthik Inbasekar and Omer Shlomovits for useful discussions. We would also like to thank Michiel Van Beirendonck for updating us regarding the usage of the NTT trick in Zprize.

References

- [1] Wei Wang and Xinming Huang. Fpga implementation of a large-number multiplier for fully homomorphic encryption. In *2013 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2589–2592, 2013.
- [2] nuFHE. Polynomial multiplication. https://nufhe.readthedocs.io/en/latest/implementation_details.html.
- [3] Niall Emmart and Charles Weems. High precision integer multiplication with a gpu. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 1781–1787, 2011.

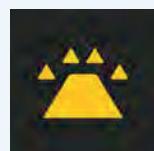
Radix	Twiddle factors	Internal multiplications	Improvement factor
2	1	0	1
4	3	1	3/4
8	7	5	7/12
16	15	17	15/32
32	31	49	31/80
64	63	129	21/64

Table 1: Multiplication reduction factor of different NTT architectures

- [4] nuFHE. nucypher. <https://github.com/nucypher/nufhe>.
- [5] Dan Boneh. Zkp workshop 2022. https://youtu.be/6psLQv5Hf_I.
- [6] Binayi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. Hyperplonk: Plonk with linear-time prover and high-degree custom gates. Cryptology ePrint Archive, Paper 2022/1355, 2022. <https://eprint.iacr.org/2022/1355>.
- [7] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. *J. ACM*, 39(4):859–868, oct 1992.
- [8] Mike Hamburg. Ed448-goldilocks, a new elliptic curve. Cryptology ePrint Archive, Paper 2015/625, 2015. <https://eprint.iacr.org/2015/625>.
- [9] Thomas Pornin. Ecgfp5 : a sepcialized elliptic curve. <https://github.com/pornin/ecgfp5/blob/main/doc/ecgfp5.pdf>.
- [10] Polygon-Hermez. Polygon zkvm. <https://github.com/0xpolygonhermez>.
- [11] Polygon-Miden VM. polygon miden. <https://wiki.polygon.technology/docs/miden/intro/overview/>.
- [12] Polygon-Zero. Plonky2. <https://github.com/mir-protocol/plonky2>.
- [13] StarkWare. ethstark documentation. Cryptology ePrint Archive, Paper 2021/582, 2021. <https://eprint.iacr.org/2021/582>.

NTT Mini: Exploring Winograd's Heuristic for Faster NTT

Carol Danvers
Ingonyama



NTT Mini: Exploring Winograd’s Heuristic for Faster NTT

Carol Danvers
carol@ingonyama.com

Abstract

We report on the Winograd-based implementation for the Number Theoretical Transform. It uses less multiplications than the better-known Cooley-Tuckey alternative. This optimization is important for very high order finite-fields. Unfortunately, the Winograd scheme is difficult to generalize for arbitrary sizes and is only known for small-size transforms. We open-source our hardware implementation for size 32 based on [1].

1 Motivation

Zero Knowledge Proofs (ZKP) rely on a small number of computationally intensive primitives such as Multi Scalar Multiplication (MSM) and Number Theoretic Transform (NTT). The acceleration of these primitives is a necessary enabler for global adoption of these technologies. In [2], we discussed MSM. The focus of this note is NTT.

NTT is a generalization of the Discrete Fourier Transform (DFT) for finite-fields. Being a linear transform, it can be written in a matrix form

$$\vec{y} = \mathbf{F} \vec{x} \tag{1}$$

The transform matrix \mathbf{F} has a special form:

$$\mathbf{F} = \begin{pmatrix} 1 & 1 & 1 & \dots \\ 1 & \omega_N & \omega_N^2 & \dots \\ 1 & \omega_N^2 & \omega_N^4 & \dots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} \tag{2}$$

where ω_N is the root-of-unity of order N and N is the transform length.

Multiplication is computationally expensive, so our goal is to minimize the number of multiplications, (this comes at the expense of more additions). For an arbitrary full-rank matrix of size N , computing (1) costs N^2 multiplications. The special form of the NTT matrix \mathbf{F} allows factorization to a product of sparse matrices where many of the non-zero elements are ± 1 .

The Cooley-Tuckey (CT) factorization can be applied recursively for any N depending on its prime factorization. For N that is a power of a small prime, CT achieves a transform cost of $N \log N$ multiplications. Of particular interest is N that is a power of 2.

The Winograd factorization, discussed here, is a more efficient factorization, reducing \mathbf{F} to a product of sparse matrices with only ± 1 's, and a single diagonal matrix with non-trivial values. The rank r of the diagonal matrix is always $N \leq r < N \log N$. For Winograd r is actually the number of required multiplications. By definition, the Winograd factorization is not limited by N , though it is not recursive and only known for some small values of N . Below is a table comparing CT to Winograd for $N = 16, 32$.

Size	CT	Winograd
16	17	13
32	49	40

Table 1: Number of multiplications for different transform sizes

2 Theoretical Background

Winograd, much like CT, can be presented both as a series of recursive steps and as a factorization of the DFT matrix. The symmetries in the DFT matrix allow elegant factorization.

2.1 Notation

Define the following notations used hereafter:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \quad (3)$$

$$A \otimes B = \begin{pmatrix} a_{11}B & a_{12}B \\ a_{21}B & a_{22}B \end{pmatrix} \quad (4)$$

$$A \oplus B = \begin{pmatrix} A & 0 \\ 0 & B \end{pmatrix} \quad (5)$$

Additionally, let us denote by \mathbf{M}_n the matrix of dimension 2^n .

2.2 Cooley-Tuckey Factorization

We follow the presentation of the factorization from [3].

Theorem 2.1 (Cooley-Tuckey Factorization) *The $2^n \times 2^n$ DFT matrix F_n can be factored as:*

$$\mathbf{F}_n = \mathbf{P}_n \mathbf{A}_n^{(0)} \dots \mathbf{A}_n^{(n-1)} \quad (6)$$

where, for $k = 0, \dots, n - 1$:

$$\mathbf{A}_n^{(k)} = \mathbf{I}_{n-k-1} \otimes \mathbf{B}_{k+1} \quad (7)$$

$$\mathbf{B} = (\mathbf{I}_k \oplus \boldsymbol{\Omega}_k)(H \otimes \mathbf{I}_k) \quad (8)$$

$$\boldsymbol{\Omega}_k = \text{diag}(w_{2^{k+1}}^0, \dots, w_{2^{k+1}}^{2^k-1}) \quad (9)$$

and \mathbf{P}_n is a bit reversal permutation that satisfies:

$$\mathbf{P}_n(v_1 \otimes \dots \otimes v_n) = v_n \otimes \dots \otimes v_1 \quad (10)$$

Notice that $\mathbf{A}_n^{(k)}$ is a sparse matrix, containing only 2 non-zero entries in each row. Thus, multiplying a vector by this matrix can be done in $O(2^n)$ time, which is linear in the dimension 2^n . The number of $\mathbf{A}_n^{(k)}$ matrices in the factorization is n , which is logarithmic in the dimension 2^n . Thus, this leads to a total of $O(n \cdot 2^n)$ time (or $N \log N$ for $N = 2^n$).

2.2.1 Winograd Factorization

Winograd utilizes different symmetries in the DFT matrix.

Theorem 2.2 (Winograd's Heuristic Factorization) *The $2^n \times 2^n$ DFT matrix F_n can be factored as:*

$$\mathbf{F}_n = (H_2 \cdot \mathbf{I}_{n-1})(\mathbf{F}_n \oplus \mathbf{Q}_{n-1} \mathbf{P}_n^{\pi_n}) \quad (11)$$

where:

$$\mathbf{P}_n^{\pi_n} = \begin{pmatrix} \mathbf{I}_{n-1} \otimes \boldsymbol{\Psi}_{2 \otimes 4} \\ \mathbf{I}_{n-1} \otimes \boldsymbol{\Psi}_{2 \otimes 4} \mathbf{I}_{n-1}^{1 \rightarrow} \end{pmatrix} \quad (12)$$

$$\boldsymbol{\Psi}_{2 \otimes 4} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \quad (13)$$

\mathbf{Q}_{n-1} is a prefix matrix, and $\mathbf{I}_{n-1}^{1 \rightarrow}$ is the matrix obtained from the identity matrix by shifting its columns by one position to the right.

Intuitively, the main goal of this factorization is to decompose the DFT matrix as follows:

$$\mathbf{F}_n = \mathbf{M}_1 \cdot \dots \cdot \mathbf{M}_k \cdot \mathbf{D} \cdot \mathbf{M}_{k+1} \dots \mathbf{M}_n \quad (14)$$

where \mathbf{M}_i is a sparse matrix consisting of ± 1 entries, and \mathbf{D} is a diagonal matrix (typically of dimension greater than 2^n).

The strategy of [1] is to gradually decompose the matrices \mathbf{F}_n , \mathbf{Q}_n by utilizing the above theorem, as well as several permutations and the following decomposition rules that capitalize on inherent symmetries in each of these matrices.

Claim 2.3 For $n \times n$ matrices A, B, C , the following holds.

$$\begin{pmatrix} A & B \\ A & -B \end{pmatrix} = (H_2 \otimes I_n)(A \oplus B) \quad (15)$$

$$\begin{pmatrix} A & A \\ B & -B \end{pmatrix} = (A \oplus B)(H_2 \otimes I_n) \quad (16)$$

$$\begin{pmatrix} A & B \\ B & A \end{pmatrix} = \frac{1}{2}(H_2 \otimes I_n)((A + B) \oplus (A - B))(H_2 \otimes I_n) \quad (17)$$

$$\begin{pmatrix} A & B \\ B & -A \end{pmatrix} = \frac{1}{2}(T \otimes I_n) \begin{pmatrix} A - B & 0 & 0 \\ 0 & -(A + B) & 0 \\ 0 & 0 & B \end{pmatrix} (T \otimes I_n) \quad (18)$$

$$\begin{pmatrix} A & B \\ C & A \end{pmatrix} = (Q \otimes H \otimes I_{n-1}) \begin{pmatrix} C - A & 0 & 0 \\ 0 & B - A & 0 \\ 0 & 0 & A \end{pmatrix} (T \otimes H \otimes I_{n-1}) \quad (19)$$

where

$$T = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix} \quad (20)$$

$$Q = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix} \quad (21)$$

Using these rules, the authors of [1] managed to decrease the dimension of the factorization by 2. The caveat of this scheme is the non-uniformity of the factorization of \mathbf{Q}_n requiring a well-designed permutation to exploit its symmetries.

3 Results

This note summarizes our initial experience with Winograd factorizations of size $N = 2^n$, based on the derivations of [1]. We used Symbolic Algebra System (SAS) tools to automate the ad-hoc factorizations for arbitrary finite-fields. We outline our workflow as follows:

1. SAS code generates C++ template. The template captures the structure of size 2^n transform only, and does not depend on specific finite field selection. It implements
 - (a) sparse matrix multiplication (14), and
 - (b) diagonal matrix \mathbf{D} computation
2. C++ template is instantiated for a specific finite field as C++ code
3. C++ code is compatible with High Level Synthesis EDA tools, that eventually produce RTL (Verilog, VHDL), and, finally, FPGA bitstreams or GL1 for ASICs.

We open-source a C++ template for NTT of size $2^5 = 32$ together with a C++ instance for the scalar finite field of the Elliptic Curve *bn254*. Template ntt32_winograd uses arbitrary precision unsigned integers to represent the elements of the finite field. Specifically, we use type template ap_uint<W> from Xilinx' Vitis HLS toolchain [4]. The template takes the vector $\vec{x} = (x_0, \dots, x_{31})$ as input and returns the output $\vec{y} = (y_0, \dots, y_{31})$ by reference:

```
template<int W> void ntt32_winograd(
    const ap_uint<W> x0, ... , const ap_uint<W> x31 ,
    ap_uint<W>* y0, ... , ap_uint<W>* y31
) { ... }
```

All finite field matrix operations are unrolled and call scalar functions basic_add_mod(), basic_sub_mod() and mult_red(), which are specific for the selected finite field.

The template is instantiated in function ntt32_winograd_bn254_scalar

```
void ntt32_winograd_bn254_scalar(
    const ap_uint<254> x0, ... , const ap_uint<254> x31 ,
    ap_uint<254>* y0, ... , ap_uint<254>* y31 ,
)
{
    ntt32_winograd(x0, ... , x31 , y0, ... , y31 );
    return;
}
```

The header file ntt32_winograd_bn254_scalar.hpp declares the above instance and optimized finite field operations for scalar bn254 field.

When using our example, this header is the only file to include.

```
#include "ntt32_winograd_bn254_scalar.hpp"
...
// define input vector x
...
ntt32_winograd_bn254_scalar(x0, ... , x31 , &y0, ... ,& y31 );
// use output vector y
...
```

4 Usage

1. Make sure you have g++ toolchain installed.
2. Make sure you have Xilinx Vitis installed. Environment variable XILINX_HLS should be defined and point to the distribution. This will allow the toolchain to find Xilinx's arbitrary precision headers.
3. Make sure you have C++ Boost library [5]. Environment variable BOOST should point to the installation. We need this library for tests only.

4. Download our code from here [6]
5. Run make test

5 Future Directions

In this note, we demonstrated the Winograd factorization only for small degree polynomials. Real-world instances of Zero Knowledge Proofs and Fully Homomorphic Encryption require higher degree polynomial arithmetic, with common sizes of N reaching 2^{15} and often higher. There are various interesting directions to proceed. One is to extend the work of [1], finding the Winograd factorization for specific power-of-two N 's larger than 32, potentially discovering a closed-form extendable expression. A second, more immediate, is to utilize the recursive structure of NTT, together with the small-size Winograd building-blocks, to extend the construction to higher N 's similarly to what was done by CT.

Our HLS code correctness was verified in C++ and Verilog simulations. We will soon integrate it as part of our Cloud-ZK dev-kit [7], enabling developers to integrate with a fast NTT implementation running on AWS F1 FPGA instances.

We hope that our implementation will lead to a better intuition on the complexity of Winograd. The number of multiplications, dominating the computation time is behaving as $\mathcal{O}(N)$. We think it will be interesting to compare the theoretical complexity to concrete measurements. In general, Winograd takes more additions than CT, which might become a non-negligible factor in total running time.

References

- [1] Mateusz Raciborski and Aleksandr Cariow. On the derivation of winograd-type dft algorithms for input sequences whose length is a power of two. *Electronics*, 11:1342, 04 2022.
- [2] Charles F Xavier. Pipemsm: Hardware acceleration for multi-scalar multiplication. *Cryptology ePrint Archive*, 2022.
- [3] Daan Camps, Roel Van Beeumen, and Chao Yang. Quantum fourier transform revisited. *Numerical Linear Algebra with Applications*, 28(1), sep 2020.
- [4] Vitis high-level synthesis user guide (ug1399). <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Overview-of-Arbitrary-Precision-Integer-Data-Types>.
- [5] C++ boost library. <https://www.boost.org/>.
- [6] Winograd ntt32 code. <https://github.com/ingonyama-zk/ntt-winograd.git>.
- [7] Cloud zk. <https://github.com/ingonyama-zk/cloud-ZK>.

The RISC Zero Proof System: A Step-By-Step Description

Frank Laub and Paul Gafni
RISC Zero



The RISC Zero Proof System: A Step-By-Step Description

What is a Computational Receipt?

When a method executes inside the RISC Zero zkVM, the zkVM produces a **computational receipt** along with the output.

The receipt serves as a cryptographic authentication that the given method was executed faithfully. In particular, the receipt includes a **MethodID**, which serves as an identifier for a particular computational method and a **seal** which indicates that the associated execution trace satisfies the rules of the RISC-V ISA.

By linking the MethodID to the asserted output of the computation, computational receipts offer a powerful new model for trust in software. The option to check a computational receipt opens the doors to the [era of verifiable computing](#), where we use mathematics and science to bring trustable software into trustless environments.

For a more technical description of RISC Zero's receipts, see the [Proof System Sequence Diagram](#), the [Seal Construction Explainer](#) and the [STARKs reference page](#).

What is an Execution Trace?

When a piece of code runs on a machine, the **execution trace** is a complete record of the computation: a snapshot of the full state of the machine at each clock cycle of the computation.

It's typical to write an execution trace as a rectangular array, where each row shows the complete state of the machine at a given moment in time, and each column shows a temporal record of some particular aspect of the computation (say, the value stored in a particular RISC-V register) at each clock cycle. A line-by-line analysis of the trace allows for a computational audit with respect to the program instructions and the underlying computer architecture.

RISC Zero's computational receipts use cutting-edge technology to audit an execution trace while preserving computational privacy.

For more technical description of the process of creating a computational receipt, see the [proof system sequence diagram](#), the [seal construction explainer](#) and the [STARKs reference page](#).

Constructing a Seal

The seal is the part of the receipt that allows third-parties to authenticate the validity of the journal. It's the zero-knowledge proof that sits at the crux of our technology, showing that the journal was faithfully constructed (according to the program defined by the methodID).

The construction of a seal is highly technical, relying on several recent advances in the world of zero-knowledge cryptography. In this series of 10 brief lessons, we walk through the construction of a RISC Zero seal with as little technical jargon as possible.

If you make sense of these 10 lessons, you'll have a solid handle on the mechanics of a zk-STARK (and we'd likely love to hire you).

The proof system sequence diagram describes this process in more generality; we suggest going back and forth between this document and the sequence diagram.

Lesson 1: The Execution Trace

When any code executes in the RISC Zero virtual machine, each step of that execution is recorded in an Execution Trace.

We show a simplified example, computing 4 steps of a Fibonacci sequence modulo 97, using two user-specified inputs.

Input 1	24				Asserted Output	28			
Input 2	30								
Modulo									
Execution Trace - Initialization	Clock Cycle	Data Column 1	Data Column 2	Data Column 3	Control Column - Initialization	Control Column - Transition	Control Column - Termination		
Execution Trace - Transition	0	24	30	54	1	0	0	0	0
Execution Trace - Transition	1	30	54	84	0	1	0	1	0
Execution Trace - Transition	2	54	84	41	0	1	0	1	0
Execution Trace - Termination	3	84	41	28	0	1	0	1	1

In this example, our trace consists of 6 [columns](#).

- Each of the first three columns is a record of the internal state of a register at each clock cycle from initialization until termination. We call these [Data Columns](#).
- The next three columns are [control columns](#), which we use to mark initialization and termination points.

In the full RISC Zero protocol, the Data Columns hold the state of the RISC-V processor, including ISA registers, the program counter, and various microarchitecture details such as instruction decoding data, ALU registers, etc., while the Control Columns handle system initialization and shutdown, the initial program code to load into memory before execution, and other control signals that don't depend on the programs execution.

The full implementation also has [PLONK Columns](#), allowing for RISC-V memory emulation. The [PLONK Columns](#) are not necessary in this simplified example.

Lesson 2: Rule checks to validate a computation

Here, we introduce a number of rule-checking cells in order to demonstrate the validity of the execution. In this example, we show six rules specific to generating fibonacci numbers.

Input 1	24				Asserted Output	28							
Input 2	30												
Modulo													
Execution Trace - Initialization	Clock Cycle	Data Column 1	Data Column 2	Data Column 3	Control Column - Initialization	Control Column - Transition	Control Column - Termination	Does Fibonacci relation hold? 1?	Does Initialization for Data Column 1 match User Input Input 2?	Does Initialization for Data Column 2 match User Input Input 2?	Does Termination value for Data Column 3 match Output?	Does entry i from Input Column 1 match entry i-1 from Input Column 2?	Does entry i from Input Column 2 match entry i-1 from Output Column?
Execution Trace - Transition	0	24	30	54	1	0	0	0	0	0	0	0	0
Execution Trace - Transition	1	30	54	84	0	1	0	0	0	0	0	0	0
Execution Trace - Transition	2	54	84	41	0	1	0	0	0	0	0	0	0
Execution Trace - Termination	3	84	41	28	0	1	1	0	0	0	0	0	0

Each rule check is written as the product of two terms, modulo 97. The first term equals zero when the rule holds. The second term equals zero when we don't want to enforce the rule.

Each rule checking column can be expressed as a multi-input, single-output polynomial, where the inputs are some combination of entries in the trace; we call these [rule-checking polynomials](#).

In the full RISC-V implementation, the rules make up what it means to execute RISC-V instructions properly (i.e. checking that the program counter increments after every instruction). We check thousands of rules in order to validate the execution trace.

Lesson 3: Padding the Trace

Before encoding each column as a polynomial, we append random padding to the end of the Execution Trace, which allows for a zero-knowledge protocol. This random noise is generated by the host system's cryptographically secure pseudorandom number

generator. We set the Control columns to 0 for these random noise rows, in order to turn off our rule checks.

	Clock Cycle	Data Column 1	Data Column 2	Data Column 3	Control Column - Initialization	Control Column - Transition	Control Column - Termination	Does Fibonacci relation hold?	Does Initialization for Data Column 1 match User Input 1?	Does Initialization for Data Column 2 match User Input 2?	Does Termination value for Data Column 3 match User Output?	Does entry i from Input Column 1 match entry i-1 from Input Column 2?	Does entry i from Input Column 2 match entry i-1 from Output Column?
Execution Trace - Initialization	0	24	30	54	1	0	0	0	0	0	0	0	0
Execution Trace - Transition	1	30	54	84	0	1	0	0	0	0	0	0	0
Execution Trace - Transition	2	54	84	41	0	1	0	0	0	0	0	0	0
Execution Trace - Termination	3	84	41	28	0	1	1	0	0	0	0	0	0
Random Padding	4	78	30	29	0	0	0	0	0	0	0	0	0
Random Padding	5	15	82	51	0	0	0	0	0	0	0	0	0
Random Padding	6	29	52	25	0	0	0	0	0	0	0	0	0
Random Padding	7	50	62	8	0	0	0	0	0	0	0	0	0

Lesson 4: Constructing Trace Polynomials

Let's remove the `rule-checking columns` for a minute and turn our attention toward encoding our Trace data in terms of polynomials. Throughout these lessons, all of the arithmetic takes place in \mathbb{F}_{97} .

From here on, the lessons assume some familiarity with finite fields. If finite fields are foreign to you, fear not! This finite fields primer covers just enough to make sense of RISC Zero's use of finite fields.

Every element of \mathbb{F}_{97} can be written as a power of 5. In other words, the elements of \mathbb{F}_{97} are $0, 5^0, 5^1, \dots, \text{and } 5^{95}$. We write $\mathcal{D}(5^{12})$ for the set of powers of 5^{12} and $\mathcal{D}(5^3)$ for the set of powers of 5^3 . Each of these sets is "evenly spaced" throughout \mathbb{F}_{97} , which facilitates the use of number theoretic transforms.

Put succinctly, running an iNTT on a trace column gives the coefficients of a trace polynomial.

Column	Python Code (from sympy import intt)	Code Output (Coefficients of Trace Polynomials)							
d1	<code>d1 = intt([24, 30, 54, 84, 78, 15, 29, 50], prime=97)</code>	94	68	41	69	25	72	85	55
d2	<code>d2 = intt([30, 54, 84, 41, 2, 77, 21, 36], prime=97)</code>	31	31	0	87	76	66	6	24
d3	<code>d3 = intt([54, 84, 41, 28, 71, 17, 92, 33], prime=97)</code>	4	14	83	44	12	44	12	35
c1	<code>c1 = intt([1, 0, 0, 0, 0, 0, 0], prime=97)</code>	85	85	85	85	85	85	85	85
c2	<code>c2 = intt([0, 1, 1, 0, 0, 0, 0], prime=97)</code>	61	80	12	37	12	60	12	17
c3	<code>c3 = intt([0, 0, 1, 0, 0, 0, 0], prime=97)</code>	85	89	27	18	12	8	70	79

The table above shows the python code and associated input/output arrays; the input arrays correspond to the `trace columns` and the output arrays are the coefficients of the `trace polynomials`. The key features about the `trace polynomials` are that

- they are at most degree 7 and
- the `trace polynomial` evaluations on $\mathcal{D}(5^{12})$ precisely match the data in the padded execution trace.

Trace Polynomials
$d1(x)=94+68x+41x^2+69x^3+25x^4+72x^5+85x^6+55x^7$
$d2(x)=31+31x+0x^2+87x^3+76x^4+66x^5+6x^6+24x^7$
$d3(x)=4+14x+83x^2+44x^3+12x^4+44x^5+12x^6+35x^7$
$c1(x)=85+85x+85x^2+85x^3+85x^4+85x^5+85x^6+85x^7$
$c2(x)=61+80x+12x^2+37x^3+12x^4+60x^5+12x^6+17x^7$
$c3(x)=85+89x+27x^2+18x^3+12x^4+8x^5+70x^6+79x^7$

Evaluating the trace polynomial on the larger set $\mathcal{D}(5^3)$ gives a Reed-Solomon encoded trace block.

We say the `block` is a `degree 4 expansion` of the `column`, and that the Reed-Solomon encoding has `rate` $\frac{1}{4}$. Reed-Solomon encoding improves the soundness of the protocol by amplifying any errors in an invalid trace.

<code>Input 1</code>	24							<code>Asserted Output</code>	28
<code>Input 2</code>	30								
<code>Modulo</code>	97								
		<code>Reed Solomon Input</code> (Exponent Notation in terms of 28)	<code>Reed Solomon Input</code> (Exponent Notation in terms of 5)	<code>Reed Solomon Input</code> (Simplified)	<code>Clock Cycle</code>	<code>Data Column 1</code>	<code>Data Column 2</code>	<code>Data Column 3</code>	<code>Control Column - Initialization</code>
Execution Trace - Initialization	$28^0 \bmod 97$	$5^0 \bmod 97$	1	0	24	30	54	1	0
RS Redundancy Row	$28^1 \bmod 97$	$5^1 \bmod 97$	28		27	33	43	23	35
RS Redundancy Row	$28^2 \bmod 97$	$5^2 \bmod 97$	8		74	14	27	45	31
RS Redundancy Row	$28^3 \bmod 97$	$5^3 \bmod 97$	30		77	31	88	53	63
Execution Trace - Transition			64	1	30	54	84	0	1
RS Redundancy Row			46		37	76	67	72	32
RS Redundancy Row			27		62	85	34	83	63
RS Redundancy Row			77		3	84	63	70	30
Execution Trace - Transition			22	2	54	84	41	0	1
RS Redundancy Row			34		42	14	11	10	58
RS Redundancy Row			79		96	18	86	60	59
RS Redundancy Row			78		69	3	29	25	20
Execution Trace - Termination			50	3	84	41	28	0	1
RS Redundancy Row			42		36	15	54	53	8
RS Redundancy Row			12		26	16	31	11	91
RS Redundancy Row			45		37	77	1	68	51
Random Padding			96	4	78	2	71	0	0
RS Redundancy Row			69		71	90	82	2	38
RS Redundancy Row			89		24	15	14	62	57
RS Redundancy Row			67		70	66	75	13	66
Random Padding			33	5	15	77	17	0	0
RS Redundancy Row			51		31	8	76	26	65
RS Redundancy Row			70		38	20	67	13	36
RS Redundancy Row			20		71	19	14	86	9
Random Padding			75	6	29	21	92	0	0
RS Redundancy Row			63		40	43	42	46	81
RS Redundancy Row			18		54	72	72	87	86
RS Redundancy Row			19		19	92	90	80	70
Random Padding			47	7	50	36	33	0	0
RS Redundancy Row			55		80	66	45	60	74
RS Redundancy Row			85		87	8	89	28	65
RS Redundancy Row	$28^{31} \bmod 97$		52		18	70	60	91	82
	$28^{32} \bmod 97$		1						

Lesson 5: ZK Commitments of the Trace Data

The next step is for the Prover to commit the `trace polynomials` into a Merkle tree.

In order to maintain a Zero-Knowledge protocol, the Prover evaluates each trace polynomial over a shift of $\mathcal{D}(5^3)$.

Specifically, we evaluate each `trace polynomial` over $x = 5, 5^4, 5^7, \dots, 5^{94}$.

	Reed Solomon Input (Exponent Notation in terms of 28)	Reed Solomon Input (Exponent Notation in terms of 5)		Reed Solomon Input (Simplified)	Clock Cycle	Data Column 1	Data Column 2	Data Column 3	Control Column - Initialization	Control Column - Transition	Control Column - Termination
Disguised Execution Trace	$5^{28}0 \bmod 97$	$5^11 \bmod 97$		5	0	31	39	12	82	81	2
Disguised RS Redundancy Row	$5^{28}1 \bmod 97$	$5^4 \bmod 97$		43		15	36	11	18	72	32
Disguised RS Redundancy Row	$5^{28}2 \bmod 97$	$5^7 \bmod 97$		40		96	65	6	32	73	65
Disguised RS Redundancy Row	$5^{28}3 \bmod 97$	$5^{10} \bmod 97$		53		79	41	88	68	10	22
Disguised Execution Trace				29	1	69	35	49	81	64	32
Disguised RS Redundancy Row				36		31	85	50	41	58	16
Disguised RS Redundancy Row				38		16	41	69	18	40	38
Disguised RS Redundancy Row				94		71	24	89	86	56	50
Disguised Execution Trace				13	2	35	77	46	92	16	47
Disguised RS Redundancy Row				73		10	40	9	59	83	24
Disguised RS Redundancy Row				7		53	54	58	14	20	67
Disguised RS Redundancy Row				2		28	7	95	44	92	35
Disguised Execution Trace				56	3	67	81	80	43	61	82
Disguised RS Redundancy Row				16		26	2	73	31	21	18
Disguised RS Redundancy Row				60		0	54	76	8	64	32
Disguised RS Redundancy Row				31		45	36	80	92	4	68
Disguised Random Padding				92	4	91	59	35	10	22	81
Disguised RS Redundancy Row				54		94	39	57	71	34	41
Disguised RS Redundancy Row				57		18	82	19	50	40	18
Disguised RS Redundancy Row				44		80	36	44	89	28	86
Disguised Random Padding				68	5	54	12	61	2	48	92
Disguised RS Redundancy Row				61		39	46	78	32	64	59
Disguised RS Redundancy Row				59		16	16	16	65	72	14
Disguised RS Redundancy Row				3		19	49	95	22	31	44
Disguised Random Padding				84	6	57	23	47	32	55	43
Disguised RS Redundancy Row				24		74	89	18	16	37	31
Disguised RS Redundancy Row				90		40	96	42	38	26	8
Disguised RS Redundancy Row				95		43	54	72	50	9	92
Disguised Random Padding				41	7	57	19	90	47	44	10
Disguised RS Redundancy Row				81		75	8	27	24	22	71
Disguised RS Redundancy Row				37		28	34	37	67	56	50
Disguised RS Redundancy Row	$5^{28}31 \bmod 97$			66		96	1	51	35	64	89
	$28^{32} \bmod 97$			5							

Note that because of the shift, the yellow and blue cells in Data Columns 1, 2, and 3 no longer match the Inputs and Asserted Outputs. In fact, this shift in the evaluation domain disguises all the Trace Data.

We only reveal information about the disguised trace, and the random padding we appended is sufficient to prevent an attacker from deducing any connection between the disguised trace and the actual trace.

Lesson 6: Constraint Polynomials

Now that we've encoded our trace columns into `trace polynomials`, let's return to our original Reed-Solomon domain and add back in our `rule-checking cells`.

	Reed Solomon Input (Exponent Notation in terms of 28)	Reed Solomon Input (Exponent Notation in terms of 5)	Read Solomon Input (Simplified)	Clock Cycle	Data Column 1	Data Column 2	Data Column 3	Control Column - Initialization	Control Column - Transition	Control Column - Termination	Does Fibonacci relation hold?	Does Initialization for Data Column 1 match User Input 1?	Does Initialization for Data Column 2 match User Input 2?	Does Initialization for Data Column 3 match User Input 3?	Does Termination value for Data Column 1 match User Input 1?	Does Termination value for Data Column 2 match User Input 2?	Does Termination value for Data Column 3 match User Input 3?	Does entry i from Input Column 1 match entry i-1 from Input Column 2?	Does entry i from Input Column 2 match entry i-1 from Input Column 3?	Does entry i from Input Column 3 match entry i-1 from Output Column?
Execution Trace - Initialization	28^0 mod 97	5^0 mod 97	1	0	24	30	54	1	0	0	0	0	0	0	0	0	0	0	0	0
RS Redundancy Row	28^1 mod 97	5^1 mod 97	28		27	33	43	23	35	26	27	69	69	2	30	0	0	0	0	0
RS Redundancy Row	28^2 mod 97	5^2 mod 97	8		74	14	27	45	31	13	3	19	19	94	0	3	0	0	0	0
RS Redundancy Row	28^3 mod 97	5^3 mod 97	30		77	31	88	53	63	86	24	80	53	19	52	16	0	0	0	0
Execution Trace - Transition			64	1	30	54	84	0	1	0	0	0	0	0	0	0	0	0	0	0
RS Redundancy Row			46		37	76	67	72	32	46	14	60	14	46	31	0	0	0	0	0
RS Redundancy Row			27		62	85	34	83	63	87	55	98	6	27	17	65	0	0	0	0
RS Redundancy Row			77		3	84	63	70	30	80	45	82	94	84	38	74	0	0	0	0
Execution Trace - Transition			22	2	54	84	41	0	1	0	0	0	0	0	0	0	0	0	0	0
RS Redundancy Row			34		42	14	11	10	58	60	86	83	84	41	65	0	0	0	0	0
RS Redundancy Row			79		96	18	86	60	59	28	55	52	96	172	47	28	0	0	0	0
RS Redundancy Row			78		69	3	29	25	20	91	88	36	4	97	86	81	0	0	0	0
Execution Trace - Termination			50	3	84	41	28	0	1	1	0	0	0	0	0	0	0	0	0	0
RS Redundancy Row			42		36	15	54	53	8	23	38	54	79	76	79	0	0	0	0	0
RS Redundancy Row			12		26	16	31	11	91	45	32	22	40	13	49	33	0	0	0	0
RS Redundancy Row			45		37	77	1	68	51	53	61	11	92	24	25	14	0	0	0	0
Random Padding			96	4	78	2	71	0	0	0	0	0	0	0	0	0	0	0	0	0
RS Redundancy Row			69		71	90	82	2	35	72	79	94	23	8	81	10	0	0	0	0
RS Redundancy Row			89		24	15	14	62	57	83	91	0	40	3	68	58	0	0	0	0
RS Redundancy Row			67		70	66	75	13	66	70	29	16	80	81	23	16	0	0	0	0
Random Padding			33	5	15	77	17	0	0	0	0	0	0	0	0	0	0	0	0	0
RS Redundancy Row			51		31	8	76	26	65	10	31	85	10	12	45	42	0	0	0	0
RS Redundancy Row			70		38	20	67	13	36	60	11	85	54	12	57	23	0	0	0	0
RS Redundancy Row			20		71	19	14	86	9	25	89	85	24	34	45	16	0	0	0	0
Random Padding			75	6	29	21	92	0	0	0	0	0	0	0	0	0	0	0	0	0
RS Redundancy Row			63		40	43	42	46	81	53	88	87	18	83	75	43	0	0	0	0
RS Redundancy Row			18		54	72	72	87	86	11	65	88	15	88	18	45	0	0	0	0
RS Redundancy Row			19		19	92	90	80	70	68	78	85	15	45	0	15	0	0	0	0
Random Padding			47	7	50	36	33	0	0	0	0	0	0	0	0	0	0	0	0	0
RS Redundancy Row			55		80	66	46	60	74	2	58	82	28	14	22	22	0	0	0	0
RS Redundancy Row			85		87	8	89	28	65	62	40	18	83	38	5	11	0	0	0	0
	28^31 mod 97		52		18	70	60	91	82	13	38	38	31	28	43	0	0	0	0	0
	28^32 mod 97		1																	

Of course, we shouldn't expect these rule checks to evaluate to 0 in the redundancy rows , as they're not directly associated with the data from the trace

Conveniently, by writing these rule checks in terms of our trace polynomials, we can convert our multi-input rule checking polynomials into single-input polynomials, which we call constraint polynomials.

Note that each `constraint polynomial` will evaluate to 0 at the input values that are associated with actual trace data.

Lesson 7: Mixing Constraint Polynomials

Here, we add one new column, which `mixes` our `constraint polynomials` into a single `mixed constraint polynomial`.

After the Prover commits a Merkle root for the `control polynomials` and the `data polynomials` (and the `PLONK polynomials` in the full implementation), those Merkle roots are used as entropy to randomly generate a `constraint mixing parameter` α_1 .

Letting c_i denote the constraint polynomials, we write:

$$C(x) = \alpha_1^0 * c_0(x) + \alpha_1^1 c_1(x) + \dots + \alpha_1^5 c_5(x)$$

Reed Solomon Input (Expanded Notation in terms of 5)	Reed Solomon Input (Simplified)	Clock Cycle	Data Column 1	Data Column 2	Data Column 3	Control Column - Initialization	Control Column - Transition	Control Column - Termination	Mixed Constraint Polynomial		Does Initialization for Data Column 1 match User Input 1?	Does Initialization for Data Column 2 match User Input 2?	Does Initialization for Data Column 3 match User Input 3?	Does Termination value for Data Column 1 match entry i-1 from Input Column 27?	Does entry i from Input Column 2 match entry i-1 from Input Column 27?	Does entry i from Input Column 2 match entry i-1 from Output Column?
Execution Trace - Initialization																
RS Redundancy Row	5^4 mod 97	5	0	31	39	12	82	81	2	52	13	89	95	0	2	45
RS Redundancy Row	5^4 mod 97	43		15	36	11	16	72	32	47	67	27	11	35	15	36
RS Redundancy Row	5^4 mod 97	40		96	65	6	32	73	65	45	34	72	52	25	84	7
RS Redundancy Row	5^4 mod 97	53		79	41	88	68	10	22	32	1	34	68	51	4	94
Execution Trace - Transition																
RS Redundancy Row	29	1	69	35	49	81	64	32	86	102	58	11	38	77	17	
RS Redundancy Row	36		31	85	50	41	58	16	77	73	83	24	81	11	24	
RS Redundancy Row	38		16	41	69	18	40	38	95	65	53	4	9	77	43	
RS Redundancy Row	94		71	24	89	86	56	50	62	12	45	98	43	31	8	
Execution Trace - Transition																
RS Redundancy Row	13	2	35	77	46	92	16	47	90	53	42	30	79	0	60	
RS Redundancy Row	73		10	40	9	59	83	24	81	41	47	8	23	60	43	
RS Redundancy Row	7		53	54	58	14	20	67	8	85	78	45	78	46	88	
RS Redundancy Row	2		28	7	95	44	92	35	45	78	78	35	11	77	22	
Execution Trace - Termination																
RS Redundancy Row	56	3	67	81	80	43	61	82	27	93	8	58	93	60	1	
RS Redundancy Row	16		26	2	73	31	21	18	54	46	83	1	14	94	27	
RS Redundancy Row	60		0	54	76	8	64	32	73	87	2	96	81	36	35	
RS Redundancy Row	31		45	36	80	92	4	68	23	10	89	87	44	50	35	
RS Redundancy Row	92	4	91	59	35	18	22	81	71	2	89	98	84	29	23	
RS Redundancy Row	54		94	39	57	71	34	41	83	88	23	21	81	28	8	
RS Redundancy Row	57		18	82	19	50	40	18	43	79	88	78	22	15	46	
RS Redundancy Row	44		80	36	44	89	28	86	44	31	30	40	18	86	25	
Random Padding																
RS Redundancy Row	68	5	54	12	61	2	48	92	16	86	81	24	51	20		
RS Redundancy Row	61		39	46	78	32	64	59	65	79	62	27	48	0	72	
RS Redundancy Row	59		16	16	16	65	72	14	52	9	82	80	24	-1	75	
RS Redundancy Row	3		19	49	95	22	31	44	18	6	94	32	14	69	58	
RS Redundancy Row	84	6	57	23	47	32	55	43	4	75	88	87	41	50	44	
RS Redundancy Row	24		74	89	18	16	37	31	18	42	24	71	78	86	18	
RS Redundancy Row	90		40	96	42	38	26	8	63	22	20	82	75	42	41	
RS Redundancy Row	95		43	54	72	50	9	92	7	8	77	88	81	48	19	
RS Redundancy Row	41	7	57	19	96	47	44	16	4	36	98	95	38	41	29	
RS Redundancy Row	81		75	8	27	24	22	71	22	44	83	54	29	80	71	
RS Redundancy Row	37		28	34	37	67	56	50	62	40	14	14	62	72	37	
RS Redundancy Row	88		96	1	51	35	64	89	57	82	85	82	18	69	15	
			5													

Note that if each c_i evaluates to 0 at some input x , then C will also evaluate to 0 for that input.

In this example, the degree of the `mixed constraint polynomial` is equal to the degree of the `trace polynomials`, because the `rule-checking` involved is particularly simple. In more complicated examples, composing our `rule checking polynomials` with our `trace polynomials` would yield `high degree constraint polynomials`. In that case, we'd add an extra step at the end of Lesson 9 to split our `high degree validity polynomial` into a few `low degree validity polynomials`.

Lesson 8: The Core of the RISC Zero STARK

The Prover constructs the validity polynomial by dividing the mixed constraint polynomial from the previous lesson by the publicly known zeros polynomial. $V(x) = C(x)/Z(x)$

In our example, the `zeros polynomial` is

$Z(x) = (x - 1)(x - 47)(x - 75)(x - 33)(x - 96)(x - 50)(x - 22)(x - 64)$, where the 8 terms are the elements of $\mathcal{D}(5^{12})$.

Input 1	24	Asserted Output	25
Input 2	30	Constraint Mixing Parameter	3
Modulo	97		
Reed Solomon Input			
Reed Solomon Input (Exponent Notation in terms of 28)	Reed Solomon Input (Exponent Notation in terms of 5)	Reed Solomon Input (Simplified)	Clock Cycle
28^0 mod 97	5^0 mod 97	5	0
28^1 mod 97	5^3 mod 97	43	
28^2 mod 97	5^6 mod 97	40	
28^3 mod 97	5^9 mod 97	53	
Execution Trace - Initialization			
RS Redundancy Row	28^0 mod 97	5^0 mod 97	31
RS Redundancy Row	28^1 mod 97	5^3 mod 97	39
RS Redundancy Row	28^2 mod 97	5^6 mod 97	12
RS Redundancy Row	28^3 mod 97	5^9 mod 97	82
Execution Trace - Transition			
RS Redundancy Row	28^0 mod 97	5^0 mod 97	72
RS Redundancy Row	28^1 mod 97	5^3 mod 97	32
RS Redundancy Row	28^2 mod 97	5^6 mod 97	65
RS Redundancy Row	28^3 mod 97	5^9 mod 97	73
Execution Trace - Termination			
RS Redundancy Row	28^0 mod 97	5^0 mod 97	68
RS Redundancy Row	28^1 mod 97	5^3 mod 97	10
RS Redundancy Row	28^2 mod 97	5^6 mod 97	22
RS Redundancy Row	28^3 mod 97	5^9 mod 97	32
Random Padding			
RS Redundancy Row	28^0 mod 97	5^0 mod 97	56
RS Redundancy Row	28^1 mod 97	5^3 mod 97	77
RS Redundancy Row	28^2 mod 97	5^6 mod 97	34
RS Redundancy Row	28^3 mod 97	5^9 mod 97	46
Random Padding			
RS Redundancy Row	28^0 mod 97	5^0 mod 97	13
RS Redundancy Row	28^1 mod 97	5^3 mod 97	2
RS Redundancy Row	28^2 mod 97	5^6 mod 97	35
RS Redundancy Row	28^3 mod 97	5^9 mod 97	77
Random Padding			
RS Redundancy Row	28^0 mod 97	5^0 mod 97	16
RS Redundancy Row	28^1 mod 97	5^3 mod 97	26
RS Redundancy Row	28^2 mod 97	5^6 mod 97	2
RS Redundancy Row	28^3 mod 97	5^9 mod 97	24
Random Padding			
RS Redundancy Row	28^0 mod 97	5^0 mod 97	60
RS Redundancy Row	28^1 mod 97	5^3 mod 97	0
RS Redundancy Row	28^2 mod 97	5^6 mod 97	54
RS Redundancy Row	28^3 mod 97	5^9 mod 97	24
Random Padding			
RS Redundancy Row	28^0 mod 97	5^0 mod 97	31
RS Redundancy Row	28^1 mod 97	5^3 mod 97	45
RS Redundancy Row	28^2 mod 97	5^6 mod 97	36
RS Redundancy Row	28^3 mod 97	5^9 mod 97	80
Random Padding			
RS Redundancy Row	28^0 mod 97	5^0 mod 97	92
RS Redundancy Row	28^1 mod 97	5^3 mod 97	4
RS Redundancy Row	28^2 mod 97	5^6 mod 97	91
RS Redundancy Row	28^3 mod 97	5^9 mod 97	59
Random Padding			
RS Redundancy Row	28^0 mod 97	5^0 mod 97	92
RS Redundancy Row	28^1 mod 97	5^3 mod 97	4
RS Redundancy Row	28^2 mod 97	5^6 mod 97	91
RS Redundancy Row	28^3 mod 97	5^9 mod 97	59
Random Padding			
RS Redundancy Row	28^0 mod 97	5^0 mod 97	92
RS Redundancy Row	28^1 mod 97	5^3 mod 97	4
RS Redundancy Row	28^2 mod 97	5^6 mod 97	91
RS Redundancy Row	28^3 mod 97	5^9 mod 97	59
Random Padding			
RS Redundancy Row	28^0 mod 97	5^0 mod 97	68
RS Redundancy Row	28^1 mod 97	5^3 mod 97	39
RS Redundancy Row	28^2 mod 97	5^6 mod 97	16
RS Redundancy Row	28^3 mod 97	5^9 mod 97	71
Random Padding			
RS Redundancy Row	28^0 mod 97	5^0 mod 97	57
RS Redundancy Row	28^1 mod 97	5^3 mod 97	18
RS Redundancy Row	28^2 mod 97	5^6 mod 97	50
RS Redundancy Row	28^3 mod 97	5^9 mod 97	40
Random Padding			
RS Redundancy Row	28^0 mod 97	5^0 mod 97	44
RS Redundancy Row	28^1 mod 97	5^3 mod 97	36
RS Redundancy Row	28^2 mod 97	5^6 mod 97	44
RS Redundancy Row	28^3 mod 97	5^9 mod 97	28
Random Padding			
RS Redundancy Row	28^0 mod 97	5^0 mod 97	68
RS Redundancy Row	28^1 mod 97	5^3 mod 97	12
RS Redundancy Row	28^2 mod 97	5^6 mod 97	61
RS Redundancy Row	28^3 mod 97	5^9 mod 97	48
Random Padding			
RS Redundancy Row	28^0 mod 97	5^0 mod 97	68
RS Redundancy Row	28^1 mod 97	5^3 mod 97	16
RS Redundancy Row	28^2 mod 97	5^6 mod 97	64
RS Redundancy Row	28^3 mod 97	5^9 mod 97	59
Random Padding			
RS Redundancy Row	28^0 mod 97	5^0 mod 97	59
RS Redundancy Row	28^1 mod 97	5^3 mod 97	16
RS Redundancy Row	28^2 mod 97	5^6 mod 97	65
RS Redundancy Row	28^3 mod 97	5^9 mod 97	59
Random Padding			
RS Redundancy Row	28^0 mod 97	5^0 mod 97	84
RS Redundancy Row	28^1 mod 97	5^3 mod 97	23
RS Redundancy Row	28^2 mod 97	5^6 mod 97	47
RS Redundancy Row	28^3 mod 97	5^9 mod 97	32
Random Padding			
RS Redundancy Row	28^0 mod 97	5^0 mod 97	84
RS Redundancy Row	28^1 mod 97	5^3 mod 97	24
RS Redundancy Row	28^2 mod 97	5^6 mod 97	49
RS Redundancy Row	28^3 mod 97	5^9 mod 97	38
Random Padding			
RS Redundancy Row	28^0 mod 97	5^0 mod 97	95
RS Redundancy Row	28^1 mod 97	5^3 mod 97	43
RS Redundancy Row	28^2 mod 97	5^6 mod 97	72
RS Redundancy Row	28^3 mod 97	5^9 mod 97	50
Random Padding			
RS Redundancy Row	28^0 mod 97	5^0 mod 97	41
RS Redundancy Row	28^1 mod 97	5^3 mod 97	57
RS Redundancy Row	28^2 mod 97	5^6 mod 97	19
RS Redundancy Row	28^3 mod 97	5^9 mod 97	47
Random Padding			
RS Redundancy Row	28^0 mod 97	5^0 mod 97	81
RS Redundancy Row	28^1 mod 97	5^3 mod 97	75
RS Redundancy Row	28^2 mod 97	5^6 mod 97	27
RS Redundancy Row	28^3 mod 97	5^9 mod 97	24
Random Padding			
RS Redundancy Row	28^0 mod 97	5^0 mod 97	37
RS Redundancy Row	28^1 mod 97	5^3 mod 97	28
RS Redundancy Row	28^2 mod 97	5^6 mod 97	34
RS Redundancy Row	28^3 mod 97	5^9 mod 97	67
Random Padding			
RS Redundancy Row	28^0 mod 97	5^0 mod 97	66
RS Redundancy Row	28^1 mod 97	5^3 mod 97	96
RS Redundancy Row	28^2 mod 97	5^6 mod 97	1
RS Redundancy Row	28^3 mod 97	5^9 mod 97	51
Random Padding			
RS Redundancy Row	28^0 mod 97	5^0 mod 97	5

Normally, when we divide two low degree polynomials, we don't expect to get another low degree polynomial. But for an honest prover, it's not hard to see that $V(x)$ will be lower degree than $C(x)$, since the roots of $Z(x)$ line up perfectly with roots of $C(x)$.

The Prover evaluates $V(x)$ over the $5, 5^4, \dots, 5^{94}$, commits the values to a [validity Merkle tree](#), and appends the root to the seal.

The construction of these polynomials is the core conceptual thrust of RISC Zero's proof of trace validity. All of the information necessary to confirm the validity of the original Execution trace can be described in the following assertions about these polynomials:

- (i) $V(x) = C(x) / Z(x)$ for all x
- (ii) The degree of the validity polynomial and each trace polynomials are less than or equal to 7.

The FRI protocol is the technique we use for proving (ii). Those details are omitted from this simplified example.

In the original STARK protocol, the Verifier tests (i) at a number of test points; the soundness of the protocol depends on the number of tests. The DEEP-ALI technique allows us to achieve a high degree of soundness with a single test. The details of DEEP are described in the following lesson.

Lesson 9: The DEEP Technique

Here, we use the [trace polynomials](#) and the [validity polynomial\(s\)](#) to construct the [DEEP polynomials](#).

The DEEP polynomials allow the Verifier to test $V(x) = C(x) / Z(x)$ outside the original Merkle tree commitments, which substantially improves the robustness of the Verifier's test.

Without the DEEP technique, the Prover would assert that the trace polynomials $d_1, d_2, d_3, c_1, c_2, c_3$, and the validity polynomial V were all low degree polynomials. With the DEEP technique, the Prover argues instead that $d'_1, d'_2, d'_3, c'_1, c'_2, c'_3$, and V' are low degree polynomials.

With commitments of the `trace polynomials` and the `validity polynomial` in place, the Verifier uses the entropy of the seal to randomly choose a `DEEP test point`, z . We use $z = 93$ in this example.

The Verifier would like to be able to compute the `mixed constraint polynomial`, $C(93)$. The Prover sends $V(93)$ and the `necessary evaluations` of the `trace polynomials` to allow the Verifier to compute $C(93)$.

In this example, the `necessary evaluations` are $d_1(93), d_2(93), d_3(93), c_1(93), c_2(93), c_3(93), d_2(93 \cdot 5^{-12}), d_3(93 \cdot 5^{-12})$, shown in salmon. Note that 5^{-12} is a pointer backwards 1 computational step; by pointing forward and backward, the `taps` allow for checking the rules that span multiple clock-cycles.

Input 1		24				Input 2		30				Asserted Output		25				
												Constraint Mixing Parameter		7				
												Random Test Point		93				
Modulo		93																
Reed-Solomon Input (Exponent Notation in terms of 28)	Reed-Solomon Input (Exponent Notation in terms of 5)	Reed-Solomon Input (Exponent Notation in terms of 5)	Reed-Solomon Input (Simplified)	Clock Cycle	Data Column 1	DEEP Polynomial 1	Data Column 2	DEEP Polynomial 2	Data Column 3	DEEP Polynomial 3	Control Column + Initialization	DEEP Polynomial 4	Control Column + Transition 5	DEEP Polynomial 5	Control Column + Termination	DEEP Polynomial 6	Validity Polynomial: $V(x) = C(x) / Z(x)$	DEEP Validity Polynomial
Disguised Execution Trace	$5^{28} \equiv 0 \pmod{97}$	$5^{11} \equiv 59 \pmod{97}$	5	0	31	50	19	19	12	84	82	47	81	4	2	95	53	
Disguised RS Redundancy Row	$5^{28} \equiv 1 \pmod{97}$	$5^{14} \equiv 37 \pmod{97}$	1	15	34	36	74	11	63	18	84	72	79	32	85	67	84	
Disguised RS Redundancy Row	$5^{28} \equiv 2 \pmod{97}$	$5^{17} \equiv 59 \pmod{97}$	40	96	58	65	82	6	89	32	68	73	80	65	87	49	32	
Non-Merkle Points	$5^{28} \equiv 3 \pmod{97}$	$5^{10} \equiv 67 \pmod{97}$	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	
Disguised RS Redundancy Row	$5^{28} \equiv 4 \pmod{97}$	$5^{13} \equiv 59 \pmod{97}$	53	79	70	41	13	88	93	68	31	10	13	22	63	53	53	
Disguised Execution Trace	$5^{28} \equiv 5 \pmod{97}$	$5^{16} \equiv 37 \pmod{97}$	29	1	69	53	35	48	49	75	81	51	64	77	32	18	37	
Disguised RS Redundancy Row	$5^{28} \equiv 6 \pmod{97}$	$5^{19} \equiv 59 \pmod{97}$	36	31	84	85	30	50	38	41	92	58	27	16	29	65	7	
Disguised RS Redundancy Row	$5^{28} \equiv 7 \pmod{97}$	$5^{11} \equiv 59 \pmod{97}$	36	16	45	41	17	69	11	18	94	40	53	38	42	28	3	
Non-Merkle Points	$5^{28} \equiv 8 \pmod{97}$	$5^{18} \equiv 67 \pmod{97}$	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	
Disguised RS Redundancy Row	$5^{28} \equiv 9 \pmod{97}$	$5^{10} \equiv 67 \pmod{97}$	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	
Disguised Execution Trace	$5^{28} \equiv 10 \pmod{97}$	$5^{13} \equiv 59 \pmod{97}$	54	71	5	24	58	89	80	86	39	56	11	50	30	37		
Disguised RS Redundancy Row	$5^{28} \equiv 11 \pmod{97}$	$5^{16} \equiv 37 \pmod{97}$	13	35	21	77	93	48	44	92	54	16	4	47	13	16	1	
Disguised RS Redundancy Row	$5^{28} \equiv 12 \pmod{97}$	$5^{19} \equiv 59 \pmod{97}$	73	10	61	40	63	9	23	59	77	83	66	24	58	56	2	
Disguised RS Redundancy Row	$5^{28} \equiv 13 \pmod{97}$	$5^{11} \equiv 59 \pmod{97}$	7	53	87	54	64	58	40	14	84	20	33	67	66	13	63	
Disguised Execution Trace	$5^{28} \equiv 14 \pmod{97}$	$5^{18} \equiv 67 \pmod{97}$	2	28	26	7	42	95	11	44	48	92	24	35	51	23	4	
Disguised Execution Trace	$5^{28} \equiv 15 \pmod{97}$	$5^{10} \equiv 67 \pmod{97}$	56	3	87	76	81	45	80	80	43	84	61	52	82	86	83	

These 8 points, together with the publicly known rule-checking functions, allow the Verifier to manually compute $C(93)$ and therefore $V(93)$.

The Prover also constructs the `DEEP polynomials`, interpolates each one, and sends the coefficients of each DEEP polynomial to the Verifier. The `DEEP polynomials` are defined as follows:

$$d'_1(x) = \frac{d_1(x) - d_1(93)}{x - 93}$$

$d'_2(x) = \frac{d_2(x) - \overline{d_2}(x)}{(x - 93)(x - 6)}$ where $\overline{d_2}(x)$ is constructed by interpolating $(6, d_2(6))$ and $(93, d_2(93))$.

$d'_3(x) = \frac{d_3(x) - \overline{d_3}(x)}{(x - 93)(x - 6)}$ where $\overline{d_3}(x)$ is constructed by interpolating $(6, d_3(6))$ and $(93, d_3(93))$

$$c'_1(x) = \frac{c_1(x) - c_1(93)}{x - 93}$$

$$c'_2(x) = \frac{c_2(x) - c_2(93)}{x - 93}$$

$$c'_3(x) = \frac{c_3(x) - c_3(93)}{x - 93}$$

$V'(x) = \frac{V(x) - V(93)}{x - 93}$ where the Prover computes $V(93)$ by running `INTT(ValidityColumn)` and then evaluating the resulting `validity polynomial` at $z = 93$.

Lesson 10: Mixing for FRI

After using the `DEEP polynomials` to check the relation between the Trace Polynomial, the `validity polynomial`, and the `zeros polynomial` at $z=93$, the only thing left for the Prover to do is to show that the `DEEP polynomials` are low-degree.

The FRI protocol provides a mechanism for the Verifier to confirm the low-degree-ness of polynomials, with very little computation required of the Verifier. In order to reduce this assertion of low-degree-ness to a single application of FRI, the Prover mixes the `DEEP`

`polynomials` into a single FRI polynomial, using the DEEP Mixing parameter, $\backslashalpha_2\alpha2$.

Letting $c'_1, c'_2, c'_3, d'_1, d'_2, d'_3$, and V' denote the `DEEP polynomials`, we mix the `DEEP polynomials` to construct the FRI polynomial, $f_0(x) = \alpha_2^0 c'_1(x) + \alpha_2^1 c'_2(x) + \dots + \alpha_2^6 V'(x)$

Input 1	24
Input 2	30
Modulo	97

Asserted Output	28
Constraint Mixing Parameter	3
Random Test Point	93
DEEP Mixing Parameter	21

	Reed Solomon Input (Exponent Notation in terms of 28)	Reed Solomon Input (Exponent Notation in terms of 5)	Reed Solomon Input (Simplified)	Clock Cycle	DEEP Polynomial 1	DEEP Polynomial 2	DEEP Polynomial 3	DEEP Polynomial 4	DEEP Polynomial 5	DEEP Polynomial 6	DEEP Validity Polynomial	FRI Polynomial
Disguised Execution Trace												
Disguised RS Redundancy Row	$5^{28}0 \bmod 97$	$5^1 \bmod 97$	5	0	50	19	84	47	4	95	53	53
Disguised RS Redundancy Row	$5^{28}1 \bmod 97$	$5^4 \bmod 97$	43		34	74	63	84	79	89	84	69
Disguised RS Redundancy Row	$5^{28}2 \bmod 97$	$5^7 \bmod 97$	40		58	82	89	68	80	87	32	63
Disguised RS Redundancy Row	$5^{28}3 \bmod 97$	$5^{10} \bmod 97$	53		70	13	93	31	13	63	52	38
Disguised Execution Trace												
Disguised RS Redundancy Row	$5^{28}4 \bmod 97$	$5^{13} \bmod 97$	29	1	53	48	75	51	77	18	37	46
Disguised RS Redundancy Row	$5^{28}5 \bmod 97$	$5^{16} \bmod 97$	36		84	30	36	92	27	29	7	13
Disguised RS Redundancy Row	$5^{28}6 \bmod 97$	$5^{19} \bmod 97$	38		45	17	11	94	53	42	3	68
Disguised RS Redundancy Row	$5^{28}7 \bmod 97$	$5^{22} \bmod 97$	94		5	58	80	39	11	30	37	58
Disguised Execution Trace												
Disguised RS Redundancy Row			13	2	21	93	44	54	4	13	1	38
Disguised RS Redundancy Row			73		61	63	23	77	66	58	2	3
Disguised RS Redundancy Row			7		87	64	40	94	33	66	63	95
Disguised RS Redundancy Row			2		26	42	11	48	24	51	4	23
Disguised Execution Trace					56	3	76	45	80	84	52	79
Disguised RS Redundancy Row			16		95	10	33	38	57	29	88	39
Disguised RS Redundancy Row			60		2	21	38	10	20	79	28	62
Disguised RS Redundancy Row			31		77	71	50	29	21	18	84	19
Disguised Random Padding			92	4	72	1	72	37	23	36	22	62
Disguised RS Redundancy Row			54		54	52	33	74	55	89	0	58
Disguised RS Redundancy Row			57		66	72	35	8	19	27	14	41
Disguised RS Redundancy Row			44		69	71	78	13	34	62	22	67
Disguised Random Padding			68	5	16	60	8	60	93	1	72	89
Disguised RS Redundancy Row			61		16	10	8	52	57	20	23	41
Disguised RS Redundancy Row			59		30	8	86	28	42	23	87	58
Disguised RS Redundancy Row			3		21	44	36	38	95	45	7	74
Disguised Random Padding			84	6	1	41	9	34	42	19	58	95
Disguised RS Redundancy Row			24		28	5	25	37	69	87	51	98
Disguised RS Redundancy Row			90		41	14	88	3	71	4	26	72
Disguised RS Redundancy Row			95		37	16	95	50	79	36	26	20
Disguised Random Padding			41	7	58	49	65	0	28	86	66	82
Disguised RS Redundancy Row			81		72	15	22	10	10	20	36	33
Disguised RS Redundancy Row			37		18	78	50	62	5	93	38	0
Disguised RS Redundancy Row	$5^{28}31 \bmod 97$	$5^{93} \bmod 97$	66		42	41	91	22	46	19	59	16
Disguised RS Redundancy Row	$5^{28}32 \bmod 97$	$5^{96} \bmod 97$	5									

To complete the argument, the Prover constructs a FRI proof that $f_0(x)$ is a low degree polynomial. With this process, the Prover has constructed a zero-knowledge argument of computational integrity. We omit the details of FRI for brevity, but we can check our work by running an iNTT on the evaluations of f_0 :

iNTT([53,69,63,30,46,13,60,50,38,3,95,23,75,39,62,19,62,58,41,67,89,41,50,24,95,90,72,20,82,33,0,16

returns

[19, 56, 34, 48, 43, 37, 10, 0]

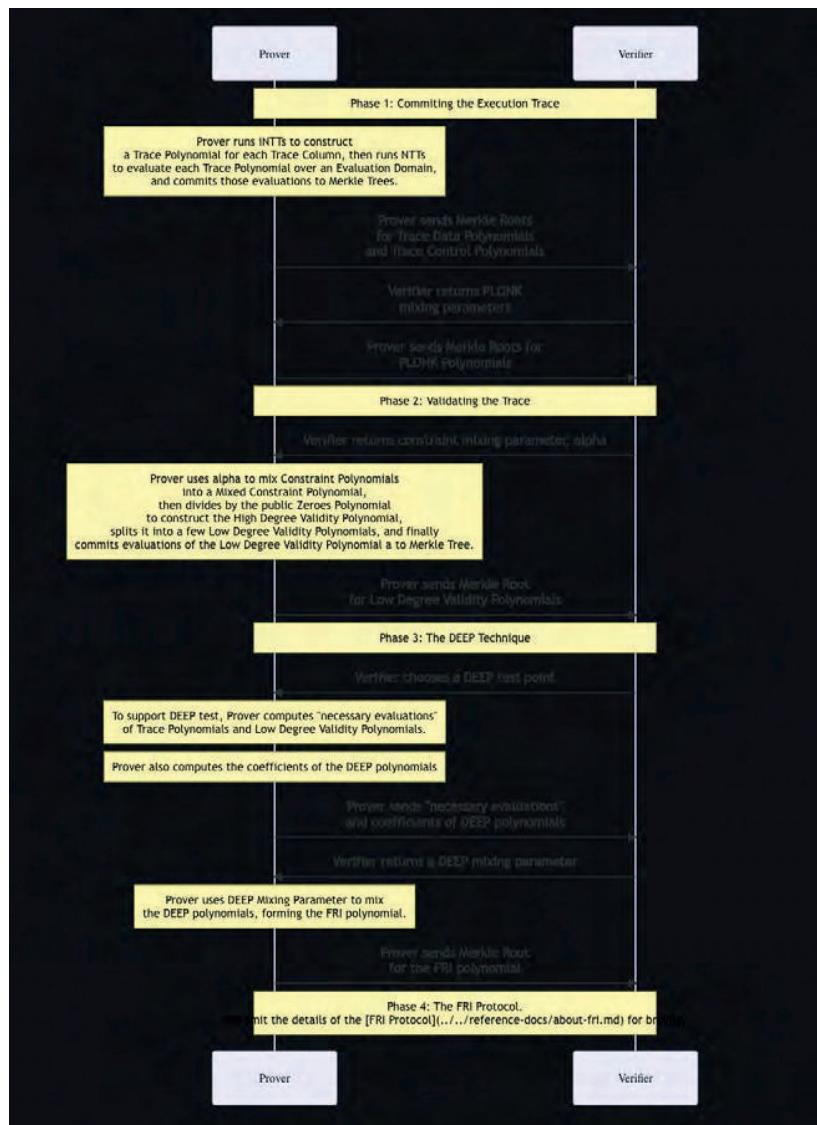
Writing this array as the coefficients of a polynomial, we see that the values in the FRI polynomial column do, in fact, correspond to the following low-degree polynomial:

$$f_0(x) = 19 + 56x + 34x^2 + 48x^3 + 43x^4 + 37x^5 + 10x^6$$

ZKP Sequence Diagram and Spec

RISC Zero offers a computational receipt for any code that runs within the RISC Zero zkVM, which serves to verifiably link the code that ran to the asserted output. RISC Zero's receipts are built on the shoulders of several recent advances in the world of Zero-Knowledge Cryptography: zk-STARKs, PLONK, and DEEP-ALI.

In this document, we present a succinct introduction to the RISC Zero Proof system, including a sequence diagram and a step-by-step description of the RISC Zero Non-Interactive Argument of Knowledge. We encourage readers to follow along with Constructing a Seal for a more concrete description of the protocol.



The RISC Zero Proof System: A Step-By-Step Description

Phase 1: Committing the Execution Trace

- The Prover runs a computation in order to generate an `Execution Trace`.
 - The `trace` is organized into `columns`, and the columns are categorized as `control columns`, `data columns`, and `PLONK columns`.
 - The `control columns` handle system initialization and shutdown, the initial program code to load into memory before execution, and other control signals that don't depend on the program execution.
 - The `data columns` contain the input and the computation data, both of which are private. These columns are committed in two orderings:
 - in order of program execution, and
 - re-ordered by register first and clock cycle second. The re-ordered columns allow for efficient validation of RISC-V memory operations.
 - The `PLONK columns` are used to show the validity that the re-ordered data columns are a valid permutation of the original data, as per the PLONK permutation argument.
 - After computing the `data columns` and `PLONK columns`, the Prover adds some random `noise` to the end of those columns in order to ensure that the protocol is zero-knowledge.
- The Prover encodes the `trace` as follows:
 - The Prover converts each `column` into a polynomial using an `iNTT`. We'll refer to these as `Trace Polynomials`, denoted $P_i(x)$.
 - The Prover evaluates the `data polynomials` and the `control polynomials` over an expanded domain and commits the evaluations into two separate Merkle Trees.
 - Using these Merkle roots as an entropy-source, we use Fiat-Shamir to choose `PLONK mixing parameters`, using a SHA-2 CRNG.
 - Then, the Prover uses the `PLONK mixing parameters` to generate the `PLONK columns`, interpolates them to form the `PLONK polynomials`, evaluates those polynomials over a larger domain, and commits those evaluations to a Merkle tree (see the PLONK paper for details).
 - The Prover sends the Merkle root of each tree to the Verifier.
 - Using these three Merkle roots as an entropy-source, we use Fiat-Shamir to choose a `constraint mixing parameter` α , using a SHA-2 CRNG.

Phase 2: Validating the Trace: the Core of the STARK Proof

- The Prover uses the `constraint mixing parameter`, the `Trace Polynomials`, and the `Rule Checking Polynomials` to construct a few `Low Degree Validity Polynomials`. The details are as follows:
 - By writing k publicly known `Rule Checking Polynomials`, R_0, R_1, \dots, R_{k-1} , in terms of the private `Trace Polynomials`, the Prover generates k `Constraint Polynomials`, $C_j(x)$.
 - The key point about these polynomials is that for each of the k rules and each input z that's associated with the trace, $C_j(z)$ will return 0 if the trace "passes the test," so to speak.
 - Using the `constraint mixing parameter` α , the Prover combines the `Constraint Polynomials`, C_j , into a single `Mixed Constraint Polynomial`, C , by computing $C(x) = \alpha^0 C_0(x) + \dots + \alpha^{k-1} C_{k-1}(x)$.
 - Note that if each C_j returns 0 at some point z , then C will also return 0 at z .
 - Using a publicly known `Zeros Polynomial`, the Prover computes the `High Degree Validity Polynomial`, $V(x) = \frac{C(x)}{Z(x)}$.

- The `Zeros Polynomial` $Z(x)$ is a divisor of any honest construction of $C(x)$. In other words, an honest prover will construct $V(x)$ to be a polynomial of lower degree than $C(x)$. We call V "high degree" relative to the Trace Polynomials, P_i .
 - The Prover `splits` the `High Degree Validity Polynomial` into 4 `Low Degree Validity Polynomials`, $v_0(x), v_1(x), \dots, v_3$.
 - The Prover evaluates the `Low Degree Validity Polynomials`, encodes them in a Merkle Tree, and sends the Merkle root to the Verifier.
 - We use Fiat-Shamir to choose the `DEEP Test Point`, z .

Phase 3: The DEEP Technique

- The Verifier would like to check the asserted relation between C , Z , and V at the `DEEP Test Point`, z . Namely, the Verifier would like to confirm that $V(z)Z(z) = C(z)$.
 - The Prover sends the evaluations of each v_i at z , which allows the Verifier to compute $V(z)$.
 - Computing $C(z)$ is slightly more complicated. Because `rule checks` can check relationships across multiple `columns` and multiple `clock cycles`, evaluating $C(z)$ requires numerous evaluations of the form $P_i(\omega^n z)$ for varying `columns` i and `cycles` n . The Prover sends these `necessary evaluations` of each P_i to allow the Verifier to evaluate $C(z)$. We refer to the `necessary evaluations` $P_i(\omega^n z)$ as the `taps` of P_i at z .
 - The Verifier can now check $V(z)Z(z) = C(z)$.
 - Although these asserted evaluations have no associated Merkle branches, the DEEP technique offers an alternative to the usual Merkle proof.
- The Prover constructs the DEEP polynomials using the `taps` :
 - Denoting the `taps` of P_i at z as $(x_1, P_i(x_1)), \dots, (x_n, P_i(x_n))$, the Prover constructs the DEEP polynomial $P_i'(x) = \frac{P_i(x) - \overline{P}_i(x)}{(x-x_1)\dots(x-x_n)}$ where $\overline{P}_i(x)$ is the polynomial formed by interpolating the taps of P_i . The Prover computes P_i' runs an iNTT on the result, and sends the coefficients of P_i' to the Verifier. Using this technique, the Prover constructs and sends a DEEP polynomial for each P_i and each v_i .
 - At this point, the claim of trace validity has been reduced to the claim that each of the DEEP polynomials is actually a low-degree polynomial. To conclude the proof, the Prover mixes the DEEP polynomials into the `FRI Polynomial` using a `DEEP mixing parameter` and use the FRI protocol to show that the `FRI Polynomial` is a low-degree polynomial.

Phase 4: The FRI Protocol

- We omit the details of the FRI Protocol for brevity.

RISC Zero zkVM: Scalable, Transparent Arguments of RISC-V Integrity

Jeremy Bruestle and Paul Gafni
RISC Zero



RISC Zero zkVM: Scalable, Transparent Arguments of RISC-V Integrity

January 3, 2023

Contents

1	Introduction	1
1.1	Verifiable Computing	1
1.2	Verifiable RISC-V	1
1.3	A Formally Verified Verifier	1
1.4	The RISC Zero Argument System	2
1.5	Paper Organization	2
2	Background	2
2.1	The Columns of the Execution Trace	2
2.2	Enforcing Computational Integrity via Constraints	3
2.3	Arguments and Proofs of Knowledge	3
3	The RISC Zero IOP Protocol	4
3.1	Overview of the Protocol	4
3.2	IOP Definitions	5
3.3	IOP Protocol Specification	6
3.4	Soundness Analysis in the IOP Model	7
3.4.1	Error Bound on PLOOKUP	8
3.4.2	Error Bound on DEEP-ALI	8
3.4.3	Error Bound on FRI	8
	Appendices	11
A	Preliminaries	11
A.1	Cyclic, Sequential Indexing	11
A.2	Blocks: Indexing, Metrics, and Operations	12
A.2.1	The Structure of a Block	12
A.2.2	Block Distance and Divergence	12
A.2.3	Operations on Blocks	13
A.3	Reed Solomon Codes	14
A.4	Block Operations on RS Codes	15
A.4.1	Motivation	15
A.4.2	Operations on Valid Blocks	16
A.4.3	Operations and Block Proximity	16
A.5	Constraints and Taps	19
A.5.1	Constraints	19
A.5.2	Trace Taps and Polynomial Taps	20
B	Merkle Tree Structure	21
C	Protocol Details	22
C.1	Constructing $\mathbf{w}_{\text{control}}$ and \mathbf{w}_{data}	22
C.2	Constructing $\text{Com}(\mathbf{w}_{\text{control}})$ and $\text{Com}(\mathbf{w}_{\text{data}})$	23
C.3	Randomness for PLOOKUP Accumulation	23
C.4	Constructing $\text{Com}(\mathbf{w}_{\text{accum}})$	23
C.5	DEEP-ALI: Constructing $\text{Com}(\mathbf{w}_{\text{validity}})$ and the DEEP answer sequence	23

C.5.1	Randomness for Algebraic Linking	23
C.5.2	Constructing $\text{Com}(\mathbf{w}_{\text{validity}})$	23
C.5.3	Randomness for DEEP	24
C.5.4	Intuition for DEEP technique	24
C.5.5	Constructing the DEEP answer sequence	24
C.6	The Batched FRI Protocol	25
D	Key Theorems for Soundness Analysis	27

Draft

Abstract

RISC Zero offers an open source zero-knowledge virtual machine (zkVM) along with a zero-knowledge proof system for verifying the validity of any computational method that executes within the zkVM. The zkVM and the proof system combine to form a zero-knowledge, scalable, transparent argument of computational integrity. Put technically, the RISC Zero protocol is a zk-STARK system using the original FRI protocol and DEEP-ALI, using SHA-2 HMACs as PRNGs in order to implement Fiat-Shamir. In this paper, we formally articulate the RISC Zero Proof System with as few technical barriers to understanding as possible, including a detailed construction of a RISC Zero cryptographic seal.

1 Introduction

1.1 Verifiable Computing

The era of verifiable computing is upon us: we now live in a world where the actions of untrusted parties in distributed systems can be authenticated as trustworthy quickly and easily, using *arguments of computational integrity*. This technology is the result of decades of incremental progress in the field of zero-knowledge cryptography and interactive proofs [14, 1, 7]. Over the past few years, it has become practical and impactful to implement in real-world applications [6, 18]. Although initial applications were mostly focused on privacy in blockchains, the technology has evolved to the point that arguments of computational integrity are staged to take over as a key infrastructural piece of the way we interact digitally. We expect that verifiable computation is not only the answer to the question of blockchain scaling, but also to fixing problems like Twitter bots and telephone spam. In a world where trust is becoming more and more scarce, verifiable computing provides a path forward.

Thousands of developers are eager to start writing verifiable software. But current systems for writing verifiable software require developers to learn brand new languages with various limitations and challenges. In order to make the world of verifiable software development available in languages like Rust and C++, **RISC Zero has built a mechanism for demonstrating the integrity of any RISC-V computation.**

1.2 Verifiable RISC-V

When a RISC-V binary executes in the RISC Zero zkVM, the output is paired with a computational receipt. The receipt contains a cryptographic seal, which serves as a zero-knowledge argument of computational integrity. By offering computational receipts for any code that will compile to RISC-V, our zkVM offers a platform to build truly trustable software, where skeptical third parties can verify authenticity in milliseconds. The idea of “running a verifier” to assure the integrity of massive computations is a novel addition to the world of digital security.

1.3 A Formally Verified Verifier

Formal verification is a process of translating code into a mathematical proof system. Formal verification is used to ensure that a piece of code is actually doing what it’s supposed to be doing. To ensure that the RISC Zero verifier (which checks the receipts) is functioning without security bugs, we’re working on a formal verification of the Verifier.

An argument of computational integrity paired with a formally verified verifier and a formally verified compiler like CompCert [16] creates a deeper formal verification stack than the industry has seen before.

1.4 The RISC Zero Argument System

RISC Zero’s argument system is based on [4]. The seal on a RISC Zero receipt is a zk-STARK[4] that uses a AIR[4] arithmetization, a PLOOKUP argument [11], DEEP-ALI[8], and FRI[3].

1.5 Paper Organization

In this paper, we specify the RISC Zero protocol and analyze its knowledge soundness. The paper is organized as follows:

In Section 2, we introduce some key ideas and background. In particular, we introduce the following idea: *An assertion of computational integrity can be viewed as an assertion that an execution trace satisfies a set of constraints.*

In Section 3, we present the interactive version of the protocol and analyze its soundness in the Interactive Oracle Proof model [7].

2 Background

The argument system behind RISC Zero’s receipts is built in terms of an execution trace and a number of constraints that enforce checks of computational integrity. We start by introducing those terms as they’re used in the context of the RISC Zero protocol.

2.1 The Columns of the Execution Trace

When a piece of code runs on a machine, the execution trace (or simply, the trace) is a complete record of the computation: a snapshot of the full state of the machine at each clock cycle of the computation. It’s typical to write an execution trace as a rectangular array, where each row shows the complete state of the machine at a given moment in time, and each column shows a temporal record of some particular aspect of the computation (say, the value stored in a particular RISC-V register) at each clock cycle.

The columns of the RISC Zero execution trace are categorized as follows:

Control Columns - Public The data in these columns describe the RISC-V architecture, the encoding of the program, and various control signals. In order to execute the instructions contained in the RISC-V ELF file, the instructions are first loaded into the Control Columns; these columns are read-only after the binary file has been loaded.

Data Columns - Private The data in these columns represents the running state of the processor and memory. In order to efficiently check the integrity of RISC-V memory operations, each register associated with RISC-V memory operations has two associated columns: one in the original execution order and the second sorted first by memory location and then by clock-cycle.

Accumulator Columns - Private To ensure the integrity of the duplicated data columns, we use PLOOKUP[11], a derivative of PLONK[12].

2.2 Enforcing Computational Integrity via Constraints

An assertion of computational integrity can be re-framed as an assertion that an execution trace satisfies a certain set of constraints. These constraints enforce that the step-by-step computation was carried out faithfully according to the underlying machine instructions and instruction set architecture.

Example 1. The constraint $(k)(k - 1) = 0$ enforces that k is either 0 or 1.

Example 2. The constraint $j - 2k = 0$ enforces that $j = 2k$.

Enforcing computational integrity of our implementation of the RISC-V instruction set architecture involves thousands of constraints, each of which is expressed as a multi-variable polynomial¹. At a high level, the constraints enforce that each of the following phases of the zkVM operation was executed as claimed:

- Init: Initialization of all registers to 0
- Setup: Prepare to load ELF file
- Load: Loading the RISC-V binary into memory
- Reset: Ends the loading phase and prepares for execution.
- Body: Main execution phase
- RamFini: Generates the memory-based grand product accumulation values for our memory permutation[12]
- BytesFini: Generates the bytes-based grand product accumulation values for PLOOKUP[11]

Together, these constraints enforce that the purported output of the computation agrees with the expected rv32im execution.

2.3 Arguments and Proofs of Knowledge

The seal on the RISC Zero receipt is a STARK – a scalable, transparent argument of knowledge [4]. An argument of knowledge allows a Prover to justify an “assertion of knowledge.” More formally, the Prover asserts knowledge of a witness \mathbf{w} that satisfies some constraints \mathbf{x} . In the case of an assertion of computational integrity, the witness is the execution trace and the constraints are the various rules that define computational integrity.

In Section 3.3, we present the protocol as an Interactive Oracle Proof (IOP) [7]. IOPs involve a series of interactions between a Prover and a Verifier, where the Prover’s messages depend on randomness supplied by the Verifier at various points throughout the protocol. The IOP protocol is a theoretical model; in practice, the zkVM uses a non-interactive version of this protocol where a SHA-2 HMAC is used

¹The inputs to these polynomials may include various register-values at various clock-cycles.

as a CRNG to simulate the Verifier participation.

In the context of the IOP protocol, the seal constitutes a proof of knowledge. In code, the *proof* becomes an *argument*². More specifically, the seal is a STARK. The Fiat-Shamir Heuristic allows us to derive security results for our STARK based on soundness analysis of the IOP protocol [10].

3 The RISC Zero IOP Protocol

In this section, we present the interactive version of the RISC Zero protocol and analyze the knowledge soundness of the protocol in the context of the IOP model [7].

3.1 Overview of the Protocol

In an interactive oracle proof, a Prover convinces a skeptical Verifier of some assertion via a series of interactions. In each round, the Prover commits to evaluations of one or more functions over a domain known to both parties. The Verifier may *query* these Prover messages without reading them in full. The IOP model idealizes these queries using the concept of *oracle access*. This theoretical model allows us to prove soundness of our protocol without reference to any cryptographic primitives.³

The Prover writes to the seal throughout the protocol. When the zkVM finishes execution of a method, the resultant seal serves as a zero-knowledge proof of computational integrity, linking a cryptographic methodID (which identifies the RISC-V binary file that was executed) to the asserted code output in a way that third-parties can quickly verify.

The protocol consists of a transparent⁴ set-up phase and a main phase. The set-up phase establishes certain protocol parameters known to both the Prover and the Verifier, including the number and length of the trace columns as well as a full enumeration of the constraints that are to be enforced.

With the set-up complete, the Prover commits to the execution trace. To generate the trace commitment, the trace columns are encoded into trace blocks using Reed-Solomon encoding, and the trace blocks are committed to Merkle trees, according to the column organization described in Section 2.1⁵. For each tree, a Merkle cap⁶ is committed to the seal.

²The argument depends on the security of SHA-256 whereas the proof has no cryptographic primitives.

³In code, the zkVM uses SHA-based Merkle trees to implement these queries, which introduces the sole cryptographic primitive in our security analysis.

⁴SNARKs typically depend on a trusted set-up phase; the T in STARKs (transparent) highlights this differentiating factor.

⁵One Merkle Tree containing the Control Blocks as leaf contents and another for the Data Blocks. After committing these two trees, the Prover constructs *accum* blocks and commits them to a third Merkle Tree, which allows verification of the trace re-ordering.

⁶Because we reveal many branches from each tree, Merkle caps offer an efficiency improvement over Merkle roots. See Appendix B and [9] for more details.

If this encoding process is carried out faithfully, the trace blocks will be **valid Reed-Solomon codewords**. Moreover, the entries of the trace blocks will be algebraically related by the various constraints defined by the RISC-V ISA and the permutation check argument. In other words, encoding the execution trace using Reed-Solomon encoding reduces the assertion of computational integrity to an assertion that these Merkle caps correspond to algebraically related Reed-Solomon codewords.

As in [18] and [19], we use the DEEP-ALI protocol[8] to further reduce this assertion to a collection of Reed-Solomon Proximity Testing (RPT) problems and use the batched FRI protocol[5] to solve the resulting RPT problems.

We introduce the notation for our IOP protocol in Section 3.2, articulate the protocol in detail in Section 3.3, and analyze the soundness of the protocol in Section 3.4.

As much as possible, we align our notation and terminology with that of [18].

3.2 IOP Definitions

Recall the definition of interactive oracle proof from [7]. In the IOP model, our protocol satisfies the definition of STIK from [4].⁷

Our IOP proves knowledge of a witness that satisfies the RAP[2] (randomized AIR with pre-processing) instance $A = (\mathbb{F}, \mathbb{K}, e, w_{RAP}, n_{\sigma_{mem}}, n_{\sigma_{bytes}}, h, d, s, \omega, l, C_{set})$, where

- $\mathbb{F} = \mathbb{F}_{2^{31}-2^{27}+1}$, which we call the Baby Bear field.
- \mathbb{K} is an extension of \mathbb{F} of size q^e , where $e = 4$.
- w_{RAP} is the number of columns in the RAP witness. We write $w_{RAP} = w_{control} + w_{data} + w_{perm}$ where
 - $w_{control}$ is the number of control columns,
 - w_{data} is the number of data columns, and
 - $w_{perm} = 2e \cdot n_{\sigma_{mem}} + 2e \cdot n_{\sigma_{bytes}}$ is the number of accumulator columns associated with the PLOOKUP argument.⁸
- $n_{\sigma_{mem}}$ is the number of duplicated data columns in the witness that appear due to the permutation from trace_{time} to trace_{mem} .
- $n_{\sigma_{bytes}}$ is the number of duplicated data columns in the witness that appear due to the bytes lookup in our PLOOKUP implementation.
- 2^h is the size of the trace domain (i.e., the length of the columns).

⁷The difference in acronyms between STIK and STARK is a substitution of “IOP” for “ARguement.”

⁸The PLOOKUP argument depends on 4 grand product computations: a pre- and a post- accumulator for the permuted memory columns as well as a pre- and a post- accumulator for the bytes lookup. In our system, $n_{\sigma_{mem}} = 5$ and $n_{\sigma_{bytes}} = 15$

- d is the maximum degree of the rule-checking polynomials.
- ω is the generator of the trace domain.
- I is the set of indices for the tapset⁹.
- C_{set} is the set of constraints which enforce the computational integrity checks. Each constraint, $C_i \in \mathbb{F}^{\leq d}[Y]$ is a multi-variate polynomial over the tapset variables, of total degree at most d , called the i^{th} rule-checking polynomial. Each constraint is enforced over the entire trace domain.
- s is the number of RAP constraints.

In addition to the inputs to the RAP instance listed above, the IOP also uses the following auxiliary inputs, denoted $\text{aux} = (D, k, \text{aux}_{\text{FRI}})$.

- D is the zk commitment domain¹⁰, which is a non-trivial coset¹¹ of a multiplicative subgroup D_0 where $H \subset D_0 \subset \mathbb{F}$.
- 2^k is the size of the zk commitment domain. We define the rate of the IOP by $\rho := \frac{2^h}{2^k}$. (TODO Why does ethSTARK use k' here?)
- aux_{FRI} is defined later (TODO)

3.3 IOP Protocol Specification

1. **Prover** generates IOP witness $w_{\text{control}} \cup w_{\text{data}}$ and sends commitments $\text{Com}(w_{\text{control}})$ and $\text{Com}(w_{\text{data}})$.¹²
2. **Verifier** samples $n_\sigma = n_{\sigma_{\text{mem}}} + n_{\sigma_{\text{bytes}}}$ elements of \mathbb{K} , the randomness used in our permutation argument.
3. **Prover** generates witness w_{accum} and sends $\text{Com}(w_{\text{accum}})$.

Using the definitions and notation from Section 5 of [18], $w_{\text{AIR}} = w_{\text{control}} \cup w_{\text{data}} \cup w_{\text{accum}}$ is a witness that satisfies the RAP instance $A_{\text{RAP}} = (\mathbb{F}, \mathbb{K}, e, w, n_{\sigma_{\text{mem}}}, n_{\sigma_{\text{mem}}}, h, d, s, \omega, I, C_{\text{set}})$.

The rest of the protocol implements DEEP-ALI + Batched FRI as in [18], [8], [5], using auxiliary IOP parameters $\text{aux} = (D, k, \text{aux}_{\text{FRI}})$. Unlike those references, but as in [15] and [19], we use powers of a single verifier randomness for constraint batching and a single verifier randomness for FRI batching.

4. **Verifier** samples $\alpha_{\text{constraints}} \in \mathbb{K}$, the randomness used in DEEP-ALI constraint batching.¹³

⁹A tap is a reference to an entry in the trace; the constraints are expressed as a function of various taps. For more details on taps, see Appendix A.5.2. Note that ethSTARK uses the term *mask* where we use *tapset*.

¹⁰ethSTARK calls this the evaluation domain.

¹¹Using D_0 as a commitment domain would mean queries might reveal information about the execution trace. The coset D doesn't intersect D_0 , which is important for zero-knowledge purposes.

¹²For details on this step, see Appendix C.1.

¹³ethSTARK samples 2 random parameters per constraint.

5. **Prover** generates the validity witness $\mathbf{w}_{\text{validity}}$ and sends $\text{Com}(\mathbf{w}_{\text{validity}})$.¹⁴
6. **Verifier** samples $z \in \mathbb{K} \setminus (\mathcal{H} \cup \mathcal{D})$ the random evaluation point used as a query for DEEP-ALI.
7. **Prover** uses z and \mathbf{w}_{AIR} to construct $\mathbf{w}_{\text{BatchedFRI}}$. Prover sends a DEEP answer sequence consisting of the full tapset of \mathbf{w} and coefficients of each polynomial in $\mathbf{w}_{\text{BatchedFRI}}$.¹⁵
8. **Verifier** samples $\alpha_{\text{FRI}} \in \mathbb{K}$, the randomness used for FRI batching.
9. **Both parties** apply FRI with auxiliary information $\mathbf{aux}_{\text{FRI}}$ to check proximity to the code $\text{RS}[\mathbb{K}, \mathcal{D}, \rho]$ of the function

$$f_{\text{FRI}} = \sum_{i=0}^{|\mathbf{w}_{\text{BatchedFRI}}|-1} \alpha_{\text{FRI}}^i \cdot f_i(x)$$

where f_i are the polynomials of $\mathbf{w}_{\text{BatchedFRI}}$.

3.4 Soundness Analysis in the IOP Model

We present soundness analysis of the IOP protocol using results from [5] and [8]. To translate this IOP analysis into conclusions about our non-interactive protocol, we use the Fiat-Shamir Heuristic, using SHA-2 HMACs as CRNGs.

Our soundness analysis follows that of [18], aside from the following differences:

- **PLOOKUP Argument:** Our protocol begins with a PLOOKUP[11] argument which doesn't have an analog in [18]. We bound the soundness error here using the Schwartz-Zippel Lemma.
- **Constraint Batching:** DEEP-ALI includes a step that compresses all the constraints into a single "Combined Constraint." As in [15], our protocol uses powers of a single random field element whereas [18] uses a vector of field elements. More technically, we use parametric batching rather than affine batching.
- **FRI Batching:** Similarly, the "batched FRI protocol" begins by using verifier randomness to compress a number of FRI instances into a single instance. As in [15], our protocol uses parametric batching for the "FRI batching" step, whereas [18] uses affine batching.
- **Degree reduction of $\mathbf{w}_{\text{validity}}$:** The construction in [4] and [18] results in a pair of FRI assertions, one for the trace $(\mathbf{w}_{\text{control}} \cup \mathbf{w}_{\text{data}} \cup \mathbf{w}_{\text{accum}})$ and one for the "validity polynomial." As in [15], we split¹⁶ the validity polynomial into 4 low-degree validity polynomials so that we can compress this into a single

¹⁴For details on this step, see Section C.5.2

¹⁵For details on this step, see Section C.5.5.

¹⁶The split function is the same as the one used in the FRI commit rounds. For more details on this function, see Appendix A.2.3.

FRI assertion. Each leaf of $\mathbf{w}_{\text{validity}}$ contains one evaluation for each of the 4 low-degree validity polynomials. [15] uses the term “segment polynomials” to refer to this splitting technique.

We bound the soundness of our protocol by

$$\epsilon_{\text{IOP}} \leq \epsilon_{\text{PLOOKUP}} + \epsilon_{\text{DEEP-ALI}} + \epsilon_{\text{FRI}}$$

where $\epsilon_{\text{PLOOKUP}}$, $\epsilon_{\text{DEEP-ALI}}$, and ϵ_{FRI} are the soundness error bounds for our PLOOKUP argument, our DEEP-ALI implementation, and our FRI implementation, respectively. In the following subsections, we bound $\epsilon_{\text{PLOOKUP}}$, $\epsilon_{\text{DEEP-ALI}}$, and ϵ_{FRI} , respectively.

Using θ as the proximity parameter for the FRI low-degree test, the analysis in this section is proven to hold within the list-decoding radius ($\theta < 1 - \sqrt{\rho}$) and conjectured to hold up to code capacity ($\theta < 1 - \rho$). The proofs for these soundness results follow from Theorems 1.5 and 8.3 of [5] and Theorem 6.2 of [8]. The conjectured security follows from Conjecture 8.4 in [5] and Conjecture 2.3 of [8]. We include these theorems and conjectures in Appendix D.

3.4.1 Error Bound on PLOOKUP

There are two grand-product accumulators involved in PLOOKUP; one associated with memory values and one associated with bytes. For each, a simple application of the Schwartz-Zippel Lemma yields a bound on soundness error. We bound the error on the first by $\frac{\epsilon \cdot n \sigma_{\text{mem}} \cdot 2^h}{|\mathbb{K}|}$ and the second by $\frac{\epsilon \cdot n \sigma_{\text{bytes}} \cdot 2^h}{|\mathbb{K}|}$.

3.4.2 Error Bound on DEEP-ALI

In our implementation of DEEP-ALI, the verifier samples a single randomness $\alpha_{\text{constraints}}$ for combining constraints and then $z \in \mathbb{K} \setminus (\mathcal{H} \cup \mathcal{D})$ as a DEEP query. Each of these random samplings has an associated error term. We write $\epsilon_{\text{DEEP-ALI}} = \epsilon_{\text{ALI}} + \epsilon_{\text{DEEP}}$, where ϵ_{ALI} is the soundness error associated with combining constraints and ϵ_{DEEP} is the soundness error associated with the DEEP query point.

We differ from [18] here in that we use powers of a single randomness for combining constraints. Our results here align with [15]; we find $\epsilon_{\text{ALI}} = \frac{s_L}{|\mathbb{K}|}$ and $\epsilon_{\text{DEEP}} = \text{TODO}$

3.4.3 Error Bound on FRI

The major results for the soundness of FRI in the list-decoding regime are Theorems 1.5 and 8.3 from [5]. As above, our soundness results for this part differ from the presentation in [5] and [18] in that we use powers of a single random field element for FRI batching. Again, our results here align with [15]; we find

$$\epsilon_{\text{FRI}} = \left(L - \frac{1}{2}\right) \cdot \frac{(m + \frac{1}{2})^7}{3\sqrt{\rho}^3} \cdot \frac{|D_0|^2}{|F|} + \frac{(2m + 1) \cdot (|D_0| + 1) \cdot \sum_{i=1}^r a_i}{\sqrt{\rho} \cdot |F|} + (1 - \theta)^s$$

References

- [1] Sanjeev Arora and Shmuel Safra. Probabilistic checking of proofs: A new characterization of np. *J. ACM*, 45(1):70–122, jan 1998.
- [2] Aztec. From airs to raps - how plonk-style arithmetization works, 2020.
- [3] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast reed-solomon interactive oracle proofs of proximity. *Electron. Colloquium Comput. Complex.*, TR17, 2017.
- [4] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *IACR Cryptol. ePrint Arch.*, page 46, 2018.
- [5] Eli Ben-Sasson, Dan Carmon, Yuval Ishai, Swastik Kopparty, and Shubhangi Saraf. Proximity gaps for reed-solomon codes. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 900–909, 2020.
- [6] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474, 2014.
- [7] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. *Cryptology ePrint Archive*, Paper 2016/116, 2016. <https://eprint.iacr.org/2016/116>.
- [8] Eli Ben-Sasson, Lior Goldberg, Swastik Kopparty, and Shubhangi Saraf. Deep-fri: Sampling outside the box improves soundness, 2019.
- [9] Alessandro Chiesa and Eylon Yogev. Subquadratic snargs in the random oracle model. *Cryptology ePrint Archive*, Paper 2021/281, 2021. <https://eprint.iacr.org/2021/281>.
- [10] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Proceedings on Advances in Cryptology—CRYPTO ’86*, page 186–194, Berlin, Heidelberg, 1987. Springer-Verlag.
- [11] Ariel Gabizon and Zachary J. Williamson. plookup: A simplified polynomial protocol for lookup tables. *Cryptology ePrint Archive*, Paper 2020/315, 2020. <https://eprint.iacr.org/2020/315>.
- [12] 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.
- [13] Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo – a turing-complete stark-friendly cpu architecture. *Cryptology ePrint Archive*, Paper 2021/1063, 2021. <https://eprint.iacr.org/2021/1063>.

- [14] S Goldwasser, S Micali, and C Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, pages 291–304, New York, NY, USA, 1985. Association for Computing Machinery.
- [15] Ulrich Haböck. A summary on the fri low degree test. Cryptology ePrint Archive, Paper 2022/1216, 2022. <https://eprint.iacr.org/2022/1216>.
- [16] Xavier Leroy. A formally verified compiler back-end. *CoRR*, abs/0902.2137, 2009.
- [17] Irving S. Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of The Society for Industrial and Applied Mathematics*, 8:300–304, 1960.
- [18] StarkWare. ethstark documentation. Cryptology ePrint Archive, Paper 2021/582, 2021. <https://eprint.iacr.org/2021/582>.
- [19] Polygon Zero Team. Plonky2: Fast recursive arguments with plonk and fri. *Bitcoin.-URL: https://bitcoin.org/bitcoin.pdf*, Draft, Sept 7, 2022.

Appendices

A Preliminaries

A.1 Cyclic, Sequential Indexing

In this section, we define the indexing for an execution trace. RISC Zero’s architecture is finite-field based, and the indexing is based on powers of $\omega \in \mathbb{F}_q$.

In order to index a sequence of l elements of \mathbb{F}_q , we choose an element $\omega \in \mathbb{F}_q$ with multiplicative order l and use the sequence $\{\omega^0, \omega^1, \dots, \omega^{l-1}\}$ as an indexing set. This indexing is cyclical, in the sense that $\omega^z = \omega^{z-l}$ for any integer z . We formalize this indexing in the remainder of this section.

Let \mathbb{F}_q be the finite field of order q . In our current system, $q = 2^{31} - 2^{27} + 1$.

Definition 1 (Domain). We call any non-empty subset $\mathcal{D} \subseteq \mathbb{F}_q$ a **domain**.

Definition 2 (Cyclic domain). Given $\omega \in \mathbb{F}_q$, the set $\mathcal{D}(\omega) = \{w^i : i \in \mathbb{N}\}$ is a domain. If $\mathcal{D} = \mathcal{D}(\omega)$ for some non-zero $\omega \in \mathbb{F}_q$, then \mathcal{D} is said to be a **cyclic domain** with **counter** ω .

Definition 3 (Cycle). Given a cyclic domain $\mathcal{D}(\omega)$, we call c a **cycle** if $c \in \mathcal{D}(\omega)$.

We note that $|\mathcal{D}(\omega)|$ is equal to the multiplicative order of ω and therefore divides $q - 1$.

Definition 4 (Cyclic Indexing). Let $\mathcal{D}(\omega) \subset \mathbb{F}_q$ be the cyclic domain with counter ω . We use the powers of ω to define an indexing for $\mathcal{D}(\omega)$:

$$\begin{aligned} \text{Index-Cycle} &: \mathbb{N} \longrightarrow \mathcal{D}(\omega) \\ \text{Index-Cycle} &: n \longmapsto \omega^n \end{aligned}$$

This cyclic indexing on $\mathcal{D}(\omega)$ serves as our notion of sequential time throughout the protocol. We write **the k^{th} cycle** or **the cycle at clocktime k** to mean ω^k . We’ll also use sequential language like **next cycle** and **previous cycle** based on this indexing, which we define as follows:

$$\begin{aligned} \text{Next-Cycle} &: \mathcal{D}(\omega) \longrightarrow \mathcal{D}(\omega) \\ \text{Next-Cycle} &: c \longmapsto \omega c \end{aligned}$$

$$\begin{aligned} \text{Prev-Cycle} &: \mathcal{D}(\omega) \longrightarrow \mathcal{D}(\omega) \\ \text{Prev-Cycle} &: c \longmapsto \omega^{-1}c \end{aligned}$$

We emphasize the multiplicative nature of this indexing: we can express the relationship between the j th cycle, c_j and the $(j+1)^{\text{th}}$ cycle, c_{j+1} by writing $c_{j+1} = \omega c_j$.

A.2 Blocks: Indexing, Metrics, and Operations

In this section, we articulate the mathematical structure of blocks, including a sequential indexing, two metrics, and a number of algebraic operations on blocks.

A.2.1 The Structure of a Block

A **block** \mathbf{u} over a cyclic domain $\mathcal{D}(\omega)$ is a function¹⁷ $\mathcal{D}(\omega) \rightarrow \mathbb{F}_q$. We define the i^{th} entry of \mathbf{u} , denoted $\mathbf{u}[i]$ as follows:

$$\mathbf{u}[i] = \mathbf{u}(\omega^i)$$

Given $\mathbf{u}[n]$, we'll use sequential language like **next entry**, $\mathbf{u}[n + 1]$ and **previous entry**, $\mathbf{u}[n - 1]$. The **length** of \mathbf{u} is the number of entries of \mathbf{u} , which is equal to the multiplicative order of ω .

Example

The entry of \mathbf{u} that appears n cycles before $\mathbf{u}(c)$ can be written as $\mathbf{u}(w^{-n}c)$.

A.2.2 Block Distance and Divergence

Let $\mathcal{D}(\omega) \subset \mathbb{F}_q$ be the cyclic domain with counter ω , and let \mathbf{u}, \mathbf{v} be blocks indexed by ω . We define the **distance** between blocks (aka **Hamming Distance**) to be the number of entries that differ between \mathbf{u} and \mathbf{v} . Formally,

$$\delta(\mathbf{u}, \mathbf{v}) = |\{\omega^n \in \mathcal{D}(\omega) : \mathbf{u}(\omega^n) \neq \mathbf{v}(\omega^n)\}|$$

We say that \mathbf{u} is δ -close to \mathbf{v} if the distance from \mathbf{u} to \mathbf{v} is less than or equal to δ .

We define the **divergence** between \mathbf{u} and \mathbf{v} to be the proportion of entries that differ.

$$\Delta(\mathbf{u}, \mathbf{v}) = \frac{\delta(\mathbf{u}, \mathbf{v})}{|\mathcal{D}(\omega)|}$$

We note that for any **divergence** and **distance** are both metrics over the set of valid blocks.

Defining Divergence of a Block and a Set

Let $\mathcal{D}(\omega) \subset \mathbb{F}_q$ be the cyclic domain with counter ω . Let \mathbf{u} be a block indexed by ω , and let V be a set of blocks indexed by ω . We define the **distance** between \mathbf{u} and V to be the distance between \mathbf{u} and the closest block of V .

$$\delta(\mathbf{u}, V) = \min_{\mathbf{v} \in V} \delta(\mathbf{u}, \mathbf{v})$$

We define the **divergence** of \mathbf{u} and V as follows:

$$\Delta(\mathbf{u}, V) = \frac{\delta(\mathbf{u}, V)}{|\mathcal{D}(\omega)|}$$

¹⁷Intuitively, a “block” is just a sequence of elements of \mathbb{F}_q indexed by powers of ω .

A.2.3 Operations on Blocks

Blocks are vectors over \mathbb{F}_q , and we define addition and scalar multiplication in the familiar way.

Addition of Blocks

Given blocks \mathbf{u} and \mathbf{v} , we define the block $\mathbf{u} + \mathbf{v}$:

$$\begin{aligned} (\mathbf{u} + \mathbf{v}) : \mathcal{D}(\omega) &\longrightarrow \mathbb{F}_q \\ (\mathbf{u} + \mathbf{v}) : \quad x &\longmapsto (\mathbf{u}(x) + \mathbf{v}(x)) \end{aligned}$$

Scalar Multiplication of Blocks

Given a block \mathbf{u} and $\alpha \in \mathbb{F}_q$, we define the block $\alpha \cdot \mathbf{u}$:

$$\begin{aligned} \alpha \cdot \mathbf{u} : \mathcal{D}(\omega) &\longrightarrow \mathbb{F}_q \\ \alpha \cdot \mathbf{u} : \quad x &\longmapsto \alpha \cdot \mathbf{u}(x) \end{aligned}$$

With these definitions, we see that the set of blocks indexed by ω forms a vector space over \mathbb{F}_q .

Mixing Blocks We define a method of mixing n blocks into one, using a mixing parameter α . This method is used to mix the constraint polynomials, to mix the DEEP polynomials, and throughout the FRI protocol.

Let $D(\omega) \subset \mathbb{F}_q$ be a cyclic domain. Let $\mathbf{U} = \mathbf{u}_0, \dots, \mathbf{u}_{m-1}$ be a sequence of blocks indexed by ω , and let $\alpha \in \mathbb{F}_q$ be a **mixing parameter**. We define the **mix** of \mathbf{U} by α as

$$\text{mix}(\mathbf{U}, \alpha) = \alpha^0 \mathbf{u}_0 + \alpha^1 \mathbf{u}_1 + \dots + \alpha^{m-1} \mathbf{u}_{m-1}$$

We note that a mix of \mathbf{U} is a linear combination of \mathbf{u}_i .

NTTs and iNTTs An **iNTT** converts an array of evaluations over a domain \mathcal{D} to an array of coefficients of the minimal-degree interpolating polynomial. Given a block \mathbf{u} over cyclic domain $\mathcal{D}(\omega)$, we write $f_{\mathbf{u}} = \text{iNTT}(\mathbf{u}, \mathcal{D})$.

An **NTT** converts an array of coefficients to an array of evaluations over a domain \mathcal{D} . Given a polynomial f , we write $\mathbf{u}_f = \text{NTT}(f, \mathcal{D})$.

Splitting Blocks We define a method of splitting 1 block of length bl into b blocks of length l . This method is used as part of the recursive step in the FRI Protocol and as a means of reducing the degree of the validity polynomial.

Given:

1. a block \mathbf{u} over $\mathcal{D}^{(0)} = \beta \mathcal{D}(\omega)$, where ω has order bl , and
2. a subset $\mathcal{D}^{(1)} \subset \mathcal{D}^{(0)}$ where $\frac{|\mathcal{D}^{(0)}|}{|\mathcal{D}^{(1)}|} = b$,

we construct $b\text{-split}(\mathbf{u}) = \mathbf{v}_0, \mathbf{v}_1, \dots, \mathbf{v}_{b-1}$, as follows:

1. Compute $P_{\mathbf{u}_i} = \text{iNTT}(\mathbf{u}_i, \mathcal{D}^{(0)})$

- Sort the coefficients of $P_{\mathbf{u}_i}$ to construct $P_{\mathbf{v}_0}, \dots, P_{\mathbf{v}_{b-1}}$, where

$$P_{\mathbf{u}}(x) = \sum_{i=0}^{b-1} x^i P_{\mathbf{v}_i}(x^b)$$

Succinctly, the coefficient of the degree j term in $P_{\mathbf{v}_i}$ is the $(bj+i)^{\text{th}}$ coefficient of $P_{\mathbf{u}}$.

- Compute each $\mathbf{v}_i = \text{NTT}(P_{\mathbf{v}_i}, \mathcal{D}^{(1)})$

A.3 Reed Solomon Codes

In this section, we define Reed Solomon codes and outline their place in the RISC Zero proof system.

Put succinctly, a Reed Solomon Code \mathcal{R} encodes a message¹⁸ \mathbf{m} into a block \mathbf{u} by associating \mathbf{m} with a polynomial $f_{\mathbf{u}} = \text{iNTT}(\mathbf{m}, \mathcal{D})$, and evaluating $f_{\mathbf{u}}$ over a cyclic domain. In RISC Zero’s applications, we construct f_u using an iNTT and evaluate $f_{\mathbf{u}}$ using an NTT.

Let $\omega \in \mathbb{F}_q$ have multiplicative order bl and let $\mathcal{D}(\omega^b) \subseteq \mathcal{D}(\omega) \subseteq \mathbb{F}_q$. Let $\mathcal{B}(\omega^b)$ be the set of all blocks over $\mathcal{D}(\omega^b)$ and $\mathcal{B}(\omega)$ be the set of all blocks over $\mathcal{D}(\omega)$.

We define a Reed-Solomon Encoding, \mathcal{R} , where

$$\begin{aligned} \mathcal{R} : \mathcal{B}(\omega^b) &\rightarrow \mathcal{B}(\omega) \\ \mathcal{R} : u &\mapsto \bar{u} \end{aligned}$$

where the i^{th} entry of \bar{u} is $P_{\mathbf{u}}(\omega^i)$ and $P_{\mathbf{u}}(x)$ is the unique polynomial of degree $l - 1$ that agrees with \mathbf{u} on $\mathcal{D}(\omega^b)$. We say \mathcal{R} has **rate** $\frac{1}{b}$.

Given $\mathcal{R} : \mathcal{B}(\omega^b) \rightarrow \mathcal{B}(\omega)$, we say a block $\mathbf{v} \in \mathcal{B}(\omega)$ is **valid** if there exists $u \in \mathcal{B}(\omega^b)$ such that $\mathcal{R}(u) = v$. We write $\mathcal{R}(\omega)$ to refer to the set of blocks over $\mathcal{D}(\omega)$ that are valid with respect to \mathcal{R} .

¹⁸Intuitively, you can think of a message as a column of the execution trace and a block as the evaluation of the associated Trace polynomial. By constructing a Reed-Solomon “block,” we create a mechanism for error amplification. Since two degree n polynomial will agree at no more than n locations, a single discrepancy before RS-encoding will result in a discrepancy at $1 - \frac{n}{l}$ locations after RS-encoding. This means that the Verifier’s probabilistic queries are very likely to check a discrepancy in the trace.

Note also that a column is formally a block – the data structures for a column/message/block in RISC Zero are all equivalent. RISC Zero adds random padding to the trace blocks before implementing Reed-Solomon encoding.

A.4 Block Operations on RS Codes

In this section, we present various results about block operations in relation to block validity and block proximity¹⁹.

A.4.1 Motivation

Fundamentally, most of the protocols for the proof system described here work by operating on Reed-Solomon codes, combining and transforming them in various ways.

This transformations in turn modify the polynomials they RS codes represent. But because of the redundancy introduced by the error correcting code, once the prover cryptographically commits to the codes before and after an operations in some way, we can verify the correctness of the operation by randomly spot checking.

Generally, we will say that if the result of the operations is a (mostly) correct RS code, and the spot checks success with high likelihood, that input was (very likely) a (mostly) correct RS code as well, and that the operation was done correctly across the whole code, in the sense that if we corrected both codes, we would find the entire process was error free. Of course the details here matter a great deal, as do all of the precise probabilities and divergences.

We often care that the divergence of a given RS code from the set of valid code words is below some critical distance δ . We say such a code is δ -close.

In the next sub-sections we will introduce the the core building blocks we will use for the protocols, and provide proofs of their correctness (directly or via reference).

In all cases below, we presume δ is within the unique decode range. The key concepts are (informally)

Mixing If we randomly mix RS codes, and the output is δ close and, it is very likely that all of inputs were also δ close.

Splitting We can split RS-code of rate ρ into b smaller RS-codes of rate ρ , each of size $1/n$, and can evaluate the split is correct by spot checking, so long as the outputs are all δ close.

Evaluating We can verify the evaluation of the message polynomial of an RS-code at an arbitrary field element by computing another RS-code, and then spot checking them for a certain relationship (so long as both are δ close).

One can immediately see that by applying splitting and mixing recursively, one could check an arbitrarily large RS-code was very likely δ close in $O(\log(N))$ steps, N being the length of the code. This is in fact how the FRI protocol works.

¹⁹Proximity in this context refers to the distance between a given block and the nearest valid block.

A.4.2 Operations on Valid Blocks

For proving completeness of the protocol, we need to show that if we start with valid blocks and perform algebraic block operations, the results will also be valid blocks. In this section, we show that the set of valid blocks, $\mathcal{R}(\omega)$ is closed under block addition and scalar multiplication, mixing, and splitting.

Let \mathbf{u}, \mathbf{v} be valid blocks and let $\alpha \in \mathbb{F}_q$. Since \mathbf{u} and \mathbf{v} are valid blocks, we can associate each of them with a low-degree polynomial: $P_{\mathbf{u}}$ and $P_{\mathbf{v}}$.

Addition

Block addition is defined as pointwise addition, so $\mathbf{u} + \mathbf{v}$ can be expressed as $P_{\mathbf{u}+\mathbf{v}}$ where $P_{\mathbf{u}+\mathbf{v}}$ is the polynomial formed via point-wise addition of $P_{\mathbf{u}}$ and $P_{\mathbf{v}}$.

$$P_{\mathbf{u}+\mathbf{v}}(x) = P_{\mathbf{u}}(x) + P_{\mathbf{v}}(x)$$

Given that \mathbf{u} and \mathbf{v} are valid, we conclude that $\mathbf{u} + \mathbf{v}$ is valid since $\deg(P_{\mathbf{u}+\mathbf{v}}) \leq \max(\deg(P_{\mathbf{u}}), \deg(P_{\mathbf{v}}))$.

Scalar Multiplication

Similarly, $\alpha\mathbf{u}$ can be expressed as $P_{\alpha\mathbf{u}}$, where

$$P_{\alpha\mathbf{u}}(x) = \alpha P_{\mathbf{u}}(x)$$

Given that \mathbf{u} is valid, we conclude that $\alpha\mathbf{u}$ is valid since $\deg(P_{\alpha\mathbf{u}}) = \deg(P_{\mathbf{u}})$.

Mix

It follows immediately that the **mix** of valid blocks is also valid, since **mix** is just a linear combination of blocks.

Split

It is similarly immediate that if \mathbf{u} is a valid codeword on $\mathcal{D}(\omega)$, then each \mathbf{v}_i in **b-split**(\mathbf{u}) is a valid codeword on $\mathcal{D}(\omega^b)$. This follows from the definition of **split**, since each \mathbf{v}_i is constructed as the evaluation of a low-degree polynomial $P_{\mathbf{v}_i}$.

A.4.3 Operations and Block Proximity

Throughout the protocol, the Prover has been using the **mix** function in order to consolidate multiple arguments about **block proximity** to a single argument about block proximity. For proving soundness of the protocol, we need to show that if the block associated with the final FRI polynomial f_r is δ -close to a valid block, then the original trace blocks and validity blocks are also δ -close to a valid block.

The major theorem we rely on for this argument is presented in Theorem 1.5 of Eli Ben-Sasson et al, 2020. Put succinctly, the premise is that if \mathbf{U} is a sequence of blocks and some **mix** of \mathbf{U} gives a result that is δ -close to $\mathcal{R}(\omega)$, then it's extremely likely that each of the blocks of \mathbf{U} are also δ -close to $\mathcal{R}(\omega)$. Moreover, the δ -closeness of the **mix**(\mathbf{U}) allows us to conclude that the locations-of-agreement in the pre-mixed blocks are **correlated**.

This correlation is non-trivial and quite useful: typically if \mathbf{a} and \mathbf{b} are blocks that are each δ -close to $\mathcal{R}(\omega)$, $\mathbf{a} + \mathbf{b}$ will not be δ -close. But if \mathbf{a} and \mathbf{b} have correlated agreement, we can conclude that:

$$\delta(\mathbf{a} + \mathbf{b}, \mathcal{R}(\omega)) \leq \max[\delta(\mathbf{a}, \mathcal{R}(\omega)), \delta(\mathbf{b}, \mathcal{R}(\omega))]$$

Without correlated agreement, we'd have the weaker

$$\delta(\mathbf{a} + \mathbf{b}, \mathcal{R}(\omega)) \leq \delta(\mathbf{a}, \mathcal{R}(\omega)) + \delta(\mathbf{b}, \mathcal{R}(\omega))$$

Correlated Agreement of Mixed Codes

The following result allows the Verifier to conclude that the proximity of \mathbf{u} to \mathcal{V} implies the proximity of each \mathbf{u}_i as well. We now present the result:

Theorem 1 (Correlated agreement over parameterized curves).

Let $\omega \in \mathbb{F}_q$ have multiplicative order bl , and let $\mathcal{R} : \mathcal{D}(\omega^b) \rightarrow \mathcal{D}(\omega)$ be a Reed-Solomon encoding. Let \mathcal{V} be the collection of valid blocks of \mathcal{R} .
 Let $\mathbf{U} = \mathbf{u}_0, \mathbf{u}_1, \dots, \mathbf{u}_{m-1}$ be a sequence of blocks over $\mathcal{D}(\omega^b)$.
 Let δ be a distance less than unique decoding radius: $\delta \in [0, \frac{1-\rho}{2})$
 Let ϵ be a probability defined as $\epsilon = \frac{mbl}{q}$

If $\Pr_{\alpha \in \mathbb{F}_q}[\Delta(\text{mix}(\mathbf{U}, \alpha), \mathcal{V}) \leq \delta] > \epsilon$, then there exists a $\mathcal{D}' \subset \mathcal{D}(\omega)$ and $\mathbf{v}_0, \mathbf{v}_1, \dots, \mathbf{v}_n \in V$ satisfying

Density: $|\mathcal{D}'|/|\mathcal{D}(\omega)| \geq 1 - \delta$

Agreement: For all $i \in \{0, 1, \dots, l\}$, and $x \in \mathcal{D}'$

$$u_i(x) = v_i(x)$$

Restated informally, Theorem 1 basically says: If we randomly mix together some set of blocks, and the result is δ -close to a valid code more than a small amount ϵ of the time, then there is a large subset of \mathcal{D} in which the elements are in fact all parts of valid code. Of course, this means that in fact the original blocks we mixed together are in fact also δ close. We capture this simplified understanding below

Corollary 1 (Mixing). *Let \mathcal{V} , \mathbf{u} , δ and ϵ be as defined in Theorem 1.*

If $\Pr_{\alpha \in \mathbb{F}_q}[\text{mix}(\mathbf{u}, \alpha) \text{ is } \delta\text{-close}] > \epsilon$, then all $u \in \mathbf{u}$ are also δ -close.

Proof. By Theorem 1, there exists a set \mathcal{D}' such that for each \mathbf{u}_i , $\mathbf{u}_i(x) = \mathbf{v}_i(x)$ for all x in \mathcal{D}' , and $\mathbf{v}_i \in V$. Since \mathbf{u}_i and \mathbf{v}_i agree on all points of \mathcal{D}' , and $|\mathcal{D}'|/|\mathcal{D}| \geq 1 - \delta$, we have $\Delta(\mathbf{u}_i, \mathcal{V}) \leq \delta$, and thus $\Delta(\mathbf{u}_i, \mathcal{V}) \leq \delta$. \square

Split and δ -closeness

Haven't thought about this yet.

Evaluate and δ -closeness

Copying and pasting from zkstark.tex; TODO: clean-up/integrate/citation [TODO: Words go here]

Definition 5. Let

- \mathcal{D} be a domain over field \mathbb{F}_q
- \mathbf{u} be a codeword over \mathcal{D}
- $e = ((x_1, y_1), \dots, (x_m, y_m))$ be a sequence of m pairs $(\mathbb{F}_q \times \mathbb{F}_q)$, with all x_i unique.
- $b = \text{interpolate}(e)$

We define a function $\text{eval} : (\mathbb{F}_q^{\mathcal{D}}, (\mathbb{F}_q \times \mathbb{F}_q)^n) \rightarrow \mathbb{F}_q^{\mathcal{D}}$

$$\mathbf{u}' = \text{eval}(u, e)$$

where

$$\mathbf{u}'(z) = \frac{u(z) - b(z)}{\prod_{i=1}^m (z - x_i)}$$

[TODO: More words go here]

Theorem 2 (Reed-Solomon unique decoding). *Given an RS code, $\text{RS}[\mathbb{F}_q, \mathcal{D}, k]$ of block length n , rate ρ , and a block $\mathbf{u} : \mathcal{D} \rightarrow \mathbb{F}_q$ if*

$$\Delta(\mathbf{u}, V) < (1 - \rho)/2$$

then there exists a unique closest $\mathbf{u}' \in V$, $\Delta(\mathbf{u}, \mathbf{u}') = \Delta(\mathbf{u}, V)$, and such \mathbf{u}' can be found in polynomial time.

Theorem 3. *Let*

- $V = \text{RS}[\mathbb{F}_q, \mathcal{D}, k]$, be an RS code with rate ρ and block size n .
- \mathbf{u} and \mathbf{u}' be blocks in $\mathbb{F}_q^{\mathcal{D}}$ which are δ -close to V , $\delta < \frac{1-\rho}{2}$
- $e = ((x_1, y_1), \dots, (x_m, y_m))$ be a sequence of m pairs $(\mathbb{F}_q \times \mathbb{F}_q)$, with all x_i unique.
- $b = \text{interpolate}(e)$

If

$$\Pr_{z \in \mathcal{D}} [u(z) = \mathbf{u}'(z) \prod_{i=1}^m (z - x_i) + b(z)] \geq \frac{k+m}{n} \quad (1)$$

Then

1. *The polynomial u can be corrected uniquely to some $v \in V$*
2. *For all $0 < i < n$, $y_i = v(x_i)$*

Proof. Claim 1 follows directly from the requirements on \mathbf{u} and theorem 2. Given the requirement of \mathbf{u}' , it also has a unique decoding via theorem 2, which we call \mathbf{v}' .

If $\mathbf{v} = \mathbf{v}' \prod(z - x_i) + b$, then it is clear that \mathbf{v} meets the requirements of claim 2, since $\mathbf{v}' \prod(z - x_i)$ will be 0 at all x_i , and thus adding \mathbf{b} will thus satisfy claim 2.

On the other hand, if $\mathbf{v} \neq \mathbf{v}' \prod(z - x_i) + b$, then the difference $d = \mathbf{v} - (\mathbf{v}' \prod(z - x_i) + b)$ is a non-zero polynomial of degree at most $k + n - 1$. Thus it can have at most $k + m - 1$ zeros. But this would contradict equation 1, since it implies at least $k + m$ zeros over the n element of \mathcal{D} . \square

RS Codes over Subfields

Recently copy-pasted from zkstark.tex. TODO: Figure out exactly how this fits into the picture. DEEP-ALI only?

One important performance enhancement used in Risc0's protocol involves the use of a subfield \mathbb{F}_p of a larger field $\mathbb{F}_q = \mathbb{F}_{p^n}$ for some of the protocol. The idea is that many of the RS codes will use the smaller subfield to reduce the computational burden, but will be interpreted later as RS codes in the full field. Of course it is immediately clear that for $\omega \in \mathbb{F}_p$, $\text{RS}[\mathbb{F}_p, \mathcal{D}(w), k] \subset \text{RS}[\mathbb{F}_{p^n}, \mathcal{D}(w), k]$. That is, all elements of the RS code of the small field are elements of the RS code of the larger field. However, we also want to prove that any block from the larger field's code (or even a block close to a valid code), for which *most* of the elements are in \mathbb{F}_p , is in fact close to the smaller field's code.

Formally, we have:

Theorem 4 (RS Subfield Theorem). *Let*

- $V_p = \text{RS}[\mathbb{F}_p, \mathcal{D}, k]$, be an RS code with rate ρ and block size n .
- $q = p^m$
- $V_q = \text{RS}[\mathbb{F}_q, \mathcal{D}, k]$, be an RS code with rate ρ and block size n .
- $u \in \mathbb{F}_q^{\mathcal{D}}$ be a block over which is δ -close to V_q , $\delta < \frac{1-\rho}{2}$
- $\mathbf{u}' \in \mathbb{F}_p^{\mathcal{D}}$ be the block which a projection of \mathbf{u} into \mathbb{F}_p , via the rule that: $\mathbf{u}'(x) = \mathbf{u}(x)$ if $\mathbf{u}(x) \in \mathbb{F}_p$, and $\mathbf{u}'(x) = 0$ otherwise.

If

$$\Pr_{x \in \mathcal{D}} [\mathbf{u}(x) \notin \mathbb{F}_p] < \delta \quad (2)$$

Then \mathbf{u}' is δ close to V_p .

Proof. Given that fact the \mathbf{u} is within the unique decoding radius, there is some unique decoding $v \in V_q$, such that $\Delta(v, u) < \delta$. Therefore at most δ fraction of the elements are not in \mathbf{v} , and we also have via 2 that at most δ fractions of elements are not in \mathbb{F}_p . Thus at least $b > 1 - 2\delta$ elements must be both in \mathbf{v} and in \mathbb{F}_p .

Since $\delta < \frac{1-\rho}{2}$, we have $b > \rho$, or counting the number of elements rather than the fraction $B = bn > \rho n > k$. Thus there are k elements in \mathbf{v} which are in \mathbb{F}_p . But via interpolation over the field \mathbb{F}_p , we can find coefficients in \mathbb{F}_p , and therefore all values in \mathbf{v} are in \mathbb{F}_p , and thus $v \in V_p$, and thus \mathbf{u}' is δ close to V_p . \square

A.5 Constraints and Taps

A.5.1 Constraints

Checking that an execution trace satisfies a particular RISC-V rule involves checking some arithmetic relationship involving multiple blocks across multiple clock cycles.

Given a particular rule of RISC-V that we want to check at clock cycle c , we can plug the associated taps into some equation that will return zero if the trace is

valid.²⁰.

In other words, we can write a rule-checking function, \mathbf{r} , whose inputs are taps of the trace, such that when \mathbf{T} is a valid trace over cyclic domain $\mathcal{D}(\omega)$, $\mathbf{r}(t_1(c), \dots, t_k(c)) = 0$ for all $c \in \mathcal{D}(\omega)$.

This gives a construction of a single-input constraint function C :

$$\begin{aligned} C : \mathcal{D}(\omega) &\rightarrow \mathbb{F}_q \\ C : c &\mapsto \mathbf{r}(t_1(c), \dots, t_k(c)) \end{aligned}$$

By writing the taps in terms of trace polynomials, the Prover can express C as a polynomial. Putting this all together, we construct **constraint polynomials**²¹, C , as follows:

$$C(c) = \mathbf{r}(\bar{t}_1, \dots, \bar{t}_n)$$

and each \bar{t}_i is a polynomial tap of the trace. In practice, we have one constraint polynomial, \mathbf{c}_i , per rule-checking function, \mathbf{r}_i .

A.5.2 Trace Taps and Polynomial Taps

Trace taps offer a way to express references across multiple blocks and across multiple cycles. Concretely, the **tap** (i, j) at c is the entry in \mathbf{u}_i that appears j clock cycles after c . Concretely, the tap $(1, 2)$ at ω^3 is $\mathbf{u}_1(\omega^{(3+2)})$.

Polynomial taps $\bar{t}_i = (a_i, b_i)$ are a natural extension taps. Whereas a tap points to an entry in a trace, a **polynomial tap** points to an evaluation of a trace polynomial. Formally, the polynomial tap $\bar{t}_i = (a_i, b_i)$ is defined as follows:

$$\bar{t}_i(c) = P_{a_i}(w^{b_i} c)$$

where P_{a_i} is the trace polynomial specified by the i^{th} tap.

Formal Definitions

Given a trace $\mathbf{T} = \mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_n$ over cyclic domain $\mathcal{D}(\omega)$, we define a **tap** $t = (i, j)$ as a function

$$\begin{aligned} t : \mathcal{D}(\omega) &\rightarrow \mathbb{F}_q \\ t : c &\mapsto \mathbf{u}_i(\omega^j c) \end{aligned}$$

²⁰The control blocks allow for a construction of rule-checking functions that are time-symmetric: by using control blocks to turn on and off the enforcement of the rules, we construct rule-checking functions that evaluate to 0 at each cycle associated with a valid trace.

²¹The key feature of the Constraint Polynomials is that they have roots at each $c \in \mathcal{D}(\omega^4)$.

Given a collection of Trace Polynomials P_1, P_2, \dots, P_n , we define a **polynomial tap** $\bar{t} = (i, j)$ as a function

$$\begin{aligned}\bar{t} : \mathcal{D}(\omega) &\rightarrow \mathbb{F}_q \\ \bar{t} : c &\mapsto P_i(\omega^j c)\end{aligned}$$

B Merkle Tree Structure

In this section, we describe the structure of the Merkle commitments and Merkle proofs.

Merkle Leaves and Trees

Throughout the protocol, each Merkle leaf is a vector in \mathbb{F}_{TODO} , constructed by evaluating a sequence of polynomials at a single point in $\mathcal{D}(\text{TODO})$.

These polynomials are organized into **polyGroups**, where one Merkle tree is constructed per polyGroup.

The Prover generates a Merkle root associated with each of the following polyGroups:

1. Control polyGroup
2. Data polyGroup
3. accum polyGroup
4. Validity polyGroup
5. FRI polyGroup (1 per round of FRI)

Merkle Branch Structure

Given that we're doing lots of queries from the same Merkle Tree, these queries are expected to have quite a bit of overlap close to the root. To minimize computation, rather than checking all the way to the root, we choose a cut-off point called `top_layer`.

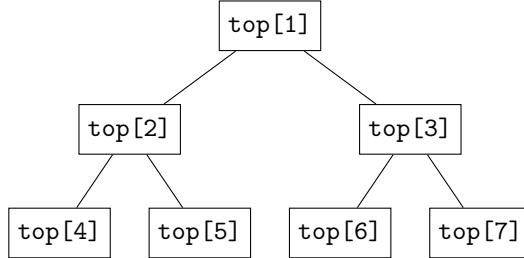
We store that entire layer, and for each query, we check our Merkle branches from `leaf` to `top_layer`. The paths from `top_layer` to the Merkle root only need to be checked once.

More concretely, we initialize a vector `top` that will hold the top of the Merkle Tree. We initialize a vector `top` of size $2 * \text{top_size}$.²² We populate the second half of this vector with digests from `iop.read-digests`. Then, for `i` in $(1.. \text{top_size})$, in reverse order, we compute

²²You can think of `top-size` as the size of `top-layer`.

```
top[i] = hash(top[2*i], top[2*i +1]).
```

We end with the top of the Merkle tree stored in `top`, with the `top[0]` untouched and the Merkle Root in `top[1]`.



Now, to check the validity of a branch that passes through `top[4]`, this optimization allows us to check the root matches `top[1]` and that the branch from `leaf` to `top[4]` is valid.

C Protocol Details

C.1 Constructing $\mathbf{w}_{\text{control}}$ and \mathbf{w}_{data}

As in [4], the trace columns are encoded into trace blocks using Reed-Solomon encoding [17]. $\mathbf{w}_{\text{control}}$ and \mathbf{w}_{data} correspond to the control blocks and data blocks, respectively, as explained in Section 2.1.

The trace columns (\mathbf{m}_i) are encoded as **trace blocks**²³ (\mathbf{u}_i) as follows:

1. Each (padded) column, \mathbf{m}_i is treated as a Reed-Solomon[17] “message.” Each message is used to construct a low-degree polynomial, $P_{\mathbf{u}_i}(x)$ using an iNTT (over $\mathcal{D}(\omega^4)$, as defined in Appendix A.1).
2. The resulting polynomial $P_{\mathbf{u}_i}(x)$ is then evaluated over $\beta\mathcal{D}(\omega)$ using an NTT, where $\beta\mathcal{D}(\omega)$ is a multiplicative coset²⁴ of $\mathcal{D}(\omega)$:

$$\beta\mathcal{D}(\omega) = \{\beta x \in \mathbb{F}_q : x \in \mathcal{D}(\omega)\}$$

The resulting array of evaluations is called a **block**. Succinctly, we define the i^{th} trace block as follows: $\mathbf{u}_i = P_{\mathbf{u}_i}|_{\beta\mathcal{D}(\omega)}$.

We’ll sometimes write this succinctly as $\mathbf{u}_i = \text{NTT}(\text{iNTT}(\mathbf{m}_i))$. Note that $\mathbf{u}_i \neq \mathbf{m}_i$ since the domain of the NTT and the iNTT differ.

²³We define blocks in Appendix A.2.1 and Reed-Solomon encoding in Appendix A.3.

²⁴Note that while most discussions of NTTs and iNTTs occur over a multiplicative subgroup of a finite field, RISC Zero’s NTT and iNTT implementation works over a coset of a multiplicative subgroup as well. We write NTTs and iNTTs as two-argument functions that receive (1) an array over an (implied) finite field and (2) a cyclic domain or a coset of a cyclic domain. See Appendix A.2.3 for more details.

C.2 Constructing $\text{Com}(\mathbf{w}_{\text{control}})$ and $\text{Com}(\mathbf{w}_{\text{data}})$

The Prover constructs two Merkle Trees, a **Control Tree** and a **Data Tree**, using the trace blocks (i.e., the evaluations of the Trace Polynomials on $\beta D(\omega)$) as the leaves. Each leaf's address corresponds to a point of the coset $\beta \mathcal{D}(\omega)$, and the contents of the leaf at address z consist of the trace polynomials evaluated at z . As articulated in Appendix B, the Prover computes the Merkle cap of the trees, M_C and M_D , and commits them to the seal.

C.3 Randomness for PLOOKUP Accumulation

In the interactive version of the protocol, the Verifier sends some randomly chosen PLOOKUP Accumulation Parameters, after receiving the Merkle caps M_C and M_D . In the non-interactive version, we generate the PLOOKUP Accumulation Parameters by running a SHA-2 CRNG on $M_C || M_D$.²⁵

C.4 Constructing $\text{Com}(\mathbf{w}_{\text{accum}})$

As in [13] and [2], we instantiate our system as a randomized AIR with pre-processing (RAP). The random accumulation parameter defined in Appendix C.3 is used (along with \mathbf{w}_{data}) to generate accumulator polynomials; these accumulator polynomials serve to authenticate the permuted memory columns described in Section 2.1. The evaluations of these accumulator polynomials are called $\mathbf{w}_{\text{accum}}$; the resulting Merkle cap, M_P , is committed to the seal. The seal now reads

$$M_C || M_D || M_P$$

C.5 DEEP-ALI: Constructing $\text{Com}(\mathbf{w}_{\text{validity}})$ and the DEEP answer sequence

The DEEP-ALI portion of the protocol includes two elements of randomness and two Prover commitments.

C.5.1 Randomness for Algebraic Linking

In the interactive version of the protocol, the Verifier sends a randomly chosen Mixing Parameter, $\alpha_{\text{constraints}}$, after receiving the Merkle root M_P . In the non-interactive version, we generate the mixing parameter by running a SHA-2 CRNG on $M_C || M_D || M_P$.

C.5.2 Constructing $\text{Com}(\mathbf{w}_{\text{validity}})$

The Prover first generates a number of **Constraint Blocks**, \mathbf{c}_i using time-symmetric rule-checking functions and the Trace Blocks²⁶. Then, these \mathbf{c}_i are mixed together

²⁵We use $||$ to denote concatenation.

²⁶We use “trace block” to refer to $\text{NTT}(P_{\mathbf{u}_i}, \mathcal{D}(\omega))$. Note that the i^{th} trace block has 4x the length of trace column \mathbf{u}_i

using the Constraint Mixing Parameter $\alpha_{\text{constraints}}$ to form the **Mixed Constraint Block, \mathbf{C}** :

$$\mathbf{C} = \mathbf{mix}_{\alpha_{\text{constraints}}}(C_1, \dots, C_v) = \alpha_{\text{constraints}} C_1 + \dots + \alpha_{\text{constraints}}^n C_v$$

We note that \mathbf{C} is a polynomial of degree at most $5|\mathcal{D}(\omega)|$, and that $\mathbf{C}(c) = 0$ for each $c \in \mathcal{D}(\omega)$. The Prover divides \mathbf{C} by a publicly known **Zeros Block**²⁷ to form the **High Degree Validity Block**. To wrap up the construction, the Prover splits the High Degree Validity Block into 4 **Low Degree Validity Blocks**.²⁸, then commits those blocks to a new Merkle Tree (the Validity Tree) and appends the root of the tree, M_v , to the seal. At this stage, the seal reads as follows:

$$M_C || M_D || M_P || M_v$$

C.5.3 Randomness for DEEP

In the interactive version of the protocol, the Verifier sends a randomly chosen test point, $z \in \mathbb{F}_q$, after receiving the Merkle root M_v . In the non-interactive version, we generate z by running a SHA-2 CRNG on $M_C || M_D || M_P || M_v$.

C.5.4 Intuition for DEEP technique

The Prover will send the DEEP answer sequence, which allows the Verifier to enforce to check the $\mathbf{C}(z) = Z(z)V(z)$ at the DEEP Test point, z . The DEEP answer sequence includes the **tapset**, which allows the Verifier to compute the $\mathbf{C}(z)$. Additionally, the Verifier sends the evaluation of V at z .

Since z is not in the Merkle Tree commitments for the Trace Polynomials or the Validity Polynomials, the Prover sends additional information in order enforce that the values provided agree with the earlier Merkle commitments.

The core of the DEEP technique is the insight that if $f(z) = a$, then $x - z$ divides $f(x) - a$ as polynomials. Moreover, if $f(z_1) = a_1$ and $f(z_2) = a_2$, then the polynomial $(x - z_1)(x - z_2)$ divides $f(x) - \bar{f}(x)$, where \bar{f} is the interpolation of (z_1, a_1) and (z_2, a_2) .

C.5.5 Constructing the DEEP answer sequence

The DEEP answer sequence consists of the **tapset**, the evaluation of V at z , and the coefficients of each DEEP polynomial.

We define a DEEP polynomial for each Trace Polynomial P_i and each low-degree

²⁷The Zeros Block consists of evaluations of the Zeros Polynomial over $\beta\mathcal{D}(\omega)$.

²⁸We abuse terminology here in referring to the “degree” of a block. We write “low degree block” to mean that the polynomial associated with the block has degree less than or equal to the degree of the trace polynomials, n (which is equal to the length of a column of the trace). Enforcing RISC-V rules uses relations up to degree 5, which means the degree of the mixed constraint block is $5n$. After dividing by the zeros block, the High Degree Validity Block has degree $4n$, and the Low Degree Validity Polynomials have degree n .

validity polynomial \mathbf{v}_j . Given trace polynomial P_i , we define the DEEP polynomial d_i as:

$$d_i(x) = \frac{P_i(x) - \overline{P}_i(x)}{(x - x_1) \cdots (x - x_n)}$$

where $(x_1, P_i(x_1)), \dots, (x_n, P_i(x_n))$ are the taps²⁹ of P_i at z and $\overline{P}_i(x)$ is the polynomial interpolation of those taps.

DEEP polynomials for v'_j are defined analogously and denoted by d_{n_P+j} where n_P is the total number of trace polynomials. Since there are always 4 validity polynomials in our use cases, we end up with 4 DEEP validity polynomials $d_{n_P}, \dots, d_{n_P+3}$.

The Prover interpolates each \overline{P}_i and sends the coefficients to the Verifier, unencrypted. The seal now reads:

$$M_C || M_D || M_P || M_v || \text{DEEP-Answer-Sequence}$$

The Verifier can now check³⁰ that $V(z)C(z) = Z(z)$.

C.6 The Batched FRI Protocol

To prove Assertion (a). , RISC Zero uses the batched FRI protocol as described in Section 8.2 of the Proximity Gaps for Reed-Solomon Codes paper.

First, the DEEP blocks are batched and then re-indexed to form a single block $f^{(0)}$. At this stage, the Prover has reduced the assertion of computational integrity to an assertion that $f^{(0)}$ is less than some n .

Then, over r recursive steps, the Prover reduces the assertion that $\deg(f^{(0)}) < n$ to an assertion that $\deg(f^{(r)}) < 256$.

Batching

The Verifier uses an HMAC to choose a DEEP Mixing Parameter, α_2 . The Prover uses α_2 to mix d_0, \dots, d_{n_P+3} . The result of this mixing is denoted $d(x)$:

$$d(x) = \mathbf{mix}_{\alpha_2}(d_0, \dots, d_{n_P+3}) = \sum_{i=0}^{n_P+3} \alpha_2^i d_i(x)$$

Re-indexing

Rather than evaluating d over the coset $\beta\mathcal{D}(\omega)$, the Prover first defines $f^{(0)}$ as a re-indexing³¹ of d :

$$f^{(0)}(x) = d(\beta^{-1}x)$$

²⁹For details on taps, see Appendix A.5.2.

³⁰(TODO Check with Jeremy) The Verifier computes the LHS by evaluating the DEEP validity polynomials at z^4 and using those values to compute $V(z)$. The Verifier computes the RHS by evaluating $d_i(z)$ for each trace polynomial and uses those values to compute $C(z)$ and then $V(z)$.

³¹This re-indexing serves to simplify our application of the FRI protocol from “coset FRI” to “subgroup FRI.”

Now, the Prover evaluates $f^{(0)}$ over $\mathcal{D}(\omega)$ (which yields the same array as evaluating d over $\beta\mathcal{D}(\omega)$). The Prover commits the evaluations to a Merkle tree and adds the associated Merkle root $M_{f^{(0)}}$ to the seal.

The seal now reads:

$$M_C || M_D || M_P || M_v || \text{DEEP-Answer-Sequence} || M_{f^{(0)}}$$

FRI Commit Rounds

Over the course of r rounds of FRI, the Prover constructs $f^{(1)}, \dots, f^{(r)}$, committing a Merkle root $M_{f^{(i)}}$ for each.

RISC Zero uses a *split factor* of 16 throughout FRI, but we write the split factor as l for generality.³². Each Merkle tree $M_{f^{(i)}}$ is constructed by evaluating $f^{(i)}$ over $\mathcal{D}^{(i)} = \mathcal{D}(\omega^{l^i})$.³³

For $i = 1, \dots, r$, each $f^{(i)}$ is constructed by computing $\text{mix}_{\alpha^{(i)}}(\text{split}(f_{i-1}))$. The functions **mix** and **split** are defined in Appendix A.2, and the mixing parameters $\alpha^{(i)}$ are verifier-supplied³⁴ random parameters.

The Prover constructs a Merkle tree and commits a root for each $f^{(1)}, \dots, f^{(r-1)}$. For $f^{(0)}$, the Prover interpolates the result and sends the coefficients.

$$M_C || M_D || M_P || M_v || \text{Coefficients-for-DEEP} || M_{f^{(0)}} || M_{f^{(1)}} || \dots || \text{Coefficients-for-} f^{(r)}$$

FRI Queries

After the FRI commit rounds, the Verifier makes a number of **queries** to check the integrity of the FRI commitments.

For a single query, the Verifier specifies an element $g^{(0)} \in \mathcal{D}^{(0)}$, which then induces:

1. a choice of $g^{(i)} \in \mathcal{D}^{(i)}$ for each $i = 1, \dots, r$. Specifically, $g^{(i)} = (g^{(i-1)})^l$.
2. a coset $C^{(i)}$ for each $i = 0, \dots, r-1$. Specifically, $C^{(i)} \subset \mathcal{D}^{(i)}$ contains the l^{th} roots of $g^{i+1} \in \mathcal{D}^{(i+1)}$.

The Prover returns $f^{(i)}|_{C^{(i)}}$ for each $i = 0, \dots, r-1$ and $f^{(r)}(g^{(r)})$.

The verify checks:

1. The FRI batching commitment matches against the DEEP polynomials at $g^{(0)}$.

³²Ben-Sasson, et. al., write the split factor as $l^{(i)}$ to allow for the possibility of varying the split factor throughout the protocol.

³³Note that $\mathcal{D}^{(r)} \subset \dots \subset \mathcal{D}^{(0)}$ and that $\frac{|\mathcal{D}^{(i)}|}{|\mathcal{D}^{(i+1)}|} = l$.

³⁴Generated using the Fiat-Shamir Heuristic

2. The **split** and **mix** operations match from round-to-round. This operation can be checked locally, using only the values revealed for the query. In particular, the value of $f^{(i+1)}(g^{(i+1)})$ can be computed using the values $f^{(i)}|_{C^{(i)}}$.

D Key Theorems for Soundness Analysis

We state the key theorems from [5] here, using the notation from [15]:

Theorem 5. (*Correlated Agreement Theorem*) Let $\text{RS}_k = \text{RS}_k[F, D]$ be the Reed-Solomon code over a finite field F with defining set $D \subseteq F$ and rate $\rho = \frac{k}{|D|}$. Given a proximity parameter $\theta \in (0, 1 - \sqrt{\rho})$ and words $f_0, f_1, \dots, f_{N-1} \in F^D$ for which

$$\frac{|\{\lambda \in F : \delta(f_0 + \lambda \cdot f_1 + \dots + \lambda^{N-1} \cdot f_{N-1}, \text{RS}_k) \leq \theta\}|}{|F|} > \epsilon$$

where ϵ is as in (1) and (2) below. Then there exist polynomials $p_0(X), p_1(X), \dots, p_{N-1}(X)$ belonging to RS_k , and a set $A \subseteq D$ of density $\frac{|A|}{|D|} \geq 1 - \theta$ on which f_0, \dots, f_{N-1} jointly coincide with p_0, \dots, p_{N-1} , respectively. In particular,

$$\delta(f_0 + \lambda \cdot f_1 + \dots + \lambda^{N-1} \cdot f_{N-1}, \text{RS}_k) \leq \theta$$

for every $\lambda \in F$.

Quoting [15]: Depending on the decoding regime, the following values for ϵ are obtained by [5]:

1. Unique decoding regime.

For $\theta \in (0, \frac{1-\rho}{2})$, the theorem above holds with

$$\epsilon = (N-1) \cdot \frac{|D|}{|F|}$$

2. List decoding regime.

For $\theta \in (\frac{1-\rho}{2}, 1 - \sqrt{\rho})$, the theorem above holds with

$$\epsilon = (N-1) \cdot \frac{k^2}{|F| \cdot \min(\frac{1}{m}, \frac{1}{10})^7} \approx (N-1) \cdot m^7 \cdot \rho^{-\frac{3}{2}} \cdot \frac{|D|^2}{|F|}$$

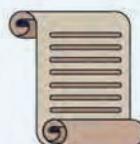
Theorem 6. (*Batched FRI soundness error*) Suppose that $q_i \in F^D, i = 0, \dots, L-1$, is a batch of functions given by their domain evaluation oracles. If an adversary passes batched FRI for $\text{RS}_k[F, D]$ and proximity parameter $\theta = 1 - \sqrt{\rho} \cdot (1 + \frac{1}{2m})$, $m \geq 3$, with a probability larger than

$$\epsilon = (L - \frac{1}{2}) \cdot \frac{(m + \frac{1}{2})^7}{3\sqrt{\rho}^3} \cdot \frac{|D_0|^2}{|F|} + \frac{(2m+1) \cdot (|D_0| + 1) \cdot \sum_{i=1}^r a_i}{\sqrt{\rho} \cdot |F|} + (1 - \theta)^s$$

then the functions $q_i \in F^D, i = 0, \dots, L-1$, have correlated agreement with $\text{RS}_k[D, F]$ on a set of density of at least $\alpha > (1 + \frac{1}{2m}) \cdot \sqrt{\rho}$.

zkEVM

Ye Zhang
Scroll



Scroll, zkEVM

We thank Vitalik Buterin, Barry Whitehat, Chih-Cheng Liang, Kobi Gurkan and Georgios Konstantopoulos for their reviews and insightful comments.

TL;DR

We believe zk-Rollup to be the holy grail — a best-in-class Layer 2 scaling solution that is very cheap and secure. However, existing zk-Rollups are application-specific, which makes it hard to build general composable DApps inside a zk-Rollup and migrate existing applications. We introduce zkEVM, which can generate zk proofs for general EVM verification. This allows us to build a fully EVM-compatible zk-Rollup, which any existing Ethereum application can easily migrate to.

In this article, we identify the design challenges of zkEVM and why it's possible now. We also give a more specific intuition and describe the high-level ideas of how to build it from scratch.

Background

[zk-Rollup](#) is recognized as the best scaling solution for Ethereum. It is as secure as Ethereum Layer 1 and has the shortest finalizing time comparing to all other Layer 2 solutions (Detailed comparisons [here](#)).

In the medium to long term, ZK rollups will win out in all use cases as ZK-SNARK technology improves. — Vitalik Buterin

The basic idea of zk-Rollup is to aggregate a huge number of transactions into one Rollup block and generate a succinct proof for the block off-chain. Then the smart contract on Layer 1 only needs to verify the proof and apply the updated state directly without re-executing those transactions. This can help save gas fee for an order of magnitude since proof verification is much cheaper than re-executing computations. Another saving comes from data compression (i.e., only keep minimum on-chain data for verification)

Although zk-Rollup is secure and efficient, its applications are still limited to payments and swaps. It's hard to build general-purpose DApps due to the following two reasons.

- First, if you want to develop DApps in a zk-Rollup, you need to write all your smart contract logic using a special language (i.e. [R1CS](#)). Not only is the syntax of required language complicated, but doing so also demands extremely strong expertise in zero-knowledge proof.
- Second, current zk-Rollup doesn't support composability[1]. It means different zk-Rollup applications can't interact with each other within Layer 2. Such quality significantly undermines the composability of DeFi applications.

In a nutshell, zk-Rollup is developer-unfriendly and has limited functionality for now. That's the biggest problem we want to tackle. We want to provide the best developer experience and support composability within Layer 2 by supporting native EVM verification

directly, so that existing Ethereum applications can simply migrate over onto the zk-Rollup as is.

Build general DApps in zk-Rollup

There are two ways to build general DApps in zk-Rollup.

- One is building application-specific circuit (“ASIC”) for different DApps.
- The other is building a universal “EVM” circuit for smart contract execution.
“circuit” refer to the program representation used in zero-knowledge proof. For example, if you want to prove $\text{hash}(x) = y$, you need to re-write the hash function using the circuit form. The circuit form only supports very limited expressions (i.e., R1CS only support addition and multiplication). So, it’s very hard to write program using the circuit language — you have to build all your program logic (including if else, loop and so on) using add and mul.

The first approach requires developer to design specialized “ASIC” circuits for different DApps. It is the most traditional way to use zero-knowledge proof. Each DApp will have a smaller overhead through customized circuit design. However, it brings the problem of composability since the circuit is “static” and terrible developer experience since it needs strong expertise in circuit design[2].

The second approach doesn’t require any special design or expertise for developer. The high-level idea of such machine-based proof is that any program will eventually run on CPU, so we only need to build a universal CPU circuit to verify the low-level CPU step. Then we can use this CPU circuit to verify any program execution. In our scenario, program is smart contract and CPU is EVM. However, this approach is not commonly adopted in the past years due to its large overhead. For example, even if you only want to prove the result of `add` is correct in one step, you still need to afford the overhead of an entire EVM circuit. If you have thousands of steps in your execution trace, it will be 1000x EVM circuit overhead on the prover side.[3]

Recently, there has been a lot of research going on to optimize zk proofs following those two approaches, including (i) proposing new zk-friendly primitives i.e. [Poseidon hash](#) can achieve 100x efficiency than SHA256 in circuit, (ii) ongoing work on improving efficiency of general-purpose verifiable VMs, as in [TinyRAM](#), and (iii) a growing number of general-purpose optimization tricks like Plookup, and even more generally faster cryptographic libraries.

In our [previous article](#), We propose to design “ASIC” circuit for each DApp and let them communicate through cryptographic commitments. However, based on the feedback from the community, we changed our priority and will focus on building a general EVM circuit (so called “zkEVM”) first following the second approach. zkEVM will allow exactly the same developer experience as developing on Layer 1. Instead of leaving design complexity to the developer, we will take over it and solve the efficiency problem through customized EVM circuit design.

Design challenges of zkEVM

zkEVM is hard to build. Even though the intuition is clear for years, no one has built a native EVM circuit successfully. Different from TinyRAM, it's even more challenging to design and implement zkEVM due to the following reasons:

- **First, EVM has limited support of elliptic curves.** For now, EVM only supports BN254 pairing. It's hard to do proof recursion since [cyclic elliptic curve](#) is not directly supported. It's also hard to use other specialized protocols under this setting. The verification algorithm has to be EVM friendly.
- **Second, EVM word size is 256bit.** EVM operates over 256-bit integers (much like most regular VMs operate over 32-64 bit integers), whereas zk proofs most "naturally" work over prime fields. Doing "mismatched field arithmetic" inside a circuit requires range proofs, which will add ~100 constraints per EVM step. This will blow up EVM circuit size by two orders of magnitudes.
- **Third, EVM has many special opcodes.** EVM is different from traditional VM, it has many special [opcodes](#) like CALL and it also has error types related to the execution context and gas. This will bring new challenges to circuit design.
- **Fourth, EVM is a stack-based virtual machine.** The [SyncVM](#) (zksync) and [Cario](#) (starkware) architecture defines its own IR/AIR in the register-based model. They built a specialized compiler to compile smart contract code into a new zk-friendly IR. Their approach is language compatible instead of native EVM-compatible. It's harder to prove for stack-based model and support native tool chain directly.
- **Fifth, Ethereum storage layout carries a huge overhead.** The Ethereum storage layout highly relies on Keccak and a huge [MPT](#)[4], both of them are not zk-friendly and have a huge proving overhead. For example, Keccak hash is 1000x larger than Poseidon hash in circuit. However, if you replace Keccak with another hash, it will cause some compatibility problems for the existing Ethereum infrastructure.
- **Sixth, machine-based proof has a gigantic overhead.** Even if you can handle all the aforementioned problems properly, you still need to find an efficient way to compose them together to get a complete EVM circuit. As I mentioned in previous section, even simple opcodes like add might result in the overhead of the entire EVM circuit.

Why possible now?

Thanks for the great progress made by researchers in this area, more and more efficiency problems are solved in the last two years, the proving cost for zkEVM is eventually feasible! The biggest technology improvement comes from the following aspects:

- **The usage of polynomial commitment.** In the past few years, most succinct zero-knowledge proof protocols stick to R1CS with PCP query encoded in an application-specific trusted setup. The circuit size usually blows up and you can't do many customized optimizations since the degree of each constraint needs to be 2 ([bilinear pairing](#) only allows one multiplication in the exponent). With [polynomial commitment schemes](#), you can lift your constraints to any degree with a universal setup or even transparent setup. This allows great flexibility in the choice of backend.
- **The appearance of lookup table arguments and customized gadgets.** Another strong optimization comes from the usage of lookup tables. The optimization is

firstly proposed in [Arya](#) and then gets optimized in [Plookup](#). This can save A LOT for zk-unfriendly primitives (i.e., bitwise operations like AND, XOR, etc.) . [Customized gadgets](#) allow you to do high degree constraints with efficiency. [TurboPlonk](#) and [UltraPlonk](#) defines elegant program syntax to make it easier to use lookup tables and define customized gadgets. This can be extremely helpful for reducing the overhead of EVM circuit.

- **Recursive proof is more and more feasible.** Recursive proof has a huge overhead in the past since it relies on special pairing-friendly cyclic elliptic curves (i.e. [MNT curve-based construction](#)). This introduces a large computation overhead. However, more techniques are making this possible without sacrificing the efficiency. For example, [Halo](#) can avoid the need of pairing-friendly curve and amortize the cost of recursion using special inner product argument. Aztec shows that you can do proof aggregation for existing protocols directly (lookup tables can reduce the overhead of [non-native field operation](#) thus can make the verification circuit smaller). It can highly improve the scalability of supported circuit size.
- **Hardware acceleration is making proving more efficient.** To the best of our knowledge, we have made the fastest GPU and ASIC/FPGA accelerator for the prover. [Our paper](#) describing ASIC prover has already been accepted by the largest computer conference (ISCA) this year. The GPU prover is around 5x-10x faster than [Filecoin's implementation](#). This can greatly improve the prover's computation efficiency.

Ok, so how does it work and how to build it?

Besides the strong intuition and technology improvement, we still need to have a more clear idea of what we need to prove and figure out a more specific architecture. We will introduce more technical details and comparisons in the follow up articles. Here, we describe the overall workflow and some key ideas.

Workflow for Developers and Users

For developers, they can implement smart contracts using any EVM-compatible language and deploy the compiled bytecode on Scroll. Then, users can send transactions to interact with those deployed smart contracts. The experience for both users and developers will be the exactly the same as Layer 1. However, the gas fee is significantly reduced and transactions are pre-confirmed instantly on Scroll (withdraw only takes a few minutes to finalize).

Workflow for zkEVM

Even if the workflow outside remains the same, the underlying processing procedure for Layer 1 and Layer 2 are entirely different:

- Layer 1 relies on the re-execution of smart contracts.
- Layer 2 relies on the validity proof of zkEVM circuit.

Let's give a more detailed explanation of how things are going differently for transactions on Layer 1 and Layer 2.

In Layer 1, the bytecodes of the deployed smart contracts are stored in the Ethereum storage. Transactions will be broadcasted in a P2P network. For each transaction, each

full node needs to load the corresponding bytecode and execute it on EVM to reach the same state (transaction will be used as input data).

In Layer 2, the bytecode is also stored in the storage and users will behave in the same way. Transactions will be sent off-chain to a centralized zkEVM node. Then, instead of just executing the bytecode, zkEVM will generate a succinct proof to prove the states are updated correctly after applying the transactions. Finally, Layer 1 contract will verify the proof and update the states without re-executing the transactions.

Let's take a deeper look at the execution process and see what zkEVM needs to prove at the end of the day. In native execution, EVM will load the bytecode and execute the opcodes in the bytecode one by one from beginning. Each opcode can be thought as doing the following three sub-steps : (i) Read elements from stack, memory or storage (ii) Perform some computation on those elements (iii) Write back results to stack, memory or storage.[5] For example, add opcode needs to read two elements from stack, add them up and write the result back to stack.

So, it's clear that the proof of zkEVM needs to contain the following aspects corresponding to the execution process

- The bytecode is correctly loaded from persistent storage
(You are running the correct opcode loaded from a given address)
- The opcodes in the bytecode are executed one by one consistently
(The bytecode is executed in order without missing or skipping any opcode)
- Each opcode is executed correctly
(Three sub-steps in each opcode are carried out correctly, R/W + computation)

zkEVM Design highlight

When designing the architecture for zkEVM, we need to handle/address the aforementioned three aspects one by one.

1. We need to design a circuit for some cryptographic accumulator.
This part acts like a “verifiable storage”, we need some technique to prove we are reading correctly. A cryptographic accumulator can be used to achieve this efficiently.[6]
Let's take Merkle Tree as an example. The deployed bytecode will be stored as a leaf in the Merkle Tree. Then, verifier can verify the bytecode is loaded correctly from a given address using a succinct proof (i.e., verify Merkle Path in circuit). For Ethereum storage, we need the circuit to be compatible with Merkle Patricia Trie and Keccak hash function.
2. We need to design a circuit to link the bytecode with the real execution trace.
One problem to move bytecode into a static circuit is conditional opcodes like jump (corresponding to loop, if else statement in smart contracts). It can jump anywhere. The destination is not determined before one has run the bytecode with specific input. That's the reason why we need to verify the real execution trace. The execution trace can be thought as “unrolled bytecode”, it will include the sequence of opcodes in the real execution order (i.e., if you jump to another position, the trace will contain the destined opcode and position).
Prover will provide the execution trace directly as witness to the circuit. We need to

prove that the provided execution trace is indeed the one “unrolled” from the bytecode with specific input. The idea is forcing the value of program counter to be consistent. To deal with the undetermined destination, the [idea](#) is letting prover provide everything. Then you can check the consistency efficiently using a lookup argument (i.e., prove the opcodes with proper global counter is included in the “bus”).

3. We need to design circuits for each opcode (Prove read, write and computations in each opcode are correct).

This is the most important part — Prove each opcode in the execution trace is correct and consistent. It will bring a huge overhead if you put all the things together directly. The important optimization idea here is that

1. We can separate R/W and computation into two proofs. One will fetch the elements needed for all the opcodes into a “bus”. The other will prove the computation performed on the elements from the “bus” is carried out correctly. This can greatly reduce the overhead of each part (i.e., you don’t need to consider the entire EVM storage in the computation proof). In a [more detailed specification](#), the first one is called “state proof” and the second one is called “EVM proof”. Another observation is that “bus mapping” can be efficiently handled by the lookup argument.
2. We can design higher degree customized constraint for each opcode (i.e., EVM word can be solved efficiently by chopping off it into several chunks). We can choose whether to “open” a constraint through a selector polynomial according to our need. This can avoid the overhead of the entire EVM circuit in each step.

This architecture is firstly specified by Ethereum Foundation. It’s still at an early stage and under active development. We are collaborating with them closely on this to find the best way to implement the EVM circuit. So far, the most important traits are defined and some opcodes have already been implemented [here](#) (using UltraPlonk syntax in the Halo2 repo). More details will be introduced in the follow up articles. We refer interested readers to read this [document](#). The development process will be transparent. This will be a community-effort and fully open-sourced design. Hope more people can join and contribute to this.

What else it can bring?

zkEVM is much more than just Layer 2 scaling. It can be thought as a direct way to scale Ethereum Layer 1 via Layer-1 validity proof. That means you can scale existing Layer 1 without any special Layer 2.

For example, you can use zkEVM as a full node. The proof can be used for proving transitions between existing states directly — No need to port anything to Layer 2, you can prove for all Layer 1 transactions directly! More broadly, you can use zkEVM to generate a succinct proof for the whole Ethereum like Mina. The only thing you need to add is proof recursion (i.e. embed the verification circuit of a block to the zkEVM)[7].

Conclusion

zkEVM can provide the same experience for developers and users. It’s order of magnitudes cheaper without sacrificing security. There has been proposed architecture to

build it in a modular way. It leverages recent breakthrough in zero-knowledge proof to reduce the overhead (including customized constraint, lookup argument, proof recursion and hardware acceleration). We look forward to seeing more people joining the zkEVM community effort and brainstorming with us!

A little about us

Scroll Tech is a newly built tech-driven company. We aim to build an EVM-compatible zk-Rollup with a strong proving network ([See an overview here](#)). The whole team is now focusing on the development. We are actively hiring more passionate developers, reach out to us at hire@scroll.io. If you have any question about the technical content, reach out to me at ye@scroll.io. DM is also open.

Footnotes

[1]: Starkware claims to achieve composability a few days ago ([reference here](#))

[2]: Circuit is fixed and static. For example, you can't use variable upper bound loop when implementing a program as a circuit. The upper bound has to be fixed to its maximum value. It can't deal with dynamic logic.

[3]: To make it more clear, We elaborate about the cost of EVM circuit here. As we described earlier, circuit is fixed and static. So, EVM circuit needs to contain all possible logic (10000x larger than pure add). That means even if you only want to prove for add, you still need to afford the overhead of all possible logics in the EVM circuit. It will 10000x amplify the cost. In the execution trace, you have a sequence of opcodes to prove and each opcode will have such a large overhead.

[4]: EVM itself is not tightly bound to the Merkle Patricia tree. MPT is just how Ethereum states are stored for now. A different one can easily be plugged in (i.e., the current proposal to replace MPT with [Verkle trees](#)).

[5]: This is a highly simplified abstraction. Technically, the list of “EVM state” is longer including PC, gas remaining, call stack (all of the above plus address and staticness per call in the stack), a set of logs, and transaction-scoped variables (warm storage slots, refunds, self-destructs). Composability can be supported directly with additional identifier for different call context.

[6]: We use accumulator for storage since the storage is huge. For memory and stack, one can use editable Plookup (“RAM” can be implemented efficiently in this way).

[7]: It's non-trivial to add a complete recursive proof to the zkEVM circuit. The best way to do recursion is still using cyclic elliptic curves (i.e., Pasta curve). Need some “wrapping” process to make it verifiable on Ethereum Layer 1.

A Proposal for Fractal Scaling

Kalman and Ago Lajko
Slush



Slush, a proposal for Fractal scaling Comment

Author: Ago Lajko & Kalman Lajko

TL; DR

The basic question of this document is what fractal scaling will look like, how to make bridging cheap between the different rollups, and how to make the whole fractal system secure for chained validiums.

We start with fractal scaling, then we follow with what bridging will look like between the rollups, first for users, then for smart contracts.

Then we discuss the question of validium security: exit strategies and the required DA. We find that the existing solutions are not scalable, and propose our solution, a common consensus mechanism for the descendant validiumss.

Then we generalise this solution in two parts, the first generalisation enables a majority of descendant validiumss to take control of the parent validium, making exiting easier in case of a malicious parent layer. The second generalisation replaces L1 PoS consensus mechanism with the new validium consensus (basically enabling the consensus mechanism to handle forking).

Following this we shape the tree structure to make the resulting system have fast confirmation time and low data requirements on each node. Finally we summarise our results ($TIDr^2$): a method of fractal scaling that enables transaction throughput that is nearly linear $O(N/\log(N))O(N/\log(N))$ in the number of nodes, with $O(\log(n))O(\log(n))$ finality time, and $O(\log(N))O(\log(N))$ storage requirements.

Document long outline

- We first summarise fractal scaling of ZK rollups and why there will be a need for bridging between different rollups instead of routing everything through L1.
- We continue with an overview of what the current bridging solutions are: trustless swapping, authority bridging, routing and pooled routing, and we discuss their drawbacks.
- Bridging

- Then we describe our bridging solution as an extreme form of pooled routing, and its benefits, and how different versions of it might be deployed.
 - Then we extend this bridging so that smart contracts can use it (i.e. atomicity). This comes in two parts, first we enable smart contracts to confirm the bridging process, or if there is no confirmation we enable them to route the transaction through the parent rollups (this is expensive but it is only a fallback option).
 - Then we look at a toy example of an AMM in this sharded setting.
- Security
 - Then we look at the security of rollups/validiums, and how they rely on the L1's security. We discuss two different solutions for secure exiting, the first is broadly how Starkex validium apps do security, the second is how Starknet rollup does it. We will see that neither of these solutions scale when trying to do iterated validiums. We then discuss a way of exiting that is scalable, but instead of relying only on the L1 security this mechanism enables descendant validiums to exit (the parent validiums) if a majority of the descendants are honest.
 - Then we generalise this solution in two parts. The first generalisation enables a majority of descendant validiums to take control of the parent validium, so that it is not the large number of descendant validiums that have to leave if the parent layer is malicious, but the single parent layer. This makes economic sense in case there is conflict between the parent layer and the descendants, as there are less parent nodes than descendant nodes. The second generalisation replaces L1 PoS consensus mechanism with the new validium consensus. For this the consensus mechanism has to be able to handle forking at the L1 level.
 - After this we look at the tree structure that makes this structure the most efficient in terms of confirmation time (the time it takes for a transaction to be confirmed and irreversible), and in terms of required data storage. We realise two things here, the first is that it makes sense for the tree to be narrow and deep (each validium should only have a few child validiums), and secondly that the content of each validium (except the child validiums ZKP) can be externalised into a separate validium, making the parent validium contain ZKPs.

- Finally we summarise our results: a method of fractal scaling that enables nearly linear $O(N/\log(n))$ transaction throughput in the number of nodes, all relying on a common consensus mechanism.

$O(N/\log(n))$

- As an Appendix we include some calculations.

Zk Rollups and Current bridging

First we review the existing fractal scaling and bridging landscape.

Zk Rollups

First let's look at the fractal-like, multiple layer ZK Rollup solution to scaling. In the first image, we have Virtual Machines (Rectangles) that output their compressed state as ZKPs to VMs higher in the tree. The ZKPs are verified by ZKP verifier smart contracts on the parent layer.

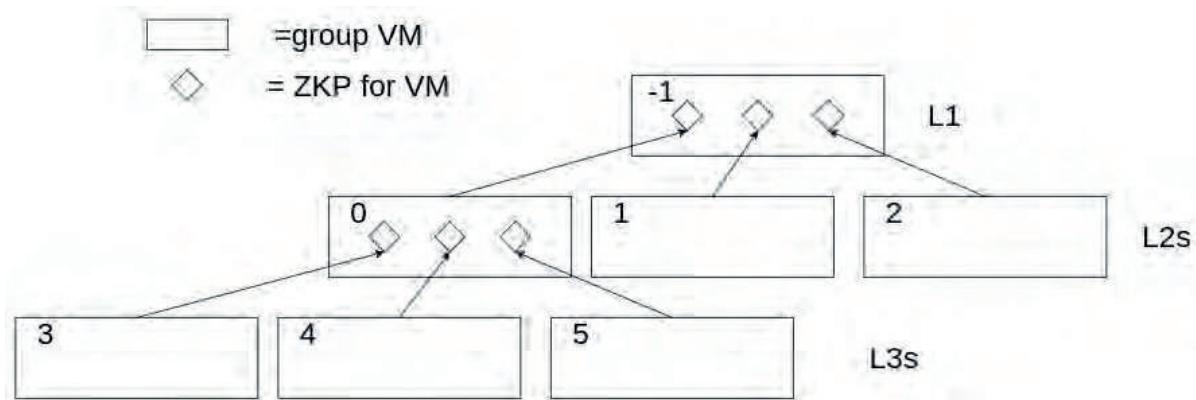


Fig. 1. Basic setup

When a state update happens, in our image VM 4 is updated with an internal transaction, the ZKP is updated in the hosting group VM, and the hosting VM's ZKP changes as well. This propagates up the tree to the L1, which is VM -1. We show this with the purple colour.

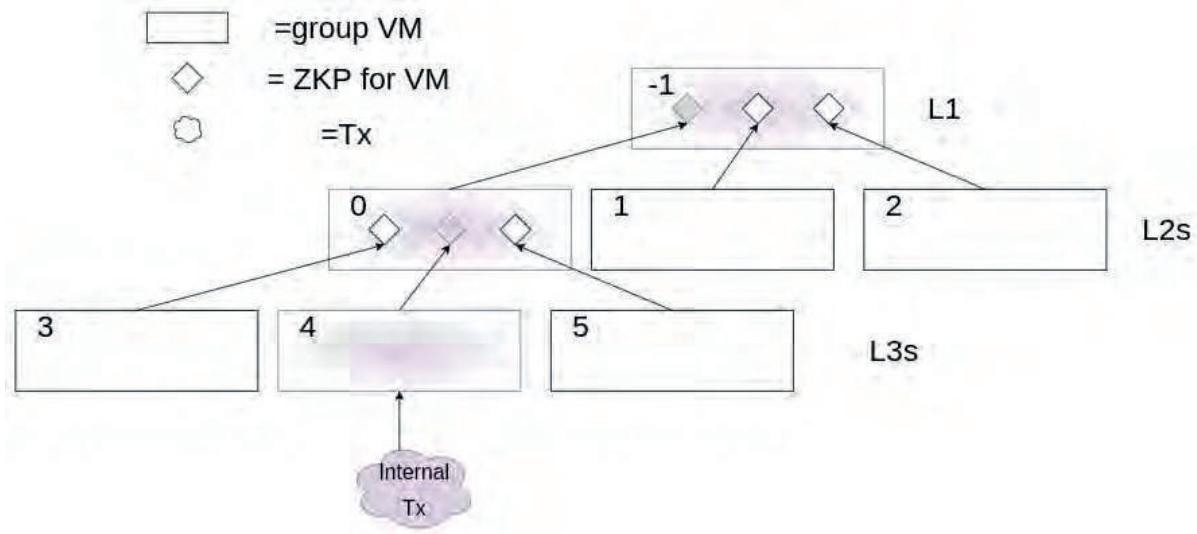


Fig. 2. A simple state update

In this setting bridging is the act of taking data from one VM to another on a different branch of the tree, without burdening L1. This is necessary as the different rollups can be assumed to be constant size in tx throughput, and the L1 could not handle the throughput if everything were routed through it.

Current Bridging

The area of trustless bridging is saturated with companies like Hop, Celer, deBridge or Connex. First we will broadly categorise the mechanisms of these companies and find that there is a trilemma. Then we briefly explain the way out of the trilemma, its cost, and how this will be used in the ideal bridging solution.

The first option to bridging would be routing your funds from source rollup up to L1 and then down to the destination rollup. This option is trustless, however if everyone used this option it would overload L1, so this option will always be expensive. However, the costs can be alleviated if you pool funds with other people, if you're going to the same destination. This is pooled routing. This is essentially how Hop network does bridging. When going from L3 or deeper, the different pools can be further pooled on L2. However, this pooling can only be used for applications that have a large user base on both the sending and receiving rollup.

Alternatively you can go directly between the layers (such as Connex). Although these are substantially faster and cheaper than going through L1, these systems only swap your funds on one chain for funds on another chain, which means there needs to be some other mechanism that compensates for net fund movement between the layers. This incurs additional costs, but more importantly means that this method actually relies on other solutions (such as pooled routing).

There is another way of going directly between layers, this is done by an authority that burns and mints the bridged tokens on each layer (such as Wormhole). This authority usually takes the form of a smaller blockchain, and there are security risks with having to trust these small blockchains. The drawbacks of such systems are obvious.

The trilemma

As we see we have a trilemma: the systems do not satisfy all three required properties of 1. being trustless, 2. not touching L1 , 3. and having real transfers not just swapping. The reason for this is that if we have real transfers and not swapping, then the funds have to be burned on one layer and minted on another. This minting can only happen if the burning has already happened. Confirming that the burning happened needs either trust in some authority, or a message passed down from L1, as the two layers are only connected through L1 trustlessly. This is the reason the trilemma exists.

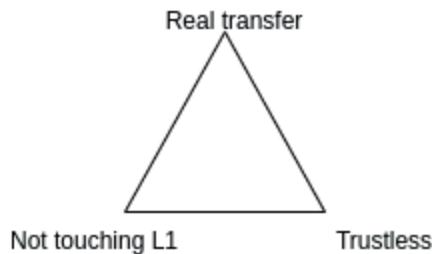


Fig. 3. The trilemma, choose 2.

Disclaimer on Stargate

After writing this, I found [Stargate](#). They use a similar idea that I will outline, proof of transaction membership based on the block's state hash. What is worse, they use the same language of “bridging trilemma”! However, their trilemma is different, it is about “instant guaranteed finality, unified liquidity, and native asset transactions”. They also do bridging between different L1s, and not on the fractal tree. In our classification they are not trustless, and only do swaps. And more importantly, the main part of our document is the second Security part, and our bridging process is optimised for the structures described in that part of the document.

Virtual tokens vs Bridged tokens

If we want to avoid L1 fees, and do not want to move funds on L1 we can only transact using virtual tokens, similar to Hop protocols hTokens. But these can be backed on L1 with the real tokens that they represent. These virtual tokens can be burned and minted as necessary. These could over time outcompete the native bridged tokens on each rollup, as it would be cheaper to transfer to other rollups. However they would be less secure, as an attack on a single rollup can compromise all the locked funds on L1. So it makes sense only to deploy these tokens on sufficiently trusted rollups.

Our trustless bridging

The trilemma's “solution”

In our bridging we make a different compromise. We can trustlessly confirm that the virtual token was burned on the sender chain if we link the burning together with the L1 state root in a ZKP*. This is enough proof for the destination chain, as they have access to the L1 state root trustlessly, and such a ZKP cannot be faked.

As a solution this is in fact an extreme form of pooled routing, as it effectively pools a bridging transaction not only with other bridging transactions to the destination chain, but with every other transaction happening on the rollup. So when the rollup's ZKP changes on the parent chain, that is the “pooled” transaction that contains the burning.

So in this solution the compromise we make in the trilemma is that the message does not have to pass through L1, we just need indirect inclusion in L1 via the ZKP-s. This proving can happen off-chain, which is cheap. The verification has to happen on-chain, but only at the destination, and if that is a n-th layer, that means that computation is cheap there. The L1 hash needs to be passed down from L1, but this is trustless and not a lot of data. With this in mind, we can explain the mechanism of this bridging method.

- Technical Note: This linking via a ZKP can be done as the burning on L_n is included in the L_n state hash via a Merkle tree, and this state hash is included in the L_n-1 state via the L_n's ZKP. Repeating this iteratively, we can chain the ZKP-s together until we reach L1.

New bridging

Now we get to the interesting part, sending a transaction from VM 3 to VM 2 trustlessly. First the transaction has to be sent on VM 3. This propagates up the tree

to the root. Now we can create a ZKP, that contains the L1's hash, and proves that the transaction happened. We denote this by the red triangle.

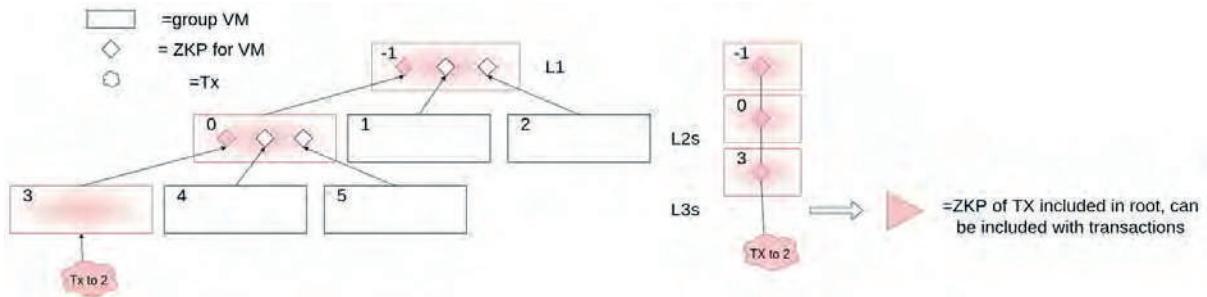


Fig. 4. Starting a transaction

Now we can create a transaction that includes this ZKP. We can send this tx on VM 2. VM 2 has access to the L1's hash, so it can verify that this transaction is valid. Now it can update its state accordingly. This means that the transaction happened, the bridging is complete.

Then we can also produce a yellow triangle ZKP that shows the receiving happened, we will use this later.

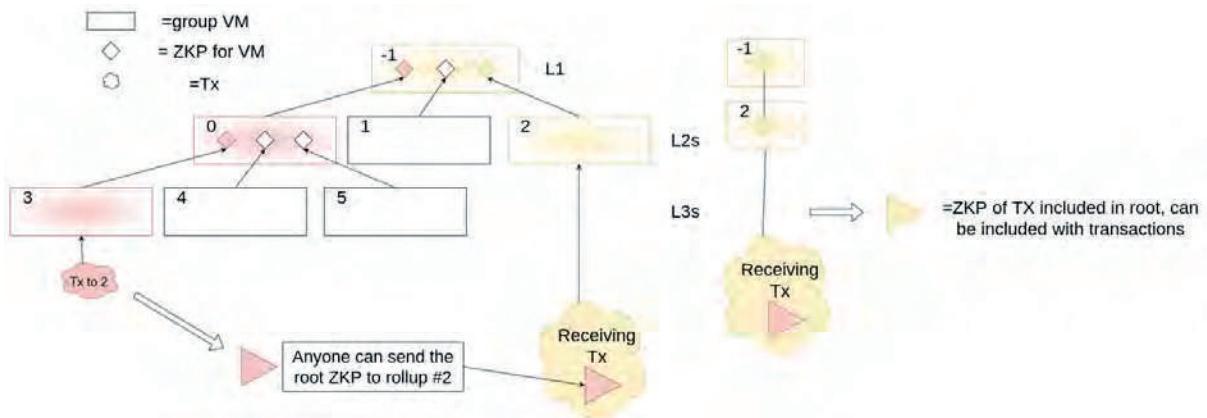


Fig. 5. Receiving a transaction

So on VM 3 the transaction was sent, and on VM 2 it can be claimed. We have achieved our objectives, this method is trustless, we did not incur L1 costs, and we did actually transfer funds between different layers.

Summary of Bridging

Here we explained a mechanism for transferring a virtual token, meaning it has to be burned on the sender and minted on the receiving rollup. But of course, any kind of data or even function call can be bridged via this method, and the price is

independent of how much other people use this method. Another advantage is that there are absolutely no L1 fees involved, as that is shared with other users of the rollups. The drawback of this solution is the proof verification, which is a substantial calculation that has to happen on-chain, even if only at the destination layer. And as a final note, this solution needs special smart contracts that verify the ZKPs and mint the tokens.

Automatic bridging and the sharded VM

This bridging method is great, but it has a weakness: smart contracts can't use it. Ideally, smart contracts could also bridge funds to other rollups, then that way they could make function calls and there would be interoperability between the different rollups. Up to this point we do not have this capability.

If I bridge funds for myself, and don't perform the receiving part of the bridging, then I just lose the funds. Smart contracts need an assurance that the transaction is actually received. (Incidentally, we do have this assurance when sending transactions to child rollups, as the new ZKP of the child rollup has to consume the transactions sent to it.) This section is about having that assurance even when sending to parent rollups, thereby enabling smart contracts to use this bridging mechanism to send data anywhere in the tree.

There can in general be no such guarantee, as the receiving rollup might have failed and stopped updating. But we can guarantee that if the receiving rollup continues updating, they will have to take the transaction as an input. We guarantee this by routing the transaction through L1. But as we know, that is expensive, and does not scale. So we use routing only as a failsafe, and most of the time will use a shortcut: our direct bridging method. If there are bots that scan the tree, and pass the transactions between the rollups, then we can send, receive, and confirm the transactions all with our bridging method. However, if this bridging does not happen, we have to revert to routing. This means that after the transaction was sent on the sender rollup, the rollup has to be frozen until it either confirms that the transaction was received, or until it starts routing the transaction through L1.

Bridging to solve automatic bridging

Our goal here is that if a smart contract sends a transaction via our previous bridging method to another rollup, then the destination rollup has to receive the transaction.

The sender rollup cannot by itself send the transaction over, it needs an external bot to do that. But if the bot does facilitate this information transfer, then it can quite

easily verify that the transaction did go through, we just need to bridge back a ZKP confirmation that the transaction was received, using our bridging method!

Continuing with our previous images, after VM 2 receives the transaction the yellow triangle ZKP can be computed (we did not use this ZKP in the previous step). This ZKP can be sent back to VM 3 for confirmation. This is similar to the previous update when VM 2 received the transaction. Importing this verification unfreezes VM 3's state, they can do transactions and internal state updates again. This confirmation and freezing is necessary so that funds are not lost, if VM 3 sends something, VM 2 needs to receive it.

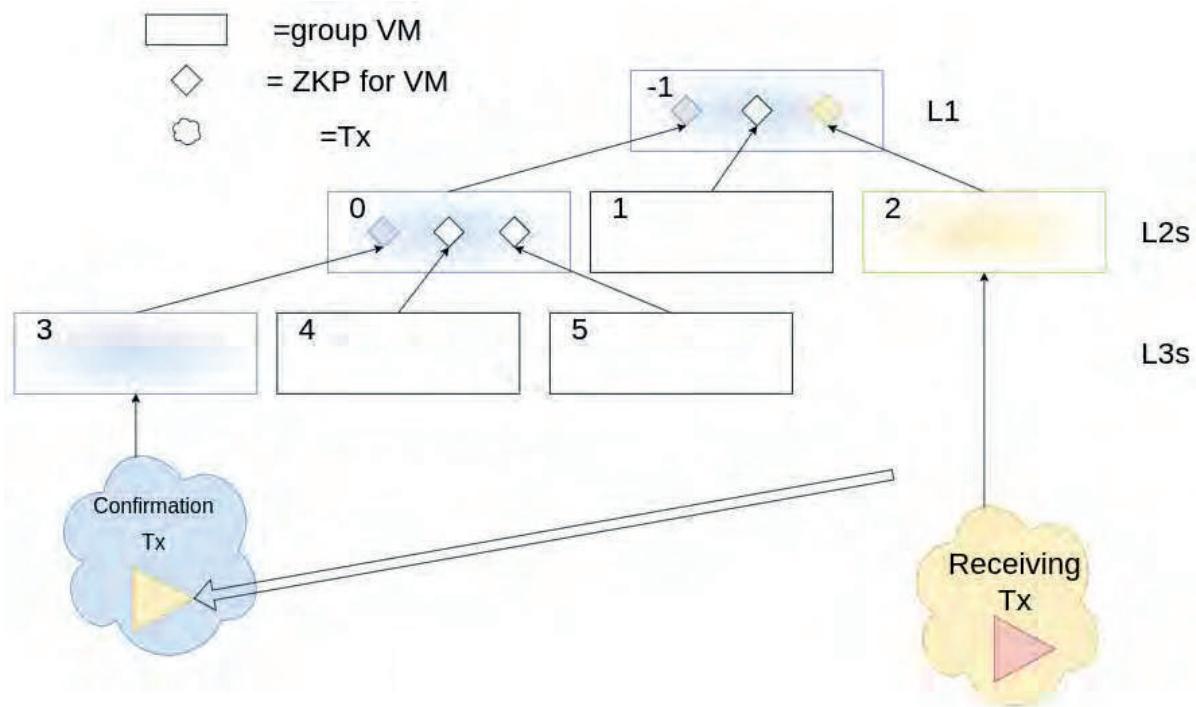


Fig. 6. Confirming a transaction

Fallback option

What happens however if no bot sends the transaction to the receiving chain, or brings the confirmation back? There is already a method that can incentivise rollups to perform operations: forced operations. This is used by apps using Starkex that might not have censorship-resistance, and it allows users to send transactions to the ZKP verifier contract on L1, and the prover has to process the transaction in a given timeframe, or risk freezing the app. We can use the same method here! If the nodes running the layer do not confirm that a transaction happened, then the only ZKP update possible is the one in which the transaction is pushed up the tree. After this it has to be pushed upward toward the common ancestor of the sender and recipient,

and then further forced operations can pull the transaction down to the recipient. This means that there are three options, the transaction is confirmed, it is routed via forced operations, or one of the rollups stops updating after it does not complete a forced operation.

This is the failsafe mechanism in case VM 3 and VM 2 have some communication error or one of them is compromised. This corresponds to reverting back to the traditional message passing in this setting, which means that we pass the message up towards the root, then we pull it down towards the recipient.

If VM 2 does not receive the ZKP, or VM 3 does not receive the confirmation, then the last two images (Fig. 5. and Fig. 6.) do not happen. This means the transaction was sent but it still needs to be received. In this case node 3's state is locked, and they can unlock it by pushing the ZKP of the original transaction upwards towards the parents. The parents then push it upwards as high as it is necessary, in this case that is the VM -1.

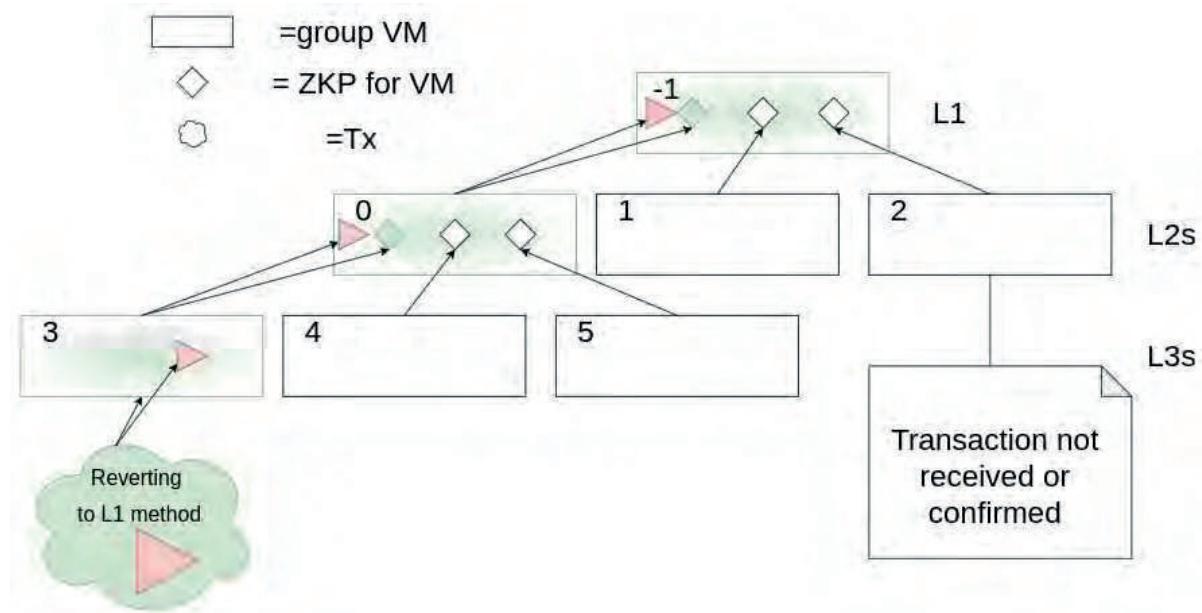


Fig. 7. Pushing a transaction upwards

After the transaction has been pushed upwards, the children should only be able to update their state if they pull and process the transaction to their VM.

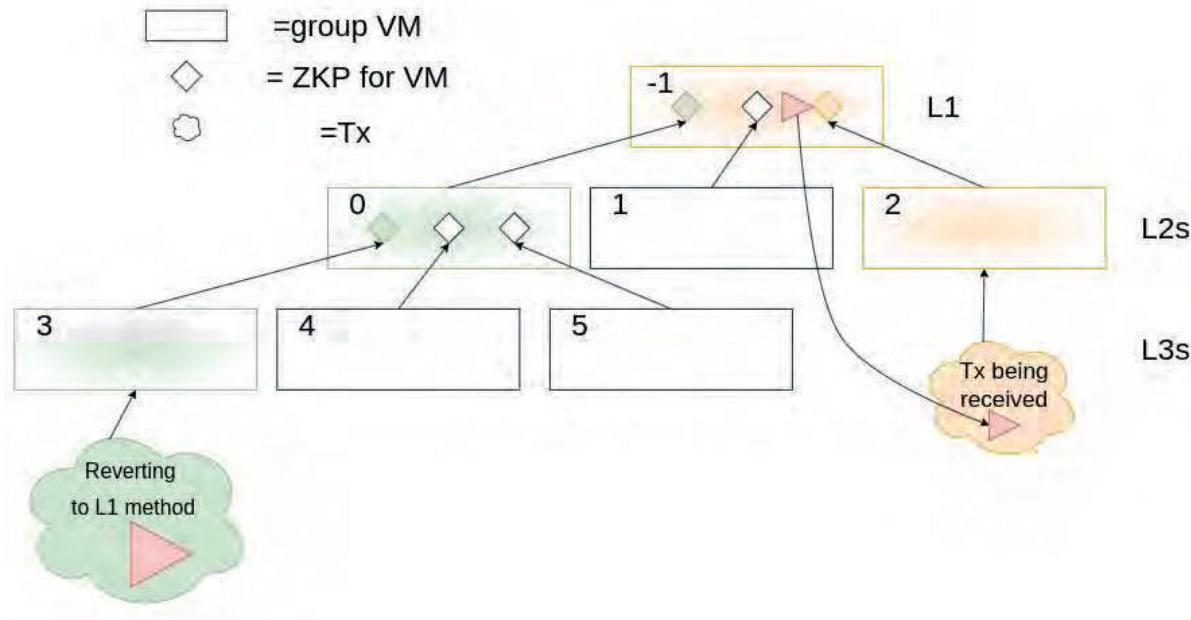


Fig. 8. Pulling a transaction downward

Sharded Virtual machine with AMM as an example

With this interoperability property, smart contracts can call and send data to other rollups. With this it would make less sense to have multiple AMM-s on each rollup than to have a separate rollup for an AMM that can handle transactions from and to all the other rollups.

The AMM could be hosted on a censorship resistant and decentralised rollup. This would communicate with other rollups with our bridging method, meaning other rollups and this rollup needs the smart contracts that verify against the state root. Once we have this, calling a smart contract would burn one type of virtual tokens on the sender rollup, the AMM rollup can receive this transaction via a bot, convert the funds to the other type of virtual token, and send this back to the original rollup. Then the new type of virtual token can be minted.

Operational methods, incentives

Now we see that it is possible to treat the different rollups as separate shards handling different parts of a single big VM. The passing of messages between the rollups need to be incentivized for bots, or done by the user themselves. This is based on the open nature of the rollups, sending to closed rollups is still possible if the rollup's host accepts the message.

Security, and the problem of iterated exits

Security has two parts, on the one hand it means verifying that all the transactions and state updates are correct, at which both ZK rollups validiums are great at because of the ZK proofs. But security also means avoiding censorship by the rollup/validium's operator and being able to continue updating the validium in case the operators stop. This is necessary so that the users funds aren't locked into the rollup/validium. Satisfying these conditions are non-trivial even for validiums, as we shall see later. Relying only on the L1's security means an L1 node should be able to reclaim funds from the rollup in case of censorship/failure.

Broadly speaking there are two big approaches to solving the second part of rollup/validium security.

- The first is the one employed by Starkex, which focuses on trustlessly exiting the validium in case it fails.
- The second solution is making sure that the rollup's consensus operates in a decentralised, censorship resistant way, with adequate data availability for L1 nodes to continue the consensus mechanism in case of failure. This is what Starknet rollup is aiming for.

Up to this point, we could have been talking about bridging between L2-s. But we can also do L3s, L4s, etc. This makes execution of computation cheaper but increases security concerns, as iterated rollups means multiple sequential exits or multiple consensus mechanisms that we have to trust. When thinking about security, we will also consider the security of these layers.

With iterated rollups, iterated validiums security is more and not less important on L2, as if L2 nodes start censoring transactions or stop updating the L2 state, then this ruins the network for all of the L3, L4 rollups/ validiums.

Trustless exits

The first solution is giving the rollups exits similar to Starkex. What this means is that we can exit L2 by making a transaction on L1, and then the L2 will have to process the transaction or freeze. This is a forced transaction. This is still not perfect, as the L2 might freeze, in which case our funds are lost. This can be solved however. We can first prove on L1 that we have some funds in the L2. Then the ZKP verifier smart contract on L1 should allow us to withdraw from L2, without there being an L2 state update.

Rollup quitting rollups

This exit strategy is even possible for smart contracts . This would mean proving on L1 that a smart contract with all its data was part of L2, and redeploying it on the L1 with its data.

With this method we could even exit the L2 not with any, but with a ZKP verifier smart contract of an L3. This would mean redeploying the L3 on L1 so that it becomes an L2. This means that child rollups can exit a parent rollups.

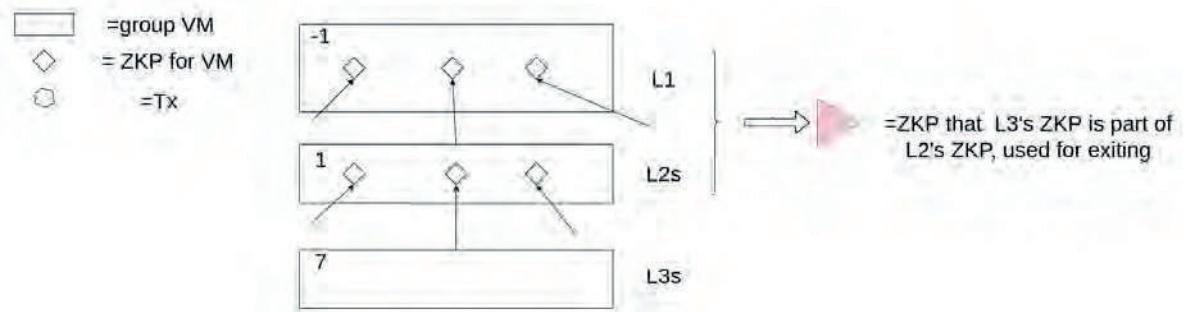


Fig. 9. Preparing for exit

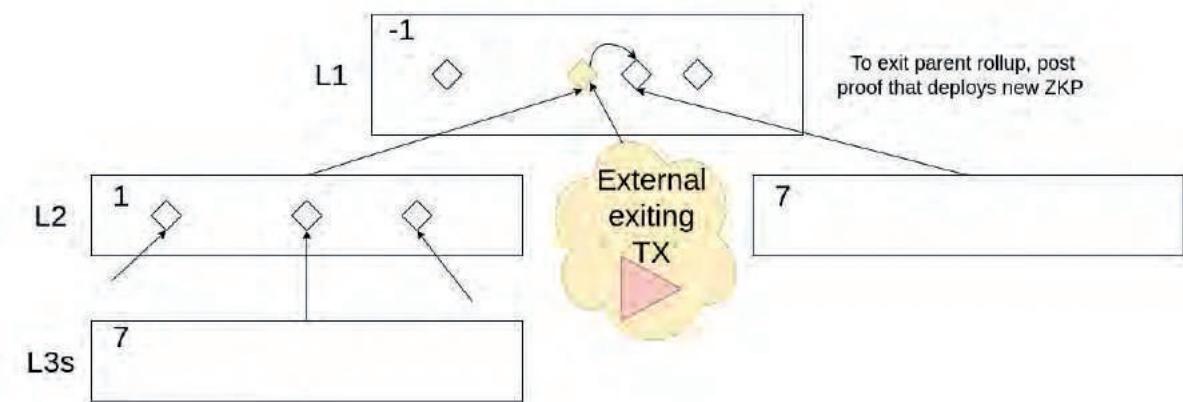


Fig. 10. Exiting

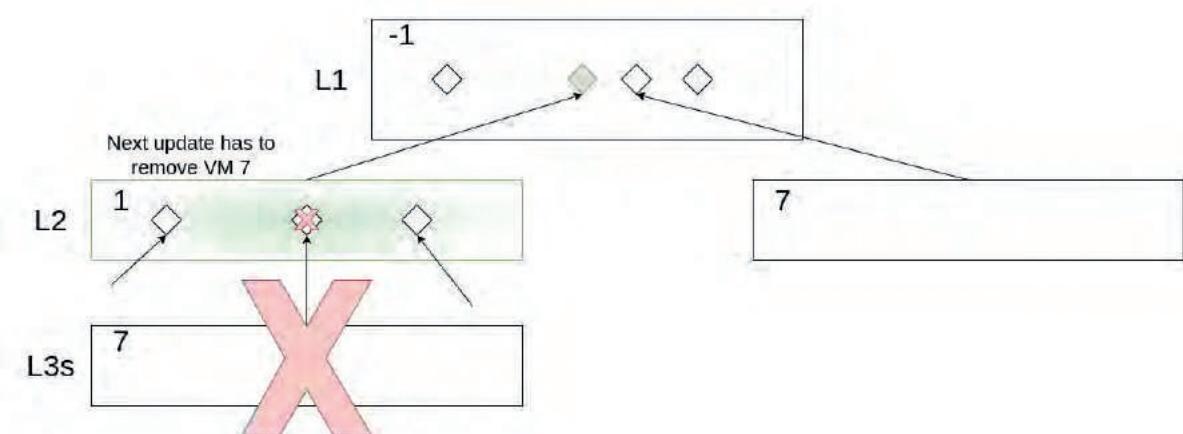


Fig. 11. Post exit update

Data challenges

There are some technical problems however if we apply this same mechanism for L2 validiums. In order to construct the proof for exiting, some amount of data has to be published on L1 so that our proofs can be verified. For example the hash of the smart contracts might be published with the ZKP of L2 on L1. Then it is easy to exit the L2 with a smart contract, we just need to know all the data of our smart contract to compose the proof. This is unfortunately untenable for validiums, putting every smart contract's data onto L1 would go against the definition of a validium.

At the other end of the spectrum we can put the minimal amount of data possible onto L1, namely the state hash of the L2. Then, if we have all the data of L2, we can also compose a proof which shows that our smart contract is in fact part of the state hash of the L2, via a Merkle tree. For this to work, we need to know the state of L2, so that we can prove that our smart contract was part of it.

So here we have three options: either the validium is only partially trustless compared to L1 (freezing the updates), we need to push data on L1 in a way that only scales with L1, or we need to solve data availability problems some other way. There is a cheaper version of solving the DA question, that we will see in the next section when discussing the other solution for L2 validium security. This solution ties the L2 validium's consensus mechanism together with its DA.

Security via L2 consensus mechanism

Starknet is currently aiming for decentralising the operator role (=sequencer and prover) of the rollup. This would enable censorship resistance. In addition to this, when a new ZKP is pushed to L1, the L2 state difference is also pushed as call data to L1. This solves the data availability question for the L2, as the L1 nodes can query the events on L1 for the state differences. If all the L2 nodes fail, then we can reconstruct the L2 state from the past events on L1. This allows the L2 consensus system to be permissionless. This is also relatively cheap, as call data is cheap on L1.

However, this solution “only works once”, it does not scale to iterated rollups, we will again have the data availability problem. If we have an L3 running on the L2, and both the L2 and L3 operators fail at the same time, then either the L3 state diffs also have to be pushed to L1 (in which case we the solution worked once, and also does not scale as we have to push more and more data to L1 as we add more and more

layers (meaning we scale with L1s DA capacity)), or if they are only pushed to L2 then they are lost as the L2 events are lost, we only save the state on L1.

This means that we run into issues of data availability when trying to secure this consensus mechanism for multiple rollups. The similarities with the previous solution are clear, we either have to push the L3's data to L1 as call data, or we have to find some other way to make the data available to the child rollup nodes. **What this really means is that with a permissionless consensus mechanism, the question of rollup/validium security is actually the question of data availability.**

As we can see, neither pushing to L1 storage nor pushing to L1 calldata is a solution that scales well for iterated validiums. **For iterated validiums, we need to address the DA question for the children validiums head on, without relying on L1.**

The challenge of iterated Security

As we have seen, there are two options for solving security, pushing data up to L1, or making the data available to the children nodes. Starknet is aiming for a blend of the two solutions, pushing data up to L1 as call data. This is the data availability solution, as this calldata cannot be used on L1 for security, but due to L1 security, the data is available for everyone. However it inherits a bad property of the “pushing up data” solution, namely that it does not scale well for iterated validiums. But this is necessary if you want to rely only on the L1's security.

Here we can make a tradeoff: we do not need the L2 validium to purely rely on the L1's security. Not relying purely on the L1's security means that L1 nodes' relationship to the L2 will not be trustless, they will not have access to the L2 state in case of failure. We will instead rely on another consensus mechanism to solve the DA question. This will mean additional trust assumptions in the L2's new consensus mechanism, but if we can solve the security of iterated validiums, then this is worth it. The nodes we can include in this consensus must include the L2 nodes, and descendant validiums i.e. L3, L4 nodes. What would be the best way of guaranteeing data availability for them?

Challenges

To solve the L2 DA question for L2 validium nodes would be easy, after all they are L2 nodes, so a PoS consensus mechanism would be enough. The children nodes are harder. Let's sketch the problems.

- The descendant nodes need to receive DA for the L2 state, even though they are not interested in every smart contract on the L2 state.

- The L2 nodes should also be able to update the L2 state without the child nodes permission.
- If the L2 nodes update, but do not send the new data to the descendant nodes, the descendant nodes still need to exit based on the state for which they did have DA.
- This means some evidence of older L2 states should remain on L1 if there are DA problems for the descendant nodes and the child nodes want to use the previous L2 state hash to exit.
- However we do not want to store all the previous L2 state hashes on L1, as that would cause state bloat.
- The descendant nodes should not be able to exit the L2 using arbitrarily old L2 states, as exiting means reversing the potential updates of the exiting validium, as they might exit with an outdated state.

We can solve all of this if the descendant nodes have some decision power over the old L2 state hashes on L1. If a majority of them have DA for a state hash, they need to be able to signal this to the L1. We only keep the last state hash for which a majority of them DA, the older ones should be erased, to save space on L1. But another and more important reason for deleting the older hashes is that the descendant validiums need to have a finality time, when they can no longer exit L2 using a very old hash, and thus revert their state to what it was in that hash. This also means, that child validium states are considered finalised not when it reaches L1, but when a majority of the descendant nodes have DA for it in a provable way.

The descendant nodes need to get DA sometimes, they can do this before they submit updates. The exits for child nodes only come into play in emergencies when they try to update and get DA, but find that it is not provided. Then they can exit using the old state on L1. It might happen that they do not have DA for the old hash on L1, as a majority of the descendant nodes could have updated since the last time our specific validium did. This is not a problem, as if we have an honest majority among the validium nodes, then at least one of the honest validiums does have the data associated with the current old hash (and will share that data).

The mechanism

We can create a consensus mechanism based on these principles. The consensus mechanism works as a normal PoS for the L2 nodes. When a child validium wants to update its ZKP, it needs to ask for the L2 state. If they receive it, then they have DA, they store it in case they need to exit the L2. After receiving it, they can push the

ZKP update. This means that we can tie together ZKP updates and DA, the child nodes will only update their ZKP if they have DA. This method allows us to “measure” the DA for descendant nodes, and only consider an L2 hash on L1 accepted, if enough descendant nodes have updated onto it. This means that there have to be multiple L2 hashes on L1, the oldest hash being the last one that the majority of descendant nodes have accepted.

How does this measurement happen? Similarly to the L1, on each validium we have to store for each child ZKP verifier contract different hashes of the child validium state, some older and some newer. Each of these hashes represents a point in time. (For synchronising time we will use the L1’s blocknumber.) For each such hash, (which represents a time and the corresponding state on the child validium), we will have a number pair (a, b) where “a” gives us how many of descendant nodes have pushed an update since that time, and “b” gives us the total number of descendant nodes. Here “a” changes with each child ZKP update, while “b” is normally constant. We will calculate the validium’s pair based on the child validiums’ pairs. For the validium each state corresponds to a time (=L1 blocknumber), similarly to the child validiums. When pushing an update the corresponding state hash will have acceptance ratio (0, D+1) (where D is the total number of descendants), as no child has pushed a state later than it has, and the validium has also not updated after that hash (you get DA of a state after you push the *next* update, so the initial number is 0). After that each ZKP update changes the acceptance rate of that hash (and new hashes are also added). We calculate the new acceptance rate for when the child validiums have pairs (a₁, b₁), (a₂, b₂), (a₃, b₃) as (a₁+ a₂ +a₃+1, b₁ +b₂ +b₃+1=D+1) for times later than our hash. When this acceptance ratio reaches 50% percent, we do not have to store earlier hashes for exits on the validium itself.

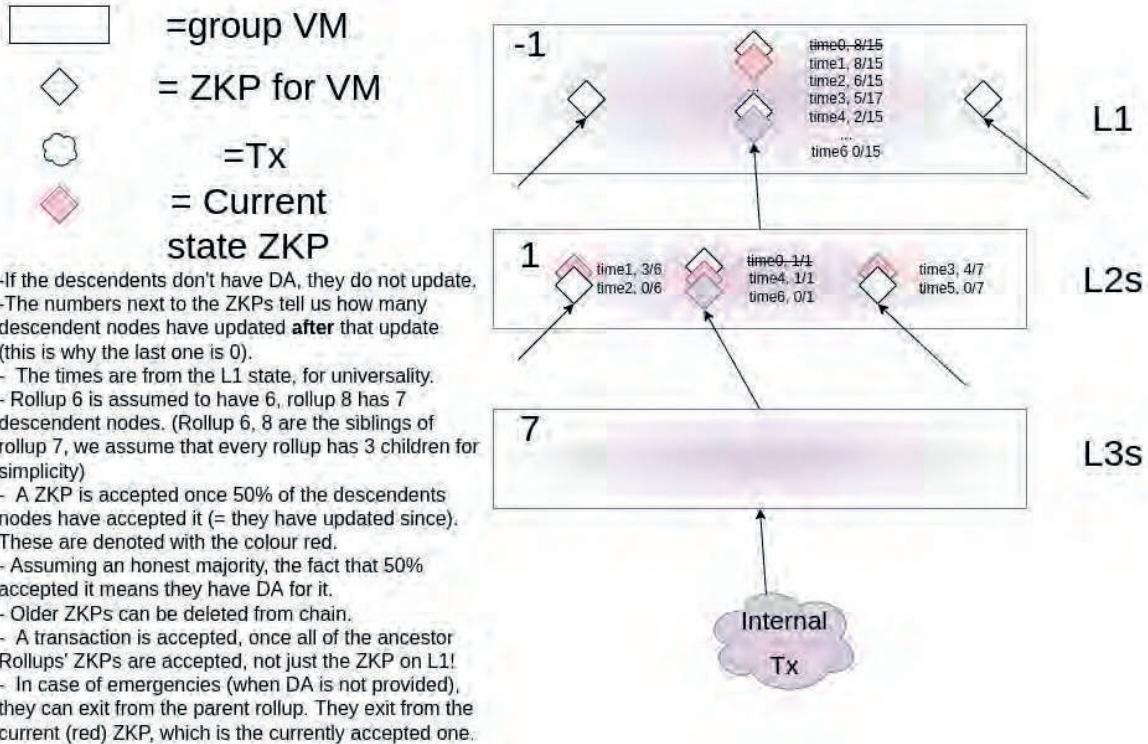


Fig. 12. Measuring descendant acceptance

(Note: all validiums have equal weight in the above calculation. This means that when deploying a validium, it has to be given permission. Otherwise there could be a sybil attack via creating arbitrary number of validiums. This permission for simplicity can be via a staking mechanism. Other weighting solutions are possible which do not require permission, but those require different trust requirements, i.e. more trust in nodes that are higher up in the tree.)

What happens when L2 nodes stop sharing data with their descendants, let's say L3 nodes, assuming that a majority of the L3 nodes are honest? The L3 nodes have access to some previous L2 state, and this state has a hash that is still on L1. The L3 nodes can use this hash to exit the L2, and to move their validium to someplace else in the fractal tree.

With this exit mechanism we will not have data build up on L1. The data that is required to be stored on L1 are the old hashes of L2. If we assume that the validiums are updating at constant pace (e.g. 1 block / 15 secs), then the amount of time it takes for a majority of validiums to confirm an update is $O(d)=O(\log(N))$ ($O(d)=O(\log(N))$, where N total number of validiums, d is the average depth). So the number of hashes stored on L1 will also be proportional to this. Where we will actually have substantial data build up is the bottom, each node

has to record the data of each of its ancestor validiums, which means $O(d)O(d)$ data at depth d (this data is needed for the exits).

Liquid consensus

There can be another way of exiting the L2 validium. After all, if the L2 nodes are malicious, then they ruin the tree for all the L3, L4... validiums. This means that a minority (L2 nodes) of all the validium nodes ruin the L2 for all the descendant nodes. Worse, all the descendants have to exit separately! It would be much easier for the majority of the descendant nodes, if they could simply replace the L2 nodes with new ones. (This means changing some public keys which specify who can push updates onto L1.) For the L2 validium this merely means the L2 nodes will have to exit with their state on L1, instead of all the descendant nodes.

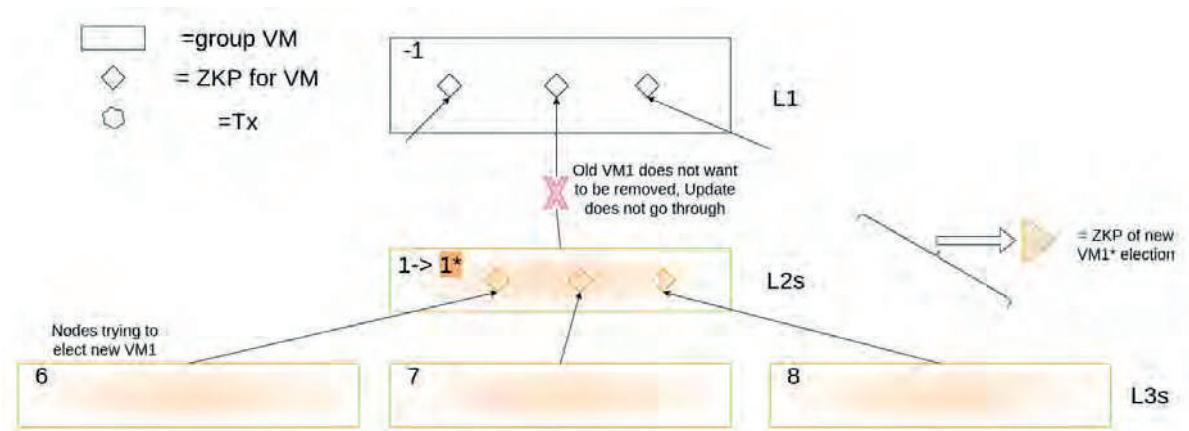


Fig. 13. Replacing L2 more is more efficient than exiting, but the simple method does not work

In this scenario if the L3 nodes do not receive the data from the L2 nodes before updating, they need to tell other L3 validiums. These validiums nodes verify that DA is compromised. A majority of the L3 nodes need to be able to take “emergency control” of the L2’s ZKP. To prevent anybody from taking “emergency control” we could store each L3 nodes public key on L1. This does not scale when iterating, we would have to store the L4, L5 keys as well. To solve this in a linearly scalable way we can use the restricting factor that the L3 ZKPs cannot be updated by anybody, only the L3 nodes (or their descendants, if they take emergency control). So here the L3 ZKP verifier smart contracts are not just any smart contracts, but are in a special position in the consensus mechanism of the L2.

So after the L3 nodes realise that the L2 node is not providing DA for the latest L2 state on L1, they take the old L2 state for which they have DA and is on L1. They

can each push a new ZKP to this L2 state, which “elects” new L2 nodes (changes the public keys). The new L2 nodes (who were just elected) then can receive the old L2 state and new ZKP’s which elect them, do the proving, and push the new L2 ZKP to L1. (Here the L2 hash on L1 is “forked” as the L2 state that was pushed to L1 by the previous nodes (the state that we did not have DA for), was also on L1. However, this L2 state was not accepted by the majority of the descendant nodes, so this is not actually forking (an update is finalised once a majority of the descendant nodes have accepted it.))

For the old L2 nodes security is also guaranteed, after the emergency control happens, they still have the state of the L2 validium. So they can exit with that on L1.

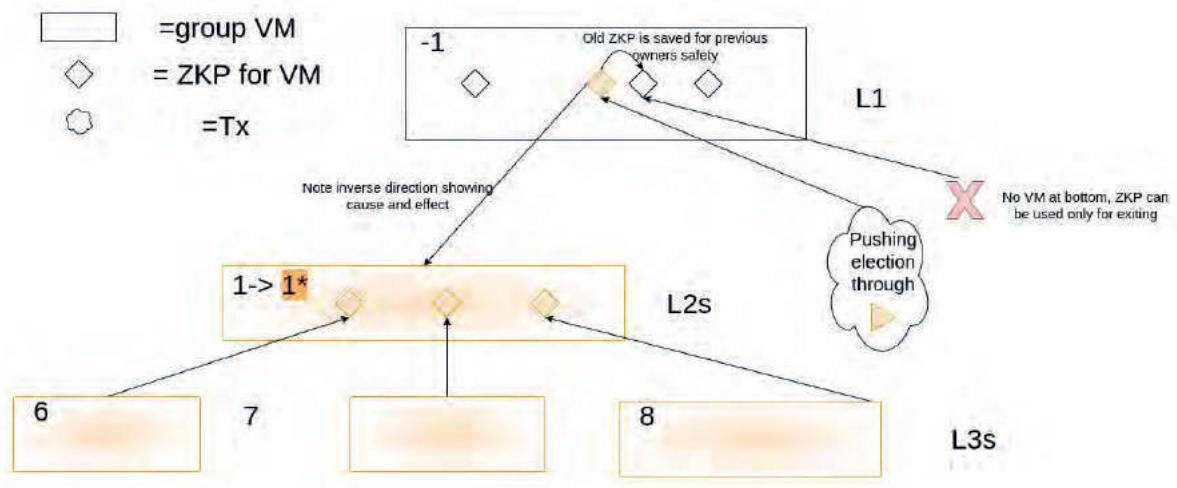


Fig. 14. A successful replacement of L2, circumventing the malicious parent node

(Note: There is one potential victim here, if every L2 node and the majority of L3 validiums are malicious, then a child node might not have access to the L2 state, and might be left out of the new emergency update as well. In this case their data is lost, and they cannot reclaim it on L1, as they do not have access to the L2 data. This is only a critical factor if we cannot trust the majority of validiums.)

Can this method work for iterated validiums? Yes it can, we can use the same principle for L4 nodes. When both the L3 and L2 nodes are malicious, but the L4 nodes are not, then the L4 nodes can first control and update the old L4’s ZKP on the old L3 state, and then the L3’s ZKPs on the old L2 state, with which they can control and update the L2’s ZKP on L1, which still has the old L2 state hash. So with this method, we can turn the secure exit strategy into a consensus system that relies on the majority of validiums being honest.

This is a logical consequence of the first proposal, after all we can think of the measurements as Stakers (descendant nodes) accepting a state update. The state updates are proposed by a single Proposer (the L2). If the Proposer misbehaves, a new one is chosen.

Replacing L1

Disclaimer: This part still needs further research. The broad outlines are clear, the precise simulation that stops forking quickly still needs careful analysis.

Up to this point we had an underlying L2 with a traditional (PoW/PoS) consensus system. This is fine, that is compatible with this system, we have the potential for parallel transactions between different validiums in a fractal tree that are efficiently secured by their common consensus mechanism. However, we have two separate consensus mechanisms here, the validium consensus and the PoS of L2. We can actually get rid of the PoS consensus mechanism on each level, after all the validium consensus mechanism already guarantees DA, and validity is verified by the verifier contracts, and forking is stopped by PoS on L1. This means that a single node can run a single layer, and be verified by all the descendant nodes.

A more ambitious goal is replacing the L1 PoS consensus mechanism with the common validium consensus mechanism. This would enable a single node to run L1, similar to each validium. For this we need to ask, what does the L1 PoS currently provide that the validium consensus does not? The answer is forking, validity is checked on each validium, but we use L1 to stop the forking of L2, and hence indirectly of the deeper layers as well.

To replace PoS with the validium consensus, we need to ensure that the L1 node cannot fork the L1 by itself (if a majority of the descendant validiums are also malicious, then forking cannot be avoided). What we would like is if we could make this validium consensus mechanism stronger, so that descendant validiums could stop forking at the L1 level. We already have a way to measure if the descendant validiums accept the parent validium's state, namely that they push a ZKP to it, and on each validium we can measure what percentage of the descendant validiums have accepted the state. However when discussing forking this is not enough, as the same ZKP can be published to multiple forks. We can stop this however, as the ZKP could also include a choice of the L1 state hash. When a validium receives ZKPs from its children, each child also gives it an L1 state hash that it accepts as the valid choice of history. If these state hashes are consistent (meaning that they follow each other in the L1 state), then the validium can prove this, and pick the latest one in time. Then they can include this latest hash (or perhaps even pick a later hash),

and include that with their own ZKP and pass that to their parent. When the L1 receives the different ZKPs and the included L1 hashes, it will not be able to include the ZKP into both of the forks, since the included hashes pick exactly one of the forks which it can be added to. At this point the L1 will not be able to produce a new valid block, so it is as if the node stopped working.

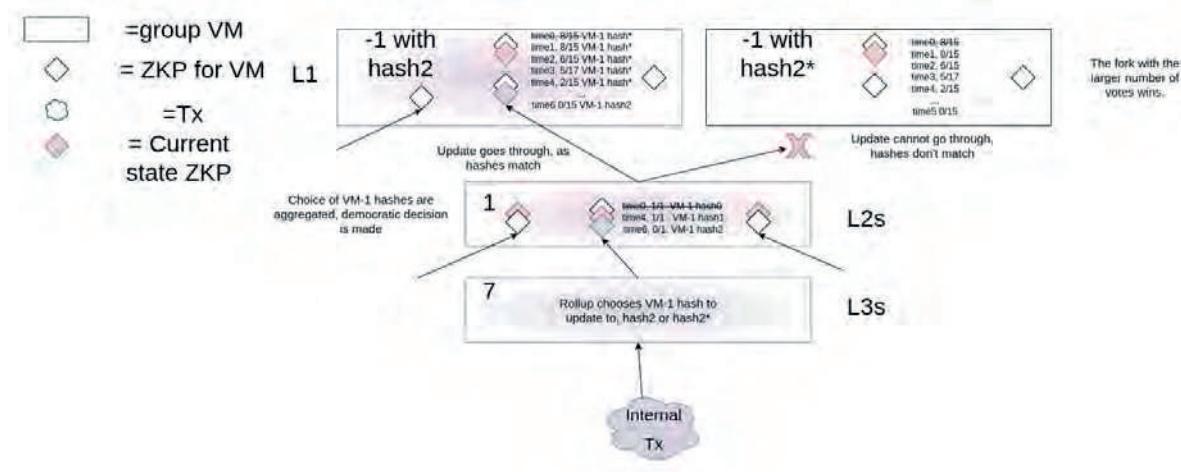


Fig. 15. Forking

With the traditional L1, it was implicitly guaranteed that there were always new blocks, here that is not the case. Fortunately we already have a method in the validium consensus that enables the descendant nodes to keep updating the system if a validium node stops updating, they take emergency control, elect a new node for the parent validium (remember each validium can be run by a single node), and the new parent node can publish a new ZKP. We can also do this here, each node could run a virtual L0 node solely to see whether the L1 node pushes valid updates or not (these updates also have to be accepted by enough descendant nodes, as described in the previous paragraph). If the L1 nodes does not push updates, then the L1 node is considered faulty/malicious, and a new one has to be elected by the L2 nodes (the old L1 node could also stake some tokens, which could be slashed for good measure). If the L2 nodes do not elect an L1 nodes, they are also considered faulty/malicious, and they need to be reelected as well.

So together the methods of fork choosing, checking for updates and elections enable the L1 to be run by a single node secured by the common validium consensus mechanism, such that the L1 state stops updating if the L1 node tries to fork it, and if the L1 state stops updating, then the children nodes can elect a new L1 node, and also punish the old one.

In this way, the traditional security model of L1-L2 can be reversed. It is no longer many L1 nodes that store the data and create the security for rollups (like in Danksharding, where the rollup data is posted to L1), but the many different validium nodes store and secure their ancestors data in the tree, and hence create the security for the L1.

Speed and efficiency and numbers

Quick summary of current result, we have a fractal scaling solution, each validium running on a single node, with trustless bridging directly between validiums, all of the validiums are secured with a common consensus mechanism, and L1 is also included in this consensus mechanism in such a way that forking is avoided.

What are the most important constraints for each node with this mechanism? We have two, the first is confirmation time, which is the time that it takes an update to be included into the L1 state, and for a majority of nodes to confirm that transaction (which means pushing their own ZKPs). The other constraint is the amount of data stored, as for the security of each validium, they have to store the data of each of the parent validiums. How do we make this efficient, decrease the confirmation time, and minimise the stored data?

Up to this point, we have not discussed the shape of the tree. For a given amount of N validiums, there are multiple ways of distributing them into a tree, but broadly speaking there are two big options, we can either have a wide and shallow tree, where each of the validiums have a large number of children, but the depth of the tree is small, or we can have deep and narrow tree, with each validium hosting a small number of other validiums.

In current implementations we have the wide and shallow option, as we had a relatively big L1 and big L2, as security was guaranteed by the L1 so iterated validiums were less secure, and there was no way to bridge funds directly between different validiums, so being close to L1 (where most of the activity happens) made sense.

In this new setting these problems do not matter, we have a common consensus mechanism, and direct bridging, so position in the tree is less relevant. This enables us to consider the deep and narrow shape for the tree. This is the efficient way to speed up confirmation times for the nodes, and to minimise stored data. (Check Quick Estimates for actual numbers.)

Another way of making the parents state updates quicker is by minimising each validium by emptying out its “own” state (the part of the VM that is not the ZKP of the

children nodes) into a separate ZKP and VM. In the pictures we show these VMs with circles, and the VMs of the actual validiums that only host ZKP-s are rectangles. This makes updates quicker, as the whole state does not need to be proved, only a ZKP of it. This also minimises the stored data, as only the ZKP has to be stored, not the state itself. This is already how we depicted the tree, with 4 ZKP-s in each group VM, 3 corresponding to the children's VM, and 1 for the state.

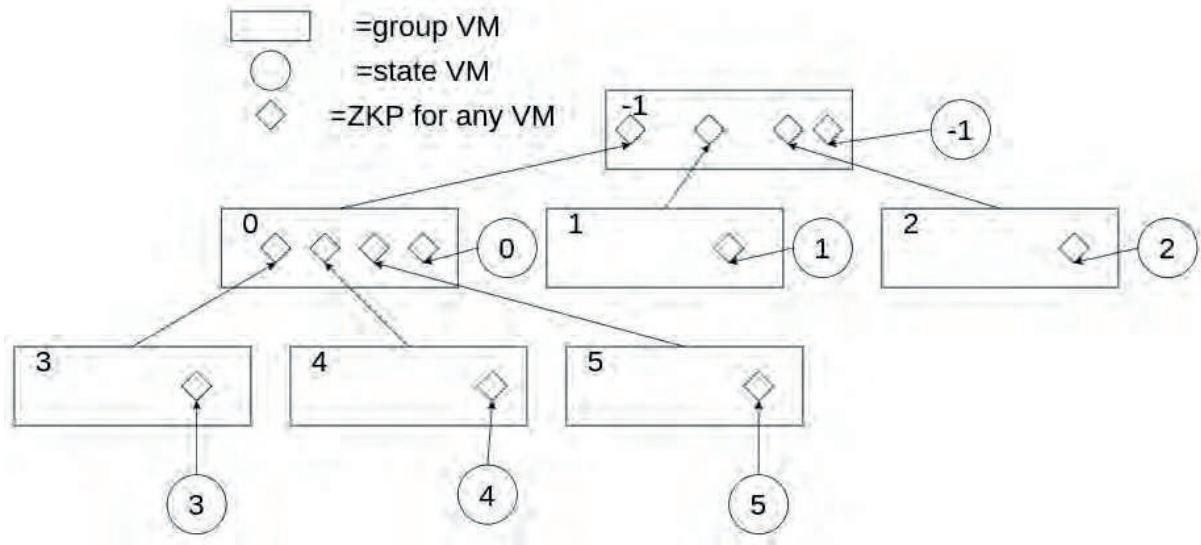


Fig. 16. The new layout

Once the tree has been narrowed down and nodes' states' size higher up in the tree have been minimised, it is possible for lower nodes to store their ancestor group VM-s. This means storing $4PS\log(n)=4PSd$ data for n nodes in the tree, with the tree being d deep, and PS being the average proof size. The confirmation time is also $4Cd$, where C is the time to prove the verification of a single ZKP.

Summary

In the fractal scaling of ZK rollups there will be a need for bridging between different rollups. We started with an overview of what the current solutions are, and their drawbacks. Then we described our bridging solution, and its benefits: it has a stable price and can be used to transfer any kind of data. Then we generalise this bridging, giving the sender rollup guarantees the transaction is received. This guarantee enables smart contracts to finish the bridging process, instead of just being able to start it. This means bridging can handle function calls between the rollups. This makes the different layers host different shards of a sharded virtual machine, which fundamentally relies on the security of L1. Then we look at the security requirements for fractal scaling, we find that the current solutions scale with the DA of L1. Then we

propose our own validium consensus mechanism which solves the security problem. Then we extend this so that the failure of a single layer does not mean every descendant node will have to exit. Then we extend this consensus mechanism to L1 such that forking will not be possible. Finally we look at the tree structure and ways to make the mechanism more efficient.

Quick estimates

1. If a tree with N nodes is $w > 3$ wide (each node has w children) with depth $d = \log_w(N)$, then if we refactor this tree to be $w' = 3$ wide, keeping the number of nodes $N' = N$, then the new depth becomes $d' = \log_3(N) = \log_w(N) * \log_3(w) = d * \log_3(w)$. If the proving time is linear with the number of children, then for an update to be processed by each parent node takes $(C * w * d)$ time (where C is some constant), which is worse than if the tree is narrow, as then this becomes $(C * 3 * \log_3(w) * d)$ (where we assume that C is the same constant). So it makes sense to make the tree narrow.
2. Also, as we can do transactions in this tree that do not touch L1, and the DA question is solved locally, we can have the “block size” (which we approximate by the total number of new transactions in a given period) growing linearly with the number of nodes, (compared to linearly with the root of number of nodes, as it is with Celestia). We have this linearity, because multiple transactions can happen in parallel between different nodes. This means that parallel transactions can be executed on different hardware. (There are existing attempts for parallelising transactions in Starknet as well, but there it is only a virtual separation, as all of the transactions will be proved by the same prover).f
3. However there is a downside compared to single layer blockchains, tx finalisation is not instantaneous. It takes $4 * C * d$ time to confirm a transaction, where C is the time required to prove the verification of one ZKP. As the sender node cannot send multiple transactions to other rollups in parallel, this time is important.
4. The previous points together gives our transaction throughput, which is the number of transactions / time to finality. This is $O(N/\log(N))$.

FAQ: How does this stop double spending on node 3? Double spending is stopped by the inner VM logic and the ZKP updates. You can only update the ZKP in the parent node by taking the previous VM state and ZKP proof, verifying that this ZKP proves the previous state, and applying a valid VM tx to the VM state, resulting in the result VM. Now we can take this whole computation, make a ZKP of it, and update

the previous ZKP. This method stops double spending, as only a single valid state transition is accepted on the parent node.

Unity is Strength: A Formalization of Cross-Domain Maximal Extractable Value

Vaibhav Chellani
Socket



Unity is Strength: A Formalization of Cross-Domain Maximal Extractable Value

Alexandre Obadia¹, Alejo Salles², Lakshman Sankar³, Tarun Chitra⁴, Vaibhav Chellani⁵, and Philip Daian⁶

^{1,2,6}Flashbots Research (*{alex, alejo, phil}@flashbots.net*)

³Ethereum Foundation (*lsankar4033@gmail.com*)

³Gauntlet (*tarun@gautlet.network*)

⁵Movr Network (*vaibhav@movr.network*)

December 7, 2021

Abstract

The multi-chain future is upon us. Modular architectures are coming to maturity across the ecosystem to scale bandwidth and throughput of cryptocurrency. One example of such is the Ethereum modular architecture, with its beacon chain, its execution chain, its Layer 2s, and soon its shards. These can all be thought as separate blockchains, heavily inter-connected with one another, and together forming an ecosystem.

In this work, we call each of these interconnected blockchains ‘domains’, and study the manifestation of Maximal Extractable Value (MEV, a generalization of “Miner Extractable Value”) across them. In other words, we investigate whether there exists extractable value that depends on the ordering of transactions in two or more domains jointly.

We first recall the definitions of Extractable and Maximal Extractable Value, before introducing a definition of Cross-Domain Maximal Extractable Value. We find that Cross-Domain MEV can be used to measure the incentive for transaction sequencers in different domains to collude with one another, and study the scenarios in which there exists such an incentive. We end the work with a list of negative externalities that might arise from cross-domain MEV extraction and lay out several open questions.

We note that the formalism in this work is a work-in-progress, and we hope that it can serve as the basis for formal analysis tools in the style of those presented in Clockwork Finance [1], as well as for discussion on how to mitigate the upcoming negative externalities of substantial cross-domain MEV.

1 Introduction

MEV was originally defined in [3] to study the effects of application-layer activity on consensus-level incentive perturbations, as well as to inform better application design. To understand how this concept of MEV may affect a cross-domain, multi-blockchain future, including how the incentives of this future may be threatened or destabilized, we must first define the concept of a *domain*.

1.1 What are domains?

definition. A *domain* is a self-contained system with a globally shared state. This state is mutated by various players through actions (often referred to as “transactions”), that execute in a shared execution environment’s semantics.

definition. A *sequencer* is an actor that can control the order of actions within a domain before they are executed, and thus influence future states of this domain.

Layer 1s, Layer 2s, side-chains, shards, centralized exchanges are all examples of domains. For the purpose of this work, there is no need to understand how these systems work, as we will abstract such details away.

Often, in a blockchain, transactions are applied sequentially to a domain's current state, according to the state transition function of the domain. Rather than transactions, we use the concept of actions in the rest of this work to represent their impact on the state to avoid confusion and force generality, as some domains, such as centralized exchanges, may experience state changes that are not effected by ACID-style transactions (Atomicity, Consistency, Isolation, and Durability).

definition. An *action* a is a mapping of a state to another state.

We write the effect of a sequence of actions $a_1 \dots a_n$ on a state s as:

$$s \xrightarrow{a_1 \dots a_n} s'$$

where s' is the state arrived at after the sequence of actions has been applied to s .

1.2 Motivation: multi-domain arbitrage

We will now motivate our definition by providing an example of cross-domain Extractable Value that exists in the wild. We will then extend the definition in [1] to capture the changes in economic incentives introduced by this cross-domain MEV, and use this example to illustrate the power and generality of our definition.

Applications such as automated market makers and lending markets are being instantiated on each new domain that sees the light of day.

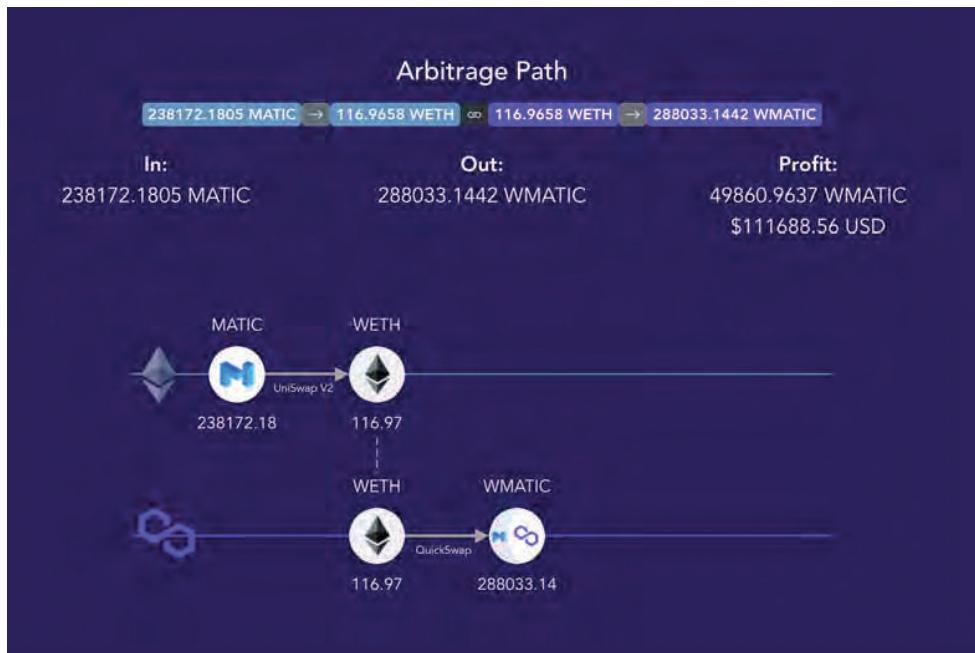


Figure 1: Example of 2-domain arbitrage between Ethereum and Polygon. Source: Westerngate¹

For example, the popular automated market maker Sushiswap with about \$3B weekly volume on Ethereum², started by existing only in Ethereum a year ago³, and now exists also on Xdai, Polygon, Fantom⁴, BSC, Arbitrum, Avalanche, Celo, Palm and Harmony⁵.

¹<https://westerngate.xyz/>

²<https://dune.xyz/nascent/SushiSwap>

³<https://github.com/sushiswap/sushiswap/commit/842679342047ecd0e0126f05b6089f8b727cb063>

⁴<https://twitter.com/josephdelong/status/1367226781393166336?s=20>

⁵<https://twitter.com/SushiSwap/status/1432987097783103496?s=20>

This means there likely exist liquidity pools representing the same asset pairs on each domain, yet with different volume, depth and activity. So, there will be a point in time where pools in different domains for the same asset pairs will be relatively imbalanced, creating an arbitrage opportunity.

Figure 1 shows an example of one such opportunity observed by a real-time arbitrage detection tool. In this instance, Uniswap V2 on Ethereum was offering the ability to swap 238172.18 MATIC for 116.97 WETH, which could then be swapped for 288033.14 WMATIC on Polygon. Any trader which values WMATIC and MATIC at the same 1:1 price ratio (a reasonable assumption, as MATIC and WMATIC can be exchanged at 1:1 over a bridge with a small time penalty and fee) would have profited 49860.9637 MATIC for bridging this price gap, a net profit of \$111,688 USD. Such opportunities will occur any time a user trades on *any AMM* without perfectly splitting their trade across all pools, a direct extension of the 2-AMM instability result in [1].

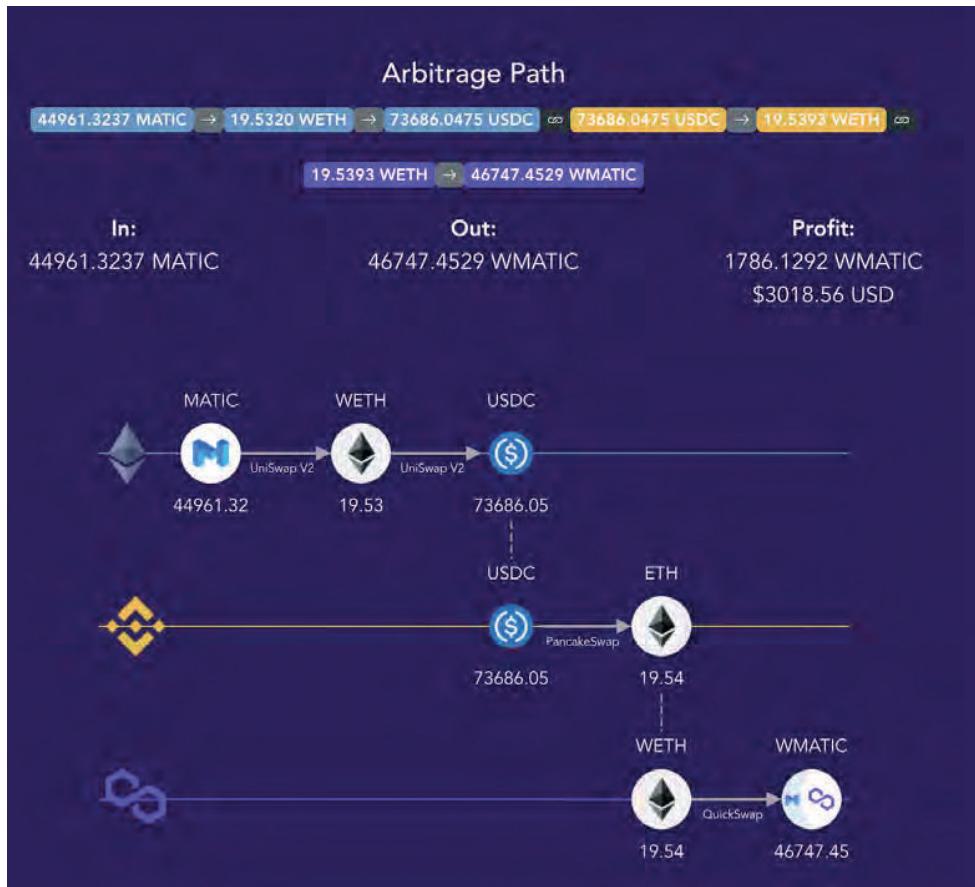


Figure 2: Example of 3-domain arbitrage between Ethereum, Binance Smart Chain, and Polygon. Source: Westerngate⁷

This is not limited to the 2-domain case: Figure 2 shows a similar opportunity identified between all three domains, resulting in a profit of \$3018.56. Unlike the 2-domain case, this extraction requires moving or bridging assets across three domains rather than two.

At first glance, these cross-domain opportunities look very similar to a pure-profit arbitrage opportunity that forms the basis for the classical notion of MEV in [3]. However, unlike in [3], the opportunity now depends on the ordering of transactions in multiple domains rather than a single one, the assets our trader starts with and ends with are different (Matic and WMatic), and the mechanics by which this opportunity is seized seems to involve transferring assets across domains, a non-atomic operation that breaks the atomicity described in [3].

⁷<https://westerngate.xyz/>

There are many substantive differences to untangle, each with their own implications on consensus protocol design, user fairness, and the economics of all blockchain protocols and their MEV ecosystems. We will now attempt to capture these differences in a formal definition.

2. Defining Cross-Domain MEV

Before introducing a definition of Cross-Domain Maximal Extractable Value, we first introduce the concepts of *reachable states*, *single-domain extractable value* and *single-domain maximal extractable value*. We re-use definitions proposed in [1].

2.1 Reachable States

Each domain has its own state transition function with specific validity rules. An example of a validity rule is that an account cannot spend more than it has. The set of all valid actions a player P can take represents P 's power in the MEV extraction game on that domain.

We define a set of actions A_P which represent all valid actions available to player P . We say a sequence of actions $a_1 \dots a_n \in A_P$ when $\forall i, a_i \in A$ (where A is the global set of all actions for all players in the protocol), and $\forall i, j, a_i \neq a_j$ (each action is allowed and unique). For example, pending transactions in the mempool able to be executed by a sequencer as described in [1] may each form an available action in L1 transaction sequencing.

For each sequence of allowed actions, there exists a state s' that will be the state of our domain in the future after each action is applied sequentially to the current state s .

All s 's together are the *reachable states* of our domain in the future under the influence of player P (S'_P), given its current state s and a set of actions. We define:

$$s \xrightarrow{A} S'_P$$

such that $S'_P = \{s' | s \xrightarrow{a_1, \dots, a_n \in A_P} s'\}$

In practice, the available set of actions A may depend on the initial world state s (for example, a validator may be able to include certain transactions only when s has enough balance to pay their fees). We will thus consider A to be defined as a function $A_P(s)$, but will omit this extra parameter for convenience in notation.

S'_P captures which future states are reachable given the behavior of a specific player P . In the remainder of this work, we will often omit P wherever it occurs in the notation for simplicity. In this case, we will assume P is a generic player with an arbitrary address, and must define the capital available to this address in s as well as the action space available to P in a specific domain (e.g. re-ordering transaction as in [1]).

2.2 Single-Domain Extractable Value

definition. *Extractable Value* is the value between one or more blocks accessible to any user in a domain, given any arbitrary re-ordering, insertion and censorship of pending or existing transactions.

More formally, and using the terms we introduced above, we define a user P 's *extractable value* (ev_i) in domain i after executing a sequence of actions $a_1 \dots a_n$ on an initial state s as follows:

$$ev_i(P, s, a_1 \dots a_n) = b_i(s', P) - b_i(s, P)$$

where $s \xrightarrow{a_1 \dots a_n} s'$ and $b_i(s, P)$ is the balance of a player P at a state s .

Note that we define the extractable value for a sequence of actions as the change in a player's balance across the state change between s and s' , where this state change results from executing this sequence of actions. Our subscript here indicates in which domain a player's balance change is effected (that is, in which domain

value is being extracted). This balance will likely be a numerical entry stored in a state, making the balance function for most domains a simple state lookup.

We note that in the original MEV definition in [1], miner-extractable value is provided as an upper-bound, and therefore considers actors P that are miners in Proof-of-Work Ethereum, the most privileged actors in the system. However, we transition below and throughout this paper to the notion of *maximal* rather than miner-extractable value, to generalize to non-PoW chains and non-blockchain domains. Rather than considering specifically the action space of a special sequencer (or miner) that can reorder, insert, or censor, as in [1], we generalize our definition to allow custom action spaces for each player.

2.3 Single-Domain Maximal Extractable Value

definition. *Maximal Extractable Value* (MEV) is the maximal value extractable between one or more blocks, given any arbitrary re-ordering, insertion or censorship of pending or existing transactions.

We can then simply take the maximum of our definition above over all valid sequences of actions for player P , essentially looking for the reachable state where our balance is maximized.

$$mev_i^j(P, s) = \max_{a_1 \dots a_n \in A_j} \{ev_i(P, s, a_1 \dots a_n)\}$$

Note that we introduced a new superscript, which binds the action set available to the player (in which domains the player can act). mev_i^j therefore can be interpreted as the MEV that can be extracted from a balance change in domain i , given actions in domain j . The single-domain case is therefore the case when $i = j$, and the player extracting value acts in the same domain in which the value is also extracted:

$$mev_i^i(P, s) = \max_{a_1 \dots a_n \in A_i} \{ev_i(P, s, a_1 \dots a_n)\}$$

It is however also useful to reason about cases where $i \neq j$. In our model, state is not domain-specific, and domains can read and write to each others' states if allowed to in the action space. One example of this would be any Layer 2 (L2) protocol, which reads ETH state to check for deposits and fraud proofs. If we thus consider the case where i is a Layer 2 protocol and j is Ethereum, we can ask questions such as "what is the maximum one can extract inside the domain of the L2 through manipulating L1 state only", which may be useful for quantifying and isolating the economic effect of cross-domain state interactions.

Note also that there is no strict or formal distinction in the state space of our two domains. Our definition models the state of the world as a monolithic entity, considering a moment in time in which a player P is able to act and affect one or both domains. We model the separation between domains and their trust guarantees by restricting the action space. For example, applying mempool transactions on ETH (a set of actions) will naturally affect only ETH-native state, scoping the influence of a particular player to a particular domain. We intentionally allow for fuzzier distinctions in actions that affect multiple domains simultaneously, to allow for modeling cross-chain communication protocols, bridges, and other interactions between domains as their own actions that act simultaneously on multiple domains.

2.4 Cross-Domain Maximal Extractable Value

2.4.1 Two-Domain MEV Let's consider now domains i and j , and for simplicity assume there is a single entity controlling the sequencing of both domains for the moment we're considering (generalizing MEV to include a notion of time beyond the instantaneous is left to future work).

What that entity is informally looking for is to maximize its balance across both domains. Otherwise stated, given its account balance on each domain b_i and b_j , what are the sequences of actions in A_i and A_j that will give the entity the highest sum $(b_i + b_j)$?

More formally, we define the cross-domain maximal extractable value of domains i, j as follows:

$$mev_{i,j}^{i,j}(P, s) = \max_{a_1 \dots a_n \in A_i \cup A_j} \{ev_i(P, s, a_1 \dots a_n) + p_{j \rightarrow i}(ev_j(P, s, a_1 \dots a_n))\}$$

The key insight here is that what happens after a state change in domain j is that the set of *reachable states* in domain i might change. A simple example is how a user's deposit in an L1 might unlock funds in an L2, allowing now a new state in the L2 where the user's balance is larger. We model this by allowing these domains to share a single state, and to access each other's state. While most mempool actions will act only on the state of that domain, some actions, such as bridging funds, may cause the state of one domain to affect the state of another.

Note that we introduce a pricing function $p_{j \rightarrow i}$, which helps us translate native balances across different domains. We compute the balance in domain j in units of the native asset of domain i by simply multiplying it by this pricing function. This is required as to meaningfully add the balances in two domains, some notion of their equivalent conversion prices must exist.

In our previous examples in Figures 1,2, we would assume a 1:1 pricing of MATIC and WMATIC, with an action space on Ethereum including performing a Uniswap transaction, and an action space on Polygon/BSC including bridging ETH into the equivalent wrapped assets as well as performing Uniswap trades on each domain. In Figure 1, the balance on Ethereum would decrease by 238172.18 MATIC, and the balance on Polygon would increase by 288033.14 WMATIC. Plugging into the above formula, we obtain an Extractable Value of $-238172.18 + (1.0)288033.14 = 49860$, as in the example. Interestingly enough, we note that even if we value WMATIC at a significant discount to MATIC, say at 90%, due to bridging costs, an MEV opportunity still exists. In this case, the total profit will be $-238172.18 + (0.9) * 288033.14 = 21057$ MATIC, still a tidy profit of approximately \$35,600.

We further assume that $p_{i \rightarrow j} = \frac{1}{p_{j \rightarrow i}}$. This means that exchange rate of inverse exchanges are multiplicative inverses, and allows us to simplify the above definition by removing a corresponding term inside our maximization where profit is denominated in asset j . While this property is guaranteed if assets share a single global orderbook, it may not hold for all decentralized assets. More complex pricing curves may be considered as part of future work, including pricing models that are dependent on quantity exchanged, modeling supply/demand under complex liquidity conditions with higher accuracy. We also assume $p_{i \rightarrow i} = 1$, that is that there is no cost to convert an asset into itself.

2.4.2 generalizing We can generalize our 2-domain definition to a n-domain definition, looking at a player P that has access to an action space representing abilities across multiple domains $A = A_1 \cup A_2 \cup \dots \cup A_n$, and that wants to maximize balances across domains $B = B_1 \cup B_2 \cup \dots \cup B_n$ with respective pricing functions of each asset priced in domain B_1 as $p_{B_1 \rightarrow B_1}, \dots, p_{B_n \rightarrow B_1}$ ⁸:

$$mev_B^A(P, s) = \max_{a_1 \dots a_n \in A} \left\{ \sum_{b \in B} p_{b \rightarrow B_1}(ev_b(P, s, a_1 \dots a_n)) \right\}$$

We can see that the MEV is the maximum of the sum of final balances across all considered domains into a single base asset (canonically the first domain considered), when some mix of actions across all those domains are executed together.

We can also reason about the total MEV available to extract in the world, by allowing A to represent the joint action-space of all domain sequencers (as well as any protocol bridges and the expected action-space of cross-domain communication infrastructure), and by allowing B to represent all assets on all domains.

Having defined cross-domain MEV with full generality, we will now reason about the logical implications of this definition on blockchain protocols, and prescribe important areas for future study that are likely to be impactful as a multi-chain future proliferates.

⁸ B_1 is chosen as the canonical base asset without loss of generality, due to the inverse property of exchange. Generalizing this definition to pricing functions without the inverse property would require summing across all possible base assets, or perhaps inventing a synthetic base asset which represents the relative desirability of each domain's asset.

3. Sequencer collusion

In our 2-domain case, we assumed for simplicity that a single entity controls the ordering on both domains. In reality, there will likely be different sequencers for each domain.

MEV extraction has historically been thought of as a self-contained process on a single domain, with a single actor (traditionally the miner) earning an atomic profit that serves as an implicit transaction fee. In a multi-chain future, extracting the maximum possible value from multiple domains will likely require collaboration, or collusion, of each domain's sequencers if action across multiple domains is required to maximize profit.

To analyze this, we introduce a term α , which represents the cost in the base asset for a set of multiple sequencers across multiple domains to collude. The incentive for two sequencers to collude can be split into three categories:

- if $mev_{i,j}^{i,j}(s) > mev_i^i(s) + mev_j^j(s) + \alpha$, then the sequencers make more value through collusion, as the benefit of collusion over acting independently outweighs the cost α .
- if $mev_{i,j}^{i,j}(s) = mev_i^i(s) + mev_j^j(s) + \alpha$, then there is no difference between colluding and not doing so.
- if $mev_{i,j}^{i,j}(s) < mev_i^i(s) + mev_j^j(s) + \alpha$, then the sequencers do not make more value by colluding.

Example. Suppose there exist two automated market makers with some (potentially different) on-chain pricing function: Uniswap and Toroswap. They are both markets between ETH and DAI. Uniswap is on domain i , and Toroswap is on domain j .

Suppose they have the same amount of liquidity and are both indicating a price of 20 DAI/ETH. Further suppose there is no other activity on each of these domains.

Let a single large buy-ETH transaction push the Uniswap market to 30 DAI/ETH.

Since Toroswap is still at 20 DAI/ETH, there exists an arbitrage opportunity between Uniswap and Toroswap, that will result in each pool indicating a price of 25. Further assume the profit from this arbitrage opportunity is 1 eth and that the cost of collusion α is 0.

In this example, we have A_i consisting of making a Uniswap trade, and A_j consisting of making a Toroswap trade. We will assume the player P has enough balance in each state to perform its choice of trade (aka that P 's balance exceeds the arbitrage opportunity). We now have that:

$$mev_i^i(s) = \max_{a_1 \dots a_n \in A_i} \{ev_i(s, a_1 \dots a_n)\} = 0$$

$$mev_j^j(s) = \max_{a_1 \dots a_n \in A_j} \{ev_j(s, a_1 \dots a_n)\} = 0$$

$$mev_{i,j}^{i,j}(s) = \max_{a_1 \dots a_n \in A_j \cup A_i} \{ev_i(s, a_1 \dots a_n) + p_{j \rightarrow i} ev_j(s, a_1 \dots a_n)\} = 1 \text{ eth}$$

In the last case, performing an arbitrage trade as described in the 2-AMM case in [1] results in a guaranteed profit, increasing a user's ETH balance if it can perform trades atomically across both AMMs.

So we've found an example of a case where the first inequality laid out above, $mev_{i,j}^{i,j}(s) > mev_i^i(s) + mev_j^j(s)$, holds. In Appendix B, we consider an illustrated example with 4 AMMs, two in i and two in j .

We expect that, given the deployment of AMMs and other MEV-laden technologies across multiple domains, the benefit of extracting MEV across multiple domains will often outweigh the cost of collusion α .

3.1 Trade Mechanics & Market Structure

So far, we've ignored the mechanics of actually seizing such an opportunity. Since it exists across domains and given it is finite, the competition for such opportunities will be fierce and it is likely no bridge will be fast enough to execute a complete arbitrage transaction as exemplified in Figure 1.

One observation is that a player that already has assets across both domains does not need to bridge funds to capture this MEV profit, reducing the time, complexity, and trust required in the transaction. This means

that cross-domain opportunities may be seized in two simultaneous transactions, with inventory management across many domains being internal to a player’s strategy to optimize their MEV rewards.

Such behavior is similar to the practice of inventory management that market makers and bridges in traditional finance do, which primarily consists of keeping assets scattered across multiple heterogeneous-trust domains (typically centralized exchanges), managing risks associated with these domains, and determining relative pricing. Some key differences include the ability to coordinate with actors in the system other than themselves, such as through a DAO or a system like Flashbots. However, given these conclusions, it is likely that traditional financial actors may have a knowledge-based advantage in cross-domain MEV risk management, which may induce centralization vectors that come from such actors being able to run more profitable validators.

Despite being grim, this fact is important as it reveals a key property that cross-domain interactions are subject to: the loss of composability. There is no more atomic execution. This introduces additional execution risk, as well as requires higher capital requirements, further raising the barriers-to-entry required to extract MEV. We expect bridges to play an extremely important role in such an MEV ecosystem, as the cheaper, more ubiquitous, and faster bridges become, the more competitive these arbitrage transactions naturally become by decreasing the inequality of the action space across players as a function of their capital.

3.2 Where does the cost of collusion come from?

Assuming the sequencers of i and j are distinct, they need a way to communicate and trust each other in order to apply the necessary orderings to their respective domains to maximize profit and split rewards between themselves.

From past results in [5]⁹, we know that cross-chain communication is impossible without either a trusted third party or a synchrony assumption other than asynchrony. Since ordering is time-sensitive, it is likely a synchrony assumption cannot be made and so a trusted third party is required to access these MEV opportunities. The presence of a trusted third-party can be thought about as an additional cost to facilitate cross-domain collusion.

However, if the sequencers of i and j are controlled by a single entity, then no trust is needed between them and this cost is now close to 0. This could create unwanted behaviours where entities will seek voting power across domains in order to bring their cost of cross-domain MEV extraction (α) down. This also means that colluding sequencers will naturally have an advantage over single-chain sequencers in the MEV market, as they will have access to a wider action space. It is worth noting that in practice, mechanisms like Flashbots or an SGX-based DAO may lower the cost of collusion even for single-chain sequencers. While these systems provide some promise to allow non-colluding sequencers to remain profitable, these systems require additional trust guarantees. If the profit from colluding is substantial, it is likely such collusion mechanisms will be used in practice, becoming de-facto defaults.

Another cost which we have not considered is the indirect cost of collusion. Even in the cases where $mev_{i,j}^{i,j}(s)$ is far greater than $mev_i^i(s) + mev_j^j(s) + \alpha$, there may exist social norms against colluding.

If sequencers were to infringe on these norms, one could imagine they could be punished by the community or that the domain’s token price could go down as trust in the domain’s fairness diminishes, which could impact the sequencer’s holdings. It may be possible to estimate this implicit cost by observing cases where entities have access to cross-domain MEV, and observing at which threshold of market growth they begin extraction activities.

4 Negative externalities

Cross-domain extractable value might surface new negative externalities the community should be aware of. While they warrant further study, we introduce some of them here:

Cross-domain sequencer centralization

⁹described in https://ls.mirror.xyz/5FM0KUurAEkN7yDXLAI8i9kCha_yMlzguVqiHEGMjPU

Cross-domain extractable value may create an incentive for sequencers (i.e. validators in most domains) to amass votes across the networks with the most extractable value.

This is especially relevant when realizing there already exist large validators and staking providers running infrastructure across many networks.¹⁰ It is unlikely that those who are for-profit entities will forego access to MEV revenue long term if such revenue is substantial.

Cross-domain time bandit attacks

Time-bandit attacks were first introduced in [3], and consist in looking at cases where the miner has a direct financial incentive to re-org the chain it is mining on.

In a cross-domain setting, there may now exist incentives to re-org multiple domains, or to re-org weaker domains in order to execute attacks resembling double-spends. This will be particularly relevant to the security of bridges.

Super-traders

While the risks above are worrisome, historically the negative externalities around extractable value have surfaced not from the sequencers' (miners) behaviour but from the dynamics that the pursuit of this value create in the markets.

One general worry is of the potential economies of scale, or moats, that a trader could create across domains which would end up increasing the barrier to entry for new entrants and enshrine existing players' dominance.

example. Suppose a domain orders transactions on a first-in first-out basis (FIFO). Such a domain effectively creates a latency race between traders going after the same opportunity. As seen in traditional markets, traders will likely invest in latency infrastructure in order to stay competitive, possibly reducing the efficiency of the market [2].

If several domains also have such ordering rules, traders could engage in a latency race in each of them, or a latency race to propagate arbitrage across domains, making the geographical points in which these systems operate targets for latency arbitrage. It is likely the infrastructure developed to optimize one domain, can be used across multiple domains. This could create a ‘super’ latency player, and certainly advantages entities which already have considerable expertise in building such systems in traditional finance. Such latency-sensitive systems may erode the security of systems that do not rely on latency, by advantaging latency-optimizing players in the cross-domain-MEV game. This area warrants substantial further study, as it is well-known that global network delays require relatively long block times in Nakamoto-style protocols to achieve security under asynchrony [4], and it is possible cross-domain-MEV may erode the fairness of validator rewards in such protocols.

Subjectivity of MEV

One important consequence of this definition is the introduction of heterogeneous pricing models across different players in the blockchain ecosystem. Previously, MEV was an unambiguous quantity denominated in a common base asset, ETH. In a multi-chain future, the relative price differences between actors is not only relevant in calculating MEV, it can in fact *create* MEV. For example, a previous validator may leave a system in what is in their pricing model a 0-MEV state, but the next validator, who disagrees with this pricing model, may see opportunities to rebalance MEV to increase assets it subjectively values more. This provides yet another intuition for why MEV is fundamental to global, permissionless systems even if they do not suffer from ordering manipulation of the kind described in [3].

5 Open questions

We end with a set of open questions. These are questions we're thinking about and looking to collaborate on.

How do we best define the action space?

¹⁰<https://www.stakingrewards.com/providers/>

Astute readers will note that we have punted many hard problems, including defining the boundaries between contracts on various domains, modeling the trust assumptions of oracles and bridges, and modeling probabilistic interactions, e.g. between centralized and decentralized exchanges. While some of this ambiguity is intentional, we believe it is also manageable. [1] provides one example for how the model in this paper can be instantiated in any transaction-based blockchain setting, including Ethereum, leaderless L1 protocols, Layer 2 protocols, and more. One natural application of this work is to extend the models therein to include other Ethereum-style domains, with action spaces defined similarly. This would prove sufficient for reasoning about much of the MEV we have explored in this work. We leave more complex modeling to future work, and would like this work to open discussion about what the most useful models would be in practice. We are optimistic that an open-source, mathematically formal, and executable set of models as proposed in [1] are a feasible path for the community to rigorously tackle MEV going forward.

Aside from cross-domain arbitrage, what are other forms of cross-domain MEV?

In the near future, there will be cross-domain smart contract calls enabling applications such as cross-domain lending and voting. Is there any new cross-domain extractable value that will be created from these applications? What about the extractable value created from bridges, oracles, governance, and other cross-domain systems?

What does a protocol for sequencer collusion look like and what are its desirable properties?

If cross-domain MEV extraction is inevitable, what does a good mechanism look like for it to be extracted? More concretely, how can two sequencers that do not trust each other share information about the state, and share profits from their collaboration, across domains with different finality and consensus mechanisms?

How can we identify and quantify cross-chain MEV extraction taking place?

While [Westerngate.xyz](https://westerngate.xyz)¹¹ does a great job at identifying potential cross-chain arbs, identifying historical extraction in practice seems a lot more difficult. How can we do so? If it is not possible, are we headed into a world where this activity is a lot more opaque than it has been so far?

What can we learn from existing distributed and parallel programming literature?

The concept of a domain extends beyond the world of cryptocurrency, and finds an analogue in classical parallel/distributed programming. The differences arise in the fact the systems we study need to consider adversarial behavior. Are there existing results from these subjects' literature that we can re-use or inspire ourselves of? In Appendix A we dive into this in further detail.

Acknowledgements

Thanks to Flashbots¹² for supporting and funding this research.

Thanks to Vitalik Buterin, Kushal Babel, Patrick McCorry and Georgios Konstantopoulos for commenting on versions of this work, and to the Flashbots crew for many stimulating conversations that lead to it.

References

- [1] Kushal Babel, Philip Daian, Mahimna Kelkar, and Ari Juels. Clockwork finance: Automated analysis of economic security in smart contracts. *arXiv preprint arXiv:2109.04347*, 2021.
- [2] Eric Budish, Peter Cramton, and John Shim. The high-frequency trading arms race: Frequent batch auctions as a market design response. *The Quarterly Journal of Economics*, 130(4):1547–1621, 2015.

¹¹<https://westerngate.xyz>

¹²<https://flashbots.net>

- [3] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 910–927. IEEE, 2020.
- [4] Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the blockchain protocol in asynchronous networks. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 643–673. Springer, 2017.
- [5] Alexei Zamyatin, Mustafa Al-Bassam, Dionysis Zindros, Eleftherios Kokoris-Kogias, Pedro Moreno-Sánchez, Aggelos Kiayias, and William J Knottenbelt. Sok: Communication across distributed ledgers. In *International Conference on Financial Cryptography and Data Security*, pages 3–36. Springer, 2021.

Appendix:

A. Comparison to classical parallel programming

The concept of a domain extends beyond the world of cryptocurrency, and finds an analogue in classical parallel/distributed programming.

In this Appendix, we briefly lay out the similarities and differences between traditional distributed systems concepts, and their analogues in censorship-resistant distributed systems. By doing so, our hope is that this will open up the questions here to a wider audience, who will be able to draw parallels with existing research from their field.

We will start by looking at the program execution model in Linux and map existing abstractions to domains.

In Linux, the default self-contained unit for executing transactions in a synchronous manner is a *thread*. Threads are created by a *process*, which manages the stored state and synchronization amongst different threads. The *kernel* manages the set of processes and handles scheduling process execution and mappings of virtualized state and bytecode to hardware.

Domains are most like processes in that they manage the shared state amongst a number of contracts. Individual contracts can be thought of as threads and the execution model of whether threads are run sequentially or in parallel depends on the host chain.

Example 1. In Solana, one can provide a dependency graph between contracts and contracts in different strongly connected components of this dependency graph are executed in parallel.

Example 2. On the other hand, in the current Ethereum Virtual Machine (EVM), contracts are always executed sequentially with the order determined by the sequencer.

In classical parallel/distributed programming, one often does not make a distinction between the process (domain) and the kernel that executes the process (sequencer). However, in censorship-resistant decentralized systems we are forced to separate these two to ensure that we minimize excess profit taken by the execution layer.

Part of the aim of this paper is to provide some insight into the case where there are multiple sequencers (e.g. multiple kernels) that are interacting.

Multiple domains (processes) communicate via a communication channel that sends messages whose validity can be quickly verified via a cryptographic hash or commitment. Cross-domain communication is analogous to futures and promises¹³ from traditional distributed programming, where one process or kernel submits a remote deferred computation and waits for a response. Again, in order to guarantee censorship resistance, we have to separate the actual message sent from the entity executing the relay of the message. One of the main differences between traditional deferred computation and what is found in cryptocurrencies is the excess overhead for verifying computation on both sides. We will provide a brief overview, but please read this review article¹⁴ for more details.

¹³https://en.wikipedia.org/wiki/Futures_and_promises

¹⁴<https://stonecoldpat.github.io/images/validatingbridges.pdf>

Currently, the best solutions in cryptocurrency for cross-domain communications are *bridges*. Bridges involve three principal agents:

- Sequencer of the source chain
- Sequencer of the destination chain
- Relayer of messages from the source and destination chain

The sequencer of the source chain produces the outbound transaction and verifies it in the bridge contract. The bridge contract effectively acts as the store of deferred computation (almost like a thunk¹⁵). The relayer runs blockchain clients for both the source and destination chain and upon finding a valid “forward” event from the source chain, submits a transaction on the destination chain. The sequencers of the destination chain validate the forward event in the receive side bridge contract. Upon being validated, applications can interact with the relayed message. In order to guarantee that a relayer cannot send an invalid message or censor a valid message, cryptographic and economic techniques are utilized.

One can view the communication complexity as similar to the synchronization cost that one naturally has to deal with in parallel programming. For instance, you might have n threads, but your algorithm needs \sqrt{n} queues/spinlocks to handle communication, so you only end up with a \sqrt{n} improvement. However, unlike the parallel programming scenario, there is now an adversarial component to how you distribute your computation and communication. One needs to adjust their threat model to include that adversaries will grief the user by forcing them to pay more than necessary or will increase latency dramatically.

B. 4-AMMs case example

While our 2-AMM example in section 3 above presents a simple case where the inequality holds. It does not consider any activity within each domain that could conflict with the realization of a cross-domain opportunity.

In this example, we show how for price inefficiencies for AMMs within and across domains, the inequality still holds.

example. Suppose there exist four market makers with some (potentially different) on-chain pricing function: Uniswap, Sushiswap, Toroswap and Unagiswap. They are all markets between ETH and DAI. The first two are on domain i , and the second two are on domain j .

Suppose they all have the same amount of liquidity and are all indicating a price of 20 DAI/ETH. Further suppose there is no activity on each of these domains aside from DEX trades and arbitrage trades.

Let a single large buy-ETH transaction tx_1^i push the Uniswap market to 30.

Since Sushiswap is still at 20 DAI/ETH, Uniswap can now be rebalanced through arbitrage to 25 via transactions tx_2^i and tx_3^i , so that Uniswap and Sushiswap pools will indicate a price of 25. Further assume the profit from this arbitrage opportunity is 1 eth and that the cost of collusion α is 0.

However, Toroswap and Unagi are still at 20, and so they can be further rebalanced through arbitrage between them and (Uni,Sushi) via transactions $tx_4^i, tx_5^i, tx_1^j, tx_2^j$, such that now all markets are at 22.5, netting an additional profit of 0.3 eth for each rebalancing.

In this example, we have A_i consisting of making a Uniswap trade, and A_j consisting of making a Toroswap trade. We will assume the player P has enough balance in each state to perform its choice of trade (aka that P 's balance exceeds the arbitrage opportunity). We now have that:

- tx_1^i : is an ETH-buy transaction on Uniswap in i
- tx_2^i : is a ETH-buy transaction on SushiSwap in i
- tx_3^i : is a ETH-sell transaction on Uniswap in i
- tx_4^i : an ETH-sell transaction on Uniswap in i
- tx_5^i : an ETH-sell transaction on Sushiswap in i

¹⁵<https://en.wikipedia.org/wiki/Thunk>

- tx_1^j : an ETH-buy transaction on UnagiSwap in j
- tx_2^j : an ETH-buy transaction on Toroswap in j

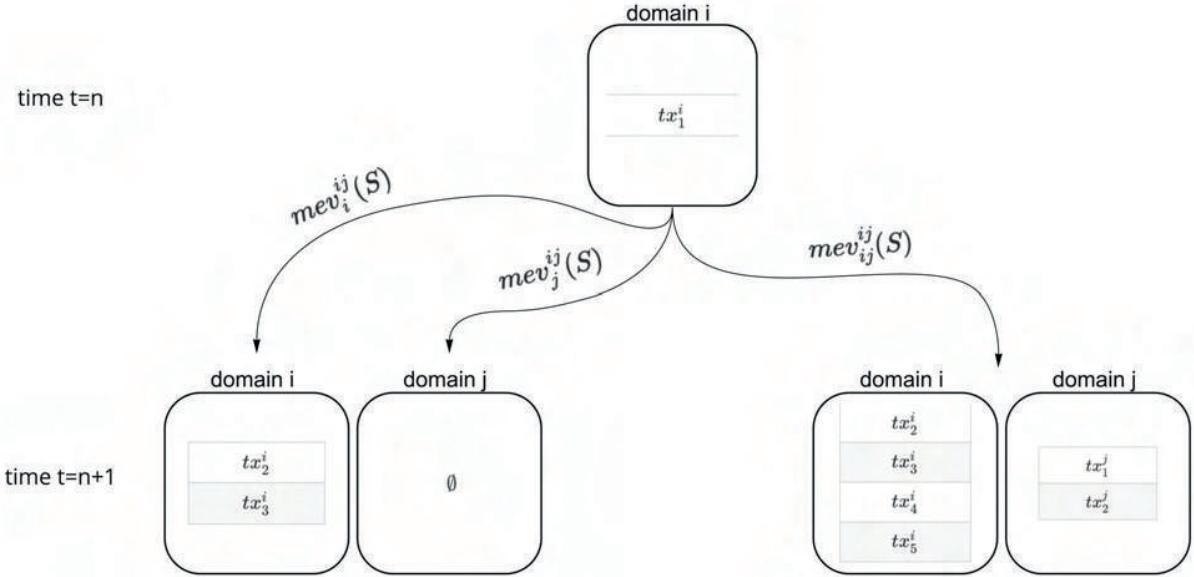


Figure 3: Example of multi-AMM multi-domain MEV under collusion. The left side shows each domain optimized for MEV individually, while the right side shows transaction optimization under sequencer collusion.

$$mev_i^i(s) = 1 \text{ eth}$$

$$mev_j^j(s) = 0 \text{ eth}$$

$$mev_{i,j}^{i,j}(s) = 1 + 0.3 + 0.3 = 1.6 \text{ eth}$$

so we've found a case where:

$$mev_{i,j}^{i,j}(s) > mev_i^i(s) + mev_j^j(s)$$

Caveats In addition, and again for clarity, our example considers next-block arbitrage rather than back-running, and ignores other transactions in each domain blocks that may be conflicting with each other.

While this example is simplified, one can see how this holds in more complex settings for cross-domain arbitrage. Since arbitrage is the bulk of MEV extraction today, one can see how such a scenario may happen regularly.

Assessing Blockchain Bridges

Joel John
Decentralised.co



Socket, Assessing Blockchain Bridges

Author: Joel John

Note: Fair warning. This piece is somewhat long and technical. I try to maintain narratives in what I write, but given the nature of what I am writing about today - there is ample jargon. Do excuse it. I'll make up for it in the coming piece.

Also, follow the team at [Socket.tech](#), who did most of the heavy lifting behind this piece. I have thoroughly enjoyed learning from them throughout this piece.

Last time, I wrote about what bridges are and their role in the evolution of Web3.

Unfortunately, a prominent bridge named Nomad got hacked for some \$190 million in the following days. This is quite a common occurrence. The [famous Ronin hack](#), which involved some \$700 million, was also one involving bridges. Bridges are the equivalent of banks; anyone, anywhere in the world, can break into them with a laptop. And that's why we need better tools to assess the reliability of bridges before we park more money there. Platforms like Coinmarketcap and DeFillama were crucial in growing the adoption of tokens and DeFi primitives. There needs to be an equivalent for bridges as they evolve.

I have been in touch with the team from Socket for quite some time. For those not in the know - Socket is a bridge aggregator. When an individual wants to go from, say, USDC on Solana to USDT on Avalanche, applications like Socket make it possible to find the best place to make that transfer happen. The challenge is when somebody doing a transfer wants to restrict how the capital is transferred. Say they want to ensure a transfer happens within ten minutes and is routed specifically through bridges with certain security practices. A framework like the one designed by Socket makes it easier for founders integrating bridges to quantify the quality of a bridge and route capital based on specific requirements.

Today's piece will look at a methodology to understand how standalone users can assess if a bridge is reliable or not. I am sharing this early framework to assess bridges so other collaborators interested in critiquing the model can reach out and help us iterate on it. The model suggested below will be integrated into a platform querying live data.

With that out, let's dig into what makes bridges reliable.

What makes great Bridges

A blockchain bridge is a financial service that runs at scale, powered by smart contracts. This attribute makes them somewhat similar to traditional fintech platforms like Paypal. Instead of humans enabling a transaction - logic and economic incentives are driving these systems. It helps us draw parallels to some attributes paramount to making bridges great. It boils down to

- Security - How secure your parked assets are on a bridge
- Connectivity - The number of networks a bridge is connected to
- Extractable value - The possibility of flashbots or other intermediaries extracting a portion of the transaction
- Performance - The economic model behind a bridge-related transaction

- Capability - The extent of assets supported by a bridge

Last we checked, there are close to 60 bridges supporting digital assets. We will likely see increasing amounts of specialisation. Some bridges will optimise for speed, while others will focus on the variety of assets they support. The framework developed by the team at Socket is reasonably broad, so some of your favourite bridges likely rank lower overall inspite of being one of the best at a single feature. To make it easier to read, I have broken down each section's parameters and given the max score that an auditor can allocate in a tabular format. We have stuck to using quantitative frameworks as much as possible, but given the nascent nature of the industry - some aspects are qualitative. Let's dig in.

1. Security

Category	Allocated Score	Max Score
Degree of Liveness Assumption	<ul style="list-style-type: none"> • Less than 2 hour challenge period - (1) • 2 hour to 1 day challenge period - (2) • 1 day challenge period - (3) • 1 day to 7 day challenge period - (4) • No such requirement - (5) 	
Validator Collusion	<ul style="list-style-type: none"> • Single actor can steal funds - (4) • 5 majority of validators can steal funds - (9) • 6 majority of validators can steal funds - (11) • Validators with dynamic selection schemes - (5) • Off-chain actors cannot steal funds - (5) 	5 + 5 + 7 = 20
Measures for Worst Case Scenarios	<ul style="list-style-type: none"> • Complete loss of funds - (1) • Can be restored if insured (6) • Can be lost but insured (1) • Frozen temporarily - (3) • Cannot be lost - (5) 	
Soundness of Code	<ul style="list-style-type: none"> • None - (-1) • Only code audits (0) • 1 code audit + Open bounties - (1) • > or = 2 code audits + Open bounties < \$1 mil - (3) • > or = 2 code audits + Open bounties greater than \$1 mil - (5) 	

We break down security into four key aspects. If I strip away the jargon - the degree of liveness assumption primarily checks how long a bridge has to dispute a transaction that could potentially be a hack. In the case of banks, there is no written law on how long they have to do AML/KYC on a transaction before it needs to be released. Smart contracts, on the contrary, need pre-denied parameters.

Bridges with longer dispute times are ranked higher as users know that a capital transaction could be stuck if validators on the network doubt something is o. A recent attack on Synapse was [aged by validators on the bridge](#) - eventually taking the whole system o. It helped the bridge save \$8 million overnight.

Ronin's \$600 million+ hack was one of the largest in the industry. It involved breaking into a senior engineer's computer with a [fake job offer](#) and replicating 5 of the 11 [validator's keys](#). The ideal

bridge is one where validators cannot access user funds. The framework we use proposes that single validators having access to tokens should be penalised, while those with validators holding no access to user funds would be ideal.

If a bridge does get hacked, the team can typically make users in one of two ways. One is through bridges acquiring insurance through DeFi primitives such as Nexus Mutual - and the other is through issuing native tokens of the bridge to users in proportion to the amount of capital they had.

The challenge with the latter approach is that users may immediately sell the native tokens they received, creating a ywheel where the bridge's native asset trends to zero. **The ideal bridge is one where pools of capital are kept aside - in a separate smart contract, through token incentives to make users whole in the event of a hack.** This would be somewhat similar to the [insurance funds](#) maintained by certain exchanges.

Lastly - under security, we observe the number of audits a bridge has had along with incentives for hackers to notify a bridge about the possibility of it being broken into. Audits on their own don't mean much. That's why we emphasise the need for multiple audits and bounties.

Bounties offered on open platforms like [Immune](#) are effectively open calls by teams to audit what they have built. Allocating large sums of money to open bounties can attract some of the brightest minds to check smart contracts for potential bugs and report them. Wormhole's [payout of \\$10 million](#) earlier this year is one of the largest bounties notifying about a potential bug. This is why - for the soundness of the code section, we have taken a mix of the count of audits and capital allocated as a measure for scores.

2. Performance

Category	Allocated Score	Max Score
Cost of Bridging	<ul style="list-style-type: none"> Increases Exponentially after \$10k - (1) Increases exponentially after \$100k - (2) Increases linearly after \$10k - (3) Increases linearly after \$100k - (4) Fixed fee regardless of the amount - (5) 	
Liquidity Rebalancing Needs	<ul style="list-style-type: none"> Not applicable - (+0) If required - (2) Required but has efficient rebalancing - (3) Not required - (5) 	5+5+5 = 15
Latency	<ul style="list-style-type: none"> 1hour to 7days - (1) 30 mins to 1 hour - (2) 10 mins to 30 mins - (3) 5 mins to 10 mins - (4) 2 mins to 5 mins - (5) 	

With most bridges we observed, the cost of switching USDC between networks is either a fixed amount (~1%) or free. The stable asset is routinely moved across chains for yield farming. The costs move exponentially for asset transfers involving cross-chain exchanges where an automated market-maker is involved. What does that mean? Say you are making a transfer of ETH from

Ethereum to USDC on Optimism. The fees you pay increase exponentially with the size of the assets involved.

This is because the liquidity for the exchange is sourced from an AMM pool where the cost of exchange increases exponentially. External factors such as the depth of the pool and how it is rebalanced that feed into this. Hashow, for instance, quotes prices directly from market-makers and is typically able to quote prices that are almost on par with exchanges for multi-million dollar asset exchanges.

We allocate a high score of 5 for pools that require no rebalancing and zeroed costs while penalising bridges with -1 each for not offering hop transactions and

high fees after a low barrier of \$10k. An added factor to consider here is the time taken for bridging. We penalise bridges that take north of 1 hour for a bridge while providing 5 to the ones that bridge under a minute. Finally, it is worth noting that some layer 1s like Ethereum may be at a disadvantage here due to longer confirmation times for blocks at times of high congestion.

3. Extractable Value

Category	Allocated Score	Max Score
MEV Leak	<ul style="list-style-type: none"> Low - 1 Medium - 2 High - 3 	
Censorship resistance and position on the permission spectrum	<ul style="list-style-type: none"> Permissioned and can be censored - (-1) Permissionless but can be censored - (0) Permissioned but cannot be censored - (1) Permissionless and cannot be censored - (2) 	3+2+5 = 10
Churn	<ul style="list-style-type: none"> Between 0 and 1 - (1) Between 1 and 5 - (2) Between 5 to 10 - (3) Between 10 to 100 - (4) Beyond 100 - (5) 	

An added layer of cost for the end user comes through MEV extraction. Again, without going into the specifics - it is when an individual can front-run a transaction occurring on-chain to book a small amount in profit. So far, [some \\$180 million](#) have been extracted as MEV revenue on Ethereum-based dexes alone. One way we could have quantified this metric is through the amount of capital that has gone through MEV extraction on a bridge.

However, high amounts of MEV extraction from a bridge could simply mean it is a highly used platform. Therefore, a qualitative scale has been given based on how hard it is to extract value from a bridge's transaction. It is worth noting that bridges that interact with chains that don't have MEV by default will rank higher here. Bridges building on chains with a high amount of MEV may choose to use [protective measures like Cowswap](#) - a DEX aggregator on Ethereum does today.

Given the extent of scrutiny, Tornado has come under, we believe bridges will be centre-points for sanctions in the future. Currently, sanctions have been done at the address level. At some point, we likely see entire networks, especially ones oriented towards privacy and shielding transactions,

being blacklisted. It is hard to quantify censorship resistance on a spectrum - so scoring here would be relative, with a maximum of 2 points given to permissionless and censorship-resistant bridges.

The last aspect we cover here is of capital churn. In my last piece, I mentioned that it is likely that we will see an increasing number of blockchain bridges optimised for lower capital requirements. **I define “capital churn” as the amount of capital owing through a bridge over 30 days, divided by the total value locked in it.** So, for example, certain bridges will have a billion dollars in TVL but enable only ~\$100 million in transactions over a month. In this case (\$100mil/\$1bil), a churn of 0.1 indicates bad capital efficiency.

Note: Given the number of chains involved, finding churn data for all bridges has been difficult. If you are analytical and want to build this using Covalent’s API - drop me a note.

On the other hand, there are bridges - like Hyphen and Hashow that have been doing billions in bridging with a capital requirement of just ~10 million. In this case, the churn is over 100 - and indicates that the system can put capital to complete use without leaving any of it idle. But, again, the metric is raw in that depending on how niche an asset is - and the demand for it, often, bridges will likely have idle assets by default.

4. Connectivities

Category	Allocated Score	Max Score
Types of Chains Supported	<ul style="list-style-type: none"> • Native bridges alone - 0 • Only cross L2 or cross L1 - (1) • Cross L2 and Cross L1 - (2) • Cross L2 + Cross L1 + Non-EVM Support - (3) • Cross L2 + Cross L1 + Non-EVM Support + Non-smart contract support - 4 	6+4= 10
Number of Chains Supported	<ul style="list-style-type: none"> • Two chains only - (1) • Max four chains (2) • Max Six chains (3) • Max eight chains (4) • Max 10 chains (5) • More than 10 chains- (6) 	

Connectivities look at the permutations and combinations in which a bridge can interact with different networks. A domain is a layer or network in which an asset is moved. Some bridges have deep liquidity pools focused only on EVM-based chains (ETH, Avax), while others optimise for the breadth of chains. We rank native bridges (like the one Polygon or Celo uses) the lowest as they are usually oriented towards inbound liquidity and limit user choices.

During the earliest stages of bridges, we used to see asset-specific transfers occurring at scale. Wrapped bitcoin moving from Bitcoin to Ethereum was a good example. The next step involved support towards and from L2 solutions like Optimism. The amount of capital owing between the likes of Solana, Avalanche and ETH native L2s has incentivised capital flow between them strongly.

We split the types and number of domains supported in the scoring system. Part of the reason for this is supporting multiple domain types (eg: L2, L1, EVM etc.) does not imply they can communicate with one another. In many instances, bridges restrict the flow of assets depending on their pool rebalancing mechanisms. The amount of capital in a bridge's TVL determines how assets can flow. Today's restricting factor is the effort needed to rebalance pools across EVM and layer types. The ideal bridge can instantaneously support the easy flow of assets across all the domain types they support.

5. Capabilities

Category	Allocated Score	Max Score
ERC-20 Support	<ul style="list-style-type: none"> • No Support - (-1) • Yes - (3) • Greater than ten assets - (5) 	
NFT Support	<ul style="list-style-type: none"> • No Support - (0) • Yes - (3) • Greater than ten assets - (5) 	5+5+5 = 15
Contract Calls	<ul style="list-style-type: none"> • No support - (0) • Support Enabled (5) 	

We end the scoring system with support for the types of assets supported and the number of assets. We emphasise ERC-20 support due to the high amount of DeFi and consumer applications built on Ethereum today. However, the number of assets supported is kept at ten. In my opinion, that is an arbitrary, low number. For instance, automated market-makers like Pancake swap already support tens of thousands of asset pairs. It is still early in the evolutionary arc of Bridges, in contrast.

We see the need for bridges to support multi-chain NFTs through the likes of OpenSea. Today's largest NFT marketplace already supports NFTs on Polygon, Ethereum and Solana. What if users wanted to port assets between those bridges? Or even better - shortly, we may see cross-chain NFT lending occurring. This would involve querying an asset's price in its most liquid market (eg: Ethereum), trading it through Polygon and taking the loan on Solana. Products like [Xp.network](#) have long been building towards this vision. We do not penalise a lack of NFT support in the scoring system.

The asset flow mentioned above will require the ability of a bridge to interact with a smart contract on the recipient chain. We define this as a "contract call". Today, applications like DeFiSaver allow users to bridge to optimism and take a loan on Aave in a single click. This makes it possible to create increasingly sophisticated primitives using the composability that historically allowed the DeFi ecosystem to grow into what it became. One instance of this playing out in the wild is [Connext's integration with Gelato network](#) last year.

Putting It All Together

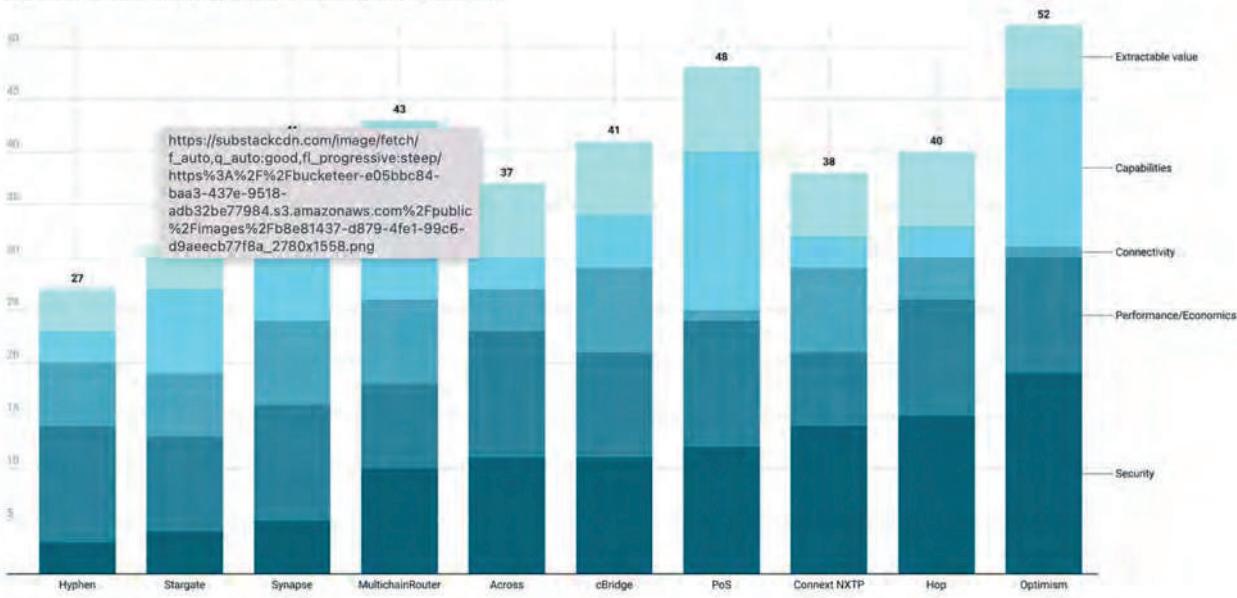
This framework, as it stands, is a theoretical approach to rating bridges. Its biggest flaw is that specific attributes are qualitative and require individuals with expertise to give a rating. Just like smart contract audits, the subjective opinions of individuals could be flawed. It also brings relative centralisation and incentive misalignment to the picture. That is why Socket.tech had reached out to L2beat and me to make this framework a collaborative effort. There will likely be multiple iterations before we have something deemed "ideal".

I do not anticipate individuals to use the framework to assess bridges. For the average person, trying to move assets between Solana and Ethereum - doing a point-based assessment is futile. Instead, I anticipate its usage with stand-alone platforms like DeFi Llama or [L2Beat](#). Providing information for users in a quantitative fashion that ranks bridges could simultaneously help bridges figure out where they lack and direct users towards better service providers.

We tried the scoring methodology on ten bridges to get an estimate of how they rank. For this scoring, we have given all bridges a standard score of (3) for churn. This is disadvantageous to a few bridges that specialise in capital efficiency, but we had to do it due to a lack of readily available data across all the bridges.

Bridge Scoring Framework

Based off the framework developed by Socket.tech in collaboration with Decentralised.co



Depending on who you ask, this chart is either a crime or a piece of work.

Part 3 | Wider Industry Research



Caulk: Lookup Arguments in Sublinear Time

Arantxa Zapico, Vitalik Buterin, Dmitry Khovratovich, Mary Maller, Anca Nitulescu, Mark Simkin,

Pompeu Fabra University
Ethereum Foundation
Protocol Labs



Caulk: Lookup Arguments in Sublinear Time

Arantxa Zapico^{*}¹, Vitalik Buterin², Dmitry Khovratovich², Mary Maller²,
Anca Nitulescu³, and Mark Simkin²

¹ Universitat Pompeu Fabra[†]

² Ethereum Foundation[‡]

³ Protocol Labs[§]

Abstract

We present *position-hiding linkability* for vector commitment schemes: one can prove in zero knowledge that one or m values that comprise commitment \mathbf{cm} all belong to the vector of size N committed to in \mathbf{C} . Our construction Caulk can be used for membership proofs and lookup arguments and outperforms all existing alternatives in prover time by orders of magnitude.

For both single- and multi-membership proofs the Caulk protocol beats SNARKed Merkle proofs by the factor of 100 even if the latter is instantiated with Poseidon hash. Asymptotically our prover needs $O(m^2 + m \log N)$ time to prove a batch of m openings, whereas proof size is $O(1)$ and verifier time is $O(\log(\log N))$.

As a lookup argument, Caulk is the first scheme with prover time sublinear in the table size, assuming $O(N \log N)$ preprocessing time and $O(N)$ storage. It can be used as a subprimitive in verifiable computation schemes in order to drastically decrease the lookup overhead.

Our scheme comes with a reference implementation and benchmarks.

1 Introduction

A vector commitment is a basic cryptographic scheme, which lies at the foundation of numerous constructions and protocols. In a nutshell, a vector commitment is a compact data structure that contains a potentially very large number of elements and allows *proving* that a specific element has been committed to it. A natural requirement is that a proof is *succinct* and *unforgeable*. A Merkle tree is a well-known example of a vector commitment.

For privacy-preserving applications it is vital to make proofs zero-knowledge, i.e. hiding the element that is asserted to be in the commitment, while still establishing a certain relationship, or *link*, to that element. A vector commitment to $\mathbf{c} = (c_1, \dots, c_N)$ is linkable, if it permits proving that you know a secret s_i mathematically linked to c_i . The simplest example is a proof of authorization where a party proves knowledge of a secret key belonging to one of multiple public keys in a set. A more elaborate example is a *proof of coin ownership* in private cryptocurrencies: coins are stored as hashes of a secret k and values v in a list or a tree and to spend v one proves knowledge of v and k without revealing them. A third example are lookup arguments in verifiable computation: prove that intermediate values a_1, a_2, \dots, a_m are all contained in a certain table, e.g., a table of all 16-bit numbers for the purpose of overflow checks in financial or mathematical computations. Other applications also include membership proofs, ring signatures, anonymous credentials and other schemes.

Currently, all of the above examples are being solved using heavy cryptography machinery involving significant computational overheads, which limits their scalability and adoption. The first version of

^{*}This work was done while Arantxa Zapico was an intern at the Ethereum Foundation.

[†]arantxa.zapico@upf.edu

[‡]{v_buterin, mary.maller, mark.simkin}@ethereum.org, khovratovich@gmail.com

[§]anca@protocol.ai

the Zcash cryptocurrency [30] used a SHA-2-based Merkle tree to store the coins and the Groth16 [20] SNARK to prove coin ownership. The relatively high costs of Groth16 and the large prime-field circuits of SHA-2 made the resulting prover time of 40 seconds barely usable in practice. Even the most recent developments of algebraic hashes [1, 19] reduce *prover time* by an order of magnitude only. Another application of concern, lookup tables, so far has required the generic construction of Plookup [17], that makes the prover be *at least as big* as the table itself, no matter how many values they look up.

1.1 Our Contributions

In this paper we present a novel construction, named **Caulk**, that allows to link a public set with a hidden subset in zero-knowledge and performs with unprecedented efficiency. We construct a proof of membership, with asymptotic complexity of $O(\log N)$ for N -sized commitments, with a concrete efficiency improvement of a factor of 100x over SNARKs on top of a Merkle trees that uses the Poseidon hash function. The prover benefits of our construction are even more extreme when compared with Merkle trees that use SHA-2. Our construction achieves statistical zero-knowledge and soundness in the algebraic group model, requires a universal setup, and $O(N)$ storage.

Our construction naturally extends to proof of subset memberships, thus leading the way to more efficient lookup arguments. We are the first to remove the bottleneck of big tables by achieving a $O(m \log N + m^2)$ prover cost for m -subvector lookups. The verifier is succinct as it requires only $O(\log(\log N))$ scalar operations as well as constant number of pairings to verify a constant-size proof. We envision the widespread deployment of our construction both in generic lookup-equipped proof systems [17, 27] and specific applications with membership proofs.

We have implemented **Caulk**¹ in Rust, and we use that implementation for concrete comparison with other solutions as well.

1.2 Paper Structure

We start with a technical overview of **Caulk** in Section 2 and related work is discussed in Section 3. In Section 4 we provide a self-contained description of the tools we use, in particular the polynomial commitment scheme by Kate, Zaverucha and Goldberg [22] (KZG) and associated precomputation techniques, which can be skipped by a knowledgeable reader.

In Section 5 we identify our constructions as special cases of a more general family of protocols that add a property that we call *position-hiding linkability* to vector commitment schemes. This primitive asserts that all (hidden) entries committed in an element \mathbf{cm} are also (publicly) committed to in \mathbf{C} . Position-hiding refers to the fact that no information about which elements were taken to construct \mathbf{cm} should be leaked. We formalize its definition as well as the security notions it should satisfy.

In Section 6 we formally describe **Caulk** for the case of proving membership of a single element ($m = 1$) and show that it is sound in the algebraic group model and statistically zero-knowledge. As an important building block we also present a construction of a proof system that demonstrates that a Pedersen commitment contains a root of unity. In Section 7 we extend **Caulk** even further to m -subset ($m > 1$) proofs, with some values possibly repeating. In this scenario **Caulk** can be seen as a lookup table, and is thus a prover efficient alternative to schemes such as Plookup [17]. We discuss various optimizations in Section 8.

Caulk comes with an open source reference implementation in Rust using arkworks library. In Section 9 we compare its efficiency with some rival schemes.

2 Caulk in a nutshell

In the following we explain the high-level ideas behind our constructions for the case of proving membership of a single element ($m = 1$) and the case of proving membership of multiple elements ($m > 1$). The starting point of both is the KZG polynomial commitment scheme, which we describe in Section 4.2, that allows for committing to a polynomial $C(X)$ and then later on opening evaluations $C(\alpha)$ for some publicly known α . We note that a vector \vec{c} can be encoded as a polynomial $C(X) = \sum_{i=1}^N c_i \lambda_i(X)$, where $\{\lambda_i(X)\}_{i=1}^N$ are the Lagrange interpolation polynomials corresponding to some set of roots of unity

¹<https://github.com/caulk-crypto/caulk>

$\mathbb{H} = \{1, \omega, \dots, \omega^{N-1}\}$ with $\omega^N = 1$. That is, $\lambda_i(\omega^{i-1}) = 1$ and $\lambda_i(\omega^j) = 0$ for all $j \neq i - 1$. Opening position i in the vector is done by simply revealing the corresponding evaluation of the polynomial at element ω^{i-1} .

A KZG commitment to $C(X)$ is an element $\mathbf{C} = \sum_{i=1}^N c_i [\lambda_i(x)]_1$ where x is secret and $[.]_1$ denotes it is given in the source group \mathbb{G}_1 of some (asymmetric) bilinear group. A proof of opening for value v at position i is an element $[Q_i]_1$ such that

$$e(\mathbf{C} - [v]_1, [1]_2) = e([Q_i]_1, [x - \omega^{i-1}]_2).$$

A proof of opening for a subset of positions $I \subset [N]$ is an element $[H_I]_1$, such that if $C_I(X) = \sum_{i \in I} c_i \tau_i(X)$ and $z_I(X) = \prod_{i \in I} (X - \omega^{i-1})$, where $\{\tau_i(X)\}_{i \in I}$ are the Lagrange interpolation polynomials of $\mathbb{H}_I = \{\omega^{i-1}\}_{i \in I}$, then

$$e(\mathbf{C} - [C_I(x)]_1, [1]_2) = e([H_I]_1, [z_I(x)]_2).$$

Our prover time is almost unaffected by the computation of the non-hiding KZG proofs $[Q_i]_1$ and $[H_I]_1$. Indeed, the former can be pre-computed along with all proofs for individual positions using $N \log N$ group operations, and the latter can be obtained from the pre-computed proofs for all $i \in I$, in time dependent on $|I|$, as shown in [28, 13] and discussed in Section 4.3. As a result, note that our prover does require linear storage.

In our case, we would like to show that a secret committed value (or a set of committed values) is at a secret position of our committed vector. On a very high level, the idea behind Caulk is to re-randomize the values provided as part of a KZG opening by appropriate blenders, such that no information about which element is at which position is revealed. The main technical challenge lies in efficiently proving that the blinded KZG opening is still well-formed. We outline the technical ideas between the single and multiple element cases separately.

Single Element. Instead of directly revealing value v , the prover now demonstrates knowledge of v and r behind a Pedersen commitment $\mathbf{cm} = [v + hr]_1$, for unknown h given as $[h]_1$ in the setup. Next, the prover would like to convince the verifier that v is stored somewhere in the vector. For this, the prover publishes $[z(x)]_2 = [a(x - \omega^{i-1})]_2$ and shows that it is a blind commitment to polynomial $X - \omega^{i-1}$, which implies proving that it is a polynomial of degree 1 and that ω^{i-1} is an N th root of unity i.e. that $(\omega^{i-1})^N = 1$.

To prove well-formation of $z(X)$, the prover additionally commits to an auxiliary polynomial $f(X)$ of degree $n = \log(N) + 6$, which effectively encodes a set of constraints on $z(X)$. Crucially important for efficiency, we define $f(X)$ over a small subgroup of roots of unity $\mathbb{V}_n = \{1, \dots, \sigma^{n-1}\}$ with $\sigma^n = 1$. Concretely, the first 5 coefficients of $f(X)$ are used to, by comparing it to $z(X)$, extract ω^{i-1} , the next $\log(N)$ coefficients are used to obtain the 2-powers of $(\omega^{i-1})^{-1}$ up to $2^{\log(N)} = N$, and the last one to prove that $((\omega^{i-1})^{-1})^{2^{\log(N)}} = ((\omega^{i-1})^{-1})^N = (\omega^{i-1})^N = 1$.

Multiple Elements. For the case of multiple elements, the prover would like to convince the verifier that all elements in vector $\vec{a} = (a_1, \dots, a_m)$ that are committed to in a KZG commitment \mathbf{cm} , are also somewhere in the vector \vec{c} committed as \mathbf{C} . We first encode \vec{a} as a polynomial $\phi(X) = \sum_{j=1}^m a_j \mu_j(X)$, where $\{\mu_j(X)\}_{j=1}^m$ are Lagrange interpolation polynomials over a subgroup of roots of unity $\mathbb{V}_m = \{1, \nu, \dots, \nu^{m-1}\}$ with $\nu^m = 1$, and set $\mathbf{cm} = [\phi(x)]_1$.

To prove linkability between \vec{c} and \vec{a} , the prover first sets \vec{c}_I to be the subvector of \vec{c} that contains all the elements c_i such that $c_i = a_j$ for some a_j , without repetitions, and computes $C_I(X)$ using the Lagrange polynomials $\{\tau_i(X)\}_{i \in I}$ that correspond to $\mathbb{H}_I = \{\omega^{i-1}\}_{i \in I}$. Using KZG proofs of openings for blinded commitments to $C_I(X)$ and $z_I(X)$, the prover sends $[H_I(x)]_1$ where $H_I(X)$ is a blinded version of the polynomial $H'_I(X)$ such that

$$C(X) - C_I(X) = z_I(X) H'_I(X).$$

Then, it remains to prove that $z_I(X)$ has the right form and $[C_I(x)]_1$ is a commitment to the same values as $\mathbf{cm} = \sum_j^m a_j \mu_j(X)$, just in a different basis, namely $\{\tau_i(X)\}$ vs $\{\mu_j(X)\}$. For the first statement we again introduce an auxiliary polynomial $u(X) = \sum_{j=1}^m \omega^{i_j-1} \mu_j(X)$ that includes all the ω^{i-1} with $i \in I$, but with the corresponding repetitions. We prove that $u(X)$'s coefficients are N th roots of unity by providing a proof that $u_j(X) = u_{j-1}(X) u_{j-1}(X)$ for $j = 1, \dots, m$, when evaluated at elements in \mathbb{V}_m ,

and showing that $u_0(X) = u(X)$ and $u_n(X) = 1$. Then it remains to prove that $z_I(X)$ vanishes at every coefficient of $u(X)$ i.e. $z_I(u(X))$ vanishes at all elements of \mathbb{V}_m . This is done by providing $H_2(X)$ such that $z_I(u(X)) = z_H(X)H_2(X)$. Note that the argument holds also when $u(X)$ has repeating coefficients.

For the first statement, we introduce an auxiliary polynomial $u(X) = \sum_{j=1}^m \omega^{i_j-1} \mu_j(X)$ that includes all the ω^{i-1} with $i \in I$ but with the corresponding repetitions. We also define polynomials $\{u_j(X)\}_{j=0}^n$ and show that $u(X)$'s coefficients are N th roots of unity by providing a proof that $u_j(X) = u_{j-1}(X)u_{j-1}(X)$ for $j = 1, \dots, m$, when evaluated at elements in \mathbb{V}_m , and that $u_0(X) = u(X)$ and $u_n(X) = 1$. Then it remains to prove that $z_I(X)$ vanishes at every coefficient of $u(X)$ i.e. $z_I(u(X))$ vanishes at all elements of \mathbb{V}_m . This is done by providing $H_2(X)$ such that $z_I(u(X)) = z_H(X)H_2(X)$. Note that the argument holds also when $u(X)$ has repeating coefficients.

(ii) is proven by asserting the polynomial equation

$$C_I(u(X)) - \phi(X) = z_H(X)H_3(X)$$

holds for some $H_3(X)$, thus linking an input $\phi(X)$ in the known basis $\{\mu_j(X)\}_{j=1}^m$ to $C_I(X)$ in the unknown basis $\{\tau_i(X)\}_{i \in I}$.

3 Related Work

Merkle-SNARK. Zcash protocol [30] proposed a SNARK over a circuit describing a Merkle tree opening for the anonymous proof of coin ownership. It remains a very popular approach for various set membership proof protocols [29, 31]. The prover costs are logarithmic in the number of tree leafs, but the concrete efficiency varies depending on the hash function that comprises the tree [1, 19]. Regular hash functions such as SHA-2 are known to be very slow, whereas algebraic alternatives are rather novel and some applications are reluctant to use them.

Pairing Based. Camenisch et al.[9] describe a vector commitment that only requires constant prover and verifier costs. However the commitments themselves are computed by a trusted third party and have linear size because the prover requires access to $[\frac{1}{x \cdot c_i}]_1$ for all c_i in the vector and x secret. Benaroch et al. introduced in [5] what we define as position-hiding linkability for a commitment C corresponding to the PST vector commitment scheme [26] and a commitment cm to one element using Pedersen's scheme. Similar to ours, their construction consists on opening a public polynomial encoding a vector at some hiding position s (instead of at element ω^{i-1}) and prove that the output is the element committed in cm , along with well formation of the input (by showing that $s < N$). Still, their construction has a proof of size logarithmic in N and asks the verifier to perform $O(\log N)$ group operations and $\log(N)$ pairings.

Discrete-Log Based. In the discrete-logarithm setting a series of works have looked into achieving logarithmic sized zero-knowledge membership proof [3, 21, 7, 8]. These have the advantage that there is no trusted setup or pairings. The prover and verifier costs are asymptotically dominated by a linear number of field operations. For modest sized vectors this can be practical because the number of more computationally intensive group operations is logarithmic.

RSA Accumulators. Camenisch and Lysyanskaya [10] design a proof of knowledge protocol for linking a commitment over a prime ordered group to an RSA accumulator. There are no a-priori bounds on the size of the vector and nicely, RSA based schemes have constant size public parameters. This approach is used by Zerocoins [25] which is a privacy preserving payments system (the predecessor to Zerocash [4]). Benaroch et al. [5] improve on this result by allowing the use of prime ordered groups of “standard” size, e.g., 256 bits, whereas [10] needs a much larger group. As opposite to Merkle tree constructions, [5] has prover time constant on the size of the table, and gets up to almost four times faster for elements of arbitrary size and between 4.5 and 23.5 for elements that are large prime numbers; as drawback, proof size goes from 4 to 5 KB. Later, Campanelli et al. [11] present also an scheme for position-hiding linkability of RSA accumulators for large prime numbers and Pedersen commitments. Their proving times does not depend on the size of the accumulator and outperforms Merkle tree approaches by orders of magnitude; however they require either a trusted RSA modulus or class groups.

4 Preliminaries

A bilinear group gk is a tuple $gk = (q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, [1]_1, [1]_2)$ where $\mathbb{G}_1, \mathbb{G}_2$ and \mathbb{G}_T are groups of prime order q , the elements $[1]_1, [1]_2$ are generators of $\mathbb{G}_1, \mathbb{G}_2$ respectively. We also consider $[\mathbf{h}]_1$ another

Scheme	Trusted Params	$ \text{srs} $	Proof size	Prover work	Verifier work
Merkle trees + zkSNARKs	Updatable	$m \log(N)$	$13\mathbb{G}_1, 8\mathbb{F}$	$\tilde{O}(m \log(N))$	2P
RSA accumulators	Yes	$O(1)$	$2\mathcal{G}$	$O(\log(m))$	$m \exp$
Caulk single opening (Sec. 6)	Updatable	$O(N)$	$6\mathbb{G}_1, 2\mathbb{G}_2, 4\mathbb{F}$	$\tilde{O}(\log(N))$	4P
Caulk lookup (Sec. 7)	Updatable	$O(N)$	$14\mathbb{G}_1, 1\mathbb{G}_2, 4\mathbb{F}$	$\tilde{O}(m^2 + m \log(N))$	4P

Table 1: Cost comparison of our scheme with alternative proofs for membership and lookups. N is the size of the table and m the size of the set to be opened. We consider that Merkle trees + zk-SNARKs are implemented using Marlin [12] and note that these numbers are different with other SNARKs. Note that the asymptotic prover work for the Merkle trees + zkSNARKs hides the large constants involved in arithmeticising hash functions. The RSA accumulator asymptotics hides large constants: for example \mathcal{G} denotes a hidden order group that has larger size than $\mathbb{G}_1, \mathbb{G}_2$.

generator of \mathbb{G}_1 , where h is unknown and $\mathsf{h}[1]_1 = [\mathsf{h}]_1$. $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ is an efficiently computable, non-degenerate bilinear map, and there is no efficiently computable isomorphism between \mathbb{G}_1 and \mathbb{G}_2 . Elements in \mathbb{G}_γ , are denoted implicitly as $[a]_\gamma = a[1]_\gamma$, where $\gamma \in \{1, 2, T\}$ and $[1]_T = e([1]_1, [1]_2)$. With this notation, $e([a]_1, [b]_2) = [ab]_T$.

Let $\lambda \in \mathbb{N}$ denote the security parameter and 1^λ its unary representation. A function $\text{negl} : \mathbb{N} \rightarrow \mathbb{R}$ is called *negligible* if for all $c > 0$, there exists k_0 such that $\text{negl}(k) < \frac{1}{k^c}$ for all $k > k_0$. For a non-empty set S , let $x \leftarrow S$ denote sampling an element of S uniformly at random and assigning it to x .

Let PPT denote probabilistic polynomial-time. Algorithms are randomized unless explicitly noted otherwise. Let $y \leftarrow A(x; r)$ denote running algorithm A on input x and randomness r and assigning its output to y . Let $y \leftarrow A(x)$ denote $y \leftarrow A(x; r)$ for a uniformly random r .

Lagrange Polynomials and Roots of Unity. We use ω to denote a root of unity such that $\omega^N = 1$, and define $\mathbb{H} = \{1, \omega, \dots, \omega^{N-1}\}$. Also, we let $\lambda_i(X)$ denote the i^{th} lagrange polynomial, i.e., $\lambda_i(X) = \prod_{s \neq i-1} \frac{X - \omega^s}{\omega^{i-1} - \omega^s}$ and $z_H(X) = \prod_{i=0}^{N-1} (X - \omega^i) = X^N - 1$ the vanishing polynomial of \mathbb{H} . We will additionally consider smaller groups of roots of unity in Sections 6, 7 and 7.2, that will be introduced accordingly.

4.1 Cryptographic Assumptions

The security of our protocols holds in the Algebraic Group Model (AGM) of Fuchsbauer et al. [15], using the *bilinear* version of the *dlog*, *qDHE*, *qSFRAC*, and *qSDH* assumptions [18, 6]. In the AGM adversaries are restricted to be *algebraic* algorithms, namely, whenever \mathcal{A} outputs a group element $[y]$ in a cyclic group \mathbb{G} of order p , it also outputs its representation as a linear combination of all previously received group elements. In other words, if $[y] \leftarrow \mathcal{A}([x_1], \dots, [x_m])$, \mathcal{A} must also provide \vec{z} such that $[y] = \sum_{j=1}^m z_j [x_j]$. This definition generalizes naturally in asymmetric bilinear groups with a pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$, where the adversary must construct new elements as a linear combination of elements in the same group.

4.2 The KZG Polynomial Commitment Scheme

Our constructions heavily rely on the KZG polynomial commitment scheme (Def. A.3) that we describe below, as well as its adaptation for vector commitments that we explain in the next section. For efficiency, we slightly modify the polynomial commitment in order to add degree checks to the original protocol, without incurring in extra proof elements or pairings. The polynomial commitment introduced by Kate, Zaverucha and Goldberg in [22] is a tuple of algorithms ($\text{KZG}.\text{Setup}$, $\text{KZG}.\text{Commit}$, $\text{KZG}.\text{Open}$, $\text{KZG}.\text{Verify}$) such that:

- $\text{srs}_{\text{KZG}} \leftarrow \text{KZG}.\text{Setup}(\text{par}_{\text{KZG}}, d)$: On input the system parameters and a degree bound d , it outputs a structured reference string $\text{srs}_{\text{KZG}} = ([x^i]_{1,2})_{i=1}^d$.
- $\mathsf{C} \leftarrow \text{KZG}.\text{Commit}(\text{srs}_{\text{KZG}}, p(X))$: It outputs $\mathsf{C} = [p(x)]_1$.

- $(s, \pi_{\text{KZG}}) \leftarrow \text{KZG.Open}(\text{srs}_{\text{KZG}}, p(X), \alpha)$: Let $\deg < d$ be the degree of $p(X)$. Prover computes

$$q(X) = \frac{p(X) - p(\alpha)}{X - \alpha},$$

sets $s = p(\alpha)$, $[Q]_1 = [q(x)x^{d-\deg+2}]_1$, and outputs $(s, \pi_{\text{KZG}} = [Q]_1)$.

- $1/0 \leftarrow \text{KZG.Verify}(\text{srs}_{\text{KZG}}, \mathcal{C}, \deg, \alpha, s, \pi_{\text{KZG}})$: Verifier accepts if and only if

$$e(\mathcal{C} - s, [x^{d-\deg+2}]_2) = e([Q]_1, [x - \alpha]_2).$$

Security. It has been proven in [22, 12, 16] that the original KZG protocol, i.e., where $[Q]_1 = [q(x)]_1$ and the pairing equation is $e(\mathcal{C} - s, [1]_2) = e([Q]_1, [x - \alpha]_2)$, is a polynomial commitment scheme that satisfies completeness, evaluation blinding and extractability as in Def. A.3 in the AGM, under the dlog assumption. What is more, Marlin presents an alternative version of KZG with degree checks that does not require additional powers in \mathbb{G}_2 . For our construction, we claim that adding $x^{d-\deg+2}$ to the pairing and element $[Q]_1$ does not affect completeness or extractability. We also argue that under the AGM, no PPT adversary \mathcal{A} can break soundness by providing a commitment to a polynomial $p(X)$ such that $\deg(p) > \deg$. Indeed, if that is the case, $\deg(Q) = d + 1$ for $Q(X)$ the algebraic representation of $[Q]_1$, which will imply an attack to the d -DHE assumption, as the srs only contains powers $[x^i]_1$ up to d .

4.3 KZG as Vector Commitment Scheme

There is a natural isomorphism between vectors of size m and polynomials of degree $m - 1$; where we can represent $\vec{c} = (c_1, \dots, c_m) \in \mathbb{F}^m$ as $C(X) = \sum_{j=1}^m c_j B_j(X)$, where $\mathcal{B} = \{B_j(X)\}_{j=1}^m$ is a basis of the space of polynomials of degree up to $m - 1$, and vice versa. This fact implies as well a natural relation between polynomial and vector commitments (Def. A.2), where in particular, the former implies the latter. What is more, when the basis \mathcal{B} chosen to encode the vector consists of Lagrange polynomials we have vector commitments with *easy* individual position openings: evaluating $V(X)$ in the $i - 1$ th interpolation point returns c_i .

In this work we will use the protocol by Kate et al. for both cases, polynomial and vector commitments. For the latter, we will not only consider individual openings but also subset openings. In particular, let $\mathbb{H} = \{1, \omega, \dots, \omega^{N-1}\}$ be a set of roots of unity and $\{\lambda_i(X)\}_{i=1}^N$ its corresponding Lagrange interpolation set, with vanishing polynomial $z_H(X)$. That is, $\lambda_i(\omega^{i-1}) = 1$ and $\lambda_i(\omega^j) = 0$ for all $j \neq i - 1$. We have that for some polynomial $H(X)$,

$$C(X) - s = (X - \omega^{i-1})H(X) \text{ if and only if } C(\omega^{i-1}) = c_i = s.$$

For a polynomial $C_I(X) = \sum_{i \in I} s_i \tau_i(X)$ where s_i are claimed values for v_i and $\{\tau_i(X)\}_{i \in I}$ the Lagrange interpolation polynomials of the set $\{\omega^{i-1}\}_{i \in I}$,

$$C(X) - C_I(X) = \prod_{i \in I} (X - \omega^{i-1})H(X) \text{ iff } V(\omega^{i-1}) = c_i = s_i \text{ for all } i \in I.$$

4.4 Subset openings

For a vector $\vec{c} \in \mathbb{F}^m$ and a subset $I \subset [m]$, the subvector opening scheme of Tomescu et. al [28] that works for the VC inspired by KZG presented above, consists on algorithms *Open* and *Verify* such that:

- $\text{Open}(\text{srs}_{\text{KZG}}, I, \vec{c}_I)$: Compute $C_I(X) = \sum_{i \in I} c_i \tau_i(X)$, where $\{\tau_i(X)\}$ are the Lagrange interpolation polynomials of the set $\{\omega^{i-1}\}_{i \in I}$, and find $H(X)$ such that for $z_I(X) = \prod_{i \in I} (X - \omega^{i-1})$,

$$C(X) - C_I(X) = z_I(X)H(X).$$

Output $\pi_I = [H]_1 = [H(x)]_1$.

- $\text{Verify}(\text{srs}_{\text{KZG}}, \mathcal{C}, I, \vec{c}_I, \pi_I)$: Compute $[z_I]_2 = [z_I(x)]_2$, $C_I(X) = \sum_{i \in I} c_i \tau_i(X)$, and $\mathcal{C}_I = [C_I(x)]_1$ and output 1 if and only if

$$e(\mathcal{C} - \mathcal{C}_I, [1]_2) = e([H]_1, [z_I]_2).$$

Open as aggregation of individual proofs We will additionally use a result by Tomescu et al. [28] that allows the prover to compute $[H]_1$ in time $\mathcal{O}(m \log^2(m))$ given it already has stored proofs $\{[H_i]_1\}_{i \in I}$ that $C(\omega^{i-1}) = c_i$. Indeed the prover sets

$$[H]_1 = \sum_{i \in I} \left(\prod_{k=1, k \neq i}^m \frac{1}{(\omega^{i-1} - \omega^{k-1})} \right) [H_i]_1$$

Remark 1. We remark that precomputing all the proofs $[H_1]_1, \dots, [H_N]_1$ that $C(\omega^{i-1}) = c_i$ can be achieved in time $\mathcal{O}(N \log N)$ using techniques by Feist and Khovratovich [13]. The overview of this technique by Tomescu et al. ([28], Section 3.4.4, ‘‘Computing All u_i ’s Fast’’) is explained well.

4.5 Multiple Openings

A KZG proof of opening can naturally be extended to open one polynomial in many points. Indeed, let $p(X)$ be a polynomial, $\vec{\alpha} \in \mathbb{F}^m$ a vector of opening points and \vec{s} such that $s_i = p(\alpha_i)$ for all $i = 1, \dots, m$. Define $C_{\vec{\alpha}}(X)$ as the unique polynomial of degree $m-1$ such that $C_{\vec{\alpha}}(\alpha_i) = s_i$ for all $i \in [m]$. We have that $p(\alpha_i) = s_i$ for all $i = 1, \dots, m$ if and only if there exists $Q(X)$ such that

$$p(X) - C_{\vec{\alpha}}(X) = \prod_{i=1}^m (X - \alpha_i) Q(X)$$

We can thus redefine the KZG prover and verifier the following way:

- $(s, \pi_{\text{KZG}}) \leftarrow \text{KZG.Open}(\text{srs}_{\text{KZG}}, p(X), \vec{\alpha})$: Prover computes $\{\tau_i(X)\}_{i=1}^m$ the interpolation Lagrange polynomials for the set $\{\alpha_i\}_{i=1}^m$, $z_\alpha(X) = \prod_{i=1}^m (X - \alpha_i)$ and define $C_{\vec{\alpha}}(X) = \sum_{i=1}^m p(\alpha_i) \tau_i(X)$. Then, it computes

$$Q(X) = \frac{p(X) - C_{\vec{\alpha}}(X)}{z_\alpha(X)},$$

sets $s_i = p(\alpha_i)$, $[Q]_1 = [Q(x)]_1$, and outputs $(\vec{s}, \pi_{\text{KZG}} = [Q]_1)$.

- $1/0 \leftarrow \text{KZG.Verify}(\text{srs}_{\text{KZG}}, C, \vec{\alpha}, \vec{s}, \pi_{\text{KZG}})$: The verifier computes $\{\tau_i(X)\}_{i=1}^m$, $C_{\vec{\alpha}} = [C_{\vec{\alpha}}(x)]_1$, $[z_\alpha(x)]_2$ and verifies

$$p(X) - C_{\vec{\alpha}}(X) = Q(X) z_\alpha(X)$$

by making the pairing check

$$e(C - C_{\vec{\alpha}}, [1]_2) = e([Q]_1, [z_\alpha(x)]_2),$$

and outputs 1 if and only if the equation is satisfied and $\deg(p) \leq d$.

4.6 KZG for Bivariate Polynomials

For the protocol in Section 7.2 we will use bivariate polynomials, or polynomials of higher degree. What this mean is that, if we have a bivariate polynomial $P(X, Y)$ with degree up to $d_1 - 1$ in X and $d_2 - 1$ in Y then we require a universal setup with $d_1 d_2$ powers. We work with a version of KZG that uses a univariate setup because these are already available for multiple different curves (i.e. we do *not* need a specialist setup just for our protocol and can work with prior KZG setups).

We observe that, by using the KZG open algorithm, we can commit to $P(X, Y)$ as $[P(x^{d_2}, x)]_1$. We must open $P(X, Y)$ in two steps. First we *partially* open $P(X, Y)$ at some point $X = \alpha$ to a commitment $[P(\alpha, x)]_1$. The partial proof is given by a commitment $[w_\alpha(x^{d_2}, x)]$ to a partial witness

$$w_\alpha(X, Y) = \frac{P(X, Y) - P(\alpha, Y)}{X - \alpha}$$

We then fully evaluate $P(\alpha, Y)$ at $Y = \beta$ via a standard KZG proof with a degree bound of $d_2 - 1$ on $[P(\alpha, x)]_1$.

4.7 Proof of Opening of a Pedersen Commitment

Pedersen commitment schemes are a particular case of vector commitments. We will consider them for committing to single values in a zero knowledge way. Thus, the srs will additionally output $[h]_1$ for some secret h and the commitment to some element s is computed as $v[1]_1 + r[h]_1 = [v + hr]$, for some randomly sampled $h \in \mathbb{F}$. We suggest a standard Fiat-Shamire Sigma protocol [24] to demonstrate knowledge of v, r such that $\text{cm} = [v + hr]_1$ for some v, r :

$$R_{\text{ped}} = \{(cm; (v, r)) : cm = [v + hr]_1\}$$

The proof consists of $R = [s_1 + hs_2]_1$, $t_1 = s_1 + vc$ and $t_2 = s_2 + rc$, where $c = H(cm, R)$ and s_1, s_2 are elements chosen by the verifier. At the end, the verifier checks that $R + c \cdot cm = [t_1 + ht_2]_1$.

5 Position-Hiding Linkable Vector Commitments

We introduce the concept of position-hiding linkable vector commitment schemes. Informally, two vector commitment schemes VC_1 and VC_2 are position-hiding linkable if a prover is able to convince a verifier that for a given commitments C corresponding to VC_1 and cm corresponding to VC_2 , it is true that all the elements in the vector committed in cm are also elements of the vector committed in C .

Basicallly, position-hiding linkability allows the prover to extract or isolate in zero-knowledge elements from some public set or table, and later prove further attributes on them. This new primitive should satisfy three security notions: completeness, as usual; *linkability*, that captures the fact that if the proof verifies then there is no element committed in cm that is not also committed in C ; and *position-hiding*, which holds only if no information about the set of elements in C that have been used to construct cm is leaked.

Definition 5.1 (Position-Hiding Linkability for Vector Commitments). *Two vector commitment schemes VC_1 and VC_2 are position-hiding linkable if there exist algorithms $(\text{Setup}_{\text{link}}, \text{Prove}_{\text{link}}, \text{Verify}_{\text{link}}, \text{Simulate}_{\text{link}})$ that behave as follows,*

- $\text{Setup}_{\text{link}}(1^\lambda, d_1, d_2)$: takes as input the security parameter, bounds on the length of vectors in VC_1 and VC_2 , and outputs common parameters srs that include $\text{srs}_1 = VC_1.\text{srs}$ and $\text{srs}_2 = VC_2.\text{srs}$ as well as trapdoor x , including the corresponding trapdoors x_1 and x_2 .
- $\text{Prove}_{\text{link}}(\text{srs}, r, r', \vec{v}, \vec{a})$: on input the srs , commitment randomness r to vector $\vec{v} \in \mathbb{F}^N$ and commitment randomness r' to $\vec{a} \in \mathbb{F}^m$, outputs a proof π that there exists some $I \subset [N]$ such that for all $j = 1, \dots, m$, $a_j = v_i$ for some $i \in I$.
- $\text{Verify}_{\text{link}}(\text{srs}, C, cm, \pi)$: On input the srs , commitments C and cm , and proof π , accepts or rejects.
- $\text{Simulate}_{\text{link}}(x, C, cm)$: On input the trapdoors x and commitments C and cm , outputs a simulated proof π_{sim} ,

and satisfy the following properties:

Completeness: For all N, m with $N \leq d_1, m \leq d_2$, all $\vec{v} \in \mathbb{F}^N$, and all $\vec{a} \in \mathbb{F}^m$ such that for all $j = 1, \dots, m$, $a_j = v_i$ for some $i \in I$, it holds that:

$$\Pr \left[\text{Verify}_{\text{link}}(\text{srs}, C, cm, \pi) = 1 \middle| \begin{array}{l} (\text{srs}, x) \leftarrow \text{Setup}_{\text{link}}(1^\lambda, d_1, d_2); \\ C \leftarrow VC_1.\text{Commit}(\text{srs}_1, \vec{v}, r); \\ cm \leftarrow VC_2.\text{Commit}(\text{srs}_2, \vec{a}, r'); \\ \pi \leftarrow \text{Prove}_{\text{link}}(\text{srs}, r, r', \vec{v}, \vec{a}) \end{array} \right] = 1.$$

Linkability For all N, m with $N \leq d_1, m \leq d_2$, and all PPT adversaries, there exists an extractor $\mathcal{X}_{\mathcal{A}}$ such that:

$$\Pr \left[\begin{array}{l} \text{Verify}_{\text{link}}(\text{srs}, C, \text{cm}, \pi) = 1 \wedge \\ |\vec{v}| = N \wedge \\ (\exists j \in [m] \text{ s.t. } a_j \neq c_i \forall i \in [N]) \vee \\ \text{VC}_2.\text{Commit}(\text{srs}_2, \vec{a}, r') \neq \text{cm} \end{array} \middle| \begin{array}{l} (\text{srs}, x) \leftarrow \text{Setup}_{\text{link}}(1^\lambda, d_1, d_2); \\ \vec{v} \leftarrow \mathcal{A}(\text{srs}); \\ C \leftarrow \text{VC}_1.\text{Commit}(\text{srs}_1, \vec{v}); \\ (\pi, \text{cm}) \leftarrow \mathcal{A}(\text{srs}, C); \\ (\vec{a}, r') \leftarrow \mathcal{X}_{\mathcal{A}}(\text{cm}, \pi) \end{array} \right] = \text{negl}(\lambda).$$

Position-Hiding For all N, m with $N \leq d_1, m \leq d_2$, for all \vec{v} and \vec{a} , all PPT adversaries \mathcal{A} , there exists a PPT algorithm $\text{Simulate}_{\text{link}}$ such that:

$$\left[\mathcal{A}(\text{srs}, C, \text{cm}, \pi) = 1 \middle| \begin{array}{l} (\text{srs}, x) \leftarrow \text{Setup}_{\text{link}}(1^\lambda, d_1, d_2) \\ C \leftarrow \text{VC}_1.\text{Commit}(\text{srs}_1, \vec{v}, r) \\ \text{cm} \leftarrow \text{VC}_2.\text{Commit}(\text{srs}_2, \vec{a}, r') \\ \pi \leftarrow \text{Prove}_{\text{link}}(\text{srs}, r, r', \vec{v}, \vec{a}) \end{array} \right] \approx \left[\mathcal{A}(\text{srs}, C, \text{cm}, \pi_{\text{sim}}) = 1 \middle| \begin{array}{l} (\text{srs}, x) \leftarrow \text{Setup}_{\text{link}}(1^\lambda, d_1, d_2) \\ C \leftarrow \text{VC}_1.\text{Commit}(\text{srs}_1, \vec{v}, r) \\ \text{cm} \leftarrow \text{VC}_2.\text{Commit}(\text{srs}_2, \vec{a}, r') \\ \pi_{\text{sim}} \leftarrow \text{Simulate}_{\text{link}}(x, C, \text{cm}) \end{array} \right]$$

In the next sections, we introduce position-hiding linkability for KZG commitments of arbitrary size and Pedersen commitments for single elements (Section 6), as well as for two KZG commitments (Section 7).

6 Linking Vectors with Elements

In this section we present a method to link a commitment C to a vector $\vec{c} \in \mathbb{F}^N$ (computed as $C = [C(x)]_1$ with $C(X) = \sum_{i=1}^N c_i \lambda_i(X)$), to a Pedersen commitment cm . By this we mean a method for a prover to convince a verifier that there exists an i such that C opens to v at some N th rooth of unity ω^{i-1} and $\text{cm} = [v + \text{hr}]_1$.

We will consider two groups of roots of unity:

- $\mathbb{H} = \{1, \omega, \dots, \omega^{N-1}\}$ of size N with $\omega^N = 1$, Lagrange interpolation polynomials $\{\lambda_i(X)\}_{i=1}^N$ where $\lambda_i(\omega^{i-1}) = 1$ and $\lambda_i(\omega^j) = 0$ if $j \neq i - 1$, and vanishing polynomial $z_H(X)$.
- $\mathbb{V}_n = \{1, \sigma, \dots, \sigma^{n-1}\}$ of size $n = \log(N) + 6$ with $\sigma^n = 1$, Lagrange interpolation polynomials $\{\rho_s(X)\}_{s=1}^n$ and vanishing polynomial $z_{V_n}(X)$ ².

Our construction can be divided into three main components. The first one is a proof of knowledge for the element v committed in cm , that is a proof for relation R_{ped} as defined in Section 4.7. The second is a modified protocol for computing blinded versions of KZG openings for statements $C(\omega^{i-1}) = v$ that does not reveal the coordinate i or the evaluation v , which we describe below. The high-level idea here is to re-randomize a regular KZG opening with an additional blinding factor. Our third component then proves that the re-randomized vanishing polynomial used for the KZG opening is well-formed, i.e., a NIZK argument (as in Def. A.1) for the relation

$$R_{\text{unity}} = \{(\text{srs}, [z]_2; (a, i)) : [z]_2 = [a(x - \omega^{i-1})]_2 \wedge (\omega^{i-1})^N = 1\}$$

6.1 Our Blinded Evaluation Construction

Our prover takes $(r' = \perp, \vec{c})$ and (r, v) as input, where the first tuple represents the vector inside the (deterministic) KZG commitment and the second tuple represents the randomness and value for the pedersen commitment. Let $C(X) = \sum_{i=1}^N c_i \lambda_i(X)$ be the polynomial encoding vector \vec{c} . In a regular KZG opening for position i , the prover would compute $Q(X) = \frac{C(X) - v}{X - \omega^{i-1}}$ and reveal $Q = [Q(x)]_1$. Instead, our prover computes a special kind of obfuscated commitment to ω^{i-1} by selecting a random a and committing to $z(X) = aX - b = a(X - \omega^{i-1})$ where $\omega^{i-1} = \frac{b}{a}$, as an element $[z]_2 = [z(x)]_2$. The blinding factor is necessary, because the set $\{\omega^{i-1}\}_{i=1}^m$ is polynomial sized, so revealing $[x - \omega^{i-1}]_1$ would allow

²For simplicity, we describe our scheme for $n = \log(N) + 6$. Still, a subgroup with such size will most probably not exist, in which case we instantiate the protocol with the smallest subgroup of size bigger than n .

the verifier to do a brute force search to find the index. The prover then computes $[T]_1 = [T(x)]_1$ and $[S]_2 = [S(x)]_2$, where

$$T(X) = \frac{Q(X)}{a} + hs, \quad S(X) = -r - sz(X),$$

and s is a uniformly random value chosen by the prover. $T(X)$ is the KZG quotient polynomial $Q(X)$ divided by a (the blinding factor above) to compensate for $z(X)$ having that blinding factor. The additional term $[hs]_1$ mixed in to fully blind the evaluation $[\frac{Q(X)}{a}]_1$ and preserve zero-knowledge. $[S]_2$ is a term that compensates for the h terms in both $[T]_1$ and cm . In the pairing equation that checks these points, $[S]_2$ will be paired with h to ensure that it can only cancel out terms containing h and cannot make incorrect quotient polynomials appear correct.

We use two proofs of knowledge π_{ped} and π_{unity} as described in Section 4.7 and Section 6.2 respectively. The proof π_{ped} is for v, r such that $\text{cm} = [v + hr]_1$. The proof π_{unity} is for a, b such that $[z]_2 = [ax - b]_2$ and $a^N = b^N$. The verifier checks the pairing equation

$$e(C - \text{cm}, [1]_2) = e([T]_1, [z]_2) + e([h]_1, [S]_2).$$

This equation asserts that, for the polynomials $C(X), T(X), z(X), S(X)$ encoded in $C, [T]_1, [z]_2$, and $[S]_2$ respectively, it holds that

$$C(X) - v - hr = T(X)z(X) + hS(X).$$

Now, because $T(X) = \frac{Q(X)}{a} + sh$, $z(X) = a(X - \omega^{i-1})$, and $S(X) = -r - sz(X)$, this is

$$C(X) - v - hr = \left(\frac{Q(X)}{a} + sh \right) z(X) - hr - hsz(X) \Leftrightarrow C(X) - v = \left(\frac{Q(X)}{a} \right) z(X).$$

The full description of our protocol is given in Figure 1.

Prover: Sample blinders $a, s \leftarrow \mathbb{F}$

Using $C(X) = \sum_{i=1}^N c_i \lambda_i(X)$, encoding of \vec{c} and v, r such that $\text{cm} = v[1]_1 + r[h]_1$

Define

$$z(X) = a(X - \omega^{i-1}), \quad T(X) = \frac{C(X) - v}{z(X)} + sh, \quad S(X) = -r - sz(X)$$

$\pi_{\text{ped}} \leftarrow \text{Prove}(R_{\text{ped}}, \text{cm}, (v, r))$

$\pi_{\text{unity}} \leftarrow \text{Prove}(R_{\text{unity}}, (\text{srs}, [z]_2), (a, a\omega^{i-1}))$

Set $[z]_2 = [z(x)]_2, [T]_1 = [T(x)]_1, [S]_2 = [S(x)]_2$ and return $([z]_2, [T]_1, [S]_2, \pi_{\text{ped}}, \pi_{\text{unity}})$

Verifier: Accept if and only if the following conditions hold

$$\begin{aligned} e(C - \text{cm}, [1]_2) &= e([T]_1, [z]_2) + e([h]_1, [S]_2) \\ 1 &\leftarrow \text{Verify}_{\text{ped}}(\text{srs}, \text{cm}, \pi_{\text{ped}}) \\ 1 &\leftarrow \text{Verify}_{\text{unity}}(\text{srs}, [z]_2, \pi_{\text{unity}}) \end{aligned}$$

Figure 1: Zero-knowledge proof of membership. Shows that (v, r) is an opening of cm and that C opens to v at ω^{i-1} .

Theorem 1. Let R_{ped} and R_{unity} be relations for which zero-knowledge argument of knowledge systems are given. The construction in Figure 1 implies position-hiding linkability for the commitment schemes corresponding to C and cm in the algebraic group model under the qSDH and dlog assumptions.

Intuition. The arguments of knowledge for R_{ped} and R_{unity} imply well formation of cm and $[z]_2$, i.e. assert that except with negligible probability, cm is a pedersen commitment to a value v and $[z]_2$ is a commitment to a polynomial $z(X) = a(X + \omega^{i-1})$ for some $i \in [N]$.

Then, the fact that the first verification equation is satisfied imply there exist polynomials $T(X), S(X)$ such that $C(X) - (v + \text{hr}) = T(X)z(X) + \text{h}S(X)$. Because the prover does not know h in the field, this either implies that the prover gets to know h from $[\text{h}]_1$, breaking dlog , or that they output a valid KZG proof for $C(\omega^{i-1}) = v$, therefore either the statement is true, or the adversary breaks $q\text{SDH}$.

The full proof is given in Appendix B.

6.2 Correct computation of $z(X)$

The purpose of this section is provide a zero-knowledge proof of knowledge for relation R_{unity} , i.e. that the prover knows a, b such that $[z]_2 = [ax - b]_2$ and $a^N = b^N$. This proof is used as a subprotocol in Fig. 1's construction for linkability of vector commitments.

In order to prove that $\frac{a}{b}$ is inside the evaluation domain i.e. is an N th root of unity, we prove that its N th power is one. This can be done in time $\log(N)$ by defining elements $f_0, \dots, f_{\log(N)}$ such that satisfy the following conditions: (i) $f_0 = \frac{a}{b}$, (ii) for $i = 1, \dots, \log(N)$ $f_i = f_{i-1}^2$, and (iii) $f_{\log(N)} = 1$.

Because we want to assert $f_1 = \frac{a}{b}$ for the same elements a, b in $z(X) = aX + b$ and we want to do it without giving $z(X)$ in the field, we will assert this relation by adding 4 extra elements and replacing step (i) with the following constraints:

- $f_0 = z(1) = a - b$
- $f_1 = z(\sigma) = a\sigma - b$
- $f_2 = \frac{f_0 - f_1}{1 - \sigma} = \frac{a(1 - \sigma)}{1 - \sigma} = a$
- $f_3 = \sigma f_2 - f_1 = \sigma a - a\sigma + b = b$, and finally
- $f_4 = \frac{f_2}{f_3} = \frac{a}{b}$.

Once we have (i), we redefine the other conditions: (ii) For $i = 0, \dots, \log(N) - 1$, $f_{5+i} = f_{4+i}^2$, and (iii) $f_{4+\log(N)} = 1$. For succinctness, we aggregate all these constraints in a polynomial $f(X)$ whose coefficients in the Lagrange basis associated to \mathbb{V}_n are the f'_i 's, i.e. such that $f(\sigma^i) = f_i$ using the following lemma:

Lemma 1. Let $z(X)$ be a polynomial of degree 1, $n = \log(N) + 6$ and σ such that $\sigma^n = 1$. If there exists a polynomial $f(X) \in \mathbb{F}[X]$ such that

1. $f(X) = z(X)$ for $1, \sigma$.
2. $f(\sigma^2)(1 - \sigma) = f(1) - f(\sigma)$
3. $f(\sigma^3) = \sigma f(\sigma^2) - f(\sigma)$
4. $f(\sigma^4)f(\sigma^3) = f(\sigma^2)$
5. $f(\sigma^{4+i+1}) = f(\sigma^{4+i})^2$, for all $i = 0, \dots, \log(N) - 1$
6. $f(\sigma^{5+\log(N)}\sigma^{-1}) = 1$

Then, $z(X) = aX - b$ where $\frac{b}{a}$ is an N -th root of unity.

The proof is given in Appendix C and we also depict the constraints acting on the evaluations of $f(X)$ in Fig. 2. In this Lemma we have assumed for simplicity that $n = \log(N) + 6$ divides $|\mathbb{F}|$, however it is possible to remove this requirement with appropriate padding.

The prover will construct the polynomial $f(X)$ as

$$f(X) = (a - b)\rho_1(X) + (a\sigma - b)\rho_2(X) + a\rho_3(X) + b\rho_4(X) + \sum_{i=0}^{\log(N)} \left(\frac{a}{b}\right)^{2^i} \rho_{5+i}(X). \quad (1)$$

and commit to it in zero-knowledge. Then, it will show it is correct by comparing $f(\sigma^i)$ with the corresponding values from the constraints in Lemma 1. Namely, for some α chosen by the verifier, it

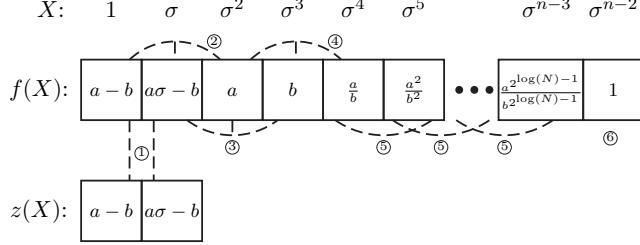


Figure 2: Coefficients of $f(X)$ in the basis $\{\rho_s(X)\}$ and relation with those in $z(X)$ in Lemma 1.

sets $\alpha_1 = \sigma^{-1}\alpha$, $\alpha_2 = \sigma^{-2}\alpha$ and sends $v_1 = f(\alpha_1)$ and $v_2 = f(\alpha_2)$ along with the corresponding proofs of opening. Given v_1, v_2 it then shows that the following polynomial, which proves the constraints in Lemma 1, evaluates to 0 in α :

$$\begin{aligned} p_\alpha(X) = & -h(X)z_{V_n}(\alpha) + (f(X) - z(X))(\rho_1(\alpha) + \rho_2(\alpha)) + ((1 - \sigma)f(X) - f(\alpha_2) + f(\alpha_1))\rho_3(\alpha) \\ & + (f(X) + f(\alpha_2) - \sigma f(\alpha_1))\rho_4(\alpha) + (f(X)f(\alpha_1) - f(\alpha_2))\rho_5(\alpha) \\ & + (f(X) - f(\alpha_1)f(\alpha_1)) \prod_{i \notin [5, \dots, 4+\log(N)]} (\alpha - \sigma^i) + (f(\alpha_1) - 1)\rho_n(\alpha). \end{aligned}$$

Note that the polynomials that are already evaluated in α in $p_\alpha(X)$ are such that either the verifier can compute them, or they are opened by the prover.

Using v_1, v_2 , the commitments to $h(X), f(X)$ and after computing $\rho_i(\alpha)$ for $i = 1, 2, 3, 4, n-1, n$ and $\prod_{i \notin [5, \dots, 4+\log(N)]} (\alpha - \sigma^i)$, the verifier computes a commitment $[P]_1$ to $p_\alpha(X)$ and checks that (i) v_1, v_2 are correct openings of $f(X)$ at $\alpha_1 = \sigma^{-1}\alpha$ and $\alpha_2 = \sigma^{-2}\alpha$, (ii) 0 is a correct opening of $p_\alpha(X)$ at α , and (iii) $[z]_2$ has degree 1.

For this last check, we ask the prover to include a term $X^{d-1}z(X)$ in $h(X)$ and then the verifier computes $[P]_1$ without the terms including $z(X)$, i.e., without $-X^d z(X)z_{V_n}(\alpha) - z(X)(\rho_1(\alpha) + \rho_2(\alpha))$. It will instead add them in the group via the pairing later, to assure that it cannot be the case that $\deg(z) > 1$, unless $\deg(p_\alpha) > d$, which is not possible under the AGM.

We describe the interactive protocol in Fig. 3. In order to turn this public-coin interactive argument into a NIZK we can apply the Fiat-Shamir heuristic: all challenges sent by the verifier are instead generated from a cryptographic hash function.

Theorem 2. *The protocol in Fig. 3 is a knowledge-sound argument (as defined in Def.A.1) for relation R_{unity} if KZG is a sound polynomial commitment scheme, under the the Algebraic Group and Random Oracle models. When used as a building block in the argument of Figure 1, the whole protocol satisfies zero-knowledge³.*

Intuition. We first define an extractor that will use the algebraic representations provided by the adversary. We must show that the output of this extractor is a valid witness with overwhelming probability. The proof proceeds via a series of games where the final game is statistically hard. Game_0 is the knowledge soundness game for the protocol in Fig 3. Game_1 is defined by Game_0 except that it checks whether $f(\alpha_1) = v_1$, $f(\alpha_2) = v_2$ and $p_\alpha(\alpha) = 0$. The advantage of \mathcal{A} in Game_1 is negligible close to the one in Game_0 or it breaks soundness of the KZG polynomial commitment scheme. Game_2 is defined as Game_1 except that it also checks whether the degree of $z(X)$, the algebraic representation of $[z]_2$, is one. Note that if $\deg(z) > 1$ then $p(X)$, the algebraic representation of $[P]_1$, would be a polynomial of degree higher than d , where d is the bound for the powers of x the adversary has access to. The advantage of the adversary in Game_2 then is the same as in Game_1 unless they are able to break qDHE and compute $[x^{d+1}]$.

Now, because α is sent by the verifier after the prover sends $[F]_1, [H]_1$, under the ROM we have that either $p(X) - ((\rho_n(X) + \rho_1(X)) + z_{V_n}(X)X^{d-1})z(X) = 0$ or α is one of its roots, so we conclude that the polynomial equation holds with overwhelming probability. Finally, note that its evaluation in each of the elements of V_n , implies satisfiability of one of the constraints in Lemma 1 and as it includes them all, we have well formation of the polynomial $z(X)$ such that $[z]_2 = [z(x)]_2$.

³When used as an independent argument, $[z]_2$ must be an output of the prover in the first round, or in any round of the main scheme when plugged into other protocols.

Common input: $[z]_2$

Prover: Sample $r_0, r_1, r_2, r_3 \xleftarrow{\$} \mathbb{F}$ and let $r(X) \leftarrow r_1 + r_2X + r_3X^2$

$$f(X) = (a - b)\rho_1(X) + (a\sigma - b)\rho_2(X) + a\rho_3(X) + b\rho_4(X) + \sum_{i=0}^{\log(N)} \left(\frac{a}{b}\right)^{2^i} \rho_{5+i}(X) \\ + r_0\rho_{5+\log(N)}(X) + r(X)z_{V_n}(X),$$

$$p(X) = (f(X) - (aX - b))(\rho_1(X) + \rho_2(X)) + ((1 - \sigma)f(X) - f(\sigma^{-2}X) + f(\sigma^{-1}X))\rho_3(X) \\ + (f(X) + f(\sigma^{-2}X) - \sigma f(\sigma^{-1}X))\rho_4(X) + (f(X)f(\sigma^{-1}X) - f(\sigma^{-2}X))\rho_5(X) \\ + (f(X) - f(\sigma^{-1}X)f(\sigma^{-1}X)) \prod_{i \notin [5; 4+\log(N)]} (X - \sigma^i) + (f(\sigma^{-1}X) - 1)\rho_n(X),$$

Set $\hat{h}(X) = \frac{p(X)}{z_{V_n}(X)}$, $h(X) = \hat{h}(X) + X^{d-1}z(X)$ and output $([F]_1 = [f(x)]_1, [H]_1 = [h(x)]_1)$.

Verifier: Send challenge $\alpha \in \mathbb{F}$

Prover: $\alpha_1 = \sigma^{-1}\alpha$, $\alpha_2 = \sigma^{-2}\alpha$;

$$p_\alpha(X) = -z_{V_n}(\alpha)h(X) + (f(X) - z(X))(\rho_1(\alpha) + \rho_2(\alpha)) + ((1 - \sigma)f(X) - f(\alpha_2) + f(\alpha_1))\rho_3(\alpha) \\ + (f(X) + f(\alpha_2) - \sigma f(\alpha_1))\rho_4(\alpha) + (f(X)f(\alpha_1) - f(\alpha_2))\rho_5(\alpha) \\ + (f(X) - f(\alpha_1)f(\alpha_1)) \prod_{i \notin [5; 4+\log(N)]} (\alpha - \sigma^i) + (f(\alpha_1) - 1)\rho_n(\alpha),$$

Compute

$$((v_1, v_2), \pi_1) \leftarrow \text{KZG.Open}(\text{srs}_{\text{KZG}}, f(X), \deg = \perp, (\alpha_1, \alpha_2)) \\ (0, \pi_2) \leftarrow \text{KZG.Open}(\text{srs}_{\text{KZG}}, p_\alpha(X), \deg = \perp, \alpha),$$

and output (v_1, v_2, π_1, π_2) .

Verifier: Set $\alpha_1 = \sigma^{-1}\alpha$; $\alpha_2 = \sigma^{-2}\alpha$,

$$[P]_1 = -z_{V_n}(\alpha)[H]_1 + (\rho_1(\alpha) + \rho_2(\alpha))[F]_1 + \rho_3(\alpha)((1 - \sigma)[F]_1 + v_1 - v_2) + \rho_4(\alpha)([F]_1 + v_2 - \sigma v_1) \\ + \rho_5(\alpha)(v_1[F]_1 - v_2) + \rho_n(\alpha)(v_1 - 1) + \prod_{i \notin [5, \dots, 4+\log(N)]} (\alpha - \sigma^i)([F]_1 - v_1^2),$$

Parse $\pi_2 = [q]_1$ and accept if and only if

$$1 \leftarrow \text{KZG.Verify}(\text{srs}_{\text{KZG}}, [F]_1, \deg = \perp, (\alpha_1, \alpha_2), (v_1, v_2), \pi_1), \\ e([P]_1, [1]_2) + e(-(\rho_1(\alpha) + \rho_2(\alpha)) - z_{V_n}(\alpha)[x^{d-1}]_1, [z]_2) = e([q]_1, [x - \alpha]_2)$$

Figure 3: NIZK argument of knowledge for R_{unity} and $\deg(z) \leq 1$.

The full proof is in Appendix D.

7 Lookup tables for hiding values

In this section we present the algorithms for position-hiding linkability of KZG vector commitment schemes. The aim is to prove that a commitment cm contains a *subset* of some larger vector committed in \mathbf{C} . We refer to a subset and not to a subvector since our scheme proves that all the elements committed in cm are also committed in \mathbf{C} , but with no specific order and possible repetitions. This is essentially a lookup table if we consider that \mathbf{C} contains the honestly generated table.

Concrete efficiency. Our lookup proof has preprocessing time for \mathbf{C} of $N \log N \mathbb{G}_2$ operations, for N the size of the table. Prover time is $m \log(N)$ scalar multiplications for m the size of the subset, proof size is constant and verifier time $\log \log N$ scalar multiplications and constant number of pairing checks; additionally, update of proofs can be done in $O(N) \mathbb{G}_2$ operations;

Preliminaries We will consider three evaluation domains

1. $\mathbb{H} = \{1, \omega, \dots, \omega^{N-1}\}$ is a group of roots of unity with Lagrange and vanishing polynomials $\{\lambda_i(X)\}_{i=1}^N, z_H(X)$.
2. For subset $\mathbb{H}_I = \{\omega^{i-1}\}_{i \in I}$ of \mathbb{H} defined by $I \subset [N]$, $\{\tau_i(X)\}_{i \in I}$ is the set of its interpolation Lagrange polynomials with degree $|I| - 1$ and $z_I(X)$ its vanishing polynomial. Note that typically \mathbb{H}_I is not a subgroup.
3. For some constant m that bounds the size of the vector committed in cm , we consider another group of roots of unity $\mathbb{V}_m = \{1, \nu, \dots, \nu^{m-1}\}$, where $\nu^m = 1$, as well as its Lagrange and vanishing polynomials, $\{\mu_j(X)\}_{j=1}^m$ and $z_{V_m}(X)$.

7.1 Technical Overview

Our scheme uses as subprotocol a NIZK argument of knowledge for relation R_{unity} ,

$$R_{\text{unity}} = \left\{ (\text{srs}, [z_I]_2, N; (I, r)) : I \subset [N] \wedge [z_I]_2 = r \prod_{i \in I} [x - \omega^{i-1}]_2, \text{ s.t. } (\omega^{i-1})^N = 1, \forall i \in I \right\}$$

The proof for this relation will be divided in two parts, one is a proof of relation

$$R'_{\text{unity}} = \left\{ (\text{srs}, [u]_1, \mathbb{H}, \mathbb{V}) : [u]_1 = [u(x)]_1 \text{ for } u(X) \text{ s.t. } \forall \nu^j \in \mathbb{V}, u(\nu^j) = \omega^i, \text{ for some } \omega^i \in \mathbb{H} \right\},$$

and the other a proof that there exists some polynomial $H(X)$ s.t. $z_I(u(X)) = z_{V_m} H(X)$.

In our protocol, the prover takes as input a commitment $C(X) = \sum_{i=1}^N c_i \lambda_i(X)$ to the lookup table \vec{c} , a structured reference string srs , a commitment

$$\text{cm} = [\phi(x)]_1 = \left[\sum_{j=1}^m a_j \mu_j(x) + a_{m+1} z_{V_m}(x) \right]_1$$

to some vector \vec{a} and the opening witness $\vec{a} = (a_1, \dots, a_{m+1})$. Here a_{m+1} is a random field element that blinds cm . The prover must show that it knows an opening $\phi(X) = \sum_{j=1}^m a_j \mu_j(X) + a_{m+1} z_{V_m}(X)$ to cm such that $a_j \in \{c_i\}_{i=1}^N$ for all $1 \leq j \leq m$. The full argument is given in Fig. 4 and can be divided into three steps.

First, the prover considers the subset $I \subset [N]$ such that for all $j = 1, \dots, m$, $a_j = c_i$ for some $i \in I$, and constructs the subvector $\vec{c}_I = (c_i)_{i \in I}$ of \vec{c} . It commits to it in the Lagrange basis corresponding to $\{\omega^{i-1}\}_{i \in I}$; namely, $C_I(X) = \sum_{i \in I} c_i \tau_i(X)$. Basically, the prover isolates the elements of \vec{c} that will compare with \vec{a} so they can work with polynomials of smaller degree.

To convince the verifier that all the elements in $C_I(X)$ are elements of $C(X)$, it provides commitments to $z_I(X), H_1(X)$ such that

$$C(X) - C_I(X) = z_I(X)H_1(X). \quad (2)$$

Here is the place where the precomputation is used: $C(X)$ has degree N and so does $H_1(X)$. In order to compute a commitment to $H_1(X)$, we use the method described in Section 4.4. This is at the same time the most expensive step in updating a proof whenever $C(X)$ is changed. However, if c_i values are updated in known order, and we precompute an opening for τ_i , then whenever new c_i is available all openings can be updated in $O(N)$ time, hence the claimed update cost.

Our challenge now is hiding $C_I(X)$ and $z_I(X)$ from the verifier without breaking soundness. In our solution the prover first demonstrates that $z_I(X)$ is of the right form, meaning it is the vanishing polynomial of some subset \mathbb{H}_I of \mathbb{H} ; specifically, we need not only a hiding commitment but also a zero-knowledge proof of well formation of $z_I(X)$.

We divide the proof of well formation of $z_I(X)$ in two steps. First, the prover creates the polynomial $u(X) = \sum_{j=1}^m \omega^{ij} \mu_j(X)$ of degree $m - 1$ whose coefficients are the roots of unity $\{\omega^{i-1}\}_{i \in I}$ and prove, in zero knowledge, its well formation. For that, it demonstrates that for all $\nu^j \in \mathbb{V}$ it is the case that $(u(\nu^j))^N = 1$, via a call to a subprotocol $\Pi_{\text{unity}'}$ that we describe in Section 7.2. This guarantees that $u(X)$ is a commitment to elements in \mathbb{H} . Secondly, on input a commitment to $u(X)$ as above and given that $u(X)$ passes the verification of $\Pi_{\text{unity}'}$, we prove well formation of $z_I(X)$ and thus that it satisfies relation R_{unity} . To achieve this we use the fact that all the coefficients of $u(X)$ in the basis $\{\mu_j(X)\}_{j=1}^m$ are roots of $z_I(X)$. For that, prover convinces verifier that

$$z_I(u(X)) = z_{V_m}(X)H_2(X), \text{ for some polynomial } H_2(X). \quad (3)$$

Finally, note that $C_I(X)$ has been committed to in an unknown-to-the-verifier Lagrange basis, which is $\{\tau_i(X)\}$. So the last step of our argument consists on linking the commitment to $C_I(X)$ with $[\phi(x)]_1$, which is an input to the argument and a commitment to the same element in a known basis. The prover does so by providing $H_3(X)$ such that

$$C_I(u(X)) - \phi(X) = z_{V_m}(X)H_3(X). \quad (4)$$

In order to achieve zero-knowledge, upon receiving an aggregation challenge χ from the verifier, the prover actually provides one commitment $[H_2]_1 + \chi[H_3]_1$ to prove equations 3 and 4 together.

Note that for equation 2 to be satisfied, $C_I(X)$ cannot take more than once each of the coefficients of $C(X)$. On the other hand, when linking $C_I(X)$ and $\phi(X)$ through equation 4, we can only prove that all the coefficients of $\phi(X)$ in the basis $\{\mu_j(X)\}_{j=1}^m$ are also coefficients of $C_I(X)$ in the basis $\{\tau_i(X)\}_{i \in I}$, but we cannot say in which order or how many times each of them appears. At the end, what we get, is a lookup table argument that assures that some element $[\phi(x)]_1$ is a commitment in the Lagrange basis $\{\mu_j(X)\}_{j=1}^m$ to some vector $\vec{a} = (a_1, \dots, a_m)$ such that for all $j = 1, \dots, m$ there exists some $i \in I$ such that $a_j = c_i$, i.e., a lookup table for potentially repeated indexes.

Theorem 3. Suppose that the argument of Fig. 4 is instantiated with a knowledge-sound scheme for relation $\mathcal{R}'_{\text{unity}}$. Then in the AGM with non-programmable ROs, either the argument of Fig. 4 implies linkability for the vector commitment schemes of \mathbf{C} and \mathbf{cm} , or there exists an adversary that breaks the q -SDH assumption.

Intuition. We prove linkability through a sequence of games. Game_0 is the linkability game for the protocol of Fig. 4. Game_1 additionally checks that: (i) $[u]_1$ is the commitment to a polynomial $u(X)$ such that $u(\alpha) = v_1$, (ii) $[P]_1$ encodes a polynomial $P_1(X) = z_I(X) + \chi C_I(X)$ such that $P_1(v_1) = v_2$, i.e., $P_1(u(\alpha)) = z_I(u(\alpha)) + \chi C_I(u(\alpha)) = v_2$, and (iii) $[P]_2$ is the commitment to a polynomial $P_2(X) = v_2 - \chi\phi(X) - z_{V_m}(\alpha)H_2(X)$ such that $P_2(\alpha) = 0$, that is, $z_I(u(\alpha)) + \chi C_I(u(\alpha)) - \chi\phi(\alpha) = z_{V_m}(\alpha)H_2(\alpha)$. Soundness of the KZG polynomial commitment scheme assures that the advantages of \mathcal{A} in both games have a negligible difference.

Game_2 behaves identically to Game_1 but it also verifies that $u(X)$ is such that $u(\nu^j)^N = 1$. The advantage of \mathcal{A} in Game_2 is then the same as in Game_1 , due to knowledge soundness of the argument for $\mathcal{R}'_{\text{unity}}$. Game_3 works as Game_2 but further checks that (iv) $C(X) - C_I(X) = z_I(X)H_1(X)$. The advantage of the adversary in Game_3 is the same as in Game_2 , unless the trapdoor x is a root to the polynomial, in which case we can use \mathcal{A} as a subroutine for a successful adversary against q SDH. This

Common input: $\mathbf{C} = [C(x)]_1$, for $C(X) = \sum_{i=1}^N c_i \lambda_i(X)$ and $\mathbf{cm} = [\phi(x)]_1$.

Prover: Take as input \mathbf{srs} and $\phi(X)$ and proof $[Q(x)]_2$ attesting that $\{c_i\}_{i \in I}$ are openings of \mathbf{C} . I.e., a commitment to $Q(X) = \frac{C(X) - \sum_{i \in I} c_i \tau_i(X)}{\prod_{i \in I} (X - \omega^{i-1})}$.

- Choose blenders $r_1, r_2, r_3, r_4, r_5, r_6, r_7 \xleftarrow{\$} \mathbb{F}$ uniformly at random.
- For $\mathbb{H}_I = \{\omega^{i-1}\}_{i \in I}$, compute the interpolation polynomials $\{\tau_i(X)\}_{i \in I}$.
- Define $z_I(X) = r_1 \prod_{i \in I} (X - \omega^{i-1})$ and $C_I(X) = \sum_{i \in I} c_i \tau_i(X) + (r_2 + r_3 X + r_4 X^2) z_I(X)$.
- Compute $[H_1(x)]_2 = [r_1^{-1} Q(x) - (r_2 + r_3 x + r_4 x^2)]_2$.
- Define ω^{i_j} as the j th element in $\{\omega^{i-1}\}_{i \in I}$ and compute

$$u(X) = \sum_{j=1}^m \omega^{i_j} \mu_j(X) + (r_5 + r_6 X + r_7 X^2) z_{V_m}(X).$$

- Compute a proof $\pi_{\text{unity}'}$ as in Fig. 5, proving that $[u]_1$ satisfies R'_{unity} .
- Output $[C_I]_1 = [C_I(x)]_1$, $[z_I]_1 = [z_I(x)]_1$, $[u]_1 = [u(x)]_1$, $[H_1]_2 = [H_1(x)]_2$, $\pi_{\text{unity}'}$.

Verifier: Send challenge $\chi \in \mathbb{F}$

Prover:

- Find $H_2(X)$ such that $z_I(u(X)) + \chi(C_I(u(X)) - \phi(X)) = z_{V_m}(X) H_2(X)$
- Output $[H_2]_1 = [H_2(x)]_1$,

Verifier: Send challenge $\alpha \in \mathbb{F}$

Prover: Compute

$$\begin{aligned} p_1(X) &\leftarrow z_I(X) + \chi C_I(X) \\ p_2(X) &\leftarrow z_I(u(\alpha)) + \chi(C_I(u(\alpha)) - \phi(X)) - z_{V_m}(\alpha) H_2(X) \\ (v_1, \pi_1) &\leftarrow \text{KZG.Open}(\mathbf{srs}_{\text{KZG}}, u(X), \deg = \perp, \alpha) \\ (v_2, \pi_2) &\leftarrow \text{KZG.Open}(\mathbf{srs}_{\text{KZG}}, p_1(X), \deg = \perp, v_1) \\ (0, \pi_3) &\leftarrow \text{KZG.Open}(\mathbf{srs}_{\text{KZG}}, p_2(X), \deg = \perp, \alpha) \end{aligned}$$

Output $(v_1, v_2, \pi_1, \pi_2, \pi_3)$.

Verifier: Compute $[P_1]_1 \leftarrow [z_I]_1 + \chi [C_I]_1$ and $[P_2]_1 \leftarrow v_2 - \chi \mathbf{cm} - z_{V_m}(\alpha) [H_2]_1$.

Accept if and only if (i) $V_{\pi'_{\text{unity}}}$ accepts, (ii)

$$\begin{aligned} 1 &\leftarrow \text{KZG.Verify}(\mathbf{srs}_{\text{KZG}}, [u]_1, \deg = \perp, \alpha, v_1, \pi_1) \\ 1 &\leftarrow \text{KZG.Verify}(\mathbf{srs}_{\text{KZG}}, [P_1]_1, \deg = \perp, v_1, v_2, \pi_2) \\ 1 &\leftarrow \text{KZG.Verify}(\mathbf{srs}_{\text{KZG}}, [P_2]_1, \deg = \perp, \alpha, 0, \pi_3), \text{ and} \\ (iii) \quad e([C]_1 - [C_I]_1, [1]_2) &= e([z_I]_1, [H_1]_2) \end{aligned} \tag{5}$$

Figure 4: Lookup table for non-repeated indexes that uses a proof for R'_{unity} as blackbox.

polynomial equation implies then that $C(\omega^{i-1}) - C_I(\omega^{i-1}) = 0$ for all $i \in I$ and thus $C_I(X)$ encodes the subvector \vec{c}_I of \vec{c} . Lastly, we show that the advantage of \mathcal{A} in Game_3 is negligible.

Because α was sent after prover sends $[C_I]_1, [z_I]_1, [u]_1, [H_1]_1$ and $[H_2]_1$, except with negligible probability, condition (iii) holds as a polynomial equation for all X , that is, $z_I(u(X)) + \chi C_I(u(X)) - \chi \phi(X) = z_{V_m}(X) H_2(X)$. Similarly, because χ was sampled by the verifier after receiving $[C_I]_1, [z_I]_1, [u]_1$, and $[H_1]_1$, we have that there exist $H_{21}(X)$ and $H_{22}(X)$ such that $H_2(X) = H_{21}(X) + \chi H_{22}(X)$, $z_I(u(X)) = z_{V_m}(X) H_{21}(X)$ and $C_I(u(X)) - \phi(X) = z_{V_m}(X) H_{22}(X)$.

The first equation says that $z_I(X)$ is a polynomial with the coefficients of $u(X)$ in the basis $\{\mu_j(X)\}_{j=1}^m$ (that are N th roots of unity) as roots (it may have more); on the other hand, $C_I(u(X)) - \phi(X) = z_{V_m}(X) H_{22}(X)$ implies that the values that $C_I(X)$ takes in the elements of \mathbb{H}_I are the values that $\phi(X)$ takes in \mathbb{V}_m .

The full proof is given in Appendix. E

Subtables There is another nice feature that can be derived by the protocol in Fig. 4 and is the creation of sub-lookup tables. Namely, for some $I \subset [N]$, prover generates $t(X) = \prod_{i \in I} (X - c_i)$. To prove well formation of it, after having some $C_I(X)$ that has been proven correct, it shows that there exists some $H_3(X)$ such that

$$t(\tilde{C}_I(X)) = z_{V_m}(X) H_3(X).$$

Then, for any polynomial $a(X)$ of degree up to $m-1$, if there exists $H_4(X)$ such that

$$t(a(X)) = z_{V_m}(X) H_4(X),$$

then the coefficients of $a(X)$ in the basis $\{\mu_j(X)\}_{j=1}^m$ are coefficients of $C_I(X)$ in basis $\{\tau_i(X)\}_{i \in I}$, with no specific order and potential repetitions.

7.2 Multi-Unity Proof or Proving well formation of $u(X)$

The aim of this section is to prove in zero-knowledge that a commitment $[u]_1$ is well formed, that is, encodes the polynomial $u(X) = \sum_{j=1}^m \omega^{i_j} \mu_j(X) + r(X) z_{V_m}(X)$, where ω^{i_j} is the j -th element in I . Namely, that $u(X) = \sum_{j=1}^m u_j \mu_j(X) + r(X) z_{V_m}(X)$ is such that all its coefficients are elements in \mathbb{H} and thus, they are all N th roots of unity, or what is the same, that $u_j^N = 1$ for all $j = 1, \dots, m$.

For this argument, we will consider another group of roots of unity $\mathbb{V}_n = \{1, \sigma, \dots, \sigma^{n-1}\}$ of size $n = \log(N)$, with $\sigma^n = 1$, Lagrange interpolation polynomials $\{\rho_s(X)\}_{s=1}^n$ and vanishing polynomial $z_{V_n}(X)$.

Techniques. The prover first defines $\vec{u}_0 = (u_1, \dots, u_m) \in \mathbb{F}^m$ to be the vector whose elements are the coefficients of $u(X)$. They then iteratively define $\vec{u}_j = \vec{u}_{j-1} \circ \vec{u}_{j-1}$. I.e., they set

- $\vec{u}_1 = \vec{u}_0 \circ \vec{u}_0 = (u_1^2, \dots, u_m^2);$
- and for all $j = 2, \dots, n$, $\vec{u}_j = \vec{u}_{j-1} \circ \vec{u}_{j-1} = (u_1^{2^j}, \dots, u_m^{2^j}).$

They then must prove three conditions to the verifier: (i) \vec{u}_0 consists on the coefficients of $u(X)$, (ii) equation $\vec{u}_j = \vec{u}_{j-1} \circ \vec{u}_{j-1}$ holds for all $j = 1, \dots, n-1$ and (iii) $\vec{u}_{n-1} \circ \vec{u}_{n-1} = \vec{1}$. Together this gives that all the coefficients u_j are N th roots of unity.

As we are working with encodings as polynomials rather than vectors, the prover sets $u_0(X) = u(X)$, $u_n(X) = \text{id}(X)$ (for $\text{id}(X)$ the polynomial that evaluates to 1 over \mathbb{V}_m), and shows to the verifier that each of the following equations hold:

$$u(X)u(X) - u_1(X) \equiv z_{V_m}(X) H_1(X),$$

\vdots

$$u_{n-1}(X)u_{n-1}(X) - \text{id}(X) \equiv z_{V_m}(X) H_n(X),$$

To aggregate all of these checks into one verification equation we consider $\{\rho_s(Y)\}$ the linear independent Lagrange interpolation polynomials over \mathbb{V}_n and demonstrate that

$$\left(u^2(X) \rho_1(Y) + \sum_{s=2}^n u_{s-1}^2(X) \rho_s(Y) \right) - \left(\sum_{s=1}^{n-1} u_s(X) \rho_s(Y) + \text{id}(X) \rho_n(Y) \right) = z_{V_m}(X) h_2(X, Y), \quad (6)$$

for some polynomial $h_2(X, Y)$.

In the remainder of this section the prover aims to demonstrate that (6) holds at a challenge point (α, β) .

Proving (6): Strategy We prove (6) by showing that for some polynomial $h_1(Y)$, the polynomial

$$p(Y) = \underbrace{\left(u^2(\alpha)\rho_1(\beta) + \sum_{s=2}^n u_{s-1}^2(\alpha)\rho_s(\beta) + z_{V_n}(\beta)(-h_1(\beta) + h_1(Y)) \right)}_{\text{Denote } \xi_1} - \underbrace{\left(\sum_{s=1}^{n-1} u_s(\alpha)\rho_s(\beta) + \text{id}(\alpha)\rho_n(\beta) \right)}_{\text{Denote } \xi_2} - \underbrace{z_{V_m}(\alpha)h_2(\alpha, Y)}_{\text{Denote } \xi_4}$$

evaluates to 0 at $Y = \beta$. For this the prover sends several values needed to reconstruct the commitment $[P]_1$ to $p(Y)$, and then provides a proof that $[P]_1$ opens to 0 at β .

Proving (6): Extra Notation First note that since the polynomials $\rho_s(Y)$ take 1 and 0 values only, we obtain that for all $Y \in \mathbb{V}_n$

$$u^2(X)\rho_1(Y) + \sum_{s=2}^n u_{s-1}^2(X)\rho_s(Y) = (u(X)\rho_1(Y) + \sum_{s=2}^n u_{s-1}(X)\rho_s(Y))^2$$

We denote $\bar{U}(X, Y) = \sum_{s=2}^n u_{s-1}(X)\rho_s(Y)$ and $U(X, Y) = u(X)\rho_1(Y) + \bar{U}(X, Y)$. The prover begins by sending one commitment $[\bar{U}]_1$ to $\bar{U}(X, Y)$ and a second commitment $[h_2]$ to $h_2(X, Y)$. These are *bivariate* commitments. While there exist bivariate polynomial commitment schemes [26], these are incompatible with universal power-of-tau setups that are publicly available [23]. We thus instead view $\bar{U}(X, Y)$ and $h_2(X, Y)$ as the univariate polynomials $U(X^n, X)$ and $h_2(X^n, X)$. See Section 4.6 for more details.

The verifier responds with a random challenge $X = \alpha$.

Proving (6): Definition of and commitment to h_1 The prover now wishes to find h_1 such that $p(Y)$ can be fully defined and its commitment can be computed by the verifier. They first provide a partial opening $[\bar{U}_\alpha]_1$ to $\bar{U}(\alpha, Y)$ and proves this is consistent with $[\bar{U}]_1$. They also open $[u(x)]_1$ at α to get $v_1 = u(\alpha)$. This allows the verifier to compute a commitment to the polynomial $U(\alpha, Y)$ as $U = [u(\alpha)]_1\rho_1(x) + [\bar{U}_\alpha]_1$.

The prover sends a commitment $[h_1]_1 = [h_1(x)]_1$ to $h_1(Y)$ such that

$$\sum_{s=1}^n u_{s-1}^2(\alpha)\rho_s(Y) = (U(\alpha, Y))^2 + h_1(Y)z_{V_n}(Y). \quad (7)$$

The verifier responds with a second random challenge $Y = \beta$ and then (7) appears as

$$\sum_{s=1}^n u_{s-1}^2(\alpha)\rho_s(\beta) = (U(\alpha, \beta))^2 + h_1(\beta)z_{V_n}(\beta) \quad (8)$$

Proving (6): Degree bound The prover must show that $\bar{U}_1(X, 1) = 0$ i.e. that there is no $\rho_1(Y)$ term. This convinces the verifier that the first term of $U(\alpha, Y)$ is indeed $u(\alpha)\rho_1(Y)$. When opening $[\bar{U}_\alpha]_1$ we enforce a degree bound of $n - 1$. This is necessary because we are capturing bivariate polynomials with a univariate polynomial commitment scheme and we need to enforce that there are no X^n terms lingering in $\bar{U}(\alpha, X)$.

Proving (6): Sending ξ_1 The prover communicates ξ_1 by opening $[\bar{U}_\alpha]_1$ to v_2 at $Y = \beta$ and verifier gets

$$\xi_1 = \{(8)\} = U(\alpha, \beta)^2 = (u(\alpha)\rho_1(\beta) + \bar{U}(\alpha, \beta))^2 = (v_1\rho_1(\beta) + v_2)^2$$

Proving (6): Sending ξ_2 The prover communicates

$\xi_2 = \sum_{s=1}^{n-1} u_s(\alpha)\rho_s(\beta)$. To do this we open $[\bar{U}(\alpha, Y)] = [\bar{U}_\alpha]_1$ to v_3 at $Y = \sigma\beta$ for σ the generator of \mathbb{V}_n . Indeed

$$\bar{U}(\alpha, \sigma\beta) = \sum_{s=2}^n u_{s-1}(\alpha)\rho_s(\sigma\beta) = \sum_{s=1}^{n-1} u_s(\alpha)\rho_s(\beta) \quad (9)$$

Proving (6): Finale Finally the verifier can compute a commitment to $p(Y)$ as $[p(Y)]_1 = [(v_2 + v_1\rho_1(\beta))^2]_1 + z_{V_n}(\beta)[h_1]_1 - [v_3 + \text{id}(\alpha)\rho_n(\beta)]_1 - z_{V_m}(\alpha)[h_2]_1$. Thus the prover finishes by demonstrating that $p(\beta) = 0$.

The protocol is shown in Figure 5.

Efficiency. In the protocol of Fig. 4, the work of the prover is dominated by the computation of $H(X)$ and $p_2(X)$ which have degree m^2 , because $[H_1]$ is formed in time m by using the pre-computed individual proofs, and all the other proof elements are commitments to polynomials of degree m . In the protocol of Fig. 5, prover work is dominated by the computation of $[\bar{U}]_1$ and $[h_2]_1$ that are commitments to polynomials of degree $m \log(N)$.

Theorem 4. *The protocol in Figure 5 is a knowledge-sound argument for relation R'_{unity} under the algebraic group model and random oracle model if the qSDH, qDHE, and qSFrac assumptions hold.*

Intuition. We first define an extractor that will use the algebraic representations provided by the adversary. We must show that the output of this extractor is a valid witness with overwhelming probability. The proof proceeds via a series of games where the final game is statistically hard. Game_0 is the knowledge-soundness game for the protocol in Fig. 5. Game_1 behaves identically except that it checks whether $u(\alpha) = v_1$, $\bar{U}_\alpha(1) = 0$, $\bar{U}_\alpha(\beta) = v_2$, $\bar{U}_\alpha(\beta\sigma) = v_3$, and $p(\beta) = 0$ for $u(X)$, $\bar{U}_\alpha(X)$, $p(X)$ being the algebraic representations of $[u]_1$, $[\bar{U}_\alpha]_1$, and $[P]_1$, respectively. Soundness of the KZG polynomial commitment asserts that the difference of the advantages of \mathcal{A} in Game_0 and Game_1 is negligible.

Game_2 is the same as Game_1 but it additionally checks that $\deg(\bar{U}_\alpha) \leq n-1$ and $\deg(h_2) \leq n-1$, and aborts otherwise. To instantiate the protocol, we use the KZG polynomial commitment with the modification exposed in Section 4.2 and, as stated there, a prover that outputs a valid proof of opening for a polynomial with higher degree than the one declared, implies an attack to qDHE. Therefore, $\text{Adv}_{\mathcal{A}}^{\text{Game}_2} \leq \text{Adv}_{\mathcal{A}}^{\text{Game}_1} + \text{Adv}_{\mathcal{A}}^{\text{qDHE}}$.

Game_3 behaves as Game_2 but additionally verifies that $\bar{U}(\alpha, Y) = \bar{U}_\alpha(Y)$, $h_2(\alpha, Y) = h_{2,\alpha}(Y)$, for $\bar{U}_\alpha(X)$, $h_2(X)$, $h_{2,\alpha}(X)$, the algebraic representations of $[\bar{U}_\alpha]_1$, $[h_2]_1$, and $[h_{2,\alpha}]_1$. That is, Game_3 checks correctness of the partial evaluations (see Section 4.6). In the full proof, we show that if it is not the case, the adversary can be used as a subroutine for a successful adversary against qSFrac.

Finally, since $p(\beta) = 0$ and since $[u]_1, [\bar{U}]_1, [h_1]_1, [h_2]_1$ have been sent by the prover before it sees challenge β , and $[u]_1, [\bar{U}]_1, [h_2]_1$ before it sees challenge α , with overwhelming probability

$$\begin{aligned} p(X) &= (u(X)\rho_1(Y) + \bar{U}(X, Y))^2 - h_1(X)z_{V_n}(Y) \\ &\quad - (\bar{U}(X, Y\sigma) + \text{id}(X)\rho_n(Y)) - z_{V_m}(X)h_2(X, Y). \end{aligned}$$

Note that the latter is equation 6 as at the beginning of Section 7.2. That is, it is the aggregation of the constraints that prove $u(X)$ is a polynomial such that all its coefficients in the Lagrange basis $\{\mu_j(X)\}$ are N th roots of unity.

Proof. We proceed through a series of games to show that the protocol defined in Fig. 4 satisfies knowledge soundness. We set Game_0 to be the knowledge soundness game as defined in Definition A.1 and consider an algebraic adversary \mathcal{A} against it which has advantage $\text{Adv}_{\mathcal{A}}^{\text{k-sound}}(\lambda)$. We define Game_1 and Game_2 and specify reductions \mathcal{B}_1 and \mathcal{B}_2 such that

$$\begin{aligned} \text{Adv}_{\mathcal{A}}^{\text{k-sound}}(\lambda) &= \text{Adv}_{\mathcal{A}}^{\text{Game}_0}(\lambda) \leq \text{Adv}_{\mathcal{A}}^{\text{Game}_1}(\lambda) + \text{Adv}_{\mathcal{B}_1}^{\text{qSDH}}(\lambda) \\ &\leq \text{Adv}_{\mathcal{A}}^{\text{Game}_2}(\lambda) + \text{Adv}_{\mathcal{B}_1}^{\text{qSDH}}(\lambda) + \text{Adv}_{\mathcal{B}_2}^{\text{qDHE}}(\lambda) \\ &\leq \text{Adv}_{\mathcal{A}}^{\text{Game}_3}(\lambda) + \text{Adv}_{\mathcal{B}_1}^{\text{qSDH}}(\lambda) + \text{Adv}_{\mathcal{B}_2}^{\text{qDHE}}(\lambda) + \text{Adv}_{\mathcal{B}_3}^{\text{qSFrac}}(\lambda) \\ &\leq \text{Adv}_{\mathcal{B}_1}^{\text{qSDH}}(\lambda) + \text{Adv}_{\mathcal{B}_2}^{\text{qDHE}}(\lambda) + \text{Adv}_{\mathcal{B}_3}^{\text{qSFrac}}(\lambda) + \text{negl}(\lambda) \end{aligned}$$

Common input: $[u]_1$ where $[u]_1 = [u_0(X)]_1$

Prover: Take as input srs and $u(X)$

Samples blenders $t_1, \dots, t_n \leftarrow \mathbb{F}$.

For $s = 1, \dots, n$, define $u_s(X) = \sum_{j=1}^m (\omega^{ij})^{2^s} \mu_j(X) + t_s z_{V_m}(X)$,

Define $U(X, Y) = \sum_{s=1}^n u_{s-1}(X) \rho_s(Y)$.

Define $\bar{U}(X, Y) = U(X, Y) - u(X) \rho_1(Y)$

Define $h_2(X) = \sum_{s=1}^n \rho_s(Y) H_s(X)$ for $H_s(X) = (u_{s-1}^2(X) - u_s(X)) / z_{V_m}(X)$

Output $([\bar{U}]_1 = [\bar{U}(x^n, x)]_1, [h_2]_1 = [h_2(x^n, x)]_1)$

Verifier: Send challenge $\alpha \in \mathbb{F}$

Prover: Define $h_1(Y) \leftarrow (U^2(\alpha, Y) - \sum_{s=1}^n u_{s-1}^2(\alpha) \rho_s(Y)) / z_{V_n}(Y)$

Output $[h_1]_1 = [h_1(x)]_1$

Verifier: Send challenge $\beta \in \mathbb{F}$

Prover:

$$\begin{aligned} p(Y) &\leftarrow (U^2(\alpha, \beta) - h_1(Y) z_{V_n}(\beta)) - \bar{U}(\alpha, \beta \sigma) + \text{id}(\alpha) \rho_n(\beta) - z_{V_m}(\alpha) h_2(\alpha, Y) \\ (v_1, \pi_1) &\leftarrow \text{KZG.Open}(\text{srs}, u(X), \deg = \perp, X = \alpha) \\ ([\bar{U}(\alpha, x)]_1, \pi_2) &\leftarrow \text{KZG.Open}(\text{srs}, \bar{U}(X, Y), \deg = \perp, X = \alpha) \\ ([h_2(\alpha, x)]_1, \pi_3) &\leftarrow \text{KZG.Open}(\text{srs}, h_2(X, Y), \deg = \perp, X = \alpha) \\ ((0, v_2, v_3), \pi_4) &\leftarrow \text{KZG.Open}(\text{srs}, \bar{U}(\alpha, Y), \deg = n-1, Y = (1, \beta, \beta \sigma)) \\ (0, \pi_5) &\leftarrow \text{KZG.Open}(\text{srs}, p(Y), \deg = n-1, Y = \beta) \end{aligned}$$

Set $([\bar{U}_\alpha]_1 = [\bar{U}(\alpha, x)]_1, [h_{2,\alpha}]_1 = [h_2(\alpha, x)]_1)$ and output $([\bar{U}_\alpha]_1, [h_{2,\alpha}]_1, v_1, v_2, v_3, \pi_1, \pi_2, \pi_3, \pi_4, \pi_5)$

Verifier: Compute $U \leftarrow v_1 \rho_1(\beta) + v_2$, $[P]_1 \leftarrow U^2 - [h_1]_1 z_{V_n}(\beta) - (v_3 + \text{id}(\alpha) \rho_n(\beta)) - z_{V_m}(\alpha) [h_{2,\alpha}]_1$

Accept if and only if

$$\begin{aligned} 1 &= \text{KZG.Verify}(\text{srs}_{\text{KZG}}, [u]_1, \deg = \perp, X = \alpha, v_1, \pi_1) \\ 1 &= \text{KZG.Verify}(\text{srs}_{\text{KZG}}, [\bar{U}]_1, \deg = \perp, X = \alpha, [\bar{U}_\alpha]_1, \pi_2) \\ 1 &= \text{KZG.Verify}(\text{srs}_{\text{KZG}}, [h_2]_1, \deg = \perp, X = \alpha, [h_{2,\alpha}]_2, \pi_3) \\ 1 &= \text{KZG.Verify}(\text{srs}_{\text{KZG}}, [\bar{U}_\alpha]_1, \deg = n-1, Y = (1, \beta, \beta \sigma), (0, v_2, v_3), \pi_4) \\ 1 &= \text{KZG.Verify}(\text{srs}_{\text{KZG}}, [P]_1, \deg = n-1, Y = \beta, 0, \pi_5) \end{aligned}$$

Figure 5: Argument for proving that some polynomial $u(X)$ has N th roots of unity as coefficients in the basis $\{\mu_j(X)\}_{j=1}^m$.

In Game_0 the adversary will return $[u]_1 = [u(x)]$ along with a proof. We define Game_1 identically to Game_0 , but after the adversary returns $[u]_1$ and a proof, Game_1 additionally checks whether for $u(X), \bar{U}_\alpha(X), p(X)$ the algebraic representations of $[u]_1, [\bar{U}_\alpha]_1, [P]_1$, it is true that $u(\alpha) = v_1$, $\bar{U}_\alpha(1) = 0$, $\bar{U}_\alpha(\beta) = v_2$, $\bar{U}_\alpha(\beta\sigma) = v_3$, and $p(\beta) = 0$; and it aborts if one of the conditions does not hold.

The reduction \mathcal{B}_1 takes as input the challenge $[y_1]_1, \dots, [y_q]_1$. It runs the following reduction \mathcal{B}_{KZG} as a subroutine. The \mathcal{B}_{KZG} runs the adversary \mathcal{A} against Game_0 over an srs in which $[x]_1 = [y_1]_1$. Whenever \mathcal{A} returns an output which wins the Game_0 game, if $(f(X), \mathbf{v}, \mathbf{z})$ for some $(f(X), \mathbf{v}, \mathbf{z}) \in \{(u(X), v_1, \alpha), (\bar{U}_\alpha(X), (1, \beta, \sigma\beta), (0, v_2, v_3)), (p(X), \beta, 0)\}$ is such that $f(v_i) \neq z_i$, then \mathcal{B}_{KZG} computes $f(z) = v'$ and a valid proof π' . It outputs $([f(x)]_1, \mathbf{z}, \mathbf{v}, \pi)$ and $([f(x)]_1, \mathbf{z}, \mathbf{v}', \pi')$ and wins evaluation binding as they are both proofs that verify and open to different elements. Then \mathcal{B}_{qSDH} can extract a $qSDH$ solution from these openings following the proof in Theorem 3 of [22]. Thus

$$\text{Adv}_{\mathcal{A}}^{\text{k-sound}}(\lambda) = \text{Adv}_{\mathcal{A}}^{\text{Game}_0}(\lambda) \leq \text{Adv}_{\mathcal{A}}^{\text{Game}_1}(\lambda) + \text{Adv}_{\mathcal{B}_1}^{\text{qSDH}}(\lambda)$$

Now Game_2 behaves identically as Game_1 but it additionally checks that $\deg(\bar{U}_\alpha) \leq n - 1$ and $\deg(h_2) \leq n - 1$. If it is not the case, it aborts. Suppose \mathcal{A} returns either $\deg(\bar{U}_\alpha) = n - 1 + d$ or $\deg(h_2) = n - 1 + d$ for some $d > 0$. We argue the advantage of \mathcal{A} in Game_1 and Game_2 is the same unless we can build an adversary \mathcal{B}_2 that succeeds against $qDHE$. The \mathcal{B}_2 takes as input the challenge $[y_1]_1, \dots, [y_{q+d-1}]_1$ and runs the adversary \mathcal{A} against Game_1 over an srs in which $[x]_1 = [y_1]_1$. Whenever \mathcal{A} returns an output which wins the Game_1 game, if $(f(X), \mathbf{v}, \mathbf{z})$ for

$$(f(X), \mathbf{v}, \mathbf{z}) \in \{(\bar{U}_\alpha(X), (1, \beta, \sigma\beta), (0, v_2, v_3)), (p(X), \beta, 0)\}$$

is such that $f(X)$ has degree greater than $n - 1$, then the corresponding proof $\pi = [q(x)]_1$ has a representation $q(X)$ has degree $q + 1$. Thus \mathcal{B}_2 succeeds in returning $[\pi - \sum_{i=0}^{q-1} x^i]_1$ and

$$\text{Adv}_{\mathcal{A}}^{\text{Game}_1}(\lambda) \leq \text{Adv}_{\mathcal{A}}^{\text{Game}_2}(\lambda) + \text{Adv}_{\mathcal{B}_2}^{\text{qDHE}}(\lambda)$$

We define Game_3 identically to Game_2 , but after the adversary returns $[u]_1$ and a proof, Game_3 additionally checks whether for $\bar{U}(X), h_2(X), \bar{U}_\alpha(X), h_{2,\alpha}(X)$ the algebraic representations of $[\bar{U}]_1, [\bar{U}_\alpha]_1, [h_2]_1, [h_{2,\alpha}]_1$, it is true that

$$\bar{U}_\alpha(X) = \sum_{i,j} \alpha^i \bar{U}_{ni} X^j \text{ and } h_{2,\alpha}(X) = \sum_{i,j} \alpha^i h_{2,ni} X^j$$

and it aborts if one of the conditions does not hold.

The reduction \mathcal{B}_3 against $qSfrac$ [18] takes as input the challenge $[y_1]_1, \dots, [y_q]_1$ and runs the adversary \mathcal{A} against Game_2 over an srs in which $[x]_1 = [y_1]_1$. Whenever \mathcal{A} returns an output which wins the Game_2 game, if $(f(X), V, z)$ for

$$(f(X), \phi(X), z) \in \{(\bar{U}(X), \bar{U}_\alpha(X), \alpha), (h_2(X), h_{2,\alpha}(X), \alpha)\}$$

is such that $\phi(X) \neq \phi'(X) = \sum_{i,j} \alpha^i f_{ni} X^j$, then set π be the proof for $(f(X), \phi(X), z)$. Then \mathcal{B}_3 returns

$$\phi(X) - \phi'(X), (X^n - z), \pi - \left[\frac{f(x) - \phi'(x)}{x^n - z} \right]_1$$

We have that $\deg(\phi(X) - \phi'(X)) < \deg(X^n - z)$ because $\phi(X)$ has degree bounded by $n - 1$. Hence this is as a valid solution and

$$\text{Adv}_{\mathcal{A}}^{\text{Game}_2}(\lambda) \leq \text{Adv}_{\mathcal{A}}^{\text{Game}_3}(\lambda) + \text{Adv}_{\mathcal{B}_3}^{\text{qSfrac}}(\lambda)$$

Lets see that the advantage of \mathcal{A} in Game_3 is negligible.

Consider $h_1(X), h_2(X, Y)$ the algebraic representations of $[h_1]_1, [h_2]_1$. We can use the equations verified by Game_1 and replace the corresponding values in $p(X)$, obtaining

$$\begin{aligned} p(X) &= (v_1\rho_1(\beta) + v_2)^2 - h_1(X)z_{V_n}(\beta) - (v_3 + \text{id}(\alpha)\rho_n(\beta)) - z_{V_m}(\alpha)h_{2,\alpha}(X) \\ &= (u(\alpha)\rho_1(\beta) + \bar{U}_\alpha(\beta))^2 - h_1(X)z_{V_n}(\beta) - (\bar{U}_\alpha(\beta\sigma) + \text{id}(\alpha)\rho_n(\beta)) - z_{V_m}(\alpha)h_{2,\alpha}(X) \\ &= (u(\alpha)\rho_1(\beta) + \bar{U}(\alpha, \beta))^2 - h_1(X)z_{V_n}(\beta) - (\bar{U}(\alpha, \beta\sigma) + \text{id}(\alpha)\rho_n(\beta)) - z_{V_m}(\alpha)h_2(\alpha, X) \end{aligned}$$

From the fact that $p(\beta) = 0$ we get that

$$0 = (u(\alpha)\rho_1(\beta) + \bar{U}(\alpha, \beta))^2 - (\bar{U}(\alpha, \beta\sigma) + \text{id}(\alpha)\rho_n(\beta)) - z_{V_m}(\alpha)h_2(\alpha, \beta) - h_1(\beta)z_{V_n}(\beta)$$

Since $[u]_1, [\bar{U}]_1, [h_1]_1, [h_2]_1$ have been sent by the prover before it sees challenges β , we have that except in the case where $(Y = \beta)$ is a root of the polynomial below, which happens with negligible probability, for all Y ,

$$0 = (u(\alpha)\rho_1(Y) + \bar{U}(\alpha, Y))^2 - (\bar{U}(\alpha, Y\sigma) + \text{id}(\alpha)\rho_n(Y)) - z_{V_m}(\alpha)h_2(\alpha, Y) - h_1(Y)z_{V_n}(Y) \quad (10)$$

Thus we have that

$$\begin{aligned} i = 0 &\Rightarrow 0 = u^2(\alpha) - \bar{U}(\alpha, \sigma^1) - z_{V_m}(\alpha)h_2(\alpha, \sigma^1) \\ 1 \leq i \leq n-1 &\Rightarrow 0 = \bar{U}^2(\alpha, \sigma^i) - \bar{U}(\alpha, \sigma^{i+1}) - z_{V_m}(\alpha)h_2(\alpha, \sigma^i) \\ i = n &\Rightarrow 0 = \bar{U}^2(\alpha, \sigma^{n-1}) - \text{id}(\alpha) - z_{V_m}(\alpha)h_2(\alpha, \sigma^1) \end{aligned}$$

Since $[u]_1, [\bar{U}]_1, [h_2]_1$ have been sent by the prover before it sees challenges α , we have that except in the case where $(X = \alpha)$ is a root of the polynomial below, which happens with negligible probability, for all X ,

$$\begin{aligned} i = 0 &\Rightarrow 0 = u^2(X) - \bar{U}(X, \sigma^1) - z_{V_m}(X)h_2(X, \sigma^1) \\ 1 \leq i \leq n-1 &\Rightarrow 0 = \bar{U}^2(X, \sigma^i) - \bar{U}(X, \sigma^{i+1}) - z_{V_m}(X)h_2(X, \sigma^i) \\ i = n &\Rightarrow 0 = \bar{U}^2(X, \sigma^{n-1}) - \text{id}(X) - z_{V_m}(X)h_2(X, \sigma^1) \end{aligned}$$

Over $\nu \in V_m$ we thus have that

$$\begin{aligned} i = 0 &\Rightarrow 0 = u^2(\nu) - \bar{U}(\nu, \sigma^1) \\ 1 \leq i \leq n-1 &\Rightarrow 0 = \bar{U}^2(\nu, \sigma^i) - \bar{U}(\nu, \sigma^{i+1}) \\ i = n &\Rightarrow 0 = \bar{U}^2(\nu, \sigma^{n-1}) - 1 \end{aligned}$$

Together these gives us the desired requirement that $u^N(\nu) = 1$ for all $\nu \in V_m$ except with negligible probability. \square

Theorem 5. *The protocol in Fig. 4 and 5 implies position-hiding linkability between the vector commitment schemes of C and cm, provided that the zk proof for R'_{unity} is instantiated with a the protocol in Fig. 5 and that $\log(N) > 6$.*

The proof is in Appendix F.

8 Optimizations

In this section we describe some optimizations we apply to the protocols in Fig. 4 and 5 in order to achieve the efficiency claimed in Table 1.

Opening t polynomials in one point. As noted in [16],[12], whenever we have t openings of different polynomials at the same point i.e. for $t = 2$ this would be of the form

$$\begin{aligned} \pi_1 &\leftarrow \text{KZG.Open}(\text{srs}_{\text{KZG}}, f_1(X), \deg = d, \alpha) \\ \pi_2 &\leftarrow \text{KZG.Open}(\text{srs}_{\text{KZG}}, f_2(X), \deg = d, \alpha) \end{aligned}$$

then we can send a single opening proof π as opposed to t opening proofs π_1, \dots, π_t .

Batching Pairings. We also apply standard techniques to batch pairings that share the same elements in one of the two groups. Namely, we can aggregate the equations

$$\begin{aligned} e([a]_1, [b_1]_2) &= e([c_1]_1, [d]_2) \text{ and } e([a]_1, [b_2]_2) = e([c_2]_1, [d]_2), \\ \text{as } e([a]_1, [b_1 + \gamma b_2]_2) &= e([c_1 + \gamma c_2]_1, [d]_2) \end{aligned}$$

for γ some random field element sampled by the verifier.

Note that we can adapt KZG openings equations so they can be batched further, namely if we parse the verification pairing as $e([F_1]_1 - s_1 + [Q_1]_1\alpha, [1]_2) = e([Q_1]_1, [x]_2)$, then two openings of different polynomials at different points can be verified by two pairings.

Fig. 1 and 3: In Fig. 1 proofs have the form $([z]_2, [T]_1, [S]_2, \pi_{\text{ped}}, \pi_{\text{unity}})$. See that π_{ped} consists of 1 \mathbb{G}_1 and 2 \mathbb{F} elements. In Fig. 3 proofs have the form $([F]_1, [H]_1, v_1, v_2, \pi_1, \pi_2)$ which amounts to 4 \mathbb{G}_1 and 2 \mathbb{F} . Thus we have a total of 6 \mathbb{G}_1 , 2 \mathbb{G}_2 and 4 \mathbb{F} .

For the verifier, their first pairing check in Fig. 1 uses pairings of the form $e(*, [1]_2)$, $e(*, [z]_2)$, and $e([h]_1, *)$ amounting to 3 pairings. The Pedersen verifier uses no pairings. In Fig. 3 we have a KZG verifier which uses pairings of the form $e(*, [1]_2)$, $e(*, [x]_2)$, and a pairing check that uses pairings of the form $e(*, [1]_2)$, $e(*, [z]_2)$, and $e(*, [x]_2)$. Thus we can batch the pairing checks to get a total of 4 unique pairings over the two constructions.

Fig. 4 and 5: In Fig. 4 proofs have the form $([C_I]_1, [z_I]_1, [u]_1, [H_1]_2, [H_2]_1, v_1, v_2, \pi_1, \pi_2, \pi_3, \pi_{\text{unity}'})$. Here the π_1, π_3 are both openings at the same α and can be batched into one proof. Thus there are 7 \mathbb{G}_1 , 1 \mathbb{G}_2 and 2 \mathbb{F} in addition to the $\pi_{\text{unity}'}$. In 3 proofs have that form $([\bar{U}]_1, [h_2]_1, [h_1]_1, [\bar{U}_\alpha]_1, [h_{2,\alpha}]_1, v'_1, v'_2, v'_3, \pi'_1, \pi'_2, \pi'_3, \pi'_4, \pi'_5)$. Here we can send the same verifier challenge α in both Fig. 4 and Fig. 5 (assuming we run the protocols in parallel) which allows us to avoid sending v'_1, π'_1 in Fig. 5. Further, this allows us to batch the proofs (π'_2, π'_3) with the proof for (π_1, π_3) because these all use the same α . Thus $\pi_{\text{unity}'}$ contributes 7 \mathbb{G}_1 , and 2 \mathbb{F} . Thus we have a total of 14 \mathbb{G}_1 , 1 \mathbb{G}_2 and 4 \mathbb{F} .

For the verifier, their pairing check in Fig. 4 uses pairings of the form $e(*, [1]_2)$ and $e([z_I]_1)$. We also have 3 KZG verifiers which use pairings of the form $e(*, [1]_2)$, $e(*, [x]_2)$. This amounts to 2 batched pairings. In Fig. 3 we have a 5 KZG verifiers. Two use a degree check and thus use pairings of the form $e(*, [1]_2)$, $e(*, [x]_2)$, and $e(*, [x^{d-n+1}]_2)$. The others have the usual pairings as these do not have degree checks. Thus we can batch the pairing checks to get a total of 4 unique pairings over the two constructions.

9 Implementation

We have implemented our scheme in Rust using the arkworks library [2], and have released the implementation in open source. The code contains a subroutine that computes all KZG openings, which we need for fast proof preprocessing and which can be used in other projects. For all the schemes different from Caulk, we used the Legosnark implementation⁴. All the benchmarks included in this section have been obtained by running the corresponding codes in a laptop with CPU i7-8565U and 8GB of RAM; which allowed us to run the code for public sets of size up to 2^{22} for the single case and 2^{20} for lookups.

In Table 2 we compare Caulk’s prover and verifier time as well as proof size with its alternatives in the scenario where $m = 1$ and for different values of N . In Figure 6, we highlight prover time in the y axis, while N is represented in the x axis on a logarithmic scale. We consider the following schemes:

- Caulk: the $m = 1$ version;
- MT-Pos: SNARKed Merkle Poseidon tree with N elements.
- MT-SHA: SNARKed Merkle SHA-2 tree with N elements.
- Harisa [11]: RSA-2048 accumulator of N elements.

⁴<https://github.com/matteocam/libsnark-lego/>

We see that Caulk’s prover is almost 100 times as fast as Merkle trees instantiated with a Poseidon Hash and Groth16 zkSNARK on top, and 10 times as fast as the RSA accumulator. Although the latter stays constant while Caulk’s time grows slowly, we claim Caulk will still perform better for all values N that can be considered practical.

	Prover Time (s)					Verifier Time (s)					Proof Size (KB)
$\log(N) =$	6	10	14	18	22	6	10	14	18	22	
MTPos	2.360	4.235	5.279	6.881	8.953	0.025	0.027	0.026	0.028	0.300	0.290
MTSha	52.310	77.619	110.183	141.280	160.027	0.030	0.028	0.028	0.026	0.027	0.290
Harisa	0.029					0.011					1.170
Caulk	0.0164	0.0164	0.0249	0.0294	0.0299	0.009	0.009	0.009	0.011	0.011	0.600

Table 2: Comparison Table for individual openings

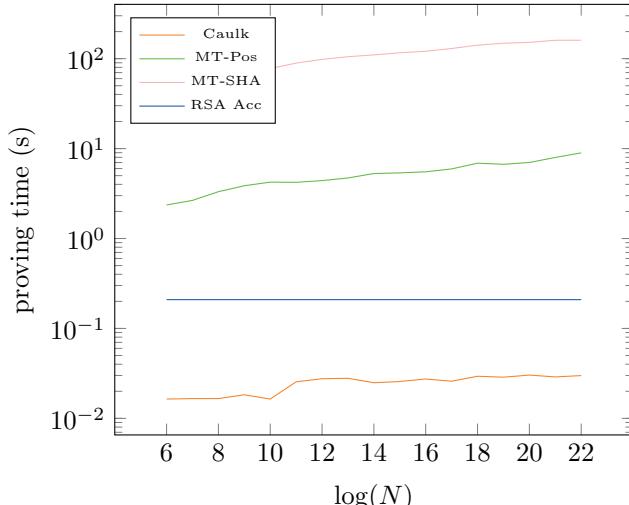


Figure 6: Comparison for single openings

	Prover Time (s)					Verifier Time (s)					Proof Size (KB)
$m =$	10	16	20	32	50	10	16	20	32	50	
MTPos8	26.820	41.290	53.027	81.605	126.940	0.027	0.028	0.031	0.031	0.032	0.290
Caulk8	0.087	0.113	0.183	0.255	0.469	0.038	0.042	0.043	0.044	0.042	0.890
Harisa	1.228	2.014	2.374	3.939	6.011	0.011	0.011	0.012	0.012	0.013	1.170
MTPos20	69.715	102.607	128.766	200.975	271.400	0.029	0.033	0.032	0.027	0.028	0.290
Caulk20	0.565	0.803	0.991	1.468	2.767	0.045	0.046	0.041	0.043	0.0483	0.890

Table 3: Comparison table for lookups

We compare Caulk’s performance for lookup tables in Table 3 with its most direct competitors. We consider the following schemes:

- MTPos-20: SNARKed Merkle tree with Poseidon hashes and $N = 2^{20}$ elements.
- MTPos-8: SNARKed Merkle tree with Poseidon hashes and $N = 2^8$ elements.
- Caulk-8: Caulk for vectors of size $N = 2^8$.
- Caulk-20: Caulk for vectors of size $N = 2^{20}$.
- Harisa [11]: RSA-2048 accumulator for vectors of size $N = 2^{16}$ elements. The performance of the prover in RSA accumulators is independent on the size of the vector.

In Figure 7, the y axis represent prover time, while the x axis represent the value of m . The size of the vector is different for every color line⁵. Caulk is faster than Harisa for all the values of N we were able to compute, but approaches as N grows, and will perform worse for bigger tables. Also, we consider small values for m (up to 50) but we expect that for larger values of m the quadratic component of Caulk’s prover time would make it unpractical. Both constructions are significantly faster than Merkle-SNARK.

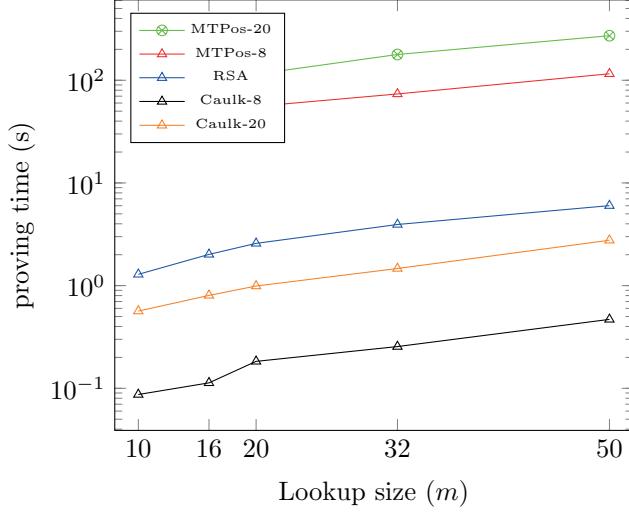


Figure 7: Comparison for lookup tables

For pre-processing the powers of x in \mathbb{G}_1 and \mathbb{G}_2 as well as the single opening proofs, we use a laptop Dell XPS 17, CPU: Intel Core i9-11900H @2.5 Ghz, 16 GB RAM. The computation was single-core and the times are shown in Table 4.

$\log(N)$	8	12	16	20
Time(sec)	3.5	100	874	32830

Table 4: Pre-processing times

Acknowledgments

We thank Matteo Campanelli for his help on running the LegoSNARK code.

⁵The values of m have been chosen over the available numbers for RSA accumulators in the Legosnark codebase.

References

- [1] M. R. Albrecht, L. Grassi, C. Rechberger, A. Roy, and T. Tiessen. MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity. In *ASIACRYPT 2016*, volume 10031 of *LNCS*, pages 191–219, 2016.
- [2] arkworks contributors. `arkworks` zksnark ecosystem, 2022.
- [3] S. Bayer and J. Groth. Zero-knowledge argument for polynomial evaluation with application to blacklists. In T. Johansson and P. Q. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*, pages 646–663. Springer, 2013.
- [4] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 459–474. IEEE Computer Society, 2014.
- [5] D. Benarroch, M. Campanelli, D. Fiore, K. Gurkan, and D. Kolonelos. Zero-knowledge proofs for set membership: Efficient, succinct, modular. In N. Borisov and C. Díaz, editors, *Financial Cryptography and Data Security - 25th International Conference, FC 2021, Virtual Event, March 1-5, 2021, Revised Selected Papers, Part I*, volume 12674 of *Lecture Notes in Computer Science*, pages 393–414. Springer, 2021.
- [6] D. Boneh and X. Boyen. Short signatures without random oracles. In *EUROCRYPT*, volume 3027 of *Lecture Notes in Computer Science*, pages 56–73. Springer, 2004.
- [7] J. Bootle, A. Cerulli, P. Chaidos, E. Ghadafi, J. Groth, and C. Petit. Short accountable ring signatures based on DDH. In G. Pernul, P. Y. A. Ryan, and E. R. Weippl, editors, *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part I*, volume 9326 of *Lecture Notes in Computer Science*, pages 243–265. Springer, 2015.
- [8] J. Bootle and J. Groth. Efficient batch zero-knowledge arguments for low degree polynomials. In M. Abdalla and R. Dahab, editors, *Public-Key Cryptography - PKC 2018 - 21st IACR International Conference on Practice and Theory of Public-Key Cryptography, Rio de Janeiro, Brazil, March 25-29, 2018, Proceedings, Part II*, volume 10770 of *Lecture Notes in Computer Science*, pages 561–588. Springer, 2018.
- [9] J. Camenisch, R. Chaabouni, and A. Shelat. Efficient protocols for set membership and range proofs. In J. Pieprzyk, editor, *Advances in Cryptology - ASIACRYPT 2008, 14th International Conference on the Theory and Application of Cryptology and Information Security, Melbourne, Australia, December 7-11, 2008. Proceedings*, volume 5350 of *Lecture Notes in Computer Science*, pages 234–252. Springer, 2008.
- [10] J. Camenisch and A. Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In M. Yung, editor, *Advances in Cryptology - CRYPTO 2002, 22nd Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 2002, Proceedings*, volume 2442 of *Lecture Notes in Computer Science*, pages 61–76. Springer, 2002.
- [11] M. Campanelli, D. Fiore, S. Han, J. Kim, D. Kolonelos, and H. Oh. Succinct zero-knowledge batch proofs for set accumulators. *IACR Cryptol. ePrint Arch.*, page 1672, 2021.
- [12] A. Chiesa, Y. Hu, M. Maller, P. Mishra, P. Vesely, and N. Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In A. Canteaut and Y. Ishai, editors, *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Virtual Conference, May 1-15, 2020, Proceedings, Part I*, volume 12105 of *Lecture Notes in Computer Science*, pages 738–768. Springer, 2020.
- [13] D. Feist and D. Khovratovich. Fast amortized kate proofs.

- [14] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In A. M. Odlyzko, editor, *Advances in Cryptology - CRYPTO 1986*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer, 1987.
- [15] G. Fuchsbauer, E. Kiltz, and J. Loss. The algebraic group model and its applications. In H. Shacham and A. Boldyreva, editors, *CRYPTO 2018, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part II*, volume 10992 of *LNCS*, pages 33–62. Springer, 2018.
- [16] A. Gabizon and Z. J. Williamson. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *IACR Cryptol. ePrint Arch.*, page 953, 2019.
- [17] A. Gabizon and Z. J. Williamson. plookup: A simplified polynomial protocol for lookup tables. *IACR Cryptol. ePrint Arch.*, page 315, 2020.
- [18] E. Ghadafi and J. Groth. Towards a classification of non-interactive computational assumptions in cyclic groups. *IACR Cryptol. ePrint Arch.*, page 343, 2017.
- [19] L. Grassi, D. Khovratovich, A. Roy, C. Rechberger, and M. Schafnecker. Poseidon: A new hash function for zero-knowledge proof systems. *Usenix Security 2021*, 2021.
- [20] J. Groth. On the size of pairing-based non-interactive arguments. In *EUROCRYPT (2)*, volume 9666 of *Lecture Notes in Computer Science*, pages 305–326. Springer, 2016.
- [21] J. Groth and M. Kohlweiss. One-out-of-many proofs: Or how to leak a secret and spend a coin. In E. Oswald and M. Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26–30, 2015, Proceedings, Part II*, volume 9057 of *Lecture Notes in Computer Science*, pages 253–280. Springer, 2015.
- [22] A. Kate, G. M. Zaverucha, and I. Goldberg. Constant-size commitments to polynomials and their applications. In *ASIACRYPT*, volume 6477 of *Lecture Notes in Computer Science*, pages 177–194. Springer, 2010.
- [23] M. Kohlweiss, M. Maller, J. Siim, and M. Volkov. Snarky ceremonies. In *ASIACRYPT (3)*, volume 13092 of *Lecture Notes in Computer Science*, pages 98–127. Springer, 2021.
- [24] U. M. Maurer. Unifying zero-knowledge proofs of knowledge. In *AFRICACRYPT*, volume 5580 of *Lecture Notes in Computer Science*, pages 272–286. Springer, 2009.
- [25] I. Miers, C. Garman, M. Green, and A. D. Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19–22, 2013*, pages 397–411. IEEE Computer Society, 2013.
- [26] C. Papamanthou, E. Shi, and R. Tamassia. Signatures of correct computation. In *TCC*, volume 7785 of *Lecture Notes in Computer Science*, pages 222–242. Springer, 2013.
- [27] L. Pearson, J. Fitzgerald, H. Masip, M. Bellés-Muñoz, and J. L. Muñoz-Tapia. Plonkup: Reconciling plonk with plookup. *IACR Cryptol. ePrint Arch.*, page 86, 2022.
- [28] A. Tomescu, I. Abraham, V. Buterin, J. Drake, D. Feist, and D. Khovratovich. Aggregatable subvector commitments for stateless cryptocurrencies. In C. Galdi and V. Kolesnikov, editors, *Security and Cryptography for Networks - 12th International Conference, SCN 2020, Amalfi, Italy, September 14–16, 2020, Proceedings*, volume 12238 of *Lecture Notes in Computer Science*, pages 45–64. Springer, 2020.
- [29] Tornado cash privacy solution version 1.4, 2021. https://tornado.cash/Tornado.cash_whitelist_v1.4.pdf.
- [30] ZCash protocol specification, 2022, 1st February. <https://github.com/zcash/zips/blob/master/protocol/protocol.pdf>.
- [31] Zksync rollup protocol, 2021. <https://github.com/matter-labs/zksync/blob/master/docs/protocol.md>.

A Definitions

Let \mathcal{R} be a family of universal relations. Given a relation $R \in \mathcal{R}$ and an instance x we call w a *witness* for x if $(x, w) \in R$, $\mathcal{L}(R) = \{x \mid \exists w : (x, w) \in R\}$ is the language of all the x that have a witness w in the relation R , while $\mathcal{L}(R)$ is the language of all the pairs (x, R) such that $x \in \mathcal{L}(R)$. We will assume R it is implicit as prover and verifier input.

Definition A.1. A Non-Interactive Zero-Knowledge Argument of Knowledge is a tuple of PPT algorithms $(\text{Setup}, \text{Prove}, \text{Verify}, \text{Simulate})$ such that:

- $(\text{srs}, x) \leftarrow \text{Setup}(\mathcal{R})$: On input a family of relations \mathcal{R} , Setup outputs a structured reference string srs and a trapdoor x ;
- $\pi \leftarrow \text{Prove}(\text{srs}, (x, w))$: On input a pair $(x, w) \in R$, it outputs a proof π of the fact that $x \in \mathcal{L}(R)$;
- $1/0 \leftarrow \text{Verify}(\text{srs}, x, \pi)$: On input the srs , the instance x and the proof, it produces a bit expressing acceptance (1), or rejection (0);
- $\pi_{\text{sim}} \leftarrow \text{Simulate}(\text{srs}, x, x)$: The simulator has the srs , the trapdoor x and the instance x as inputs and it generates a simulated proof π_{sim} ,

and that satisfies completeness, knowledge soundness and zero-knowledge as defined below.

Completeness: holds if an honest prover will always convince an honest verifier. Formally, $\forall R \in \mathcal{R}, (x, w) \in R$,

$$\Pr \left[\begin{array}{l} \text{Verify}(\text{srs}, x, \pi) = 1 \\ (\text{srs}, x) \leftarrow \text{Setup}(\mathcal{R}) \\ \pi \leftarrow \text{Prove}(\text{srs}, (x, w)) \end{array} \right] = 1.$$

Knowledge-Soundness: captures the fact that a cheating prover cannot, except with negligible probability, create a proof π accepted by the verification algorithm unless it has a witness w such that $(x, w) \in R$. Formally, for all PPT adversaries \mathcal{A} , there exists a PPT extractor \mathcal{E} such that the following probability is negligible in λ

$$\Pr \left[\begin{array}{l} (x, w) \notin R \wedge \text{Verify}(\text{srs}, x, \pi) = 1 \\ (\text{srs}, x) \leftarrow \text{Setup}(\mathcal{R}) \\ (x, \pi) \leftarrow \mathcal{A}(\text{srs}) \\ w \leftarrow \mathcal{E}(\text{srs}, x, \pi) \end{array} \right]$$

Zero-Knowledge: $(\text{Setup}, \text{Prove}, \text{Verify}, \text{Simulate})$ is zero-knowledge if for all $R \in \mathcal{R}$, instances x and PPT adversaries \mathcal{A} ,

$$\Pr \left[\begin{array}{l} \mathcal{A}(\text{srs}, \pi) = 1 \\ (\text{srs}, x) \leftarrow \text{Setup}(\mathcal{R}) \\ x \leftarrow \mathcal{A}(\text{srs}) \\ \pi \leftarrow \text{Prove}(\text{srs}, (x, w)) \end{array} \right] \approx \Pr \left[\begin{array}{l} \mathcal{A}(\text{srs}, \pi_{\text{sim}}) = 1 \\ (\text{srs}, x) \leftarrow \text{Setup}(\mathcal{R}) \\ x \leftarrow \mathcal{A}(\text{srs}) \\ \pi_{\text{sim}} \leftarrow \text{Simulate}(\text{srs}, x, x) \end{array} \right].$$

Definition A.2 (Vector Commitment Scheme). A Vector Commitment Scheme is a tuple of algorithms $(\text{Setup}, \text{Commit}, \text{Open}, \text{Verify})$ such that:

- $(x, \text{srs}) \leftarrow \text{Setup}(\text{par}, d)$: On input the system parameters and a bound d on the size of the vectors, it outputs a structured reference string and trapdoor x .
- $C \leftarrow \text{Commit}(\text{srs}, \vec{v}, r)$: On input the srs , a vector \vec{v} , and randomness r it outputs a commitment C .
- $(v_i, \pi) \leftarrow \text{Open}(\text{srs}, \vec{v}, r, i)$: On input the srs , the vector, its size, the commitment randomness, and a position $i \in [m]$ it outputs $v_i \in \mathbb{F}$ and proof π that v_i is the i th element of vector \vec{v} .
- $1/0 \leftarrow \text{Verify}(\text{srs}, C, i, v_i, \pi)$: On input the srs , the commitment, position, claimed value v_i , and the proof, it outputs a bit indicating acceptance or rejection.

A vector commitment scheme should satisfy the following properties:

Correctness: It captures the fact that an honest prover will always convince an honest verifier. Namely, for all vectors $\vec{v} \in \mathbb{F}^N$ and $i \in [N]$

$$\Pr \left[\begin{array}{l} \text{Verify}(\text{srs}, \mathcal{C}, i, v_i, \pi) = 1 \\ \quad \text{srs} \leftarrow \text{Setup}(\text{par}, N) \\ \quad \mathcal{C} \leftarrow \text{Commit}(\text{srs}, \vec{v}, r) \\ (v_i, \pi) \leftarrow \text{Open}(\text{srs}, \vec{v}, r, i) \end{array} \right] = 1$$

(Weak) Position Binding: Captures the fact that no PPT adversary \mathcal{A} should be able to present for one commitment two valid openings for the same position. Formally:

$$\Pr \left[\begin{array}{l} \text{Verify}(\text{srs}, \mathcal{C}, i, y, \pi) = 1, \quad \text{srs} \leftarrow \text{Setup}(\text{par}, N) \\ \text{Verify}(\text{srs}, \mathcal{C}, i, y', \pi') = 1 \quad (\vec{v}, r, i, y, y', \pi, \pi') \leftarrow \mathcal{A}(\text{srs}) \\ \quad \text{and } y \neq y' \quad \mathcal{C} \leftarrow \text{Commit}(\text{srs}, \vec{v}, r) \end{array} \right] \approx 0$$

(Strong) Position Binding: Captures the fact that no PPT adversary \mathcal{A} should be able to present for one commitment two valid openings for the same position. Formally:

$$\Pr \left[\begin{array}{l} \text{Verify}(\text{srs}, \mathcal{C}, i, y, \pi) = 1, \quad \text{srs} \leftarrow \text{Setup}(\text{par}, N) \\ \text{Verify}(\text{srs}, \mathcal{C}, i, y', \pi') = 1 \quad (\mathcal{C}, i, y, y', \pi, \pi') \leftarrow \mathcal{A}(\text{srs}) \\ \quad \text{and } y \neq y' \end{array} \right] \approx 0$$

Knowledge Soundness: Captures the fact that whenever the prover provides a valid opening, it knows a valid pair $(p(X), p(\alpha)) \in \mathbb{F}[X] \times \mathbb{F}$, where $\deg(p) \leq \deg$. Formally, for all PPT adversaries \mathcal{A} there exists an efficient extractor \mathcal{E} such that:

$$\Pr \left[\begin{array}{l} \text{Verify}(\text{srs}, \mathcal{C}, i, y, \pi) = 1 \quad \text{srs} \leftarrow \text{Setup}(\text{par}, N) \\ \quad \wedge v_i \neq y \quad \mathcal{C} \leftarrow \mathcal{A}(\text{srs}) \\ \quad \vec{v} \leftarrow \mathcal{E}(\text{srs}, \mathcal{C}, N) \\ (i, y, \pi) \leftarrow \mathcal{A}(\text{srs}, \vec{v}, N, i) \end{array} \right] \approx 0$$

Definition A.3 (Polynomial Commitment Scheme). A Polynomial Commitment Scheme is a tuple of algorithms (Setup , Commit , Open , Verify) such that:

- $(x, \text{srs}) \leftarrow \text{Setup}(\text{par}, d)$: On input the system parameters and a degree bound d , it outputs a structured reference string and trapdoor x .
- $\mathcal{C} \leftarrow \text{Commit}(\text{srs}, p(X), r)$: On input the srs and a polynomial $p(X)$, and randomness r it outputs a commitment \mathcal{C} to $p(X)$.
- $(s, \pi) \leftarrow \text{Open}(\text{srs}, p(X), r, \alpha)$: On input the srs, the polynomial, commitment randomness r , a query point $\alpha \in \mathbb{F}$, it outputs $s \in \mathbb{F}$ and an evaluation proof π that $s = p(\alpha)$.
- $1/0 \leftarrow \text{Verify}(\text{srs}, \mathcal{C}, \deg, \alpha, s, \pi)$: On input the srs, the commitment, degree bound, query and evaluation points α, s , and the proof of correct evaluation, it outputs a bit indicating acceptance or rejection.

A polynomial commitment scheme should satisfy the following properties:

Completeness: It captures the fact that an honest prover will always convince an honest verifier. Formally, for any polynomial $p(X)$ such that $\deg(p) \leq d$ and query point $\alpha \in \mathbb{F}$ the following probability is 1:

$$\Pr \left[\begin{array}{l} \text{Verify}(\text{srs}, \mathcal{C}, \deg, \alpha, s, \pi) = 1 \quad \text{srs} \leftarrow \text{Setup}(\text{par}, d) \\ \quad \mathcal{C} \leftarrow \text{Commit}(\text{srs}, p(X), r) \\ \quad s = p(\alpha), \deg(p) = \deg \\ (s, \pi) \leftarrow \text{Open}(\text{srs}, p(X), r, \alpha) \end{array} \right]$$

Soundness: Captures the fact that a cheating prover should not be able to convince the verifier of a false opening. Formally, for all stateful PPT adversaries \mathcal{A} :

$$\Pr \left[\begin{array}{l} (p(\alpha) \neq s \vee \deg(p) > \deg) \quad \text{srs} \leftarrow \text{Setup}(\text{par}, d) \\ \wedge \quad (p(X), \mathcal{C}) \leftarrow \mathcal{A}(\text{srs}) \\ \text{Verify}(\text{srs}, \mathcal{C}, \deg, \alpha, s, \pi) = 1 \quad \alpha \leftarrow \mathbb{F} \\ \quad (s, \pi) \leftarrow \mathcal{A}(\alpha) \end{array} \right] \approx 0$$

Evaluation Binding: Captures the fact that no PPT adversary \mathcal{A} should be able to present two valid openings for different values but same evaluation point. Formally:

$$\Pr \left[\begin{array}{l} \text{Verify}(\text{srs}, C, \deg, \alpha, s, \pi) = 1, \\ \text{Verify}(\text{srs}, C, \deg, \alpha, s', \pi') = 1 \\ \text{and } s \neq s' \end{array} \middle| \begin{array}{l} \text{srs} \leftarrow \text{Setup}(\text{par}, N) \\ (C, \alpha, s, s', \pi, \pi') \leftarrow \mathcal{A}(\text{srs}) \end{array} \right] \approx 0$$

Extractability: Captures the fact that whenever the prover provides a valid opening, it knows a valid pair $(p(X), p(\alpha)) \in \mathbb{F}[X] \times \mathbb{F}$, where $\deg(p) \leq \deg$. Formally, for all PPT adversaries \mathcal{A} there exists an efficient extractor \mathcal{E} such that:

$$\Pr \left[\begin{array}{l} \text{Verify}(\text{srs}, C, \deg, \alpha, s, \pi) = 1 \\ \wedge \\ (p(\alpha) \neq s \vee \deg(p) > \deg) \end{array} \middle| \begin{array}{l} \text{srs} \leftarrow \text{Setup}(\text{par}, \deg) \\ C \leftarrow \mathcal{A}(\text{srs}) \\ p(X) \leftarrow \mathcal{E}(\text{srs}, C, \deg) \\ \alpha \leftarrow \mathcal{A}(\text{srs}, C, \deg) \\ (s, \pi) \leftarrow \mathcal{A}(\text{srs}, p(X), \deg, \alpha) \end{array} \right] \approx 0$$

B Proof of Thm 1

Proof. We will proceed through a series of games to show that the protocol defined in Fig. 1 satisfies the linkability property. Let \mathcal{A} be an arbitrary algebraic PPT adversary in the linkability game and let $\text{Adv}_{\mathcal{A}}^{\text{linkability}}(\lambda)$ be their advantage. Let Game_0 be defined as in Definition 5.1, which is where we want to bound the adversary's success probability. We define Game_1 , Game_2 and denote $\text{Adv}_{\mathcal{A}}^{G_i}$ as the advantage of the adversary \mathcal{A} in game i . We also specify reductions $\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3, \mathcal{B}_4$ such that

$$\begin{aligned} \text{Adv}_{\mathcal{A}}^{\text{linkability}} &= \text{Adv}_{\mathcal{A}}^{\text{Game}_0} \leq \text{Adv}_{\mathcal{A}}^{\text{Game}_1}(\lambda) + \text{Adv}_{\mathcal{B}_1}^{\text{unity}}(\lambda) \\ &\leq \text{Adv}_{\mathcal{A}}^{\text{Game}_2}(\lambda) + \text{Adv}_{\mathcal{B}_2}^{\text{ped}}(\lambda) + \text{Adv}_{\mathcal{B}_1}^{\text{unity}}(\lambda) \\ &\leq \text{Adv}_{\mathcal{B}_1}^{\text{unity}}(\lambda) + \text{Adv}_{\mathcal{B}_2}^{\text{ped}}(\lambda) + \text{Adv}_{\mathcal{B}_3}^{\text{dlog}}(\lambda) + \text{Adv}_{\mathcal{B}_4}^{\text{qSDH}}(\lambda) \end{aligned}$$

In Game_0 the adversary will return cm along with a proof $([z]_2 = [z(x)]_2, [T]_1 = [T(x)]_1, [S]_2 = [S(x)]_2, \pi_{\text{ped}}, \pi_{\text{unity}})$. We define Game_1 identically to Game_0 , but after the adversary returns cm along with the proof, Game_1 additionally checks whether there exists a, b such that $z(X) = a(X - b)$ with $a^N = b^N$ and abort if this is not the case. Note that Game_1 can extract $z(X)$, the algebraic representation of $[z]_2$, because the adversary \mathcal{A} is algebraic.

We observe that the adversary's advantage in Game_0 and Game_1 is identical, unless it manages to break the knowledge soundness of R_{unity} . Given such an \mathcal{A} , we can thus directly get a reduction \mathcal{B}_1 against the knowledge soundness of R_{unity} and let the advantage of this adversary be $\text{Adv}_{\mathcal{B}_1}^{\text{unity}}$. The reduction \mathcal{B}_1 simply runs \mathcal{A} and returns π_{unity} that is returned by \mathcal{A} . It thus holds that

$$\text{Adv}_{\mathcal{A}}^{\text{linkability}}(\lambda) = \text{Adv}_{\mathcal{A}}^{\text{Game}_0}(\lambda) \leq \text{Adv}_{\mathcal{A}}^{\text{Game}_1}(\lambda) + \text{Adv}_{\mathcal{B}_1}^{\text{unity}}(\lambda).$$

Now define Game_2 , which is identical to Game_1 , but after the (algebraic) adversary \mathcal{A} outputs cm the game Game_2 extracts v and r such that $\text{cm} = [v + hr]_1$. If this extraction fails, meaning that cm is not correctly formed, then Game_2 aborts. We note that the \mathcal{A} 's advantage in Game_1 is identical to its advantage in Game_2 , unless it manages to break the knowledge soundness of R_{ped} . Given \mathcal{A} , we can construct a reduction \mathcal{B}_2 against the knowledge soundness of R_{ped} analogously to the reduction above and let the advantage of this adversary be $\text{Adv}_{\mathcal{B}_2}^{\text{ped}}$. We observe that

$$\text{Adv}_{\mathcal{A}}^{\text{Game}_1}(\lambda) \leq \text{Adv}_{\mathcal{A}}^{\text{Game}_2}(\lambda) + \text{Adv}_{\mathcal{B}_2}^{\text{ped}}(\lambda).$$

Recall that any adversary who successfully wins Game_2 must output a proof that satisfies the following equation from the verification procedure

$$\begin{aligned} C(x) - v - hr &= T(x)z(x) + hS(x) \Leftrightarrow \\ C(x) - v &= T(x)a(x - \omega^i) + h(r + S(x)), \end{aligned}$$

while at the same time it must hold that

$$C(X) - v \neq (X - \omega^i)aT(X)$$

for any polynomial $aT(X)$, since v is not in the committed vector \vec{c} . Intuitively, the adversary cannot satisfy this equation, since \mathbf{h} is unknown to the prover and thus $(r + S(X))$ is chosen independently of \mathbf{h} . More formally, we consider two cases here. If

$$C(x) - v \neq T(x)a(x - \omega^i)$$

then we can construct a reduction \mathcal{B}_3 breaking the discrete logarithm problem. Else if

$$C(x) - v = T(x)a(x - \omega^i)$$

then we can construct a reduction \mathcal{B}_4 breaking the q SDH problem.

The reduction \mathcal{B}_3 takes as input a challenge $[y]_1$. It runs the adversary \mathcal{A} against Game_2 over an srs in which $[\mathbf{h}]_1 = [y]_1$ and \mathcal{B}_3 's choice of x (where x is the trapdoor information of the KZG commitment). Whenever the adversary returns an output $([z]_2 = [z(x)]_2, [T]_1 = [T(x)]_1, [S]_2 = [S(x)]_2, \pi_{\text{ped}}, \pi_{\text{unity}})$ which wins the Game_2 game, then \mathcal{B}_3 returns

$$\mathbf{h} = \frac{C(x) - v - T(x)z(x)}{r + S(x)},$$

where $T(X), r$ and $S(X)$ are extracted from the outputs of \mathcal{A} . The reduction's success probability is exactly the success probability of the adversary conditioned on $(r + S(x)) \neq 0$.

The reduction \mathcal{B}_4 takes as input the challenge $[y_1]_1, \dots, [y_q]_1$. It runs the following reduction \mathcal{B}_4 as a subroutine. The \mathcal{B}_{KZG} runs the adversary \mathcal{A} against Game_2 over an srs in which $[x]_1 = [y_1]_1$ and \mathcal{B}_{KZG} 's choice of \mathbf{h} . Whenever the adversary returns an output $([z]_2 = [z(x)]_2, [T]_1 = [T(x)]_1, [S]_2 = [S(x)]_2, \pi_{\text{ped}}, \pi_{\text{unity}})$ which wins the Game_2 game, then \mathcal{B}_{KZG} returns the KZG openings

$$(v, [a^{-1}T]_1) \text{ and } (C(\omega^i), [\frac{C(x) - C(\omega^i)}{x - \omega^i}]_1)$$

for $v \neq C(x)$. Then \mathcal{B}_4 can extract a q SDH solution from these openings following the proof in Theorem 1 of [22].

We can thus conclude that

$$\text{Adv}_{\mathcal{A}}^{\text{linkability}}(\lambda) \leq \text{Adv}_{\mathcal{B}_1}^{\text{uniquity}}(\lambda) + \text{Adv}_{\mathcal{B}_2}^{\text{ped}}(\lambda) + \text{Adv}_{\mathcal{B}_3}^{\text{dlog}}(\lambda) + \text{Adv}_{\mathcal{B}_4}^{\text{qSDH}}(\lambda).$$

Lastly, we prove the position hiding property of our construction. We define a simulator Simulate that has access to the trapdoor x of srs that is indistinguishable from an honest prover. First, Simulate calls the simulators of R_{ped} and R_{unity} on input the trapdoor x , and gets simulated proofs π'_{ped} and π'_{unity} . Then, it samples $a, r, s \leftarrow \mathbb{F}$ and sets $[z']_2 = [a]_2, [S']_2 = [s]_2, [T']_1 = (\mathbf{C} \cdot \mathbf{cm}^{-1} - [\mathbf{hs}]_1)/a$, and outputs $([z']_2, [T']_1, [S']_2, \pi'_{\text{ped}}, \pi'_{\text{unity}})$. Note that honestly generated $[z]_2, [S]_2$ are randomized by a and s , respectively, and thus indistinguishable from $[z']_2, [S']_2$. Finally, $[T']_1$ is the only element satisfying the verifying equation for given $[z']_2, [S']_2$ and thus indistinguishable from honest $[T]_1$ as well, which concludes the proof. \square

C Proof of Lemma 1

Proof. Because $z(X)$ has degree 1, there exist $a, b \in \mathbb{F}$ such that $z(X) = aX - b$.

From the first condition, we have $f(1) = a(1) = a - b$, and $f(\sigma) = a(\sigma) = a\sigma - b$. From items 2 and 3,

$$\begin{aligned} f(\sigma^2) &= \frac{f(1) - f(\sigma)}{1 - \sigma} = \frac{a - a\sigma}{1 - \sigma} = a, \\ f(\sigma^3) &= \sigma f(\sigma^2) - f(\sigma) = \sigma a - a\sigma + b = b \end{aligned}$$

By substituting $f(\sigma^2) = a$ and $f(\sigma^3) = b$ into condition 4 we see that $f(\sigma^4) = \frac{a}{b}$. Therefore, from item 5 we have that for every $i = 0, \dots, \log(N) - 1$, $f(\sigma^{4+i+1}) = f(\sigma^{4+i})^2 = (\frac{a}{b})^{2^{i+1}}$. In particular, $f(\sigma^{4+(\log(N)-1)+1}) = (\frac{a}{b})^{2^{\log(N)}} = (\frac{a}{b})^N$, that equals 1 by the 5th condition, proves that $\frac{a}{b}$ is a N th root of unity as required. \square

D Proof of Thm. 2

Proof. We proceed through a series of games to show that the protocol defined in Fig. 3 satisfies knowledge soundness. We set Game_0 to be the soundness game as in Def. A.1 and consider an algebraic adversary \mathcal{A} against it which has advantage $\text{Adv}_{\mathcal{A}}^{\text{k-sound}}$. We define Game_1 , Game_2 and specify reductions \mathcal{B}_1 and \mathcal{B}_2 such that

$$\begin{aligned}\text{Adv}_{\mathcal{A}}^{\text{k-sound}}(\lambda) &= \text{Adv}_{\mathcal{A}}^{\text{Game}_0}(\lambda) \leq \text{Adv}_{\mathcal{A}}^{\text{Game}_1}(\lambda) + \text{Adv}_{\mathcal{B}_1}^{\text{qSDH}}(\lambda) \\ &\leq \text{Adv}_{\mathcal{A}}^{\text{Game}_2}(\lambda) + \text{Adv}_{\mathcal{B}_2}^{\text{qDHE}}(\lambda) + \text{Adv}_{\mathcal{B}_1}^{\text{qSDH}}(\lambda) \\ &\leq \text{Adv}_{\mathcal{B}_1}^{\text{qSDH}}(\lambda) + \text{Adv}_{\mathcal{B}_2}^{\text{qDHE}}(\lambda) + \text{negl}(\lambda).\end{aligned}$$

In Game_0 the adversary will return $[z]_2$ along with a proof $([F]_1 = [f(x)]_1, [H]_2 = [h(x)], v_1, v_2, \pi_1, \pi_2)$. We also consider $\hat{p}(X)$, the algebraic representation of $[P]_1$ as constructed by the verifier. Note that π_2 is KZG opening proof for $p(X) = \hat{p}(X) - z(X)(\rho_1(\alpha) + \rho_2(\alpha) - z_{V_n}(\alpha)X^{d-1})$ opening to 0 at α . We define Game_1 identically to knowledge soundness, but after the adversary returns $[z]_2$ along with the proof, Game_2 additionally checks whether $f(\alpha_1) = v_1$, $f(\alpha_2) = v_2$, $p(\alpha) = 0$ and aborts otherwise. Note that Game_1 can extract $f(X), h(x)$ because the adversary \mathcal{A} is algebraic, and $p(X)$ is constructed from them.

We show the probability that $f(\alpha_1) = v_1$, $f(\alpha_2) = v_2$, $p(\alpha) = 0$ is bounded by $q\text{SDH}$. We construct a reduction \mathcal{B}_1 that takes as input a challenge $[y]_1, \dots, [y_q]_1$. It runs the following reduction \mathcal{B}_{KZG} as a subroutine. The \mathcal{B}_{KZG} runs the adversary \mathcal{A} against Game_0 over an srs in which $[x]_1 = [y]_1$. Whenever the adversary returns an output $([F]_1 = [f(x)]_1, [H]_2 = [h(x)], v_1, v_2, \pi_1, \pi_2, \pi_3)$ that wins the Game_0 but not the Game_1 game, then \mathcal{B}_{KZG} returns the KZG openings

$$\begin{aligned}&\left((v_1, [F]_1) \text{ and } (f(\alpha_1), [\frac{f(x) - f(\alpha_1)}{x - \alpha_1}]) \right), \left((v_2, [F]_1) \text{ and } (f(\alpha_2), [\frac{f(x) - f(\alpha_2)}{x - \alpha_2}]) \right) \text{ or} \\ &\left((0, [P]_1) \text{ and } (p(\alpha), [\frac{p(x) - p(\alpha)}{x - \alpha}]) \right)\end{aligned}$$

for either $v_1 \neq f(\alpha_1)$, $v_2 \neq f(\alpha_2)$ or $p(\alpha) \neq 0$. Then \mathcal{B}_1 can extract a $q\text{-SDH}$ solution from these openings following the proof of Theorem 3 in [22]. Thus

$$\text{Adv}_{\mathcal{A}}^{\text{k-sound}}(\lambda) = \text{Adv}_{\mathcal{A}}^{\text{Game}_0}(\lambda) \leq \text{Adv}_{\mathcal{A}}^{\text{Game}_1}(\lambda) + \text{Adv}_{\mathcal{B}_1}^{\text{qSDH}}(\lambda).$$

We define Game_2 as Game_1 except that Game_2 additionally checks whether $\deg(z) \leq 1$ for $z(X)$ being the algebraic representation of $[z]_2$, and aborts otherwise. We show that \mathcal{A} 's advantage in both games is the same unless it breaks $q\text{DHE}$. Indeed, assume $\deg(z) = 2$, we construct an adversary \mathcal{B}_2 against $q\text{DHE}$. The \mathcal{B}_2 takes as input the challenge $[y]_1, \dots, [y_q]_1$ and runs \mathcal{A} against Game_1 over an srs in which $[x]_1 = [y]_1$. When \mathcal{A} returns an output $([F]_1 = [f(x)]_1, [H]_2 = [h(x)], v_1, v_2, \pi_1, \pi_2)$ that wins the Game_1 but not the Game_2 game, then \mathcal{B}_2 extracts $\hat{p}(X) = \sum_{s=0}^{d+1} \hat{p}_s X^s$ as the algebraic representation of $[P]_1$ computed by the verifier. Note that, since $(-\rho_1(\alpha) - \rho_2(\alpha) - z_{V_n}(\alpha)X^{d-1})z(X)$ does not vanish at $X = \alpha$, we have that $\hat{p}_{d+1} \neq 0$. Then, \mathcal{B}_2 sets $\hat{P}(X) = P(X) - \hat{p}_{d+1}X^{d+1}$ and outputs $([P]_1 - [\hat{P}(x)]_1) \frac{1}{\hat{p}_{d+1}} = [x^{d+1}]_1$, winning $d\text{-DHE}$. Thus

$$\text{Adv}_{\mathcal{A}}^{\text{Game}_1}(\lambda) = \text{Adv}_{\mathcal{A}}^{\text{Game}_2}(\lambda) + \text{Adv}_{\mathcal{B}_2}^{\text{qDHE}}(\lambda).$$

Finally, let us show that

$$\text{Adv}_{\mathcal{A}}^{\text{Game}_2}(\lambda) \leq \text{negl}(\lambda).$$

Consider $f(X), h(X)$ the algebraic representations of $[F]_1, [H]_1$. The algebraic representation of the element $[P]_1$ that the verifier constructs is

$$\begin{aligned}p(X) &= -z_{V_n}(\alpha)h(X) + (\rho_1(\alpha) + \rho_2(\alpha))f(X) + \rho_3(\alpha)((1 - \sigma)f(X) + v_1 - v_2) + \rho_4(\alpha)(f(X) + v_2 - \sigma v_1) \\ &\quad + \rho_5(\alpha)(v_1 f(X) - v_2) + \rho_n(\alpha)(v_1 - 1) + \prod_{i \notin [5, \dots, 4 + \log(N)]} (\alpha - \sigma^i)(f(X) - v_1^2)\end{aligned}$$

Since Game_2 checks that $v_1 = f(\sigma^{-1}\alpha), v_2 = f(\sigma^{-2}\alpha)$, we can replace these values and see that

$$\begin{aligned} p(X) &= -z_{V_n}(\alpha)h(X) + (\rho_1(\alpha) + \rho_2(\alpha))f(X) + \rho_3(\alpha)((1-\sigma)f(X) + f(\sigma^{-1}\alpha) - f(\sigma^{-2}\alpha)) \\ &\quad + \rho_4(\alpha)(f(X) + f(\sigma^{-2}\alpha) - \sigma f(\sigma^{-1}\alpha)) + \rho_5(\alpha)(f(\sigma^{-1}\alpha)f(X) - f(\sigma^{-2}\alpha)) \\ &\quad + \rho_n(\alpha)(f(\sigma^{-1}\alpha) - 1) + \prod_{i \notin [5, \dots, 4+\log(N)]} (\alpha - \sigma^i)(f(X) - f(\sigma^{-1}\alpha)^2) \end{aligned}$$

Now, because $p(\alpha) = 0$ and α has been chosen by the verifier after the prover has sent $[H]_1, [F]_1$, except in the negligible case that α is a root of $p(X)$, we have that $p(X) \equiv 0$, i.e,

$$\begin{aligned} z_{V_n}(X)h(X) &= -(\rho_1(X) + \rho_2(X))f(X) + \rho_3(X)((1-\sigma)f(X) + f(\sigma^{-1}X) - f(\sigma^{-2}X)) \\ &\quad + \rho_4(X)(f(X) + f(\sigma^{-2}X) - \sigma f(\sigma^{-1}X)) + \rho_5(X)(f(\sigma^{-1}X)f(X) - f(\sigma^{-2}X)) \\ &\quad + \rho_n(X)(f(\sigma^{-1}X) - 1) + \prod_{i \notin [5, \dots, 4+\log(N)]} (X - \sigma^i)(f(X) - f(\sigma^{-1}X)^2) \end{aligned}$$

$z_{V_n}(X)$ divides the right side of the equation and thus, the latter vanishes for all the powers $\{\sigma^i\}_{i=0}^{n-1}$. This implies that

- $f(1) = a(1), f(\sigma) = a(\sigma)$
- $f(\sigma^2) = \frac{v_2 - v_1}{1-\sigma} = \frac{f(\sigma^2\sigma^{-2}) - f(\sigma^2\sigma^{-1})}{1-\sigma} = \frac{f(1) - f(\sigma)}{1-\sigma}$
- $f(\sigma^3) = rf(\sigma^3\sigma^{-1}) - f(\sigma^3\sigma^{-2}) = rf(\sigma^2) - f(\sigma)$
- $f(\sigma^4)f(\sigma^4\sigma^{-1}) = f(\sigma^4\sigma^{-2})$, i.e, $f(\sigma^4)f(\sigma^3) = f(\sigma^2)$
- $1 = f(\sigma^{5+\log(N)}\sigma^{-1}) = f(\sigma^{4+\log(N)})$
- $(f(\sigma^{4+i+1}) - f(\sigma^{4+i+1}\sigma^{-1})f(\sigma^{4+i+1}\sigma^{-1}))(\sigma^i - \sigma^{5+\log(N)}) \prod_{j=1}^5 (\sigma^i - \sigma^j) = 0$ for all $i = 0, \dots, \log(N) - 1$.
- 1. Note that $\prod_{j \notin [5, \dots, 4+\log(N)]} (\sigma^i - \sigma^j) \neq 0$ implies that $0 = f(\sigma^{4+i+1}) - f(\sigma^{4+i+1}\sigma^{-1})f(\sigma^{4+i+1}\sigma^{-1}) = f(\sigma^{4+i+1}) - f(\sigma^{4+i})^2$.

By Lemma 1 we have that $z(X) = aX - b$ where $\frac{a}{b}$ is an N -th root of unity.

For zero-knowledge, we define a simulator Simulate that has access to the trapdoor of srs and is indistinguishable from an honest prover. The simulator first chooses s_1, s_2, v_1, v_2 uniformly at random and sets $[F]_1 = [s_1]_1$ and $[H]_1 = [s_2]_1$. It computes $\alpha_1 = \sigma^{-1}\alpha, \alpha_2 = \sigma^{-2}\alpha$. It then computes $[w_1]_1 = ([F]_1 - v_1\tau_1(x) - v_2\tau_2(x))\frac{1}{(x-\alpha_1)(x-\alpha_2)}$, for $\tau_1(x) = \frac{x-\alpha_2}{\alpha_1-\alpha_2}, \tau_2(x) = \frac{x-\alpha_1}{\alpha_2-\alpha_1}$.

It sets $[P]_1$ the same as the verifier i.e.

$$\begin{aligned} [P]_1 &= -[H]_1 z_{V_n}(\alpha) + [F]_1(\rho_1(\alpha) + \rho_2(\alpha)) + ([F]_1(1-\sigma) - v_2 + v_1)\rho_3(\alpha) + ([F]_1 + v_2 - \sigma v_1)\rho_4(\alpha) \\ &\quad + ([F]_1 v_1 - v_2)\rho_5(\alpha) + (v_1 - 1)\rho_n(\alpha) + ([F]_1 - v_1^2) \prod_{i \notin [5, \dots, 4+\log(N)]} (\alpha - \sigma^i) \end{aligned}$$

and then computes $[w_2]_1 = ([P]_1 - (\rho_n(\alpha) + \rho_1(\alpha) + z_{V_n}(\alpha)x^{d-1})z)\frac{1}{x-\alpha}$, where $z = a$ is the output of the simulator in the proof of Theorem 1. It returns $([F]_1, [H]_1, v_1, v_2, \pi_1 = [w_1]_1, \pi_2 = [w_2]_2)$.

We must argue that the simulators output is distributed identically to the honest provers. Then the provers components are randomised by

$$\begin{array}{ll} F : & r_0\rho_{5+\log(N)}(x) \\ v_1 : & r(\sigma^{-1}\alpha)z_{V_n}(\alpha) \end{array} \quad \begin{array}{ll} H : & r(x) \\ v_2 : & r(\sigma^{-2}\alpha)z_{V_n}(\alpha) \end{array}$$

and the elements $[w]_1, [w]_2$ are the unique elements satisfying the verifies equations given $[F]_1, [H]_1, v_1, v_2$.

The probability that the values $r_1\rho_{6+\log(N)}(x)$, $r(x)$, $r(\sigma^{-1}\alpha)z_{V_n}(\alpha)$, $r(\sigma^{-2}\alpha)z_{V_n}(\alpha)$ are dependent for random α is negligible because $r(X)$ is a random degree 2 polynomial and the probability that $\sigma^{-1}\alpha = x$ or $\sigma^{-2}\alpha = x$ is $\frac{2}{|\mathbb{F}|}$. Where the simulators terms $[F]_1, [H]_1, v_1, v_2$ are chosen uniformly at random and $[w_1]_1, [w_2]_1$ are the unique terms that satisfy the verifiers equations, we have that these distributions are identical except with negligible probability. \square

E Proof of Thm. 3

Proof. We will proceed through a series of games to show that the protocol defined in Fig. 4 satisfies linkability as defined in Def. 5.1. Let \mathcal{A} be an arbitrary PPT adversary in the linkability game with advantage $\text{Adv}_{\mathcal{A}}^{\text{linkability}}(\lambda)$. We define Game_1 , Game_2 and specify reductions \mathcal{B}_1 and \mathcal{B}_2 such that

$$\text{Adv}_{\mathcal{A}}^{\text{linkability}}(\lambda) \leq \text{Adv}_{\mathcal{B}_1}^{\text{qSDH}}(\lambda) + \text{Adv}_{\mathcal{B}_2}^{\text{k-sound}}(\lambda) + \text{negl}(\lambda)$$

Let us transition from the linkability game for the protocol of Fig. 4 to a game Game_1 . Game_1 behaves as linkability except that when \mathcal{A} returns v_1, v_2 , Game_1 checks whether $u(\alpha) = v_1$, $p_1(v_1) = v_2$, and $p_2(\alpha) = 0$, for $u(X), p_1(X), p_2(X)$, the algebraic representations of $[u]_1, [P_1]_1 = [z_I]_1 + \chi[C_I]_1$, and $[P_2]_1 = v_2 - \chi\text{cm} - z_{V_m}(\alpha)[H_2]_1$. If not then Game_1 aborts. We design \mathcal{B}_1 such that

$$\text{Adv}_{\mathcal{A}}^{\text{linkability}}(\lambda) \leq \text{Adv}_{\mathcal{A}}^{\text{Game}_1}(\lambda) + \text{Adv}_{\mathcal{B}_1}^{\text{qSDH}}(\lambda)$$

Indeed, assume that \mathcal{A} succeeds against linkability but not Game_1 . Then this corresponds to the case where \mathcal{A} returns verifying $v_1, v_2, \pi_1, \pi_2, \pi_3$ but the equality does not hold for some $p(X) \in \{u(X), p_1(X), p_2(X)\}$. Thus \mathcal{B}_1 takes as input a challenge $[y_1]_1, \dots, [y_q]_1$ and runs the following reduction \mathcal{B}_{KZG} as a subroutine. The \mathcal{B}_{KZG} runs the adversary \mathcal{A} against Game_0 over an srs in which $[x]_1 = [y_1]_1$. Whenever the adversary wins the Game_0 but not the Game_1 game, then \mathcal{B}_{KZG} returns the KZG opening

$$(v, \pi) \text{ and } (f(\alpha), [(f(x) - f(\alpha))/(x - \alpha)]_1)$$

for $(v, f(X))$ corresponding to either $(v_1, u(X)), (v_2, p_1(X)), (v_3, p_2(X))$ and π the corresponding proof. Then $\mathcal{B}_{\text{qSDH}}$ can extract a solution from these openings following the proof in Theorem 1 in [22].

Now let us transition to a new game. Game_2 behaves identically except that when \mathcal{A} returns $[u]_1$, then Game_2 checks whether its algebraic representation $u(X)$ is such that $u(\nu^j)^N = 1$ for all j . If not then Game_2 aborts. We design \mathcal{B}_2 such that

$$\text{Adv}_{\mathcal{A}}^{\text{Game}_1}(\lambda) \leq \text{Adv}_{\mathcal{A}}^{\text{Game}_2}(\lambda) + \text{Adv}_{\mathcal{B}_2}^{\text{k-sound}}(\lambda)$$

Assume that \mathcal{A} succeeds against Game_1 but not Game_2 . Then \mathcal{B}_2 chooses $[u]_1 = [u(x)]_1$ in its own game and uses it as input to run \mathcal{A} . When \mathcal{A} returns π_{unity} , \mathcal{B}_2 forwards it and wins knowledge-soundness of Π_{unity} whenever \mathcal{A} succeeds.

Next we transition to a game Game_3 that behaves as Game_2 except that when \mathcal{A} returns its proof, Game_3 checks whether $C(X) - C_I(X) = z_I(X)H_1(X)$, for $C(X), C_I(X), z_I(X), H_1(X)$ the algebraic representations of $[C]_1, [C_I]_1, [H_1]_2, [z_I]_1$. If not then Game_3 aborts. We design \mathcal{B}_3 such that

$$\text{Adv}_{\mathcal{A}}^{\text{Game}_2}(\lambda) \leq \text{Adv}_{\mathcal{A}}^{\text{Game}_3}(\lambda) + \text{Adv}_{\mathcal{B}_3}^{\text{qSDH}}(\lambda)$$

The \mathcal{B}_3 takes as input a challenge $[y_1]_1, \dots, [y_q]_1$ and runs the adversary \mathcal{A} against Game_2 over an srs in which $[x]_1 = [y_1]_1$. Whenever the adversary wins the Game_2 but not the Game_3 game, then \mathcal{B}_3 learns

$$d(X) = C(X) - C_I(X) - z_I(X)H_1(X)$$

such that $d(x) = 0$ and $d(X) \neq 0$. Thus \mathcal{B}_3 returns $(1, [1/(x-1)]_1)$ as a valid q -SDH solution.

Finally we show that the probability that Game_3 returns 1 but that for some $j \in [m]$, and for \vec{c} such that $C(X) = \sum_{i=1}^N c_i \lambda_i(X)$,

$$\phi(\nu^j) \notin \vec{c}$$

is negligible.

Recall that $p_2(\alpha) = v_2 - \chi\text{cm} - z_{V_m}(\alpha)H_2(\alpha) = z_I(v_1) + \chi C_I(v_1) - \chi\text{cm} - z_{V_m}(\alpha)H_2(\alpha) = z_I(u(\alpha)) + \chi C_I(u(\alpha)) - \chi\text{cm} - z_{V_m}(\alpha)H_2(\alpha) = 0$. First, because α has been sent by the verifier after the prover commits to $\phi(X), z_I(X), u(X), H_2(X)$ and $C_I(X)$, we have that

$$z_I(u(X)) + \chi C_I(u(X)) - \chi\phi(X) - z_{V_m}(X)H_2(X) = 0$$

for all X except with negligible probability. Further, because χ has been sent by the verifier after the prover commits, we have that there exists $H_{2,1}(X)$ and $H_{2,2}(X)$ such that

$$\begin{aligned} 0 &= z_I(u(X)) - z_{V_m}(X)H_{2,1}(X) \\ 0 &= C_I(u(X)) - \phi(X) - z_{V_m}(X)H_{2,2}(X) \end{aligned}$$

except with negligible probability.

Thus,

$$z_I(u(\nu^j)) = z_I(\omega^{i_j}) = 0 \text{ for all } j = 1, \dots, m.$$

and $z_I(X) = \prod_{j=1}^m (X - \omega^{i_j})\hat{z}(X) = \prod_{i \in I} (X - \omega^i)\hat{z}(X)$, for some polynomial $\hat{z}(X)$. From the second equation we also have that

$$C_I(u(\nu^j)) = \phi(\nu^j) \quad \forall j \in [m], \text{i.e., } C_I(\omega^{i_j}) = \phi(\nu^j).$$

Using

$$C(u(X)) - C_I(u(X)) = z_I(u(X))H_1(u(X))$$

we hence gets that

$$0 = C(u(\nu^j)) - C_I(u(\nu^j)) = C(\omega^k) - \phi(\nu^j)$$

which concludes the proof. \square

F Proof of Thm. 5

Proof. We first define a simulator **Simulate** and then argue that their transcript is indistinguishable from an honest provers transcript. The **Simulate** subverts the setup algorithm such that it knows the secret x contained in $[x]_1, [x^2]_1, [x^3]_1, \dots$. It takes as input some instance (C, cm) and aims to generate a verifying transcript.

It samples $s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8 \leftarrow \mathbb{F}$ at random and outputs $[C_I]_1 = [s_1]_1, [z_I]_1 = [s_2]_1, [u]_1 = [s_3]_1, [H_1]_2 = [(C - s_1)/s_2]_2$ and a simulated proof π_{unity} that we describe in the next paragraph. After receiving χ it outputs $[H_2]_1 = [s_4]_1$. After receiving α it outputs $v_1 = s_5, v_2 = s_6$, and

$$\begin{aligned} \pi_1 &= [(u - v_1)/(x - \alpha)]_1 \\ \pi_2 &= [(z_I + \chi C_I)/(x - v_1)]_1 \\ \pi_3 &= [(v_2 - \chi\text{cm} - z_{V_m}(\alpha)H_2)/(x - \alpha)]_1 \end{aligned}$$

To simulate π_{unity} the simulate **Simulate** outputs $[\bar{U}]_1 = [s_7]_1, [h_2]_1 = [s_8]_1$. After receiving α it outputs $[h_1]_1 = [s_9]_1$. After receiving β it outputs $[\bar{U}_\alpha]_1 = [s_{10}]_1, [h_{2,\alpha}] = [s_{11}]_1$ and $v_1 = s_{12}, v_2 = s_{13}, v_3 = s_{14}$ and

$$\begin{aligned} \pi_1 &= [(u - v_1)/(x - \alpha)]_1 \\ \pi_2 &= [(\bar{U} + \bar{U}_\alpha)/(x - \alpha)]_1 \\ \pi_3 &= [(h_2 - h_{2,\alpha})/(x - \alpha)]_1 \\ \pi_4 &= [x^{\max_deg-n}(\bar{U}_\alpha + \ell(x))/(x - 1)(x - \beta)(x - \beta\sigma)]_1 \\ \pi_5 &= [x^{\max_deg-n}((v_1\rho_1(\beta) + v_2)^2 - h_1z_{V_n}(\beta) - (v_3 + \text{id}(\alpha)\rho_n(\beta)) - z_{V_m}(\alpha)h_{2,\alpha})/(x - \beta)]_1 \end{aligned}$$

where $\ell(x)$ is the polynomial that interpolates to $(0, v_2, v_3)$ at $(1, \beta\beta\sigma)$.

We now argue **Simulate**'s output is indistinguishable from an honest prover's output.

We consider each of the elements in Fig. 4 separately and argue they are identically distributed with overwhelming probability.

- $[C_I]_1$ is blinded by r_2 for the prover and s_1 for the simulator.
- $[z_I]_1$ is blinded by r_1 for the prover and s_2 for the simulator.
- $[u]_1$ is blinded by r_5 for the prover and s_3 for the simulator.
- $[H_1]_2$ is the unique element satisfied by the pairing check for both the prover and simulator given $[C_I]_1$ and $[z_I]_1$.
- $[H_2]_1$ is blinded by r_3 for the prover and s_4 for the simulator. Note that $r_3 \frac{xu(x)z_I(u(x))}{z_{V_m}(x)}$ is non-zero with overwhelming probability.
- v_1 is blinded by r_6 for the prover and s_5 for the simulator. Note that $r_6 \alpha z_{V_m}(\alpha)$ is non-zero with overwhelming probability.
- v_2 is blinded by r_4 for the prover and s_6 for the simulator. Note that $r_4 u^2 \alpha z_I(u(\alpha))$ is non-zero with overwhelming probability.
- π_1, π_2, π_3 are the unique elements satisfied by the KZG opening checks for both the prover and the simulator.

Finally we consider each of the elements in Fig. 5 separately and argue they are identically distributed with overwhelming probability.

- $[\bar{U}]_1$ is blinded by t_1 for the prover and s_7 for the simulator.
- $[h_2]_1$ is blinded by t_2 for the prover and s_8 for the simulator. Note that there exists a $\rho_2(x)t_2$ term in the provers $[h_2]_1$ which is linearly independent from all other terms and thus not cancelled with overwhelming probability.
- $[h_1]_1$ is blinded by t_3 for the prover and s_9 for the simulator. Note that there is a $t_3^2 z_{V_m}^2(\alpha) \frac{\rho_4^2(x) - \rho_4(x)}{z_{V_n}(x)}$ term in the provers $[h_1]_1$ which is linearly independent from all other terms.
- $[\bar{U}_\alpha]_1$ is blinded by t_4 for the prover and s_{10} for the simulator. Note that there is a $t_4 z_{V_m}(\alpha) \rho_5(x)$ term in the provers $[\bar{U}_\alpha]_1$ which is linearly independent from all other terms.
- $[h_{2,\alpha}]_1$ is blinded by t_5 for the prover and s_{11} for the simulator. Note that there is a $\rho_2(x)t_2$ term in the provers $[h_{2,\alpha}]_1$ which is linearly independent from all other terms.
- v_1 is blinded by r_7 for the prover and s_{12} for the simulator.
- v_2 is blinded by t_5 for the prover and s_{13} for the simulator. Note that there is a $t_5 z_{V_m}(\alpha) \rho_6(\beta)$ term in the provers v_2 which is linearly independent from all other terms.
- v_3 is blinded by t_6 for the prover and s_{14} for the simulator. Note that there is a $t_6 z_{V_m}(\alpha) \rho_7(\beta)$ term in the provers v_3 which is linearly independent from all other terms.
- $\pi_1, \pi_2, \pi_3, \pi_4, \pi_5$ are the unique elements satisfied by the KZG opening checks for both the prover and the simulator.

□

Caulk+: Table-independent lookup arguments

Jim Posen, Assimakis A. Kattis,
New York University



Caulk+: Table-independent lookup arguments

Jim Posen¹ and Assimakis A. Kattis²

¹ Ulvetanna `jimpo` AT `ulvetanna.io`

² New York University `kattis` AT `cs.nyu.edu`

Abstract. The recent work of `Caulk` [ZBK⁺22] introduces the security notion of *position hiding linkability* for vector commitment schemes, providing a zero-knowledge argument that a committed vector’s elements comprise a subset of some other committed vector. The protocol has very low cost to the prover in the case where the size m of the subset vector is much smaller than the size n of the one containing it. The asymptotic prover complexity is $O(m^2 + m \log n)$, where the $\log n$ dependence comes from a subprotocol showing that the roots of a blinded polynomial are all n th roots of unity. In this work, we show how to simplify this argument, replacing the subprotocol with a polynomial divisibility check and thereby reducing the asymptotic prover complexity to $O(m^2)$, removing any dependence on n .

Keywords: polynomial commitments · vector commitments · zero-knowledge

1 Introduction

The work in [ZBK⁺22], named `Caulk`, introduces the notion of *position-hiding linkability* for vector commitment schemes. Two efficient schemes for linking to a committed n -length vector are provided: one for arguing membership of a single committed element and one for arguing multimembership of an m -length subvector. The two protocols use the polynomial commitment scheme of [KZG10], or KZG commitment, over appropriately sized subgroups as the vector commitment scheme of choice. In particular, an n -length vector commitment can be constructed as a commitment to the polynomial interpolation of the vector elements over an order- n multiplicative subgroup of the field.

The asymptotic prover efficiency for the single-element and m -element subvector membership arguments are $O(\log n)$ and $O(m \log n + m^2)$ respectively. `Caulk` achieves sublinear proving times by precomputing vector commitment opening witnesses that take $O(n)$ time to compute naively. Since the evaluation set over which the vector elements lie is a multiplicative subgroup, there is an efficient method to aggregate the precomputed witnesses into a batched witness [TAB⁺20]. These elements are blinded with randomly sampled elements during the proving phase to provide the position-hiding property.

1.1 Our Contribution

We present an improvement to the position-hiding linkability arguments of `Caulk` which reduces the prover complexity to $O(m^2)$ for an m -element membership proof, removing any dependence on the value of n . The $\log n$ asymptotic factor in `Caulk` comes from a subprotocol for the claim that certain blinded evaluation points of the committed polynomial are n -th roots of unity. Our optimization stems from replacing this with a pairing check constraining the evaluation points to be roots of a polynomial dividing $X^n - 1$. While this modification requires the prover to precompute and store one extra witness element per vector index in addition to the one already required in the original scheme, it enjoys improved concrete efficiency and a simpler implementation.

There are two challenges to overcome in constructing this protocol. The prover generates a randomized polynomial of the form $Z(X) = r \prod_{i \in I} (X - \omega^i)$ with a multiplicative blinding factor r , where ω is a primitive n -th root of unity and I is a non-empty subset of $[n]$. A divisibility check that $Z(X)H(X) = X^n - 1$ for some quotient polynomial H guarantees the claim, except for the condition that Z has at least one root. Since Z has only a single degree of blinding, we must be careful when showing that Z has a root so as not to leak any information about that root. The other challenge is that computing the quotient polynomial H can take $O(n)$ time. To circumvent this, we leverage even further the ability to precompute and store witness elements for pairing checks, which is already a core part of the original proof system.

Scheme	Proof size	Prover work	Verifier work
Caulk, lookup table	$14\mathbb{G}_1, 1\mathbb{G}_2, 4\mathbb{F}$	$O_\lambda(m^2 + m \log n)$	$O_\lambda(\log m), 4P$
Caulk, Pedersen link	$6\mathbb{G}_1, 2\mathbb{G}_2, 4\mathbb{F}$	$O_\lambda(\log n)$	$O_\lambda(1), 4P$
This work, lookup table	$7\mathbb{G}_1, 1\mathbb{G}_2, 2\mathbb{F}$	$O_\lambda(m^2)$	$O_\lambda(\log m), 3P$
This work, Pedersen link	$10\mathbb{G}_1, 1\mathbb{G}_2, 5\mathbb{F}$	$O_\lambda(1)$	$O_\lambda(1), 3P$

Figure 1: Comparison of this work with prior work

2 Preliminaries

2.1 Notation

Let $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ be cyclic groups of prime order p , written with additive notation. The finite field \mathbb{F}_p with p elements will sometimes be abbreviated as \mathbb{F} . Let $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ be a pairing: an efficiently computable, non-degenerate bilinear map. Let there be generators $[1]_1, [1]_2, [1]_T$ of $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$, respectively, with $[1]_T = e([1]_1, [1]_2)$. For all elements $\alpha \in \mathbb{F}$ and $\gamma \in \{1, 2, T\}$, the notation $[\alpha]_\gamma$ represents the element $\alpha[1]_\gamma \in \mathbb{G}_\gamma$. The set of polynomials over \mathbb{F} of degree at most d is written as $\mathbb{F}_{\leq d}[X]$. For any set $S \subset \mathbb{F}$, $Z_S(X) := \prod_{v \in S} (X - v) \in \mathbb{F}[X]$ is the monic polynomial of degree $|S|$ which vanishes on S . The set of powers of a value can be written $x^S := \{x^i\}_{i \in S}$ for $S \subset \mathbb{Z}$.

2.2 Algebraic Group Model

We analyze security of our protocols in the Algebraic Group Model (AGM) [FKL18]. In the AGM, whenever an adversary outputs a group element $\mathbf{a} \in \mathbb{G}_\gamma$ with $\gamma \in \{1, 2\}$, they also output an *algebraic representation* as a linear combination of the \mathbb{G}_γ elements that the adversary has access to from the public parameters and structured reference string (SRS).

2.3 Real and Ideal Pairing Checks

We borrow the terminology of real and ideal pairing checks from [GWC19]. An SRS has degree q if its elements equal $\text{SRS}_i = [f(x)]_i$ for uniformly sampled $x \in_R \mathbb{F}$ and some $f \in \mathbb{F}_{<q}[X]$, where $i \in [q]$. Let $f_{i,j}$ denote the corresponding polynomial for the j -th element of SRS_i and a, b be the vectors in \mathbb{F}^q whose encodings in $\mathbb{G}_1, \mathbb{G}_2$ are returned by algebraic adversary \mathcal{A} . A *real pairing check* is defined as:

$$(a \cdot T_1) \cdot (T_2 \cdot b) = 0,$$

for some matrices T_1, T_2 over \mathbb{F} . Real pairing checks can be efficiently computed from the encoded elements and pairing function $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$.

If we operate in the AGM, for each output $[a_j]_i$, \mathcal{A} also outputs a vector v for which $a_j = \sum v_\ell f_{i,\ell}(x) = R_{i,j}(x)$ for $R_{i,j}(X) := \sum v_\ell f_{i,\ell}(X)$. For $i \in \{1, 2\}$, let $R_i = (R_{i,j})_j$ be a vector of polynomials over \mathbb{F} . The corresponding ideal check then is given by:

$$(R_1 \cdot T_1) \cdot (T_2 \cdot R_2) \equiv 0.$$

2.4 Cryptographic Assumptions

We use the formulation of the q -DLOG assumption from [GWC19], Definition 2.1.

Definition 1 (q -DLOG assumption, [GWC19] Definition 2.1, verbatim). Fix integer q . The q -DLOG assumption states that given

$$[1]_1, [x]_1, \dots, [x^q]_1, [1]_2, [x]_2, \dots, [x^q]_2$$

for uniformly chosen $x \in \mathbb{F}$, the probability of an efficient \mathcal{A} outputting x is negligible.

We also use Lemma 2.2 from [GWC19], which follows from the q -DLOG assumption in the AGM.

Lemma 1 ([GWC19] Lemma 2.2, verbatim). *Assume the q -DLOG for $(\mathbb{G}_1, \mathbb{G}_2)$. Given an algebraic adversary \mathcal{A} participating in a protocol with a degree q SRS, the probability of any real pairing check passing is larger by at most an additive negligible factor than the probability the corresponding ideal check holds.*

2.5 Vector Commitments

We recall vector commitment schemes with a trusted setup. Here PP denotes the public parameters of the protocol.

Setup(PP) \rightarrow (SRS, x). Given the public parameters, perform the trusted setup, producing structured reference string SRS and trapdoor x .

Commit(PP, SRS, \vec{c} , r) $\rightarrow C$. Given an input vector \vec{c} and randomness r , produce a commitment C .

Prove_{Open}(PP, SRS, C , i , c_i , \vec{c} , r) $\rightarrow \pi_i$. Given a commitment C and claimed opening (i, c_i) along with the committed vector \vec{c} and randomness r , produce a proof π_i .

Verify_{Open}(PP, SRS, C , i , c_i , π_i) $\rightarrow \{0, 1\}$. Given a commitment C and claimed opening (i, c_i) , verify the opening proof π_i .

Prove_{Open} and **Verify**_{Open} can be generalized to public-coin interactive protocols allowing interaction between prover and verifier. We write **Verify**_{Open} ^{\mathcal{P} for the verification algorithm interacting with a prover \mathcal{P} . The following security property is associated with vector commitments:}

Definition 2 (Position Binding). A vector commitment is *position binding* if for all efficient adversaries \mathcal{A} the following probability is negligible:

$$\Pr \left[\begin{array}{l} c_i \neq c'_i \\ \text{Verify}_{\text{Open}}^{\mathcal{A}}(\text{PP}, \text{SRS}, C, i, c_i, \pi) = 1 \\ \text{Verify}_{\text{Open}}^{\mathcal{A}}(\text{PP}, \text{SRS}, C, i, c'_i, \pi') = 1 \end{array} \middle| \begin{array}{l} \text{SRS} \leftarrow \text{Setup}(\text{PP}) \\ (C, i, c_i, \pi, c'_i, \pi') \leftarrow \mathcal{A}(\text{PP}, \text{SRS}) \end{array} \right].$$

2.6 Position-Hiding Linkability

We restate the definition of position-hiding linkable vector commitments as stated in [ZBK⁺22], which extends the definition of a vector commitment scheme. A vector commitment scheme has position-hiding linkability if there is a zero-knowledge argument of knowledge for the following witness relation:

$$\mathcal{R}_{\text{Link}} := \left\{ \left(\begin{array}{c} \text{PP, SRS} ; \\ C, A, n, m ; \\ \vec{c}, r_c, \vec{a}, r_a \end{array} \right) \middle| \begin{array}{l} \text{Commit}(\vec{c}, r_c) = C \\ \text{Commit}(\vec{a}, r_a) = A \\ \forall i \in [m], \exists j \in [n], a_i = c_j \end{array} \right\}.$$

2.7 Zero-Knowledge with Precomputation

We use the standard definition of honest-verifier zero-knowledge for public-coin interactive protocols: informally, that there exists an efficient algorithm **Simulate** which can produce an accepting transcript that is computationally indistinguishable from a real one between a prover and an honest verifier. In our setting, the prover is allowed to precompute advice inputs from the public parameters and SRS to reduce its online execution time when given an instance. We consider a model where the distinguisher cannot discriminate based on timing information between an execution where the prover has precomputed advice and one where they have not, assuming the precomputation is polynomial-time. In practice, if timing information is available the prover will precompute and store advice for all instances it may generate proofs for.

3 Lookup Argument Construction

Our protocol is based on the one in section 7 of [ZBK⁺22]. The commitments \mathbf{c} and \mathbf{a} are to vectors $\vec{c} \in \mathbb{F}^n$ and $\vec{a} \in \mathbb{F}^m$ respectively. Assume both n and m are powers of two¹. Let \mathbb{H} and \mathbb{V} be multiplicative subgroups of \mathbb{F} of size n and m . The vector commitment scheme used is a KZG commitment over a polynomial which evaluates to the committed vector over some multiplicative subgroup of \mathbb{F} . In particular, \mathbf{c} commits to a polynomial $C(X)$ which evaluates to \vec{c} over \mathbb{H} and \mathbf{a} commits to a polynomial $A(X)$ which evaluates to \vec{a} over \mathbb{V} . Let ω be a generator of \mathbb{H} and ν be a generator of \mathbb{V} . The protocol is then a zero-knowledge argument for the following relation $\mathcal{R}_{\text{Link}}^{\text{KZG}}$, which instantiates $\mathcal{R}_{\text{Link}}$ with KZG commitments:

$$\mathcal{R}_{\text{Link}}^{\text{KZG}} := \left\{ \left(\begin{array}{c} \{[x^{k-1}]_1, [x^{k-1}]_2\}_{k \in [d]} ; \\ \mathbf{c}, \mathbf{a}, \mathbb{H}, \mathbb{V}, \omega, \mu ; \\ C(X), A(X), I \subset [n] \end{array} \right) \middle| \begin{array}{l} \mathbf{c} = [C(x)]_1 \\ \mathbf{a} = [A(x)]_1 \\ \forall i \in [m], \exists j \in I, A(\mu^i) = C(\omega^j) \end{array} \right\}. \quad (1)$$

Let $I \subset [n]$ be the set of indices in \vec{c} that \vec{a} takes values from and $u : [m] \rightarrow I$ be a mapping such that $a_i = c_{u(i)}$ for all $i \in [m]$. The protocol begins with the prover computing polynomials $Z_I, C_I, U \in \mathbb{F}_{\leq m}[X]$ so that the following polynomial identities hold over Z_I :

$$C(X) - C_I(X) = 0 \pmod{Z_I}, \quad (2)$$

$$Z_{\mathbb{H}} = 0 \pmod{Z_I}, \quad (3)$$

and the following identities hold over $Z_{\mathbb{V}}$:

¹The vectors can always be padded with duplicate elements up to the nearest power of two length.

$$C_I(U(X)) - A(X) = 0 \pmod{Z_{\mathbb{V}}}, \quad (4)$$

$$Z_I(U(X)) = 0 \pmod{Z_{\mathbb{V}}}. \quad (5)$$

Intuitively, Z_I is a low-degree polynomial which vanishes on ω^I , C_I is a low-degree polynomial which agrees with C on ω^I , and U maps \mathbb{V} to ω^I . Concretely, the prover computes the Lagrange polynomials $\{\tau_i\}_{i \in I} \subset \mathbb{F}_{<|I|}[X]$ over ω^I and the Lagrange polynomials $\{\mu_j\}_{j \in [m]} \subset \mathbb{F}_{<m}[X]$ over \mathbb{V} . They then define:

$$\begin{aligned} Z_I(X) &= \prod_{i \in I} X - \omega^i, \\ C_I(X) &= \sum_{i \in I} c_i \tau_i(X), \\ U(X) &= \sum_{j \in [m]} u(j) \mu_j(X). \end{aligned}$$

Now, we *could* proceed with the standard compilation of polynomial IOPs to regular IOPs. However, equations 2 and 3 involve polynomials with degree up to n , so computing a KZG commitment opening would take $O(n)$ time. We notice that neither equation involves polynomial composition and so we can enforce the constraints with real pairing checks at the point x from the structured reference string SRS. This approach has the benefit that the quotient elements for the pairing check can be computed in $O(m^2)$ time from precomputed values.

Define $W_1, W_2 \in \mathbb{F}_{<n}[X]$ to be such that $C - C_I = Z_I W_1$ and $Z_{\mathbb{H}} = Z_I W_2$. The prover will look up precomputed values $\{[W_1^{(i)}(x)]_2, [W_2^{(i)}(x)]_2\}_{i \in I}$, where $W_1^{(i)}(X) = (C(X) - c_i)/(X - \omega^i)$, $W_2^{(i)}(X) = Z_{\mathbb{H}}/(X - \omega^i)$, and then compute:

$$\begin{aligned} [W_1(x)]_2 &= \sum_{i \in I} \frac{[W_1^{(i)}(x)]_2}{\prod_{j \in I, i \neq j} \omega^i - \omega^j}, \\ [W_2(x)]_2 &= \sum_{i \in I} \frac{[W_2^{(i)}(x)]_2}{\prod_{j \in I, i \neq j} \omega^i - \omega^j}. \end{aligned}$$

After sending these quotient elements for the pairing checks corresponding to equations 2 and 3, the verifier will query equations 4 and 5 at a challenge point α . The prover will provide polynomial commitment opening proofs which can be computed in $O(m \log m)$ time due to the lower degree bound on the polynomials involved.

The final ingredient is to blind Z_I, C_I, U appropriately to preserve zero-knowledge. While C_I and U can be blinded by respectively adding multiples of Z_I and $Z_{\mathbb{V}}$, Z_I can only be blinded by a single multiplicative factor. At first glace, this presents a problem because the prover must present the evaluation of Z_I at a challenge point during the last step of the protocol and there may not be sufficient degrees of randomness to blind both the evaluation and commitment to Z_I itself. Fortunately however, the KZG openings can be batched together using verifier-supplied randomness in such a way that additional blinding of C_I prevents information leakage.

Figure 2: Interactive Protocol for $\mathcal{R}_{\text{Link}}^{\text{KZG}}$

Public inputs:

- Prime order cyclic groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ with bilinear map e and generators $[1]_1, [1]_2$
- Scalar field \mathbb{F}
- Structured reference string $[x]_1, \dots, [x^{d-1}]_1, [x]_2, \dots, [x^{d-1}]_2$

Common inputs:

- Multiplicative subgroup $\mathbb{H} < \mathbb{F}^*$ with order n and generator ω
- Multiplicative subgroup $\mathbb{V} < \mathbb{F}^*$ with order m and generator ν
- KZG commitment \mathbf{c} to $C(X)$ with evaluation points in \mathbb{H}
- KZG commitment \mathbf{a} to $A(X)$ with evaluation points in \mathbb{V}

Witness inputs:

- Set of indices $I \subset [n]$
- Values $\{c_i\}_{i \in I}$
- Polynomials $C(X), A(X)$
- Mapping $u : [m] \rightarrow I$

Precomputed inputs:

- $[W_1^{(i)}(x)]_2$ for all $i \in I$ where $W_1^{(i)}(X) = (C(X) - c_i)/(X - \omega^i)$
- $[W_2^{(i)}(x)]_2$ for all $i \in I$ where $W_2^{(i)}(X) = Z_{\mathbb{H}}(X)/(X - \omega^i)$

Round 1 Prover:

- Randomly sample blinding factors r_1, \dots, r_6
- Compute the Lagrange basis polynomials $\{\tau_i(X)\}_{i \in [m]}$ over $\omega^j_{j \in I}$
- Define $Z'_I(X) = r_1 \prod_{i \in I} (X - \omega^i)$
- Define $C_I(X) = \sum_{i \in I} c_i \tau_i(X)$
- Define blinded $C'_I(X) = C_I(X) + (r_2 + r_3 X + r_4 X^2) Z'_I(X)$
- Define $U(X)$ to be the degree $m-1$ interpolation over \mathbb{V} with $U(\nu_i) = \omega^{u(i)}$, $\forall i \in [m]$
- Define blinded $U'(X) = U(X) + (r_5 + r_6 X) Z_{\mathbb{V}}(X)$
- Publish $\mathbf{z}_I = [Z'_I(x)]_1, \mathbf{c}_I = [C'_I(x)]_1, \mathbf{u} = [U'(x)]_1$

Round 2 Verifier: Send random challenge χ_1, χ_2

Round 2 Prover:

- Compute $[W_1(x) + \chi_2 W_2(x)]_2 = \sum_{i \in I} \frac{[W_1^{(i)}(x)]_2 + \chi_2 [W_2^{(i)}(x)]_2}{\prod_{j \in I, i \neq j} \omega^i - \omega^j}$
- Compute $H(X) = (Z'_I(U'(X)) + \chi_1(C'_I(U'(X)) - A(X))) / Z_{\mathbb{V}}(X)$
- Publish $\mathbf{w} = r_1^{-1} [W_1(x) + \chi_2 W_2(x)]_2 - [r_2 + r_3 x + r_4 x^2]_2, \mathbf{h} = [H(x)]_1$

Round 3 Verifier: Send random challenge α

Round 3 Prover: Output $v_1, v_2, \pi_1, \pi_2, \pi_3$ where

$$\begin{aligned} P_1(X) &\leftarrow Z'_I(X) + \chi_1 C'_I(X) \\ P_2(X) &\leftarrow Z'_I(U'(\alpha)) + \chi_1(C'_I(U'(\alpha)) - A(X)) - Z_V(\alpha)H(X) \\ (v_1, \pi_1) &\leftarrow \text{KZG.Open}(U'(X), \alpha) \\ (v_2, \pi_2) &\leftarrow \text{KZG.Open}(P_1(X), v_1) \\ (0, \pi_3) &\leftarrow \text{KZG.Open}(P_2(X), \alpha) \end{aligned}$$

Verifier: Compute $\mathbf{p}_1 = \mathbf{z}_I + \chi_1 \mathbf{c}_I$ and $\mathbf{p}_2 = [v_2]_1 - \chi_1 \mathbf{a} - Z_V(\alpha) \mathbf{h}$ and verify

$$\begin{aligned} 1 &\leftarrow \text{KZG.Verify}(\mathbf{u}, \alpha, v_1, \pi_1) \\ 1 &\leftarrow \text{KZG.Verify}(\mathbf{p}_1, v_1, v_2, \pi_2) \\ 1 &\leftarrow \text{KZG.Verify}(\mathbf{p}_2, \alpha, 0, \pi_3) \\ e((\mathbf{C} - \mathbf{c}_I) + \chi_2[x^n - 1]_1, [1]_2) &= e(\mathbf{z}_I, \mathbf{w}) \end{aligned}$$

Prover complexity is $O(m^2)$, with the limiting steps being polynomial interpolations of Z_I and C_I in round 1 and the aggregation of the precomputed KZG witnesses to produce \mathbf{w} in round 2. The verifier verifies three KZG commitment openings and one additional pairing check. Notice that we can drop one degree of blinding from $U'(X)$ as compared to Caulk because $U'(X)$ is not opened as a KZG commitment in the subprotocol to show well-formedness. Furthermore, one fewer pairing is required for verification because the degree bound check used in the subprotocol is eliminated. We can use the same standard batching techniques described in section 8 of [ZBK⁺22] to reduce the number of pairing checks from 4 to 3 and the number of \mathbb{G}_1 elements in the proof from 8 to 7.

Theorem 1. *The protocol in Figure 2 is a zero-knowledge argument of knowledge for the relation in equation 1 with verifier complexity $O_\lambda(1)$ and prover complexity $O_\lambda(m^2)$, granted the prover has precomputed KZG witnesses for C and $X^n - 1$ at all indices in I .*

Proof. The proof of knowledge soundness for Theorem 1 is given in Appendix A and the proof of zero-knowledge is given in Appendix B.

Theorem 2. *There exists a vector commitment scheme with position-hiding linkability, verifier complexity $O_\lambda(1)$, and prover complexity $O_\lambda(m^2)$, for m the size of the subset and n the size of the table. The commitment scheme requires a trusted setup and requires the prover to precompute and store a constant number of elements per linked index that take $O(n)$ time to compute each or $O(n \log n)$ time to compute as a batch.*

Proof. The KZG polynomial commitment scheme over evaluation domains which are multiplicative subgroups is a vector commitment scheme per Section 4.6, “KZG as Vector Commitment Scheme”, of [ZBK⁺22]. The protocol in Figure 2 provides position-hiding linkability with the required asymptotic complexity per Theorem 1.

4 Linking to Pedersen Commitments

Section 6 of [ZBK⁺22] presents a specific argument for linking a Pedersen commitment to an element in a committed vector. In this setting, the SRS contains one additional random element $\mathbf{h} \in \mathbb{G}_1$ for which the discrete log relations to all other SRS elements are unknown. Then we can construct a zero-knowledge argument for the witness relation:

$$\mathcal{R}_{\text{PC-Link}} := \left\{ \left(\begin{array}{c} \{[x^{k-1}]_1, [x^{k-1}]_2\}_{k \in [d]}, \mathbf{h} ; \\ \mathbf{c}, \mathbf{v}, \mathbb{H}, \omega ; \\ C(X), j, v, r \end{array} \right) \middle| \begin{array}{l} \mathbf{c} = [C(x)]_1 \\ \mathbf{v} = [v]_1 + r\mathbf{h} \\ v = C(\omega^j) \end{array} \right\}.$$

Care must be taken when modifying the argument from section 6 of [ZBK⁺22] to replace the unity subprotocol with a divisibility check. The divisibility check does not guarantee that Z_I has a root, and if it is a constant polynomial then the main pairing check does not correspond to a blinded KZG opening. In the generalized argument this is not an issue because equation 5 ensures that Z_I has a root.

Instead, we will compose the generalized lookup argument with a generalized Schnorr proof to produce an efficient argument for the single element case. It is well known that the classic Schnorr argument of knowledge of a discrete logarithm can be generalized to more complex group homomorphisms from a scalar field to a prime order group [Sch91]. In this setting we generalize Schnorr's protocol to an argument of knowledge of the shared opening to two Pedersen commitments with different bases in \mathbb{G}_1 .

The prover samples $k \leftarrow \mathbb{F}$ and computes a polynomial $A(X) = v + k(X - 1)$. The commitment to $A(X)$ is $\mathbf{a} = v[1]_1 + k[x - 1]_1$. The prover and verifier run the lookup argument as a subprotocol with \mathbf{a} and $\mathbb{V} = \{1\}$. Finally they engage in a proof of knowledge of v, r, k such that:

$$\begin{aligned} \mathbf{v} &= v[1]_1 + r\mathbf{h}, \\ \mathbf{a} &= v[1]_1 + k[x - 1]_1. \end{aligned}$$

Figure 3: Interactive Protocol for $\mathcal{R}_{\text{PC-Link}}$

Public inputs:

- Prime order cyclic groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ with bilinear map e and generators $[1]_1, [1]_2$
- Scalar field \mathbb{F}
- Structured reference string $[x]_1, \dots, [x^{d-1}]_1, [x]_2, \dots, [x^{d-1}]_2$
- Independent \mathbb{G}_1 generator \mathbf{h}

Common inputs:

- Multiplicative subgroup $\mathbb{H} < \mathbb{F}^*$ with order n and generator ω
- KZG commitment \mathbf{C} to $C(X)$ with evaluation points in \mathbb{H}
- Pedersen commitment \mathbf{v}

Witness inputs:

- Value v , Pedersen commitment randomness r , index i
- Polynomial $C(X)$

Precomputed inputs:

- $[W_1^{(i)}(x)]_2$ where $W_1^{(i)}(X) = (C(X) - c_i)/(X - \omega^i)$
- $[W_2^{(i)}(x)]_2$ where $W_2^{(i)}(X) = Z_{\mathbb{H}}(X)/(X - \omega^i)$

Round 1 Prover:

- Randomly sample blinding factors $k, \hat{v}, \hat{r}, \hat{k} \leftarrow \mathbb{F}$
- Prover outputs $\mathbf{a} = [v]_1 + k[x - 1]_1$
- Prover and verifier engage in Link protocol with $\mathbf{c}, \mathbf{a}, A(X) = v + k(X - 1), \mathbb{V} = \{1\}, I = \{i\}$ (Figure 2)

Round 2 Prover: Output $\tilde{\mathbf{v}} = [\hat{v}]_1 + \hat{r}\mathbf{h}, \tilde{\mathbf{a}} = [\hat{v}]_1 + \hat{k}[x - 1]_1$

Round 3 Verifier: Sample random χ

Round 3 Prover: Output $s_v = \hat{v} + \chi v, s_r = \hat{r} + \chi r, s_k = \hat{k} + \chi k$

Verifier: Verify that

$$\begin{aligned} [s_v]_1 + s_r \mathbf{h} &= \tilde{\mathbf{v}} + \chi \mathbf{v} \\ [s_v]_1 + s_k [x - 1]_1 &= \tilde{\mathbf{a}} + \chi \mathbf{a}. \end{aligned}$$

5 Acknowledgements

We thank Arantxa Zapico for discussions and clarifications on the original Caulk protocol. We thank Oana Ciobotaru for identifying several mistakes in the presentation, including the statement of the verifier complexity. We thank Michal Zajic, Janno Siim, Helger Lipmaa, and Roberto Parisella for identifying mistakes in the protocol specification which violated correctness and zero-knowledge guarantees.

References

- [FKL18] Georg Fuchsbauer, Eike Kiltz, and Julian Loss. The algebraic group model and its applications. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, pages 33–62, Cham, 2018. Springer International Publishing.
- [GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Paper 2019/953, 2019. <https://eprint.iacr.org/2019/953>.
- [KZG10] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In Masayuki Abe, editor, *Advances in Cryptology - ASIACRYPT 2010*, pages 177–194, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
- [Sch91] C. P. Schnorr. Efficient signature generation by smart cards. *J. Cryptol.*, 4(3):161–174, jan 1991.
- [TAB⁺20] Alin Tomescu, Ittai Abraham, Vitalik Buterin, Justin Drake, Dankrad Feist, and Dmitry Khovratovich. Aggregatable subvector commitments for stateless cryptocurrencies. In Clemente Galdi and Vladimir Kolesnikov, editors, *Security and Cryptography for Networks*, pages 45–64, Cham, 2020. Springer International Publishing.

[ZBK⁺22] Arantxa Zapico, Vitalik Buterin, Dmitry Khovratovich, Mary Maller, Anca Nitulescu, and Mark Simkin. Caulk: Lookup arguments in sublinear time. Cryptology ePrint Archive, Paper 2022/621, 2022. <https://eprint.iacr.org/2022/621>.

A Proof of Theorem 1: Knowledge Soundness

The protocol for position-hiding linking has knowledge soundness in the Algebraic Group Model of [FKL18]. Knowledge soundness is defined by a game Game_{KS} involving an algebraic adversary \mathcal{A} and an efficient extractor \mathcal{E} . Given an SRS, the adversary outputs an instance $\mathbf{c}, \mathbf{a}, \omega, \mu$ and produces an interactive argument for the verifier. The extractor then outputs polynomials C, A . The adversary wins if 1) the verifier accepts and 2) it is not the case that $\mathbf{c} = [C(x)]_1 \wedge \mathbf{a} = [A(x)]_1 \wedge \forall i \in [m], \exists j \in [n], A(\mu^i) = C(\omega^j)$. The protocol has knowledge soundness if there exists an \mathcal{E} so that no adversary wins the game with greater than negligible probability over the verifier's randomness.

Because the structured reference string consists of the powers of x up to x^{d-1} lifted to \mathbb{G}_1 and \mathbb{G}_2 , an algebraic representation of a group element in either group can be interpreted as the coefficients of a polynomial in $\mathbb{F}_d[X]$. The adversary outputs $\mathbf{z}_I, \mathbf{c}_I, \mathbf{u}, \mathbf{h} \in \mathbb{G}_1, \mathbf{w} \in \mathbb{G}_2$ along with their representations, and so the extractor learns the corresponding polynomials Z'_I, C'_I, U', W, H . The real pairing check

$$e((\mathbf{C} - \mathbf{c}_I) + \chi_2[x^n - 1]_1, [1]_2) = e(\mathbf{z}_I, \mathbf{w}),$$

corresponds to the ideal pairing check

$$C - C'_I + \chi_2(X^n - 1) = Z'_I W.$$

Consequently by Lemma 1, the above polynomial identity holds except with negligible probability. Therefore, $Z'_I \mid C - C'_I + \chi_2(X^n - 1)$. Since χ_2 is sampled after the prover commits to C'_I, Z'_I , except with probability $\frac{1}{|\mathbb{F}|}$ it must be that $Z'_I \mid C - C'_I$ and $Z'_I \mid X^n - 1$. Let I be the set of roots of Z'_I . Since $Z'_I \mid X^n - 1$, it follows that $I \subset \mathbb{H}$, and since $Z'_I \mid C - C'_I$, it follows that $C(y) = C'_I(y)$ for all $y \in I$.

By the knowledge soundness of the KZG polynomial commitment scheme and the Schwartz-Zippel lemma, the following polynomial identity holds except with negligible probability because the evaluation holds at a random point α :

$$Z'_I(U'(X)) + \chi_1(C'_I(U'(X)) - A(X)) = Z_{\mathbb{V}}(X)H(X).$$

Therefore, $Z_{\mathbb{V}}(X) \mid Z'_I(U'(X)) + \chi_1(C'_I(U'(X)) - A(X))$. Since χ_1 is sampled after the prover commits to Z'_I, C'_I, U' , except with probability $\frac{1}{|\mathbb{F}|}$ it must be that $Z_{\mathbb{V}} \mid Z'_I(U'(X))$ and $Z_{\mathbb{V}} \mid C'_I(U'(X)) - A(X)$. Then, $Z'_I(U'(y)) = 0$ and $C'_I(U'(y)) = A(y)$ for all $y \in \mathbb{V}$. Furthermore, $U'(y) \in I$ for all $y \in \mathbb{V}$, since I is the set of roots of Z'_I by definition. Now, for any $i \in [m]$, $A(\mu^i) = C'_I(U'(\mu^i))$. There exists a $y \in I$ with $U'(\mu^i) = y$ since U' maps \mathbb{V} to I . For all $y \in I$, $A(\mu^i) = C'_I(y) = C(y)$. Let j be such that $y = \omega^j$, which we know to exist because $I \subset \mathbb{H}$ and ω generates \mathbb{H} . Then when the extractor outputs C, A , it holds that for all $\forall i \in [m], \exists j \in [n], A(\mu^i) = C(\omega^j)$, meaning the adversary loses the game.

B Proof of Theorem 1: Zero Knowledge

We describe the $\text{Simulate}_{\text{Link}}$ algorithm that, given an instance \mathbf{c}, \mathbf{a} and the trapdoor value x convinces an interactive verifier to accept. This is similar to the argument in Appendix F of [ZBK⁺22]. The simulator samples $s_1, \dots, s_8 \leftarrow \mathbb{F}$ at random and outputs $\mathbf{z}_I = [s_1]_1, \mathbf{c}_I = \mathbf{c} - [s_2]_1, \mathbf{u} = [s_3]_1$. The simulator then receives χ_1, χ_2 and outputs

$\mathbf{w} = s_1^{-1}[s_2 + \chi_2 Z_{\mathbb{H}}(x)]_2, \mathbf{h} = [s_4]_2$. As in Appendix F, the simulator receives α , outputs $v_1 = s_5, v_2 = s_6$, and computes KZG evaluation proofs:

$$\begin{aligned}\pi_1 &= (x - \alpha)^{-1}(\mathbf{u} - [v_1]_1), \\ \pi_2 &= (x - v_1)^{-1}(\mathbf{z}_I + \chi_1 \mathbf{c}_I - [v_2]_1), \\ \pi_3 &= (x - \alpha)^{-1}([v_2]_1 - \chi_1 \mathbf{a} - Z_{\mathbb{V}}(\alpha) \mathbf{h}).\end{aligned}$$

It can be seen that the simulator's outputs satisfy the pairing check.

$$\begin{aligned}& e((\mathbf{C} - \mathbf{c}_I) + \chi_2[x^n - 1]_1, [1]_2) \\ &= e([s_2]_1 + \chi_2[Z_{\mathbb{H}}(x)]_1, [1]_2) \\ &= e([1]_1, [s_2]_2 + \chi_2[Z_{\mathbb{H}}(x)]_2) \\ &= e([s_1]_1, s_1^{-1}([s_2]_2 + \chi_2[Z_{\mathbb{H}}(x)]_2)) \\ &= e(\mathbf{z}_I, \mathbf{w})\end{aligned}$$

We note the distribution of output elements matches a valid distribution because:

- \mathbf{z}_I is blinded by r_1 for the prover and s_1 for the simulator,
- \mathbf{c}_I is blinded by r_2 for the prover and s_2 for the simulator,
- \mathbf{u} is blinded by r_5 for the prover and s_3 for the simulator,
- \mathbf{w} uniquely satisfies the pairing check,
- \mathbf{h} is blinded by r_3 for the prover and s_4 for the simulator,
- v_1 is blinded by r_6 for the prover and s_5 for the simulator,
- v_2 is blinded by r_4 for the prover and s_6 for the simulator,
- π_1, π_2, π_3 uniquely satisfy the KZG openings.

Flookup

Ariel Gabizon, Dmitry Khovratovich

Ethereum Foundation



Flookup: Fractional decomposition-based lookups in quasi-linear time independent of table size

Ariel Gabizon Dmitry Khovratovich
Zeta Function Technologies Ethereum Foundation

November 21, 2022

Abstract

We present a protocol for checking the values of a committed polynomial $\phi(X) \in \mathbb{F}_{<m}[X]$ over a multiplicative subgroup $\mathbb{H} \subset \mathbb{F}$ of size m are contained in a table $T \in \mathbb{F}^N$. After an $O(N \log^2 N)$ preprocessing step, the prover algorithm runs in *quasi-linear* time $O(m \log^2 m)$. We improve upon the recent breakthrough results Caulk [ZBK⁺22] and Caulk+ [PK22], which were the first to achieve the complexity sublinear in the full table size N with prover time being $O(m^2 + m \log N)$ and $O(m^2)$, respectively. We pose further improving this complexity to $O(m \log m)$ as the next important milestone for efficient zk-SNARK lookups.

1 Introduction

The *lookup problem* is fundamental to the efficiency of modern zk-SNARKs. Somewhat informally, it asks for a protocol to prove the values of a committed polynomial $\phi(X) \in \mathbb{F}_{<m}[X]$ are contained in a table T of size N of predefined legal values. When the table T corresponds to an operation without an efficient low-degree arithmetization in \mathbb{F} , such a protocol produces significant savings in proof construction time for programs containing the operation. Building on previous work of [BCG⁺18], **plookup** [GW20] was the first to explicitly describe a solution to this problem in the polynomial-IOP context. **plookup** described a protocol with prover complexity quasilinear in both m and N . This left the intriguing question of whether the dependence on N could be made *sub-linear* after performing a preprocessing step for the table T . Caulk [ZBK⁺22] answered this question in the affirmative by leveraging bi-linear pairings, achieving a run time of $O(m^2 + m \log N)$. Caulk+ [PK22] improved this to $O(m^2)$ getting rid of the dependence on table size completely.

However, the quadratic dependence on m of these works makes them impractical for a circuit with many lookup gates. We resolve this issue by giving a protocol called `Lookup` that is quasi-linear in m and has no dependence on N after the preprocessing step.

1.1 Usefulness of the result

When is it worth it to use Flookup instead of plookup ? The plookup prover runs in time $O(N \log N)$ and the Flookup prover requires time $O(m \log^2 m)$ with small constants in the $O()$. Hence, Flookup is worth it roughly when the table is larger than the number of lookups by a logarithmic factor; i.e. when $m \ll N / \log N$.

We write \ll instead of $<$ as Flookup entails other complications that make the tradeoff potentially less attractive. Notably, verification requires a pairing with a prover-defined \mathbb{G}_2 point (as do Caulk and Caulk+), which makes recursive aggregation of proofs less smooth. Another inconvenience is that Flookup doesn't have the nice linearity properties of plookup or Caulk, and so reducing a tuple lookup to a single element lookup (cf. Section 4 of [GW20]), is less efficient. Because of these drawbacks, “simple” tables, like $T = \{0, \dots, 2^t - 1\}$ for a range check, may not be the best use case for Flookup . As in such a case we can decompose into limbs and use a much smaller table; more generally, this is the case when T is a product set.

A better use case would be complex “SNARK unfriendly” operations on large ranges. For example, those arising inside SHA-256, like mapping a 32-bit input A into a 32-bit output B , via bitwise XOR of three different shifts of A . Given such a mapping f with 32-bit input and output, we can construct the table T containing all 2^{32} values $A + 2^{32} \cdot f(A)$.

To show witness values (w_1, w_2) satisfy $f(w_1) = w_2$, we check the corresponding combination $w_1 + 2^{32} \cdot w_2 \in T$ using Flookup . We additionally range constrain each w_i to 32 bits. The need for the additional range constraints stems from Flookup not having nice reductions from vector to single-element lookup. Even with them, being able to represent an arbitrary 32-bit operation in one table (and use it inside a circuit of size $\ll 2^{32}$), constitutes a significant simplification and potential efficiency boost over current use of lookups in zk-SNARKs.

1.2 Organization of the paper and recommended reading route

- In Section 2 we go over required preliminaries.
- In Section 3 we define the notion of a *bi-linear polynomial IOP* which enables us to model protocols that use pairings in addition to polynomial commitment schemes. *A reader deterred by the formality of this section might skip it on a first read; and simply keep in mind that the term “a bi-linear check” in the subsequent section translates to a pairing in the compiled protocol.*
- In section 4 we review a method of [PK22] to extract a commitment to the vanishing polynomial of a subtable using pairings. We extend it to work with arbitrary sets and not just subgroups.
- In Section 5 we give a lookup protocol given a commitment to the vanishing polynomial of the table.

- In Section 6 we combine the table extraction and subtable lookup protocols to give our final result.

2 Terminology and Conventions

We assume our field \mathbb{F} is of prime order. We denote by $\mathbb{F}_{<d}[X]$ the set of univariate polynomials over \mathbb{F} of degree smaller than d . We assume all algorithms described receive as an implicit parameter the security parameter λ .

Whenever we use the term “efficient”, we mean an algorithm running in time $\text{poly}(\lambda)$. Furthermore, we assume an “object generator” \mathcal{O} that is run with input λ before all protocols, and returns all fields and groups used. Specifically, in our protocol $\mathcal{O}(\lambda) = (\mathbb{F}, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t, e, g_1, g_2, g_t)$ where

- \mathbb{F} is a prime field of super-polynomial size $r = \lambda^{\omega(1)}$.
- $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t$ are all groups of size r , and e is an efficiently computable non-degenerate pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_t$.
- g_1, g_2 are uniformly chosen generators such that $e(g_1, g_2) = g_t$.

We usually let the λ parameter be implicit, i.e. write \mathbb{F} instead of $\mathbb{F}(\lambda)$. We write \mathbb{G}_1 and \mathbb{G}_2 additively. We use the notations $[x]_1 := x \cdot g_1$ and $[x]_2 := x \cdot g_2$.

We often denote by $[n]$ the integers $\{1, \dots, n\}$. We use the acronym e.w.p for “except with probability”; i.e. e.w.p γ means with probability *at least* $1 - \gamma$.

universal SRS-based public-coin protocols We describe public-coin (meaning the verifier messages are uniformly chosen) interactive protocols between a prover and verifier; when deriving results for non-interactive protocols, we implicitly assume we can get a proof length equal to the total communication of the prover, using the Fiat-Shamir transform/a random oracle. Using this reduction between interactive and non-interactive protocols, we can refer to the “proof length” of an interactive protocol.

We allow our protocols to have access to a structured reference string (SRS) that can be derived in deterministic $\text{poly}(\lambda)$ -time from an “SRS of monomials” of the form $\{[x^i]_1\}_{a \leq i \leq b}, \{[x^i]_2\}_{c \leq i \leq d}$, for uniform $x \in \mathbb{F}$, and some integers a, b, c, d with absolute value bounded by $\text{poly}(\lambda)$. It then follows from Bowe et al. [BGM17] that the required SRS can be derived in a universal and updatable setup requiring only one honest participant; in the sense that an adversary controlling all but one of the participants in the setup does not gain more than a $\text{negl}(\lambda)$ advantage in its probability of producing a proof of any statement.

For notational simplicity, we sometimes use the SRS srs as an implicit parameter in protocols, and do not explicitly write it.

2.1 Analysis in the AGM model

For security analysis we will use the Algebraic Group Model of Fuchsbauer, Kiltz and Loss[FKL18]. In our protocols, by an *algebraic adversary* \mathcal{A} in an SRS-based protocol we mean a $\text{poly}(\lambda)$ -time algorithm which satisfies the following.

- For $i \in \{1, 2\}$, whenever \mathcal{A} outputs an element $A \in \mathbb{G}_i$, it also outputs a vector v over \mathbb{F} such that $A = \langle v, \text{srs}_i \rangle$.

Idealized verifier checks for algebraic adversaries We introduce some terminology to capture the advantage of analysis in the AGM.

First we say our srs has *degree* Q if all elements of srs_i are of the form $[f(x)]_i$ for $f \in \mathbb{F}_{\leq Q}[X]$ and uniform $x \in \mathbb{F}$. In the following discussion let us assume we are executing a protocol with a degree Q SRS, and denote by $f_{i,j}$ the corresponding polynomial for the j 'th element of srs_i .

Denote by a, b the vectors of \mathbb{F} -elements whose encodings in $\mathbb{G}_1, \mathbb{G}_2$ an algebraic adversary \mathcal{A} outputs during a protocol execution; e.g., the j 'th \mathbb{G}_1 element output by \mathcal{A} is $[a_j]_1$.

By a “real pairing check” we mean a check of the form

$$(a \cdot T_1) \cdot (T_2 \cdot b) = 0$$

for some matrices T_1, T_2 over \mathbb{F} . Note that such a check can indeed be done efficiently given the encoded elements and the pairing function $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_t$.

Given such a “real pairing check”, and the adversary \mathcal{A} and protocol execution during which the elements were output, define the corresponding “ideal check” as follows. Since \mathcal{A} is algebraic when he outputs $[a_j]_i$ he also outputs a vector v such that, from linearity, $a_j = \sum v_\ell f_{i,\ell}(x) = R_{i,j}(x)$ for $R_{i,j}(X) := \sum v_\ell f_{i,\ell}(X)$. Denote, for $i \in \{1, 2\}$ the vector of polynomials $R_i = (R_{i,j})_j$. The corresponding ideal check, checks as a polynomial identity whether

$$(R_1 \cdot T_1) \cdot (T_2 \cdot R_2) \equiv 0$$

The following lemma from [GWC19] is inspired by [FKL18]'s analysis of [Gro16]. It tells us that for soundness analysis against algebraic adversaries it suffices to look at ideal checks. Before stating the lemma we define the Q -DLOG assumption similarly to [FKL18].

Definition 2.1. Fix integer Q . The Q -DLOG assumption for $(\mathbb{G}_1, \mathbb{G}_2)$ states that given

$$[1]_1, [x]_1, \dots, [x^Q]_1, [1]_2, [x]_2, \dots, [x^Q]_2$$

for uniformly chosen $x \in \mathbb{F}$, the probability of an efficient \mathcal{A} outputting x is $\text{negl}(\lambda)$.

Lemma 2.2. Assume the Q -DLOG for $(\mathbb{G}_1, \mathbb{G}_2)$. Given an algebraic adversary \mathcal{A} participating in a protocol with a degree Q SRS, the probability of any real pairing check passing is larger by at most an additive $\text{negl}(\lambda)$ factor than the probability the corresponding ideal check holds.

Proof. Let γ be the difference between the satisfiability of the real and ideal check. We describe an adversary \mathcal{A}^* for the Q -DLOG problem that succeeds with probability γ ; this implies $\gamma = \text{negl}(\lambda)$. \mathcal{A}^* receives the challenge

$$[1]_1, [x]_1, \dots, [x^Q]_1, [1]_2, [x]_2, \dots, [x^Q]_2$$

and constructs using group operations the correct SRS for the protocol. Now \mathcal{A}^* runs the protocol with \mathcal{A} , simulating the verifier role. Note that as \mathcal{A}^* receives from \mathcal{A} the vectors of coefficients v , he can compute the polynomials $\{R_{i,j}\}$ and check if we are in the case that the real check passed but ideal check failed. In case we are in this event, \mathcal{A}^* computes

$$R := (R_1 \cdot T_1)(T_2 \cdot R_2).$$

We have that $R \in \mathbb{F}_{<2Q}[X]$ is a non-zero polynomial for which $R(x) = 0$. Thus \mathcal{A}^* can factor R and find x . \square

Knowledge soundness in the Algebraic Group Model We say a protocol \mathcal{P} between a prover \mathbf{P} and verifier \mathbf{V} for a relation \mathcal{R} has *Knowledge Soundness in the Algebraic Group Model* if there exists an efficient E such that the probability of any algebraic adversary \mathcal{A} winning the following game is $\text{negl}(\lambda)$.

1. \mathcal{A} chooses input x and plays the role of \mathbf{P} in \mathcal{P} with input x .
2. E given access to all of \mathcal{A} 's messages during the protocol (including the coefficients of the linear combinations) outputs ω .
3. \mathcal{A} wins if
 - (a) \mathbf{V} outputs `acc` at the end of the protocol, and
 - (b) $(x, \omega) \notin \mathcal{R}$.

2.2 KZG-like Polynomial commitment schemes

We define a polynomial commitment scheme where we force the commitment procedure to be consistent with that of [KZG10]. This will be useful in the next section when we define bi-linear polynomial IOPs.

Definition 2.3. A d -polynomial commitment scheme (d -PCS) over a field \mathbb{F} consists of

- $\text{gen}(d)$ - a randomized algorithm that outputs an SRS srs that contains as a sub-string $[1]_1, [x]_1, \dots, [x^{d-1}]_1$ for uniformly chosen $x \in \mathbb{F}$ and no other \mathbb{G}_1 elements.
- $\text{com}(f, \text{srs})$ - that given a polynomial $f \in \mathbb{F}_{<d}[X]$ returns the commitment cm to f defined as $\text{com}(f) := [f(x)]_1$.
- A public coin protocol open between parties \mathbf{P}_{PC} and \mathbf{V}_{PC} . \mathbf{P}_{PC} is given $f_1, \dots, f_t \in \mathbb{F}_{<d}[X]$. \mathbf{P}_{PC} and \mathbf{V}_{PC} are both given integer $t = \text{poly}(\lambda)$, $\text{cm}_1, \dots, \text{cm}_t$ - the alleged commitments to f_1, \dots, f_t , $z_1, \dots, z_t \in \mathbb{F}$ and $s_1, \dots, s_t \in \mathbb{F}$ - the alleged correct openings $f_1(z_1), \dots, f_t(z_t)$. At the end of the protocol \mathbf{V}_{PC} outputs `acc` or `rej`.

such that

- **Completeness:** Fix integer t , $z_1, \dots, z_t \in \mathbb{F}$, $f_1, \dots, f_t \in \mathbb{F}_{\leq d}[X]$. Suppose that for each $i \in [t]$, $\text{cm}_i = \text{com}(f_i, \text{srs})$. Then if open is run correctly with values $t, \{\text{cm}_i, z_i, s_i = f_i(z_i)\}_{i \in [t]}$, V_{PC} outputs acc with probability one.
- **Binding Knowledge soundness in the algebraic group model:** For any algebraic adversary \mathcal{A} the probability of \mathcal{A} winning the following game is $\text{negl}(\lambda)$ over the randomness of \mathcal{A} and gen .
 1. Given srs , \mathcal{A} outputs $t, \text{cm}_1, \dots, \text{cm}_t$.
 2. Note that as \mathcal{A} is algebraic, in the step above it also outputs polynomials $f_1, \dots, f_t \in \mathbb{F}_{\leq d}[X]$ such that $\text{cm}_i = [f_i(x)]_1$.
 3. \mathcal{A} outputs $z_1, \dots, z_t \in \mathbb{F}$, $s_1, \dots, s_t \in \mathbb{F}$.
 4. \mathcal{A} takes the part of P_{PC} in the protocol open with common inputs $\text{cm}_1, \dots, \text{cm}_t, z_1, \dots, z_t, s_1, \dots, s_t$.
 5. \mathcal{A} wins if
 - V_{PC} outputs acc at the end of the protocol.
 - For some $i \in [t]$, $s_i \neq f_i(z_i)$.

2.3 Other notational conventions

Given a polynomial $f \in \mathbb{F}[X]$ and a subset $I \subset \mathbb{F}$ we define $f|_I$ to be the set $\{f(v)\}_{v \in I}$. Given a set $T \subset \mathbb{F}$ we denote by $Z_T(X) \in \mathbb{F}[X]$ the *vanishing polynomial* of T :

$$Z_T(X) := \prod_{i \in T} (X - i).$$

3 Bi-linear polynomial IOPs

While most recent works on zk-SNARKs have leveraged the power of polynomial commitment schemes [KZG10], [ZBK⁺22] has additionally leveraged the power of pairings to essentially take products of commitments. This enables checking degree two identities between polynomials without needing to compute the polynomials themselves, but only their commitments - which can be much faster when they are a small linear combination of preprocessed polynomials. We formalize a framework to capture protocols using pairings in addition to polynomial openings.

Definition 3.1. Fix positive integer d and field \mathbb{F} . A d -bi-linear polynomial IOP over \mathbb{F} (d -BLIOP) is a multiround protocol between a prover P_{poly} , verifier V_{poly} and trusted party \mathcal{I} that proceeds as follows.

1. The protocol definition includes two sets of preprocessed polynomials $P_1, P_2 \subset \mathbb{F}_{\leq d}[X]$.

2. The messages of P_{poly} are sent to \mathcal{I} and are of the form (f, i) for $f \in \mathbb{F}_{<d}[X]$ and $i \in \{1, 2\}$. If P_{poly} sends a message not of this form, the protocol is aborted. P_{poly} may also send other messages directly to V_{poly} .
3. The messages of V_{poly} to P_{poly} are always random coins.
4. At the end of the protocol,
 - For $i \in \{1, 2\}$, let F_i denote the set of polynomials f that were sent from P_{poly} to \mathcal{I} as part of a message (f, i) . And denote $A_i := F_i \cup P_i$.
 - V_{poly} may ask \mathcal{I}
 - (a) evaluation queries of the form (f, x) for $f \in A_1$ and $x \in \mathbb{F}$. \mathcal{I} responds with the value $f(x)$.
 - (b) bi-linear identity queries of the form $\sum_{j \in [k]} c_j f_j(X) h_j(X) \stackrel{?}{=} 0$, where k is some positive integer, $c_j \in \mathbb{F}, f_j \in A_1, h_j \in A_2$ for each $j \in [k]$. \mathcal{I} responds with true or false according to whether the identity holds.
 - After concluding her queries V_{poly} outputs acc or rej by a deterministic procedure depending only on the query results.

We define bi-linear polynomial iops for relations and languages in the natural way.

Definition 3.2. Given a relation \mathcal{R} , a d-BLIOP for \mathcal{R} is a d-BLIOP with the following additional properties.

1. At the beginning of the protocol, P_{poly} and V_{poly} are both given - in addition to the preprocessed polynomial sets P_1, P_2 - an input x . The description of P_{poly} assumes possession of ω such that $(x, \omega) \in \mathcal{R}$.
2. **Completeness:** If P_{poly} follows the protocol correctly using a witness ω for x , V_{poly} accepts with probability one.
3. **Knowledge Soundness:** There exists an efficient E , that given access to the messages of P_{poly} to \mathcal{I} , and the random coins of V_{poly} outputs ω such that, for any strategy of P_{poly} , the probability of the following event is $\text{negl}(\lambda)$.
 - (a) V_{poly} outputs acc at the end of the protocol, and
 - (b) $(x, \omega) \notin \mathcal{R}$.

Definition 3.3. Given a language \mathcal{L} , a d-BLIOP for \mathcal{L} is a d-BLIOP with the following additional properties.

1. At the beginning of the protocol, P_{poly} and V_{poly} are both given an input x .
2. **Completeness:** If $x \in \mathcal{L}$, and P_{poly} follows the protocol correctly using x , V_{poly} accepts with probability one.
3. **Soundness:** If $x \notin \mathcal{L}$ then any strategy of P_{poly} will result in V_{poly} rejecting e.w.p $\text{negl}(\lambda)$.

3.1 From bi-linear polynomial IOPs to protocols against algebraic adversaries

We wish to “compile” BLIOPs to protocols against algebraic adversaries. We will define a few terms to enable us to track the compilation efficiency in terms of the resultant prover and verifier efficiency.

Note that given a polynomial protocol \mathcal{P} and fixed input (x, ω) for the protocol. We have some distribution over the sets of polynomials A_1, A_2 sent during the protocol.

Thus, we can define , $D_1(\mathcal{P}, x, \omega) := \sum_{f \in F_1} (\deg(f) + 1)$. And $D_2(\mathcal{P}, x, \omega)$ as the number of \mathbb{G}_2 scalar multiplications required to compute $[f]_2$ for all $f \in F_2$. Also, there will be some distribution over the set of evaluation queries (f, z) asked during the protocol.

Define \mathcal{O} to be the set of tuples $([f]_1, z, f(z); f)$ when iterating over all evaluation queries asked by V_{poly} .

Finally, define $E(\mathcal{P}, x, \omega)$ to be the total number of summands in all bi-linear queries asked by V_{poly} during protocol execution.

Lemma 3.4. *Assume the d-DLOG assumption holds for $(\mathbb{G}_1, \mathbb{G}_2)$. Given a d-BLIOP \mathcal{P} over \mathbb{F} and a d-PCS \mathcal{S} we can construct a protocol \mathcal{P}^* for \mathcal{R} with knowledge soundness against algebraic adversaries such that*

1. *Preprocessing time: For $i = 1, 2$, $C_i(\mathcal{P}) \mathbb{G}_i$ scalar multiplications, where $C_i(\mathcal{P})$ is the number of \mathbb{G}_i scalar multiplications required to compute $[f]_i$ for all $f \in P_i$.*
2. *Prover efficiency: The prover \mathbf{P} in \mathcal{P}^* consists of running P_{poly} on the same inputs; $D_i(\mathcal{P}, x, \omega) \mathbb{G}_i$ scalar multiplications for $i \in \{1, 2\}$ and running the prover of \mathcal{S} with input \mathcal{O} .*
3. *Verifier efficiency: The verifier \mathbf{V} in \mathcal{P}^* consists running V_{poly} on the same inputs; $E(\mathcal{P}, x, \omega)$ pairings and \mathbb{G}_t exponentiations, and running the verifier of \mathcal{S} with input \mathcal{O} .*
4. *Proof size: For $i \in \{1, 2\}$, Let $B_i(\mathcal{P})$ be the number of messaged (f, i) sent during a protocol execution by an honest P_{poly} ; and assume this number doesn't depend on (x, ω) . The final proof consists of $B_i(\mathcal{P}) \mathbb{G}_i$ -elements, and a proof of \mathcal{S} with input \mathcal{O} .*

Proof. Let $\mathcal{S} = (\text{gen}, \text{com}, \text{open})$. Let P_1, P_2 be the sets of preprocessd polynomials in the definition of \mathcal{P} . The SRS of \mathcal{P}^* consists of

- $\mathsf{srs} = [1]_1, \dots, [x^{d-1}]_1, [1]_2, \dots, [x^{d-1}]_2,$
- $\{[f(x)]_1\}_{f \in P_1}, \{[h(x)]_2\}_{h \in P_2}$

Given \mathcal{P} we describe \mathcal{P}^* . \mathbf{P} and \mathbf{V} behave identically to P_{poly} and V_{poly} , except in the following two cases.

- Whenever P_{poly} sends a message (f, i) , for $f \in \mathbb{F}_{<d}[X]$ and $i \in \{1, 2\}$ to \mathcal{I} in \mathcal{P} ; \mathbf{P} instead sends $[f]_i$ to \mathbf{V} .

- Instead of making evaluation queries to \mathcal{I} , \mathbf{V} does the following.
 1. For each evaluation query (f, z) made by V_{poly} to \mathcal{I} , \mathbf{V} instead sends the query directly to \mathbf{P} which responds with the alleged value $s = f(z)$. Let \mathcal{O} be the set of tuples $([f]_1, z, s; f)$ obtained by all evaluation queries.
 2. \mathbf{P} and \mathbf{V} engage in the `open` protocol with input \mathcal{O}
 3. If \mathbf{V} outputs `rej` in this execution of `open`, it also outputs `rej` in \mathcal{P}^* .
- When V_{poly} makes a bi-linear query $\sum_{i \in [k]} c_i f_i(X) h_i(X) \stackrel{?}{=} 0$, \mathbf{V} instead checks the pairing equation
$$\prod_{i \in [k]} e([f_i(x)]_1, [h_i(x)]_2)^{c_i} = 1$$
and proceeds as if the query reply was `true` if and only if the pairing equation held.

- Finally, \mathbf{V} outputs `acc` or `rej` according to whether V_{poly} did given the query replies it has obtained.

To prove the claim about knowledge soundness in the AGM we must describe the extractor E for the protocol \mathcal{P}^* . For this purpose, let $E_{\mathcal{P}}$ be the extractor of the protocol \mathcal{P} as guaranteed to exist from Definition 3.2, and $E_{\mathcal{S}}$ be the extractor for the Knowledge Soundness game of \mathcal{S} as in Definition 2.3.

Now assume an algebraic adversary \mathcal{A} is taking the role of \mathbf{P} in \mathcal{P}^* .

1. When \mathcal{A} sends a message $[f]_i$ to \mathbf{V} then E receives the coefficients of f from \mathcal{A} and adds f to a set A_i .
2. At the end of the protocol E sends A_1, A_2 and the random coins of \mathbf{V} to $E_{\mathcal{P}}$, and receives ω in return.
3. E returns ω .

Note that we can think of A_1, A_2 as random variables of the randomness of \mathbf{V}, \mathcal{A} and `gen`.

Now let us define three events (also over the randomness of \mathbf{V}, \mathcal{A} and `gen`):

1. We let A be the event that for some $(\mathsf{cm}, z, s; f) \in \mathcal{O}$ $f(z) \neq s$, and at the same time V_{PC} has output `acc` when `open` was run by \mathbf{P} and \mathbf{V} . By the KS of \mathcal{S} , $\Pr(A) = \text{negl}(\lambda)$.
2. Let B be the event that for one of the bi-linear queries, we had $\sum c_i f_i(X) h_i(X) \not\equiv 0$; but $\prod_{i \in [k]} e([f_i(x)]_1, [h_i(x)]_2)^{c_i} = 1$. The latter is equivalent to $\sum c_i f_i(x) h_i(x) = 0$, where x is the “secret” in `srs`. We show the probability of this event is $\text{negl}(\lambda)$: Note that in the above event we have that x is a root of $P(X) := \sum c_i f_i(X) h_i(X)$. Thus, we can define an algorithm \mathcal{A}' for finding x given `srs` that runs \mathcal{P}^* between \mathcal{A} and \mathbf{V} , and for each bi-linear query of \mathbf{V} attempts to factor the corresponding P , and checks for each of its roots x' if it’s equal to x . The success probability of \mathcal{A}' is at least the probability of the event C , and thus must be $\text{negl}(\lambda)$ as otherwise we would contradict the d -DLOG assumption for $(\mathbb{G}_1, \mathbb{G}_2)$.

3. We think of an adversary $\mathcal{A}_{\mathcal{P}}$ participating in \mathcal{P} , where V_{poly} is using the same randomness as V , and using the polynomials A_1, A_2 as their messages to \mathcal{I} . We define C to be the event that V_{poly} outputs acc but $(x, \omega) \notin \mathcal{R}$. By the KS of \mathcal{P} , $\Pr(C) = \text{negl}(\lambda)$.

Now look at the event D that V outputs acc , but E failed in the sense that $(x, \omega) \notin \mathcal{R}$. This is the event we need to show has probability $\text{negl}(\lambda)$. We claim that $D \subset A \cup B \cup C$. If A, B didn't happen, it means that V and V_{poly} have received exactly the same answers to their queries, and thus will have the same output. In particular, if we are outside of the event $A \cup B$, V will output acc only when V_{poly} does. In other words, $D \setminus (A \cup B) \subset C$. \square

Remark 3.5. *The above also implies a similar transformation for protocols for languages rather than relations: Given a language \mathcal{L} we can define a relation $\mathcal{R} = \{(x, \omega) | x \in \mathcal{L}\}$. A sound protocol for \mathcal{L} will be knowledge sound for \mathcal{R} (e.g. by defining an extractor that always outputs $\omega = 0$), and vice versa.*

3.2 Conventions for describing BLIOPs and PIOPs

1. When a d -BLIOP doesn't include any bi-linear checks, and accordingly A_2 is empty, we call it a d -polynomial IOP or d -PIOP. In this case we abbreviate " P_{poly} sends $(f, 1)$ " to " P_{poly} sends f ".
2. When we say V_{poly} "checks the identity $P(f_1(X), \dots, f_k(X))$ ", for $f_i \in A_1$, we mean that V_{poly} chooses a random $\alpha \in \mathbb{F}$, queries $f_1(\alpha), \dots, f_k(\alpha)$, computes the value $z = P(f_1(\alpha), \dots, f_k(\alpha))$ and outputs rej if $z \neq 0$. Note that when analyzing soundness or knowledge-soundness of a d -BLIOP, we can assume the event that $g(X) := P(f_1(X), \dots, f_k(X))$ is not the zero-polynomial but $g(x) = 0$ didn't happen as it has $\text{negl}(\lambda)$ probability.
3. When we say V_{poly} "checks the identity $P(f_1(X), \dots, f_k(X))$ on H ", for $f_i \in A_1$ and a set $H \subset \mathbb{F}$, we mean that P_{poly} sends the quotient $T(X) := P(f_1(X), \dots, f_k(X))/Z_H(X)$ and that V_{poly} checks the identity $P(f_1(X), \dots, f_k(X)) = Z_H(X)T(X)$.
4. When describing the efficiency of specific PIOPs and BLIOPs in the rest of the paper, we implicitly use the compilation lemma above, and actually describe the efficiency of the resultant protocol against algebraic adversaries. For example, when we state a BLIOP "requires $t \mathbb{G}_1$ -scalar multiplications on input x ", we are implicitly claiming $D_1(\mathcal{P}, x) = t$ and therefore this is the number of scalar multiplications in the resultant protocol against algebraic adversaries.

4 Protocol for subtable extraction

The protocol in the following section is similar to one implicit in Caulk+ [PK22]. Based on the innovation of Caulk, Caulk+ uses fractional decomposition to efficiently "extract"

a vanishing polynomial Z_I of a subset $I \subset T$ from Z_T . In [PK22], the large set T is always a multiplicative subgroup. This is fine for their protocol, as there T represents *indices* of table values, rather than *the table values themselves*.

Our main innovation in this section is an algorithm that computes all subtable commitments of size $|T| - 1$ efficiently - this insures that also when T is an arbitrary set, our preprocessing remains quasilinear rather than quadratic. This allows us later to work with vanishing polynomials representing the actual table values.

Lemma 4.1. *Given $T \subset \mathbb{F}$ of size N and $\{[x^i]_2\}_{i \in \{0, \dots, N-1\}}$ there is an algorithm using $O(N \log^2 N)$ \mathbb{G}_2 -scalar multiplications and \mathbb{F} -operations for computing the set of elements*

$$\mathcal{T} = \left\{ [Z_{T \setminus \{i\}}(x)]_2 \right\}_{i \in T}$$

Proof. Denote by $\overline{Z_{T \setminus \{i\}}}$ the vector $[a_0 \ a_1 \ \dots \ a_{N-1}]$ (N columns, 1 row) of coefficients of the polynomial $Z_{T \setminus \{i\}}$. Then consider a matrix whose rows are coefficients of $Z_{T \setminus \{i\}}(X)$:

$$Z_{T \setminus *} = \begin{bmatrix} \overline{Z_{T \setminus \{0\}}} \\ \overline{Z_{T \setminus \{1\}}} \\ \vdots \\ \overline{Z_{T \setminus \{N-1\}}} \end{bmatrix}$$

Thus we have to compute $\mathcal{T} = Z_{T \setminus *} \times SRS$ where

$$SRS = [[1]_2, [x]_2, \dots, [x^{N-2}]_2, [x^{N-1}]_2]^T$$

Before we describe an algorithm to compute \mathcal{T} , we first introduce the ideas behind it:

1. For polynomials $a(X), b(X)$ of degree $N/2$ and $c(X)$ such that $c(X) = a(X)b(X)$ with coefficient vectors $\bar{c}, \bar{a}, \bar{b}$ it holds that

$$\begin{aligned} & [c_0 \ c_1 \ c_2 \ \dots \ c_{N-2} \ c_{N-1} \ c_N] = \\ & \quad \begin{bmatrix} b_0 & b_1 & b_2 & b_3 & \dots & b_{N/2} & \dots & 0 \\ 0 & b_0 & b_1 & b_2 & \dots & b_{N/2-1} & \dots & 0 \\ 0 & 0 & b_0 & b_1 & \dots & b_{N/2-2} & \dots & 0 \\ & & & & \ddots & & & \\ 0 & 0 & 0 & 0 & \dots & b_2 & \dots & 0 \\ 0 & 0 & 0 & 0 & \dots & b_1 & \dots & 0 \\ 0 & 0 & 0 & 0 & \dots & b_0 & \dots & b_{N/2} \end{bmatrix} \\ & = [a_0 \ a_1 \ a_2 \ \dots \ a_{N/2-1} \ a_{N/2}] \cdot \end{aligned}$$

or in matrix form

$$\bar{c} = \bar{a} \times A_b,$$

where A_b is a matrix with $N/2 + 1$ rows and $N + 1$ columns.

2. Let us split T into $T_1 = \{v_1, v_2, \dots, v_{N/2}\}$ and $T_2 = \{v_{N/2+1}, \dots, v_N\}$. Then we have

$$Z_{T \setminus *} = \begin{bmatrix} \overline{Z_{T_1 \setminus \{0\}} \cdot Z_{T_2}} \\ \overline{Z_{T_1 \setminus \{1\}} \cdot Z_{T_2}} \\ \dots \\ \overline{Z_{T_2 \setminus \{N-1\}} \cdot Z_{T_1}} \end{bmatrix} = \begin{bmatrix} \overline{Z_{T_1 \setminus \{0\}}} \times A_{Z_{T_2}} \\ \overline{Z_{T_1 \setminus \{1\}}} \times A_{Z_{T_2}} \\ \dots \\ \overline{Z_{T_2 \setminus \{N-1\}}} \times A_{Z_{T_1}} \end{bmatrix} = \begin{bmatrix} Z_{T_1 \setminus *} \times A_{Z_{T_2}} \\ Z_{T_2 \setminus *} \times A_{Z_{T_1}} \end{bmatrix}$$

Therefore

$$Z_{T \setminus *} \times SRS = \begin{bmatrix} Z_{T_1 \setminus *} \times A_{Z_{T_2}} \times SRS \\ Z_{T_2 \setminus *} \times A_{Z_{T_1}} \times SRS \end{bmatrix}$$

Algorithm to compute $\mathcal{T} = Z_{T \setminus *} \times SRS$

1. Compute coefficients of $Z_T(X)$ and its tree of subproducts in $O(N \log^2 N)$ time (see below).
2. Split T into halves T_1 and T_2 . Retrieve $Z_{T_2}(X)$ and $Z_{T_1}(X)$.
3. Compute vectors $a_2 = A_{Z_{T_2}} \times SRS$ and $a_1 = A_{Z_{T_1}} \times SRS$ as Toeplitz matrix-vector multiplication in $O(N \log N)$ time [GVL13].
4. Apply algorithm recursively (go to step 2) to compute $b_1 = Z_{T_1 \setminus *} \times a_2$ and $b_2 = Z_{T_2 \setminus *} \times a_1$.
5. Output concatenation of b_1 and b_2 .

We have an equation for the complexity $C_1(N)$ of the algorithm:

$$C_1(N) = 2C_1(N/2) + O(N \log N)$$

which gives $C_1(N) = O(N \log^2 N)$.

Algorithm to compute Z_T and subproducts: This algorithm is a direct adaptation of [vzGG] (Alg. 10.3).

1. Split T to T_1 and T_2 .
2. Compute Z_{T_1} and Z_{T_2} by two recursive calls to this algorithm with $T = T_1$ and $T = T_2$.
3. Multiply Z_{T_1} by Z_{T_2} in $O(N \log N)$ time using FFT.

We have

$$C_2(N) = 2C_2(N/2) + O(N \log N)$$

which gives $C_2(N) = O(N \log^2 N)$. This ends the proof. \square

We proceed to describe the subtable extraction protocol.

IsVanishingSubtable_T($g(X)$)

Preprocessed polynomials: Let $P_1 = \{Z_T\}$. For each $i \in T$ insert into P_2 the polynomial $Z_{T \setminus \{i\}}$.

Inputs: $g(X) \in \mathbb{F}_{<d}[X]$.

Protocol:

- \mathbf{P}_{poly} sends $(Z_{T \setminus S}, 2)$ to \mathcal{I} .
- \mathbf{V}_{poly} makes the bi-linear query $g \cdot Z_{T \setminus S} \stackrel{?}{=} Z_T$ and outputs `acc` iff it returns `true`.

Lemma 4.2. *IsVanishingSubtable_T is a d-BLIOP for the language $\mathcal{L} := \{g(X) \in \mathbb{F}_{<d}[X] | g(X) = Z_S(X) \text{ for some } S \subseteq T\}$. On input $g = Z_S$, the prover complexity is $O(m \log^2 m)$ \mathbb{F} -operations and $O(m)$ \mathbb{G}_1 and \mathbb{G}_2 -scalar multiplications, where $m = |S|$. Denoting $|T| = N$, preprocessing takes $O(N \log^2 N)$ \mathbb{G}_2 -scalar multiplications and \mathbb{F} -operations.*

Proof. Correctness and soundness are obvious: The check in Step 4 passes if and only if g divides Z_T which happens if and only if $g = Z_S$ for some $S \subseteq T$.

We turn to analyzing efficiency. Denote $m = |S|$. The coefficients of polynomial Z_S can be computed in time $O(m \log^2 m)$ (Lemma 4.1). Then, as noted in [TAB⁺20, vzGG], it holds that

$$Z_{T \setminus S}(X) = \sum_{i \in S} c_i Z_{T \setminus \{i\}}(X)$$

where coefficients c_i are computed via the derivative $Z'_S(X)$ as $c_i = \frac{1}{Z'_S(i)}$ in $O(m \log^2 m)$ time. Thus we can compute $[Z_{T \setminus S}(x)]_2$ with m \mathbb{G}_2 scalar multiplications from the elements of P_2 .

We move to analyze the cost of preprocessing. We must compute given $\{[x^j]_2\}_{j \in \{0, \dots, N-1\}}$ the set

$$\mathcal{T} = \left\{ [Z_{T \setminus \{i\}}(x)]_2 \right\}_{i \in T}.$$

The complexity of this is $O(N \log^2 N)$ \mathbb{G}_2 -scalar multiplications and \mathbb{F} -operations according to Lemma 4.1. \square

5 A PIOP for lookups when given the table in vanishing form

The previous section gives us a way to extract the vanishing polynomial of the subtable we are interested in. We could use a grand product argument to convert the subtable into evaluation/Lagrange form, and then use a lookup protocol like `plookup` that expects to have the table in this form. Instead, we give a protocol that works directly with the vanishing form of the table. In fact, it is considerably more efficient than `plookup` in group operations: It requires roughly $3m$ \mathbb{G}_1 -scalar multiplications, when both witness and table are of size m , as opposed to `plookup` requiring roughly $5m$ (Lemma 3.2 in [GW20]).

Unnormalized rational Lagrange functions Central to our analysis is the idea of defining “rational Lagrange functions” also for points outside of the relevant set. Roughly speaking, this allows us to check inclusion in the set by checking if we ended up with a polynomial or rational function. Details follow.

Fix a set $T \subset \mathbb{F}$. For $v \in \mathbb{F}$, we denote by Γ_v^T the rational function

$$\Gamma_v^T(X) := \frac{Z_T(X)}{X - v}$$

Note that Γ_v^T is a polynomial exactly when $v \in T$.

The following lemma allows us to reduce lookups to distinguishing between polynomials and rational functions.

Lemma 5.1. *Fix any vectors $v, a \in \mathbb{F}^m$, and any subset $T \subset \mathbb{F}$. Define the rational function $R(X) := \sum_{j \in [m]} a_j \Gamma_{v_j}^T(X)$.*

1. *If for all $j \in [m]$, $v_j \in T$; then $R(X) \in \mathbb{F}[X]$.*
2. *Let $S \subset [m]$ be the set of $j \in [m]$ such that $v_j \notin T$. Assume that $S \neq \emptyset$. Then if $\sum_{j \in S} a_j \neq 0$, $R(X) \notin \mathbb{F}[X]$. In particular, assuming $|\text{char}(\mathbb{F})| > m$, $R(X) \notin \mathbb{F}[X]$ when taking $a_j = 1$ for all $j \in [m]$.*

Proof. The first item in the lemma is obvious - a sum of polynomials is a polynomial. We prove the second. Let S be as in the lemma statement and assume it is non-empty. Our task is essentially to show the rational functions do not “cancel out” and create a polynomial. Let $a \in \mathbb{F}^m$ be such that $\sum_{j \in S} a_j \neq 0$. We can write

$$R(X) = R_1(X) + R_2(X)$$

where $R_1(X) = \sum_{j \in [m] \setminus S} \Gamma_{v_j}^T(X)$ and $R_2(X) := \sum_{j \in S} a_j \Gamma_{v_j}^T(X)$. Since R_1 is a polynomial, R is a polynomial if and only if R_2 is. If $R_2(X) \in \mathbb{F}[X]$, then we have the polynomial identity

$$Z_T(X) \sum_{j \in S} \frac{a_j}{X - v_j} = R_2(X).$$

Multiplying denominators, we get

$$Z_T(X)Q'(X) = R_2(X)Q(X)$$

where $Q'(X) := \sum_{j \in S} a_j \prod_{i \in S \setminus \{j\}} (X - v_i)$ and $Q(X) := \prod_{j \in S} (X - v_j)$. We first rule out the possibility $R_2(X) \equiv 0$. If this was the case, we would have $Q'(X) \equiv 0$. However, the coefficient of $X^{|S|-1}$ in Q' is $\sum_{j \in S} a_j$ which we are assuming is non-zero.

So assume now that $R_2(X) \not\equiv 0$. Since none of the factors of Q divide Z_T , we must have $Q|Q'$. However, we have $\deg(Q) = |S|$ and $\deg(Q') < |S|$; so Q doesn’t divide Q' . Therefore, $R_2 \notin \mathbb{F}[X]$.

In summary, $R \notin \mathbb{F}[X]$ in this case, and the second item in the lemma holds. □

The above lemma suggests the following protocol. Let $\mathbb{H} = \{\mathbf{g}, \mathbf{g}^2, \dots, \mathbf{g}^m = 1\} \subset \mathbb{F}$ be a multiplicative subgroup of size m with generator \mathbf{g} . Given a polynomial $\phi(X) \in \mathbb{F}_{<d}[X]$, a set $T \subset \mathbb{F}$ and \mathbb{H} , define $R_{T,\phi}(X) := \sum_{v \in \mathbb{H}} \Gamma_{\phi(v)}^T(X)$ (\mathbb{H} is an implicit parameter in this definition). We commit to $R_{T,\phi}(X)$ and prove the commitment is correct. This will show $R_{T,\phi}$ is a polynomial and therefore $\phi|_{\mathbb{H}} \subset T$ according to the lemma. To show the commitment is indeed to $R_{T,\phi}(X)$, we open it at random $\beta \in \mathbb{F}$, and compare the value to an independent evaluation of $R_{T,\phi}(\beta)$. To compute this independent evaluation of $R_{T,\phi}(\beta)$ we use a “grand sum argument” suggested by Justin Drake [Dra] similar to [GWC19]’s grand product argument.

Below we denote by $\{L_i(X)\}_{i \in [m]}$ the Lagrange basis of \mathbb{H} . That is, $L_i(X) \in \mathbb{F}_{<m}[X]$, $L_i(\mathbf{g}^i) = 1$ and $L_i(\mathbf{g}^j)$ for $i \neq j \in [m]$. Given subsets $\mathbb{H}, T \subset \mathbb{F}$ we define the following protocol.

IsInVanishing $_{\mathbb{H}, T}(\phi)$

Preprocessed polynomials: Let $P_1 = \{Z_T\}$

Inputs: A polynomial $\phi \in \mathbb{F}_{<d}[X]$

Protocol:

1. P_{poly} sends the polynomial $g(X) := R_{T,\phi}(X)$
2. V_{poly} chooses and sends random $\beta \in \mathbb{F}$ and queries the value $z := g(\beta)$.
3. P_{poly} computes and sends a polynomial $Z(X) \in \mathbb{F}_{<m}[X]$ defined as follows
 - For each $i \in [m]$, $Z(\omega^i) = \sum_{j=1}^i \Gamma_{\phi(\mathbf{g}^i)}^T(\beta)$
4. V_{poly} queries the value $Z_T(\beta)$.
5. V_{poly} checks on \mathbb{H} the identities
 - (a) $L_1(X)(Z(X)(\beta - \phi(X)) - Z_T(\beta)) = 0$.
 - (b) $(X - \mathbf{g}) \left(Z(X) - Z(X/\mathbf{g}) \frac{Z_T(\beta)}{\beta - \phi(X)} \right) = 0$.
 - (c) $L_m(X)(Z(X) - z) = 0$.

Theorem 5.2. $\text{IsInVanishing}_{\mathbb{H}, T}$ is a d -PIOP for the language $\mathcal{L} := \{\phi(X) \in \mathbb{F}_{<d}[X] | \phi|_{\mathbb{H}} \subset T\}$. When $\deg(\phi), |T| = O(m)$, the prover runs in time $O(m \log^2 m)$.

Additionally, after a preprocessing phase depending on T consisting of $O(m \log^2 m)$ \mathbb{G}_1 -scalar multiplications; the prover only requires $O(m \log m)$ \mathbb{F} -operations and $O(m)$ \mathbb{G}_1 -scalar multiplications.

Proof. Assume that $\phi \notin \mathcal{L}$. We show that V_{poly} accepts with $\text{negl}(\lambda)$ probability. Let E be the event that $g(\beta) = R_{T,\phi}(\beta)$. When $\phi(X) \notin \mathcal{L}$, Lemma 5.1 implies that $R_{T,\phi(X)} \notin \mathbb{F}[X]$. As $g \in \mathbb{F}_{<d}[X]$, E has probability $\text{negl}(\lambda)$.

Note that the checks in step 5 passing imply that $R_{T,\phi}(\beta) = g(\beta)$:

- The first check implies $Z(\mathbf{g}) = \Gamma_{\phi(\mathbf{g})}^T(\beta)$.
- The second check implies for $i \in \{2, \dots, m\}$, $Z(\mathbf{g}^i) = Z(\mathbf{g}^{i-1}) + \Gamma_{\phi(\mathbf{g}^i)}^T(\beta)$. Thus, the two first checks together imply $Z(\mathbf{g}^m) = \sum_{i \in [m]} \Gamma_{\phi(\mathbf{g}^i)}^T(\beta) = R_{T,\phi}(\beta)$.
- The third check implies $Z(\mathbf{g}^m) = z = g(\beta)$. Hence together with the first two checks, we have $R_{T,\phi}(\beta) = g(\beta)$.

Thus, V_{poly} outputs `acc` only during a $\text{negl}(\lambda)$ probability event.

Turning to analyze the P_{poly} 's runtime, the heaviest component is computing the coefficients of $R_{T,\phi}$. We show that we can derive $R_{T,\phi}$'s values on T in time $O(m \log^2 m)$. From there, we can interpolate the coefficients in time $O(m \log^2 m)$. To do so, P_{poly} computes in $O(m \log m)$ time¹ the values $\{a_v\}_{v \in T}$ where a_v is defined to be the number of $x \in \mathbb{H}$ with $\phi(x) = v$. We have that

$$R_{T,\phi}(X) = \sum_{v \in T} a_v \Gamma_v^T(X).$$

Let $\{\tau_v(X)\}_{v \in T}$ be the Lagrange base of T . We have for $v \in T$, that $\Gamma_v^T(X) = c_v \tau_v(X)$ for the constant $c_v := \prod_{v \neq i \in T} (v - i)$. Thus, we have that

$$R_{T,\phi}(X) = \sum_{v \in T} a_v c_v \tau_v(X).$$

Equivalently, $R_{T,\phi}(v) = a_v c_v$ for each $v \in T$. Thus, once we obtain the values $\{a_v c_v\}_{v \in T}$, we can interpolate the coefficients of $R_{T,\phi}(X)$ in $O(m \log^2 m)$ time.

For this purpose, similarly to the proof of Lemma 4.2, we note that the constants $\{c_v\}_{v \in T}$ are precisely the evaluations of the derivative $Z'_T(X)$ of $Z_T(X)$ at T . Thus they can be computed in the required time bound.

To prove the “additionally” part of the lemma, note that given T , we can precompute the elements $S_v := [\Gamma_v^T(x)]_1$ for all $v \in T$ using the algorithm of Lemma 4.1 in $O(m \log^2 m)$ operations. Given those values we can compute the KZG-commitment $[R_{T,\phi}(x)]_1$ as $\sum_{v \in T} a_v \cdot S_v$, in $O(m)$ \mathbb{G}_1 -scalar multiplications and $O(m \log m)$ \mathbb{F} -operations. We also compute during preprocessing the values $\{c_v\}_{v \in T}$. Given these values, we can similarly compute the KZG opening proof of $R_{T,\phi}$ at $r \in \mathbb{F} \setminus T$ to value $z := R_{T,\phi}(r)$ as

$$\left[\frac{R_{T,\phi}(x) - z}{x - r} \right]_1 = \sum_{v \in T} \frac{a_v - z/c_v}{v - r} \cdot S_v.$$

□

¹In applications, this will typically be $O(m)$ time as ϕ is often given in evaluation form over \mathbb{H} .

6 Putting it all together

$\text{IsInVanishingTable}_{\mathbb{H}, \mathsf{T}}(\phi)$

Preprocessed polynomials: $P_1 = \{Z_T\}$ where

Input: $\phi \in \mathbb{F}_{<d}[X]$

Protocol:

1. $\mathsf{P}_{\mathsf{poly}}$ computes the set $I \subseteq T$ such that $I = \phi|_{\mathbb{H}}$.
2. $\mathsf{P}_{\mathsf{poly}}$ computes and sends Z_I .
3. $\mathsf{P}_{\mathsf{poly}}$ and $\mathsf{V}_{\mathsf{poly}}$ run $\text{IsVanishingSubtable}_{\mathsf{T}}(Z_I)$
4. $\mathsf{P}_{\mathsf{poly}}$ and $\mathsf{V}_{\mathsf{poly}}$ run $\text{IsInVanishing}_{\mathbb{H}, Z_I}(\phi)$.

Combining Lemma 4.2 and Theorem 5.2 we have

Theorem 6.1. *Let $N = |T|$. $\text{IsInVanishingTable}_{\mathbb{H}, \mathsf{T}}$ is a d-BLIOP for the language $\{\phi \in \mathbb{F}_{<d}[X] \mid \phi|_{\mathbb{H}} \subset T\}$ such that*

- $O(N \log^2 N)$ \mathbb{G}_2 -scalar multiplications and \mathbb{F} -operations are required in preprocessing.
- The prover requires $O(m \log^2 m)$ \mathbb{F} -operations and $O(m)$ \mathbb{G}_1 and \mathbb{G}_2 -scalar multiplications.

Proof. The only thing left to address given previous sections is that the computation of Z_I in the second step can be done in time $O(m \log^2 m)$. This, again, follows from algorithm 10.3 in [vzGG]. \square

Acknowledgements

The first author thanks Aztec Network for support of this work. We thank Mary Maller and Arantxa Zapico for helpful discussions. The construction in Section 5 is inspired by a construction of Carla Ràfols and Arantxa Zapico for a similar problem. We thank Yuncong Zhang for a correction.

References

- [BCG⁺18] J. Bootle, A. Cerulli, J. Groth, S. K. Jakobsen, and M. Maller. Arya: Nearly linear-time zero-knowledge proofs for correct program execution. In Thomas

- Peyrin and Steven D. Galbraith, editors, *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part I*, volume 11272 of *Lecture Notes in Computer Science*, pages 595–626. Springer, 2018.
- [BGM17] S. Bowe, A. Gabizon, and I. Miers. Scalable multi-party computation for zk-snark parameters in the random beacon model. Cryptology ePrint Archive, Report 2017/1050, 2017. <https://eprint.iacr.org/2017/1050>.
 - [Dra] J. Drake. <https://youtu.be/tbnaud5wgxm?t=2251>.
 - [FKL18] G. Fuchsbauer, E. Kiltz, and J. Loss. The algebraic group model and its applications. In *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part II*, pages 33–62, 2018.
 - [Gro16] J. Groth. On the size of pairing-based non-interactive arguments. In *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II*, pages 305–326, 2016.
 - [GVL13] Gene H Golub and Charles F Van Loan. *Matrix computations*. JHU press, 2013.
 - [GW20] A. Gabizon and Z. J. Williamson. plookup: A simplified polynomial protocol for lookup tables. *IACR Cryptol. ePrint Arch.*, page 315, 2020.
 - [GWC19] A. Gabizon, Z. J. Williamson, and O. Ciobotaru. PLONK: permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *IACR Cryptology ePrint Archive*, 2019:953, 2019.
 - [KZG10] A. Kate, G. M. Zaverucha, and I. Goldberg. Constant-size commitments to polynomials and their applications. pages 177–194, 2010.
 - [PK22] J. Posen and A. A. Kattis. Caulk+: Table-independent lookup arguments. 2022.
 - [TAB⁺20] A. Tomescu, I. Abraham, V. Buterin, J. Drake, D. Feist, and D. Khovratovich. Aggregatable subvector commitments for stateless cryptocurrencies. In Clemente Galdi and Vladimir Kolesnikov, editors, *Security and Cryptography for Networks - 12th International Conference, SCN 2020, Amalfi, Italy, September 14-16, 2020, Proceedings*, volume 12238 of *Lecture Notes in Computer Science*, pages 45–64. Springer, 2020.
 - [vzGG] J. von zur Gathen and J. Gerhard. Fast polynomial evaluation and interpolation. *Modern Computer Algebra, chapter 10*, pages 295–310.

- [ZBK⁺22] A. Zapico, V. Buterin, D. Khovratovich, M. Maller, A. Nitulescu, and M. Simkin. Caulk: Lookup arguments in sublinear time. *IACR Cryptol. ePrint Arch.*, page 621, 2022.

Baloo: Nearly Optimal Lookups

Arantxa Zapico, Ariel Gabizon, Dmitry Khovratovich, Mary
Maller, Carla Ràfols

Ethereum Foundation
Pompeu Fabra University
Zeta Function Technologies



Baloo: Nearly Optimal Lookup Arguments

Arantxa Zapico^{*}, Ariel Gabizon³, Dmitry Khovratovich¹, Mary Maller¹, and Carla Ràfols²

¹ Ethereum Foundation

² Universitat Pompeu Fabra

³ Zeta Function Technologies

arantxa.zapico@upf.edu, ariel.gabizon@gmail.com, khovratovich@gmail.com, mary.maller@ethereum.org,
carla.rafols@upf.edu

Abstract. We present *Baloo*, the first protocol for lookup tables where the prover work is linear on the amount of lookups and independent of the size of the table. *Baloo* is built over the lookup arguments of Caulk and Caulk+, and the framework for linear relations of Ràfols and Zapico.

Our protocol supports *commit-and-prove expansions*: the prover selects the suitable containing the elements used in the lookup, that is unknown to the verifier, commits to it and later prove relation with the committed element. This feature makes *Baloo* especially suitable for prover input-output relations on hash functions, and in particular to instantiate the Ethereum Virtual Machine (EVM).

1 Introduction

The rise of succinct proving systems in the recent decade has brought us close to one of the Holy Grails of computer science. Namely, being able to prove a large computation while spending not much more time on the proof than on the computation itself. We know how to make a proof only a handful of bytes large, and how to make the verifier run in a millisecond – but the prover time remains a bottleneck. Even though it is asymptotically a linear function of the computation time, the constants in these asymptotics are far too big to prove even moderate-sized programs.

Quite recently, some great hurdles have been overcome for reducing the prover time. First, we learned how to prove not only finite field statements but also lookups in tables such as caches or databases with Plookup [GW20]. These are crucial to implement regular (2^n) integer arithmetic and bit-oriented algorithms such as modern hash functions. The new technique, however, lower bounds the prover time to the table size N and thus limits the usage of big tables. And here came the second breakthrough, Caulk [ZBK⁺22,PK22], as the first method to prove m lookups in more reasonable $O(m^2)$ time, i.e. independent of N . Where Caulk is suitable for big tables but inconvenient for big lookups, Caulk has been calling for the last and final improvement, where one could finally prove m lookups in linear time.

At the same time, a number of computationally powerful blockchains, with Ethereum being the most prominent example, barely withstand the demand for higher transaction rate and computational bandwidth. One bold attempt is to get consensus on the computation without every node repeating work is to use a SNARK as a certificate of correctness. However, efforts to build a prover for Ethereum’s virtual machine [BaCCL21,Eth22,Pol22,Sta22,Zha22,zks22] have been hindered by the cost of proving the Keccak hash function, even if a prover is lookup-enhanced. As Keccak is used in Merkle tree of the blockchain state, a proof for all state transitions in one block results in tens of millions of lookups — the amount insurmountable even for Caulk.

In this work we present *Baloo*, a protocol that finally achieves the goal of proving m lookups in (almost) linear time. The prover is quasilinear in the field and linear in the group. This is thanks to a number of new techniques designed around proving statements over sets that are not multiplicative subgroups (where we cannot use Fast Fourier Transforms). The *Baloo* protocol works smoothly with several multicolumn tables and *Baloo* can be used as a drop-in replacement to the Halo2 lookup argument with much better prover

^{*} This work was done while Arantxa Zapico was a PhD student at Universitat Pompeu Fabra, funded by Protocol Labs PhD Fellowship PL-RGP1-2021-062.

efficiency. In other words *Baloo* is backwards compatible with instantiations of the Halo2 SNARK that use KZG commitments. This means that *Baloo* is potentially the difference between a zkEVM protocol being viable or not.

2 Related Work

In Table 1 we compare the concrete costs of the closest schemes to this work, when compiled using the KZG polynomial commitment scheme [KZG10] (the schemes [GW20], [BGH20], [GK22] are described in the IOP setting only). Plookup [GW20] and Halo2 [BGH20] require no preprocessing but the prover work in the group is quasilinear on the size of the table. They can be compiled using any polynomial commitment scheme including solutions that do not require pairings. Caulk [ZBK⁺22] introduced the first solution with prover work that is sublinear in the size of the table by using preprocessing , but they incur a quadratic cost in the number of lookups. Posen and Kattis introduced Caulk+[PK22] , an improvement over Caulk that leads to a table-independent prover, still quadratic on number of lookups. Gabizon and Khovratovich [GK22] have recently reduced the prover complexity to quasi-linear on the lookups while retaining a table-independent prover. However, their techniques rely on committing to a table as roots of a polynomial instead of coefficients. This means their commitments are not homomorphic, which limits the applicability of their solution to stand alone set membership proofs and makes it challenging to use their lookup to speed up SNARK provers (see Section 8). *Baloo* also has prover complexity that is quasi-linear on the lookups in the field and linear on the group while retaining a table-independent prover, and the commitments are homomorphic. Currently there does not exist a pairing free lookup argument with quasilinear prover work in the size of the table.

Scheme	Preprocessing	Proof size	Prover work		Verifier work
			group	field	
Plookup [GW20]	–	5G ₁ , 9F	O(N)	O(N log N)	2P
Halo2 [BGH20]	–	6G ₁ , 5F	O(N)	O(N log N)	2P
Caulk [ZBK ⁺ 22]	O(N log N)	14G ₁ , 1G ₂ , 4F	15m	O(m ² + m log(N))	4P
Caulk+ [PK22]	O(N log N)	7G ₁ , 1G ₂ , 2F	8m	O(m ²)	3P
Flookup [GK22]	O(N log ² N)	7G ₁ , 1G ₂ , 4F	O(m)	O(m log ² m)	3P
This work: Baloo	O(N log N)	12G ₁ , 1G ₂ , 4F	14m	O(m log ² m)	5P

Table 1. Cost comparison of our scheme with other pairing-based lookups. N is the size of the table and m the size of the set to be opened. The preprocessing costs are given in the number of group operations.

Other approaches, such as discrete-log based [BG13][GK15][BCC⁺15][BG18] require no trusted setup but incur a linear verifier. Bootle et al. [BCG⁺18] initially suggested the use of lookup arguments to improve the prover time in proving machine computations. Their solution was targetted the TinyRAM virtual machine [BCG⁺13]. Campanelli et al. [CFH⁺21] present also an scheme for position-hiding linkability of RSA accumulators for large prime numbers and Pedersen commitments. Concretely they achieve good efficiency: their proof size is constant and their proving times do not depend on the size of the accumulator. Further, they can support larger lookup tables than *Baloo* because they are not constrained by the size of their setup. However, their scheme crucially relies on groups of hidden order such as a trusted RSA modulus or class groups.

Lookup arguments are often used in the context of key-value lookups in verifiable registries [CDGM19]. Multiple works [TBP⁺19,MKL⁺20,HHK⁺21,TFBT21] explore how to ensure the correctness of the table that is used in verifiable registries. Campanelli et al. [CEO22] demonstrate how homomorphic commitments can be used to build key-value lookups. Their solution is zero-knowledge, does not require a trusted setup

or pairings, and uses techniques similar to Section 8. However their prover runs in linear time. Agrawal and Raghuraman [AR20] build key-value lookups using hidden order groups. Campanelli et al. [CFG⁺20] use lookup arguments to construct *verifiable decentralized storage* and achieve a sublinear prover assuming preprocessing.

Benarroch et al. [BCF⁺21] discuss commit-and-prove set membership proofs which is a useful primitive for constructing modular zero-knowledge proofs.

3 Preliminaries

3.1 Notation

A bilinear group gk is a tuple $gk = (q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, [1]_1, [1]_2)$ where $\mathbb{G}_1, \mathbb{G}_2$ and \mathbb{G}_T are groups of prime order q , the elements $\mathcal{P}_1, \mathcal{P}_2$ are generators of $\mathbb{G}_1, \mathbb{G}_2$ respectively, $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ is an efficiently computable, non-degenerate bilinear map, and there is no efficiently computable isomorphism between \mathbb{G}_1 and \mathbb{G}_2 . Elements in \mathbb{G}_γ , are denoted implicitly as $[a]_\gamma = a\mathcal{P}_\gamma$, where $\gamma \in \{1, 2, T\}$ and $[1]_T = e(\mathcal{P}_1, \mathcal{P}_2)$. With this notation, $e([a]_1, [b]_2) = [ab]_T$.

Let $\lambda \in \mathbb{N}$ denote the security parameter and 1^λ its unary representation. A function $\text{negl} : \mathbb{N} \rightarrow \mathbb{R}^+$ is called *negligible* if for all $c > 0$, there exists k_0 such that $\text{negl}(k) < \frac{1}{k^c}$ for all $k > k_0$. For a non-empty set S , let $x \leftarrow S$ denote sampling an element of S uniformly at random and assigning it to x .

PPT denotes probabilistic polynomial-time, and algorithms are randomized unless explicitly noted otherwise. Let $y \leftarrow A(x; r)$ denote running algorithm A on input x and randomness r and assigning its output to y and $y \leftarrow A(x)$ denotes $y \leftarrow A(x; r)$ for a uniformly random r .

Lagrange Polynomials and Roots of Unity. Along this work, we consider two groups of roots of unity.

We use ω to denote a primitive root of unity such that $\omega^N = 1$, and define $\mathbb{H} = \{\omega, \dots, \omega^N\}$. $\lambda_s(X)$ denotes the s th Lagrange interpolation polynomial, i.e., $\lambda_s(X) = \prod_{i \neq s} \frac{X - \omega^i}{\omega^s - \omega^i}$ and $z_H(X) = \prod_{s=1}^N (X - \omega^s) = X^N - 1$ the vanishing polynomial of \mathbb{H} . For a set of indexes $I \subset [N]$, we consider the subset $\mathbb{H}_I \subset \mathbb{H}$ of size $|I| = k$ such that $\mathbb{H}_I = \{\omega^s\}_{s \in I} = \{\xi_i\}_{i=1}^k$. $\{\tau_i(X)\}_{i=1}^k$ and $z_I(X)$ are the corresponding Lagrange and vanishing polynomials. We also assume ν to be a primitive root of unity of order m , and $\mathbb{V} = \{\nu^1, \dots, \nu^m\}$. For simplicity, we may use also ν_j for ν^j . The associated Lagrange interpolation polynomials will be denoted as $\{\mu_j(X)\}_{j=1}^m$ and the vanishing polynomial as $z_V(X)$.

3.2 Cryptographic Assumptions

The security of our protocols holds in the Algebraic Group Model (AGM) of Fuchsbauer et al. [FKL18], using the *bilinear* version of the q-dlog and q-sfrac assumptions [GG17, BB04]. In the AGM, adversaries are restricted to be *algebraic* algorithms, namely, whenever \mathcal{A} outputs a group element $[y]$ in a cyclic group \mathbb{G} of order p , it also outputs its representation as a linear combination of all previously received group elements. In other words, if $[y] \leftarrow \mathcal{A}([x_1], \dots, [x_m])$, \mathcal{A} must also provide \vec{z} such that $[y] = \sum_{j=1}^m z_j[x_j]$. This definition generalizes naturally in asymmetric bilinear groups with a pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$, where the adversary must construct new elements as a linear combination of elements in the same group.

3.3 The KZG Polynomial Commitment Scheme

Our construction heavily relies on the polynomial commitment introduced by Kate, Zaverucha and Goldberg in [KZG10] that we described below. As noted in Caulk [ZBK⁺22], the protocol can be slightly modified to support degree checks, so it consists on a tuple $(\text{KZG.Setup}, \text{KZG.Commit}, \text{KZG.Open}, \text{KZG.Verify})$ such that:

- $\text{srs}_{\text{KZG}} \leftarrow \text{KZG.Setup}(\text{par}_{\text{KZG}}, d)$: On input the system parameters and a degree bound d , it outputs a structured reference string $\text{srs}_{\text{KZG}} = (\{[x^i]_{1,2}\}_{i=1}^d)$.
- $C \leftarrow \text{KZG.Commit}(\text{srs}_{\text{KZG}}, p(X))$: On input polynomial $p(X)$, it outputs $C = [p(x)]_1$.

- $(s, \pi_{\text{KZG}}) \leftarrow \text{KZG.Open}(\text{srs}_{\text{KZG}}, p(X), \alpha)$: Let $\deg < d$ be the degree of $p(X)$. Given $\alpha \in \mathbb{F}$, prover computes

$$q(X) = \frac{p(X) - p(\alpha)}{X - \alpha},$$

- sets $s = p(\alpha)$, $[Q]_1 = [q(x)x^{d-\deg+1}]_1$, and outputs $(s, \pi_{\text{KZG}} = [Q]_1)$.
- $1/0 \leftarrow \text{KZG.Verify}(\text{srs}_{\text{KZG}}, C, \deg, \alpha, s, \pi_{\text{KZG}})$: Verifier accepts if and only if

$$e(C - s, [x^{d-\deg+1}]_2) = e([Q]_1, [x - \alpha]_2).$$

Multiple Openings. We also implement the optimization noted in [ZBK⁺22] to open one polynomial to many distinct points. In a nutshell, given the polynomial $p(X)$, a vector of opening points $\vec{\alpha} \in \mathbb{F}^t$ and \vec{s} such that $s_i = p(\alpha_i)$ for all $i = 1, \dots, t$, prover and verifier define $C_{\vec{\alpha}}(X)$ as the unique polynomial of degree $t - 1$ such that $C_{\vec{\alpha}}(\alpha_i) = s_i$ for all $i \in [t]$. Then, $p(\alpha_i) = s_i$ for all $i = 1, \dots, y$ if and only if there exists $q(X)$ such that

$$p(X) - C_{\vec{\alpha}}(X) = \prod_{i=1}^m (X - \alpha_i) q(X).$$

Subset openings. It is crucial for our construction the subvector opening scheme of Tomescu et. al [TAB⁺20] that works for the vector commitment inspired by KZG.

Given an encoding $C(X) = \sum_{s=1}^N c_s \lambda_s(X)$ to a vector $\vec{c} \in \mathbb{F}^N$ and $C_I(X) = \sum_{i=1}^k \hat{c}_i \tau_i(X)$, where $\{\tau_i(X)\}$ are the Lagrange interpolation polynomials of the set $\{\xi_i\}_{i=1}^k = \{\omega^s\}_{s \in I}$, they note that for $z_I(X) = \prod_{s \in I} (X - \omega^s)$,

$$C(X) - C_I(X) = z_I(X) Q_I(X),$$

for some polynomial $Q_I(X)$ if and only if $\hat{c}_i = c_s$ for the unique pair (i, s) such that $\xi_i = \omega^s$.

What is more, it is demonstrated in [TAB⁺20] that the prover can compute $[Q_I]_1$ by performing k group and $\mathcal{O}(k \log^2(k))$ field operations, given they already have stored proofs $\{[Q_s]_1\}_{s \in I}$ that $C(\omega^s) = c_s$. Precomputing all the proofs $\{[Q_s]_1\}_{s=1}^N$ can be done in time $\mathcal{O}(N \log N)$ using techniques by Feist and Khovratovich [FK20]. More details are given in Section 7.

3.4 Caulk+ core

We use a subroutine of the lookup argument Caulk+, which we call **Caulk+ core**, as a building block of *Baloo*. Caulk+, by Posen and Kattis [PK22], is an improvement on Caulk [ZBK⁺22] that takes prover computation in the group from $O(m^2 + m \log(B))$ to just $O(m^2)$ for m lookups on tables of size N . Following their blueprint, our first step is to create a subtable. That is, given a public vector \vec{c} , encoded as polynomial $C(X)$, and elements $\mathbf{t} \in \mathbb{G}_1$ and $k \in \mathbb{N}$, **Caulk+ core** proves that \mathbf{t} is a commitment to a subvector $\vec{t} \in \mathbb{F}^k$ of \vec{c} . In other words, there exist $I \subset [N]$ such that $\vec{t} = (c_s)_{s \in I}$.

Here **Caulk+ core** considers the subvector length k as a public parameter, and proves the following relation

$$\mathbb{R}_{\text{subtable}} = \left\{ (C(X), \mathbf{t}, [z_I]_2); (\mathbb{H}_I, t(X)) \mid \begin{array}{l} \mathbb{H}_I = \{\xi_1, \dots, \xi_k\} \subset \mathbb{H} \\ \forall \xi \in \mathbb{H}_I, t(\xi) = C(\xi) \\ \mathbf{t} = [t(x)]_1 \\ [z_I]_2 = [z_I(x)]_2 \text{ for } z_I(X) = \prod_{i=1}^k (X - \xi_i) \end{array} \right\},$$

where $C(X) = \sum_{s=1}^N c_s \lambda_s(X)$ for $\{\lambda_s(X)\}_{s=1}^N$ the Lagrange interpolation polynomials of a subgroup of roots of unity $\mathbb{H} = \{\omega^s\}_{s=1}^N$.

In Figure 1 we describe the protocol for $\mathbb{R}_{\text{subtable}}$ [PK22]. The prover running time of **Caulk+ core** is quasilinear in k for field and linear in k for group operations (assuming some precomputations), as we see below (more details in Section 7).

Theorem 1. *The protocol in Fig. 1 is knowledge sound in the algebraic group model assuming the **q-dlog** and the **q-sfrac** assumptions hold.*

We refer the reader to Appendix A for the proof.

$$\text{srs} = \{[x^s]_{1,2}\}_{s=1}^N, \mathsf{C} = [C(x)]_1, [z_H(x)]_1 = [\prod_{s=1}^N (x - \omega^s)]_1, [z_I]_2, \mathsf{t}$$

Prover_{C+}: takes as input $\mathbb{H}_I, t(X), \{[Q_i(x)]_1 = [(C(x) - C(\xi_i))/(x - \xi_i)]_1, [H_i(x)]_1 = [z_H(x)/(x - \xi_i)]_1\}_{i=1}^k$

- Set $z_I(X) = \prod_{i=1}^k (X - \xi_i)$ and $z_{H \setminus I} = \prod_{s \in [N] \setminus I} (X - \omega^s)$
- Compute $W_1 = \left[\frac{C(x) - t(x)}{z_I(x)} \right]_1$ and $W_2 = [z_{H \setminus I}(x)]_1$
- Compute $W_3 = [(z_I(x) - x^k)x^{N-k+1}]_1$

It outputs $\pi_{C+} = (W_1, W_2, W_3)$.

Verifier_{C+}: Accepts if and only if

- (i) $e(\mathsf{C} - \mathsf{t}, [1]_2) = e(W_1, [z_I]_2)$
- (ii) $e([z_H(x)]_1, [1]_2) = e(W_2, [z_I]_2)$
- (iii) $e([x^{N-k+1}]_1, [z_I]_2 - [x^k]_2) = e(W_3, [1]_2)$

Fig. 1. The Caulk+ core [PK22] quasilinear-time protocol for proving $\mathbb{R}_{\text{subtable}}$ that a commitment contains a subtable.

3.5 Generalized Sumcheck

Following [BCR⁺19], in Section 5.4 we construct a scheme for inner product relations that rely on the univariate sumcheck argument for the elements in the set \mathbb{H}_I . Since the latter is not enforced to be a group of roots of unity, but just a subset of one, we use the generalized variant of the sumcheck:

Theorem 2 (Generalized Sumcheck [RZ21]). *Let $\mathbb{H}_I = \{\xi_i\}_{i=1}^k$ be an arbitrary subset of size k in some finite field \mathbb{F} and $z_I(X)$ its vanishing polynomial. For any $P(X) \in \mathbb{F}[X]$, $\sum_{i=1}^k P(\xi_i) = \sigma$ if and only if there exist polynomials $Q(X) \in \mathbb{F}[X]$, $R(X) \in \mathbb{F}_{\leq k-2}[X]$ such that*

$$P(X)N_{\mathbb{H}_I}(X) - \sigma = XR(X) + z_I(X)Q(X),$$

where $N_{\mathbb{H}_I}(X) = \sum_{i=1}^k \tau_i(0)^{-1} \tau_i(X)$ and $\tau_i(X)$ is the i th Lagrange polynomial associated to ξ_i and the set \mathbb{H}_I .

4 Overview

In this section we provide a technical overview of Baloo protocol, which proves the following statement:

Given element cm and a public set represented as vector $\vec{c} \in \mathbb{F}^N$, there exists $\vec{a} \in \mathbb{F}^m$ such that all elements of \vec{a} are elements of \vec{c} and cm is a commitment to \vec{a} .

We approach this statement in two steps. First we select a subvector \vec{t} of \vec{c} by trimming all elements not in \vec{a} , and use Caulk+ core protocol [PK22] to prove, in a committed form, its well formation. Second, as in Caulk and Caulk+, we prove, again in a committed form, that \vec{a} is a result of some expansion of \vec{t} , i.e., we design a lookup argument for an unknown table \vec{t} . However, and this is our main contribution, we replace the (implicit) lookup argument in [ZBK⁺22,PK22] by a variant of the linear argument in [RZ21] so that both steps are quasilinear in the lengths k, m of \vec{t} and \vec{a} , assuming some precomputations. As precomputations are the same as in Caulk [ZBK⁺22], we get a prover with quasilinear of m online time and quasilinear of N offline (precomputation) time in the field and just $O(m)$ group operations in the online phase, as announced.

Select a subvector. Following Caulk+ core (Section 3.4), we denote by \mathbf{C} a commitment to \vec{c} . Then we create a commitment \mathbf{t} to a subvector $\vec{t} \in \mathbb{F}^k$ defined by a set of index $I \subset [N]$, that is, $\vec{t} = (c_s)_{s \in I}$. To prove well formation of \mathbf{t} , the prover provides $W_1, [z_I]_2$ such that

$$e(\mathbf{C} - \mathbf{t}, [1]_2) = e(W_1, [z_I]_2),$$

and a proof of well formation of $[z_I]_2$ which convinces the verifier that $[z_I]_2$ is a commitment to the vanishing polynomial of some set $\mathbb{H}_I \subset \mathbb{H}$. Let us denote the roots of unity elements of \mathbb{H}_I by $\{\xi_1, \xi_2, \dots, \xi_k\}$ i.e. $\xi_i = \omega^s$ for some $s \in I$, following the order in \mathbb{H} .

If the prover has access to precomputed proofs $\{[Q_s]_1\}_{s=1}^N$ of opening for all the elements $c_s \in \vec{c}$ and individual proofs $\{[H_s]_1\}_{s=1}^N$ of statements “ $(X - \omega^s)|z_H(X)$ ”, it performs only $2k$ group operations to compute W_1 and thus convince the verifier that \mathbf{t} is a commitment to some subvector of \vec{c} .

Expansion as a linear relation. Now we engage in the second task: given elements \mathbf{t}, \mathbf{cm} that commit to unknown vectors $\vec{t} \in \mathbb{F}^k, \vec{a} \in \mathbb{F}^m$ using the Lagrange basis corresponding to some unknown set \mathbb{H}_I of size k and $\{\mu_j(X)\}_{j=1}^m$ to public subgroup \mathbb{V} , respectively, prove that for every $j \in [m]$ there exists $i \in [k]$ such that $a_j = t_i$. The latter equation has a simple algebraic representation: as \vec{a} is a vector generated using elements of \vec{t} , there exists a matrix $M \in \mathbb{F}^{m \times k}$ of 1s and 0s such that

$$\mathbf{M}\vec{t} = \vec{a}. \tag{1}$$

Concretely, the non-zero elements are $m_{j,i}$ for j, i such that $a_j = t_i$. We then prove that relation (1) also holds in the committed form. Here \mathbf{t} and \mathbf{cm} are commitments to \vec{t} and \vec{a} , whereas we employ a special technique to commit to a matrix.

Proving linear relations is usually done via a lincheck argument, e.g. [BCR⁺19]. Ràfols and Zapico in [RZ21] separate a lincheck argument into two parts. First, prover and verifier engage in a Checkable Subspace Argument (CSS) where prover convinces the verifier that a polynomial $D(X)$ encodes a random vector \vec{d} in the rowspace of \mathbf{M} sampled with the verifiers’ coins. This can also be seen as a partial evaluation problem: if one defines a bivariate polynomial that encodes the matrix

$$M(X, Y) = (\mu_1(X), \dots, \mu_m(X)) \mathbf{M} \begin{pmatrix} \tau_1(X) \\ \vdots \\ \tau_k(X) \end{pmatrix},$$

the goal of a CSS argument is to show that $D(X) = M(\alpha, X) = \sum_{i=1}^k \sum_{j=1}^m M_{i,j} \tau_i(X) \mu_j(\alpha)$. This allows to reduce the statement in eq. (1) to a single inner product relation $\vec{d} \cdot \vec{t} = \sum_{j=1}^m a_j \mu_j(\alpha)$, that can be proven with the univariate sumcheck [BCR⁺19].

We modify the framework for linear relation of Ràfols and Zapico in [RZ21] in several ways:

- **CSS.** We give a new definition of Commit-and-Prove CSS which works for matrices that are chosen in a commit-and-prove fashion [CFQ19], that is, the prover selects matrix \mathbf{M} , communicates it to the verifier in a succinct manner and then convinces them that $\mathbf{M}\vec{t} = \vec{a}$. Importantly, the prover must convince the verifier that the committed matrix \mathbf{M} has a certain form, i.e., that its rows are unit vectors, so the linear relation represents a lookup. Also, the basis to encode the vectors in the rowspace $\vec{r}(X)$ is also communicated succinctly by the prover. This is in contrast to the constructions of CSS schemes in the holographic model considered in [RZ21], where the matrix M was fixed and preprocessed offline and where $\vec{r}(X)$ was fixed.

The checkable subspace sampling (CSS) technique ensures that a commitment $[D]_1$ is to $D(X)$, as defined by verifier’s coins. For this we adapt a construction of CSS given in [RZ21] for so called basic matrices, with only a non-zero element per column. Since a lookup matrix has only a non-zero element per row, it is the transpose of a basic matrix. Since \mathbf{M} is only known by the prover in our new construction, we replace the offline phase usually performed by some untrusted party with a commitment phase performed by the prover itself.

To adapt the CSS in [RZ21] to our case, we observe that if $M(X, Y) = \sum_{i=1}^k \sum_{j=1}^m M_{i,j} \tau_i(X) \mu_j(Y)$ is the encoding of a matrix, $E(X) = M(\beta, X)$ encodes a vector sampled in the column space of \mathbf{M} and a vector sampled in the row space of its transpose, a basic matrix, using same coins β . So we use the argument in [RZ21] for well formation of $E(X)$ and then use the fact that if $D(X) = D(X, \alpha)$ encodes a sampling in the row space using coins α , it must be the case that $E(\alpha) = D(\beta)$.

- **Inner Product.** In Section 5.4 we present a scheme for proving inner product relations between encodings $D(X)$ and $t(X)$ to vector \vec{d} as described above and table \vec{t} , respectively. Both $t(X)$ and $D(X)$ can be naturally written in the Lagrange basis $\{\tau_i(X)\}_{i=1}^k$ corresponding to set \mathbb{H}_I . This is not a subgroup of the field, so use a generalized univariate sumcheck that works in this setting due to [RZ21]. We show that this result allows to prove a “twisted inner product relation” $\sum_{i=1}^k t_i d_i \tau_i(0)$. To cancel out the undesired $\tau_i(0)$ factors, in the CSS argument we will in fact sample set \vec{d} to be a vector in the rowspace of M where coordinate i is divided by $\tau_i(0)^{-1}$.

Summary. Overall, the expansion protocol takes as input $\phi(X), t(X), z_I(X)$ and aims to show that $\phi(\xi) = t(\xi)$ for all $(X - \xi)$ dividing $z_I(X)$. For that, we

1. Prove that an element $[D]_1$ is the commitment to $D(X) = M(X, \alpha) = \sum_{j=1}^m \mu_j(\alpha) \tau_{\text{col}(j)}(X) (\tau_{\text{col}(j)}(0))^{-1}$, where α is a random point sampled by the verifier, and $\text{col}(j)$ is the column of non-zero element in row j i.e. $M_{j,\text{col}(j)} = 1$. Here $\{\mu_j(X)\}_{j=1}^m$ is a set of known Lagrange polynomials and $\{\tau_i(X)\}_{i=1}^k$ is a set of unknown Lagrange polynomials defined by the set of points $\mathbb{H}_I = \{\xi_i\}_{i=1}^k$ such that $(X - \xi_i)$ divides $z_I(X)$. For this step, we create $E(X) = M(\beta, X)$ and prove its well formation, i.e., we prove it is the encoding of a vector sampled in the row space of \mathbf{M}^\top . Then, we prove $E(\alpha) = D(\beta)$ and thus $D(X)$ is a vector sampled in the row space of \mathbf{M} .
2. From $t, [D]_1$, prove that $\vec{d} \cdot \vec{t} = \sum_{j=1}^m a_j \mu_j(\alpha) = \phi(\alpha)$.

We combine this scheme with **Caulk+ core** (Fig. 1) and obtain **Baloo**, a lookup argument proving that all the elements in \vec{a} , committed to in **cm**, are included in \vec{c} .

5 Building Blocks

In this section we introduce argument **cp-expansion**, a polynomial holographic proof (PHP) [CFF⁺21] to show that the vector \vec{a} encoded in a polynomial $a(X)$ is an expansion of the vector \vec{t} encoded in a polynomial $t(X)$, which means that all for all $j \in [m]$ there exists $i \in [k]$ such that $a_j = t_i$. **cp-expansion** uses two core building blocks: (i) a checkable subspace sampling proof to prove that some polynomial $D(X)$ encodes a vector in the row space of a matrix \mathbf{M} defining the expansion, and (ii) a proof that some inner product argument holds between the vectors encoded in $D(X), t(X)$, and $a(X)$. We present these building blocks as PHP and compile them in Section 6.

5.1 Commit-and-Prove Checkable Subspace Sampling

The first step for proving the relation between \vec{t} and \vec{a} through polynomial encodings, is that prover and verifier agree on a polynomial $D(X)$ encoding a random element in the row space of matrix \mathbf{M} such that $\mathbf{M}\vec{t} = \vec{a}$. In the setting of SNARKs, matrix \mathbf{M} is part of the instance and, therefore, known by prover and verifier, while in our case, \mathbf{M} is decided by the prover and the verifier only needs to be convinced that it has the correct form. More concretely, \vec{t} is an *expansion* of \vec{a} if and only if \mathbf{M} is a matrix that has unit vectors in its rows, that is, there exists only one non-zero element in each row of \mathbf{M} and it equals 1. Overall, the prover chooses \mathbf{M} .

For some technical reasons⁴, it will be simpler to define the argument avoiding explicit reference to matrix \mathbf{M} and refer instead only to polynomials. In these terms, what we want to prove is that $D(X)$ is the correct

⁴ The matrix \mathbf{M} needs to depend on the order of $\tau_i(X)$, although the argument does not enforce any order of \mathbb{H}_I . Using polynomials allows us to completely ignore the order of the elements of the set \mathbb{H}_I .

linear combination of the polynomials $\vec{M}(X) = (M_1(X), \dots, M_k(X))^\top$ that are some encoding of the rows \vec{M}_i of \mathbf{M} .

We call this variant of CSS schemes a commit-and-prove CSS, a PHP where instead of having an indexer that in an offline phase computes polynomials describing the matrix, we have that the prover commits to a matrix and then proves attributes of it. In our commit-and-prove CSS, the prover and verifier first engage in a commit and sample phase, during which they jointly agree on the statement being proven. Afterwards the prover and verifier engage in a proving phase where the prover demonstrates that the statement is correct.

Definition 1 (Commit-and-Prove CSS). *A commit-and-prove checkable subspace sampling argument over a field \mathbb{F} , is a PHP that defines a set of allowed matrices \mathcal{M}_I and runs in four different stages:*

- **Commit Phase:** For some set of maximal size k , the prover \mathcal{P}_{CSS} sends a commitment $\text{cm}_{\mathbb{H}_I}$ to a set \mathbb{H}_I and a commitment $\text{cm}_{\vec{M}}$ to a vector $\vec{M}(X)$ in some allowed set $\mathcal{M}_I(X)$.
- **Sampling Phase:** Prover \mathcal{P}_{CSS} and verifier \mathcal{V}_{CSS} engage in an interactive protocol Sampling. In some round, the verifier sends $\text{cns} \leftarrow \mathcal{C}$, and the prover replies with a polynomial $D(X) = \vec{s}^\top \vec{M}(X)$, for $\vec{s} = \text{Smp}(\text{cns})$.
- **Proving Phase:** \mathcal{P}_{CSS} and \mathcal{V}_{CSS} engage in an interactive protocol to prove that

$$(\text{cm}_{\mathbb{H}_I}, \text{cm}_{\mathbf{M}}, \text{cns}, D(X)) \in \mathcal{L}_{\text{CSS}}$$

for

$$\mathbb{R}_{\text{CSS}} = \left\{ (\text{cm}_{\mathbb{H}_I}, \text{cm}_{\mathbf{M}}, \text{cns}, D(X)), (\mathbb{H}_I, \vec{M}(X), D(X)) \middle| \begin{array}{l} \text{cm}_{\mathbb{H}_I} = \text{Commit}(\mathbb{H}_I), |\mathbb{H}_I| = k, \\ \text{cm}_{\vec{M}} = \text{Commit}(\vec{M}(X)), \vec{M}(X) \in \mathcal{M}_I(X) \\ \vec{s} = \text{Smp}(\text{cns}); \\ D(X) = \vec{s}^\top \vec{M}(X) \end{array} \right\}.$$

- **Decision Phase:** When the proving phase is concluded, the verifier outputs a bit indicating acceptance or rejection.

We note that here the term commit it is used in a loose sense to refer to some well-defined polynomial encoding of the vectors.

Soundness. A Commit-and-Prove checkable subspace sampling argument is ϵ -sound if for any polynomial time prover $\mathcal{P}_{\text{CSS}}^*$:

$$\text{Prob} \left[\begin{array}{l} \text{instance} \notin \mathbb{R}_{\text{CSS}} \\ b = 1 \end{array} \middle| \begin{array}{l} \text{cm} \leftarrow \mathcal{P}_{\text{CSS}}^*; \text{instance} \leftarrow \text{Sample}(\mathcal{P}_{\text{CSS}}^*(\mathbb{H}_I, \vec{M}(X)), \mathcal{V}_{\text{CSS}}(\text{cm})); \\ b \leftarrow \langle \mathcal{P}_{\text{CSS}}^*(\text{instance}), \mathcal{V}_{\text{CSS}}(\text{instance}) \rangle \end{array} \right] \leq \epsilon.$$

5.2 Our Concrete CSS Relation

Similar to [RZ21], we encode matrix \mathbf{M} following Marlin [CHM⁺20], through the bivariate polynomial

$$M(X, Y) = \sum_{j=1}^m \tau_{\text{col}(j)}(0)^{-1} \tau_{\text{col}(j)}(X) \mu_j(Y),$$

where $\text{col} : [m] \mapsto [k]$ is such that $\text{col}(j) = i$ if and only if $M_{j,i}$ is the only non-zero element in row j of \mathbf{M} , and $\{\mu_j(X)\}_{j=1}^m$, $\{\tau_i(X)\}_{i=1}^k$ are the Lagrange interpolation polynomials of some set \mathbb{V} of size m and \mathbb{H}_I of size k , respectively. As in both mentioned works, computing an encoding of a vector sampled in the row space of \mathbf{M} using verifier's coin α , is done through a partial evaluation $M(X, \alpha)$. That is,

$$D(X) = M(X, \alpha) = \sum_{j=1}^m \tau_{\text{col}(j)}(0)^{-1} \tau_{\text{col}(j)}(X) \mu_j(\alpha)$$

is an encoding of $\vec{d} = \sum_{j=1}^m \mu_j(\alpha) \vec{M}_j$, where \vec{M}_j is the j th row of \mathbf{M} and is a unit vector in \mathbb{F}^k . The only difference between their encodings and ours is that we use the set of polynomials $\{\hat{\tau}_i(X) = \frac{\tau_i(X)}{\tau_i(0)}\}_{i=1}^k$ instead of simply $\tau_i(X)$, i.e. $D(X) = \sum_{i=1}^k d_i \hat{\tau}_i(X)$. The reason is that for the inner product in Section 5.4, we will need this *normalized* variant of the Lagrange polynomials that interpolate \mathbb{H}_I .

For each I , the set of allowed vectors polynomials that encode the set of allowed matrices is. Therefore, we define the allowed set of vectors of polynomials as:

$$\mathcal{M}(X)_I = \{(\hat{\tau}_{\text{col}(1)}(X), \dots, \hat{\tau}_{\text{col}(m)}(X)) : \text{for some } \text{col} : [m] \mapsto [k]\},$$

where $\{\tau_i(X)\}_{i=1}^k$ are the Lagrange polynomials for the set \mathbb{H}_I , and $\hat{\tau}_i(X) = \tau_i(0)^{-1} \tau_i(X)$.

Proving well formation of $D(X)$ can be thought now as proving that $D(X) = \vec{s}^\top \vec{M}(X)$ for some $\vec{s}(X) \in \mathcal{M}(X)_I$ and $\vec{s} = \text{Smp}(\text{cns})$.

The commitment algorithm is given by

$$\begin{aligned} \text{Commit}(\mathbb{H}_I) &= z_I(X) = \prod_{\xi \in \mathbb{H}_I} (X - \xi) \\ \text{Commit}(\vec{M}(X)) &= v(X) \text{ where } v(\nu_j) = \xi_{\text{col}(j)}^{-1} \text{ for all } \nu_j \in \mathbb{V} \end{aligned}$$

Note that $v(X)$ uniquely defines some vector $\vec{M}(X) \in \mathcal{M}_{\mathbb{H}_I}$, as it is isomorphic to a map from $[m] \mapsto [k]$.

In the sampling phase the verifier chooses $\alpha \leftarrow \mathbb{F}$ randomly from the field and the prover sets $D(X)$ accordingly. Then the proving phase is run with respect to the relation $\mathbb{R}_{z_I, \vec{M}(X)} \in \mathbb{R}_{\text{CSS}}$.

$$\mathbb{R}_{z_I, \vec{M}(X)} = \left\{ (\alpha, D(X)), (\vec{M}(X), D(X)) : \vec{s} = (\mu_j(\alpha))_{j=1}^m, D(X) = \vec{s}^\top \vec{M}(X) = \sum_{j=1}^m \mu_j(\alpha) \hat{\tau}_{\text{col}(j)}(X) \right\}.$$

5.3 The Scheme

In Fig. 2 we provide a formal PHP for our CSS proving system. During the committing and sampling phase the prover and verifier agree an instance $z_I(X), v(X), \alpha, D(X)$ where $z_I(X) = \prod_{i=1}^k (X - \xi_i)$. Note that $z_I(X)$ is proven to be correctly formed in the Caulk+ core protocol (see Section 3.4).

To prove well formation of $D(X)$, the prover sends a commitment to the polynomial

$$E(X) = M(X, \beta) = \sum_{j=1}^m \tau_{\text{col}(j)}(0)^{-1} \tau_{\text{col}(j)}(\beta) \mu_j(X)$$

It proves that $E(X)$ has exactly this form by showing that (i) $E(X)$ has degree less than m and (ii) $E(\nu_j) = \tau_{\text{col}(j)}(0)^{-1} \tau_{\text{col}(j)}(\beta)$ for all $\nu_j \in V$. We have that point (ii) holds if and only if there exists $Q_1(X)$ such that

$$E(X)(\beta v(X) - 1) + z_I(\beta) z_I(0)^{-1} = z_V(X) Q_1(X).$$

Intuitively, this is done because we have CSS for matrices \mathbf{M}' that have one non-zero element per column (basic matrices in [RZ21]), meaning we can sample an element in the row space of the transpose to \mathbf{M} , which is a vector in the column space of \mathbf{M} . Later, we prove well formation of the desired encoding $D(X) = M(X, \alpha)$ of a vector in the row space of \mathbf{M} by linking $D(X)$ and $E(X)$. In terms of polynomials, we know how to prove partial evaluation of the polynomial $M(X, Y)$ at different values for X but not Y , so we do it and then relate $M(\beta, X)$ with $M(X, \alpha)$.

Committing Phase: \mathcal{P}_{CSS} computes and outputs $z_I(X) = \prod_{i=1}^k (X - \xi_i)$ and $v(X) = \sum_{j=1}^m \xi_{\text{col}(j)}^{-1} \mu_j(X)$.

Sampling Phase: \mathcal{V} sends $\alpha \in \mathbb{F}$ and \mathcal{P} computes and outputs $D(X) = M(X, \alpha) = \vec{\mu}(\alpha) \vec{M}(X) = \sum_{j=1}^m \mu_j(\alpha) \tau_{\text{col}(j)}(X) (\tau_{\text{col}(j)}(0))^{-1}$.

Proving Phase: \mathcal{V} sends $\beta \in \mathbb{F}$. \mathcal{P} computes $E(X) = M(\beta, X) = \sum_{j=1}^m \mu_j(X) \tau_{\text{col}(j)}(\beta) (\tau_{\text{col}(j)}(0))^{-1}$ and $Q_1(X)$ such that

$$E(X)(\beta v(X) - 1) + z_I(\beta) z_I(0)^{-1} = z_V(X) Q_1(X).$$

It outputs $(E(X), Q_1(X))$, a proof that $\deg(E) < m$, and a proof that $z_I(X)$ is a commitment to k distinct roots.

Decision Phase: Accepts if and only if (i) $\deg(E) < m$,

$$(ii) \quad E(X)(\beta v(X) - 1) + z_I(\beta) z_I(0)^{-1} = z_V(X) Q_2(X)$$

$$(iii) \quad D(\beta) = E(\alpha)$$

$$(iv) \quad z_I(X) \text{ is a commitment to } k \text{ distinct roots}^5$$

Fig. 2. Commit-and-Prove CSS for $\vec{M}(X) \in \mathcal{M}_I(X)$.

Theorem 3. *The protocol in Fig 2 satisfies completeness.*

Proof. For equation (ii) recall that $\sum_{j=1}^m \mu_j(X) = 1$ and note that

$$\begin{aligned} E(X)(\beta v(X) - 1) &= \left(\sum_{j=1}^m \tau_{\text{col}(j)}(\beta) (\tau_{\text{col}(j)}(0))^{-1} \mu_j(X) \right) \left(\beta \sum_{j=1}^m \xi_{\text{col}(j)}^{-1} \mu_j(X) - \sum_{j=1}^m \mu_j(X) \right) \\ &= \left(\sum_{j=1}^m \tau_{\text{col}(j)}(\beta) (\tau_{\text{col}(j)}(0))^{-1} \mu_j(X) \right) \left(\sum_{j=1}^m (\beta \xi_{\text{col}(j)}^{-1} - 1) \mu_j(X) \right) \\ &= \sum_{j=1}^m \tau_{\text{col}(j)}(\beta) (\tau_{\text{col}(j)}(0))^{-1} (\beta \xi_{\text{col}(j)}^{-1} - 1) \mu_j(X) \mod z_V(X) \\ &= \sum_{j=1}^m \prod_{i \neq \text{col}(j)} \frac{\beta - \xi_i}{\xi_{\text{col}(j)} - \xi_i} \prod_{i \neq \text{col}(j)} \frac{\xi_{\text{col}(j)} - \xi_i}{-\xi_i} \xi_{\text{col}(j)}^{-1} (\beta - \xi_{\text{col}(j)}) \mu_j(X) \mod z_V(X) \\ &= \sum_{j=1}^m \prod_{i \neq \text{col}(j)} \frac{\beta - \xi_i}{-\xi_i} \xi_{\text{col}(j)}^{-1} (\beta - \xi_{\text{col}(j)}) \mu_j(X) \mod z_V(X) \\ &= -z_I(\beta) z_I(0)^{-1} \sum_{j=1}^m \mu_j(X) \mod z_V(X) = -z_I(\beta) z_I(0)^{-1} \mod z_V(X) \end{aligned}$$

Thus, there exists $Q_1(X)$ such that $E(X)(\beta v(X) - 1) + z_I(\beta) z_I(0)^{-1} = Q_1(X) z_V(X)$

Equation (iii) follows as $D(X) = M(X, \alpha)$ and $E(X) = M(\beta, X)$. \square

Theorem 4. *If $z_I(X) = \prod_{i=1}^k (X - \xi_i)$ for unique roots ξ_i then the CSS protocol in Fig. 2 is sound.*

Proof. Set $v_j = v(\nu_j)$ for all $j \in [m]$, then there exists $q(X) \in \mathbb{F}[X]$ such that $v(X) = \sum_{j=1}^m v_j \mu_j(X) + z_V(X) q(X)$. We first argue that the PHP enforces the form of $E(X)$ with respect to the evaluations $\{v_j\}_{j \in [m]}$.

⁵ In our protocol, well formation of $z_I(X)$ is given by Caulk+ core.

We then argue that given this $E(X)$, the form of $D(X)$ is determined by $v(X)$ and $z_I(X)$ except with negligible probability. Third we will show that v_j^{-1} is a root of $z_I(X)$ for all $j \in [m]$. Finally we will show that $D(X)$ is exactly in our allowed set of matrices.

Form of $E(X)$: We show that

$$E(X) = \sum_{j=1}^m \left(\frac{-z_I(\beta)}{z_I(0)(\beta v_j - 1)} \right) \mu_j(X)$$

Since $\deg(E) < m$, there exist coefficients $\{e_j\}_{j=1}^m$ such that $E(X) = \sum_{j=1}^m e_j \mu_j(X)$. So we have:

$$\begin{aligned} E(X)(\beta v(X) - 1) &= \left(\sum_{j=1}^m e_j \mu_j(X) \right) \left(\beta \sum_{j=1}^m v_j \mu_j(X) + z_V(X)q(X) - \sum_{j=1}^m \mu_j(X) \right) \\ &= \sum_{j=1}^m e_j (\beta v_j - 1) \mu_j(X) \mod z_V(X) \end{aligned}$$

and then equation (ii) says that for all $\nu_j \in \mathbb{V}$,

$$\sum_{j=1}^m e_j (\beta v_j - 1) \mu_j(\nu_j) + z_I(\beta) z_I(0)^{-1} = 0,$$

and then for all $j \in [m]$, $e_j (\beta v_j - 1) = -z_I(\beta) z_I(0)^{-1}$. Thus for all j , $\beta v_j - 1 \neq 0$ and it follows that $e_j = \frac{-z_I(\beta)}{z_I(0)(\beta v_j - 1)}$.

Form of $D(X)$: We show that

$$D(X) = - \sum_{j=1}^m \frac{z_I(X) \mu_j(\alpha)}{z_I(0) v_j(X - v_j^{-1})}$$

except with negligible probability. Let

$$f(X) = \sum_{j=1}^m \left(\prod_{s=1, s \neq j}^m (X v_s - 1) \right) z_I(0)^{-1} z_I(X) \mu_j(\alpha) + D(X) \prod_{j=1}^m (X v_j - 1)$$

Then at random β we have that

$$\begin{aligned} f(\beta) &= \sum_{j=1}^m \left(\prod_{s=1, s \neq j}^m (\beta v_s - 1) \right) z_I(0)^{-1} z_I(\beta) \mu_j(\alpha) + D(\beta) \prod_{j=1}^m (\beta v_j - 1) \\ \Rightarrow \frac{f(\beta)}{\prod_{j=1}^m (\beta v_j - 1)} &= \sum_{j=1}^m \frac{z_I(\beta)}{z_I(0)(\beta v_j - 1)} \mu_j(\alpha) + D(\beta) \\ \Rightarrow \frac{f(\beta)}{\prod_{j=1}^m (\beta v_j - 1)} &= -E(\alpha) + D(\beta) \end{aligned}$$

provided that $\prod_{j=1}^m (\beta v_j - 1) \neq 0$, which is the case as explained above. Now by the verifiers (iii) check we have that $-E(\alpha) + D(\beta) = 0$ and hence that $f(\beta) = 0$. If $f(X) \neq 0$ then $f(\beta) \neq 0$ except with negligible

probability because $v(X), z_I(X), D(X)$ are chosen before β . Thus

$$\begin{aligned} D(X) &= - \sum_{j=1}^m \frac{\left(\prod_{s=1, s \neq j}^m (Xv_s - 1) \right) z_I(0)^{-1} z_I(X) \mu_j(\alpha)}{\prod_{j=1}^m (Xv_j - 1)} \\ &= - \sum_{j=1}^m \frac{z_I(X) \mu_j(\alpha)}{z_I(0) v_j (X - v_j^{-1})} \end{aligned}$$

Form of $v(\mathbf{X})$: We show that for all $j \in [m]$, v_j^{-1} is a root of $z_I(X)$. In other words, there exists a map $\text{col} : [m] \mapsto [k]$ such that for all $j \in [m]$, $v_j^{-1} = \xi_{\text{col}(j)}$. Indeed, define $J = \{j : v_j^{-1} \notin \mathbb{H}_I\}$, and the set $V = \{v_j : j \in J\}$. We assume for contradiction that there is some v_j^{-1} that is not a root of $z_I(X)$, which means that V is not empty. Then,

$$D_{J^c}(X) = - \sum_{j \in [m] \setminus J} \frac{z_I(X) \mu_j(\alpha)}{z_I(0) v_j (X - v_j^{-1})}$$

is a polynomial, and we can write:

$$\begin{aligned} D(X) - D_{J^c}(X) &= - \sum_{j \in J} \frac{z_I(X) \mu_j(\alpha)}{z_I(0) v_j (X - v_j^{-1})} \\ &= - \frac{z_I(X)}{z_I(0)} \sum_{v \in V} \frac{1}{v(X - v^{-1})} \left(\sum_{j: v_j = v} \mu_j(\alpha) \right) = - \frac{z_I(X)}{z_I(0)} \sum_{v \in V} \frac{P_v(\alpha)}{v(X - v^{-1})}, \end{aligned}$$

where, for any $v \in V$, $P_v(X) = \sum_{j: v_j = v} \mu_j(X)$. Regardless of α , this identity can only hold if, for all $v \in V$, $P_v(\alpha) = 0$. Indeed, the left side is a polynomial because the prover sent $D(X)$ and $D_{J^c}(X)$ is a polynomial by definition of J , but a polynomial cannot be equal to a sum of non-trivial rational functions with different poles. The probability that $P_v(\alpha) = 0$ for all $v \in V$ can be bounded by the probability that $P_v(\alpha) = 0$ for any single $v \in V$. But this probability is at most $\frac{m-1}{|\mathbb{F}|}$, because all these polynomials were defined independently of α . We conclude that the probability that V is not empty is negligible.

$D(X)$ is in the space of allowed matrices: Let us substitute our mapping col from v_j^{-1} to the roots of $z_I(X)$ into our expression for $D(X)$:

$$D(X) = - \sum_{j=1}^m \frac{z_I(X) \mu_j(\alpha)}{z_I(0) \xi_{\text{col}(j)}^{-1} (X - \xi_{\text{col}(j)})} = \sum_{j=1}^m \frac{-\xi_{\text{col}(j)} z_I(X) \mu_j(\alpha)}{z_I(0) (X - \xi_{\text{col}(j)})}$$

Now the lagrange polynomials for the set $\{\xi_i\}_{i=1}^k$ are given by

$$\tau_i(X) = \prod_{s=1, s \neq i}^k \frac{X - \xi_s}{\xi_i - \xi_s} = \frac{z_I(X)}{(X - \xi_i) \prod_{s=1, s \neq i}^k (\xi_i - \xi_s)}$$

Thus

$$\tau_{\text{col}(j)}(0) = \frac{z_I(0)}{-\xi_{\text{col}(j)} \prod_{s=1, s \neq \text{col}(j)}^k (\xi_{\text{col}(j)} - \xi_s)} \quad \text{and} \quad \tau_{\text{col}(j)}(0)^{-1} \tau_{\text{col}(j)}(X) = \frac{-\xi_{\text{col}(j)} z_I(X)}{z_I(0) (X - \xi_{\text{col}(j)})}$$

Substituting these into our expression for $D(X)$ yields our result

$$D(X) = \sum_{j=1}^m \tau_{\text{col}(j)}(0)^{-1} \tau_{\text{col}(j)}(X) \mu_j(\alpha) = \sum_{j=1}^m \mu_j(\alpha) \hat{\tau}_{\text{col}(j)}(X)$$

□

5.4 Inner products from Generalized Sumcheck

In this section, we introduce a scheme for proving that for two vectors $\vec{a}, \vec{b} \in \mathbb{F}^k$ encoded as polynomials $a(X), b(X)$ it is true that $\vec{a} \cdot \vec{b} = \sigma$. Importantly, our scheme uses polynomial encodings of both vectors, but in the case of vector \vec{a} the encoding is *normalized*, in particular, $a(X) = \sum_{i=1}^k a_i \hat{\tau}_i(X) = \sum_{i=1}^k a_i \frac{\tau_i(X)}{\tau_i(0)}$ for the set of Lagrange interpolation polynomials corresponding to some set \mathbb{H}_I of size k . This is because we will instantiate our inner product argument setting $a(X)$ to be polynomial $D(X)$ from the previous section and $D(X) = \sum_{i=1}^k \frac{d_i}{\tau_i(0)} \tau_i(X)$, where \vec{d} is a vector in the rowspace of the lookup matrix \mathbf{M} .

Formally, we build a proof system for the following relation:

$$\mathbb{R}_{\text{ginnerprod}} = \left\{ (a(X), b(X), z_I(X)) ; (\vec{a}, \vec{b}, \mathbb{H}_I) \mid a_i = a(\xi_i) \tau_i(0)^{-1}, b_i = b(\xi_i) \text{ and } \sum_{i=1}^k a_i b_i = \sigma \right\}$$

for $\mathbb{H}_I = \{\xi_i\}_{i=1}^k$ some fixed set of known size k and $\{\tau_i(X)\}_{i=1}^k$ its Lagrange interpolation polynomials. Let $z_I(X)$ be the vanishing polynomial of \mathbb{H}_I . Inspired in the linear check of Aurora [BCR⁺19], we compute an encoding $\sum_{i=1}^k a_i b_i \hat{\tau}_i(X)$ of the Hadamard product between \vec{a}, \vec{b} and use a univariate sumcheck argument to obtain the inner product from it. Importantly, since \mathbb{H}_I may contain any set of distinct points that do not necessarily form a multiplicative group, we instantiate our inner product argument with the generalized sumcheck in Section 3.5.

The intuition is that for all Lagrange interpolation polynomials $\{\lambda_j(X)\}_{j=1}^N$ corresponding to a multiplicative subgroup \mathbb{H} of size N , we have $\lambda_j(0) = N^{-1}$. Then, for any polynomial $p(X)$, to prove that $\sum_{j=1}^N p(\omega^j) = \sigma$, we note that $p(X) = \sum_{j=1}^N p(\omega^j) \lambda_j(X) \bmod z_H(X)$, and the latter polynomial evaluates to $\sum_{j=1}^N p(\omega^j) \lambda_j(0) = \sigma N^{-1}$ in 0. When it comes to arbitrary sets \mathbb{H}_I , the corresponding Lagrange polynomials $\{\tau_i(X)\}_{i=1}^k$ take different values when evaluated in 0. The generalized sumcheck observes that as soon as one of the encoding polynomials uses *normalized* Lagrange polynomials, that is $\hat{\tau}_i(X) = \frac{\tau_i(X)}{\tau_i(0)}$, the inner product behaves the same way. The protocol is described in Fig. 3.

Committing Phase:

- Set the polynomial

$$z_I(X) = \prod_{\xi \in \mathbb{H}_I} (X - \xi)$$

Proving Phase: \mathcal{P}_{IP} computes $R(X), Q(X)$ such that

$$a(X)b(X) - \sigma = XR(X) + z_I(X)Q(X).$$

It outputs $(R(X), Q(X))$ and a proof that $z_I(X)$ has distinct k roots.

Decision Phase: \mathcal{V}_{IP} accepts if and only if (i) $\deg(R(X)) < k - 1$, and

$$a(X)b(X) = XR(X) + z_I(X)Q(X),$$

and (ii) $z_I(X)$ has distinct k roots.

Fig. 3. PHP for a generalised inner product argument. As before, well-formation of $z_I(X)$ is given by Caulk+ core.

Theorem 5 (Inner Product Polynomial Relation). *The argument in Fig. 3 is a statistically sound PHP for the relation $\mathbb{R}_{\text{ginnerprod}}$.*

Proof. Let ξ_i , $i = 1, \dots, k$ be the roots of $z_I(X)$. If we define $a_i = a(\xi_i)\tau_i(0)^{-1}$ and $b_i = b(\xi_i)$, we can represent $a(X) = \sum_{i=1}^k a_i\tau_i(0)^{-1}\tau_i(X) + z_I(X)q_1(X)$ and $b(X) = \sum_{j=1}^k b_j\tau_j(X) + z_I(X)q_2(X)$. Then there exists $q_3(X)$ such that

$$a(X)b(X) = \left(\sum_{i=1}^k a_i\tau_i(0)^{-1}\tau_i(X) \right) \left(\sum_{j=1}^k b_j\tau_j(X) \right) + z_I(X)q_3(X).$$

We recall that for $i \neq j$, $z_I(X)|\tau_i(X)\tau_j(X)$ and also $z_I(X)|\tau_i^2(X)$ for all $i \in [k]$. Hence, there exists $q_4(X)$ such that

$$a(X)b(X) = \sum_i a_i b_i \tau_i(0)^{-1} \tau_i(X) + z_I(X)q_4(X)$$

Finally observe that

$$\sum_i a_i b_i \tau_i(0)^{-1} \tau_i(X) = \sum_i a_i b_i + X R(X).$$

This is because the left hand side evaluated at 0 is $\sum_i a_i b_i \tau_i(0)^{-1} \tau_i(0) = \sum_i a_i b_i$, for some $X R(X)$ of degree strictly smaller than $z_I(X)$. Putting this together we have that

$$a(X)b(X) = \sum_i a_i b_i + X R(X) + z_I(X)q_4(X)$$

and therefore $\sum_i a_i b_i = \sigma$ if and only if

$$a(X)b(X) - \sigma = X R(X) + z_I(X)Q(X)$$

for some $R(X)$, $Q(X)$ for $R(X)$ of degree $k - 2$. \square

6 Baloo Full Construction

In this section, we provide our full *Baloo* construction for proving the relation

$$\mathbb{R}_{\text{lookup}} = \left\{ \text{cm}; \phi(X) \mid \begin{array}{l} \text{cm} = \text{Commit}(\text{srs}, \phi(X)) \\ \forall \nu \in \mathbb{V}, \phi(\nu) \in \{c_s\}_{s=1}^N \end{array} \right\}$$

where \mathbb{V} is a set of roots of unity that is *independent* from N (the size of the lookup table C) and **Commit** is the KZG commitment algorithm [KZG10]. For simplicity we have omitted \vec{c} and \mathbb{V} from the relation description. The prover for the full construction is formally given in Fig. 5 and the verifier is given in Fig. 6 with all optimisations included.

Before describing the full construction, we describe the protocol for performing commit-and-prove lookups. *Baloo* is the result of compiling our building blocks into a succinct proof and use it after the Caulk+ core protocol in Section 3.4. The compiled subprotocol is given in Fig. 4. In other words we describe a protocol for proving the relation

$$\mathbb{R}_{\text{cp-expansion}} = \left\{ \text{cm}; \phi(X) \mid \begin{array}{l} \text{cm} = \text{Commit}(\text{srs}, \phi(X)) \\ \forall \nu \in \mathbb{V}, \phi(\nu) \in \{t_i\}_{i=1}^k \end{array} \right\}$$

with respect to some table $\{t_i\}_{i=1}^k$ that is potentially unknown to the verifier. For simplicity we have omitted \vec{t} and \mathbb{V} from the relation description.

The commit and prove lookup takes as input commitments t , cm to vectors $\vec{t} \in \mathbb{F}^k$, $\vec{a} \in \mathbb{F}^m$ encoded as polynomials $t(X), \phi(X)$. It also takes as input $[z_I]_2$ a commitment to the vanishing polynomial respect to a set \mathbb{H}_I of known size k . The polynomial $t(X)$ is computed using the Lagrange interpolation basis corresponding to set \mathbb{H}_I . The polynomial $\phi(X)$ is computed using the Lagrange interpolation basis corresponding to subgroup \mathbb{V} .

The prover aims to convince the verifier that there exists some mapping $\text{col} : [m] \mapsto [k]$ such that $a_j = t_{\text{col}(j)}$ for all $j \in [m]$. The prover and verifier use as subroutine the CSS from Fig. 2 to agree on an encoding $D(X)$ such that

$$D(X) = \sum_{j=1}^m \mu_j(\alpha) \hat{\tau}_{\text{col}(j)}(X)$$

at a random point α . Then the prover and verifier engage in the generalised inner product argument from Fig. 3 so show that, if $d_i = D(\xi_i)\tau_i(0)^{-1}$, and $t(x_i) = t_i$,

$$\sum_{i=1}^k d_i t_i = \phi(\alpha).$$

Since $d_i = \sum_{j \in \text{col}^{-1}(i)} \mu_j(\alpha)$,

$$\phi(\alpha) = \sum_{i=1}^k d_i t_i = \sum_{i=1}^k t_i \sum_{j \in \text{col}^{-1}(i)} \mu_j(\alpha) = \sum_{j=1}^m \mu_j(\alpha) t_{\text{col}(j)}$$

we thus have that $s_j = \phi(\nu_j) = t_{\text{col}(j)}$ with overwhelming probability, as required.

6.1 Compilation of the **cp-expansion** Subprotocol

We compile our PHP into a non-interactive succinct argument following the compiler in [CFF⁺21], and obtain the protocol in Fig. 4. This proves soundness of our scheme under the **q-dlog** assumption.

Recall that this protocol is a subroutine of *Baloo* in Fig. 5 and thus the common inputs to the systems are the commitments to $z_I(X), C(X)$ and $t(X)$. The SRS of the full scheme is $(\{[x^s]_{1,2}\}_{s=1}^N)$, where N is the maximum degree among all polynomials. Prover and Verifier instantiate \mathcal{P}_{IP} and \mathcal{V}_{IP} for the PHP of Fig. 3. All oracle polynomials sent by \mathcal{P}_{IP} are translated into polynomials evaluated (in the source groups) at x . Polynomial equations are checked by the verifier from group elements using pairings. For quadratic checks, the prover must send the commitments to the polynomials in different source groups.

All the openings at one point, as well as the degrees of the opened polynomials, are proven using the KZG polynomial commitment. For degree checks with $\deg(p) = d < N$ and $p(X)$ a polynomial that is never opened, the prover sends a single extra polynomial $\hat{p}(X) = X^{N-d}p(X)$, and the verifier checks one extra pairing equation as explained in Section 3.3.

6.2 The Full *Baloo* Construction

The final construction *Baloo* is described in Fig. 5 and Fig. 6. It consists simply of combining the Caulk+ core protocol from Section 3.4 and the **cp-expansion** argument in Fig. 4. We also apply several efficiency optimisations which are specified below.

Efficiency Optimisations In this section we describe some optimizations that can be applied to the protocol in Fig. 4 in order to achieve a construction with smaller proof size and that requires less work from the verifier.

Opening t polynomials in one point. As noted in [GWC19],[CHM⁺20], whenever we have many openings of different polynomials at the same point, the prover can provide a joint proof after receiving a random element $\gamma \in \mathbb{F}$ from the verifier, i.e., if

$$\begin{aligned} (u_1, [w_1]_1) &\leftarrow \text{KZG.Open}(\text{srs}_{\text{KZG}}, f_1(X), \deg = d, \alpha) \\ (u_2, [w_2]_1) &\leftarrow \text{KZG.Open}(\text{srs}_{\text{KZG}}, f_2(X), \deg = d, \alpha) \end{aligned}$$

then $[w]_1 = [w_1]_1 + \gamma [w_2]_1$ is a proof that $f_1(X) + \gamma f_2(X)$ opens to $u_1 + \gamma u_2$ at α .

Common input: $\mathbf{t} = [t(x)]_1, \mathbf{cm} = [\phi(x)]_1, [z_I]_2 = [z_I(x)]_2$ and $\mathbf{srs} = \{\{[x^s]_{1,2}\}_{s=1}^N\}$

Prover_{cp-e}:

- Compute $v(X) = \sum_{j=1}^m \xi_{\text{col}(j)}^{-1} \mu_j(X)$,
- Output $\pi_1 = ([v]_2 = [v(x)]_2,)$.

Verifier_{cp-e}: Send $\alpha \in \mathbb{F}$

Prover_{cp-e}:

- Compute $D(X) = M(X, \alpha) = \sum_{j=1}^m \mu_j(\alpha) \hat{\tau}_{\text{col}(j)}(X)$ and find $R(X), Q_2(X)$ such that

$$D(X)t(X) - \phi(\alpha) = X R(X) + z_I(X)Q_2(X)$$

- Set $\hat{R} = X^{N-m+2}$
- Output $\pi_2 = ([D]_2 = [D(x)]_2, [R]_1 = [R(x)]_1, [\hat{R}]_1 = [\hat{R}(x)]_1, [Q_2]_1 = [Q_2(x)]_1)$.

Verifier_{cp-e}: Send $\beta \in \mathbb{F}$

Prover_{cp-e}:

- Compute $E(X) = M(\beta, X) = \sum_{j=1}^m \mu_j(X) \hat{\tau}_{\text{col}(j)}(\beta)$ and $Q_1(X)$ such that

$$E(X)(\beta v(X) - 1) + z_I(\beta)z_I(0)^{-1} = z_V(X)Q_1(X)$$

- $(u_1, [w_1]_1) \leftarrow \text{Open.KZG}(\mathbf{srs}_{\text{KZG}}, [E]_1, \deg = m-1, \alpha)$
- $(u_2, [w_2]_1) \leftarrow \text{Open.KZG}(\mathbf{srs}_{\text{KZG}}, \mathbf{cm}, \deg = \perp, \alpha)$
- $(u_3, u_4, [w_3]_1) \leftarrow \text{Open.KZG}(\mathbf{srs}_{\text{KZG}}, [z_I]_2, \deg = \perp, (0, \beta))$
- $(u_5, [w_4]_1) \leftarrow \text{Open.KZG}(\mathbf{srs}_{\text{KZG}}, [D]_2, \deg = \perp, \beta)$
- Output $\pi_3 = ([E]_1 = [E(x)]_1, [Q_1]_1 = [Q_1(x)]_1, (u_1, [w_1]_1), (u_2, [w_2]_1), (u_3, u_4, [w_3]_1), (u_5, [w_4]_1))$.

Verifier_{cp-e}: Accept if and only if

- (i) $e(\mathbf{t}, [D]_2) - e([1]_1 u_2, [1]_2) = e([R]_1, [x]_2) + e([Q_2]_1, [z_I]_2)$
- (ii) $e([E]_1, (\beta[v]_2 - 1)) + e([1]_1(1 - u_3^{-1}u_4), [1]_2) = e([Q_1]_1, [z_V(x)]_2)$
- (iii) $e([R]_1, [x^{N-m+2}]_2) = e([\hat{R}]_1, [1]_2)$
- (iv) $u_1 = u_5$.
- (v) $1 \leftarrow \text{KZG.Verify}(\mathbf{srs}_{\text{KZG}}, [E]_1, \deg = m-1, \alpha, u_1, [w_1]_1)$
- (vi) $1 \leftarrow \text{KZG.Verify}(\mathbf{srs}_{\text{KZG}}, \mathbf{cm}, \deg = \perp, \alpha, u_2, [w_2]_1)$
- (vii) $1 \leftarrow \text{KZG.Verify}(\mathbf{srs}_{\text{KZG}}, [z_I]_2, \deg = \perp, (0, \beta), (u_3, u_4), [w_3]_1)$
- (viii) $1 \leftarrow \text{KZG.Verify}(\mathbf{srs}_{\text{KZG}}, [D]_2, \deg = \perp, \beta, u_5, [w_4]_1)$

Fig. 4. cp-expansion argument for proving $\phi(X)$ has entries in a (potentially unknown) subtable $t(X)$.

Openings for Pairings. To save the verifier some work, we use a technique introduced in [GWC19] and attributed to M. Maller. In order to verify that $a(X)b(X) = c(X)d(X)$ for $a(X), b(X), c(X), d(X)$ the algebraic representations of $[a]_1, [b]_2, [c]_1, [d]_2$, instead of asking the verifier to check that

$$e([a]_1, [b]_2) = e([c]_1, [d]_2),$$

we ask the prover to show that $[b]_2, [d]_2$ open to u_1, u_2 at β and that $u_1[a]_1 - u_2[c]_1$ opens to zero at β . Note that now the prover can also commit to $b(X)$ and $d(X)$ in \mathbb{G}_1 instead of \mathbb{G}_2 . We apply this technique to the equations that verify the inner product relation and the well formation of $[E]_1$; that is, equations (i) and (ii). Note that we can open this equations together with other elements. Indeed, we will check equation (i) by opening a polynomial $[P_1]_1$ at β , and batch that KZG opening together with the one for $[D]_1$.

Degree checks. Degree checks as $\deg(f) \leq k < d$ can be included in a KZG proof that $f(\alpha) = u$ if the prover sets $\hat{w}(X) = \frac{f(X)-f(\alpha)}{X-\alpha}$, outputs $(u, [w]_1 = [\hat{w}(x)x^{d-k+1}])$ and the verifier checks

$$e([f]_1 - [u]_1, [x^{d-k+1}]_2) = e([w]_1, [x - \alpha]_2),$$

as explained in Section 3.3. This is conditional on α being randomly chosen after $f(X)$.

Throughout *Baloo* we require 3 degree checks: (i) that $\deg(E(X)) < m - 1$, (ii) that $\deg(z_I(X)) = m$, and (iii) that $\deg(R(X)) = m - 2$. For (i) we check via a KZG opening that $E(X)$ has bounded degree. For (ii) we check that $z_I(X) - X^m$ has degree bounded by $m - 1$ during our opening check that $z_I(0)$ is correct. Degree bounding $f(X) < k$ via an opening at 0 checks that

$$e([f]_1 - [u]_1, [1]_2) = e([w]_1, [x]_2) \text{ and } e([f]_1, [x^{d-k+2}]_2) = e([w]_2, [x]_2),$$

because 0 is not a random point.

For (iii) we recall that the polynomial $R(X)$ is sent for the inner product relation to show that $a(X)b(X) - \sigma = XR(X) + z_I(X)Q(X)$. In our optimised protocol we instead send $\bar{R}(X) = XR(X)$ and show that $\bar{R}(0) = 0$ and that $\bar{R}(X)$ has degree bounded by $m - 1$. We then show that $a(X)b(X) - \sigma = \bar{R}(X) + z_I(X)Q(X)$. This is equivalent because $\bar{R}(0) = 0$ if and only if $\bar{R}(X) = XR(X)$. Where we can batch this check with opening and degree bounding $z_I(X)$ at the same point (namely 0) and with the same degree ($m - 1$), this check is essentially free.

Batching Pairings. We also apply standard techniques to batch pairings that share the same elements in one of the two groups. Namely, upon sampling a uniform $\gamma_2 \in \mathbb{F}$, the verifier can aggregate the equations

$$\begin{aligned} e([a]_1, [b]_2) &= e([c_1]_1, [d]_2) \text{ and } e([a]_1, [b_2]_2) = e([c_2]_1, [d]_2), \\ &\text{as } e([a]_1, [b_1 + \gamma b_2]_2) = e([c_1 + \gamma c_2]_1, [d]_2) \end{aligned}$$

Note that we can adapt KZG openings equations so they can be batched further, namely if we parse the verification pairing as $e([f]_1 - u + [w]_1\alpha, [1]_2) = e([w]_1, [x]_2)$, then two openings of different polynomials at different points can be verified by two pairings.

Finally, note that in order to check $E(\alpha) = D(\beta)$, the proves needs to provide proof of both openings but can only send $u_2 = E(\alpha)$ and the verifier checks $D(\beta)$ opens to u_2 as well.

7 Baloo Prover Cost

In this section we elaborate on the Prover's computational costs while showing that those are quasilinear in m .

Common input: $C = [C(x)]_1, [z_H(x)]_1 = [\prod_{i=0}^{N-1} (x - \omega^i)]_1, \text{srs} = \{[x^i]_{1,2}\}_{i=1}^d$

1. Take as input $\{[Q_i(x)]_1, [H_i(x)]_1\}_{i=1}^N$ and $\phi(X)$
2. Choose $I \subset [N]$ such that $|I| = k$ and $\forall v \in \mathbb{V}, \exists \xi \in \mathbb{H}_I$ s.t. $\phi(v) = C(\xi)$
3. Compute $v(X) = \sum_{j=1}^m \xi_{\text{col}(j)}^{-1} \mu_j(X)$
4. Output $\pi_1 = ([z_I]_2 = [z_I(x)]_2, [v]_1 = [v(x)]_1, \mathbf{t} = [t(x)]_1)$.
5. Receive $\alpha \in \mathbb{F}$
6. Compute $D(X) = M(X, \alpha) = \sum_{j=1}^m \mu_j(\alpha) \hat{\tau}_{\text{col}(j)}(X)$
7. Find $R(X), Q_2(X)$ such that $\deg(R(X)) < m - 1$, $R(0) = 0$, and

$$D(X)t(X) - \phi(\alpha) = R(X) + z_I(X)Q_2(X)$$

8. Output $\pi_2 = ([D]_1 = [D(x)]_1, [R]_1 = [R(x)]_1, [Q_2]_1 = [Q_2(x)]_1, \dots)$.

9. Receive $\beta \in \mathbb{F}$

10. Compute $E(X) = M(\beta, X) = \sum_{j=1}^m \mu_j(X) \hat{\tau}_{\text{col}(j)}(\beta)$ and $Q_1(X)$ such that

$$E(X)(\beta v(X) - 1) + z_I(\beta)z_I(0)^{-1} = z_V(X)Q_1(X)$$

11. Output $\pi_3 = ([E]_1 = [E(x)]_1, [Q_1]_1 = [Q_1(x)]_1)$

12. Receive $\rho, \gamma \in \mathbb{F}$

13. Compute $([a_1]_1, [a_2]_1, \dots) \leftarrow \text{Prover}_{\mathsf{C+}}(t(X), \mathbb{H}_I)$ and set $[a]_1 = [a_1]_1 + \gamma[a_2]_2$
Compress Caulk+ proof.

14. Set $u_1 = E(\alpha), u_2 = \phi(\alpha),$

$$\hat{w}_1(X) = \frac{E(X) - u_1}{X - \alpha} + \gamma \frac{\phi(X) - u_2}{X - \alpha}$$

Prove that $E(\alpha) = u_1, \phi(\alpha) = u_2, \deg(E(X)) < m$

15. Set $u_3 = z_I(0)$ and

$$w_2(X) = \frac{z_I(X) - u_3}{X} + \gamma \frac{R(X)}{X} + \gamma^2 X^{d-m+1} (z_I(X) - X^m) + \gamma^3 X^{d-m+1} R(X)$$

Prove that $z_I(0) = u_3, R(0) = 0, \deg(z_I(X)) = m$ and $\deg(R(X)) < m$

16. Set $P_1(X) = t(X)D(\beta) - \phi(\alpha) - R(X) - z_I(\beta)Q_2(X), u_4 = z_I(\beta)$ and

$$w_3(X) = \frac{D(X) - u_1}{X - \beta} + \gamma \frac{z_I(X) - u_4}{X - \beta} + \gamma^2 \frac{P_1(X)}{X - \beta}$$

Prove that $D(\beta) = E(\alpha), z_I(\beta) = u_4, t(X)D(X) - \phi(\alpha) = R(X) + z_I(X)Q_2(X)$

17. Set $u_5 = E(\rho), P_2(X) = E(\rho)(\beta v(X) - 1) + z_I(\beta)z_I(0)^{-1} - z_V(\rho)Q_1(X),$

$$w_4(X) = \frac{E(X) - u_5}{(X - \rho)} + \gamma \frac{P_2(X)}{X - \rho}$$

Prove that $E(X)(\beta v(X) - 1) + z_I(\beta)z_I(0)^{-1} = z_V(X)Q_1(X)$

18. Set $s = d - m + 1$ for d the maximum power of x in srs and output

$$\pi_3 = (u_1, u_2, u_3, u_4, u_5, [a]_1, [w_1]_1 = [\hat{w}_1(x)x^s]_1, [w_2]_1 = [w_2(x)]_1, [w_3]_1 = [w_3(x)]_1, [w_4]_1 = [w_4(x)]_1).$$

Fig. 5. Optimized Baloo prover. Underlined steps are messages from Verifier (Fig. 6).

Common input: $C = [C(x)]_1, [z_H(x)]_1 = [\prod_{i=0}^{N-1} (x - \omega^i)]_1, \text{srs} = \{[x^i]_{1,2}\}_{i=1}^d$

Take cm as input.

Receive $\pi_1 = ([z_I]_2, [v]_1, [\text{t}]_1)$ and send $\alpha \in \mathbb{F}$

Receive $\pi_2 = ([D]_1, [R]_1, [Q_2]_1)$ and send $\beta \in \mathbb{F}$

Receive $\pi_3 = ([E]_1, [Q_1]_1)$ and send $\rho, \gamma \in \mathbb{F}$

Receive $\pi_4 = (u_1, u_2, u_3, u_4, u_5, [a]_1, [w_1]_1, [w_2]_1, [w_3]_1, [w_4]_1)$

Compute

$$[P_1]_1 = u_1[\text{t}] - u_2[1]_1 - [R]_1 - u_4[Q_2]_1$$

$$\# [P_1]_1 = [t(x)D(\beta) - \phi(\alpha) - R(x) - z_I(\beta)Q_2(x)]_1$$

$$[P_2]_1 = u_5(\beta[v]_1 - 1) + u_3^{-1}u_4 - z_V(\rho)[Q_1]_1$$

$$\# [P_2]_1 = [E(\rho)(\beta v(x) - 1) + z_I(\beta)z_I(0)^{-1} - z_V(\rho)Q_1(x)]_1$$

Set $s = d - m + 1$ for d the maximum power of x in srs and accept if and only if

1. $e((C - t) + \gamma[z_H(x)]_1, [1]_2) - e([a]_1, [z_I]_2) = 0$

Check that $C - t$ elements $[a_1]_1$ and $[a_2]_1$ verify.

2. $e(\alpha[w_1]_1, [1]_2) - e([w_1]_1, [x]_2) + e([E]_1 + \gamma\text{cm} - [u_1 + \gamma u_2]_1, [x^s]_2) = 0$

Check that $E(\alpha) = u_1, \phi(\alpha) = u_2, \deg(E(X)) < m$

3. $e(-[u_3]_1 + \gamma[R]_1, [1]_2) - e([w_2]_1, [x]_2) + e([1 + \gamma^2 x^{s+1}]_1, [z_I]_2) + e(-\gamma^2[x^m]_1 + \gamma^3[R]_1, [x^{s+1}]_2) = 0$

Check that $z_I(0) = u_3, R(0) = 0, \deg(z_I(X)) = m$ and $\deg(R(X)) < m$

4. $e(\beta[w_3]_1 + [D]_1 + \gamma^2[P_1]_1 - [u_1 + \gamma u_4]_1, [1]_2) - e([w_3]_1, [x]_2) + e([\gamma]_1, [z_I]_2) = 0$

Check that $D(\beta) = E(\alpha), z_I(\beta) = u_4, t(X)D(X) - \phi(\alpha) = R(X) + z_I(X)Q_2(X)$

5. $e(\rho[w_4]_1 + [E]_1 + \gamma[P_2]_1 - [u_5]_1, [1]_2) - e([w_4]_1, [x]_2) = 0$

Check that $E(\rho) = u_5$ and $E(X)(\beta v(X) - 1) + z_I(\beta)z_I(0)^{-1} = z_V(X)Q_1(X)$

These checks can be batched into 1 equation with 5 pairings.

Fig. 6. Optimized Baloo verifier.

7.1 Generic algorithms

An excellent survey of various algorithms for polynomials with pseudocode is given in [vzGG13]. Let \mathbb{F} be a domain with Fast Fourier Transform of size N . Polynomials in $\mathbb{F}[X]$ are considered as vectors of coefficients in the standard basis $\{1, X, X^2, \dots, X^N\}$ unless stated otherwise. The set I does not support FFTs. The computational costs are counted in operations in \mathbb{F} . We are using the following basic results (everywhere $d < N$):

- Multiplication: two polynomials of degree d can be multiplied in $O(d \log d)$ time.
- Inversion: given a polynomial f of degree d can be inverted modulo X^ℓ , $\ell > d$, in $O(d \log d)$ time.
- Division: a polynomial f of degree d can be divided with a remainder by a polynomial g of degree $d' < d$ in $O(d \log d)$ time, i.e. we can find $q(X), r(X)$ of degree $d'' < d$ such that

$$f(X) = q(X)g(X) + r(X)$$

- Vanishing polynomial: a polynomial $z_I(X)$ that vanishes on set I of size d can be computed in $O(d \log^2 d)$ time.
- Evaluation: a polynomial f of degree d can be evaluated in d points in $O(d \log^2 d)$ time.
- Interpolation: a polynomial f of degree d with values c_i at points x_i , $0 \leq i \leq d$, can be computed (interpolated) in $O(d \log^2 d)$ time.

7.2 Prover costs in *Baloo*

For simplicity we assume that $m = k$.

Aggregation of individual proofs . The subset opening proofs for the set of points $\mathbb{H}_I \subseteq \mathbb{H}$ are computed as

$$[H]_1 = \sum_{i \in I} \left(\prod_{s \in I, s \neq i}^m \frac{1}{(\omega^i - \omega^s)} \right) [H_i]_1$$

The coefficients $r_i = \left(\prod_{s \in I, s \neq i}^m \frac{1}{(\omega^i - \omega^s)} \right)$ are altogether computed in $O(m \log^2 m)$ time as follows. Let $Z'_I(X)$ be the derivative of $Z_I(X)$ then we have $r_i = \frac{1}{Z'_I(\omega^i)}$ [vzGG13, p. 300]. We use a vanishing polynomial reconstruction algorithm (see above) and symbolically compute $Z'_I(X)$ in $O(m \log^2 m)$ time. Then we evaluate $Z'_I(X)$ over I also in $O(d \log^2 d)$ time. Thus $[H]_1$ can be computed in m group operations.

Running time of Caulk+ core. The costs of Fig. 5 break down as follows:

- Polynomial $\frac{C(x) - t(x)}{z_I(x)}$ similarly to $[H]_1$ using $O(m \log^2 m)$ field operations. Then it takes m group operations to compute W_1 .
- The group element W_2 is computed as a linear combination of $[H_i(x)]_1$ as in [ZBK⁺22] in time m group and $O(k \log^2 k)$ field operations (see above).
- The polynomial in W_3 has $O(m)$ nonzero coefficients and thus needs at most m group operations to be computed.

In Fig. 5 the element W_3 is unused and this computation can be omitted. Overall we need $2m$ group operations and $O(m \log^2 m)$ field operations.

Running time of cp-expansion argument Fig. 4 We first note that Lagrange polynomials $\mu_j(X)$ and $\tau_i(X)$ have succinct form. Concretely we have, assuming $|\mathbb{V}| = m$:

$$\mu_j(X) = \frac{X^m - 1}{m\nu^{-j}(X - \nu^j)}, \quad \tau_i(X) = \frac{z_I(X)}{z'_I(\xi_i)(X - \xi_i)} = \frac{r_i z_I(X)}{(X - \xi_i)}$$

where $z'_I(X)$ is the derivative of $z_I(X)$. All $\mu_j(X)$ can be batch-evaluated in m points in $m \log m$ time as one evaluation is $\log m$ time. For $\tau_i(X)$ we compute $r_i = \frac{1}{z'_I(\xi_i)}$ using the evaluation algorithm for $z_I(X)$ in $O(m \log^2 m)$ time, and then inverting in $m \log(m)$ time. Then in order to batch-evaluate all $\tau_i(X)$ at some point β , we evaluate $z_I(X)$ at β in m time and each $\frac{r_i}{\beta - \xi_i}$ in $\log m$ time. These costs are all in \mathbb{F} .

The field operation costs of Fig. 4 break down as follows:

- Polynomial $v(X)$ has degree m and can be computed via interpolation in $O(m \log^2 m)$.
- Polynomial $D(X)$ is computed by interpolation as follows. We first batch-evaluate $\mu_j(X)$ at α in $O(m \log m)$ time. Then we batch-evaluate $\tau_i(X)$ at 0 in $O(m \log^2 m)$ time, so that we know all coefficients of $\tau_{\text{col}(j)}(X)$ in the sum. Those coefficients are exactly values of $D(X)$ at ξ_i . From those we interpolate $D(X)$ in $O(m \log^2 m)$ time.
- Polynomials $R(X), Q_1(X), Q_2(X)$ can be computed using the division algorithm (above) in $O(m \log^2 m)$ time
- Polynomial $E(X)$ is computed by interpolation again. As for $D(X)$ we batch-evaluate all $\tau_i(X)$ at β so that we know all coefficients of $\mu_j(X)$ in $O(m \log^2 m)$ time. As $\mu_j(X)$ are defined over a subgroup of roots of unity, the interpolation of $E(X)$ is in $O(m \log m)$ time.

Computing $[z_I(x)]_2$ takes $m \mathbb{G}_2$ operations. Computing the 12 \mathbb{G}_1 elements that Prover sends takes $11m \mathbb{G}_1$ operations as those commitments are either to polynomials of degree m (those are $t, v, D, E, Q_1, R, Q_2, w_3, w_4$) or have at most m non-zero coefficients ($[w_1]_1, [w_2]_1$), or need constant time ($[a]_1$). Overall we need $11m \mathbb{G}_1$ operations, $m \mathbb{G}_2$ operations, and $O(m \log^2 m)$ field operations.

Full running cost By summing up the prover costs for Caulk+ core and cp-expansion we get that the total Prover cost in *Baloo* is $13m \mathbb{G}_1$ operations, $m \mathbb{G}_2$ operations and $O(m \log^2 m)$ field operations.

8 Faster SNARKs using *Baloo*

Commit-and-prove lookup tables are especially suitable for the Ethereum Foundation’s zero-knowledge Ethereum Virtual Machine (zkEVM), which nowadays uses Halo2 with KZG commitments as a backend. The zkEVM is an effort to outsource the evaluation of the Ethereum Virtual Machine on some chosen inputs to a powerful prover, and then demonstrate that the result is correct to a computationally constrained verifier. Importantly, what the zkEVM aims to is a succinct proof of validity of the blocks, and since they are public, no zero-knowledge is required. In other words, the proofs are sound but not zero-knowledge, and *Baloo* can be safely used.

In this section we describe an overview of how lookups are currently used in the Halo2 proving system [BGH20] and claim *Baloo* can be used as a drop in replacement to the Halo2 lookup argument with better prover efficiency. In other words *Baloo* is backwards compatible with instantiations of Halo2 that use KZG commitments.

Baloo is a proving system for the relation that

$$\mathbb{R}_{\text{lookup}} = \left\{ \text{cm; } \phi(X) \mid \begin{array}{l} \text{cm} = \text{Commit}(\text{srs}, \phi(X)) \\ \forall \nu \in \mathbb{V}, \phi(\nu) \in \{c_s\}_{s=1}^N \end{array} \right\}$$

where \mathbb{V} is a set of roots of unity that is *independent* from N (the size of the lookup table C) and Commit is the KZG commitment algorithm [KZG10]. This lookup argument only handles a single column. Suppose instead that we want to prove $f(x) = y$ by precomputing all possible values $T = \{(x_s, f(x_s))\}_{s=1}^N$ and looking up whether $(x, y) \in T$. To achieve this we require more functionality from our lookup argument. In particular we need to be able to prove a lookup argument over *multicolumned* tables.

Multi-column Baloo We build on the Halo2 approach⁶.

Given a lookup with input column polynomials $[\phi_0(X), \dots, \phi_{k-1}(X)]$ and a multi-columned table of the form $C = \{c_{i,j}\}_{i,s=1}^{i=k, s=N}$, their prover shows that for all $\nu \in \mathbb{V}$, there exists some s such that $(\phi_0(\nu), \dots, \phi_{k-1}(\nu)) = (c_{0,s}, \dots, c_{k-1,s})$. It does this by taking a random linear combination of the input column polynomials and the table columns and then running a lookup argument over the compressed values. We present a similar compression for *Baloo* such that we can run lookups over multi-columned tables.

Similarly than in Caulk+, described in Section 3.4, the pre-processing phase commits to the table $\{c_{i,s}\}_{i,s=1}^{k,N}$ by committing to the column polynomials

$$C_s(X) = \sum_{s=1}^N c_s \lambda_s(X)$$

for $\{\lambda_s(X)\}_{s=1}^N$ a set of roots of unity $H = \{\omega^s\}_{s=1}^N$ of size N . The pre-processing phase also computes auxiliary information for the prover, namely it computes commitments to the polynomials

$$\{Q_{i,s}(X) = (C_s(X) - C_s(\omega_j))/(X - \omega^s)\}_{i,s=1}^{k,N}, H_s(X) = \{z_H(X)/(X - \omega^s)\}_{s=1}^N$$

in time $kN \log_2(N)$.

Given the input column polynomials $[\phi_0(X), \dots, \phi_{k-1}(X)]$ we sample a random value θ . Suppose that I is the set of points such that $j \in I$ if and only if $(c_{0,s}, \dots, c_{k-1,s})$ appears in the input columns i.e. $(c_{0,s}, \dots, c_{k-1,s}) \in \{(\phi_0(\nu), \dots, \phi_{k-1}(\nu))\}_{\nu \in \mathbb{V}}$. Then the prover commitments to the compressed polynomials

$$\begin{aligned} \phi_{\text{compressed}}(X) &= \phi_0(X) + \theta \phi_1(X) \dots + \theta^{k-1} \phi_{k-1}(X) &= \sum_{j=1}^m \left(\sum_{i=0}^k \theta^i \phi_i(\nu_j) \right) \mu_j(X) \\ C_{\text{compressed}}(X) &= C_0(X) + \theta C_1(X) \dots + \theta^{k-1} C_{k-1}(X) &= \sum_{s=1}^N \left(\sum_{i=0}^k \theta^i c_{i,s} \right) \lambda_s(X) \\ \text{for } s \in I, Q_{\text{compressed},j} &= Q_{0,s}(X) + \theta Q_{1,s}(X) \dots + \theta^{k-1} Q_{k-1,s}(X) &= \left(\sum_{i=0}^{k-1} \theta^i C_i(X) - \theta^i C_i(\omega^s) \right) / (X - \omega^s) \end{aligned}$$

Then we have that $C_{\text{compressed}}(X)$ describes the table $C_{\text{compressed}} = \{\sum_{i=0}^{k-1} \theta^i c_{i,s}\}_{s=1}^N$. The randomiser θ is sampled from a large field. Thus the probability that $\sum_{i=0}^k \theta^i \phi_i(\nu) \in C_{\text{compressed}}$ is negligible unless there exists some s such that $(\phi_0(\nu), \dots, \phi_{k-1}(\nu)) = (c_{0,s}, \dots, c_{k-1,s})$. The auxiliary information that the prover requires for efficiency, namely commitments to $\{H_s(X), Q_{\text{compressed},s}(X)\}_{s \in I}$, can be computed in km time where $m = |\mathbb{V}|$ is the number of lookups.

Thus for $\text{cm}_{\text{compressed}}$ a commitment to $\phi_{\text{compressed}}(X)$, the prover demonstrates that $\text{cm}_{\text{compressed}} \in R_{\text{lookup}}$ with respect to the table $C_{\text{compressed}}$.

References

- AR20. Shashank Agrawal and Srinivasan Raghuraman. Kvac: Key-value commitments for blockchains and beyond. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part III*, volume 12493 of *Lecture Notes in Computer Science*, pages 839–869. Springer, 2020. 3
- BaCCL21. Olivier Bégassat, Alexandre Bellin and Théodore Chapuis-Chkaiban, and Nicolas Liochon1. A specification for a zk-evm, 2021. <https://ethresear.ch/t/a-zk-evm-specification/11549>. 1

⁶ <https://zcash.github.io/halo2/design/proving-system/circuit-commitments.html>

- BB04. Dan Boneh and Xavier Boyen. Short signatures without random oracles. In Christian Cachin and Jan Camenisch, editors, *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proceedings*, volume 3027 of *Lecture Notes in Computer Science*, pages 56–73. Springer, 2004. 3
- BCC⁺15. Jonathan Bootle, Andrea Cerulli, Pyrrhos Chaidos, Essam Ghadafi, Jens Groth, and Christophe Petit. Short accountable ring signatures based on DDH. In Günther Pernul, Peter Y. A. Ryan, and Edgar R. Weippl, editors, *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part I*, volume 9326 of *Lecture Notes in Computer Science*, pages 243–265. Springer, 2015. 2
- BCF⁺21. Daniel Benaroch, Matteo Campanelli, Dario Fiore, Kobi Gurkan, and Dimitris Kolonelos. Zero-knowledge proofs for set membership: Efficient, succinct, modular. In Nikita Borisov and Claudia Díaz, editors, *Financial Cryptography and Data Security - 25th International Conference, FC 2021, Virtual Event, March 1-5, 2021, Revised Selected Papers, Part I*, volume 12674 of *Lecture Notes in Computer Science*, pages 393–414. Springer, 2021. 3
- BCG⁺13. Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for C: verifying program executions succinctly and in zero knowledge. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*, volume 8043 of *Lecture Notes in Computer Science*, pages 90–108. Springer, 2013. 2
- BCG⁺18. Jonathan Bootle, Andrea Cerulli, Jens Groth, Sune K. Jakobsen, and Mary Maller. Arya: Nearly linear-time zero-knowledge proofs for correct program execution. In Thomas Peyrin and Steven D. Galbraith, editors, *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part I*, volume 11272 of *Lecture Notes in Computer Science*, pages 595–626. Springer, 2018. 2
- BCR⁺19. Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. Aurora: Transparent succinct arguments for R1CS. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part I*, volume 11476 of *Lecture Notes in Computer Science*, pages 103–128. Springer, 2019. 5, 6, 13
- BG13. Stephanie Bayer and Jens Groth. Zero-knowledge argument for polynomial evaluation with application to blacklists. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*, pages 646–663. Springer, 2013. 2
- BG18. Jonathan Bootle and Jens Groth. Efficient batch zero-knowledge arguments for low degree polynomials. In Michel Abdalla and Ricardo Dahab, editors, *Public-Key Cryptography - PKC 2018 - 21st IACR International Conference on Practice and Theory of Public-Key Cryptography, Rio de Janeiro, Brazil, March 25-29, 2018, Proceedings, Part II*, volume 10770 of *Lecture Notes in Computer Science*, pages 561–588. Springer, 2018. 2
- BGH20. Sean Bowe, Jack Grigg, and Daira Hopwood. Halo2. URL: <https://github.com/zcash/halo2>, 2020. 2, 21
- CDGM19. Melissa Chase, Apoorvaa Deshpande, Esha Ghosh, and Harjasleen Malrai. Seemless: Secure end-to-end encrypted messaging with less trust. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 1639–1656. ACM, 2019. 2
- CEO22. Matteo Campanelli, Felix Engelmann, and Claudio Orlandi. Zero-knowledge for homomorphic key-value commitments with applications to privacy-preserving ledgers. In Clemente Galdi and Stanislaw Jarecki, editors, *Security and Cryptography for Networks - 13th International Conference, SCN 2022, Amalfi, Italy, September 12-14, 2022, Proceedings*, volume 13409 of *Lecture Notes in Computer Science*, pages 761–784. Springer, 2022. 2
- CFF⁺21. Matteo Campanelli, Antonio Faonio, Dario Fiore, Anaïs Querol, and Hadrián Rodríguez. Lunar: A toolbox for more efficient universal and updatable zksnarks and commit-and-prove extensions. In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2021 - 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6-10, 2021, Proceedings, Part III*, volume 13092 of *Lecture Notes in Computer Science*, pages 3–33. Springer, 2021. 7, 15

- CFG⁺20. Matteo Campanelli, Dario Fiore, Nicola Greco, Dimitris Kolonelos, and Luca Nizzardo. Incrementally aggregatable vector commitments and applications to verifiable decentralized storage. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part II*, volume 12492 of *Lecture Notes in Computer Science*, pages 3–35. Springer, 2020. 3
- CFH⁺21. Matteo Campanelli, Dario Fiore, Semin Han, Jihye Kim, Dimitris Kolonelos, and Hyunok Oh. Succinct zero-knowledge batch proofs for set accumulators. *IACR Cryptol. ePrint Arch.*, page 1672, 2021. 2
- CFQ19. Matteo Campanelli, Dario Fiore, and Anaïs Querol. Legosnark: Modular design and composition of succinct zero-knowledge proofs. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 2075–2092. ACM, 2019. 6
- CHM⁺20. Alessandro Chiesa, Yunchong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas P. Ward. Marlin: Preprocessing zksnarks with universal and updatable SRS. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part I*, volume 12105 of *Lecture Notes in Computer Science*, pages 738–768. Springer, 2020. 8, 15
- Eth22. Zkvm introduction, 2022. <https://github.com/privacy-scaling-explorations/zkvm-specs/blob/master/specs/introduction.md>. 1
- FK20. Dankrad Feist and Dmitry Khovratovich. Fast amortized kate proofs, 2020. 4
- FKL18. Georg Fuchsbauer, Eike Kiltz, and Julian Loss. The algebraic group model and its applications. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part II*, volume 10992 of *LNCS*, pages 33–62. Springer, 2018. 3
- GG17. Essam Ghadafi and Jens Groth. Towards a classification of non-interactive computational assumptions in cyclic groups. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part II*, volume 10625 of *Lecture Notes in Computer Science*, pages 66–96. Springer, 2017. 3
- GK15. Jens Groth and Markulf Kohlweiss. One-out-of-many proofs: Or how to leak a secret and spend a coin. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, volume 9057 of *Lecture Notes in Computer Science*, pages 253–280. Springer, 2015. 2
- GK22. Ariel Gabizon and Dmitry Khovratovich. flookup: Fractional decomposition-based lookups in quasi-linear time independent of table size. *Cryptology ePrint Archive*, 2022. 2
- GW20. Ariel Gabizon and Zachary J Williamson. plookup: A simplified polynomial protocol for lookup tables. *Cryptology ePrint Archive*, 2020. 1, 2
- GWC19. Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *IACR Cryptol. ePrint Arch.*, page 953, 2019. 15, 17
- HHK⁺21. Yunchong Hu, Kian Hooshmand, Harika Kalidhindi, Seung Jin Yang, and Raluca Ada Popa. Merkle²: A low-latency transparency log system. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 285–303. IEEE, 2021. 2
- KZG10. Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In Masayuki Abe, editor, *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings*, volume 6477 of *Lecture Notes in Computer Science*, pages 177–194. Springer, 2010. 2, 3, 14, 21
- MKL⁺20. Sarah Meiklejohn, Pavel Kalinnikov, Cindy S. Lin, Martin Hutchinson, Gary Belvin, Mariana Raykova, and Al Cutter. Think global, act local: Gossip and client audits in verifiable data structures. *CoRR*, abs/2011.04551, 2020. 2
- PK22. Jim Posen and Assimakis A. Kattis. Caulk+: Table-independent lookup arguments. *Cryptology ePrint Archive*, Paper 2022/957, 2022. <https://eprint.iacr.org/2022/957>. 1, 2, 4, 5
- Pol22. Polygon zkvm documentation, 2022. <https://docs.hermes.io/zkEVM/Overview/Overview/>. 1
- RZ21. Carla Ràfols and Arantxa Zapico. An algebraic framework for universal and updatable snarks. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology - CRYPTO 2021 - 41st Annual International*

- Cryptology Conference, CRYPTO 2021, Virtual Event, August 16-20, 2021, Proceedings, Part I*, volume 12825 of *Lecture Notes in Computer Science*, pages 774–804. Springer, 2021. 5, 6, 7, 8, 9
- Sta22. Starknet, 2022. <https://starkware.co/starknet/>. 1
- TAB⁺20. Alin Tomescu, Ittai Abraham, Vitalik Buterin, Justin Drake, Dankrad Feist, and Dmitry Khovratovich. Aggregatable subvector commitments for stateless cryptocurrencies. In Clemente Galdi and Vladimir Kolesnikov, editors, *Security and Cryptography for Networks - 12th International Conference, SCN 2020, Amalfi, Italy, September 14-16, 2020, Proceedings*, volume 12238 of *Lecture Notes in Computer Science*, pages 45–64. Springer, 2020. 4
- TBP⁺19. Alin Tomescu, Vivek Bhupatiraju, Dimitrios Papadopoulos, Charalampos Papamanthou, Nikos Tsiropoulos, and Srinivas Devadas. Transparency logs via append-only authenticated dictionaries. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 1299–1316. ACM, 2019. 2
- TFBT21. Nirvan Tyagi, Ben Fisch, Joseph Bonneau, and Stefano Tessaro. Client-auditable verifiable registries. *IACR Cryptol. ePrint Arch.*, page 627, 2021. 2
- vzGG13. Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra (3. ed.)*. Cambridge University Press, 2013. 20
- ZBK⁺22. Arantxa Zapico, Vitalik Buterin, Dmitry Khovratovich, Mary Maller, Anca Nitulescu, and Mark Simkin. Caulk: Lookup arguments in sublinear time. CCS '22: 2022 ACM SIGSAC Conference on Computer and Communications Security, Los Angeles, CA, USA, November 7 - 11, 2022, 2022. 1, 2, 3, 4, 5, 20
- Zha22. Ye Zhang. Introducing zkEVM, 2022. <https://scroll.io/blog/zkEVM>. 1
- zks22. zkEVM faq, 2022. <https://docs.zksync.io/zkEVM/>. 1

A Proof of Thm 1

Proof. Let \mathcal{A} be an algebraic adversary attempting to break knowledge soundness. We design an extractor \mathcal{E} that behaves as follows. When the adversary $(C(X), \mathbf{t}, [z_I]_2, \pi) \leftarrow \mathcal{A}(\mathsf{srs})$ outputs a proof $\pi = (W_1, W_2, W_3)$, then it also outputs the representations $w_1(X), w_3(X)$ of maximum degree N such that

$$W_1 = [w_1(x)]_1, W_3 = [w_3(x)]_1$$

The extractor \mathcal{E} computes

$$z_I(X) = w_3(X)X^{-(N-k+1)} + X^k, t(X) = C(X) - w_1(X)z_I(X)$$

and returns $\mathbb{H}_I, t(X)$ where \mathbb{H}_I consists of the roots of $z_I(X)$.

We show that either \mathcal{E} succeeds with overwhelming probability or we can construct a reductions \mathcal{B}_1 and \mathcal{B}_2 such that

$$\mathcal{A}_{\mathcal{A}, \mathcal{E}}^0(\lambda) \leq \mathcal{A}_{\mathcal{B}_1}^{\text{q-sfrac}}(\lambda) + \mathcal{A}_{\mathcal{B}_2}^{\text{q-dlog}}(\lambda)$$

Game⁰ \mapsto Game¹: Let Game⁰ be the original knowledge soundness game. We first transition to a game Game¹ that behaves identically to Game⁰ except that, when \mathcal{A} outputs the representation $w_3(X)$, if $z_I(X) = w_3(X)X^{-(N-k+1)} + X^k$ is not a degree k polynomial (with positive degree monomials only) then Game¹ aborts.

We show the existence of a reduction \mathcal{B}_1 such that

$$\mathcal{A}_{\mathcal{A}, \mathcal{E}}^0(\lambda) \leq \mathcal{A}_{\mathcal{A}, \mathcal{E}}^1(\lambda) + \mathcal{A}_{\mathcal{B}_1}^{\text{q-sfrac}}(\lambda)$$

The reduction \mathcal{B}_1 gets as input srs and forwards this reference string to run $(C(X), \mathbf{t}, [z_I]_2, \pi) \leftarrow \mathcal{A}(\mathsf{srs})$. When \mathcal{A} outputs a proof $\pi = (W_1, W_2, W_3)$, then it also outputs the representation $w_3(X)$ of maximum degree N such that

$$z_I = [w_3(x)x^{-N+k-1} + x^k]_2$$

Write $w_3(X) = \sum_{s=0}^N a_s X^s$. Then \mathcal{B}_1 returns

$$\sum_{s=0}^{N-k} a_s X^s, X^{N-k+1}, [z_I]_2 - [x^k]_2 - \left[\sum_{s=N-k+1}^N a_s x^s \right]_2$$

If $a_s \neq 0$ for $0 \leq s \leq N - k$, then the degree of $\sum_{s=0}^{N-k} a_s X^s$ is less than $N - k + 1$ and hence \mathcal{B}_1 breaks the q-sfrac assumption.

If $a_s = 0$ for all $0 \leq s \leq N - k$ then

$$z_I(X) = w_3(X)X^{-(N-k+1)} + X^k = \sum_{s=0}^{k-1} a_{N-k+1+s} X^s + X^k$$

which is a degree k polynomial.

Game¹ \mapsto Game² We second transition to a game Game² that behaves identically to Game¹ except that, when \mathcal{A} outputs the representation $w_3(X)$, if $z_I(X) = w_3(X)X^{-(N-k+1)} + X^k$ does not divide $z_H(X)$, then Game² aborts. We show the existence of a reduction \mathcal{B}_2 such that

$$\mathcal{A}_{\mathcal{A}, \mathcal{E}}^1(\lambda) \leq \mathcal{A}_{\mathcal{A}, \mathcal{E}}^2(\lambda) + \mathcal{A}_{\mathcal{B}_1}^{\text{qdlog}}(\lambda)$$

See that if Game² does not abort, then $z_I = [f(X)]$ for some $f(X)$ of degree k that divides $z_H(X)$. This means that $z_I(X) = \prod_{i=1}^k (X - \xi_i)$ for \mathbb{H}_I some subset of \mathbb{H} of size k .

The reduction \mathcal{B}_2 gets as input srs and forwards this reference string to run $(C(X), \mathbf{t}, [z_I]_2, \pi) \leftarrow \mathcal{A}(\text{srs})$. When \mathcal{A} outputs a proof $\pi = (W_1, W_2, W_3)$, then it also outputs the representation $w_3(X), w_2(X)$ of maximum degree d such that

$$z_I = [f(x)]_2 = [w_3(x)x^{-N+k-1} + x^k]_2, W_2 = [w_2(x)]_1$$

Then \mathcal{B}_2 computes the degree N polynomial $g(X) = Z_H(X) - z_I(X)w_2(X)$ and solves to find the N roots. It checks amongst these roots whether any solution x corresponds to the qdlog challenge and if yes it returns x . Else it aborts.

By the second verification equation we have that $Z_H(x) - g(x)w_2(x) = 0$ whenever \mathcal{A} convinces the verifier. See that if $g(X)$ does not divide $Z_H(X)$, then $Z_H(X) - g(X)w_2(X) \neq 0$. But then x must lie in the roots and \mathcal{B}_2 succeeds.

Game² \mapsto 0 We finally show that for any adversary \mathcal{A}

$$\mathcal{A}_{\mathcal{A}, \mathcal{E}}^2(\lambda) = 0$$

Indeed, when \mathcal{A} also outputs the representation $w_3(X)$, we have that either $[z_I]_2 = [z_I(x)]_2$ for $z_I(X) = w_3(X)X^{-(N-k+1)} + X^k$ a degree N polynomial dividing $z_H(X)$, or Game² aborts. The adversary \mathcal{A} also outputs a representation $w_1(X)$ of maximum degree N such that $[W]_1 = [w_1(x)]_1$. By the first verification equation we have that $t(X) = C(X) - w_1(X)z_I(X)$ is such that $\mathbf{t} = [t(X)]_1$. Further $t(\xi_i) = C(\xi_i) + 0$ for all $i \in [k]$, making $z_I(X)$ and $t(X)$ a correct witness. \square

Cached quotients for fast lookups

Liam Eagen, Dario Fiore, Ariel Gabizon,

Blockstream
IMDEA software institute
Zeta Function Technologies



cq:* Cached quotients for fast lookups

Liam Eagen
Blockstream

Dario Fiore
IMDEA software institute

Ariel Gabizon
Zeta Function Technologies

December 30, 2022

Abstract

We present a protocol for checking the values of a committed polynomial $f(X) \in \mathbb{F}_{<n}[X]$ over a multiplicative subgroup $\mathbb{H} \subset \mathbb{F}$ of size n are contained in a table $t \in \mathbb{F}^N$. After an $O(N \log N)$ preprocessing step, the prover algorithm runs in time $O(n \log n)$. Thus, we continue to improve upon the recent breakthrough sequence of results [ZBK⁺22, PK22, GK22, ZGK⁺22] starting from Caulk [ZBK⁺22], which achieve sublinear complexity in the table size N . The two most recent works in this sequence [GK22, ZGK⁺22] achieved prover complexity $O(n \cdot \log^2 n)$.

Moreover, **cq** has the following attractive features.

1. As in [ZBK⁺22, PK22, ZGK⁺22] our construction relies on homomorphic table commitments, which makes them amenable to vector lookups.
2. As opposed to [ZBK⁺22, PK22, GK22, ZGK⁺22] the **cq** verifier doesn't involve pairings with prover defined \mathbb{G}_2 points, which makes recursive aggregation of proofs more convenient.

1 Introduction

The *lookup problem* is fundamental to the efficiency of modern zk-SNARKs. Somewhat informally, it asks for a protocol to prove the values of a committed polynomial $f(X) \in \mathbb{F}_{<n}[X]$ are contained in a table T of size N of predefined legal values. When the table T corresponds to an operation without an efficient low-degree arithmetization in \mathbb{F} , such a protocol produces significant savings in proof construction time for programs containing the operation. Building on previous work of [BCG⁺18], **plookup** [GW20] was the first to explicitly describe a solution to this problem in the polynomial-IOP context. **plookup** described a protocol with prover complexity quasilinear in both n and N . This left the intriguing question of whether the dependence on N could be made *sublinear*.

*Pronounced “seek you”.

after performing a preprocessing step for the table T . **Caulk** [ZBK⁺22] answered this question in the affirmative by leveraging bi-linear pairings, achieving a run time of $O(n^2 + n \log N)$. **Caulk+** [PK22] improved this to $O(n^2)$ getting rid of the dependence on table size completely.¹

Naturally, the quadratic dependence on n of these works made them impractical for a circuit with many lookup gates. This was resolved in two more recent protocols - **baloo** [ZGK⁺22] and **Flookup** [GK22] achieving a runtime of $O(n \log^2 n)$. While **Flookup** has better concrete constants, **baloo** preserved an attractive feature of **Caulk** - using a *homomorphic commitment* to the table. This means that given commitments cm_1, cm_2 to tables T_1, T_2 with elements $\{a_i\}, \{b_i\}$ respectively; we can check membership in the set of elements $\{a_i + ab_i\}$ by running the protocol with $\text{cm} := \text{cm}_1 + \alpha \cdot \text{cm}_2$ as the table commitment. This is crucial for vector lookups that have become popular in zk-SNARKs, as described in Section 4 of [GW20].

One drawback of all four recent constructions - **Caulk**, **Caulk+**, **baloo**, **Flookup**; is that they require the verifier perform a pairing where both \mathbb{G}_1 and \mathbb{G}_2 pairing arguments are not fixed in the protocol, but prover defined. This makes it harder to recursively aggregate multiple proofs via random combination, in the style described e.g. in Section 8 of [BCMS20].

1.1 Our results

In this paper, we present a protocol called **cq** - short for “cached quotients” which is a central technical component in the construction (and arguably in all four preceding works). **cq**

1. Improves asymptotic prover performance in field operations from $O(n \log^2 n)$ to $O(n \log n)$, and has smaller constants in group operations and proof size compared to **baloo**.
2. Uses homomorphic table commitments similarly to **Caulk**, **Caulk+** and **baloo**, enabling convenient vector lookups.
3. Achieves for the first time in this line of work convenient aggregatability by having all verifier pairings use fixed protocol-defined \mathbb{G}_2 arguments.

Table 1: Scheme comparison. n = witness size, N = Table size, “Aggregatable”= All prover defined pairing arguments are in \mathbb{G}_1

Scheme	Preprocessing	Proof size	Prover Work	Verifier Work	Homomorphic?	Aggregatable?
Caulk [ZBK ⁺ 22]	$O(N \log N) \mathbb{F}, \mathbb{G}_1$	$14 \mathbb{G}_1, 1 \mathbb{G}_2, 4 \mathbb{F}$	$O(n^2 + n \cdot \log(N)) \mathbb{F}, \mathbb{G}_1$	$4P$	✓	✗
Caulk+ [PK22]	$O(N \log N) \mathbb{F}, \mathbb{G}_1$	$7 \mathbb{G}_1, 1 \mathbb{G}_2, 2 \mathbb{F}$	$O(n^2) \mathbb{F}, \mathbb{G}_1$	$3P$	✓	✗
Flookup [GK22]	$O(N \log^2 N) \mathbb{F}, \mathbb{G}_1$	$6 \mathbb{G}_1, 1 \mathbb{G}_2, 4 \mathbb{F}$	$6n \mathbb{G}_1, n \mathbb{G}_2, O(n \log^2 n) \mathbb{F}$	$3P$	✗	✗
baloo [ZGK ⁺ 22]	$O(N \log N) \mathbb{F}, \mathbb{G}_1$	$12 \mathbb{G}_1, 1 \mathbb{G}_2, 4 \mathbb{F}$	$13n \mathbb{G}_1, n \mathbb{G}_2, O(n \log^2 n) \mathbb{F}$	$5P$	✓	✗
cq (this work)	$O(N \log N) \mathbb{F}, \mathbb{G}_1$	$8 \mathbb{G}_1, 3 \mathbb{F}$	$8n \mathbb{G}_1, O(n \log n) \mathbb{F}$	$5P$	✓	✓

¹A nuance is that while the *number* of field and group operations are independent of table size, the field and group must be larger than the table in all these constructions, including this paper.

1.2 Technical Overview

We explain our protocol in the context of the line of work starting from [ZBK⁺22].

The innovation of Caulk To restate the problem, we have an input polynomial $f(X)$, a table \mathbf{t} of size N encoded as the values of a polynomial $T(X) \in \mathbb{F}_{\leq N}[X]$ on a subgroup \mathbb{V} of size N . We want to show f 's values on a subgroup \mathbb{H} of size n are contained in \mathbf{t} ; concisely that $f|_{\mathbb{H}} \subset \mathbf{t}$. We think of the parameters as $n \ll N$. We want our prover \mathbf{P} to perform a number of operations *sublinear* in N , or ideally, a number of operations depending only on n .

One natural approach - is to send the verifier \mathbf{V} a polynomial T_f encoding the n *values from \mathbf{t} actually used in f* , and then run a lookup protocol using T_f . The challenging problem is then to prove T_f *actually encodes values from T* . Speaking imprecisely, the “witness” to T_f 's correctness is a quotient Q of degree $N-n$. It would defeat our purpose to actually compute Q - as that would require $O(N)$ operations.

The central innovation of Caulk [ZBK⁺22] is the following observation: If we pre-compute commitments to certain quotient polynomials, we can compute in a number of operations depending only on n , the *commitment to Q* . Moreover, having only a commitment to Q suffices to check, via pairings, that T_f is valid.

This approach was a big step forward, enabling for the first time lookups sublinear in table size. However, it has the following disadvantage: “Extracting” the subtable of values used in f , is analogous to looking at restrictions of the original table polynomial to arbitrary sets - far from the nice subgroups we are used to in zk-SNARK world. Very roughly speaking, this is why all previous four works end up needing to work with interpolation and evaluation of polynomials on arbitrary sets. The corresponding algorithms for working on such sets have asymptotics of $O(n \cdot \log^2 n)$ rather than the $O(n \log n)$ we get for subgroups (of order 2^k for example).

Our approach The key difference between [ZBK⁺22, PK22, GK22, ZGK⁺22] and **cq** is that we use the idea of succinct computation of quotient commitments, not to extract a subtable, *but to directly run an existing lookup protocol on the original large table more efficiently*. Specifically, we use as our starting point the “logarithmic derivative based lookup” of [Eag22, Hab22].

[Hab22] utilizes the following lemma (cf. Lemma 2.4 or Lemma 5 in [Hab22]): $f|_{\mathbb{H}} \subset \mathbf{t}$ if and only if for some $m \in \mathbb{F}^N$

$$\sum_{i \in [N]} \frac{m_i}{X + t_i} = \sum_{i \in [n]} \frac{1}{X + f_i},$$

as rational functions. [Hab22] checks this identity on a random β , by sending commitments to polynomials A and B whose values correspond to the summands evaluated at β of the LHS and RHS respectively. Given commitments to A, B , we can check the

above equality holds via various sumcheck techniques, e.g. as described in [BCR⁺19] (cf. Lemma 2.1). The RHS is not a problem because it is a sum of size n . Computing A 's commitment is actually not a problem either, because the number of its non-zero values on \mathbb{V} is at most n . So when precomputing the commitments to the Lagrange base of \mathbb{V} , we can compute A 's commitment in n group operations.

The main challenge is to convince the verifier \mathbf{V} that A is correctly formed. This is equivalent to the existence of a quotient polynomial $Q_A(X)$ such that

$$A(X)(T(X) + \beta) - m(X) = Q_A(X) \cdot Z_{\mathbb{V}}(X).$$

It can be seen that this is the same $Q_A(X)$ as when writing

$$A(X)T(X) = Q_A(X)Z_{\mathbb{V}}(X) + R(X),$$

for $R(X) \in \mathbb{F}_{<N}[X]$.

Here is where our central innovation, and the term “cached quotients” come from. We observe that while computing Q_A would take too long, we can compute the commitment $[Q_A(x)]_1$ to Q_A in $O(n)$ operations as follows. We precompute for each $L_i(X)$ in the Lagrange basis of \mathbb{V} its quotient commitment when multiplying with $T(X)$, i.e. the commitment to $Q_i(X)$ such that for some remainder $R_i(X) \in \mathbb{F}_{<N}[X]$.

$$L_i(X)T(X) = Q_i(X) \cdot Z_{\mathbb{V}}(X) + R_i(X).$$

Given the commitments $[Q_i(x)]_1$, $[Q_A(x)]_1$ can be computed in $O(n)$ \mathbb{G}_1 -operations via linear combination. Moreover, all the elements $[Q_i(x)]_1$ can be computed in an $O(N \log N)$ preprocessing phase leveraging the work of Feist and Khovratovich[FK]. See Section 3 for details on this.

2 Preliminaries

2.1 Terminology and Conventions

We assume our field \mathbb{F} is of prime order. We denote by $\mathbb{F}_{<d}[X]$ the set of univariate polynomials over \mathbb{F} of degree smaller than d . We assume all algorithms described receive as an implicit parameter the security parameter λ .

Whenever we use the term *efficient*, we mean an algorithm running in time $\text{poly}(\lambda)$. Furthermore, we assume an *object generator* \mathcal{O} that is run with input λ before all protocols, and returns all fields and groups used. Specifically, in our protocol $\mathcal{O}(\lambda) = (\mathbb{F}, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t, e, g_1, g_2, g_t)$ where

- \mathbb{F} is a prime field of super-polynomial size $r = \lambda^{\omega(1)}$.
- $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t$ are all groups of size r , and e is an efficiently computable non-degenerate pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_t$.
- g_1, g_2 are uniformly chosen generators such that $e(g_1, g_2) = g_t$.

We usually let the λ parameter be implicit, i.e. write \mathbb{F} instead of $\mathbb{F}(\lambda)$. We write \mathbb{G}_1 and \mathbb{G}_2 additively. We use the notations $[x]_1 := x \cdot g_1$ and $[x]_2 := x \cdot g_2$.

We often denote by $[n]$ the integers $\{1, \dots, n\}$. We use the acronym e.w.p for “except with probability”; i.e. e.w.p γ means with probability *at least* $1 - \gamma$.

universal SRS-based public-coin protocols We describe public-coin (meaning the verifier messages are uniformly chosen) interactive protocols between a prover and verifier; when deriving results for non-interactive protocols, we implicitly assume we can get a proof length equal to the total communication of the prover, using the Fiat-Shamir transform/a random oracle. Using this reduction between interactive and non-interactive protocols, we can refer to the “proof length” of an interactive protocol.

We allow our protocols to have access to a structured reference string (SRS) that can be derived in deterministic $\text{poly}(\lambda)$ -time from an “SRS of monomials” of the form $\{[x^i]_1\}_{a \leq i \leq b}$, $\{[x^i]_2\}_{c \leq i \leq d}$, for uniform $x \in \mathbb{F}$, and some integers a, b, c, d with absolute value bounded by $\text{poly}(\lambda)$. It then follows from Bowe et al. [BGM17] that the required SRS can be derived in a universal and updatable setup requiring only one honest participant; in the sense that an adversary controlling all but one of the participants in the setup does not gain more than a $\text{negl}(\lambda)$ advantage in its probability of producing a proof of any statement.

For notational simplicity, we sometimes use the SRS srs as an implicit parameter in protocols, and do not explicitly write it.

The Aurora lemma Our sumcheck relies on the following lemma originally used in the Aurora construction ([BCR⁺19], Remark 5.6).

Lemma 2.1. *Let $H \subset \mathbb{F}$ be a multiplicative subgroup of size t . For $f \in \mathbb{F}_{<t}[X]$, we have*

$$\sum_{a \in H} f(a) = t \cdot f(0).$$

2.2 The algebraic group model

We introduce some terminology from [GWC19] to capture analysis in the Algebraic Group Model of Fuchsbauer, Kiltz and Loss[FKL18].

In our protocols, by an *algebraic adversary* \mathcal{A} in an SRS-based protocol we mean a $\text{poly}(\lambda)$ -time algorithm which satisfies the following.

- For $i \in \{1, 2\}$, whenever \mathcal{A} outputs an element $A \in \mathbb{G}_i$, it also outputs a vector v over \mathbb{F} such that $A = \langle v, \mathsf{srs}_i \rangle$.

First we say our srs has degree Q if all elements of srs_i are of the form $[f(x)]_i$ for $f \in \mathbb{F}_{\leq Q}[X]$ and uniform $x \in \mathbb{F}$. In the following discussion let us assume we are executing a protocol with a degree Q SRS, and denote by $f_{i,j}$ the corresponding polynomial for the j 'th element of srs_i .

Denote by a, b the vectors of \mathbb{F} -elements whose encodings in $\mathbb{G}_1, \mathbb{G}_2$ an algebraic adversary \mathcal{A} outputs during a protocol execution; e.g., the j 'th \mathbb{G}_1 element output by \mathcal{A} is $[a_j]_1$.

By a “real pairing check” we mean a check of the form

$$(a \cdot T_1) \cdot (T_2 \cdot b) = 0$$

for some matrices T_1, T_2 over \mathbb{F} . Note that such a check can indeed be done efficiently given the encoded elements and the pairing function $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_t$.

Given such a “real pairing check”, and the adversary \mathcal{A} and protocol execution during which the elements were output, define the corresponding “ideal check” as follows. Since \mathcal{A} is algebraic when he outputs $[a_j]_i$ he also outputs a vector v such that, from linearity, $a_j = \sum v_\ell f_{i,\ell}(x) = R_{i,j}(x)$ for $R_{i,j}(X) := \sum v_\ell f_{i,\ell}(X)$. Denote, for $i \in \{1, 2\}$ the vector of polynomials $R_i = (R_{i,j})_j$. The corresponding ideal check, checks as a polynomial identity whether

$$(R_1 \cdot T_1) \cdot (T_2 \cdot R_2) \equiv 0$$

The following lemma is inspired by [FKL18]’s analysis of [Gro16], and tells us that for soundness analysis against algebraic adversaries it suffices to look at ideal checks. Before stating the lemma we define the Q -DLOG assumption similarly to [FKL18].

Definition 2.2. Fix integer Q . The Q -DLOG assumption for $(\mathbb{G}_1, \mathbb{G}_2)$ states that given

$$[1]_1, [x]_1, \dots, [x^Q]_1, [1]_2, [x]_2, \dots, [x^Q]_2$$

for uniformly chosen $x \in \mathbb{F}$, the probability of an efficient \mathcal{A} outputting x is $\text{negl}(\lambda)$.

Lemma 2.3. Assume the Q -DLOG for $(\mathbb{G}_1, \mathbb{G}_2)$. Given an algebraic adversary \mathcal{A} participating in a protocol with a degree Q SRS, the probability of any real pairing check passing is larger by at most an additive $\text{negl}(\lambda)$ factor than the probability the corresponding ideal check holds.

See [GWC19] for the proof.

The log-derivative method We crucially use the following lemma from [Hab22].

Lemma 2.4. Given $f \in \mathbb{F}^n$, and $t \in \mathbb{F}^N$, we have $f \subset t$ as sets if and only if for some $m \in \mathbb{F}^N$ the following identity of rational functions holds

$$\sum_{i \in [n]} \frac{1}{X + f_i} = \sum_{i \in [N]} \frac{m_i}{X + t_i}.$$

3 Cached quotients

Notation: In this section and the next we use the following conventions. $\mathbb{V} \subset \mathbb{F}$ denotes a mutliplicative subgroup of order N which is a power of two. We denote by \mathbf{g} a generator

of \mathbb{V} . Hence, $\mathbb{V} = \{\mathbf{g}, \mathbf{g}^2, \dots, \mathbf{g}^N = 1\}$. Given $P \in \mathbb{F}[X]$ and integer $i \in [N]$, we denote $P_i := P(\mathbf{g}^i)$. For $i \in [N]$, we denote by $L_i \in \mathbb{F}_{< N}[X]$ the i 'th Lagrange polynomial of \mathbb{V} . Thus, $(L_i)_i = 1$ and $(L_i)_j = 0$ for $i \neq j \in [N]$.

For a polynomial $A(X) \in \mathbb{F}_{< N}[X]$, we say it is *n-sparse* if $A_i \neq 0$ for at most n values $i \in [N]$. The *sparse representation* of such A consists of the (at most) n pairs (i, A_i) such that $A_i \neq 0$. We denote $\text{supp}(A) := \{i \in [N] | A_i \neq 0\}$.

The main result of this section is a method to compute a commitment to a quotient polynomial - derived from a product with a preprocessed polynomial; in a number of operations depending only on the sparsity of the other polynomial in the product.

The result crucially relies on the following lemma based on a result of Feist and Khovratovich[FK].

Lemma 3.1. *Fix $T \in \mathbb{F}_{< N}[X]$, and a subgroup $\mathbb{V} \subset \mathbb{F}$ of size N . There is an algorithm that given the \mathbb{G}_1 elements $\{[x^i]_1\}_{i \in \{0, \dots, N-1\}}$ computes for $i \in [N]$, the elements $q_i := [Q_i(x)]_1$ where $Q_i(x) \in \mathbb{F}[X]$ is such that*

$$L_i(X) \cdot T(X) = T_i \cdot L_i(X) + Z_{\mathbb{V}}(X) \cdot Q_i(X)$$

in $O(N \cdot \log N)$ \mathbb{G}_1 operations.

Proof. Recall the definition of the Lagrange polynomial

$$L_i(X) = \frac{Z_{\mathbb{V}}(X)}{Z'_{\mathbb{V}}(\mathbf{g}^i)(X - \mathbf{g}^i)}.$$

Substituting this definition, we can write the quotient $Q_i(X)$ as

$$Q_i(X) = \frac{T(X) - T_i}{Z'_{\mathbb{V}}(\mathbf{g}^i)(X - \mathbf{g}^i)} = Z'_{\mathbb{V}}(\mathbf{g}^i)^{-1} K_i(X),$$

for $K_i(X) := \frac{T(X) - T_i}{X - \mathbf{g}^i}$. Note that the values $\{[K_i(X)]_1\}_{i \in [N]}$ are exactly the KZG opening proofs of $T(X)$ at the elements of \mathbb{V} . Thus, the algorithm of Feist and Khovratovich [FK, Tom] can be used to compute commitments to all the proofs $[K_i(X)]_1$ in $O(N \log N)$ \mathbb{G}_1 -operations. This works by writing the vector of $[K_i(X)]_1$ as a the product of a matrix with the vector of $[X^i]_1$. This matrix is a DFT matrix times a Toeplitz matrix, both of which have algorithms for evaluating matrix vector products in $O(N \log N)$ operations. Thus, all the KZG proofs can be computed in $O(N \log N)$ field operations and operations in \mathbb{G}_1 .

Finally, the algorithm just needs to scale each $[K_i(X)]_1$ by $Z'_{\mathbb{V}}(\mathbf{g}^i)$ to compute $[Q_i(X)]_1$. Conveniently, these values admit a very simple description when $Z_{\mathbb{V}}(X) = X^N - 1$ is a group of roots of unity.

$$Z'_{\mathbb{V}}(X)^{-1} = (NX^{N-1})^{-1} \equiv X/N \bmod Z_{\mathbb{V}}(X)$$

In total, the prover computes the coefficients of $T(X)$ in $O(N \log N)$ field operations, computes the KZG proofs for $T(\mathbf{g}^i) = t_i$ in $O(N \log N)$ group operations, and then scales these proofs by \mathbf{g}^i/n in $O(N)$ group operations. In total, this takes $O(N \log N)$ field and group operations in \mathbb{G}_1 . \square

Theorem 3.2. Fix integer parameters $0 \leq n \leq N$ such that n, N are powers of two. Fix $T \in \mathbb{F}_{<N}[X]$, and a subgroup $\mathbb{V} \subset \mathbb{F}$ of size N . Let $\text{srs} = \{[x^i]_1\}_{i \in [0, \dots, N-1]}$ for some $x \in \mathbb{F}$. There is an algorithm \mathcal{A} that after a preprocessing step of $O(N \log N)$ \mathbb{F} - and \mathbb{G}_1 -operations starting with srs does the following.

Given input $A(X) \in \mathbb{F}_{<N}[X]$ that is n -sparse and given in sparse representation, \mathcal{A} computes in $O(n)$ \mathbb{F} -operations and n \mathbb{G}_1 -operations each of the elements $\text{cm}_1 = [Q(x)]_1, \text{cm}_2 = [R(x)]_1$ for $Q(X), R(X) \in \mathbb{F}_{<N}[X]$ such that

$$A(X) \cdot T(X) = Q(X) \cdot Z_{\mathbb{V}}(X) + R(X).$$

Proof. The preprocessing step consists of computing the quotient commitments $[Q_i(X)]_1$ in $O(N \log N)$ operations, as described in Lemma 3.1. As stated in the lemma, for each $i \in [N]$ we have

$$L_i(X) \cdot T(X) = T_i \cdot L_i(X) + Z_{\mathbb{V}}(X) \cdot Q_i(X).$$

By assumption, the polynomial $A(X)$ can be written as a linear combination of at most n summands in the Lagrange basis of \mathbb{V} .

$$A(X) = \sum_{i \in \text{supp}(A)} A_i \cdot L_i(X)$$

Substituting this into the product with $T(X)$, and substituting each of the products $L_i(X)T(X)$ with the appropriate cached quotient $Q_i(X)$ we find

$$\begin{aligned} A(X)T(X) &= \sum_{i \in \text{supp}(A)} A_i \cdot L_i(X)T(X) = \sum_{i \in \text{supp}(A)} A_i \cdot T_i L_i(X) + A_i \cdot Z_{\mathbb{V}}(X) Q_i(X) \\ &= \sum_{i \in \text{supp}(A)} A_i \cdot T_i L_i(X) + Z_{\mathbb{V}}(X) \cdot \sum_{i \in \text{supp}(A)} A_i \cdot Q_i(X). \end{aligned}$$

Observing that the terms of the first sum are all of degree smaller than N , we get that

$$\begin{aligned} Q(X) &= \sum_{i \in \text{supp}(A)} A_i \cdot Q_i(X) \\ R(X) &= \sum_{i \in \text{supp}(A)} A_i T_i \cdot L_i(X) \end{aligned}$$

Hence, commitments to both the quotient $Q(X)$ and remainder $R(X)$ can be computed in at most n group operations as

$$\begin{aligned} [Q(x)]_1 &= \sum_{i \in \text{supp}(A)} A_i \cdot [Q_i(x)]_1 \\ [R(x)]_1 &= \sum_{i \in \text{supp}(A)} A_i T_i \cdot [L_i(x)]_1 \end{aligned}$$

□

4 CQ - our main protocol

Before describing our protocol, we give a definition of a lookup protocol secure against algebraic adversaries.

Definition 4.1. A lookup protocol is a pair $\mathcal{P} = (\text{gen}, \text{IsInTable})$ such that

- $\text{gen}(N, \mathbf{t})$ is a randomized algorithm receiving as input parameters integer N and $\mathbf{t} \in \mathbb{F}^N$. Given these inputs gen outputs a string srs of \mathbb{G}_1 and \mathbb{G}_2 elements.
- $\text{IsInTable}(\mathbf{cm}, \mathbf{t}, \text{srs}, \mathbb{H}; f)$ is an interactive public coin protocol between \mathbf{P} and \mathbf{V} where \mathbf{P} has private input $f \in \mathbb{F}_{<n}[X]$, and both parties have access to \mathbf{t}, \mathbf{cm} and $\text{srs} = \text{gen}(N, \mathbf{t})$; such that
 - **Completeness:** If $\mathbf{cm} = [f(x)]_1$ and $f|_{\mathbb{H}} \subset \mathbf{t}$ then \mathbf{V} outputs acc with probability one.
 - **Knowledge soundness in the algebraic group model:** The probability of any efficient algebraic \mathcal{A} to win the following game is $\text{negl}(\lambda)$.
 1. \mathcal{A} chooses integer parameters N, n and a table $\mathbf{t} \in \mathbb{F}^N$.
 2. We compute $\text{srs} = \text{gen}(\mathbf{t}, N)$.
 3. \mathcal{A} sends a message \mathbf{cm} and $f \in \mathbb{F}_{<d}[X]$ such that $\mathbf{cm} = [f(x)]_1$ where d is such that all \mathbb{G}_1 elements in srs are linear combinations of $\{[x^i]_1\}_{i \in \{0, \dots, d-1\}}$.
 4. \mathcal{A} and \mathbf{V} engage in the protocol $\text{IsInTable}(\mathbf{t}, \mathbf{cm}, \text{srs}, \mathbb{H})$, where $\mathbb{H} \subset \mathbb{F}$ is a subgroup of order n , with \mathcal{A} taking the role of \mathbf{P} .
 5. \mathcal{A} wins if
 - * \mathbf{V} outputs acc , and
 - * $f|_{\mathbb{H}} \not\subset \mathbf{t}$.

We say a lookup protocol is homomorphic if for any fixed parameter N and fixed randomness, $\text{gen}(\mathbf{t}, N)$ can be written as $(\text{gen}_1, \text{gen}_2(\mathbf{t}))$ such that gen_1 is fixed, and gen_2 is an \mathbb{F} -linear function of \mathbf{t} .

4.1 The CQ protocol

$\text{gen}(N, \mathbf{t})$:

1. Choose random $x \in \mathbb{F}$ compute and output $\{[x^i]_1\}_{i \in \{0, \dots, N-1\}}, \{[x^i]_2\}_{i \in \{0, \dots, N\}}$.
2. Compute and output $[Z_{\mathbf{V}}(x)]_2$.
3. Compute $T(X) = \sum_{i \in [N]} \mathbf{t}_i L_i(X)$. Compute and output $[T(x)]_2$.
4. For $i \in [N]$, compute and output:

(a) $q_i = [Q_i(x)]_1$ such that

$$L_i(X) \cdot T(X) = t_i \cdot L_i(X) + Z_{\mathbb{V}}(X) \cdot Q_i(X).$$

(b) $[L_i(x)]_1$.

(c) $\left[\frac{L_i(x) - L_i(0)}{x} \right]_1$.

Before describing **IsInTable**, we explain an optimization we use in Step 6 of Round 2. Since we know in advance we are going to open B at zero, it is more efficient to commit to the *the opening proof polynomial* $B_0(X) := \frac{B(X) - B(0)}{X}$ of B at 0 instead of committing to B . To evaluate B , \mathbf{V} can use the relation $B(X) = B_0(X) \cdot X + b_0$.

We note that it's possible to make a similar optimization for A to further reduce proof size and prover time. However, this entails an additional verifier pairing for the check in Step 11 of Round 2.

IsInTable(cm, t, srs, H; f):

Round 1: Committing to the multiplicities vector

1. \mathbf{P} computes the polynomial $m \in \mathbb{F}_{<N}[X]$ defined by setting m_i , for each $i \in [N]$, to the number of times t_i appears in $f|_{\mathbb{H}}$.
2. \mathbf{P} sends $\mathbf{m} := [m(x)]_1$.

Round 2: Interpolating the rational identity at a random β ; checking correctness of A 's values + degree check for B using pairings

1. \mathbf{V} chooses and sends random $\beta \in \mathbb{F}$.
2. \mathbf{P} computes $A \in \mathbb{F}_{<N}[X]$ such that for $i \in [N]$, $A_i = m_i / (t_i + \beta)$.
3. \mathbf{P} computes and sends $\mathbf{a} := [A(x)]_1$.
4. \mathbf{P} computes and sends $\mathbf{q}_a := [Q_A(x)]_1$ where $Q_A \in \mathbb{F}_{<N}[X]$ is such that

$$A(X)(T(X) + \beta) - m(X) = Q_A(X) \cdot Z_{\mathbb{V}}(X)$$

5. \mathbf{P} computes $B(X) \in \mathbb{F}_{<n}[X]$ such that for $i \in [n]$, $B_i = 1 / (f_i + \beta)$.
6. \mathbf{P} computes $B_0(X) \in \mathbb{F}_{<n-1}[X]$ defined as $B_0(X) := \frac{B(X) - B(0)}{X}$.
7. \mathbf{P} computes and sends $\mathbf{b}_0 := [B_0(x)]_1$.

8. **P** computes $Q_B(X)$ such that

$$B(X)(f(x) + \beta) - 1 = Q_B(X) \cdot Z_{\mathbb{H}}(X).$$

9. **P** computes and sends $\mathbf{q}_b := [Q_B(x)]_1$.

10. **P** computes and sends $\mathbf{p} = [P(x)]_1$ where

$$P(X) := B_0(X) \cdot X^{N-(n+1)}.$$

11. **V** checks that A encodes the correct values:

$$e(\mathbf{a}, [T(x)]_2) = e(\mathbf{q}_a, [Z_{\mathbb{V}}(x)]_2) \cdot e(\mathbf{m} - \beta \cdot \mathbf{a}, [1]_2)$$

12. **V** checks that B_0 has the appropriate degree:

$$e(\mathbf{b}_0, [x^{N-n-1}]_2) = e(\mathbf{p}, [1]_2).$$

Round 3: Checking correctness of B at random $\gamma \in \mathbb{F}$

1. **V** sends random $\gamma, \eta \in \mathbb{F}$.

2. **P** sends $b_{0,\gamma} := B_0(\gamma), f_\gamma := f(\gamma)$.

3. **P** computes and sends the value $a_0 := A(0)$.

4. **V** sets $b_0 := (N \cdot a_0)/n$.

5. As part of checking the correctness of B , **V** computes $Z_{\mathbb{H}}(\gamma) = \gamma^n - 1$, $b_\gamma := b_{0,\gamma} \cdot \gamma + b_0$ and

$$Q_{b,\gamma} := \frac{b_\gamma \cdot (f_\gamma + \beta) - 1}{Z_{\mathbb{H}}(\gamma)}.$$

6. To perform a batched KZG check for the correctness of the values $b_{0,\gamma}, f_\gamma, Q_{b,\gamma}$

- (a) **V** sends random $\eta \in \mathbb{F}$. **P** and **V** separately compute

$$v := b_{0,\gamma} + \eta \cdot f_\gamma + \eta^2 \cdot Q_{b,\gamma}.$$

- (b) **P** computes $\pi_\gamma := [h(x)]_1$ for

$$h(X) := \frac{B_0(X) + \eta \cdot f(X) + \eta^2 \cdot Q_B(X) - v}{X - \gamma}$$

- (c) **V** computes

$$\mathbf{c} := \mathbf{b}_0 + \eta \cdot \mathbf{cm} + \eta^2 \cdot \mathbf{q}_b$$

and checks that

$$e(\mathbf{c} - [v]_1 + \gamma \cdot \pi_\gamma, [1]_2) = e(\pi_\gamma, [x]_2).$$

7. To perform a KZG check for the correctness of a_0

(a) \mathbf{P} computes and sends $\mathbf{a}_0 := [A_0(x)]_1$ for

$$A_0(X) := \frac{A(X) - a_0}{X}$$

(b) \mathbf{V} checks that

$$e(\mathbf{a} - [a_0]_1, [1]_2) = e(\mathbf{a}_0, [x]_2).$$

Note that although the above description contains nine pairings, we can reduce to five pairings via the standard technique of randomly batching pairings that share the same \mathbb{G}_2 argument. It is easy to check that \mathbf{q} is homomorphic according to Definition 4.1.

The main things to address are the efficiency of the `gen` algorithm used for preprocessing, the efficiency of \mathbf{P} in `IsInTable`, and the knowledge soundness of `IsInTable`.

Runtime of gen: We claim that `gen` requires $O(N \log N)$ \mathbb{G}_1 - and \mathbb{F} -operations and $O(N)$ \mathbb{G}_2 -operations. The claim regarding the \mathbb{G}_2 operations is obvious. The elements $\{q_i\}$ can be computed in $O(N \log N)$ operations according to Lemma 3.1. The elements $\{[L_i(x)]_1\}$ can be computed in $O(N \log N)$ via FFT as explained in Section 3.3 of [BGG17]. Given the element $[L_i(x)]_1$, the element $\left[\frac{L_i(x) - L_i(0)}{x}\right]_1$ can be computed as

$$\left[\frac{L_i(x) - L_i(0)}{x}\right]_1 = \mathbf{g}^{-i} \cdot [L_i(x)]_1 - (1/N) \cdot [x^{N-1}]_1.$$

Runtime of \mathbf{P} : Note first that the computation of \mathbf{m}, \mathbf{a} can be done in n \mathbb{G}_1 -operations as $m(X)$ and $A(X)$ are n -sparse. The main things to address are the computation of \mathbf{q}_a ; that can be done in n \mathbb{G}_1 -operations given `srs` according to Theorem 3.2. The only step requiring $O(n \log n)$ \mathbb{F} -operations is the computation the quotient $Q_B(X)$ which involves FFT on \mathbb{H} . We also note that the commitment $\mathbf{a}_0 = \left[\frac{A(x) - A(0)}{x}\right]_1$ can be computed in n \mathbb{G}_1 -operations as the linear combination

$$\left[\frac{A(x) - A(0)}{x}\right]_1 = \sum_{i \in \text{supp}(A)} A_i \cdot \left[\frac{L_i(x) - L_i(0)}{x}\right]_1.$$

Knowledge soundness proof: Let \mathcal{A} be an efficient algebraic adversary participating in the Knowledge Soundness game from Definition 4.1. We show its probability of winning the game is $\text{negl}(\lambda)$. Let $f \in \mathbb{F}_{<N}[X]$ be the polynomial sent by \mathcal{A} in the third step of the game such that $\mathbf{cm} = [f(x)]_1$. As \mathcal{A} is algebraic, when sending the commitments $\mathbf{m}, \mathbf{a}, \mathbf{b}_0, \mathbf{p}, \mathbf{q}_a, \mathbf{q}_b, \pi_\gamma, \mathbf{a}_0$ during protocol execution it also sends polynomials $m(X), A(X), B_0(X), P(X), Q_A(X), Q_B(X), h(X), A_0(X) \in \mathbb{F}_{<N}[X]$ such that the former are their corresponding commitments. Let E be the event that \mathbf{V} outputs `acc`. Note that the event that \mathcal{A} wins the knowledge soundness game is contained in E . E implies

all pairing checks have passed. Let $A \subset E$ be the event that one of the corresponding ideal pairing checks as defined in Section 2.2 didn't pass. According to Lemma 2.3, $\Pr(A = \text{negl}(\lambda))$. Given that A didn't occur, we have

- From Round 2, Step 11

$$A(X)(T(X) + \beta) - M(X) = Q_A(X) \cdot Z_{\mathbb{V}}(X)$$

Which means that for all $i \in [N]$,

$$A_i = \frac{M_i}{T_i + \beta}$$

- From Round 2, Step 12

$$X^{N-n-1}B_0(X) = P(X),$$

which implies that $\deg(B_0) < n - 1$. Note also that we know $\deg(A) < N$ simply from $[x^{N-1}]_1$ being the highest \mathbb{G}_1 power in srs ²

- Moving to Round 3, from the checks of steps 6c and 7b, e.w.p. $n/|\mathbb{F}|$ over $\eta, \zeta \in \mathbb{F}$ (see e.g. Section 3 of [GWC19] for an explanation of batched KZG [KZG10]), we have $b_{0,\gamma} = B_0(\gamma), Q_{b,\gamma} = Q_B(\gamma), f_\gamma = f(\gamma), a_0 = A(0)$.
- Define $B(X) := B_0(X) \cdot X + b_0$ for b_0 set as in step 4. Note that we have $\deg(B) < n$. Let ω by a generator of \mathbb{H} .
- By how $b_\gamma, Q_{b,\gamma}$ are set in step 5, the above implies that e.w.p. $(N+n)/|\mathbb{F}|$ over γ

$$B(X) \cdot (f(X) + \beta) = 1 + Q_B(X)Z_{\mathbb{H}}(X),$$

which implies for all $i \in [n]$ that $B(\omega^i) = \frac{1}{f(\omega^i) + \beta}$.

- We now have using Lemma 2.1 that

$$\begin{aligned} N \cdot a_0 &= \sum_{i \in [N]} A_i = \sum_{i \in [N]} \frac{m_i}{T_i + \beta}, \\ n \cdot b_0 &= \sum_{i \in [n]} B(\omega^i) = \sum_{i \in [n]} \frac{1}{f(\omega^i) + \beta}. \end{aligned}$$

Recall that b_0 was set such that $N \cdot a_0 = n \cdot b_0$. Thus e.w.p. $(n \cdot N)/|\mathbb{F}|$ over $\beta \in \mathbb{F}$, we have that

$$\sum_{i \in [N]} \frac{m_i}{T_i + X} = \sum_{i \in [n]} \frac{1}{f(\omega^i) + X},$$

which implies $f|_{\mathbb{H}} \subset \mathbf{t}$ by Lemma 2.4.

In summary, we have shown the event that \mathbf{V} outputs `acc` while $f|_{\mathbb{H}} \not\subset \mathbf{t}$ is contained in a constant number of events with probability $\text{negl}(\lambda)$; and so \mathbf{tq} satisfies the knowledge soundness property.

²An important point is that when using an SRS built with higher degrees in \mathbb{G}_1 , A must also be degree checked via an additional pairing.

Acknowledgements

We thank Aurel Nicolas for corrections.

References

- [BCG⁺18] J. Bootle, A. Cerulli, J. Groth, S. K. Jakobsen, and M. Maller. Arya: Nearly linear-time zero-knowledge proofs for correct program execution. In Thomas Peyrin and Steven D. Galbraith, editors, *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part I*, volume 11272 of *Lecture Notes in Computer Science*, pages 595–626. Springer, 2018.
- [BCMS20] B. Bünz, A. Chiesa, P. Mishra, and N. Spooner. Recursive proof composition from accumulation schemes. In *Theory of Cryptography - 18th International Conference, TCC 2020, Durham, NC, USA, November 16-19, 2020, Proceedings, Part II*, volume 12551 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2020.
- [BCR⁺19] E. Ben-Sasson, A. Chiesa, M. Riabzev, N. Spooner, M. Virza, and N. P. Ward. Aurora: Transparent succinct arguments for R1CS. In *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part I*, pages 103–128, 2019.
- [BGG17] S. Bowe, A. Gabizon, and M. D. Green. A multi-party protocol for constructing the public parameters of the pinocchio zk-snark. *IACR Cryptology ePrint Archive*, 2017:602, 2017.
- [BGM17] S. Bowe, A. Gabizon, and I. Miers. Scalable multi-party computation for zk-snark parameters in the random beacon model. *Cryptology ePrint Archive*, Report 2017/1050, 2017. <https://eprint.iacr.org/2017/1050>.
- [Eag22] Liam Eagen. Bulletproofs++. *IACR Cryptol. ePrint Arch.*, page 510, 2022.
- [FK] D. Feist and D. Khovratovich. Fast amortized kate proofs.
- [FKL18] G. Fuchsbauer, E. Kiltz, and J. Loss. The algebraic group model and its applications. In *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part II*, pages 33–62, 2018.
- [GK22] A. Gabizon and D. Khovratovich. flookup: Fractional decomposition-based lookups in quasi-linear time independent of table size. *IACR Cryptol. ePrint Arch.*, page 1447, 2022.

- [Gro16] J. Groth. On the size of pairing-based non-interactive arguments. In *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II*, pages 305–326, 2016.
- [GW20] A. Gabizon and Z. J. Williamson. plookup: A simplified polynomial protocol for lookup tables. *IACR Cryptol. ePrint Arch.*, page 315, 2020.
- [GWC19] A. Gabizon, Z. J. Williamson, and O. Ciobotaru. PLONK: permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *IACR Cryptology ePrint Archive*, 2019:953, 2019.
- [Hab22] U. Haböck. Multivariate lookups based on logarithmic derivatives. *IACR Cryptol. ePrint Arch.*, page 1530, 2022.
- [KZG10] A. Kate, G. M. Zaverucha, and I. Goldberg. Constant-size commitments to polynomials and their applications. pages 177–194, 2010.
- [PK22] J. Posen and A. A. Kattis. Caulk+: Table-independent lookup arguments. 2022.
- [Tom] A. Tomescu. Feist-khovratovich technique for computing kzg proofs fast.
- [ZBK⁺22] A. Zapico, V. Buterin, D. Khovratovich, M. Maller, A. Nitulescu, and M. Simkin. Caulk: Lookup arguments in sublinear time. *IACR Cryptol. ePrint Arch.*, page 621, 2022.
- [ZGK⁺22] A. Zapico, A. Gabizon, D. Khovratovich, M. Maller, and C. Ràfols. Baloo: Nearly optimal lookup arguments. *IACR Cryptol. ePrint Arch.*, page 1565, 2022.

HyperPlonk

Dan Boneh, Benedikt Bünz, Binyi Chen, Zhenfei Zhang

Espresso Systems



HyperPlonk: Plonk with Linear-Time Prover and High-Degree Custom Gates

Binyi Chen
Espresso Systems

Benedikt Bünz
Stanford University,
Espresso Systems

Dan Boneh
Stanford University

Zhenfei Zhang
Espresso Systems

October 10, 2022

Abstract

Plonk is a widely used succinct non-interactive proof system that uses univariate polynomial commitments. Plonk is quite flexible: it supports circuits with low-degree “custom” gates as well as circuits with lookup gates (a lookup gate ensures that its input is contained in a predefined table). For large circuits, the bottleneck in generating a Plonk proof is the need for computing a large FFT.

We present HyperPlonk, an adaptation of Plonk to the boolean hypercube, using multilinear polynomial commitments. HyperPlonk retains the flexibility of Plonk but provides several additional benefits. First, it avoids the need for an FFT during proof generation. Second, and more importantly, it supports custom gates of much higher degree than Plonk without harming the running time of the prover. Both of these can dramatically speed up the prover’s running time. Since HyperPlonk relies on multilinear polynomial commitments, we revisit two elegant constructions: one from Orion and one from Virgo. We show how to reduce the Orion opening proof size to less than 10kb (an almost factor 1000 improvement) and show how to make the Virgo FRI-based opening proof simpler and shorter.

Contents

1	Introduction	3
1.1	Technical overview	6
1.2	Additional related work	10
2	Preliminaries	10
2.1	Proofs and arguments of knowledge.	11
2.2	Multilinear polynomial commitments.	13
2.3	PIOP Compilation	15
3	A toolbox for multivariate polynomials	16
3.1	SumCheck PIOP for high degree polynomials	16
3.2	ZeroCheck PIOP	19
3.3	ProductCheck PIOP	20
3.4	Multiset Check PIOP	21
3.5	Permutation PIOP	23
3.6	Lookup PIOP	24
3.7	Batch openings	27
3.7.1	A more efficient batching scheme in a special setting	29
4	HyperPlonk: Plonk on the boolean hypercube	31
4.1	Constraint systems	31
4.2	The PolyIOP protocol	32
5	HyperPlonk+: HyperPlonk with Lookup Gates	34
5.1	Constraint systems	34
5.2	The PolyIOP protocol	35
6	Instantiation and evaluation	37
6.1	Implementation	37
6.2	Evaluation	38
6.3	MultiThreading Performance	39
6.4	High degree gates	40
6.5	Comparisons	40
7	Orion+: a linear-time multilinear PCS with constant proof size	41
A	Zero Knowledge PIOPs and zk-SNARKs	57
A.1	Definition	57
A.2	Polynomial Masking	57
A.3	Zero Knowledge SumCheck	58
A.4	Zero Knowledge Compilation for SumCheck-based PIOPs	59
A.5	zk-SNARKs from PIOPs	61
B	The FRI-based multilinear polynomial commitment	61

1 Introduction

Proof systems [47, 6] have a long and rich history in cryptography and complexity theory. In recent years, the efficiency of proof systems has dramatically improved and this has enabled a multitude of new real-world applications that were not previously possible. In this paper, we focus on succinct non-interactive arguments of knowledge, also called SNARKs [16]. Here, succinct refers to the fact that the proof is short and verification time is fast, as explained below. Recent years have seen tremendous progress in improving the efficiency of the prover [73, 60, 77, 2, 12, 82, 33, 25, 42, 66, 22, 48, 78].

Let us briefly review what a (preprocessing) SNARK is. We give a precise definition in Section 2. Fix a finite field \mathbb{F} , and consider the relation $\mathcal{R}(\mathcal{C}, \mathbf{x}, \mathbf{w})$ that is true whenever $\mathbf{x} \in \mathbb{F}^n$, $\mathbf{w} \in \mathbb{F}^m$, and $\mathcal{C}(\mathbf{x}, \mathbf{w}) = 0$, where \mathcal{C} is the description of an arithmetic circuit over \mathbb{F} that takes $n + m$ inputs. A SNARK enables a prover \mathcal{P} to non-interactively and succinctly convince a verifier \mathcal{V} that \mathcal{P} knows a witness $\mathbf{w} \in \mathbb{F}^m$ such that $\mathcal{R}(\mathcal{C}, \mathbf{x}, \mathbf{w})$ holds, for some public circuit \mathcal{C} and $\mathbf{x} \in \mathbb{F}^n$.

In more detail, a SNARK is a tuple of four algorithms $(\text{Setup}, \mathcal{I}, \mathcal{P}, \mathcal{V})$, where $\text{Setup}(1^\lambda)$ is a randomized algorithm that outputs parameters \mathbf{gp} , and $\mathcal{I}(\mathbf{gp}, \mathcal{C})$ is a deterministic algorithm that pre-processes the circuit \mathcal{C} and outputs prover parameters \mathbf{pp} and verifier parameters \mathbf{vp} . The prover $\mathcal{P}(\mathbf{pp}, \mathbf{x}, \mathbf{w})$ is a randomized algorithm that outputs a proof π , and the verifier $\mathcal{V}(\mathbf{vp}, \mathbf{x}, \pi)$ is a deterministic algorithm that outputs 0 or 1. The SNARK must be *complete*, *knowledge sound*, and *succinct*, as defined in Section 2. Here *succinct* means that if \mathcal{C} contains s gates, and $\mathbf{x} \in \mathbb{F}^n$, then the size of the proof should be $O_\lambda(\log s)$ and the verifier's running time should be $\tilde{O}_\lambda(n + \log s)$. A SNARK is often set in the random oracle model where all four algorithms can query the oracle. If the Setup algorithm is randomized, then we say that the SNARK requires a *trusted setup*; otherwise, the SNARK is said to be *transparent* because Setup only has access to public randomness via the random oracle. Optionally, we might want the SNARK to be zero-knowledge, in which case it is called a zkSNARK.

Modern SNARKs are constructed by compiling an information-theoretic object called an Interactive Oracle Proof (IOP) [13] to a SNARK using a suitable cryptographic commitment scheme. There are several examples of this paradigm. Some SNARKs use a univariate polynomial commitment scheme to compile a Polynomial-IOP to a SNARK. Examples include Sonic [60], Marlin [33], and Plonk [42]. Other SNARKs use a multivariate linear (multilinear) commitment scheme to compile a multilinear-IOP to a SNARK. Examples include Hyrax [73], Libra [77], Spartan [66], Quarks [67], and Gemini [22]. Yet other SNARKs use a vector commitment scheme (such as a Merkle tree) to compile a vector-IOP to a SNARK. The STARK system [10] is the prime example in this category, but other examples include Aurora [12], Virgo [82], Brakedown [48], and Orion [78]. While STARKs are post-quantum secure, require no trusted setup, and have an efficient prover, they generate a relatively long proof (tens of kilobytes in practice). The paradigm of compiling an IOP to a SNARK using a suitable commitment scheme lets us build *universal* SNARKs where a single trusted setup can support many circuits. In earlier SNARKs, such as [50, 45, 18], every circuit required a new trusted setup.

The Plonk system. Among the IOP-based SNARKs that use a Polynomial-IOP, the Plonk system [42] has emerged as one of the most widely adopted in industry. This is because Plonk

proofs are very short (about 400 bytes in practice) and fast to verify. Moreover, Plonk supports custom gates, as we will see in a minute. An extension of Plonk, called PlonKup [64], further extends Plonk to incorporate lookup gates using the Plookup IOP of [42].

One difficulty with Plonk, compared to some other schemes, is the prover’s complexity. For a circuit \mathcal{C} with s arithmetic gates, the Plonk prover runs in time $O_\lambda(s \log s)$. The primary bottlenecks come from the fact that the prover must commit to and later open several degree $O(s)$ polynomials. When using the KZG polynomial commitment scheme [54], the prover must (i) compute a multi-exponentiation of size $O(s)$ in a pairing-friendly group where discrete log is hard, and (ii) compute several FFTs and inverse-FFTs of dimension $O(s)$. When using a FRI-based polynomial commitment scheme [9, 55, 82], the prover computes an $O(cs)$ -sized FFT and $O(cs)$ hashes, where $1/c$ is the rate of a certain Reed-Solomon code. The performance further degrades for circuits that contain *high-degree* custom gates, as some FFTs and multi-exponentiations have size proportional to the degree of the custom gates.

In practice, when the circuit size s is bigger than 2^{20} , the FFTs become a significant part of the running time. This is due to the quasi-linear running time of the FFT algorithm, while other parts of the prover scale linearly in s . The reliance on FFT is a direct result of Plonk’s use of *univariate* polynomials. We note that some proof systems eliminate the need for an FFT by moving away from Plonk altogether [66, 22, 48, 78, 39].

Hyperplonk. In this paper, we introduce HyperPlonk, an adaptation of the Plonk IOP and its extensions to operate over the boolean hypercube $B_\mu := \{0, 1\}^\mu$. We present HyperPlonk as a multilinear-IOP, which means that it can be compiled using a suitable multilinear commitment scheme to obtain a SNARK (or a zkSNARK) with an efficient prover.

HyperPlonk inherits the flexibility of Plonk to support circuits with custom gates, but presents several additional advantages. First, by moving to the boolean hypercube we eliminate the need for an FFT during proof generation. We do so by making use of the classic SumCheck protocol [59], and this reduces the prover’s running time from $O_\lambda(s \log s)$ to $O_\lambda(s)$. The efficiency of SumCheck is the reason why many of the existing multilinear SNARKs [73, 77, 66, 67, 22] use the boolean hypercube. Here we show that Plonk can similarly benefit from the SumCheck protocol.

Second, and more importantly, we show that the hypercube lets us incorporate custom gates more efficiently into HyperPlonk. A custom gate is a function $G : \mathbb{F}^\ell \rightarrow \mathbb{F}$, for some ℓ . An arithmetic circuit \mathcal{C} with a custom gate G , denoted $\mathcal{C}[G]$, is a circuit with addition and multiplication gates along with a custom gate G that can appear many times in the circuit. The circuit may contain multiple types of custom gates, but for now, we will restrict to one type to simplify the presentation. These custom gates can greatly reduce the circuit size needed to compute a function, leading to a faster prover. For example, if one needs to implement the S-box in a block cipher, it can be more efficient to implement it as a custom gate.

Custom gates are not free. Let $G : \mathbb{F}^\ell \rightarrow \mathbb{F}$ be a custom gate that computes a multivariate polynomial of total degree d . Let $\mathcal{C}[G]$ be a circuit with a total of s gates. In the Plonk IOP, the circuit $\mathcal{C}[G]$ results in a prover that manipulates univariate polynomials of degree $O(s \cdot d)$. Consequently, when compiling Plonk using KZG [54], the prover needs to do a group multi-exponentiation of size $O(sd)$ as well as FFTs of this dimension. This restricts custom gates in Plonk to gates of low degree.

We show that the prover’s work in HyperPlonk is much lower. Let $G : \mathbb{F}^\ell \rightarrow \mathbb{F}$ be a custom gate that can be evaluated using k arithmetic operations. In HyperPlonk, the bulk of the prover’s work

when processing $\mathcal{C}[G]$ is only $O(sk \log^2 k)$ field operations. Moreover, when using KZG multilinear commitments [63], the total number of group exponentiations is only $O(s + d \log s)$, where d is the total degree of G . This is much lower than Plonk’s $O(sd)$ group exponentiations. It lets us use custom gates of much higher degree in **HyperPlonk**.

Making Plonk and its Plonkup extension work over the hypercube raises interesting challenges, as discussed in Section 1.1. In particular, adapting the Plookup IOP [42], used to implement table lookups, requires changing the protocol to make it work over the hypercube (see Section 3.6). The resulting version of **HyperPlonk** that supports lookup gates is called **HyperPlonk+** and is described in Section 5. There are also subtleties in making **HyperPlonk** zero knowledge. In Appendix A, we describe a general compiler to transform a multilinear-IOP into one that is zero knowledge.

Batch openings and commit-and-prove SNARKs. The prover in **HyperPlonk** needs to open several multilinear polynomials at random points. We present a new sum-check-based batch opening protocol (Section 3.7) that can batch many openings into one, significantly reducing the prover time, proof size, and verifier time. Our protocol takes $O(k \cdot 2^\mu)$ field operations for the prover for batching k polynomials, compared to $O(k^2\mu \cdot 2^\mu)$ for the previously best protocol [71]. Under certain conditions, we also obtain a more efficient batching scheme with complexity $O(2^\mu)$, which yields a very efficient commit-and-prove protocol.

Improved multilinear commitments. Since **HyperPlonk** relies on a multilinear commitment scheme, we revisit two approaches to constructing multilinear commitments and present significant improvements to both.

First, in Section 7 we use our commit-and-prove protocol to improve the **Orion** multilinear commitment scheme [78]. **Orion** is highly efficient: the prover time is strictly linear, taking only $O(2^\mu)$ field operations and hashes for a multilinear polynomial in μ variables (no group exponentiations are used). The proof size is $O(\lambda\mu^2)$ hash and field elements, and the verifier time is proportional to the proof size. In Section 7 we describe **Orion+**, that has the same prover complexity, but has $O(\mu)$ proof size and $O(\mu)$ verifier time, with good constants. In particular, for security parameter $\lambda = 128$ and $\mu = 25$ the proof size with **Orion+** is only about 7 KBs, compared with 5.5 MB with **Orion**, a nearly 1000x improvement. Using **Orion+** in **HyperPlonk** gives a strictly linear time prover.

Second, in Appendix B, we show how to generically transform a univariate polynomial commitment scheme into a multilinear commitment scheme using the tensor-product *univariate* Polynomial-IOP from [22]. This yields a new construction for multilinear commitments from FRI [9] by applying the transformation to the univariate FRI-based commitment scheme from [55]. This approach leads to a more efficient FRI-based multilinear commitment scheme compared to the prior construction in [82], which uses recursive techniques. Using this commitment scheme in **HyperPlonk** gives a quantum-resistant quasilinear-time prover.

Evaluation results. When instantiated with the pairing-based multilinear commitment scheme of [63], the proof size of **HyperPlonk** is $2\mu + 8$ group elements and $4\mu + 33$ field elements¹. Using BLS12-381 as the pairing group, we obtain 6KB proofs for $\mu = 20$ and 7KB proofs for $\mu = 25$. For comparison, Kopis [67] and Gemini [22], which also have linear-time provers, report proofs of

¹The constants depend linearly on the degree of the custom gates. These numbers are for simple degree 2 arithmetic circuits.

Application	$\mathcal{R}_{\text{R1CS}}$	Spartan	$\mathcal{R}_{\text{PLONK+}}$	Jellyfish	HyperPlonk
3-to-1 Rescue Hash	288 [1]	279 ms	144 [69]	20 ms	24 ms
Zexe’s recursive circuit	2^{22} [79]	2.4 min	2^{17} [79]	5.83 s	4.66 s
Rollup of 50 private tx	2^{25}	20 min	2^{20} [69]	52.7 s	34.9 s

Table 1: The prover runtime of Hyperplonk, Spartan [66], and the Jellyfish Plonk implementation, for popular applications. The first column shows the number of R1CS constraints for each application. The third column shows the corresponding number of constraints in HyperPlonk+.

size 39KB and 18KB respectively for $\mu = 20$. In Table 1 and Figure 8 we show that our prototype HyperPlonk implementation outperforms an optimized commercial-strength Plonk system for circuits with more than 2^{14} gates. It also shows the effects of PLONK arithmetization compared to R1CS by comparing the prover runtime for several important applications. Hyperplonk outperforms Spartan [66] for these applications by a factor of over 20. We discuss the evaluation further in Section 6.

1.1 Technical overview

In this section we give a high level overview of how to make Plonk and its extensions work over the hypercube. We begin by describing Plonk in a modular way, breaking it down into a sequence of elementary components shown in Figure 1. In Section 3 we show how to instantiate each component over the hypercube.

Some components of Plonk in Figure 1 rely on the simple linear ordering of the elements of a finite cyclic group induced by the powers of a generator. On the hypercube there is no natural simple ordering, and this causes a problem in the Plookup protocol [42] that is used to implement a lookup gate. To address this we modify the Plookup argument in Section 5 to make it work over the hypercube. We give an overview of our approach below.

A review of Plonk. Let us briefly review the Plonk SNARK. Let $\mathcal{C}[G] : \mathbb{F}^{n+m} \rightarrow \mathbb{F}$ be a circuit with a total of s gates, where each gate has fan-in two and can be one of addition, multiplication, or a custom gate $G : \mathbb{F}^2 \rightarrow \mathbb{F}$. Let $\mathbf{x} \in \mathbb{F}^n$ be a public input to the circuit. Plonk represents the resulting computation as a sequence of $n + s + 1$ triples²:

$$\hat{M} := \left\{ (L_i, R_i, O_i) \in \mathbb{F}^3 \right\}_{i=0, \dots, n+s}. \quad (1)$$

This \hat{M} is a matrix with three columns and $n + s + 1$ rows. The first n rows encode the n public input; the next s rows represent the left and right inputs and the output for each gate; and the final row enforces that the final output of the circuit is zero. We will see how in a minute.

In basic (univariate) Plonk, the prover encodes the cells of \hat{M} using a cyclic subgroup $\Omega \subseteq \mathbb{F}$ of order $3(n + s + 1)$. Specifically, let $\omega \in \Omega$ be a generator. Then the prover interpolates and commits to a polynomial $M \in \mathbb{F}[X]$ such that

$$M(\omega^{3i}) = L_i, \quad M(\omega^{3i+1}) = R_i, \quad M(\omega^{3i+2}) = O_i \quad \text{for } i = 0, \dots, n + s.$$

²A more general Plonkish arithmetization [81] supports wider tuples, but triples are sufficient here.

Now the prover needs to convince the verifier that the committed M encodes a valid computation of the circuit \mathcal{C} . This is the bulk of Plonk system.

Hyperplonk. In HyperPlonk we instead use the boolean hypercube to encode \hat{M} . From now on, suppose that $n + s + 1$ is a power of two, so that $n + s + 1 = 2^\mu$. The prover interpolates and commits to a multilinear polynomial $M \in \mathbb{F}[X^{\mu+2}] = \mathbb{F}[X_1, \dots, X_{\mu+2}]$ such that

$$M(0, 0, \langle i \rangle) = L_i, \quad M(0, 1, \langle i \rangle) = R_i, \quad M(1, 0, \langle i \rangle) = O_i, \quad \text{for } i = 0, \dots, n+s. \quad (2)$$

Here $\langle i \rangle$ is the μ -bit binary representation of i . Note that a multilinear polynomial on $\mu+2$ variables is defined by a vector of $2^{\mu+2} = 4 \times 2^\mu$ coefficients. Hence, it is always possible to find a multilinear polynomial that satisfies the $3 \times 2^\mu$ constraints in (2). Next, the prover needs to convince the verifier that the committed M encodes a valid computation of the circuit \mathcal{C} . To do so, we need to adapt Plonk to work over the hypercube.

Let us start with the pre-processing algorithm $\mathcal{I}(\mathbf{gp}, \mathcal{C})$ that outputs prover and verifier parameters \mathbf{pp} and \mathbf{vp} . The verifier parameters \mathbf{vp} encode the circuit $\mathcal{C}[G]$ as a commitment to four multilinear polynomials (S_1, S_2, S_3, σ) , where $S_1, S_2, S_3 \in \mathbb{F}[X^\mu]$ and $\sigma \in \mathbb{F}[X^{\mu+2}]$. The first three are called *selector polynomials* and σ is called the *wiring polynomial*. We will see how they are defined in a minute. There is one more auxiliary multilinear polynomial $I \in \mathbb{F}[X^\mu]$ that encodes the input $\mathbf{x} \in \mathbb{F}^n$. This polynomial is defined as $I(\langle i \rangle) = \mathbf{x}_i$ for $i = 0, \dots, n-1$, and is zero on the rest of the boolean cube B_μ . The verifier, on its own, computes a commitment to the polynomial I to ensure that the correct input $\mathbf{x} \in \mathbb{F}^n$ is being used in the proof. Computing a commitment to I can be done in time $O_\lambda(n)$, which is within the verifier's time budget.

With this setup, the Plonk prover \mathcal{P} convinces the verifier that the committed M satisfies two polynomial identities:

The gate identity: Let $S_1, S_2, S_3 : \mathbb{F}^\mu \rightarrow \{0, 1\}$ be the three selector polynomials that the pre-processing algorithm $\mathcal{I}(\mathbf{gp}, \mathcal{C})$ committed to in \mathbf{vp} . To prove that all gates were evaluated correctly, the prover convinces the verifier that the following identity holds for all $\mathbf{x} \in B_\mu := \{0, 1\}^\mu$:

$$\begin{aligned} 0 &= S_1(\mathbf{x}) \cdot \left(\underbrace{M(0, 0, \mathbf{x})}_{L_{[\mathbf{x}]}} + \underbrace{M(0, 1, \mathbf{x})}_{R_{[\mathbf{x}]}} \right) + S_2(\mathbf{x}) \cdot \underbrace{M(0, 0, \mathbf{x})}_{L_{[\mathbf{x}]}} \cdot \underbrace{M(0, 1, \mathbf{x})}_{R_{[\mathbf{x}]}} \\ &\quad + S_3(\mathbf{x}) \cdot G \left(\underbrace{M(0, 0, \mathbf{x})}_{L_{[\mathbf{x}]}} , \underbrace{M(0, 1, \mathbf{x})}_{R_{[\mathbf{x}]}} \right) - \underbrace{M(1, 0, \mathbf{x})}_{O_{[\mathbf{x}]}} + I(\mathbf{x}) \end{aligned} \quad (3)$$

where $[\mathbf{x}] = \sum_{i=0}^{\mu-1} \mathbf{x}_i 2^i$ is the integer whose binary representation is $\mathbf{x} \in B_\mu$. For each $i = 0, \dots, n+s$, the selector polynomials S_1, S_2, S_3 are defined to do the “right” thing:

- for an addition gate: $S_1(\langle i \rangle) = 1, S_2(\langle i \rangle) = S_3(\langle i \rangle) = 0$ (so $O_i = L_i + R_i$)
- for a multiplication gate: $S_1(\langle i \rangle) = S_3(\langle i \rangle) = 0, S_2(\langle i \rangle) = 1$ (so $O_i = L_i \cdot R_i$)
- for a G gate: $S_1(\langle i \rangle) = S_2(\langle i \rangle) = 0, S_3(\langle i \rangle) = 1$ (so $O_i = G(L_i, R_i)$)
- when $i < n$ or $i = n+s$: $S_1(\langle i \rangle) = S_2(\langle i \rangle) = S_3(\langle i \rangle) = 0$ (so $O_i = I(\langle i \rangle)$).

The last bullet ensures that O_i is equal to the i -th input for $i = 0, \dots, n-1$, and that the final output of the circuit, O_{n+s} , is equal to zero.

The wiring identity: Every wire in the circuit \mathcal{C} induces an equality constraint on two cells in the matrix \hat{M} . In HyperPlonk, the wiring constraints are captured by a permutation $\hat{\sigma} : B_{\mu+2} \rightarrow B_{\mu+2}$. The prover needs to convince the verifier that

$$M(\mathbf{x}) = M(\hat{\sigma}(\mathbf{x})) \quad \text{for all } \mathbf{x} \in B_{\mu+2} := \{0, 1\}^{\mu+2}. \quad (4)$$

To do so, the pre-processing algorithm $\mathcal{I}(\mathbf{gp}, \mathcal{C})$ commits to a multilinear polynomial $\sigma : \mathbb{F}^{\mu+2} \rightarrow \mathbb{F}$ that satisfies $\sigma(\mathbf{x}) = [\hat{\sigma}(\mathbf{x})]$ for all $\mathbf{x} \in B_{\mu+2}$ (recall that $[\hat{\sigma}(\mathbf{x})]$ is the integer whose binary representation is $\hat{\sigma}(\mathbf{x}) \in B_{\mu+2}$). The prover then convinces the verifier that the following two sets are equal (both sets are subsets of \mathbb{F}^2):

$$\left\{ ([\mathbf{x}], M(\mathbf{x})) \right\}_{\mathbf{x} \in B_{\mu+2}} = \left\{ ([\hat{\sigma}(\mathbf{x})], M(\mathbf{x})) \right\}_{\mathbf{x} \in B_{\mu+2}}. \quad (5)$$

This equality of sets implies that (4) holds.

Proving the gate identity. The prover convinces the verifier that the Gate identity holds by proving that the polynomial defined by the right hand side of (3) is zero for all $\mathbf{x} \in B_\mu$. This is done using a ZeroCheck IOP, defined in Section 3.2. If the custom gate G has total degree d and there are s gates in the circuit, then the total degree of the polynomial in (3) is $(d+1)(s+n+1)$ which is about $(d \cdot s)$. If this were a univariate polynomial, as in Plonk, then a ZeroCheck would require a multi-exponentiation of dimension $(d \cdot s)$ and an FFT of the same dimension. When the polynomial is defined over the hypercube, the ZeroCheck is implemented using the SumCheck protocol in Section 3.1, which requires no FFTs. In that section we describe two optimizations to the SumCheck protocol for the settings where the multivariate polynomial has a high degree d in every variable:

- First, in every round of SumCheck the prover sends a polynomial commitment to a univariate polynomial of degree d , instead of sending the polynomial in the clear as in regular SumCheck. This greatly reduces the proof size.
- Second, in standard SumCheck, the prover opens the univariate polynomial commitment at three points: at 0, 1, and at a random $r \in \mathbb{F}$. We optimize this step by showing that opening the commitment at a *single* point is sufficient. This further shortens the final proof.

The key point is that the resulting ZeroCheck requires the prover to do only about $s + d \cdot \mu$ group exponentiations, which is much smaller than $d \cdot s$ in Plonk. The additional arithmetic work that the prover needs to do depends on the number of multiplication gates in the circuit implementing the custom gate G , not on the total degree of G , as in Plonk. As such, we can support much larger custom gates than Plonk.

In summary, proof generation time is reduced for two reasons: (i) the elimination of the FFTs, and (ii) the better handling of high-degree custom gates.

Proving the wiring identity. The prover convinces the verifier that the Wiring identity holds by proving the set equality in (5). We describe a set equality protocol over the hypercube in Section 3.4. Briefly, we use a technique from Bayer and Groth [8], that is also used in Plonk, to reduce this problem to a certain ProductCheck over the hypercube (Section 3.3). We then use an idea from Quarks [67] to reduce the hypercube ProductCheck to a ZeroCheck, which then reduces to a SumCheck. This sequence of reductions is shown in Figure 1. Again, no FFTs are needed.

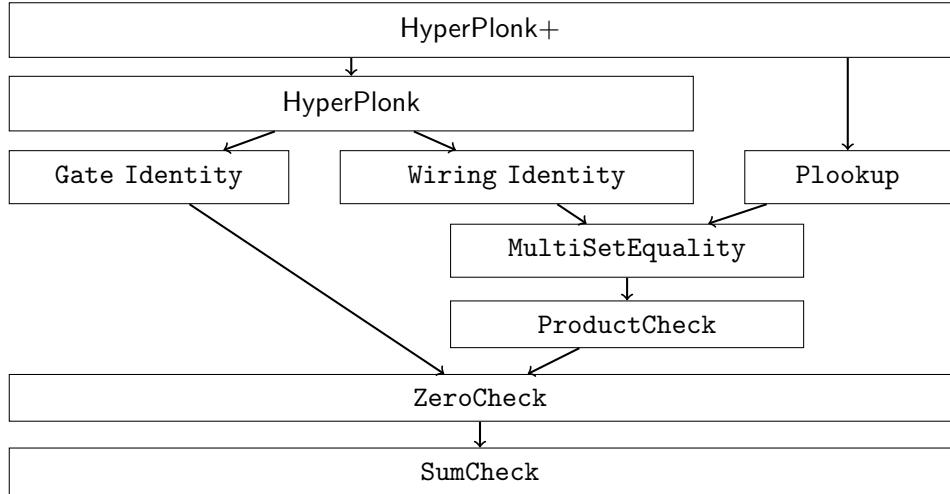


Figure 1: The multilinear polynomial-IOPs that make up HyperPlonk.

Table lookups. An important extension to Plonk supports circuits with table lookup gates. The table is represented as a fixed vector $\mathbf{t} \in \mathbb{F}^{2^\mu-1}$. A table lookup gate ensures that a specific cell in the matrix \hat{M} is contained in \mathbf{t} . For example, one can set \mathbf{t} to be the field elements in $\{0, 1, \dots, B\}$ for some B (padding the vector by 0 as needed). Now, checking that a cell in \hat{M} is contained in \mathbf{t} is a simple way to implement a range check.

Let $f, t : B_\mu \rightarrow \mathbb{F}$ be two multilinear polynomials. Here the polynomial t encodes the table \mathbf{t} , where the table values are $t(B_\mu)$. The polynomial f encodes the cells of \hat{M} that need to be checked. An important step in supporting lookup gates in Plonk is a way for the prover to convince the verifier that $f(B_\mu) \subseteq t(B_\mu)$, when the verifier has commitments to f and t . The Plookup proof system by Gabizon and Williamson [42] is a way for the prover to do just that. Caulk [80, 65] is a more recent alternative to Plookup.

The problem is that Plookup is designed to work when the polynomials are defined over a cyclic subgroup $\mathbb{G} \subseteq \mathbb{F}^*$ of order q with generator $\omega \in \mathbb{G}$. In particular, Plookup requires a function $\text{next} : \mathbb{F} \rightarrow \mathbb{F}$ that induces an ordering of \mathbb{G} . This function must satisfy two properties: (i) the sequence

$$\omega, \quad \text{next}(\omega), \quad \text{next}(\text{next}(\omega)), \quad \dots, \quad \text{next}^{(q-1)}(\omega) \quad (6)$$

should traverse all of \mathbb{G} , and (ii) the function next should be a *linear* function. This is quite easy in a cyclic group: simply define $\text{next}(x) := \omega x$.

To adapt Plookup to the hypercube we need a *linear* function $\text{next} : \mathbb{F}^\mu \rightarrow \mathbb{F}^\mu$ that traverses all of B_μ as in (6), starting with some element $\mathbf{x}_0 \in B_\mu$. However, such an \mathbb{F} -linear function does not exist. Nevertheless, we construct in Section 3.6 a quadratic function from \mathbb{F}^μ to \mathbb{F}^μ that traverses B_μ . We then show how to linearize it by modifying some of the building blocks that Plookup uses. This gives an efficient Plookup protocol over the hypercube. Finally, in Section 5 we use this hypercube Plookup protocol to support lookup gates in HyperPlonk. The resulting protocol is called HyperPlonk+.

1.2 Additional related work

The origins of SNARKs date back to the work of Kilian [56] and Micali [62] based on the PCP theorem. Many of the SNARK constructions cited in the previous sections rely on techniques introduced in the proof of the PCP theorem.

Recursive SNARKs [72] are an important technique for building a SNARK for a long computation. Early recursive SNARKs [35, 17, 14, 34] built a prover for the entire SNARK circuit and then repeatedly used this prover. More recent recursive SNARKs rely on accumulation schemes [24, 27, 19, 26, 57] where the bulk of the SNARK verifier runs outside of the prover.

Many practical SNARKs rely on the random oracle model and often use a non-falsifiable assumption. Indeed, a separation result due to Gentry and Wichs [46] suggests that a SNARK requires either an idealized model or a non-falsifiable assumption. An interesting recent direction is the construction of *batch proofs* [36, 37, 74] in the standard model from standard assumptions. These give succinct proofs for computations in \mathcal{P} , namely succinct proofs for computations that do not rely on a hidden witness. SNARKs give succinct proofs for computations in \mathcal{NP} .

2 Preliminaries

Notation: We use λ to denote the security parameter. For $n \in \mathbb{N}$ let $[n]$ be the set $\{1, 2, \dots, n\}$; for $a, b \in \mathbb{N}$ let $[a, b)$ denote the set $\{a, a+1, \dots, b-1\}$. A function $f(\lambda)$ is $\text{poly}(\lambda)$ if there exists a $c \in \mathbb{N}$ such that $f(\lambda) = O(\lambda^c)$. If for all $c \in \mathbb{N}$, $f(\lambda)$ is $o(\lambda^{-c})$, then $f(\lambda)$ is in $\text{negl}(\lambda)$ and is said to be **negligible**. A probability that is $1 - \text{negl}(\lambda)$ is **overwhelming**. We use \mathbb{F} to denote a field of prime order p such that $\log(p) = \Omega(\lambda)$.

A *multiset* is an extension of the concept of a set where every element has a positive multiplicity. Two finite multisets are equal if they contain the same elements with the same multiplicities.

Recall that a **relation** is a set of pairs (x, w) . An **indexed relation** is a set of triples $(i, x; w)$. The index i is fixed at setup time.

In defining the syntax of the various protocols, we use the following convention concerning public values (known to both the prover and the verifier) and secret ones (known only to the prover). In any list of arguments or returned tuple $(a, b, c; d, e)$, those variables listed before the semicolon are public, and those listed after it are secret. When there is no secret information, the semicolon is omitted.

Useful facts. We next list some facts that will be used throughout the paper.

Lemma 2.1 (Multilinear extensions). *For every function $f : \{0, 1\}^\mu \rightarrow \mathbb{F}$, there is a unique multilinear polynomial $\tilde{f} \in \mathbb{F}[X_1, \dots, X_\mu]$ such that $\tilde{f}(\mathbf{b}) = f(\mathbf{b})$ for all $\mathbf{b} \in \{0, 1\}^\mu$. We call \tilde{f} the multilinear extension of f , and \tilde{f} can be expressed as*

$$\tilde{f}(\mathbf{X}) = \sum_{\mathbf{b} \in \{0, 1\}^\mu} f(\mathbf{b}) \cdot \text{eq}(\mathbf{b}, \mathbf{X})$$

where $\text{eq}(\mathbf{b}, \mathbf{X}) := \prod_{i=1}^\mu (\mathbf{b}_i \mathbf{X}_i + (1 - \mathbf{b}_i)(1 - \mathbf{X}_i))$.

Lemma 2.2 (Schwartz-Zippel Lemma). *Let $f \in \mathbb{F}[X_1, \dots, X_\mu]$ be a non-zero polynomial of total degree d over field \mathbb{F} . Let S be any finite subset of \mathbb{F} , and let r_1, \dots, r_μ be μ field elements selected*

independently and uniformly from set S . Then

$$\Pr [f(r_1, \dots, r_\mu) = 0] \leq \frac{d}{|S|}.$$

Linear codes. We review the definition of linear code.

Definition 2.1 (Linear Code). *An (n, k, δ) -linear error-correcting code $E : \mathbb{F}^k \rightarrow \mathbb{F}^n$ is an injective mapping from \mathbb{F}^k to a linear subspace C in \mathbb{F}^n , such that (i) the injective mapping can be computed in linear time in k ; (ii) any linear combination of codewords is still a codeword; and (iii) the relative hamming distance $\Delta(u, v)$ between any two different codewords $u, v \in \mathbb{F}^k$ is at least δ . The rate of the code E is defined as k/n .*

2.1 Proofs and arguments of knowledge.

We define interactive proofs of knowledge, which consist of a non-interactive *preprocessing* phase run by an indexer as well as an interactive *online* phase between a prover and a verifier.

Definition 2.2 (Interactive Proof and Argument of Knowledge). *An interactive protocol $\Pi = (\text{Setup}, \mathcal{I}, \mathcal{P}, \mathcal{V})$ between a prover \mathcal{P} and verifier \mathcal{V} is an argument of knowledge for an indexed relation \mathcal{R} with knowledge error $\delta : \mathbb{N} \rightarrow [0, 1]$ if the following properties hold, where given an index \mathbf{i} , common input \mathbf{x} and prover witness \mathbf{w} , the deterministic indexer outputs $(\mathbf{vp}, \mathbf{pp}) \leftarrow \mathcal{I}(\mathbf{i})$ and the output of the verifier is denoted by the random variable $\langle \mathcal{P}(\mathbf{pp}, \mathbf{x}, \mathbf{w}), \mathcal{V}(\mathbf{vp}, \mathbf{x}) \rangle$:*

- Perfect Completeness: for all $(\mathbf{i}, \mathbf{x}, \mathbf{w}) \in \mathcal{R}$

$$\Pr \left[\langle \mathcal{P}(\mathbf{pp}, \mathbf{x}, \mathbf{w}), \mathcal{V}(\mathbf{vp}, \mathbf{x}) \rangle = 1 \middle| \begin{array}{l} \mathbf{gp} \leftarrow \text{Setup}(1^\lambda) \\ (\mathbf{vp}, \mathbf{pp}) \leftarrow \mathcal{I}(\mathbf{gp}, \mathbf{i}) \end{array} \right] = 1$$

- δ -Soundness (adaptive): Let $\mathcal{L}(\mathcal{R})$ be the language corresponding to the indexed relation \mathcal{R} such that $(\mathbf{i}, \mathbf{x}) \in \mathcal{L}(\mathcal{R})$ if and only if there exists \mathbf{w} such that $(\mathbf{i}, \mathbf{x}, \mathbf{w}) \in \mathcal{R}$. Π is δ -sound if for every pair of probabilistic polynomial time adversarial prover algorithm $(\mathcal{A}_1, \mathcal{A}_2)$ the following holds:

$$\Pr \left[\langle \mathcal{A}_2(\mathbf{i}, \mathbf{x}, \mathbf{st}), \mathcal{V}(\mathbf{vp}, \mathbf{x}) \rangle = 1 \wedge (\mathbf{i}, \mathbf{x}) \notin \mathcal{L}(\mathcal{R}) \middle| \begin{array}{l} \mathbf{gp} \leftarrow \text{Setup}(1^\lambda) \\ (\mathbf{i}, \mathbf{x}, \mathbf{st}) \leftarrow \mathcal{A}_1(\mathbf{gp}) \\ (\mathbf{vp}, \mathbf{pp}) \leftarrow \mathcal{I}(\mathbf{gp}, \mathbf{i}) \end{array} \right] \leq \delta(|\mathbf{i}| + |\mathbf{x}|).$$

We say a protocol is computationally sound if δ is negligible. If $\mathcal{A}_1, \mathcal{A}_2$ are unbounded and δ is negligible, then the protocol is statistically sound. If $A = (\mathcal{A}_1, \mathcal{A}_2)$ is unbounded, the soundness definition becomes for all $(\mathbf{i}, \mathbf{x}) \notin \mathcal{L}(\mathcal{R})$

$$\Pr \left[\langle \mathcal{A}_2(\mathbf{i}, \mathbf{x}, \mathbf{gp}), \mathcal{V}(\mathbf{vp}, \mathbf{x}) \rangle = 1 \middle| \begin{array}{l} \mathbf{gp} \leftarrow \text{Setup}(1^\lambda) \\ (\mathbf{vp}, \mathbf{pp}) \leftarrow \mathcal{I}(\mathbf{gp}, \mathbf{i}) \end{array} \right] \leq \delta(|\mathbf{i}| + |\mathbf{x}|)$$

- δ -Knowledge Soundness: There exists a polynomial $\text{poly}(\cdot)$ and a probabilistic polynomial-time oracle machine \mathcal{E} called the extractor such that given oracle access to any pair of probabilistic

polynomial time adversarial prover algorithm $(\mathcal{A}_1, \mathcal{A}_2)$ the following holds:

$$\Pr \left[\begin{array}{c} \langle \mathcal{A}_2(\mathbf{i}, \mathbf{x}, \mathbf{st}), \mathcal{V}(\mathbf{vp}, \mathbf{x}) \rangle = 1 \\ \wedge \\ (\mathbf{i}, \mathbf{x}, \mathbf{w}) \notin \mathcal{R} \end{array} \middle| \begin{array}{l} \mathbf{gp} \leftarrow \text{Setup}(1^\lambda) \\ (\mathbf{i}, \mathbf{x}, \mathbf{st}) \leftarrow \mathcal{A}_1(\mathbf{gp}) \\ (\mathbf{vp}, \mathbf{pp}) \leftarrow \mathcal{I}(\mathbf{gp}, \mathbf{i}) \\ \mathbf{w} \leftarrow \mathcal{E}^{\mathcal{A}_1, \mathcal{A}_2}(\mathbf{gp}, \mathbf{i}, \mathbf{x}) \end{array} \right] \leq \delta(|\mathbf{i}| + |\mathbf{x}|)$$

An interactive protocol is “knowledge sound”, or simply an “argument of knowledge”, if the knowledge error δ is negligible in λ . If the adversary is unbounded, then the argument is called an interactive proof of knowledge.

- Public coin An interactive protocol is considered to be public coin if all of the verifier messages (including the final output) can be computed as a deterministic function given a random public input.
- Zero knowledge: An interactive protocol $\langle \mathcal{P}, \mathcal{V} \rangle$ is considered to be zero-knowledge if there is a PPT simulator \mathcal{S} such that for every PPT adversary \mathcal{A} , auxiliary input $z \in \{0, 1\}^{\text{poly}(\lambda)}$, it holds that

$$\Pr \left[\begin{array}{c} \langle \mathcal{P}(\mathbf{pp}, \mathbf{x}, \mathbf{w}), \mathcal{A}(\mathbf{st}, \mathbf{i}, \mathbf{x}) \rangle = 1 \wedge (\mathbf{i}, \mathbf{x}, \mathbf{w}) \in \mathcal{R} \\ \wedge \\ \langle \mathcal{S}(\sigma, z, \mathbf{pp}, \mathbf{x}), \mathcal{A}(\mathbf{st}, \mathbf{i}, \mathbf{x}) \rangle = 1 \wedge (\mathbf{i}, \mathbf{x}, \mathbf{w}) \in \mathcal{R} \end{array} \middle| \begin{array}{l} \mathbf{gp} \leftarrow \text{Setup}(1^\lambda) \\ (\mathbf{i}, \mathbf{x}, \mathbf{w}, \mathbf{st}) \leftarrow \mathcal{A}_1(z, \mathbf{gp}) \\ (\mathbf{vp}, \mathbf{pp}) \leftarrow \mathcal{I}(\mathbf{gp}, \mathbf{i}) \\ (\mathbf{gp}, \sigma) \leftarrow \mathcal{S}(1^\lambda) \\ (\mathbf{i}, \mathbf{x}, \mathbf{w}, \mathbf{st}) \leftarrow \mathcal{A}_1(z, \mathbf{gp}) \\ (\mathbf{vp}, \mathbf{pp}) \leftarrow \mathcal{I}(\mathbf{gp}, \mathbf{i}) \end{array} \right] - \Pr \left[\langle \mathcal{S}(\sigma, z, \mathbf{pp}, \mathbf{x}), \mathcal{A}(\mathbf{st}, \mathbf{i}, \mathbf{x}) \rangle = 1 \wedge (\mathbf{i}, \mathbf{x}, \mathbf{w}) \in \mathcal{R} \middle| \begin{array}{l} (\mathbf{gp}, \sigma) \leftarrow \mathcal{S}(1^\lambda) \\ (\mathbf{i}, \mathbf{x}, \mathbf{w}, \mathbf{st}) \leftarrow \mathcal{A}_1(z, \mathbf{gp}) \\ (\mathbf{vp}, \mathbf{pp}) \leftarrow \mathcal{I}(\mathbf{gp}, \mathbf{i}) \end{array} \right] \leq \text{negl}(\lambda).$$

We say that $\langle \mathcal{P}, \mathcal{V} \rangle$ is statistically zero knowledge if \mathcal{A} is unbounded; and say it perfectly zero knowledge if $\text{negl}(\lambda)$ is replaced with zero. $\langle \mathcal{P}, \mathcal{V} \rangle$ is honest-verifier zero knowledge (HVZK) if the adversary \mathcal{A} honestly follows the verifier algorithm.

We introduce both notions of soundness and knowledge soundness. Knowledge soundness implies soundness, as the existence of an extractor implies that $(\mathbf{i}, \mathbf{x}) \in \mathcal{L}(\mathcal{R})$. Furthermore, we show in Lemma 2.3 that soundness directly implies knowledge soundness for certain oracle relations and oracle arguments.

PolyIOPs. SNARKs can be constructed from information-theoretic proof systems that give the verifier oracle access to prover messages. The information-theoretic proof is then compiled using a cryptographic tool, such as a polynomial commitment. We now define a specific type of information-theoretic proof system called polynomial interactive oracle proofs.

Definition 2.3. A polynomial interactive oracle proof (PIOP) is a public-coin interactive proof for a polynomial oracle relation $\mathcal{R} = \{(\mathbf{i}, \mathbf{x}; \mathbf{w})\}$. The relation is an oracle relation in that \mathbf{i} , and \mathbf{x} can contain oracles to μ -variate polynomials over some field \mathbb{F} . The oracles specify μ and the degree in each variable. These oracles can be queried at arbitrary points in \mathbb{F}^μ to evaluate the polynomial at these points. The actual polynomials corresponding to the oracles are contained in the \mathbf{pp} and the \mathbf{w} , respectively. We denote an oracle to a polynomial f by $[[f]]$. In every protocol message, the \mathcal{P} sends multi-variate polynomial oracles. The verifier in every round sends a random challenge.

We measure the following parameters for the complexity of a PIOP:

- The prover time measures the runtime of the prover.
- The verifier time measures the runtime of the verifier.
- The query complexity is the number of queries the verifier performs to the oracles.
- The round complexity measures the number of rounds. In our protocols, it is always equivalent to the number of oracles sent.
- The size of the proof oracles is the length of the transmitted polynomials.
- The size of the witness is the length of the witness polynomial.

Proof of Knowledge. As a proof system, the PIOP satisfies perfect completeness and unbounded knowledge-soundness with knowledge-error δ . Note that the extractor can query the oracle at arbitrary points to efficiently recover the entire polynomial.

Non-interactive arguments Interactive public-coin arguments can be made non-interactive using the Fiat-Shamir transform. The Fiat-Shamir transform replaces the verifier challenges with hashes of the transcript up to that point. The works by [5, 76] show that this is secure for multi-round special-sound protocols and multi-round oracle proofs.

Soundness and knowledge soundness

Lemma 2.3 (Sound PIOPs are knowledge sound). *Consider a δ -sound PIOP for oracle relations \mathcal{R} such that for all $(\mathbf{i}, \mathbf{x}, \mathbf{w}) \in \mathcal{R}$, \mathbf{w} consists only of polynomials such that the instance contains oracles to these polynomials. The PIOP has δ knowledge-soundness, and the extractor runs in time $O(|\mathbf{w}|)$*

Proof. We will show that we can construct an extractor \mathcal{E} that can produce \mathbf{w}^* such that $(\mathbf{i}, \mathbf{x}, \mathbf{w}^*) \in \mathcal{R}$ if and only if $(\mathbf{i}, \mathbf{x}) \in \mathcal{L}(\mathcal{R})$. This implies that the soundness error exactly matches the knowledge soundness error. For each oracle of a μ -variate polynomial with degree d in each variable, the extractor queries the polynomial at $(d+1)^\mu$ distinct points to extract the polynomial inside the oracle and thus w^* . If $(\mathbf{i}, \mathbf{x}, \mathbf{w}^*) \in \mathcal{R}$ then by definition $(\mathbf{i}, \mathbf{x}) \in \mathcal{L}(\mathcal{R})$. Additionally assume that $(\mathbf{i}, \mathbf{x}) \in \mathcal{L}(\mathcal{R})$ but $(\mathbf{i}, \mathbf{x}, \mathbf{w}^*) \notin \mathcal{R}$. Then there must exist $\mathbf{w}' \neq \mathbf{w}^*$ such that $(\mathbf{i}, \mathbf{x}, \mathbf{w}') \in \mathcal{R}$. Since the relation only admits polynomials as witnesses and these polynomials are degree d and μ -variate, then there cannot be two distinct witnesses that agree on $(d+1)^\mu$ oracle queries. Therefore $\mathbf{w}' = \mathbf{w}^*$ which leads to a contradiction. The extractor, therefore, outputs the unique, valid witness for every (\mathbf{i}, \mathbf{x}) in the language, and thus, the soundness and knowledge soundness error are the same. \square

2.2 Multilinear polynomial commitments.

Definition 2.4 (Commitment scheme). *A commitment scheme Γ is a tuple $\Gamma = (\text{Setup}, \text{Commit}, \text{Open})$ of PPT algorithms where:*

- $\text{Setup}(1^\lambda) \rightarrow \mathbf{gp}$ generates public parameters \mathbf{gp} ;
- $\text{Commit}(\mathbf{gp}; x) \rightarrow (C; r)$ takes a secret message x and outputs a public commitment C and (optionally) a secret opening hint r (which might or might not be the randomness used in the computation).
- $\text{Open}(\mathbf{gp}, C, x, r) \rightarrow b \in \{0, 1\}$ verifies the opening of commitment C to the message x provided with the opening hint r .

A commitment scheme Γ is **binding** if for all PPT adversaries \mathcal{A} :

$$\Pr \left[b_0 = b_1 \neq 0 \wedge x_0 \neq x_1 : \begin{array}{l} \text{gp} \leftarrow \text{Setup}(1^\lambda) \\ (C, x_0, x_1, r_0, r_1) \leftarrow \mathcal{A}(\text{gp}) \\ b_0 \leftarrow \text{Open}(\text{gp}, C, x_0, r_0) \\ b_1 \leftarrow \text{Open}(\text{gp}, C, x_1, r_1) \end{array} \right] \leq \text{negl}(\lambda)$$

A commitment scheme Γ is **hiding** if for any polynomial-time adversary \mathcal{A} :

$$\left| \Pr \left[b = b' : \begin{array}{l} \text{gp} \leftarrow \text{Setup}(1^\lambda) \\ (x_0, x_1, st) \leftarrow \mathcal{A}(\text{gp}) \\ b \overset{\$}{\leftarrow} \{0, 1\} \\ (C_b; r_b) \leftarrow \text{Commit}(\text{gp}; x_b) \\ b' \leftarrow \mathcal{A}(\text{gp}, st, C_b) \end{array} \right] - 1/2 \right| = \text{negl}(\lambda).$$

If the adversary is unbounded, then we say the commitment is statistically hiding. We additionally define polynomial commitment schemes for multi-variate polynomials.

Definition 2.5. (Polynomial commitment) A polynomial commitment scheme is a tuple of protocols $\Gamma = (\text{Setup}, \text{Commit}, \text{Open}, \text{Eval})$ where $(\text{Setup}, \text{Commit}, \text{Open})$ is a binding commitment scheme for a message space $R[X]$ of polynomials over some ring R , and

- $\text{Eval}((\text{vp}, \text{pp}), C, \mathbf{z}, y, d, \mu; f) \rightarrow b \in \{0, 1\}$ is an interactive public-coin protocol between a PPT prover \mathcal{P} and verifier \mathcal{V} . Both \mathcal{P} and \mathcal{V} have as input a commitment C , points $\mathbf{z} \in \mathbb{F}^\mu$ and $y \in \mathbb{F}$, and a degree d . The prover has prover parameters pp , and the verifier has verifier parameters vp . The prover additionally knows the opening of C to a secret polynomial $f \in \mathcal{F}_\mu^{(\leq d)}$. The protocol convinces the verifier that $f(\mathbf{z}) = y$.

A polynomial commitment scheme is **correct** if an honest committer can successfully convince the verifier of any evaluation. Specifically, if the prover is honest, then for all polynomials $f \in \mathcal{F}_\mu^{(\leq d)}$ and all points $z \in \mathbb{F}^\mu$,

$$\Pr \left[b = 1 : \begin{array}{l} \text{gp} \leftarrow \text{Setup}(1^\lambda) \\ (C; r) \leftarrow \text{Commit}(\text{gp}, f) \\ y \leftarrow f(\mathbf{z}) \\ b \leftarrow \text{Eval}(\text{gp}, c, z, y, d, \mu; f, r) \end{array} \right] = 1.$$

We require that Eval is an interactive argument of knowledge and has knowledge soundness, which ensures that we can extract the committed polynomial from any evaluation.

Multi-variate polynomial commitments can be instantiated from random oracles using the FRI protocol [82], bilinear groups [63], groups of unknown order [28] and discrete logarithm groups. We give a table of polynomial commitments with their different properties in Table 2:

Virtual oracles and commitments Given multiple polynomial oracles, we can construct virtual oracles to the functions of these polynomials. An oracle to $g([[f_1]], \dots, [[f_k]])$ for some function g is simply the list of oracles $\{[[f_1]], \dots, [[f_k]]\}$ as well as a description of g . In order to evaluate $g([[f_1]], \dots, [[f_k]])$ at some point \mathbf{x} we compute $y_i = f_i(\mathbf{x}) \forall i \in [k]$ and output $g(y_1, \dots, y_k)$. Equivalently given commitments to polynomials, we can construct a virtual commitment to a function of these polynomials in the same manner. If g is an additive function and the polynomial commitment is additively homomorphic, then we can use the homomorphism to do the evaluation. A

Scheme		Prover time: Commit+ Eval	Verifier time	Proof size	$n = 2^{25}$	Setup	Add.
KZG-based [63]	BL	$n \mathbb{G}_1$	$\log(n) P$	$\log(n) \mathbb{G}_1$	0.8KB	Univ.	Yes
Dory [58]	BL	$n\mathbb{G}_1 + \sqrt{n}P$	$\log(n) \mathbb{G}_T$	$6\log(n) \mathbb{G}_T$	30KB	Trans.	Yes
Bulletproofs [25]	DL	$n \mathbb{G}$	$n \mathbb{G}$	$2\log(n) \mathbb{G}$	1.6KB	Trans.	Yes
FRI-based (§B)	RO	$n \log(n) \rho \mathbb{F} + n\rho H$	$\log^2(n) \frac{\lambda}{\log \rho} H$	$\log^2(n) \frac{\lambda}{\log \rho} H$	250KB	Trans.	No
Orion	RO	$nH + \frac{n}{k} + k$ rec.	$\lambda \log^2 n H$	$\lambda \log^2 n H$	5.5MB	Trans.	No
Orion + (§7)	BL	$n/k \mathbb{G}_1 + nH + (k\lambda H + \frac{n}{k} \mathbb{F})$ rec.	$\log(n) P$	$4 \log n \mathbb{G}_1$	7KB	Univ.	No

Table 2: Multi-linear polynomial commitment schemes for μ -variate linear polynomials and $n = 2^\mu$. The prover time measures the complexity of committing to a polynomial and evaluating it once. The commitment size is constant for all protocols. Unless constants are mentioned, the metrics are assumed to be asymptotic. The 6th column measures the concrete proof size for $n = 2^{25}$, i.e. $\mu = 25$. Legend: BL=Bilinear Group, DL=Discrete Logarithm, RO=Random Oracle, H=Hashes, P=pairings, G=group scalar multiplications, rec.= Recursive circuit size, univ.= universal setup, trans.= transparent setup, Add.=Additive

common example is that given additive commitments C_f, C_g to polynomials $f(\mathbf{X}), g(\mathbf{X})$, we want to construct a commitment to $(1 - Y)f + Yg$. Then (C_f, C_g) serves as such a commitment and we can evaluate it at (y, \mathbf{x}) by evaluating $(1 - y)C_f + y \cdot C_g$ at \mathbf{x} .

2.3 PIOP Compilation

PIOP compilation transforms the interactive oracle proof into an interactive argument of knowledge (without oracles) Π . The compilation replaces the oracles with polynomial commitments. Every query by the verifier is replaced with an invocation of the `Eval` protocol at the query point \mathbf{z} . The compiled verifier accepts if the PIOP verifier accepts and if the output of all `Eval` invocations is 1. If Π is public-coin, then it can further be compiled to a non-interactive argument of knowledge (or NARK) using the Fiat-Shamir transform.

Theorem 2.4 (PIOP Compilation [28, 33]). *If the polynomial commitment scheme Γ has witness-extended emulation, and if the t -round Polynomial IOP for \mathcal{R} has negligible knowledge error, then Π , the output of the PIOP compilation, is a secure (non-oracle) argument of knowledge for \mathcal{R} . The compilation also preserves zero knowledge. If Γ is hiding and `Eval` is honest-verifier zero-knowledge, then Π is honest-verifier zero-knowledge. The efficiency of the resulting argument of knowledge Π depends on the efficiency of both the PIOP and Γ :*

- Prover time The prover time is equal to the sum of (i) prover time of the PIOP, (ii) the oracle length times the commitment time, and (iii) the query complexity times the prover time of Γ .
- Verifier time The verifier time is equal to the sum of (i) the verifier time of the PIOP and (ii) the verifier time for Γ times the query complexity of the PIOP.
- Proof size The proof size is equal to sum of (i) the message complexity of the PIOP times the commitment size and (ii) the query complexity times the proof size of Γ . If the proof size is $O(\log^c(|w|))$, then we say the proof is succinct.

Batching The prover time, verifier time, and proof size can be significantly reduced using batch openings of the polynomial commitments. After batching, the proof size only depends on the number

of oracles plus a single polynomial commitment opening.

3 A toolbox for multivariate polynomials

We begin by reviewing several important PolyIOPs that will serve as building blocks for HyperPlonk. Some are well-known, and some are new. Figure 1 serves as a guide for this section: we define the PolyIOPs listed in the figure following the dependency order.

Notation. From here on, we let $B_\mu := \{0, 1\}^\mu \subseteq \mathbb{F}^\mu$ be the boolean hypercube. We use $\mathcal{F}_\mu^{(\leq d)}$ to denote the set of multivariate polynomials in $\mathbb{F}[X_1, \dots, X_\mu]$ where the degree in each variable is at most d ; moreover, we require that each polynomial in $\mathcal{F}_\mu^{(\leq d)}$ can be expressed as a virtual oracle to $c = O(1)$ multilinear polynomials, that is, with the form $f(\mathbf{X}) := g(h_1(\mathbf{X}), \dots, h_c(\mathbf{X}))$ where $h_i \in \mathcal{F}_\mu^{(\leq 1)}$ ($1 \leq i \leq c$) is multilinear and g is a c -variate polynomial of total degree at most d . For polynomials $f, g \in \mathcal{F}_\mu^{(\leq d)}$, we denote $\text{merge}(f, g) \in \mathcal{F}_{\mu+1}^{(\leq d)}$ as

$$\text{merge}(f, g) := h(\mathbf{X}_0, \dots, \mathbf{X}_\mu) := (1 - \mathbf{X}_0) \cdot f(\mathbf{X}_1, \dots, \mathbf{X}_\mu) + \mathbf{X}_0 \cdot g(\mathbf{X}_1, \dots, \mathbf{X}_\mu) \quad (7)$$

so that $h(0, \mathbf{X}) = f(\mathbf{X})$ and $h(1, \mathbf{X}) = g(\mathbf{X})$. In the following definitions, we omit the public parameters $\text{gp} := (\mathbb{F}, \mu, d)$ when the context is clear. We use $\delta_{\text{XXX}}^{d, \mu}$ to denote the soundness error of the PolyIOP for relation \mathcal{R}_{XXX} with public parameter (\mathbb{F}, d, μ) , where $\text{XXX} \in \{\text{sum}, \text{zero}, \text{prod}, \text{mset}, \text{perm}, \text{lkup}\}$.

Scheme	\mathcal{P} time	\mathcal{V} time	Num of queries	Num of rounds	Proof oracle size	Witness size
SumCheck	$O(2^\mu d \log^2 d)$	$O(\mu)$	$\mu + 1$	μ	$d\mu$	$O(2^\mu)$
ZeroCheck	$O(2^\mu d \log^2 d)$	$O(\mu)$	$\mu + 1$	μ	$d\mu$	$O(2^\mu)$
ProdCheck	$O(2^\mu d \log^2 d)$	$O(\mu)$	$\mu + 2$	$\mu + 1$	$O(2^\mu)$	$O(2^\mu)$
MsetEqChk	$O(2^\mu d \log^2 d)$	$O(\mu)$	$\mu + 2$	$\mu + 1$	$O(2^\mu)$	$O(k2^\mu)$
PermCheck	$O(2^\mu d \log^2 d)$	$O(\mu)$	$\mu + 2$	$\mu + 1$	$O(2^\mu)$	$O(2^\mu)$
Plookup	$O(2^\mu d \log^2 d)$	$O(\mu)$	$\mu + 3$	$\mu + 2$	$O(2^\mu)$	$O(2^\mu)$
BatchEval	$O(2^\mu k)$	$O(k\mu)$	1	$\mu + \log k$	$O(\mu + \log k)$	$O(k2^\mu)$

Table 3: The complexity of PIOPs. d and μ denote the degree and the number of variables of the multivariate polynomials; k in MsetCheck is the length of each element in the multisets; k in BatchEval is the number of evaluations.

3.1 SumCheck PIOP for high degree polynomials

In this section, we describe a PIOP for the sumcheck relation using the classic sumcheck protocol [59]. However, we modify the protocol and adapt it to our setting of high-degree polynomials.

Definition 3.1 (SumCheck relation). *The relation \mathcal{R}_{SUM} is the set of all tuples $(\mathbf{x}; \mathbf{w}) = ((s, [[f]]); f)$ where $f \in \mathcal{F}_\mu^{(\leq d)}$ and $\sum_{\mathbf{x} \in B_\mu} f(\mathbf{x}) = s$.*

Construction. The classic SumCheck protocol [59] is a PolyIOP for the relation \mathcal{R}_{SUM} . When applying the protocol to a polynomial $f \in \mathcal{F}_\mu^{(\leq d)}$, the protocol runs in μ rounds where in every round, the prover sends a univariate polynomial of degree at most d to the verifier. The verifier then sends a random challenge point for the univariate polynomial. At the end of the protocol, the verifier checks the consistency between the univariate polynomials and the multi-variate polynomial using a single query to f .

Given a tuple $(\mathbf{x}; \mathbf{w}) = (v, f; [[f]])$ for μ -variate degree d polynomial f such that $\sum_{\mathbf{b} \in B_\mu} f(\mathbf{b}) = v$:

- For $i = \mu, \mu - 1, \dots, 1$:
 - The prover computes $r_i(X) := \sum_{\mathbf{b} \in B_{i-1}} f(\mathbf{b}, X, \alpha_{i+1}, \dots, \alpha_\mu)$ and sends the oracle $[[r_i]]$ to the verifier. r_i is univariate and of degree at most d .
 - The verifier checks that $v = r_i(0) + r_i(1)$, samples $\alpha_i \leftarrow \mathbb{F}$, sends α_i to the prover, and sets $v \leftarrow r_i(\alpha_i)$.
- Finally, the verifier accepts if $f(\alpha_1, \dots, \alpha_\mu) = v$.

Theorem 3.1. *The PIOP for \mathcal{R}_{SUM} is perfectly complete and has knowledge error $\delta_{\text{sum}}^{d,\mu} := d\mu/|\mathbb{F}|$.*

We refer to [71] for the proof of the theorem.

Sending r as an oracle. Unlike in the classic sumcheck protocol, we send an oracle to r_i , in each round, instead of the actual polynomial. This does not change the soundness analysis, as the soundness is still proportional to the degree of the univariate polynomials sent in each round. However, it reduces the communication and verifier complexity, especially if the degree of r is large, as in our application of Hyperplonk with custom gates.

Moreover, the verifier has to evaluate r_i at three points: 0, 1, and α_i . As a useful optimization, the prover can instead send an oracle for the degree $d - 2$ polynomial

$$r'_i(X) := \frac{r_i(X) - (1 - X) \cdot r_i(0) - X \cdot r_i(1)}{X \cdot (1 - X)},$$

along with $r_i(0)$. The verifier then computes $r_i(1) \leftarrow v - r_i(0)$ and

$$r_i(\alpha_i) \leftarrow r'_i(\alpha) \cdot (1 - \alpha_i) \cdot \alpha_i + (1 - \alpha_i) \cdot r_i(0) + \alpha_i \cdot r_i(1).$$

This requires only one query to the oracle of r'_i at α_i and one field element per round.

Computing sumcheck for high-degree polynomials. Consider a multi-variate polynomial $f(\mathbf{X}) := h(g_1(\mathbf{X}), \dots, g_c(\mathbf{X}))$ such that h is degree d and can be evaluated through an arithmetic circuit with $O(d)$ gates. In the sumcheck protocol, the prover has to compute a univariate polynomial $r_i(X)$ in each round using the previous verifier messages $\alpha_1, \dots, \alpha_{i-1}$. We adapt the algorithm by [70, 77] that showed how the sumcheck prover can be run in time linear in 2^μ using dynamic programming. The algorithm takes as input a description of f as well as the sumcheck round challenges $\alpha_1, \dots, \alpha_\mu$. It outputs the round polynomials r_1, \dots, r_μ . The sumcheck prover runs the algorithm in parallel to the sumcheck protocol, taking each computed r_i as that rounds message:

In [70, 77], $r^{(\mathbf{b})}(X) := h(r^{(1,\mathbf{b})}(X), \dots, r^{(c,\mathbf{b})}(X))$ is computed by evaluating h on d distinct values for X , e.g. $X \in \{0, \dots, d\}$ and interpolating the output. This works as h is a degree d

Algorithm 1 Computing r_1, \dots, r_μ [70, 77]

```

1: procedure SUMCHECK PROVER( $h, g_1(\mathbf{X}), \dots, g_c(\mathbf{X})$ )
2:   For each  $g_j$  build table  $A_j : \{0, 1\} \rightarrow \mathbb{F}$  of all evaluations over  $B_\mu$ 
3:   for  $i \leftarrow \mu, 1$  do
4:     For each  $\mathbf{b} \in B_{i-1}$  and each  $j \in [c]$ , define  $r^{(j,\mathbf{b})}(X) := (1-X)A_j[\mathbf{b}, 0] + X A_j[\mathbf{b}, 1]$ .
5:     Compute  $r^{(\mathbf{b})}(X) \leftarrow h(r^{(1,\mathbf{b})}(X), \dots, r^{(c,\mathbf{b})}(X))$  for all  $\mathbf{b} \in B_{i-1}$  using Algorithm 2 .
6:      $r_i(X) \leftarrow \sum_{\mathbf{b} \in B_{i-1}} r^{(\mathbf{b})}(X)$ .
7:     Send  $r_i(X)$  to  $\mathcal{V}$ .
8:     Receive  $\alpha_i$  from  $\mathcal{V}$ .
9:     Set  $A_j[\mathbf{b}] \leftarrow r^{(j,\mathbf{b})}(\alpha_i)$  for each  $\mathbf{b} \in B_{i-1}$ .
10:    end for
11: end procedure

```

polynomial and each $r^{j,\mathbf{b}}$ is linear. Evaluating $r^{j,\mathbf{b}}$ on d points can be done in d steps. So the total time to evaluate all $r^{j,\mathbf{b}}$ for $j \in [c]$ is $c \cdot d$. Furthermore, the circuit has $O(d)$ gates, and evaluating it on d inputs, takes time $O(d^2)$. Assuming that $c \approx d$ the total time to compute $r^{(\mathbf{b})}$ with this algorithm is $O(d^2)$ and the time to run Algorithm 1 is $O(2^\mu d^2)$.

We show how this can be reduced to $O(2^\mu \cdot d \log^2 d)$ for certain low depth circuits, such as $h := \prod_c r_c(\mathbf{X})$. The core idea is that evaluating the circuit for h *symbolically*, instead of at d individual points, is faster if fast polynomial multiplication algorithms are used.

We will present the algorithm for computing $h(X) := \prod_{j=1}^d r_j(X)$, then we will discuss how to extend this for more general h . Assume w.l.o.g. that d is a power of 2.

Algorithm 2 Evaluating $h := \prod_{j=1}^d r_j$

Require: r_1, \dots, r_d are linear functions

```

1: procedure  $h(r_1(X), \dots, r_d(X))$ 
2:    $t_{1,j} \leftarrow r_j$  for all  $j \in [d]$ .
3:   for  $i \leftarrow 1, \log d$  do
4:      $t_{i+1,j}(X) \leftarrow t_{i,2j-1}(X) \cdot t_{i,2j}(X)$                                  $\triangleright$  Using fast polynomial multiplication
5:   end for
6:   return  $h = t_{\log_2(d),1}$ 
7: end procedure

```

In round i there are $d/2^i$ polynomial multiplications for polynomials of degree 2^{i-1} . In FFT-friendly³ fields, polynomial multiplication can be performed in time $O(d \log(d))$.⁴ The total running time of the algorithm is therefore $\sum_{i=1}^{\log_2(d)} \frac{d}{2^i} 2^{i-1} \log(2^i) = \sum_{i=1}^{\log_2(d)} O(d \cdot i) = O(d \log^2(d))$.

Algorithm 2 naturally extends to more complicated, low-depth circuits. Addition gates are performed directly through polynomial addition, which takes $O(d)$ time for degree d polynomials. As long as the circuit is low-depth and has $O(d)$ multiplication gates, the complexity remains $O(d \log^2(d))$. Furthermore, we can compute $r^k(X)$ for $k \leq d$ using only a single FFT of length

³These are fields where there exists an element that has a smooth order of at least d .

⁴Recent breakthrough results have shown that polynomial multiplication is $O(d \log(d))$ over arbitrary finite fields [52] and there have been efforts toward building practical, fast multiplication algorithms for arbitrary fields [11]. In practice, and especially for low-degree polynomials, using Karatsuba multiplication might be faster.

$\deg(r) \cdot k$ for an input polynomial r . The FFT evaluates r at $\deg(r) \cdot k$ points. Then we raise each point to the power of k . This takes time $O(\deg(r) \cdot k(\log(\deg(r)) + \log(k)))$ and saves a factor of $\log(k)$ over a repeated squaring implementation.

Batching. Multiple sumcheck instances, e.g. $(s, [[f]])$ and $(s', [[g]])$ can easily be batched together. This is done using a random-linear combination, i.e. showing that $(s + \alpha s', [[f]] + \alpha [[g]]) \in \mathcal{L}(\mathcal{R}_{\text{SUM}})$ for a random verifier-generated α [73, 32]. The batching step has soundness $\frac{1}{|\mathbb{F}|}$.

Complexity. Overall, Algorithm 1 calls Algorithm 2 for each point in the boolean hypercube and then on each point in a cube of half the size. The total runtime of Algorithm 1 is, therefore, $O(2^\mu d \log^2 d)$ if h is degree d and low-depth. We summarize the complexity of the PIOP for \mathcal{R}_{SUM} with respect to $f \in \mathcal{F}_\mu^{(\leq d)}$, below:

- The prover time is $\text{tp}_{\text{sum}}^f = \mathcal{O}(2^\mu \cdot d \log^2 d)$ \mathbb{F} -ops (for low-depth f that can be evaluated in time $O(d)$).
- The verifier time is $\text{tv}_{\text{sum}}^f = \mathcal{O}(\mu)$.
- The query complexity is $\text{q}_{\text{sum}}^f = \mu + 1$, μ queries to univariate oracles, one to multi-variate f .
- The round complexity and the number of proof oracles is $\text{rc}_{\text{sum}}^f = \mu$.
- The number of field elements sent by \mathcal{P} is μ .
- The size of the proof oracles is $\text{pl}_{\text{sum}}^f = d \cdot \mu$; the size of the witness is $c \cdot 2^\mu$.

3.2 ZeroCheck PIOP

In this section, we describe a PIOP showing that a multivariate polynomial evaluates to zero everywhere on the boolean hypercube. The PIOP builds upon the sumcheck PIOP in Section 3.1 and is a key building block for product-check PIOP in Section 3.3. The zerocheck PIOP is also helpful in HyperPlonk for proving the gate identity.

Definition 3.2 (ZeroCheck relation). *The relation $\mathcal{R}_{\text{ZERO}}$ is the set of all tuples $(\mathbf{x}; \mathbf{w}) = (([[f]]); f)$ where $f \in \mathcal{F}_\mu^{(\leq d)}$ and $f(\mathbf{x}) = 0$ for all $\mathbf{x} \in B_\mu$.*

We use an idea from [66] to reduce a ZeroCheck to a SumCheck.

Construction. Given a tuple $(\mathbf{x}; \mathbf{w}) = (([[f]]); f)$, the protocol is the following:

- \mathcal{V} sends \mathcal{P} a random vector $\mathbf{r} \xleftarrow{\$} \mathbb{F}^\mu$
- Let $\hat{f}(\mathbf{X}) := f(\mathbf{X}) \cdot eq(\mathbf{X}, \mathbf{r})$ where $eq(\mathbf{x}, \mathbf{y}) := \prod_{i=1}^\mu (x_i y_i + (1 - x_i)(1 - y_i))$.
- Run a sumcheck PolyIOP to convince the verifier that $((0, [[\hat{f}]]) ; \hat{f}) \in \mathcal{R}_{\text{SUM}}$.

Batching It is possible to batch two instances $(([[f]]); f) \in \mathcal{R}_{\text{ZERO}}$ and $(([[g]]); g) \in \mathcal{R}_{\text{ZERO}}$ by running a zerocheck on $(([[f + \alpha g]]); f + \alpha g)$ for a random $\alpha \in \mathbb{F}$. The soundness error of the batching protocol $\frac{1}{|\mathbb{F}|}$.

Theorem 3.2. *The PIOP for $\mathcal{R}_{\text{ZERO}}$ is perfectly complete and has knowledge error $\delta_{\text{zero}}^{d, \mu} := d\mu/|\mathbb{F}| + \delta_{\text{sum}}^{d+1, \mu} = \mathcal{O}(d\mu/|\mathbb{F}|)$.*

Proof. **Completeness.** For every $(([[f]]); f) \in \mathcal{R}_{\text{ZERO}}$, \hat{f} is also zero everywhere on the boolean hypercube, thus the sumcheck of \hat{f} is zero, and completeness follows from sumcheck's completeness.

Knowledge soundness. By Lemma 2.3, it is sufficient to argue the soundness error of the protocol. We note that $[[f]] \in \mathcal{L}(\mathcal{R}_{\text{ZERO}})$ (i.e., $(([[f]]); f) \in \mathcal{R}_{\text{ZERO}}$) if and only if the following auxiliary polynomial

$$g(\mathbf{Y}) := \sum_{\mathbf{x} \in B_\mu} f(\mathbf{x}) \cdot eq(\mathbf{x}, \mathbf{Y})$$

is identically zero. This is because $eq(x, y)$ for a $\mathbf{x}, \mathbf{y} \in B_\mu$ is 1 if $\mathbf{x} = \mathbf{y}$ and 0 otherwise. So $g(\mathbf{y}) = f(\mathbf{y})$ for all $\mathbf{y} \in B_\mu$. Therefore, for any $[[f]] \notin \mathcal{L}(\mathcal{R}_{\text{ZERO}})$, the corresponding g is a non-zero polynomial and by Lemma 2.2,

$$g(\mathbf{r}) = \sum_{\mathbf{x} \in B_\mu} f(\mathbf{x}) \cdot eq(\mathbf{x}, \mathbf{r}) = 0$$

with probability $d\mu/|\mathbb{F}|$ over the choice of \mathbf{r} , thus the probability that the verifier accepts is at most $d\mu/|\mathbb{F}|$ plus the probability that the SumCheck PIOP verifier accepts when $((0, [[\hat{f}]]) ; \hat{f}) \notin \mathcal{R}_{\text{SUM}}$, which is $d\mu/|\mathbb{F}| + \delta_{\text{sum}}^{d+1, \mu}$ as desired. \square

Complexity. We analyze the complexity of the PIOP for $\mathcal{R}_{\text{ZERO}}$ with respect to $f \in \mathcal{F}_\mu^{(\leq d)}$.

- The prover time is $\text{tp}_{\text{zero}}^f = \text{tp}_{\text{sum}}^{\hat{f}} = \mathcal{O}(d \log^2 d \cdot 2^\mu)$ \mathbb{F} -ops.
- The verifier time is $\text{tv}_{\text{zero}}^f = \mathcal{O}(\mu)$.
- The query complexity is $\text{q}_{\text{zero}}^f = \text{q}_{\text{sum}}^{\hat{f}} = \mu + 1$.
- The round complexity and the number of proof oracles is $\text{rc}_{\text{zero}}^f = \text{rc}_{\text{sum}}^{\hat{f}} = \mu$.
- The number of field elements sent by \mathcal{P} is $\text{nf}_{\text{zero}}^f = \text{nf}_{\text{sum}}^{\hat{f}} = \mu$.
- The size of the proof oracles is $\text{pl}_{\text{zero}}^f = \text{pl}_{\text{sum}}^{\hat{f}} = d\mu$; the size of the witness is $\mathcal{O}(2^\mu)$.

3.3 ProductCheck PIOP

We describe a PIOP for the product check relation, that is, for a rational polynomial (where both the nominator and the denominator are multivariate polynomials), the product of the evaluations on the boolean hypercube is a claimed value s . The PIOP uses the idea from the Quark system [67, §5], we adapt it to build upon the zerocheck PIOP in Section 3.2. Product check PIOP is a key building block for the multiset equality check PIOP in Section 3.4.

Definition 3.3 (ProductCheck relation). *The relation $\mathcal{R}_{\text{PROD}}$ is the set of all tuples $(\mathbf{x}; \mathbf{w}) = ((s, [[f_1]], [[f_2]]); f_1, f_2)$ where $f_1 \in \mathcal{F}_\mu^{(\leq d)}$, $f_2 \in \mathcal{F}_\mu^{(\leq d)} \setminus \{0\}$ and $\prod_{\mathbf{x} \in B_\mu} f'(\mathbf{x}) = s$, where f' is the rational polynomial $f' := f_1/f_2$. In the case that $f_2 = c$ is a constant polynomial, we directly set $f := f_1/c$ and write $(\mathbf{x}; \mathbf{w}) = ((s, [[f]]); f)$.*

Construction. The Quark system [67, §5] constructs a proof system for the $\mathcal{R}_{\text{PROD}}$ relation. The proof system uses an instance of the $\mathcal{R}_{\text{ZERO}}$ PolyIOP on $\mu + 1$ variables. Given a tuple $(\mathbf{x}; \mathbf{w}) = ((s, [[f_1]], [[f_2]]); f_1, f_2)$, we denote by $f' := f_1/f_2$. The protocol is the following:

- \mathcal{P} sends an oracle $\tilde{v} \in \mathcal{F}_{\mu+1}^{(\leq 1)}$ such that for all $\mathbf{x} \in B_\mu$,

$$\tilde{v}(0, \mathbf{x}) = f'(\mathbf{x}), \quad \tilde{v}(1, \mathbf{x}) = \tilde{v}(\mathbf{x}, 0) \cdot \tilde{v}(\mathbf{x}, 1).$$

- Define $\hat{h} := \text{merge}(\hat{f}, \hat{g}) \in \mathcal{F}_{\mu+1}^{(\leq \max(2, d+1))}$ where

$$\hat{f}(\mathbf{X}) := \tilde{v}(1, \mathbf{X}) - \tilde{v}(\mathbf{X}, 0) \cdot \tilde{v}(\mathbf{X}, 1), \quad \hat{g}(\mathbf{X}) := f_2(\mathbf{X}) \cdot \tilde{v}(0, \mathbf{X}) - f_1(\mathbf{X}).$$

Run a ZeroCheck PolyIOP for $([[\hat{h}]]; \hat{h}) \in \mathcal{R}_{\text{ZERO}}$, i.e., the polynomial \tilde{v} is computed correctly.

- \mathcal{V} queries $[[\tilde{v}]]$ at point $(1, \dots, 1, 0) \in \mathbb{F}^{\mu+1}$, and checks that the evaluation is s .

Theorem 3.3. *Let $d' := \max(2, d + 1)$. The PIOP for $\mathcal{R}_{\text{PROD}}$ is perfectly complete and has knowledge error $\delta_{\text{prod}}^{d,\mu} := \delta_{\text{zero}}^{d',\mu+1} = \mathcal{O}(d'\mu/|\mathbb{F}|)$.*

Proof. **Completeness.** First, if the prover honestly generates \tilde{v} , it holds that $(([[\hat{h}]]); \hat{h}) \in \mathcal{R}_{\text{ZERO}}$, and the verifier accepts in the sub-PIOP, given that ZeroCheck is complete. Second, if $((s, [[f_1]], [[f_2]]); f_1, f_2) \in \mathcal{R}_{\text{PROD}}$, the evaluation $\tilde{v}(1, \dots, 1, 0)$ is exactly the product of f 's evaluations on the boolean hypercube B_μ (c.f. [67, §5]), which is s as desired.

Knowledge soundness. By Lemma 2.3, it is sufficient to argue the soundness error of the protocol. For any $(s, [[f_1]], [[f_2]]) \notin \mathcal{L}(\mathcal{R}_{\text{PROD}})$ and any \tilde{v} sent by a malicious prover, it holds that either \tilde{v} is not computed correctly (i.e., $(([[\hat{h}]]); \hat{h}) \notin \mathcal{R}_{\text{ZERO}}$), or the evaluation $\tilde{v}(1, \dots, 1, 0) \neq s$ and \mathcal{V} rejects. Hence the probability that \mathcal{V} accepts is at most $\max(\delta_{\text{zero}}^{d',\mu+1}, 0) = \delta_{\text{zero}}^{d',\mu+1}$ as claimed. \square

Complexity. Let \hat{h} be the polynomials described in the construction, we analyze the complexity of the PIOP for $\mathcal{R}_{\text{PROD}}$ with respect to $f' := f_1/f_2$ where $f_1, f_2 \in \mathcal{F}_\mu^{(\leq d)}$.

- The prover time is $\text{tp}_{\text{prod}}^{f'} = \text{tp}_{\text{zero}}^{\hat{h}} + 2^\mu = \mathcal{O}(d \log^2 d \cdot 2^\mu) \mathbb{F}\text{-ops}$. The term 2^μ is for computing the product polynomial \tilde{v} .
- The verifier time is $\text{tv}_{\text{prod}}^{f'} = \text{tv}_{\text{zero}}^{\hat{h}} = \mathcal{O}(\mu)$.
- The query complexity is $\text{q}_{\text{prod}}^{f'} = \text{q}_{\text{zero}}^{\hat{h}} + 1 = \mu + 2$, the additional query is for $\tilde{v}(1, \dots, 1, 0)$.
- The round complexity and the number of proof oracles is $\text{rc}_{\text{prod}}^{f'} = \text{rc}_{\text{zero}}^{\hat{h}} + 1 = \mu + 1$.
- The number of field elements sent by \mathcal{P} is $\text{nf}_{\text{prod}}^{f'} = \text{nf}_{\text{zero}}^{\hat{h}} = \mu$.
- The size of the proof oracles is $\text{pl}_{\text{prod}}^f = 2^\mu + \text{pl}_{\text{zero}}^{\hat{h}} = \mathcal{O}(2^\mu)$; the size of the witness is $\mathcal{O}(2^\mu)$.

3.4 Multiset Check PIOP

We describe a multivariate PIOP checking that two multisets are equal. The PIOP builds upon the product-check PIOP in Section 3.3. The multiset check PIOP is a key building block for the permutation PIOP in Section 3.5 and the lookup PIOP in Section 3.6. A similar idea has been proposed in the univariate polynomial setting by Gabizon in a blogpost [41].

Definition 3.4 (Multiset Check relation). *For any $k \geq 1$, the relation $\mathcal{R}_{\text{MSET}}^k$ is the set of all tuples*

$$(\mathbf{x}; \mathbf{w}) = (([[f_1]], \dots, [[f_k]], [[g_1]], \dots, [[g_k]]); (f_1, \dots, f_k, g_1, \dots, g_k))$$

where $f_i, g_i \in \mathcal{F}_\mu^{(\leq d)}$ ($1 \leq i \leq k$) and the following two multisets of tuples are equal:

$$\left\{ \mathbf{f}_x := [f_1(x), \dots, f_k(x)] \right\}_{x \in B_\mu} = \left\{ \mathbf{g}_x := [g_1(x), \dots, g_k(x)] \right\}_{x \in B_\mu}$$

Basic construction. We start by describing a PolyIOP for $\mathcal{R}_{\text{MSET}}^1$. The protocol can be obtained from a protocol for $\mathcal{R}_{\text{PROD}}$. Given a tuple $(([[f]], [[g]]); (f, g))$, the protocol is the following:

- \mathcal{V} samples and sends \mathcal{P} a challenge $r \leftarrow \mathbb{F}$.
- Set $f' := r + f$ and $g' := r + g$, run a ProductCheck PolyIOP for $((1, [[f']], [[g']]); f', g') \in \mathcal{R}_{\text{PROD}}$.

Theorem 3.4. *The PIOP for $\mathcal{R}_{\text{MSET}}^1$ is perfectly complete and has knowledge error $\delta_{mset,1}^{d,\mu} := 2^\mu/|\mathbb{F}| + \delta_{prod}^{d,\mu} = \mathcal{O}((2^\mu + d\mu)/|\mathbb{F}|)$.*

Proof. **Completeness.** For any $(([[f]], [[g]]); (f, g)) \in \mathcal{R}_{\text{MSET}}^1$, it holds that

$$\prod_{\mathbf{x} \in B_\mu} (r + f(\mathbf{x})) = \prod_{\mathbf{x} \in B_\mu} (r + g(\mathbf{x})),$$

thus $\prod_{\mathbf{x} \in B_\mu} (r + f(\mathbf{x}))/ (r + g(\mathbf{x})) = 1$, i.e., $((1, [[r + f]], [[r + g]]); r + f, r + g) \in \mathcal{R}_{\text{PROD}}$. Therefore completeness holds given that the PolyIOP for $\mathcal{R}_{\text{PROD}}$ is complete.

Knowledge soundness. By Lemma 2.3, it is sufficient to argue the soundness error of the protocol. For any $(([[f]], [[g]]) \notin \mathcal{L}(\mathcal{R}_{\text{MSET}}^1)$ (i.e., $(([[f]], [[g]]); (f, g)) \notin \mathcal{R}_{\text{MSET}}^1$), it holds that

$$F(Y) := \prod_{\mathbf{x} \in B_\mu} (Y + f(\mathbf{x})) \neq G(Y) := \prod_{\mathbf{x} \in B_\mu} (Y + g(\mathbf{x})).$$

By Lemma 2.2, $F(r) \neq G(r)$ with probability at least $1 - (2^\mu/|\mathbb{F}|)$. Conditioned on $F(r) \neq G(r)$, it holds that $((1, [[r + f]], [[r + g]]); r + f, r + g) \notin \mathcal{R}_{\text{PROD}}$. Hence the probability that \mathcal{V} accepts conditioned on $F(r) \neq G(r)$ is at most $\delta_{prod}^{d,\mu}$. In summary, the probability that \mathcal{V} accepts is at most $2^\mu/|\mathbb{F}| + \delta_{prod}^{d,\mu}$ as claimed. \square

The final construction. Next we describe the protocol for $\mathcal{R}_{\text{MSET}}^k$ for any $k \geq 1$. Given a tuple

$$(([[f_1]], \dots, [[f_k]], [[g_1]], \dots, [[g_k]]); (f_1, \dots, f_k, g_1, \dots, g_k)),$$

the protocol is the following:

- \mathcal{V} samples and sends \mathcal{P} challenges $r_2, \dots, r_k \leftarrow \mathbb{F}$.
- Run a Multiset Check PolyIOP for $(([[\hat{f}]], [[\hat{g}]]) ; (\hat{f}, \hat{g})) \in \mathcal{R}_{\text{MSET}}^1$, where $\hat{f}, \hat{g} \in \mathcal{F}_\mu^{(\leq d)}$ are defined as $\hat{f} := f_1 + r_2 \cdot f_2 + \dots + r_k \cdot f_k$ and $\hat{g} := g_1 + r_2 \cdot g_2 + \dots + r_k \cdot g_k$.

Theorem 3.5. *The PIOP for $\mathcal{R}_{\text{MSET}}^k$ is perfectly complete and has knowledge error $\delta_{mset,k}^{d,\mu} := 2^\mu/|\mathbb{F}| + \delta_{mset,1}^{d,\mu} = \mathcal{O}((2^\mu + d\mu)/|\mathbb{F}|)$.*

Proof. **Completeness.** Completeness holds since the PolyIOP for $(([[f]], [[g]]); (f, g)) \in \mathcal{R}_{\text{MSET}}^1$ is complete.

Knowledge soundness. By Lemma 2.3, it is sufficient to argue the soundness error of the protocol. Given any

$$(([[f_1]], \dots, [[f_k]], [[g_1]], \dots, [[g_k]])) \notin \mathcal{L}(\mathcal{R}_{\text{MSET}}^k),$$

let

$$U := \{\mathbf{f}_x := [f_1(\mathbf{x}), \dots, f_k(\mathbf{x})]\}_{\mathbf{x} \in B_\mu}, \quad V := \{\mathbf{g}_x := [g_1(\mathbf{x}), \dots, g_k(\mathbf{x})]\}_{\mathbf{x} \in B_\mu}$$

denote the corresponding multisets. Let W be the maximal multiset such that $W \subseteq U$ and $W \subseteq V$. We set $U' := U \setminus W$, $V' := V \setminus W$.⁵ We observe that $|U'| = |V'| > 0$ as $U \neq V$, and $U' \cap V' = \emptyset$ by definition of W . Thus there exists an element $\mathbf{x} \in \mathbb{F}^k$ where $\mathbf{x} \in U'$ but $\mathbf{x} \notin V'$. It is well-known that the map $\phi_{\mathbf{r}} : (x_1, \dots, x_k) \rightarrow x_1 + r_2 x_2 + \dots + r_k x_k$ is a universal hash family [31, 75, 68], that is, for any $\mathbf{x}, \mathbf{y} \in \mathbb{F}^k$, $\mathbf{x} \neq \mathbf{y}$, it holds that

$$\Pr_{\mathbf{r}} [\phi_{\mathbf{r}}(\mathbf{x}) = \phi_{\mathbf{r}}(\mathbf{y})] \leq \frac{1}{|\mathbb{F}|}.$$

Thus by union bound, the probability (over the choice of \mathbf{r}) that

$$\phi_{\mathbf{r}}(\mathbf{x}) \in \{\phi_{\mathbf{r}}(\mathbf{y}) : \mathbf{y} \in V'\}$$

is at most $|V'|/|\mathbb{F}| \leq 2^\mu/|\mathbb{F}|$. Conditioned on that above does not happen, we have that $(([[\hat{f}]], [[\hat{g}]]) \notin \mathcal{L}(\mathcal{R}_{\text{MSET}}^1))$ and the probability that \mathcal{V} accepts in the PolyIOP for $\mathcal{R}_{\text{MSET}}^1$ is at most $\delta_{\text{mset},1}^{d,\mu}$. In summary, the soundness error is at most $2^\mu/|\mathbb{F}| + \delta_{\text{mset},1}^{d,\mu}$ as claimed. \square

Complexity. We analyze the complexity of the PIOP for $\mathcal{R}_{\text{MSET}}$ with respect to

$$\mathbf{F} := (f_1, \dots, f_k, g_1, \dots, g_k) \in \left(\mathcal{F}_\mu^{(\leq d)}\right)^{2k}.$$

- The prover time is $\text{tp}_{k,\text{mset}}^{\mathbf{F}} = \text{tp}_{1,\text{mset}}^{\hat{f}, \hat{g}} = \text{tp}_{\text{prod}}^{f'/g'} = \mathcal{O}(d \log^2 d \cdot 2^\mu)$ \mathbb{F} -ops (for k where $\hat{f} := f_1 + r_2 \cdot f_2 + \dots + r_k \cdot f_k$ and $\hat{g} := g_1 + r_2 \cdot g_2 + \dots + r_k \cdot g_k$ can be evaluated in time $O(d)$).
- The verifier time is $\text{tv}_{\text{mset}}^{\mathbf{F}} = \text{tv}_{\text{prod}}^{f'/g'} = \mathcal{O}(\mu)$.
- The query complexity is $\text{q}_{\text{mset}}^{\mathbf{F}} = \text{q}_{\text{prod}}^{f'/g'} = \mu + 2$.
- The round complexity and the number of proof oracles is $\text{rc}_{\text{mset}}^{\mathbf{F}} = \text{rc}_{\text{prod}}^{f'/g'} = \mu + 1$.
- The number of field elements sent by \mathcal{P} is $\text{nf}_{\text{mset}}^{\mathbf{F}} = \text{nf}_{\text{prod}}^{f'/g'} = \mu$.
- The size of the proof oracles is $\text{pl}_{\text{mset}}^{\mathbf{F}} = \text{pl}_{\text{prod}}^{f'/g'} = \mathcal{O}(2^\mu)$; the size of the witness is $\mathcal{O}(k \cdot 2^\mu)$.

3.5 Permutation PIOP

We describe a multivariate PIOP showing that for two multivariate polynomials $f, g \in \mathcal{F}_\mu^{(\leq d)}$, the evaluations of g on the boolean hypercube is a predefined permutation σ of f 's evaluations on the boolean hypercube. The permutation PIOP is a key building block of HyperPlonk for proving the wiring identity.

Definition 3.5 (Permutation relation). *The indexed relation $\mathcal{R}_{\text{PERM}}$ is the set of tuples*

$$(\mathbf{i}; \mathbf{x}; \mathbf{w}) = (\sigma; ([[f]], [[g]]); (f, g))$$

, where $\sigma : B_\mu \rightarrow B_\mu$ is a permutation, $f, g \in \mathcal{F}_\mu^{(\leq d)}$, and $g(\mathbf{x}) = f(\sigma(\mathbf{x}))$ for all $\mathbf{x} \in B_\mu$.

⁵E.g., if $k = 1$ and $U = \{1, 1, 1, 2\}$ and $V = \{1, 1, 2, 2\}$, then $W = \{1, 1, 2\}$, $U' = \{1\}$ and $V' = \{2\}$.

Construction. Gabizon et. al. [44] construct a permutation argument. We adapt their scheme into a multivariate PolyIOP. The construction uses a PolyIOP instance for $\mathcal{R}_{\text{MSET}}$. Given a tuple $(\sigma; ([[f]], [[g]]); (f, g))$ where σ is the predefined permutation, the indexer generates two oracles $[[s_{\text{id}}]], [[s_\sigma]]$ such that $s_{\text{id}} \in \mathcal{F}_\mu^{(\leq 1)}$ maps each $\mathbf{x} \in B_\mu$ to $[\mathbf{x}] := \sum_{i=1}^\mu \mathbf{x}_i \cdot 2^{i-1} \in \mathbb{F}$, and $s_\sigma \in \mathcal{F}_\mu^{(\leq 1)}$ maps each $\mathbf{x} \in B_\mu$ to $[\sigma(\mathbf{x})]$.⁶ The PolyIOP is the following:

- Run a Multiset Check PolyIOP for

$$(([[[s_{\text{id}}]], [[f]], [[s_\sigma]], [[g]]]; (s_{\text{id}}, f, s_\sigma, g)) \in \mathcal{R}_{\text{MSET}}^2.$$

Theorem 3.6. *The PIOP for $\mathcal{R}_{\text{PERM}}$ is perfectly complete and has knowledge error $\delta_{\text{perm}}^{d,\mu} := \delta_{\text{mset},2}^{d,\mu} = \mathcal{O}((2^\mu + d\mu)/|\mathbb{F}|)$.*

Proof. **Completeness.** For any $(\sigma; ([[f]], [[g]]); (f, g)) \in \mathcal{R}_{\text{PERM}}$, it holds that the multiset $\{([\mathbf{x}], f(\mathbf{x}))\}_{\mathbf{x} \in B_\mu}$ is identical to the multiset $\{([\sigma(\mathbf{x})], g(\mathbf{x}))\}_{\mathbf{x} \in B_\mu}$. Thus

$$(([[[s_{\text{id}}]], [[f]], [[s_\sigma]], [[g]]]; (s_{\text{id}}, f, s_\sigma, g)) \in \mathcal{R}_{\text{MSET}}^2$$

and completeness follows from the completeness of the PolyIOP for $\mathcal{R}_{\text{MSET}}^2$.

Knowledge soundness. By Lemma 2.3, it is sufficient to argue the soundness error of the protocol. The PolyIOP has soundness error $\delta_{\text{mset},2}^{d,\mu}$ as the permutation relation holds if and only if the above multiset check relation holds. \square

Complexity. The complexity of the PIOP for $\mathcal{R}_{\text{PERM}}$ with respect to $f, g \in \mathcal{F}_\mu^{(\leq d)}$ is identical to the complexity of the PIOP for $\mathcal{R}_{\text{MSET}}^2$ with respect to $(s_{\text{id}}, f, s_\sigma, g)$.

3.6 Lookup PIOP

This section describes a multivariate PIOP checking the table lookup relation. The PIOP builds upon the multiset check PIOP (Section 3.4) and is a key building block for HyperPlonk+ (Section 5). Our construction is inspired by a univariate PIOP for the table lookup relation called **Plookup** [42]. However, it is non-trivial to adapt **Plookup** to the multivariate setting because their scheme requires the existence of a subdomain of the polynomial that is a cyclic subgroup \mathbb{G} with a generator $\omega \in \mathbb{G}$. Translating to the multilinear case, we need to build an efficient function g that generates the entire boolean hypercube; moreover, g has to be linear so that the degree of the polynomial does not blow up. However, such a linear function does not exist. Fortunately, we can construct a quadratic function from \mathbb{F}^μ to \mathbb{F}^μ that traverses B_μ . We then show how to linearize it by modifying some of the building blocks that **Plookup** uses. This gives an efficient **Plookup** protocol over the hypercube.

Definition 3.6 (Lookup relation). *The indexed relation $\mathcal{R}_{\text{LOOKUP}}$ is the set of tuples*

$$(\mathbf{i}; \mathbf{x}; \mathbf{w}) = (\mathbf{t}; [[f]]; (f, \text{addr}))$$

where $\mathbf{t} \in \mathbb{F}^{2^\mu-1}$, $f \in \mathcal{F}_\mu^{(\leq d)}$, and $\text{addr} : B_\mu \rightarrow [1, 2^\mu)$ is a map such that $f(\mathbf{x}) = \mathbf{t}_{\text{addr}(\mathbf{x})}$ for all $\mathbf{x} \in B_\mu$.

Before presenting the PIOP for $\mathcal{R}_{\text{LOOKUP}}$, we first show how to build a *quadratic* function that generates the entire boolean hypercube.

⁶Here we further require $|\mathbb{F}| \geq 2^\mu$ so that $[\mathbf{x}]$ never overflow.

A quadratic generator in \mathbb{F}_{2^μ} . For every $\mu \in \mathbb{N}$, we fix a *primitive polynomial* $p_\mu \in \mathbb{F}_2[X]$ where $p_\mu := X^\mu + \sum_{s \in S} X^s + 1$ for some set $S \subseteq [\mu - 1]$, so that $\mathbb{F}_2[X]/(p_\mu) \cong \mathbb{F}_2^\mu[X] \cong \mathbb{F}_{2^\mu}$. By definition of primitive polynomials, $X \in \mathbb{F}_2^\mu[X]$ is a generator of $\mathbb{F}_2^\mu[X] \setminus \{0\}$. This naturally defines a generator function $g_\mu : B_\mu \rightarrow B_\mu$ as

$$g_\mu(\mathbf{b}_1, \dots, \mathbf{b}_\mu) = (\mathbf{b}_\mu, \mathbf{b}'_1, \dots, \mathbf{b}'_{\mu-1})$$

, where $\mathbf{b}'_i = \mathbf{b}_i \oplus \mathbf{b}_\mu$ ($i \leq 1 < \mu$) if $i \in S$, and $\mathbf{b}'_i = \mathbf{b}_i$ otherwise. Essentially, for a polynomial $f \in \mathbb{F}_2^\mu[X]$ with coefficients \mathbf{b} , $g_\mu(\mathbf{b})$ is the coefficient vector of $X \cdot f(X)$. Hence the following lemma is straightforward.

Lemma 3.7. *Let $g_\mu : B_\mu \rightarrow B_\mu$ be the generator function defined above. For every $\mathbf{x} \in B_\mu \setminus \{0^\mu\}$, it holds that $\{g_\mu^{(i)}(\mathbf{x})\}_{i \in [2^\mu-1]} = B_\mu \setminus \{0^\mu\}$, where $g_\mu^{(i)}(\cdot)$ denotes i repeated application of g_μ .*

Directly composing a polynomial f with the generator g will blow up the degree of the resulting polynomial; moreover, the prover needs to send the composed oracle $f(g(\cdot))$. Both of which affect the efficiency of the PIOP. We address the issue by describing a trick that manipulates f in a way that simulates the behavior of $f(g(\cdot))$ on the boolean hypercube, but without blowing up the degree.

Linearizing the generator. For a multivariate polynomial $f \in \mathcal{F}_\mu^{(\leq d)}$, we define $f_{\Delta_\mu} \in \mathcal{F}_\mu^{(\leq d)}$ as

$$f_{\Delta_\mu}(\mathbf{X}_1, \dots, \mathbf{X}_\mu) := \mathbf{X}_\mu \cdot f(1, \mathbf{X}'_1, \dots, \mathbf{X}'_{\mu-1}) + (1 - \mathbf{X}_\mu) \cdot f(0, \mathbf{X}_1, \dots, \mathbf{X}_{\mu-1})$$

where $\mathbf{X}'_i := 1 - \mathbf{X}_i$ ($i \leq 1 < \mu$) if $i \in S$, and $\mathbf{X}'_i := \mathbf{X}_i$ otherwise.

Lemma 3.8. *For every $\mu \in \mathbb{N}$, let $g_\mu : B_\mu \rightarrow B_\mu$ be the generator function defined in Lemma 3.7. For every $d \in \mathbb{N}$ and polynomial $f \in \mathcal{F}_\mu^{(\leq d)}$, it holds that $f_{\Delta_\mu}(\mathbf{x}) = f(g_\mu(\mathbf{x}))$ for every $\mathbf{x} \in B_\mu$. Moreover, f_{Δ_μ} has individual degree d and one can evaluate f_{Δ_μ} from 2 evaluations of f .*

Proof. By definition, f_{Δ_μ} has individual degree d and an evaluation of f_{Δ_μ} can be derived from 2 evaluations of f . Next, we argue that $f_{\Delta_\mu}(\mathbf{x}) = f(g_\mu(\mathbf{x}))$ for every $\mathbf{x} \in B_\mu$.

First, $f_{\Delta_\mu}(0^\mu) = f(g_\mu(0^\mu))$ because $f_{\Delta_\mu}(0^\mu) = f(0^\mu)$ and $g_\mu(0^\mu) = 0^\mu$ by definition of f_{Δ_μ}, g_μ . Second, for every $\mathbf{x} \in B_\mu \setminus \{0^\mu\}$, by definition of g_μ ,

$$f(g_\mu(\mathbf{x}_1, \dots, \mathbf{x}_\mu)) = f(\mathbf{x}_\mu, \mathbf{x}'_1, \dots, \mathbf{x}'_{\mu-1}),$$

where $\mathbf{x}'_i = \mathbf{x}_i \oplus \mathbf{x}_\mu$ ($i \leq 1 < \mu$) for every i in the fixed set S , and $\mathbf{x}'_i = \mathbf{x}_i$ otherwise. We observe that $\mathbf{x}_i \oplus \mathbf{x}_\mu = 1 - \mathbf{x}_i$ when $\mathbf{x}_\mu = 1$ and $\mathbf{x}_i \oplus \mathbf{x}_\mu = \mathbf{x}_i$ when $\mathbf{x}_\mu = 0$, thus we can rewrite

$$\begin{aligned} f(\mathbf{x}_\mu, \mathbf{x}'_1, \dots, \mathbf{x}'_{\mu-1}) &= \mathbf{x}_\mu \cdot f(1, \mathbf{x}_1^*, \dots, \mathbf{x}_{\mu-1}^*) + (1 - \mathbf{x}_\mu) \cdot f(0, \mathbf{x}_1, \dots, \mathbf{x}_{\mu-1}) \\ &= f_{\Delta_\mu}(\mathbf{x}_1, \dots, \mathbf{x}_\mu) \end{aligned}$$

where $\mathbf{x}_i^* = 1 - \mathbf{x}_i$ ($i \leq 1 < \mu$) for every i in the fixed set S , and $\mathbf{x}_i^* = \mathbf{x}_i$ otherwise. The last equality holds by definition of f_{Δ_μ} . In summary, $f(g_\mu(\mathbf{x}_1, \dots, \mathbf{x}_\mu)) = f_{\Delta_\mu}(\mathbf{x}_1, \dots, \mathbf{x}_\mu)$ for every B_μ and the lemma holds. \square

Construction. Now we are ready to present the PIOP for $\mathcal{R}_{\text{LOOKUP}}$, which is an adaptation of Plookup [42] in the multivariate setting. The PIOP invokes a protocol for $\mathcal{R}_{\text{MSET}}^2$. We introduce a notation that embeds a vector to the hypercube while still preserving the vector order with respect to the generator function. For a vector $\mathbf{t} \in \mathbb{F}^{2^\mu-1}$, we denote by $t \leftarrow \text{emb}(\mathbf{t}) \in \mathcal{F}_\mu^{(\leq 1)}$ the multilinear polynomial such that $t(0^\mu) = 0$ and $t(g_\mu^{(i)}(1, 0^{\mu-1})) = \mathbf{t}_i$ for every $i \in [2^\mu - 1]$. By Lemma 3.7, t is well-defined and embeds the entire vector \mathbf{t} onto $B_\mu \setminus \{0^\mu\}$.

For an index $\mathbf{t} \in \mathbb{F}^{2^\mu-1}$, the indexer generates an oracle $[[t]]$ where $t \leftarrow \text{emb}(\mathbf{t})$. For a tuple $(\mathbf{t}; [[f]]; (f, \text{addr}))$ where $f(B_\mu) \subseteq t(B_\mu) \setminus \{0\}$, let $(\mathbf{a}_1, \dots, \mathbf{a}_{2^\mu-1})$ be the vector where $\mathbf{a}_i \in \mathbb{N}$ is the number of appearance of \mathbf{t}_i in $f(B_\mu)$. Note that $\sum_{i=1}^{2^\mu-1} \mathbf{a}_i = 2^\mu$. Denote by $\mathbf{h} \in \mathbb{F}^{2^{\mu+1}-1}$ the vector

$$\mathbf{h} := (\underbrace{\mathbf{t}_1, \dots, \mathbf{t}_1}_{1+\mathbf{a}_1}, \mathbf{t}_2, \dots, \mathbf{t}_{i-1}, \underbrace{\mathbf{t}_i, \dots, \mathbf{t}_i}_{1+\mathbf{a}_i}, \mathbf{t}_{i+1}, \dots, \mathbf{t}_{2^\mu-2}, \underbrace{\mathbf{t}_{2^\mu-1}, \dots, \mathbf{t}_{2^\mu-1}}_{1+\mathbf{a}_{2^\mu-1}}).$$

We present the protocol below:

- \mathcal{P} sends \mathcal{V} oracles $[[h]]$, where $h \leftarrow \text{emb}(\mathbf{h}) \in \mathcal{F}_{\mu+1}^{(\leq 1)}$.
- Define $g_1 := \text{merge}(f, t) \in \mathcal{F}_{\mu+1}^{(\leq d)}$ and $g_2 := \text{merge}(f, t_{\Delta_\mu}) \in \mathcal{F}_{\mu+1}^{(\leq d)}$, where merge is defined in equation (7). Run a multiset check PIOP (Section 3.4) for

$$(([g_1], [[g_2]], [[h]], [[h_{\Delta_{\mu+1}}]]); (f, t, h)) \in \mathcal{R}_{\text{MSET}}^2.$$

- \mathcal{V} queries $h(0^{\mu+1})$ and checks that the answer equals 0.

Theorem 3.9. *The PIOP for $\mathcal{R}_{\text{LOOKUP}}$ is perfectly complete and has knowledge error $\delta_{\text{lkup}}^{d,\mu} := \delta_{\text{mset},2}^{d,\mu+1} = \mathcal{O}((2^\mu + d\mu)/|\mathbb{F}|)$.*

Proof. **Completeness.** Denote by $n := 2^\mu$. For any $(\mathbf{t}; [[f]]; (f, \text{addr})) \in \mathcal{R}_{\text{LOOKUP}}$, let $\mathbf{h} \in \mathbb{F}^{2n-1}$ be the vector defined in the construction. Gabizon and Williamson [42] observed that

$$\{[\mathbf{f}_i, \mathbf{f}_i]\}_{i \in [n]} \cup \{[\mathbf{t}_i, \mathbf{t}_{(i \bmod (n-1))+1}]\}_{i \in [n-1]} = \{[\mathbf{h}_i, \mathbf{h}_{(i \bmod (2n-1))+1}]\}_{i \in [2n-1]},$$

equivalently, by definition of t, h and by Lemma 3.8, the following two multisets of tuples are equal

$$\{[f(\mathbf{x}), f(\mathbf{x})]\}_{\mathbf{x} \in B_\mu} \cup \{[t(\mathbf{x}), t_{\Delta_\mu}(\mathbf{x})]\}_{\mathbf{x} \in B_\mu \setminus \{0^\mu\}} = \{[h(\mathbf{x}), h_{\Delta_{\mu+1}}(\mathbf{x})]\}_{\mathbf{x} \in B_{\mu+1} \setminus \{0^{\mu+1}\}}.$$

By adding element $[0, 0] = [t(0^\mu), t_{\Delta_\mu}(0^\mu)] = [h(0^{\mu+1}), h_{\Delta_{\mu+1}}(0^{\mu+1})]$ on both sides, we have

$$\{[f(\mathbf{x}), f(\mathbf{x})]\}_{\mathbf{x} \in B_\mu} \cup \{[t(\mathbf{x}), t_{\Delta_\mu}(\mathbf{x})]\}_{\mathbf{x} \in B_\mu} = \{[h(\mathbf{x}), h_{\Delta_{\mu+1}}(\mathbf{x})]\}_{\mathbf{x} \in B_{\mu+1}}.$$

Hence the verifier accepts in the multiset check by completeness of the PIOP for $\mathcal{R}_{\text{MSET}}^2$.

Knowledge soundness. By Lemma 2.3, to argue knowledge soundness, it is sufficient to argue the soundness error of the protocol. Fix $n := 2^\mu$, for any $(\mathbf{t}; [[f]]) \notin \mathcal{L}(\mathcal{R}_{\text{LOOKUP}})$, denote by $\mathbf{f} \in \mathbb{F}^n$ the evaluations of f on B_μ . Gabizon et. al. [42] showed that for any $\mathbf{h} \in \mathbb{F}^{2n-1}$, it holds that

$$\{[\mathbf{f}_i, \mathbf{f}_i]\}_{i \in [n]} \cup \{[\mathbf{t}_i, \mathbf{t}_{(i \bmod (n-1))+1}]\}_{i \in [n-1]} \neq \{[\mathbf{h}_i, \mathbf{h}_{(i \bmod (2n-1))+1}]\}_{i \in [2n-1]},$$

since $t(0^\mu) = 0$ and \mathcal{V} checks that $h(0^{\mu+1}) = 0$, with a similar argument as in the completeness proof, we have

$$\{[f(\mathbf{x}), f(\mathbf{x})]\}_{\mathbf{x} \in B_\mu} \cup \{[t(\mathbf{x}), t_{\Delta_\mu}(\mathbf{x})]\}_{\mathbf{x} \in B_\mu} \neq \{[h(\mathbf{x}), h_{\Delta_{\mu+1}}(\mathbf{x})]\}_{\mathbf{x} \in B_{\mu+1}}$$

and the multiset check relation does not hold. Therefore, the probability that \mathcal{V} accepts is at most $\delta_{\text{mset},2}^{d,\mu+1}$ as claimed. \square

Complexity. Let $f, \mathbf{F} := (g_1, g_2, h, h_{\Delta_{\mu+1}}) \in (\mathcal{F}_{\mu+1}^{(\leq d)})^2 \times (\mathcal{F}_{\mu+1}^{(\leq 1)})^2$ be the polynomials defined in the construction. We analyze the complexity of the PIOP for $\mathcal{R}_{\text{LOOKUP}}$ with respect to $f \in \mathcal{F}_{\mu}^{(\leq d)}$.

- The prover time is $\mathbf{tp}_{\text{lkup}}^f = \mathbf{tp}_{\text{mset}}^{\mathbf{F}} = \mathcal{O}(d \log^2 d \cdot 2^{\mu})$ F-ops.
- The verifier time is $\mathbf{tv}_{\text{lkup}}^f = \mathbf{tv}_{\text{mset}}^{\mathbf{F}} = \mathcal{O}(\mu)$.
- The query complexity is $\mathbf{q}_{\text{lkup}}^f = 1 + \mathbf{q}_{\text{mset}}^{\mathbf{F}} = \mu + 3$.
- The round complexity and the number of proof oracles is $\mathbf{rc}_{\text{lkup}}^f = 1 + \mathbf{rc}_{\text{mset}}^{\mathbf{F}} = \mu + 2$.
- The number of field elements sent by \mathcal{P} is $\mathbf{nf}_{\text{lkup}}^f = \mathbf{nf}_{\text{mset}}^{\mathbf{F}} = \mu$.
- The size of the proof oracles is $\mathbf{pl}_{\text{lkup}}^f = 2^{\mu+1} + \mathbf{pl}_{\text{mset}}^{\mathbf{F}} = \mathcal{O}(2^{\mu})$ where $2^{\mu+1}$ is the oracle size of h . The size of the witness is $\mathcal{O}(2^{\mu})$.

3.7 Batch openings

This section describes a batching protocol proving the correctness of multiple multivariate polynomial evaluations. Essentially, the protocol reduces multiple oracle queries to different polynomials into a *single* query to a multivariate oracle. The batching protocol is helpful for HyperPlonk to enable efficient batch evaluation openings. In particular, the SNARK prover only needs to compute a single multilinear PCS evaluation proof, even if there are multiple PCS evaluations.

We note that Thaler [71, §4.5.2] shows how to batch two evaluations of a *single* multilinear polynomial. The algorithm can be generalized for multiple evaluations of *different* multilinear polynomials. However, the prover time complexity is $O(k^2 \mu \cdot 2^{\mu})$ where k is the number of evaluations, and μ is the number of variables. In comparison, our algorithm achieves complexity $O(k \cdot 2^{\mu})$ which is $k\mu$ -factor faster. Note that $O(k \cdot 2^{\mu})$ is already optimal as the prover needs to take $O(k \cdot 2^{\mu})$ time to evaluate $\{f_i(\mathbf{z}_i)\}_{i \in [k]}$ before batching.

Definition 3.7 (BatchEval relation). *The relation $\mathcal{R}_{\text{BATCH}}^k$ is the set of all tuples $(\mathbf{x}; \mathbf{w}) = ((\mathbf{z}_i)_{i \in [k]}, (y_i)_{i \in [k]}, ([f_i])_{i \in [k]}; (f_i)_{i \in [k]})$ where $\mathbf{z}_i \in \mathbb{F}^{\mu}$, $y_i \in \mathbb{F}$, $f_i \in \mathcal{F}_{\mu}^{(\leq d)}$ and $f_i(\mathbf{z}_i) = y_i$ for all $i \in [k]$.*

Remark 3.1. *The polynomials $\{f_i\}_{i \in [k]}$ are not necessarily distinct. E.g., to evaluate a single polynomial f at k distinct points, we can set $f_1 = f_2 = \dots = f_k = f$.*

Remark 3.2. *The polynomials $\{f_i\}_{i \in [k]}$ are all μ -variate. This is without loss of generality. E.g., suppose one of the evaluated polynomial f'_j has only $\mu - 1$ variable, we can define $f_j(Y, \mathbf{X}) = Y \cdot f'_j(\mathbf{X}) + (1 - Y) \cdot f'_j(\mathbf{X})$ which is essentially f'_j but with μ variables. The same trick easily extends to f'_j with arbitrary $\mu' < \mu$ variables.*

Construction. For ease of exposition, we consider the case where f_1, \dots, f_k are *multilinear*. We emphasize that the same techniques can be extended for multi-variate polynomials.

Assume w.l.o.g that $k = 2^{\ell}$ is a power of 2. We observe that $\mathcal{R}_{\text{BATCH}}^k$ is essentially a ZeroCheck relation over the set $Z := \{\mathbf{z}_i\}_{i \in [k]} \subseteq \mathbb{F}^{\mu}$, that is, for every $i \in [k]$, $f_i(\mathbf{z}_i) - y_i = 0$. Nonetheless, Z is outside the boolean hypercube, and we cannot directly reuse the ZeroCheck PIOP.

The key idea is to interpret each zero constraint as a sumcheck via multilinear extension, so that we can work on the boolean hypercube later. In particular, for every $i \in [k]$, we want to constrain $f_i(\mathbf{z}_i) - y_i = 0$. Since f_i is multilinear, by definition of multilinear extension, this is equivalent to

constraining that

$$c_i := \left(\sum_{\mathbf{b} \in B_\mu} f_i(\mathbf{b}) \cdot eq(\mathbf{b}, \mathbf{z}_i) \right) - y_i = 0. \quad (8)$$

Note that equation (8) holds for every $i \in [k]$ if and only if the polynomial

$$\sum_{i \in [k]} eq(\mathbf{Z}, \langle i \rangle) \cdot c_i$$

is identically zero, where $\langle i \rangle$ is ℓ -bit representation of $i - 1$. By Lemma 2.2, it is sufficient to check that for a random vector $\mathbf{t} \xleftarrow{s} \mathbb{F}^\ell$, it holds that

$$\sum_{i \in [k]} eq(\mathbf{t}, \langle i \rangle) \cdot c_i = \sum_{i \in [k]} eq(\mathbf{t}, \langle i \rangle) \cdot \left[\left(\sum_{\mathbf{b} \in B_\mu} f_i(\mathbf{b}) \cdot eq(\mathbf{b}, \mathbf{z}_i) \right) - y_i \right] = 0. \quad (9)$$

Next, we arithmetize equation (9) and make it an algebraic formula. For every $(i, \mathbf{b}) \in [k] \times B_\mu$, we set value $g_{i,\mathbf{b}} := eq(\mathbf{t}, \langle i \rangle) \cdot f_i(\mathbf{b})$, and define an MLE \tilde{g} for $(g_{i,\mathbf{b}})_{i \in [k], \mathbf{b} \in B_\mu}$ such that $\tilde{g}(\langle i \rangle, \mathbf{b}) = g_{i,\mathbf{b}} \forall (i, \mathbf{b}) \in [k] \times B_\mu$; similarly, we define an MLE \tilde{eq} for $(eq(\mathbf{b}, \mathbf{z}_i))_{i \in [k], \mathbf{b} \in B_\mu}$ where $\tilde{eq}(\langle i \rangle, \mathbf{b}) = eq(\mathbf{b}, \mathbf{z}_i) \forall (i, \mathbf{b}) \in [k] \times B_\mu$. Let $s := \sum_{i \in [k]} eq(\mathbf{t}, \langle i \rangle) \cdot y_i$, then equation (9) can be rewritten as

$$\sum_{i \in [k], \mathbf{b} \in B_\mu} \tilde{g}(\langle i \rangle, \mathbf{b}) \cdot \tilde{eq}(\langle i \rangle, \mathbf{b}) = s.$$

This is equivalent to prove a sumcheck claim for the degree-2 polynomial $g^* := \tilde{g}(\mathbf{Y}, \mathbf{X}) \cdot \tilde{eq}(\mathbf{Y}, \mathbf{X})$ over set $B_{\ell+\mu}$. Hence we obtain the following PIOP protocol in Algorithm 3. Note that $g^* = \tilde{g} \cdot \tilde{eq}$ is only with degree 2. Thus we can run a classic sumcheck without sending any univariate oracles.

Algorithm 3 Batch evaluation of multi-linear polynomials

- 1: **procedure** BATCH_EVAL([$f_i \in \mathcal{F}_\mu^{(\leq 1)}$, $\mathbf{z}_i \in \mathbb{F}^\mu$, $y_i \in \mathbb{F}$] $_{i=1}^k$)
 - 2: \mathcal{V} sends \mathcal{P} a random vector $\mathbf{t} \xleftarrow{s} \mathbb{F}^\ell$.
 - 3: Define sum $s := \sum_{i \in [k]} eq(\mathbf{t}, \langle i \rangle) \cdot y_i$.
 - 4: Let \tilde{g} be the MLE for $(g_{i,\mathbf{b}})_{i \in [k], \mathbf{b} \in B_\mu}$ where
- $$g_{i,\mathbf{b}} := eq(\mathbf{t}, \langle i \rangle) \cdot f_i(\mathbf{b}).$$
- 5: Let \tilde{eq} be the MLE for $(eq(\mathbf{b}, \mathbf{z}_i))_{i \in [k], \mathbf{b} \in B_\mu}$ such that $\tilde{eq}(\langle i \rangle, \mathbf{b}) = eq(\mathbf{b}, \mathbf{z}_i)$.
 - 6: \mathcal{P} and \mathcal{V} run a SumCheck PIOP for $(s, [g^*]); g^*$ $\in \mathcal{R}_{\text{SUM}}$, where $g^* := \tilde{g} \cdot \tilde{eq}$.
 - 7: Let $(\mathbf{a}_1, \mathbf{a}_2) \in \mathbb{F}^{\ell+\mu}$ be the sumcheck challenge vector. \mathcal{P} answers the oracle query $\tilde{g}(\mathbf{a}_1, \mathbf{a}_2)$.
 - 8: \mathcal{V} evaluates $\tilde{eq}(\mathbf{a}_1, \mathbf{a}_2)$ herself, and checks that

$$\tilde{g}(\mathbf{a}_1, \mathbf{a}_2) \cdot \tilde{eq}(\mathbf{a}_1, \mathbf{a}_2)$$

is consistent with the last message of the sumcheck.

- 9: **end procedure**
-

Remark 3.3. If the SNARK is using a homomorphic commitment scheme, to answer query $\tilde{g}(\mathbf{a}_1, \mathbf{a}_2)$ the prover only needs to provide a single PCS opening proof for a μ -variate polynomial

$$g'(\mathbf{X}) := \tilde{g}(\mathbf{a}_1, \mathbf{X}) = \sum_{i \in [k]} eq(\langle i \rangle, \mathbf{a}_1) \cdot eq(\mathbf{t}, \langle i \rangle) \cdot f_i(\mathbf{X})$$

on point \mathbf{a}_2 . The verifier can evaluate $\{eq(\langle i \rangle, \mathbf{a}_1) \cdot eq(\mathbf{t}, \langle i \rangle)\}_{i \in [k]}$ in time $O(k)$, and homomorphically compute g' 's commitment from the commitments to $\{f_i\}_{i \in [k]}$, and checks the opening proof against g' 's commitment. Finally, the verifier checks that $g'(\mathbf{a}_2)$ matches the claimed evaluation $\tilde{g}(\mathbf{a}_1, \mathbf{a}_2)$.

Analysis. The PIOP for $\mathcal{R}_{\text{BATCH}}$ is complete and knowledge-sound given the completeness and knowledge-soundness of the sumcheck PIOP.

Next, we analyze the complexity of the protocol: The prover time is $O(k \cdot 2^\mu)$ as it runs a sumcheck PIOP for a polynomial $g^* := \tilde{g} \cdot \tilde{eq}$ of degree 2 and $\mu + \log k$ variables, where \tilde{g} and \tilde{eq} can both be constructed in time $O(k \cdot 2^\mu)$. Note that this is already optimal as the prover anyway needs to take $O(k \cdot 2^\mu)$ time to evaluate $\{f_i(\mathbf{z}_i)\}_{i \in [k]}$ before batching. The verifier takes time $O(\mu + \log k)$ in the sumcheck; the sum s can be computed in time $O(k)$; the evaluation $\tilde{eq}(\mathbf{a}_1, \mathbf{a}_2) = \sum_{i \in [k]} eq(\mathbf{a}_1, \langle i \rangle) \cdot \tilde{eq}(\langle i \rangle, \mathbf{a}_2)$ can be derived from \mathbf{a}_1 and the k evaluations $\{\tilde{eq}(\langle i \rangle, \mathbf{a}_2) = eq(\mathbf{a}_2, \mathbf{z}_i)\}_{i \in [k]}$ where each evaluation $eq(\mathbf{a}_2, \mathbf{z}_i)$ takes time $O(\mu)$. In summary, the verifier time is $O(k\mu)$.

3.7.1 A more efficient batching scheme in a special setting

Sometimes one only needs to open a *single* multilinear polynomial at multiple points, where each point is *in the boolean hypercube*. In this setting, we provide a more efficient algorithm with complexity $O(2^\mu)$ which is k times faster than Algorithm 3. We also note that the technique can be used to construct an efficient Commit-and-Prove SNARK scheme from multilinear commitments.

Recall the sumcheck equation (9) in the general batch opening scheme, when there is only one polynomial f and assume for simplicity that $y_i = 0 \forall i \in [k]$ ⁷, we can rewrite it as

$$\sum_{i \in [k]} eq(\mathbf{t}, \langle i \rangle) \cdot \left(\sum_{\mathbf{b} \in B_\mu} f(\mathbf{b}) \cdot eq(\mathbf{b}, \mathbf{z}_i) \right) = \sum_{\mathbf{b} \in B_\mu} f(\mathbf{b}) \left(\sum_{i \in [k]} eq(\mathbf{t}, \langle i \rangle) \cdot eq(\mathbf{b}, \mathbf{z}_i) \right).$$

Denote by $d_i = eq(\mathbf{t}, \langle i \rangle)$. The above is essentially a sumcheck for polynomial $f \cdot \tilde{eq}^*$ on set B_μ , where

$$\tilde{eq}^*(\mathbf{X}) := \sum_{i \in [k]} d_i \cdot eq(\mathbf{X}, \mathbf{z}_i).$$

Thus we can reduce the batching argument to a PCS opening on polynomial f .

In the sumcheck protocol, in each round $\mu - i + 1 \in [\mu]$, the prover needs to evaluate a degree-2 polynomial $r_i(X)$ on point $x_i \in \{0, 1, 2\}$, where

$$r_i(X) := \sum_{\mathbf{b} \in B_{i-1}} f(\mathbf{b}, X, \boldsymbol{\alpha}) \cdot \tilde{eq}^*(\mathbf{b}, X, \boldsymbol{\alpha}) \tag{10}$$

⁷The algorithm can be easily extended when y_i are non-zero.

and $\boldsymbol{\alpha} = (\alpha_{i+1}, \dots, \alpha_\mu)$ are the round challenges. Note that the evaluation $f(\mathbf{b}, x_i, \boldsymbol{\alpha})$ is easy to obtain by maintaining a table $f(B_{i-1}, \{0, 1, 2\}, \boldsymbol{\alpha})$ as in Algorithm 1. Next we argue that the evaluation $r_i(x_i)$ can be computed in time $O(k)$ given the evaluations $f(B_{i-1}, \{0, 1, 2\}, \boldsymbol{\alpha})$. Since there are μ rounds and the complexity for maintaining the table is $O(2^\mu)$, the total complexity is $O(2^\mu + k\mu)$.

We observe that in equation (10), since $\{\mathbf{z}_i\}_{i \in [k]}$ are in the boolean hypercube, and

$$\begin{aligned}\tilde{eq}^*(\mathbf{b}, X, \boldsymbol{\alpha}) &= \sum_{j \in [k]} d_j \cdot eq((\mathbf{b}, X, \boldsymbol{\alpha}), \mathbf{z}_j) \\ &= \sum_{j \in [k]} d_j \cdot eq(\mathbf{b}, \mathbf{z}_j[1..i-1]) \cdot eq(X, \mathbf{z}_j[i]) \cdot eq(\boldsymbol{\alpha}, \mathbf{z}_j[i+1..]),\end{aligned}$$

by definition of eq , there are at most k choices of \mathbf{b} where $\tilde{eq}^*(\mathbf{b}, X, \boldsymbol{\alpha})$ is non-zero. In particular, the ℓ -th ($1 \leq \ell \leq k$) such vector is $\mathbf{c}_\ell := \mathbf{z}_\ell[1..i-1]$ such that

$$\tilde{eq}^*(\mathbf{c}_\ell, X, \boldsymbol{\alpha}) = \sum_{j \in [k]} d_j \cdot eq(\mathbf{z}_\ell[1..i-1], \mathbf{z}_j[1..i-1]) \cdot eq(X, \mathbf{z}_j[i]) \cdot eq(\boldsymbol{\alpha}, \mathbf{z}_j[i+1..]).$$

we note that for each $j \in [k]$, the value $eq(\boldsymbol{\alpha}, \mathbf{z}_j[i+1..])$ can be maintained dynamically; the value $eq(X, \mathbf{z}_j[i])$ can be computed in time $O(1)$. Moreover, $eq(\mathbf{z}_\ell[1..i-1], \mathbf{z}_j[1..i-1])$ equals 1 if $\mathbf{z}_\ell[1..i-1] = \mathbf{z}_j[1..i-1]$ and equals 0 otherwise. In summary, all non-zero values $\{\tilde{eq}^*(\mathbf{c}_\ell, X, \boldsymbol{\alpha})\}_{\ell \in [k]}$ can be computed in a batch in time $O(k)$. Therefore for each $x_i \in \{0, 1, 2\}$, one can evaluate $r_i(x_i)$ from evaluations $\{f(\mathbf{c}_\ell, x_i, \boldsymbol{\alpha})\}_{\ell \in [k]}$ in time $O(k)$, by evaluating $\{\tilde{eq}^*(\mathbf{c}_\ell, x_i, \boldsymbol{\alpha})\}_{\ell \in [k]}$ first and computing the inner product between $(\tilde{eq}^*(\mathbf{c}_\ell, x_i, \boldsymbol{\alpha}))_{\ell \in [k]}$ and $(f(\mathbf{c}_\ell, x_i, \boldsymbol{\alpha}))_{\ell \in [k]}$.

Applications to Commit-and-Prove SNARKs. Our batching scheme is helpful for building Commit-and-Prove SNARKs (CP-SNARKs) from multilinear commitments. In the setting of CP-SNARKs, given two commitments C_f, C_g that commit to vectors $\mathbf{f} \in \mathbb{F}^n, \mathbf{g} \in \mathbb{F}^m$ ($m \leq n$), and given two sets $I_f \subseteq [n], I_g \subseteq [m]$, one needs to prove that the values of $\mathbf{f}(I_f)$ is consistent with $\mathbf{g}(I_g)$. This problem can be solved using a variant of our special batching scheme with complexity $O(n)$.

For simplicity suppose that $n = m$,⁸ and we assume w.l.o.g that $n = 2^\mu$. The idea is to view \mathbf{f}, \mathbf{g} as the evaluations of polynomials $f, g \in \mathcal{F}_\mu^{(\leq 1)}$ on the boolean hypercube B_μ . Then the commitments C_f, C_g can be instantiated with multilinear commitments to polynomials f, g respectively. The relation that $\mathbf{f}(I_f) = \mathbf{g}(I_g)$ is a slightly more general version of the batching relation: let $k = |I_f| = |I_g|$, it is equivalent to prove that $f(\mathbf{z}_i) = g(\mathbf{u}_i)$ for all $i \in [k]$, where $\mathbf{z}_i, \mathbf{u}_i \in B_\mu$ map to the i -th index of set I_f, I_g respectively.

Similar to equation (9), we can define a sumcheck relation

$$\begin{aligned}&\sum_{i \in [k]} eq(\mathbf{t}, \langle i \rangle) \cdot \left[\left(\sum_{\mathbf{b} \in B_\mu} f(\mathbf{b}) \cdot eq(\mathbf{b}, \mathbf{z}_i) \right) - \left(\sum_{\mathbf{b} \in B_\mu} g(\mathbf{b}) \cdot eq(\mathbf{b}, \mathbf{u}_i) \right) \right] \\ &= \sum_{\mathbf{b} \in B_\mu} f(\mathbf{b}) \left(\sum_{i \in [k]} eq(\mathbf{t}, \langle i \rangle) \cdot eq(\mathbf{b}, \mathbf{z}_i) \right) - \sum_{\mathbf{b} \in B_\mu} g(\mathbf{b}) \left(\sum_{i \in [k]} eq(\mathbf{t}, \langle i \rangle) \cdot eq(\mathbf{b}, \mathbf{u}_i) \right) = 0,\end{aligned}$$

⁸the same technique applies for $n \neq m$

which is essentially a sumcheck for the degree-2 polynomial $h := f \cdot e\tilde{q}_f - g \cdot e\tilde{q}_g$ on set B_μ , where

$$e\tilde{q}_f(\mathbf{X}) := \sum_{i \in [k]} eq(\mathbf{t}, \langle i \rangle) \cdot eq(\mathbf{X}, \mathbf{z}_i), \quad e\tilde{q}_g(\mathbf{X}) := \sum_{i \in [k]} eq(\mathbf{t}, \langle i \rangle) \cdot eq(\mathbf{X}, \mathbf{u}_i).$$

We can use the same sumcheck algorithm underlying the special batching scheme. The complexity is $O(2^\mu)$. The CP-SNARK proving is then reduced to two PCS openings, one for commitment C_f and one for C_g .

4 HyperPlonk: Plonk on the boolean hypercube

Equipped with the building blocks in Section 3, we now describe the Polynomial IOP for HyperPlonk. In Section 4.1, we introduce $\mathcal{R}_{\text{PLONK}}$ — an indexed relation on the boolean hypercube that generalizes the vanilla Plonk constraint system [44]. We present a Polynomial IOP protocol for $\mathcal{R}_{\text{PLONK}}$ and analyze its security and efficiency in Section 4.2.

4.1 Constraint systems

Notation. For any $m \in \mathbb{Z}$ and $i \in [0, 2^m]$, we use $\langle i \rangle_m = \mathbf{v} \in B_m$ to denote the m -bit binary representation of i , that is, $i = \sum_{j=1}^m \mathbf{v}_j \cdot 2^{j-1}$.

Definition 4.1 (HyperPlonk indexed relation). *Fix public parameters $\text{gp} := (\mathbb{F}, \ell, n, \ell_w, \ell_q, f)$ where \mathbb{F} is the field, $\ell = 2^\nu$ is the public input length, $n = 2^\mu$ is the number of constraints, $\ell_w = 2^{\nu_w}$, $\ell_q = 2^{\nu_q}$ are the number of witnesses and selectors per constraint⁹, and $f : \mathbb{F}^{\ell_q + \ell_w} \rightarrow \mathbb{F}$ is an algebraic map with degree d . The indexed relation $\mathcal{R}_{\text{PLONK}}$ is the set of all tuples*

$$(\mathbf{i}; \mathbf{x}; \mathbf{w}) = ((q, \sigma); (p, [[w]]); w),$$

where $\sigma : B_{\mu+\nu_w} \rightarrow B_{\mu+\nu_w}$ is a permutation, $q \in \mathcal{F}_{\mu+\nu_q}^{(\leq 1)}$, $p \in \mathcal{F}_{\mu+\nu}^{(\leq 1)}$, $w \in \mathcal{F}_{\mu+\nu_w}^{(\leq 1)}$, such that

- the wiring identity is satisfied, that is, $(\sigma; ([[w]]), [[w]]); w \in \mathcal{R}_{\text{PERM}}$ (Definition 3.5);
- the gate identity is satisfied, that is, $(([[\tilde{f}]]) ; \tilde{f}) \in \mathcal{R}_{\text{ZERO}}$ (Definition 3.2), where the virtual polynomial $\tilde{f} \in \mathcal{F}_\mu^{(\leq d)}$ is defined as

$$\tilde{f}(\mathbf{X}) := f(q(\langle 0 \rangle_{\nu_q}, \mathbf{X}), \dots, q(\langle \ell_q - 1 \rangle_{\nu_q}, \mathbf{X}), w(\langle 0 \rangle_{\nu_w}, \mathbf{X}), \dots, w(\langle \ell_w - 1 \rangle_{\nu_w}, \mathbf{X})); \quad (11)$$

- the public input is consistent with the witness, that is, the public input polynomial $p \in \mathcal{F}_\nu^{(\leq 1)}$ is identical to $w(0^{\mu+\nu_w-\nu}, \mathbf{X}) \in \mathcal{F}_\nu^{(\leq 1)}$.

$\mathcal{R}_{\text{PLONK}}$ is general enough to capture many computational models. In the introduction, we reviewed how $\mathcal{R}_{\text{PLONK}}$ captures simple arithmetic circuits. $\mathcal{R}_{\text{PLONK}}$ can be used to capture higher degree circuits with higher arity and more complex gates, including state machine computations.

State machines. $\mathcal{R}_{\text{PLONK}}$ can model state machine computations, as shown by Gabizon and Williamson [43]. A state machine execution with $n - 1$ steps starts with an initial state $\text{state}_0 \in \mathbb{F}^k$ where k is the width of the state vector. In each step $i \in [0, n - 1]$, given input the previous state state_i and an online input $\text{inp}_i \in \mathbb{F}$, the state machine executes a transition function f and outputs

⁹We can pad zeroes if the actual number is not a power of two.

$\text{state}_{i+1} \in \mathbb{F}^w$. Let $\mathcal{T} := (\text{state}_0, \dots, \text{state}_{n-1})$ be the execution trace and define $\text{inp}_{n-1} := \perp$, we say that \mathcal{T} is valid for input $(\text{inp}_0, \dots, \text{inp}_{n-1})$ if and only if (i) $\text{state}_{n-1}[0] = 0^k$, and (ii) $\text{state}_{i+1} = f(\text{state}_i, \text{inp}_i)$ for all $i \in [0, n-1]$.

We build a HyperPlonk indexed relation that captures the state machine computation. W.l.o.g we assume that $n = 2^\mu$ for some $\mu \in \mathbb{N}$.¹⁰ Let ν_w be the minimal integer such that $2^{\nu_w} > 2k$. We also assume that there is a low-depth algebraic predicate f_* that captures the transition function f , that is, $f_*(\text{state}', \text{state}, \text{inp}) = 0$ if and only if $\text{state}' = f(\text{state}, \text{inp})$. For each $i \in [0, n]$:

- the online input at the i -th step is $\text{inp}_i := w(\langle 0 \rangle_{\nu_w}, \langle i \rangle_\mu)$;
- the input state of step i is $\text{state}_{\text{in},i} := [w(\langle 1 \rangle_{\nu_w}, \langle i \rangle_\mu), \dots, w(\langle k \rangle_{\nu_w}, \langle i \rangle_\mu)]$;
- the output state of step i is $\text{state}_{\text{out},i} := [w(\langle k+1 \rangle_{\nu_w}, \langle i \rangle_\mu), \dots, w(\langle 2k \rangle_{\nu_w}, \langle i \rangle_\mu)]$;
- the selector for step i is $\mathbf{q}_i := q(\langle i \rangle_\mu)$;
- the transition and output correctness are jointly captured by a high-degree algebraic map f' ,

$$f'(\text{inp}_i, \text{state}_{\text{in},i}, \text{state}_{\text{out},i}; \mathbf{q}_i) := (1 - \mathbf{q}_i) \cdot f_*(\text{state}_{\text{out},i}, \text{state}_{\text{in},i}, \text{inp}_i) + \mathbf{q}_i \cdot \text{state}_{\text{in},i}[0].$$

For all $i \in [0, n-1]$, we set $\mathbf{q}_i = 0$ so that $\text{state}_{i+1} = f_i(\text{state}_i, \text{inp}_i)$ if and only if

$$f'(\text{inp}_i, \text{state}_{\text{in},i}, \text{state}_{\text{out},i}; \mathbf{q}_i) = f_*(\text{state}_{\text{out},i}, \text{state}_{\text{in},i}, \text{inp}_i) = 0;$$

we set $\mathbf{q}_{n-1} = 1$ so that $\text{state}_{\text{in},n-1}[0] = 0$ if and only if

$$f'(\text{inp}_{n-1}, \text{state}_{\text{in},n-1}, \text{state}_{\text{out},n-1}; \mathbf{q}_{n-1}) = \text{state}_{\text{in},n-1}[0] = 0.$$

Note that we also need to enforce equality between the i -th input state and the $(i-1)$ -th output state for all $i \in [n-1]$. We achieve it by fixing a permutation σ and constraining that the witness assignment is invariant after applying the permutation.

Remark 4.1. *We can halve the size of the witness and remove the permutation check by using the polynomial shifting technique in Section 3.6. Specifically, we can remove output state columns $\text{state}_{\text{out},i}$ and replace it with $\text{state}_{\text{in},i+1}$ for every $i \in [0, n)$.*

4.2 The PolyIOP protocol

In this Section, we present a *multivariate* PIOP for $\mathcal{R}_{\text{PLOK}}$ that removes expensive FFTs.

Construction. Intuitively, the PIOP for $\mathcal{R}_{\text{PLOK}}$ builds on a zero-check PIOP (Section 3.2) for custom algebraic gates and a permutation-check PIOP (Section 3.5) for copy constraints; consistency between the public input and the online witness is achieved via a random evaluation check between the public input polynomial and the witness polynomial.

Let $\text{gp} := (\mathbb{F}, \ell, n, \ell_w, \ell_q, f)$ be the public parameters and let $d := \deg(f)$. For a tuple $(\mathbf{i}; \mathbf{x}; \mathbf{w}) = ((q, \sigma); (p, [[w]]); w)$, we describe the protocol in Figure 2.

¹⁰We can pad with dummy states if the number of steps is not a power of two.

Indexer. $\mathcal{I}(q, \sigma)$ calls the permutation PIOP indexer $([[s_{\text{id}}]], [[s_{\sigma}]]) \leftarrow \mathcal{I}_{\text{perm}}(\sigma)$. The oracle output is $([[q]], [[s_{\text{id}}]], [[s_{\sigma}]])$, where $q \in \mathcal{F}_{\mu+\nu_q}^{(\leq 1)}$, $s_{\text{id}}, s_{\sigma} \in \mathcal{F}_{\mu+\nu_w}^{(\leq 1)}$.

The protocol. $\mathcal{P}(\mathbf{gp}, \mathbf{i}, p, w)$ and $\mathcal{V}(\mathbf{gp}, p, [[q]], [[s_{\text{id}}]], [[s_{\sigma}]])$ run the following protocol.

1. \mathcal{P} sends \mathcal{V} the witness oracle $[[w]]$ where $w \in \mathcal{F}_{\mu+\nu_w}^{(\leq 1)}$.
2. \mathcal{P} and \mathcal{V} run a PIOP for the gate identity, which is a zero-check PIOP (Section 3.2) for $([[\tilde{f}]], \tilde{f}) \in \mathcal{R}_{\text{ZERO}}$ where $\tilde{f} \in \mathcal{F}_{\mu}^{(\leq d)}$ is as defined in Equation 11.
3. \mathcal{P} and \mathcal{V} run a PIOP for the wiring identity, which is a permutation PIOP (Section 3.5) for $(\sigma; ([[w]]), [[w]]); (w, w)) \in \mathcal{R}_{\text{PERM}}$.
4. \mathcal{V} checks the consistency between witness and public input. It samples $\mathbf{r} \xleftarrow{\$} \mathbb{F}^{\nu}$, queries $[[w]]$ on input $(\langle 0 \rangle_{\mu+\nu_w-\nu}, \mathbf{r})$, and checks $p(\mathbf{r}) \stackrel{?}{=} w(\langle 0 \rangle_{\mu+\nu_w-\nu}, \mathbf{r})$.

Figure 2: PIOP for $\mathcal{R}_{\text{PLOONK}}$.

Theorem 4.1. Let $\mathbf{gp} := (\mathbb{F}, \ell, n, \ell_w, \ell_q, f)$ be the public parameters where $\ell_w, \ell_q = O(1)$ are some constants. Let $d := \deg(f)$. The construction in Figure 2 is a multivariate PolyIOP for relation $\mathcal{R}_{\text{PLOONK}}$ (Definition 4.1) with soundness error $\mathcal{O}\left(\frac{2^{\mu}+d\mu}{|\mathbb{F}|}\right)$ and the following complexity:

- the prover time is $\mathbf{tp}_{\text{plonk}}^{\mathbf{gp}} = \mathcal{O}(nd \log^2 d)$;
- the verifier time is $\mathbf{tv}_{\text{plonk}}^{\mathbf{gp}} = \mathcal{O}(\mu + \ell)$;
- the query complexity is $\mathbf{q}_{\text{plonk}}^{\mathbf{gp}} = 2\mu + 4 + \log \ell_w$, that is, $2\mu + \log \ell_w$ univariate oracle queries, 3 multilinear oracle queries, and 1 query to the virtual polynomial \tilde{f} .
- the round complexity and the number of proof oracles is $\mathbf{rc}_{\text{plonk}}^{\mathbf{gp}} = 2\mu + 1 + \nu_w$;
- the number of field elements sent by the prover is $\mathbf{nf}_{\text{plonk}}^{\mathbf{gp}} = 2\mu$;
- the size of the proof oracles is $\mathbf{pl}_{\text{plonk}}^{\mathbf{gp}} = \mathcal{O}(n)$; the size of the witness is $n\ell_w$.

Remark 4.2. Two separate sumcheck PIOPs are underlying the HyperPlonk PIOP. We can batch the two sumchecks into one by random linear combination. The optimized protocol has round complexity $\mu + 1 + \log \ell_w$, and the number of field elements sent by the prover is μ . The query complexity $\mu + 3 + \log \ell_w$, that is, $\mu + \log \ell_w$ univariate queries, 2 multilinear queries, and 1 queries to the virtual polynomial \tilde{f} .

Remark 4.3. The prover's memory consumption is linear to the number of constraints. For space-bounded provers, we can split the proving work to multiple parallel parties or apply the techniques from [22] to obtain a space-efficient prover with quasilinear proving time. We leave concrete specifications of space-efficient HyperPlonk provers as future work.

Lemma 4.2. The PIOP in Figure 2 is perfectly complete.

Proof. For any $((q, \sigma); (p, [[w]]); w) \in \mathcal{R}_{\text{PLOONK}}$, by Definition 4.1, it holds that

- $([[\tilde{f}]]; \tilde{f}) \in \mathcal{R}_{\text{ZERO}}$, thus \mathcal{V} passes the check in Step 2 as the ZeroCheck PIOP is complete;
- $(\sigma; ([[w]]), [[w]]); w) \in \mathcal{R}_{\text{PERM}}$, thus \mathcal{V} passes the check in Step 3 as the permutation PIOP is complete;
- the public input polynomial $p \in \mathcal{F}_\nu^{(\leq 1)}$ is identical to $w(0^{\mu+\nu_w-\nu}, \mathbf{X}) \in \mathcal{F}_\nu^{(\leq 1)}$, thus their evaluations are always the same, and \mathcal{V} passes the check in Step 4.

In summary, the lemma holds as desired. \square

Lemma 4.3. *Let $\text{gp} := (\mathbb{F}, \ell = 2^\nu, n = 2^\mu, \ell_w = 2^{\nu_w}, \ell_q, f)$ be the public parameters and let $d := \deg(f)$. The PIOP in Figure 2 has soundness error*

$$\delta_{\text{plonk}}^{\text{gp}} := \max \left\{ \delta_{\text{zero}}^{d,\mu}, \delta_{\text{perm}}^{1,\mu+\nu_w}, \frac{\nu}{|\mathbb{F}|} \right\}.$$

Proof. For any $((q, \sigma); (p, [[w]])) \notin \mathcal{L}(\mathcal{R}_{\text{PLOONK}})$, that is, $((q, \sigma); (p, [[w]]); w) \notin \mathcal{R}_{\text{PLOONK}}$, at least one of the following conditions holds:

- $([[\tilde{f}]]; \tilde{f}) \notin \mathcal{R}_{\text{ZERO}}$;
- $(\sigma; ([[w]]), [[w]]); w) \notin \mathcal{R}_{\text{PERM}}$;
- $p(\mathbf{X}) \neq w(0^{\mu+\nu_w-\nu}, \mathbf{X})$;

In the first condition, the probability that \mathcal{V} passes the ZeroCheck in Step 2 is at most $\delta_{\text{zero}}^{d,\mu}$; in the second condition, the probability that \mathcal{V} passes the permutation check in Step 3 is at most $\delta_{\text{perm}}^{1,\mu+\nu_w}$; in the last condition, by Lemma 2.2, \mathcal{V} passes the evaluation check in Step 4 with probability at most $\nu/|\mathbb{F}|$. In summary, for any $((q, \sigma); (p, [[w]]); w) \notin \mathcal{R}_{\text{PLOONK}}$, the probability that \mathcal{V} accepts is at most $\max\{\delta_{\text{zero}}^{d,\mu}, \delta_{\text{perm}}^{1,\mu+\nu_w}, \nu/|\mathbb{F}|\}$ as claimed. \square

Zero knowledge. We refer to Appendix A for the zero-knowledge version of the HyperPlonk PIOP.

5 HyperPlonk+: HyperPlonk with Lookup Gates

This section illustrates how to integrate lookup gates into the HyperPlonk constraint system. Then we present and analyze a Polynomial IOP protocol for the extended relation.

5.1 Constraint systems

The HyperPlonk+ indexed relation $\mathcal{R}_{\text{PLOONK+}}$ is built on $\mathcal{R}_{\text{PLOONK}}$ (Definition 4.1). The difference is that $\mathcal{R}_{\text{PLOONK+}}$ further enables a set of *non-algebraic* constraints enforcing that some function over the witness values belongs to a preprocessed table. We illustrate via a simple example. Suppose we capture a fan-in-2 circuit with n addition/multiplication gates using relation $\mathcal{R}_{\text{PLOONK}}$. We need to further constrain that for a subset of gates, the sum of two input wires should be in the range $[0, \dots, B]$. What we can do is to set up a preprocessed table $\text{table} = \{0, 1, \dots, B\}$ and a selector $q_{\text{lk}} \in \mathbb{F}^n$ so that for every $i \in [n]$, $q_{\text{lk}}(i) = 1$ if the i -th gate has a range-check, and $q_{\text{lk}}(i) = 0$

otherwise. Then we prove a lookup relation that for all $i \in [n]$, the value $q_{lk}(i) \cdot (w_1(i) + w_2(i))$ is in **table**, where $w_1(i), w_2(i)$ are the first and the second input wire of gate i .

We generalize the idea above and enable enforcing *arbitrary algebraic functions* (over the selectors and witnesses) to be in the table. Namely, the index further setups an algebraic functions f_{lk} . Each constraint is of the form

$$f_{lk}(q_{lk}(\langle 0 \rangle, \langle i \rangle), \dots, q_{lk}(\langle \ell_{lk} - 1 \rangle, \langle i \rangle), w(\langle 0 \rangle, \langle i \rangle), \dots, w(\langle \ell_w - 1 \rangle, \langle i \rangle)) \in \text{table}$$

where ℓ_{lk} is the number of selectors, ℓ_w is the number of witness wires and $\langle i \rangle$ is the binary representation of i . Note that the constraint in the previous paragraph is a special case where $f_{lk} = q_{lk}(i) \cdot (w_1(i) + w_2(i))$. We formally define the relation below.

Definition 5.1 (HyperPlonk+ indexed relation). *Let $\mathbf{gp}_1 := (\mathbb{F}, \ell, n, \ell_w, \ell_q, f)$ be the public parameters for relation $\mathcal{R}_{\text{PLOONK}}$ (Definition 4.1). Let $\mathbf{gp}_2 := (\ell_{lk}, f_{lk})$ be the additional public parameters where $\ell_{lk} = 2^{\nu_{lk}}$ is the number of lookup selectors and $f_{lk} : \mathbb{F}^{\ell_{lk} + \ell_w} \rightarrow \mathbb{F}$ is an algebraic map. The indexed relation $\mathcal{R}_{\text{PLOONK}+}$ is the set of all triples*

$$(\mathbf{i}; \mathbf{x}; \mathbf{w}) = ((\mathbf{i}_1, \mathbf{i}_2); (p, [[w]]); w)$$

where $\mathbf{i}_2 := (\text{table} \in \mathbb{F}^{n-1}, q_{lk} \in \mathcal{F}_{\mu+\nu_{lk}}^{(\leq 1)})$ such that

- $(\mathbf{i}_1; \mathbf{x}; \mathbf{w}) \in \mathcal{R}_{\text{PLOONK}}$;
- there exists $\text{addr} : B_\mu \rightarrow [1, 2^\mu]$ such that $(\text{table}; [[g]]; (g, \text{addr})) \in \mathcal{R}_{\text{LOOKUP}}$ (Definition 3.6), where $g \in \mathcal{F}_\mu^{(\leq \deg(f_{lk}))}$ is defined as

$$g(\mathbf{X}) := f_{lk}(q_{lk}(\langle 0 \rangle_{\nu_{lk}}, \mathbf{X}), \dots, q_{lk}(\langle \ell_{lk} - 1 \rangle_{\nu_{lk}}, \mathbf{X}), w(\langle 0 \rangle_{\nu_w}, \mathbf{X}), \dots, w(\langle \ell_w - 1 \rangle_{\nu_w}, \mathbf{X})). \quad (12)$$

Remark 5.1 (Supporting vector lookups). *We can generalize $\mathcal{R}_{\text{PLOONK}+}$ to support vector lookups where each “element” in the table is a vector rather than a single field element. Let $k \in \mathbb{N}$ be the length of the vector. The lookup table is $\text{table} \in \mathbb{F}^{k \times (n-1)}$; the lookup function $f_{lk} : \mathbb{F}^{2^{\nu_{lk}} + 2^{\nu_w}} \rightarrow \mathbb{F}^k$ is an algebraic map that outputs k field elements.*

Remark 5.2 (Supporting multiple tables). *We can generalize $\mathcal{R}_{\text{PLOONK}+}$ to support multiple lookup tables. In particular, the index \mathbf{i}_2 can specify $k > 1$ lookup tables $\text{table}_1, \dots, \text{table}_k$ and k lookup functions $f_{lk}^{(1)}, \dots, f_{lk}^{(k)}$; and we require that all of the k lookup relations hold.*

5.2 The PolyIOP protocol

Construction. The PIOP for $\mathcal{R}_{\text{PLOONK}+}$ is a combination of the PIOP for $\mathcal{R}_{\text{PLOONK}}$ and the PIOP for a lookup relation (Section 3.6). Let $\mathbf{gp} := (\mathbf{gp}_1, \mathbf{gp}_2)$ be the public parameters where $\mathbf{gp}_1 := (\mathbb{F}, \ell, n, \ell_w, \ell_q, f)$ and $\mathbf{gp}_2 := (\ell_{lk}, f_{lk})$. We denote $d_{lk} := \deg(f_{lk})$. For a tuple $(\mathbf{i}; \mathbf{x}; \mathbf{w}) = ((\mathbf{i}_1, \mathbf{i}_2); (p, [[w]]); w)$ where $\mathbf{i}_2 := (\text{table} \in \mathbb{F}^{n-1}, q_{lk} \in \mathcal{F}_{\mu+\nu_{lk}}^{(\leq 1)})$ we describe the protocol in Figure 3.

Theorem 5.1. *Let $\mathbf{gp} := (\mathbf{gp}_1, \mathbf{gp}_2)$ be the public parameters, where $\mathbf{gp}_1 := (\mathbb{F}, \ell, n, \ell_w, \ell_q, f)$ and $\ell_w, \ell_q = O(1)$ are some constants; $\mathbf{gp}_2 := (\ell_{lk}, f_{lk})$ and $\ell_{lk} = O(1)$ is some constant. Let $d' := \max(\deg(f), \deg(f_{lk}))$ and let g be the polynomial defined in Equation 12. The construction in Figure 3 is a multivariate PolyIOP for relation $\mathcal{R}_{\text{PLOONK}+}$ with soundness error $\mathcal{O}\left(\frac{2^\mu + d'\mu}{|\mathbb{F}|}\right)$ and the following complexity:*

Indexer. $\mathcal{I}(\mathbf{i}_1, \mathbf{i}_2 = (\text{table}, q_{\text{lk}}))$ calls the HyperPlonk PIOP indexer $\mathbf{vp}_{\text{plonk}} \leftarrow \mathcal{I}_{\text{plonk}}(\mathbf{i}_1)$, and calls the Lookup PIOP indexer $\mathbf{vp}_t \leftarrow \mathcal{I}_{\text{lkup}}(\text{table})$. The oracle output is $\mathbf{vp} := ([[q_{\text{lk}}]], \mathbf{vp}_t, \mathbf{vp}_{\text{plonk}})$.

The protocol. $\mathcal{P}(\mathbf{gp}, \mathbf{i}, p, w)$ and $\mathcal{V}(\mathbf{gp}, p, \mathbf{vp})$ run the following protocol.

1. \mathcal{P} sends \mathcal{V} the witness oracle $[[w]]$ where $w \in \mathcal{F}_{\mu+\nu_w}^{(\leq 1)}$.
2. Run a HyperPlonk PIOP (Section 4.2) for $(\mathbf{i}_1; \mathbf{x}; \mathbf{w}) \in \mathcal{R}_{\text{PLONK}}$.
3. Run a lookup PIOP (Section 3.6) for $(\text{table}; [[g]]) \in \mathcal{L}(\mathcal{R}_{\text{LOOKUP}})$ where $g \in \mathcal{F}_{\mu}^{(\leq d_{\text{lk}})}$ is as defined in Equation 12.

Figure 3: PIOP for $\mathcal{R}_{\text{PLONK+}}$.

- The prover time is $\mathbf{tp}_{\text{plonk+}}^{\text{gp}} = \mathbf{tp}_{\text{plonk}}^{\text{gp}_1} + \mathbf{tp}_{\text{lkup}}^g = \mathcal{O}(nd' \log^2 d') \mathbb{F}\text{-ops}$.
- The verifier time is $\mathbf{tv}_{\text{plonk+}}^{\text{gp}} := \mathbf{tv}_{\text{plonk}}^{\text{gp}_1} + \mathbf{tv}_{\text{lkup}}^g = \mathcal{O}(\mu + \ell) \mathbb{F}\text{-ops}$.
- The query complexity is $\mathbf{q}_{\text{plonk+}}^{\text{gp}} = \mathbf{q}_{\text{plonk}}^{\text{gp}_1} + \mathbf{q}_{\text{lkup}}^g = 3\mu + 7 + \log \ell_w$, that is, 3 μ +7+ $\log \ell_w$ univariate oracle queries, 5 multilinear oracle queries, 1 query to the virtual polynomial \tilde{f} , and 1 query to the virtual polynomial g defined in Equation 12.
- The round complexity and the number of proof oracles is $\mathbf{rc}_{\text{plonk+}}^{\text{gp}} = \mathbf{rc}_{\text{plonk}}^{\text{gp}_1} + \mathbf{rc}_{\text{lkup}}^g = 3\mu + 3 + \log \ell_w$.
- The number of field elements sent by \mathcal{P} is 3μ .
- The size of the proof oracles is $\mathcal{O}(n)$; the size of the witness is $n\ell_w$.

Remark 5.3. Similar to Remark 4.2, there are 3 separate sumcheck PIOPs underlying the HyperPlonk+ PIOP. By random linear combination, we can batch the 3 sumchecks into a single one. The optimized protocol has query complexity $\mu + 7 + \log \ell_w$, round complexity $\mu + 3 + \log \ell_w$, and the number of field elements sent by the prover is μ .

Remark 5.4. We emphasize that the PolyIOP for $\mathcal{R}_{\text{PLONK+}}$ naturally works for the more general versions of $\mathcal{R}_{\text{PLONK+}}$ that involve vector lookups (Remark 5.1) or multiple tables (Remark 5.2). Because we can transform the problem of building PIOPs for the more general relations to the problem of building PIOPs for $\mathcal{R}_{\text{PLONK+}}$ by applying the randomization and domain separation techniques in Section 4 of [42].

Lemma 5.2. The PIOP in Figure 3 is perfectly complete.

Proof. For any $((\mathbf{i}_1, \text{table}, q_{\text{lk}}); (p, [[w]]); w) \in \mathcal{R}_{\text{PLONK+}}$, by Definition 5.1, it holds that

- $(\mathbf{i}_1; \mathbf{x}; \mathbf{w}) \in \mathcal{R}_{\text{PLONK}}$, thus \mathcal{V} passes the check in Step 2 as the HyperPlonk PIOP is complete;
- $(\text{table}; [[g]]) \in \mathcal{L}(\mathcal{R}_{\text{LOOKUP}})$, thus \mathcal{V} passes the check in Step 3 as the lookup PIOP is complete.

In summary, the lemma holds as desired. \square

Lemma 5.3. Let $\text{gp} := (\text{gp}_1, \text{gp}_2)$ be the public parameters. Let $n = 2^\mu \in \text{gp}_1$ denote the number of constraints. Let $f_{lk} \in \text{gp}_2$ be the lookup gate map and set $d_{lk} := \deg(f_{lk})$. The PIOP in Figure 3 has soundness error

$$\delta_{\text{plonk+}}^{\text{gp}} := \max \left\{ \delta_{\text{plonk}}^{\text{gp}_1}, \delta_{\text{lkup}}^{d_{lk}, \mu} \right\}.$$

Proof. For any $((\mathbf{i}_1, \mathbf{table}, q_{lk}); (p, [[w]])) \notin \mathcal{L}(\mathcal{R}_{\text{PLONK+}})$, that is, $((\mathbf{i}_1, \mathbf{table}, q_{lk}); (p, [[w]]); w) \notin \mathcal{R}_{\text{PLONK}}$, at least one of the following conditions holds:

- $(\mathbf{i}_1; \mathbf{x}; w) \notin \mathcal{R}_{\text{PLONK}}$;
- $(\mathbf{table}; [[g]]) \notin \mathcal{L}(\mathcal{R}_{\text{LOOKUP}})$, where $g \in \mathcal{F}_\mu^{(\leq d_{lk})}$ is as defined in Equation 12.

For the first case, the probability that \mathcal{V} accepts in the HyperPlonk PIOP is at most $\delta_{\text{plonk}}^{\text{gp}_1}$; for the second case, the probability that \mathcal{V} passes the lookup check is at most $\delta_{\text{lkup}}^{d_{lk}, \mu}$. Thus for every instance not in $\mathcal{L}(\mathcal{R}_{\text{PLONK+}})$, the probability that \mathcal{V} accepts is at most $\max(\delta_{\text{plonk}}^{\text{gp}_1}, \delta_{\text{lkup}}^{d_{lk}, \mu})$. \square

Zero knowledge. We refer to Appendix A for the zero-knowledge version of the HyperPlonk+ PIOP.

6 Instantiation and evaluation

6.1 Implementation

We implement HyperPlonk as a library using about 5500 lines of RUST. Figure 4 highlights the building blocks contributing to our HyperPlonk code base. Our backend is built on top of the Arkworks [4]. Specifically, we adopted the finite field, elliptic curve, and polynomial libraries from this project. We then build our PIOP libraries, including our core zero and permutation checks, and use merlin transcript [38] to turn it into a non-interactive protocol. We also implement a multilinear KZG commitment scheme variant that is compatible with our batch-evaluation PIOP.

Our implementation is highly modular: one may switch between different elliptic curves, other multilinear polynomial commitment schemes and various circuit frontends within our framework.

The current version of our code base has a few limitations, which do not affect the benchmarks reported in this section. Firstly, it is built for benchmarking purposes with mock circuits, but we aim to support Halo2 and Jellyfish arithmetization as frontends. Secondly, we are not yet supporting lookup tables and thus HyperPlonk+.

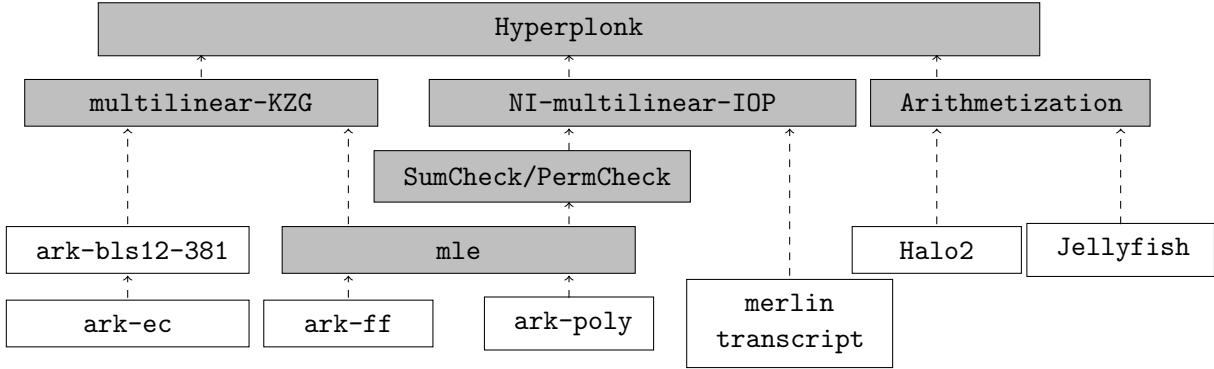


Figure 4: Stack of libraries comprising HyperPlonk. The components in grey we implemented ourselves.

6.2 Evaluation

We benchmark HyperPlonk on an AMD 5900x CPU with 12 cores and 24 threads 3.2 GHZ and 32 GB of RAM, running Ubuntu 20.04 for all tests except for the multi-threading benchmark, which we conducted over an AWS EC2 instance with 32 cores (AMD EPYC 7R13 at 2.65 GHz) and 128 GB of RAM.

Cost breakdown We present a cost breakdown of HyperPlonk’s prover cost. As we see in Figure 5a, the majority of the computation is spent on committing and (batch) opening the commitment; the actual time spent on the information-theoretic PIOPs (ZeroCheck::IOP and PermCheck::IOP) is less than 10%. We note that our implementation only batches polynomial openings with the same number of variables. Implementing batching across a different number of variables should lead to a further increase in prover performance. Figure 5b gives another breakdown which shows that around 50% of the time is spent on multi-exponentiations for both committing and evaluating multi-linear polynomials, while the next largest subroutine is SumCheck which uses 25% of the time. We note that both multi-exponentiations and sumchecks are highly parallelizable and hardware-friendly, thus we expect further performance improvement on special-purpose hardware (e.g. GPUs).

It is also worth noting that HyperPlonk never requires the explicit multiplication of polynomials. This enables high-degree custom gates for HyperPlonk.

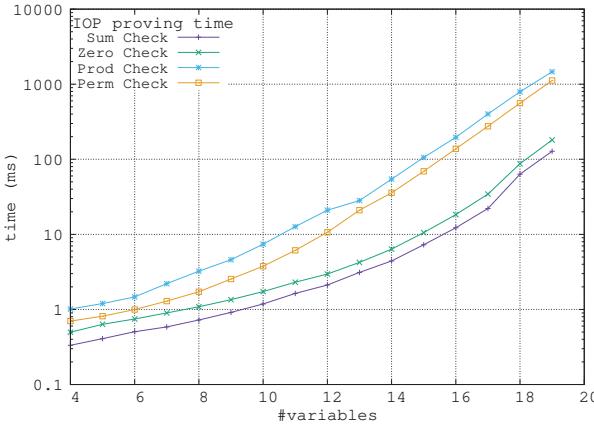


Figure 6: PIOP proving cost

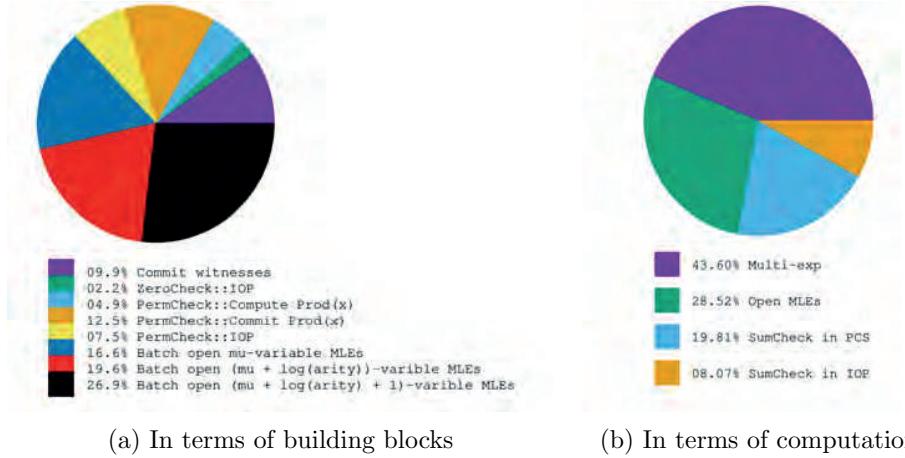
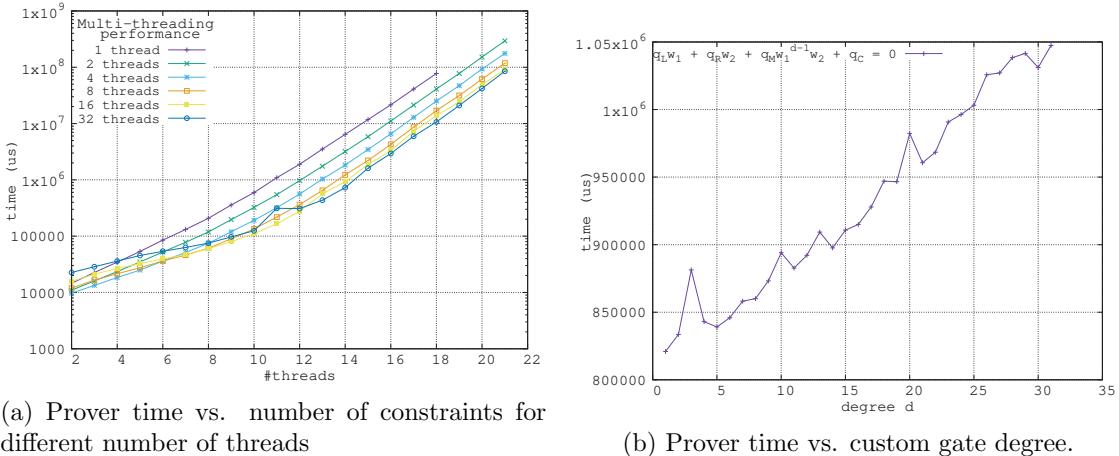


Figure 5: Cost breakdown for vanilla $\mathcal{R}_{\text{PLONK}}$ with 2^{18} constraints.

Figure 6 presents the performance of our non-interactive PIOPs, the core cost of the **HyperPlonk** PIOP, whereas the rest are PCS related and vary drastically with the actual commitment scheme. Here we see that the proving time (as a log-log plot) grows almost linearly with the number of variables. Overall, for 19 variables, the PIOP prover time is still less than 1 second.

6.3 MultiThreading Performance

A key advantage of **HyperPlonk** is that it does not rely on FFT algorithms that are less parallelizable. Indeed, in Figure 7a we observe an almost linear improvement when num of threads is small. We also observe that with low parallelization, the prover's run time is linear in the number of gates. For example, increase from a single thread to two threads, the prover time is reduced by 40% on average. In contrast, for high parallelization, we are only able to get marginal improvement. We assume this is mainly a limitation of the implementation.



(a) Prover time vs. number of constraints for different number of threads

(b) Prover time vs. custom gate degree.

Figure 7

6.4 High degree gates

It has been shown in VeriZexe [79] that custom gates, even at degree 5, allow for significant improvement of circuit size and prover time. For example, one may perform an elliptic curve group addition with just two gates; while a naive version may require 10+ gates. The better expressibility of high-degree gates enables VeriZexe to improve 9x of prover time over the previous state-of-the-art [23].

However, in a univariate Plonk system, such as [42, 64], high-degree custom gates increase the size of the required FFTs as well as the number of group operations. This limits their utility as they get larger. In comparison, in HyperPlonk, high-degree only affects the number of field operations. Our benchmark result in Figure 7b validates this observation and shows that the prover time from a degree 2 gate to a degree 32 gate only increases by 30%. These more expressive gates can significantly reduce the number of gates in the circuit which more than offsets the added cost.

6.5 Comparisons

We compare our scheme with both Jellyfish Plonk [69], and Spartan [66]. We have presented data points for a few typical applications in Table 4. Figure 8 additionally gives a detailed comparison for various constraint sizes.

Jellyfish is a highly optimized implementation of Plonk with lookup arguments. It is the state-of-the-art plonk prove system that uses Arkworks as the backend. Note that there are other implementations of Plonk, such as Halo2 [24]. The comparisons with those libraries are less illustrative as they use a different arithmetic backend.

Spartan is a multilinear ZKP system. Spartan's statements are written in Rank-1-Constraint-System (\mathcal{R}_{R1CS}), which is, in general, faster to prove but less expressive. As shown in Table 4, it requires more constraints in \mathcal{R}_{R1CS} than in \mathcal{R}_{PLONK} to express a same statement. E.g., a proof of knowledge of exponent for a 256-bits elliptic curve group element requires 3315 \mathcal{R}_{R1CS} constraints [61], while it reduces to 1870 and 783 for \mathcal{R}_{PLONK} and \mathcal{R}_{PLONK+} , respectively [79].

Note that a specific instantiation of Spartan uses inner product argument (IPA) for polynomial commitment. This removes the requirement for a universal setup, and allows for faster, non-

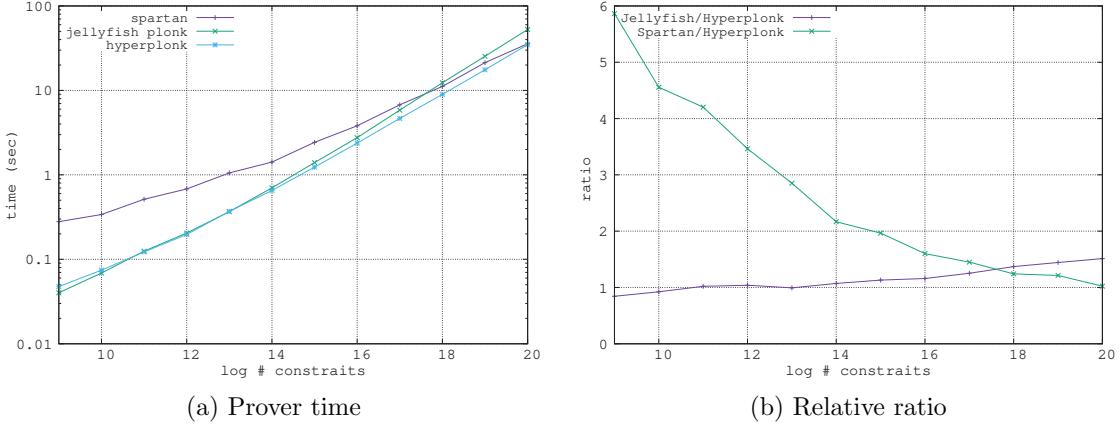


Figure 8: Comparison: Spartan vs Plonk vs HyperPlonk

pairing-friendly curves, such as curve25519 [15], with the tradeoff of larger verification time. For a fair comparison, we benchmarked a variant of Spartan, whose backend depends on Arkwork’s BLS12-381.

Application	$\mathcal{R}_{\text{R1CS}}$	Spartan	$\mathcal{R}_{\text{PLONK+}}$	Jellyfish	HyperPlonk
3-to-1 Rescue Hash	288 [1]	279 ms	144 [69]	20 ms	24 ms
PoK of Exponent	3315 [61]	681 s	783 [61]	69 ms	74 ms
ZCash circuit	2^{17} [53]	6.75 s	2^{15} [40]	1.40 s	1.23 s
Zexe’s recursive circuit	2^{22} [79]	2.4 min ^b	2^{17} [79]	5.83 s	4.66 s
Rollup of 50 private tx	2^{25}	20 min ^b	2^{20} [69]	52.7 s	34.9 s
zkEVM circuit ^a	N/A	N/A	2^{27}	2 hours ^{b,c}	1.3 hours ^{b,c}

Table 4: Prover runtime of Hyperplonk vs. Spartan[66] and the Jellyfish Plonk implementation for popular applications

^a So far, there have been no approaches to express zkEVM as an R1CS circuit. Common approaches rely heavily on lookup tables which require plonk+. ^b Estimations. ^c This assumes a linear scaling factor that is in favor of Jellyfish since we already observe a linear growth for log degree from 20 to 23.

Regarding prover time, our benchmark shows that HyperPlonk outperforms Jellyfish at 2^{14} constraints; the advantage grows when circuit size increases. This is mainly because FFTs scale worse than multi-exponentiations. HyperPlonk is faster than Spartan when constraint size is small and has similar performance when circuit size grows. We stress again that plonk+ is more expressive than $\mathcal{R}_{\text{R1CS}}$, and thus a fair comparison should be over the same application rather than the same size of constraints. Table 4 shows that HyperPlonk is $5 \sim 25$ x faster than Spartan in those applications.

7 Orion+: a linear-time multilinear PCS with constant proof size

Recently, Xie et al. [78] introduced a highly efficient multilinear polynomial commitment scheme called Orion. The prover time is strictly linear, that is, $O(2^\mu)$ field operations and hashes where μ

is the number of variables. For $\mu = 27$, it takes only 115 seconds to commit to a polynomial and compute an evaluation proof using a single thread on a consumer-grade desktop. The verifier time and proof size is $O_\lambda(\mu^2)$, which also improves the state-of-the-art [21, 48]. However, the concrete proof size is still unsatisfactory, e.g., for $\mu = 27$, the proof size is 6 MBs. In this section, we describe a variant of Orion PCS that enjoys similar proving complexity but has $O(\mu)$ proof size and verifier time, with good constants. In particular, for security parameter $\lambda = 128$ and $\mu = 27$, the proof size is less than 10KBs, which is $600\times$ smaller than Orion for $\mu = 27$.

This section is organized as follows. We start by reviewing the linear-time PCS from tensor product arguments [21, 48], which Orion builds upon, then we describe our techniques for shrinking the proof size. Finally, we analyze the security and complexity of the construction.

Linear-time PCS from tensor-product argument [21, 48]. Bootle, Chiesa, and Groth [21] propose an elegant scheme for building PCS with strictly linear-time provers. Golovnev et al. [48] later further simplify the scheme. Let $f \in \mathcal{F}_\mu^{(\leq 1)}$ be a multilinear polynomial where $f_b \in \mathbb{F}$ is the coefficient of $\mathbf{X}_b := \mathbf{X}_1^{b_1} \cdots \mathbf{X}_\mu^{b_\mu}$ for every $b \in B_\mu$. Denote by $n = 2^\mu$, $k = 2^\nu < 2^\mu$ and $m = n/k$, one can view the evaluation of f as a tensor product, that is,

$$f(\mathbf{X}) = \langle \mathbf{w}, \mathbf{t}_0 \otimes \mathbf{t}_1 \rangle \quad (13)$$

where $\mathbf{w} = (f_{\langle 0 \rangle}, \dots, f_{\langle n-1 \rangle})$, $\mathbf{t}_0 = (\mathbf{X}_{\langle 0 \rangle}, \mathbf{X}_{\langle 1 \rangle}, \dots, \mathbf{X}_{\langle k-1 \rangle})$ and $\mathbf{t}_1 = (\mathbf{X}_{\langle 0 \rangle}, \mathbf{X}_{\langle k \rangle}, \dots, \mathbf{X}_{\langle (m-1) \cdot k \rangle})$. Here $\langle i \rangle$ denotes the μ -bit binary representation of i . Let $E : \mathbb{F}^m \rightarrow \mathbb{F}^M$ be a linear encoding scheme, that is, a linear function whose image is a linear code (Definition 2.1). Golovnev et al. [48, §4.2] construct a PCS scheme as follows:

- **Commitment:** To commit a multilinear polynomial f with coefficients $\mathbf{w} \in \mathbb{F}^n$, the prover \mathcal{P} interprets \mathbf{w} as a $k \times m$ matrix, namely $\mathbf{w} \in \mathbb{F}^{k \times m}$, encodes \mathbf{w} 's rows, and obtains matrix $W \in \mathbb{F}^{k \times M}$ such that $W[i, :] = E(\mathbf{w}[i, :])$ for every $i \in [k]$. Then \mathcal{P} computes a Merkle tree commitment for each column of W and builds another Merkle tree T on top of the column commitments. The polynomial commitment C_f is the Merkle root of T .
- **Evaluation proof:** To prove that $f(\mathbf{z}) = y$ for some point $\mathbf{z} \in \mathbb{F}^\mu$ and value $y \in \mathbb{F}$, the prover \mathcal{P} translates \mathbf{z} to vectors $\mathbf{t}_0 \in \mathbb{F}^k$ and $\mathbf{t}_1 \in \mathbb{F}^m$ as above and proves that $\langle \mathbf{w}, \mathbf{t}_0 \otimes \mathbf{t}_1 \rangle = y$ (where $\mathbf{w} \in \mathbb{F}^{k \times m}$ is the message encoded and committed in C_f). To do so, \mathcal{P} does two things:
 - **Proximity check:** The prover shows that the matrix $W \in \mathbb{F}^{k \times M}$ committed by C_f is close to k codewords. Specifically, the verifier sends a random vector $\mathbf{r} \in \mathbb{F}^k$, the prover replies with a vector $\mathbf{y}_r := \mathbf{r} \cdot \mathbf{w} \in \mathbb{F}^m$ which is the linear combination of \mathbf{w} 's rows according to \mathbf{r} . The verifier checks that the encoding of \mathbf{y}_r , namely $E(\mathbf{y}_r) \in \mathbb{F}^M$, is close to $\mathbf{r} \cdot W$, the linear combination of W 's rows. This implies that the k rows of W are all close to codewords [48, §4.2].
 - **Consistency check:** The prover shows that $\langle \mathbf{w}, \mathbf{t}_0 \otimes \mathbf{t}_1 \rangle = y$ where $\mathbf{w} \in \mathbb{F}^{k \times m}$ is the k error-decoded messages from $W \in \mathbb{F}^{k \times M}$ committed in C_f . The scheme is similar to the proximity check except that we replace the random vector \mathbf{r} with \mathbf{t}_0 . After receiving the linearly combined vector $\mathbf{y}_0 \in \mathbb{F}^m$, the verifier further checks that $\langle \mathbf{y}_0, \mathbf{t}_1 \rangle = y$.

We describe the concrete PCS evaluation protocol below.

Protocol 1 (PCS evaluation [48]): The goal is to check that $\langle \mathbf{w}, \mathbf{t}_0 \otimes \mathbf{t}_1 \rangle = y$ (where $\mathbf{w} \in \mathbb{F}^{k \times m}$ is the message encoded and committed in C_f).

1. \mathcal{V} sends a random vector $\mathbf{r} \in \mathbb{F}^k$.
2. \mathcal{P} sends vector $\mathbf{y}_r, \mathbf{y}_0 \in \mathbb{F}^m$ where

$$\mathbf{y}_r = \sum_{i=1}^k \mathbf{r}_i \cdot \mathbf{w}[i, :], \text{ and } \mathbf{y}_0 = \sum_{i=1}^k \mathbf{t}_{0,i} \cdot \mathbf{w}[i, :],$$

where $\mathbf{w} \in \mathbb{F}^{k \times m}$ is the message matrix being encoded and committed.

3. \mathcal{V} sends \mathcal{P} a random subset $I \subseteq [M]$ with size $|I| = \Theta(\lambda)$.
4. \mathcal{P} opens the entire columns $\{W[:, j]\}_{j \in I}$ using Merkle proofs, where $W \in \mathbb{F}^{k \times M}$ is the row-wise encoded matrix. That is, \mathcal{P} outputs the column commitment h_j for every column $j \in I$, and provide the Merkle proof for h_j w.r.t. to Merkle root C_f .
5. \mathcal{V} checks that (i) the Merkle openings are correct w.r.t. C_f , and (ii) for all $j \in I$, it holds that

$$E(\mathbf{y}_r)_j = \langle \mathbf{r}, W[:, j] \rangle \text{ and } E(\mathbf{y}_0)_j = \langle \mathbf{t}_0, W[:, j] \rangle.$$

6. \mathcal{V} checks that $\langle \mathbf{y}_0, \mathbf{t}_1 \rangle = y$.

Note that by sampling a subset I with size $\Theta(\lambda)$ and checking that $\mathbf{r} \cdot W, \mathbf{t}_0 \cdot W$ are consistent with the encodings $E(\mathbf{y}_r), E(\mathbf{y}_0)$ on set I , the verifier is confident that $\mathbf{r} \cdot W, \mathbf{t}_0 \cdot W$ are indeed close to the encodings $E(\mathbf{y}_r), E(\mathbf{y}_0)$ with high probability. By setting $k = \sqrt{n}$, the prover takes $O(n)$ F-ops and hashes; the verifier time and proof size are both $O_\lambda(\sqrt{n})$. Orion describes an elegant code-switching scheme that reduces the proof size and verifier time down to $O_\lambda(\log^2(n))$. However, the concrete proof size is still large. Next, we describe a scheme that has much smaller proof.

Linear-time PCS with small proofs. Similar to Orion (and more generally, the proof composition technique [20, 21, 48]), instead of letting the verifier check the correctness of $\mathbf{y}_r, \mathbf{y}_0$ and the openings of the columns $W[:, j] \forall j \in I$, the prover can compute another (succinct) outer proof validating the correctness of $\mathbf{y}_r, \mathbf{y}_0, W[:, j]$. However, we need to minimize the outer proof's circuit complexity, which is non-trivial. Orion builds an efficient SNARK circuit that removes all of the hashing gadgets, with the tradeoff of larger proof size. We describe a variant of their scheme that minimizes the proof size without significantly increasing the circuit complexity.

Specifically, after receiving challenge vector $\mathbf{r} \in \mathbb{F}^k$, \mathcal{P} instead sends \mathcal{V} commitments C_r, C_0 to the messages $\mathbf{y}_r, \mathbf{y}_0$; after receiving \mathcal{V} 's random subset $I \subset [M]$, \mathcal{P} computes a SNARK proof for the following statement:

Statement 1 (PCS Eval verification):

- Witness: $\mathbf{y}_r, \mathbf{y}_0 \in \mathbb{F}^m, \{W[:, j]\}_{j \in I}$.
- Circuit statements:
 - C_r, C_0 are the commitments to $\mathbf{y}_r, \mathbf{y}_0$ respectively.
 - For all $j \in I$, it holds that
 - * $h_j = H(W[:, j])$ where H is a fast hashing scheme;
 - * $E(\mathbf{y}_r)_j = \langle \mathbf{r}, W[:, j] \rangle$ and $E(\mathbf{y}_0)_j = \langle \mathbf{t}_0, W[:, j] \rangle$.
 - $\langle \mathbf{y}_0, \mathbf{t}_1 \rangle = y$.
- Public output: $\{h_j\}_{j \in I}$, and C_r, C_0 .

Besides the SNARK proof, the prover also provides the openings of $\{h_j\}_{j \in I}$ with respect to the commitments C_f . Intuitively, the new protocol is “equivalent” to Protocol 1, because the SNARK witness $\{W[:, j]\}_{j \in I}$ and $\mathbf{y}_r, \mathbf{y}_0$ are identical to those committed in C_f, C_r, C_0 by the binding property of the commitments; and the SNARK does all of the verifier checks. Unfortunately, the scheme has the following drawbacks:

- Instantiating the commitments with Merkle trees leads to a large overhead on the proof size. In particular, the proof contains $|I|$ Merkle proofs, each with length $O(\log n)$. For 128-bit security, we need to set $|I| = 1568$, and the proof size is at least 1 MBs for $\mu = 20$.
- The random subset I varies for different evaluation instances. It is non-trivial to efficiently lookup the witness $\{E(\mathbf{y}_r)_j, E(\mathbf{y}_0)_j\}_{j \in I}$ in the circuit if the set I is dynamic (i.e. we need an efficient random access gadget).
- The circuit complexity is huge. In particular, the circuit is dominated by the commitments to $\mathbf{y}_r, \mathbf{y}_0$ and the hash commitments to $\{W[:, j]\}_{j \in I}$. This leads to $2m + k|I|$ hash gadgets in the circuit. Note that we can’t use circuit-friendly hash functions like Rescue [1] or Poseidon [49] because their running time is too slow to obtain a fast PCS prover. For $\mu = 26$, $k = m = \sqrt{n}$ and 128-bit security (where $|I| = 1568$), this leads to 13 millions hash gadgets where each hash takes hundreds to thousands of constraints, which is unaffordable.

We resolve the above issues via the following observations.

First, a large portion of the multilinear PCS evaluation proof is Merkle opening paths. We can shrink the proof size by replacing Merkle trees with multilinear PCS that enable efficient batch openings (Section 3.7). Specifically, in the committing phase, after computing the hashes of W ’s columns, instead of building another Merkle tree T of size $M = O(n/k)$ and set the Merkle root as the commitment, the prover can commit to the column hashes using a multilinear PCS (e.g. KZG). Though the KZG committing is more expensive, *the problem size has been reduced to $O(n/k)$, thus for sufficiently large k , the committing complexity is still approximately $O(n)$ F-ops*. A great advantage is that the batch opening proof for $\{h_j\}_{j \in I}$ consists of only $O(\log n)$ group/field elements, with good constant. Even better, when instantiating the outer proof with HyperPlonk(+), the openings can be batched with those in the outer SNARK and thus incur almost no extra cost in proof size.

Second, with Plookup, we can efficiently simulate random access in arrays in the SNARK circuit. For example, to extract witness $\{\mathbf{Y}_{r,j} = E(\mathbf{y}_r)_j\}_{j \in I}$, we can build an (online) table T where each element of the table is a pair $(i, E(\mathbf{y}_r)_i)$ ($1 \leq i \leq M$). Then for every $j \in I$, we build a lookup gate checking that $(j, \mathbf{Y}_{r,j})$ is in the table T , thus guarantee that $\mathbf{Y}_{r,j}$ is identical to $E(\mathbf{y}_r)_j$. The circuit description is now independent of the random set I and we only need to preprocess the circuit once in the setup phase.

Third, with the help of Commit-and-Prove-SNARKs (CP-SNARK) [29, 30, 3], there is no need to check the consistency between commitments C_r, C_0 and $\mathbf{y}_r, \mathbf{y}_0$ in the circuit. Instead, we can commit $(\mathbf{y}_r, \mathbf{y}_0)$ to a multilinear commitment C , and build a CP-SNARK proof showing that the vector underlying C is identical to the witness vector $(\mathbf{y}_r, \mathbf{y}_0)$ in the circuit. We further observe that C can be a part of the witness polynomials, which further removes the need of an additional CP-SNARK proof.

After applying previous optimizations, the proof size is dominated by the $|I|$ field elements $\{h_j\}_{j \in I}$. We can altogether remove them by applying the CP-SNARK trick again. In particular, since $\{h_j\}_{j \in I}$ are both committed in the polynomial commitment C_f and the SNARK witness commitment, it is sufficient to construct a CP-SNARK proving that they are consistent in the two

commitments with respect to set I . We refer to Section 3.7 for constructing CP-SNARK proofs from multilinear commitments.

Since the bulk of verification work is delegated to the prover, there is no need to set $k = \sqrt{n}$. Instead, we can set an appropriate $k = \Theta(\lambda / \log n)$ to minimize the outer circuit size. In particular, the circuit is dominated by 2 linear encodings (of length n/k) and $|I|$ hashes (of length k). If we use vanilla HyperPlonk+ as the outer SNARK scheme and use Reinforced Concrete [7] as the hashing scheme that has a similar running time to SHA-256, for $\mu = 30$, $k = 64$ and 128-bit security (where $|I| = 1568$), the circuit complexity is only $\approx 2^{26}$ constraints. And we can expect the running time of the outer proof to be $O_\lambda(n)$.

The resulting multilinear polynomial commitment scheme is shown in Figure 9.

Remark 7.1 (CP-SNARKs instantiation.). *We can use the algorithm in Section 3.7.1 to instantiate the CP-SNARK in Figure 9 from any multilinear PIOP-based SNARKs with minimal overhead. First, we can split the witness polynomial into two parts: one includes the vector $(\mathbf{y}_r, \mathbf{y}_0)$ while the other includes the rest. The witness polynomial commitment to $(\mathbf{y}_r, \mathbf{y}_0)$ is essentially the commitment C_{p_y} in Figure 9, so that we don't need to additionally commit to $(\mathbf{y}_r, \mathbf{y}_0)$ and provide a proof. We emphasize that C_{p_y} is sent before the prover receives the challenge set I , which is essential for knowledge soundness.*

Second, the CP-proof generation between the multilinear commitment C_f and the SNARK witness polynomial commitment (w.r.t. set I) consists of a sumcheck with $O(\log m)$ rounds and 2 PCS openings (one for C_f and one for the witness polynomial). If we instantiate the SNARK with HyperPlonk+, we can batch the proving of the CP-proof and the SNARK proof so that the CP-proof adds no extra cost to the proof size beyond the original SNARK proof.

Building blocks: A CP-SNARK scheme OSNARK; an (extractable) polynomial commitment scheme PC; a hash commitment scheme HCom; and a linear encoding scheme E with minimum distance δ .

Setup($1^\lambda, \mu^*$) $\rightarrow \text{gp}$: Given security parameter λ , upper bound μ^* on the number of variables, set m^* so that the running time of OSNARK (and PC) is $O_\lambda(2^{\mu^*})$ for circuit size (and degree) m^* . Run $\text{gp}_o \leftarrow \text{OSNARK}.\text{Setup}(1^\lambda, m^*)$, $\text{gp}_{pc} \leftarrow \text{PC}.\text{Setup}(1^\lambda, m^*)$, run the indexing phase of OSNARK for the circuit statement in Figure 10 and obtain $(\text{vp}_o, \text{pp}_o)$. Output $\text{gp} := (\text{gp}_o, \text{gp}_{pc}, \text{vp}_o, \text{pp}_o)$.

Commit($\text{gp}; f$) $\rightarrow C_f$: Given polynomial $f \in \mathcal{F}_\mu^{(\leq 1)}$ with coefficients $\mathbf{w} = (f_{(0)}, \dots, f_{(n-1)})$, set $m = n/k$ so that the running time of OSNARK (and PC) is $O_\lambda(2^\mu)$ for circuit size (and degree) m . Interpret \mathbf{w} as a $k \times m$ matrix (i.e. $\mathbf{w} \in \mathbb{F}^{k \times m}$):

- Compute matrix $W \in \mathbb{F}^{k \times M}$ such that $W[i, :] = E(\mathbf{w}[i, :]) \forall i \in [k]$. Here $E : \mathbb{F}^m \rightarrow \mathbb{F}^M$ is the linear encoding.
- For each column $j \in [M]$, compute hash commitment $h_j \leftarrow \text{HCom}(W[:, j])$, where $W[:, j] \in \mathbb{F}^k$ is the j -th column of W .
- Let p_h be the polynomial that interpolates vector $(h_j)_{j \in [M]}$. Output commitment $C_f \leftarrow \text{PC}.\text{Commit}(\text{gp}_{pc}, p_h)$.

Open(gp, C_f, f): Given polynomial $f \in \mathcal{F}_\mu^{(\leq 1)}$ with coefficients $\mathbf{w} \in \mathbb{F}^{k \times m}$, run the committing algorithm and check if the output is consistent with C_f .

Eval($\text{gp}; C_f, \mathbf{z}, y; f$): Given public parameter gp , point $\mathbf{z} \in \mathbb{F}^\mu$ and commitment C_f to polynomial $f \in \mathcal{F}_\mu^{(\leq 1)}$ with coefficients $\mathbf{w} \in \mathbb{F}^{k \times m}$, transform \mathbf{z} to vectors $\mathbf{t}_0 \in \mathbb{F}^k$ and $\mathbf{t}_1 \in \mathbb{F}^m$ as in Equation (13) such that $f(\mathbf{z}) = \langle \mathbf{w}, \mathbf{t}_0 \otimes \mathbf{t}_1 \rangle$. The prover \mathcal{P} and the verifier \mathcal{V} run the following protocol:

1. \mathcal{V} sends \mathcal{P} a random vector $\mathbf{r} \in \mathbb{F}^k$.

2. Define vectors

$$\mathbf{y}_r = \sum_{i=1}^k \mathbf{r}_i \cdot \mathbf{w}[i, :], \quad \mathbf{y}_0 = \sum_{i=1}^k \mathbf{t}_{0,i} \cdot \mathbf{w}[i, :].$$

Let p_r be the polynomial that interpolates $(\mathbf{y}_r, \mathbf{y}_0)$. \mathcal{P} sends \mathcal{V} commitment $C_{p_y} \leftarrow \text{PC}.\text{Commit}(\text{gp}_{pc}, p_y)$.

3. \mathcal{V} sends a random subset $I \subseteq [M]$ with size $t := \frac{-\lambda}{\log(1-\delta)}$.
4. \mathcal{P} sends \mathcal{V} , a CP-SNARK proof π_o showing that

- the statement in Figure 10 holds true;
- the SNARK witness $(\mathbf{y}_r, \mathbf{y}_0)$ is identical to the vector committed in C_{p_y} ;
- the SNARK witness $(h_j)_{j \in I}$ is consistent with that in the polynomial commitment C_f w.r.t. set I .

5. \mathcal{V} checks π_o with public input $(\alpha, \mathbf{r}, y, \mathbf{z})$, and commitments C_{p_y}, C_f .

Figure 9: The multilinear polynomial commitment scheme.

Witness:

- messages $\mathbf{y}_r, \mathbf{y}_0 \in \mathbb{F}^m$, encodings $\mathbf{Y}_r, \mathbf{Y}_0 \in \mathbb{F}^M$, and evaluation vectors $\mathbf{t}_0 \in \mathbb{F}^k, \mathbf{t}_1 \in \mathbb{F}^m$;
- the columns of W in subset I , that is, $W' = (W[:, j])_{j \in I} \in \mathbb{F}^{k \times |I|}$;
- the values of $\mathbf{Y}_r, \mathbf{Y}_0$ in subset I , that is $\mathbf{Y}'_r = (\mathbf{Y}_{r,j})_{j \in I} \in \mathbb{F}^{|I|}$, and $\mathbf{Y}'_0 = (\mathbf{Y}_{0,j})_{j \in I} \in \mathbb{F}^{|I|}$;
- column hashes $\mathbf{h} = (h_1, \dots, h_{|I|}) \in \mathbb{F}^{|I|}$.

Public input:

- challenge vector $\mathbf{r} \in \mathbb{F}^k$;
- random subset $I \subseteq [M]$;
- evaluation point $\mathbf{z} \in \mathbb{F}^\mu$ and claimed evaluation $y \in \mathbb{F}$.

Circuit statements:

- $\mathbf{t}_0, \mathbf{t}_1$ is the correct transformation from \mathbf{z} as in Equation (13).
- $\mathbf{Y}_r = E(\mathbf{y}_r)$ and $\mathbf{Y}_0 = E(\mathbf{y}_0)$.
- For $i = 1 \dots |I|$, let $j_i \in I$ be the i -th element in I , it holds that
 - $\mathbf{Y}'_{r,i} = \mathbf{Y}_{r,j_i}$, that is, $(j_i, \mathbf{Y}'_{r,i})$ is in the table $\{(k, \mathbf{Y}_{r,k})\}_{k \in [M]}$, and
 - $\mathbf{Y}'_{0,i} = \mathbf{Y}_{0,j_i}$, that is, $(j_i, \mathbf{Y}'_{0,i})$ is in the table $\{(k, \mathbf{Y}_{0,k})\}_{k \in [M]}$.
- For $i = 1 \dots |I|$, it holds that
 - $h_i = \text{HCom}(W'[:, i])$ where $\text{HCom} : \mathbb{F}^k \rightarrow \mathbb{F}$ is the hash commitment scheme;
 - $\mathbf{Y}'_{r,i} = \langle \mathbf{r}, W'[:, i] \rangle$ and $\mathbf{Y}'_{0,i} = \langle \mathbf{t}_0, W'[:, i] \rangle$.
- $\langle \mathbf{y}_0, \mathbf{t}_1 \rangle = y$.

Figure 10: The outer SNARK circuit statement. The circuit configuration is independent of the random set I .

Theorem 7.1. *The multilinear polynomial commitment scheme in Figure 9 is correct and binding. The PCS evaluation protocol is knowledge-sound.*

Proof. Correctness and binding. Correctness holds obviously by inspection of the protocol. We prove the binding property by contradiction. Suppose an adversary finds a commitment C_f and two polynomials f_1, f_2 with different coefficients $\mathbf{w}_1, \mathbf{w}_2 \in \mathbb{F}^{k \times m}$ such that C_f can open to both \mathbf{w}_1 and \mathbf{w}_2 . There are two cases:

1. C_f can open to two different vectors of column hash commitments $\mathbf{h}_1, \mathbf{h}_2 \in \mathbb{F}^M$, which contradicts the binding property of the PCS PC.
2. C_f binds to a single vector $\mathbf{h} \in \mathbb{F}^M$, but encoding $\mathbf{w}_1, \mathbf{w}_2$ lead to two different encoded matrices $W_1, W_2 \in \mathbb{F}^{k \times M}$. This contradicts the collision resistance of the hash function.

In summary, the binding property holds.

Knowledge soundness. We use a similar technique as in [48] that enables extracting polynomials even if the linear code E is not efficiently decodable. For any adversary \mathcal{A} that can pass the PCS evaluation check with probability more than ϵ , the extractor $\mathcal{E}^{\mathcal{A}}$ works as follows:

1. Run \mathcal{A} and obtain commitment C_f , point $\mathbf{z} \in \mathbb{F}^\mu$, and evaluation $y \in \mathbb{F}$. Run the extractors of the PCS and the hash function to recover the matrix $W' \in \mathbb{F}^{k \times M}$ underlying C_f . Abort if the extraction fails.

2. Set $S \leftarrow \emptyset$, repeat the following procedures until $|S| \geq k$ or the number of \mathbf{r} being sampled is more than $8k/\epsilon$:
 - Sample and send \mathcal{A} a random vector $\mathbf{r} \xleftarrow{\$} \mathbb{F}^k$.
 - Obtain the PCS commitments C_{p_y} . Use the PCS extractor to extract the vector $(\mathbf{y}_r, \mathbf{y}_0) \in \mathbb{F}^{2m}$. Abort and rerun with another \mathbf{r} if the extraction fails.
 - Sample and send \mathcal{A} a random subset $I \subseteq [M]$.
 - Obtain the CP-SNARK proof π_o . Add the pair $(\mathbf{r}, \mathbf{y}_r)$ into set S if the proof correctly verifies.
3. If $|S| \geq k$ and the random vectors $\{\mathbf{r}\}$ in S are linearly independent, run the Gaussian elimination algorithm to extract the witness \mathbf{w} from $S = \{(\mathbf{r}, \mathbf{y}_r)\}$, otherwise abort.

The extractor runs in polynomial time as Step 2 runs in polynomial time, and the extractor executes Step 2 for at most $8k/\epsilon$ times. Next, we argue that the extractor's success probability is non-negligible. Since \mathcal{A} succeeds with probability at least ϵ , with probability at least $\epsilon/2$ over the choice of (C_f, \mathbf{z}, y) , the adversary passes the PCS evaluation protocol $\text{Eval}(\text{gp}; C_f, \mathbf{z}, y)$ with probability at least $\epsilon/2$. We denote by B the event that the above happens.

Conditioned on event B , we first argue that with high probability, \mathcal{E} can add k pairs to S , and the \mathbf{r} 's in S are linearly-independent. Note that for each run of PCS evaluation (with a freshly sampled vector \mathbf{r}), the probability that the extractor adds a pair to S is at least $\epsilon/2 - \text{negl}(\lambda) \geq \epsilon/4$. This is because \mathcal{A} passes the checks with probability at least $\epsilon/2$, and thus with probability at least $\epsilon/2 - \text{negl}(\lambda)$, \mathcal{A} passes all the checks, and the PCS extractor succeeds. Therefore, by Chernoff bound, the probability that \mathcal{E} adds k pairs to S within $8k/\epsilon$ runs of Step 2 is at least $1 - \exp(-k/8)$. Moreover, as noted by Lemma 2 of [48], the random vectors $\{\mathbf{r}\}$ in set S are linearly independent with overwhelming probability.

Next, still conditioned on event B , we argue that with probability $1 - \text{negl}(\lambda)$, there exists a coefficient matrix $\mathbf{w} \in \mathbb{F}^{k \times m}$ that is consistent with the commitment C_f , such that $\langle \mathbf{w}, \mathbf{t}_0 \otimes \mathbf{t}_1 \rangle = y$ (i.e. the evaluation is correct) and $\mathbf{y}_r = \sum_{i=1}^k \mathbf{r}_i \cdot \mathbf{w}_i$ for every pair $(\mathbf{r}, \mathbf{y}_r)$ in set S . Let $W' \in \mathbb{F}^{k \times M}$ be the matrix extracted by \mathcal{E} at Step 1, note that W' commits to C_f . Consider each run of PCS evaluation where the extractor adds a pair $(\mathbf{r}, \mathbf{y}_r)$ to S . Since C_f, C_{p_y} are binding, and the SNARK proofs verify, it holds that w.h.p over the choice of I , $E(\mathbf{y}_r)$ is close to $\sum_{i=1}^k \mathbf{r}_i \cdot W'_i$. By Lemma 1 in [48], w.h.p. over the choice of \mathbf{r} , it also holds that W'_i is close to a codeword for all $i \in [k]$. Therefore, there exists a matrix $\mathbf{w} \in \mathbb{F}^{k \times m}$ such that (i) W'_i is close to $E(\mathbf{w}_i)$ for all $i \in [k]$, and (ii) $\mathbf{y}_r = \sum_{i=1}^k \mathbf{r}_i \cdot \mathbf{w}_i$. Moreover, by the uniqueness of encoding, \mathbf{w} is identical for every challenge vector \mathbf{r} in set S . Similarly, we can argue that $\mathbf{y}_0 = \sum_{i=1}^k \mathbf{t}_{0,i} \cdot \mathbf{w}_i$ and thus $\langle \mathbf{w}, \mathbf{t}_0 \otimes \mathbf{t}_1 \rangle = y$.

Given the above, we conclude that with high probability, it holds that (i) \mathcal{E} adds k pairs to S where the \mathbf{r} 's in S are linearly independent; and (ii) there exists $\mathbf{w} \in \mathbb{F}^{k \times m}$ that is consistent with C_f and $\langle \mathbf{w}, \mathbf{t}_0 \otimes \mathbf{t}_1 \rangle = y$ and $\mathbf{y}_r = \sum_{i=1}^k \mathbf{r}_i \cdot \mathbf{w}_i$ for every pair $(\mathbf{r}, \mathbf{y}_r)$ in set S . In summary, conditioned on event B , the extractor can extract the coefficient matrix \mathbf{w} via Gaussian elimination with high probability, which completes the proof. \square

Theorem 7.2. *When instantiating the outer SNARK with HyperPlonk+, the multilinear PCS in Figure 9 has committing and evaluation opening complexity $O_\lambda(n)$; the proof size and verifier time is $O_\lambda(\log n)$.*

Proof. Fix $k = \Theta(\lambda/\log n)$ and let $m = n/k$. The committing algorithm takes $O(n)$ \mathbb{F} -ops to encode the coefficients $\mathbf{w} \in \mathbb{F}^{k \times m}$ to $W \in \mathbb{F}^{k \times M}$, $O(n)$ hashes to compute the column commitments, and an $O(m)$ -sized MSM to commit to the vector of column commitments. The total complexity is $O_\lambda(n)$.

The evaluation proving mainly consists of the following steps:

- compute a HyperPlonk+ SNARK proof for the statement in Figure 10;
- compute a CP-SNARK proof between the commitment C_f and the SNARK witness polynomial commitment with respect to a set I .

By the linear algorithm specified in Section 3.7.1, the CP-SNARK proof generation is dominated by a multi-group-exponentiations with size s , where s is the circuit size; similarly, the HyperPlonk+ SNARK proving is also dominated by a few multi-group-exponentiations with size s . Next, we prove that the outer circuit complexity is $s = O(m)$, Hence the evaluation opening complexity is also $O_\lambda(n)$.

Lemma 7.3. *The number of constraints in the circuit in Figure 10 is $O(m) + |H| \cdot O(k\lambda)$, where $|H|$ is the number of constraints for a hash instance.*

Proof. The circuit for computing $\mathbf{t}_0, \mathbf{t}_1$ from \mathbf{z} takes $O(k)$ and $O(m)$ constraints, respectively; the circuit for encoding $\mathbf{y}_r, \mathbf{y}_0$ takes $O(m)$ constraints; the extraction of $\mathbf{Y}'_r, \mathbf{Y}'_0$ from $\{E(\mathbf{y}_r)_j, E(\mathbf{y}_0)\}_{j \in I}$, takes $2|I|$ lookup gates; the computation of $\mathbf{Y}'_r, \mathbf{Y}'_0$ takes $O(k|I|)$ constraints; the computation of y takes $O(m)$ constraints; the computation of hashes $\{h_i\}$ takes $k|I| = O(k\lambda)$ hash gadgets as $|I| = \Theta(\lambda)$, thus the number of constraints is $O(m) + |H| \cdot O(k\lambda)$. \square

The evaluation verifier checks the the CP-SNARK proof π_o which takes time $O_\lambda(\log n)$.

The evaluation proof consists of a single CP-SNARK proof π_o . As noted by Remark 7.1, the proof size is no more than that of a single HyperPlonk+ proof for circuit size $s = O(m)$. In summary, the proof size is $O_\lambda(\log n)$. \square

Remark 7.2. *We stress that the CP-SNARK proving time (between C_f and the SNARK witness) for set I is independent of the size of I , as the complexity of the special batching algorithm in Section 3.7.1 is independent of the number of evaluations. This is highly important because $|I|$ can be as large as thousands in practice.*

Remark 7.3. *If we instantiate the linear code with the generalized Spielman code proposed in [78], and instantiate the outer SNARK with vanilla HyperPlonk+, for 128-bit security and $\mu = 30$, the outer circuit size is approximately 2^{26} , thus the proof is less than 10 KBs.*

Remark 7.4. *In practice, to minimize the outer circuit complexity, we choose k such that $2 \cdot \ell(n/k) = k|I| \cdot |H|$, where $\ell(n/k)$ is the circuit size for encoding a message with length n/k . Note that $|I| = 1568$ for 128-bit security and $|I| = 980$ for 80-bit security.*

Remark 7.5. *In contrast with Orion, Orion+ requires using a pairing-friendly field. We leave the construction of linear-time PCS with succinct proofs/verifier that supports arbitrary fields as future work.*

Conclusions and open problems

We presented a SNARK with a fast prover that is an adaption of Plonk to the boolean hypercube. We also present Orion+, a significantly improved multi-linear commitment scheme with short proofs and fully-linear prover time. There are several open questions:

- Higher degree shifts: We show how to build a generator for the boolean hypercube that enables a `next` function that shifts points in the hypercube by 1. This is critical for building a lookup protocol. In some versions of the Halo 2 arithmetization, the proof system accesses machine states from more than the previous step. To implement this, we need higher degree shifts. This can be done by composing the `next` function multiple times, but this has an exponential blow-up in verifier time. Implementing higher degree `next` functions efficiently remains an open problem.
- Better codes and hash functions for Orion+: The utility of Orion+ for smaller polynomials is limited by the use of recursion. Only for large multilinear polynomials (approximately with more than 24 variables), does the recursive circuit size become less than the original polynomial size. The keys to improving this are linear codes with better distance and more efficient and circuit-friendly hash functions.
- Automatic custom gate design: HyperPlonk+ supports high-degree custom gates efficiently. Currently, designing suitable custom gates for specific applications is a task left to the circuit designer. It remains an open problem to have a more principled approach that automates and optimizes the design of the custom gates for any given application.

Acknowledgments. We want to thank Ben Fisch and Alex Oezdemir for helpful discussions and Tiancheng Xie for answering many questions regarding Orion and Virgo. This work was partially funded by NSF, DARPA, the Simons Foundation, UBRI, and NTT Research. Opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA.

References

- [1] A. Aly, T. Ashur, E. Ben-Sasson, S. Dhooghe, and A. Szepieniec. Design of symmetric-key primitives for advanced cryptographic protocols. *IACR Trans. Symm. Cryptol.*, 2020(3):1–45, 2020.
- [2] S. Ames, C. Hazay, Y. Ishai, and M. Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, editors, *ACM CCS 2017*, pages 2087–2104. ACM Press, Oct. / Nov. 2017.
- [3] D. F. Aranha, E. M. Bennedsen, M. Campanelli, C. Ganesh, C. Orlandi, and A. Takahashi. ECLIPSE: Enhanced compiling method for pedersen-committed zkSNARK engines. Cryptology ePrint Archive, Report 2021/934, 2021. <https://eprint.iacr.org/2021/934>.
- [4] arkworks contributors. `arkworks` zksnark ecosystem, 2022.

- [5] T. Attema, S. Fehr, and M. Kloß. Fiat-shamir transformation of multi-round interactive proofs. Cryptology ePrint Archive, Report 2021/1377, 2021. <https://eprint.iacr.org/2021/1377>.
- [6] L. Babai and S. Moran. Arthur-merlin games: A randomized proof system, and a hierarchy of complexity classes. *J. Comput. Syst. Sci.*, 36(2):254–276, 1988.
- [7] M. Barbara, L. Grassi, D. Khovratovich, R. Lueftnegger, C. Rechberger, M. Schafnecker, and R. Walch. Reinforced concrete: Fast hash function for zero knowledge proofs and verifiable computation. Cryptology ePrint Archive, Report 2021/1038, 2021. <https://eprint.iacr.org/2021/1038>.
- [8] S. Bayer and J. Groth. Efficient zero-knowledge argument for correctness of a shuffle. In D. Pointcheval and T. Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 263–280. Springer, Heidelberg, Apr. 2012.
- [9] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev. Fast reed-solomon interactive oracle proofs of proximity. In I. Chatzigiannakis, C. Kaklamanis, D. Marx, and D. Sannella, editors, *ICALP 2018*, volume 107 of *LIPICS*, pages 14:1–14:17. Schloss Dagstuhl, July 2018.
- [10] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Report 2018/046, 2018. <https://eprint.iacr.org/2018/046>.
- [11] E. Ben-Sasson, D. Carmon, S. Kopparty, and D. Levit. Elliptic curve fast fourier transform (ecfft) part ii: Scalable and transparent proofs over all large fields. 2022.
- [12] E. Ben-Sasson, A. Chiesa, M. Riabzev, N. Spooner, M. Virza, and N. P. Ward. Aurora: Transparent succinct arguments for R1CS. In Y. Ishai and V. Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 103–128. Springer, Heidelberg, May 2019.
- [13] E. Ben-Sasson, A. Chiesa, and N. Spooner. Interactive oracle proofs. In M. Hirt and A. D. Smith, editors, *TCC 2016-B, Part II*, volume 9986 of *LNCS*, pages 31–60. Springer, Heidelberg, Oct. / Nov. 2016.
- [14] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Scalable zero knowledge via cycles of elliptic curves. In J. A. Garay and R. Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 276–294. Springer, Heidelberg, Aug. 2014.
- [15] D. J. Bernstein. Curve25519: New Diffie-Hellman speed records. In M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, editors, *PKC 2006*, volume 3958 of *LNCS*, pages 207–228. Springer, Heidelberg, Apr. 2006.
- [16] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In S. Goldwasser, editor, *ITCS 2012*, pages 326–349. ACM, Jan. 2012.
- [17] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. Recursive composition and bootstrapping for SNARKs and proof-carrying data. Cryptology ePrint Archive, Report 2012/095, 2012. <https://eprint.iacr.org/2012/095>.

- [18] N. Bitansky, A. Chiesa, Y. Ishai, R. Ostrovsky, and O. Paneth. Succinct non-interactive arguments via linear interactive proofs. In A. Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 315–333. Springer, Heidelberg, Mar. 2013.
- [19] D. Boneh, J. Drake, B. Fisch, and A. Gabizon. Halo infinite: Proof-carrying data from additive polynomial commitments. In T. Malkin and C. Peikert, editors, *CRYPTO 2021, Part I*, volume 12825 of *LNCS*, pages 649–680, Virtual Event, Aug. 2021. Springer, Heidelberg.
- [20] J. Bootle, A. Cerulli, E. Ghadafi, J. Groth, M. Hajiabadi, and S. K. Jakobsen. Linear-time zero-knowledge proofs for arithmetic circuit satisfiability. In T. Takagi and T. Peyrin, editors, *ASIACRYPT 2017, Part III*, volume 10626 of *LNCS*, pages 336–365. Springer, Heidelberg, Dec. 2017.
- [21] J. Bootle, A. Chiesa, and J. Groth. Linear-time arguments with sublinear verification from tensor codes. In R. Pass and K. Pietrzak, editors, *TCC 2020, Part II*, volume 12551 of *LNCS*, pages 19–46. Springer, Heidelberg, Nov. 2020.
- [22] J. Bootle, A. Chiesa, Y. Hu, and M. Orrù. Gemini: Elastic SNARKs for diverse environments. In O. Dunkelman and S. Dziembowski, editors, *EUROCRYPT 2022, Part II*, volume 13276 of *LNCS*, pages 427–457. Springer, Heidelberg, May / June 2022.
- [23] S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, and H. Wu. ZEXE: Enabling decentralized private computation. In *2020 IEEE Symposium on Security and Privacy*, pages 947–964. IEEE Computer Society Press, May 2020.
- [24] S. Bowe, J. Grigg, and D. Hopwood. Halo: Recursive proof composition without a trusted setup. Cryptology ePrint Archive, Report 2019/1021, 2019. <https://eprint.iacr.org/2019/1021>.
- [25] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy*, pages 315–334. IEEE Computer Society Press, May 2018.
- [26] B. Bünz, A. Chiesa, W. Lin, P. Mishra, and N. Spooner. Proof-carrying data without succinct arguments. In T. Malkin and C. Peikert, editors, *CRYPTO 2021, Part I*, volume 12825 of *LNCS*, pages 681–710, Virtual Event, Aug. 2021. Springer, Heidelberg.
- [27] B. Bünz, A. Chiesa, P. Mishra, and N. Spooner. Recursive proof composition from accumulation schemes. In R. Pass and K. Pietrzak, editors, *TCC 2020, Part II*, volume 12551 of *LNCS*, pages 1–18. Springer, Heidelberg, Nov. 2020.
- [28] B. Bünz, B. Fisch, and A. Szepieniec. Transparent SNARKs from DARK compilers. In A. Canteaut and Y. Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 677–706. Springer, Heidelberg, May 2020.
- [29] M. Campanelli, A. Faonio, D. Fiore, A. Querol, and H. Rodríguez. Lunar: A toolbox for more efficient universal and updatable zkSNARKs and commit-and-prove extensions. In M. Tibouchi and H. Wang, editors, *ASIACRYPT 2021, Part III*, volume 13092 of *LNCS*, pages 3–33. Springer, Heidelberg, Dec. 2021.

- [30] M. Campanelli, D. Fiore, and A. Querol. LegoSNARK: Modular design and composition of succinct zero-knowledge proofs. In L. Cavallaro, J. Kinder, X. Wang, and J. Katz, editors, *ACM CCS 2019*, pages 2075–2092. ACM Press, Nov. 2019.
- [31] J. L. Carter and M. N. Wegman. Universal classes of hash functions. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 106–112, 1977.
- [32] A. Chiesa, M. A. Forbes, and N. Spooner. A zero knowledge sumcheck and its applications. Cryptology ePrint Archive, Report 2017/305, 2017. <https://eprint.iacr.org/2017/305>.
- [33] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. P. Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In A. Canteaut and Y. Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 738–768. Springer, Heidelberg, May 2020.
- [34] A. Chiesa, D. Ojha, and N. Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. In A. Canteaut and Y. Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 769–793. Springer, Heidelberg, May 2020.
- [35] A. Chiesa and E. Tromer. Proof-carrying data and hearsay arguments from signature cards. In A. C.-C. Yao, editor, *ICS 2010*, pages 310–331. Tsinghua University Press, Jan. 2010.
- [36] A. R. Choudhuri, A. Jain, and Z. Jin. Non-interactive batch arguments for NP from standard assumptions. In T. Malkin and C. Peikert, editors, *CRYPTO 2021, Part IV*, volume 12828 of *LNCS*, pages 394–423, Virtual Event, Aug. 2021. Springer, Heidelberg.
- [37] A. R. Choudhuri, A. Jain, and Z. Jin. SNARGs for \mathcal{P} from LWE. Cryptology ePrint Archive, Report 2021/808, 2021. <https://eprint.iacr.org/2021/808>.
- [38] H. de Valence. Merlin transcript, 2022.
- [39] J. Drake. Plonk-style SNARKs without FFTs. [link](#), 2019.
- [40] EspressoSystems. Specifications: Configurable asset privacy. Github, 2022.
- [41] A. Gabizon. Multiset checks in plonk and plookup. <https://hackmd.io/@arielg/ByFgSDA7D>.
- [42] A. Gabizon and Z. J. Williamson. plookup: A simplified polynomial protocol for lookup tables. Cryptology ePrint Archive, Report 2020/315, 2020. <https://eprint.iacr.org/2020/315>.
- [43] A. Gabizon and Z. J. Williamson. Proposal: The turbo-plonk program syntax for specifying snark programs. https://docs.zkproof.org/pages/standards/accepted-workshop3/proposal-turbo_plonk.pdf, 2020.
- [44] A. Gabizon, Z. J. Williamson, and O. 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>.
- [45] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In T. Johansson and P. Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 626–645. Springer, Heidelberg, May 2013.

- [46] C. Gentry and D. Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In L. Fortnow and S. P. Vadhan, editors, *43rd ACM STOC*, pages 99–108. ACM Press, June 2011.
- [47] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989.
- [48] A. Golovnev, J. Lee, S. Setty, J. Thaler, and R. S. Wahby. Brakedown: Linear-time and post-quantum SNARKs for R1CS. Cryptology ePrint Archive, Report 2021/1043, 2021. <https://eprint.iacr.org/2021/1043>.
- [49] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schafnecker. Poseidon: A new hash function for zero-knowledge proof systems. In M. Bailey and R. Greenstadt, editors, *USENIX Security 2021*, pages 519–535. USENIX Association, Aug. 2021.
- [50] J. Groth. On the size of pairing-based non-interactive arguments. In M. Fischlin and J.-S. Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 305–326. Springer, Heidelberg, May 2016.
- [51] U. Haböck. A summary on the fri low degree test. *Cryptology ePrint Archive*, 2022.
- [52] D. Harvey and J. Van Der Hoeven. Polynomial multiplication over finite fields in time. *Journal of the ACM (JACM)*, 69(2):1–40, 2022.
- [53] D. Hopwood, S. Bowe, T. Hornby, and N. Wilcox. Zcash protocol specification. version 2022.3.8. Online, 2022. <https://zips.z.cash/protocol/protocol.pdf>.
- [54] A. Kate, G. M. Zaverucha, and I. Goldberg. Constant-size commitments to polynomials and their applications. In M. Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 177–194. Springer, Heidelberg, Dec. 2010.
- [55] A. Kattis, K. Panarin, and A. Vlasov. RedShift: Transparent SNARKs from list polynomial commitment IOPs. Cryptology ePrint Archive, Report 2019/1400, 2019. <https://eprint.iacr.org/2019/1400>.
- [56] J. Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *24th ACM STOC*, pages 723–732. ACM Press, May 1992.
- [57] A. Kothapalli, S. Setty, and I. Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. Cryptology ePrint Archive, Report 2021/370, 2021. <https://eprint.iacr.org/2021/370>.
- [58] J. Lee. Dory: Efficient, transparent arguments for generalised inner products and polynomial commitments. In K. Nissim and B. Waters, editors, *TCC 2021, Part II*, volume 13043 of *LNCS*, pages 1–34. Springer, Heidelberg, Nov. 2021.
- [59] C. Lund, L. Fortnow, H. Karloff, and N. Nisan. Algebraic methods for interactive proof systems. *Journal of the ACM (JACM)*, 39(4):859–868, 1992.

- [60] M. Maller, S. Bowe, M. Kohlweiss, and S. Meiklejohn. Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. In L. Cavallaro, J. Kinder, X. Wang, and J. Katz, editors, *ACM CCS 2019*, pages 2111–2128. ACM Press, Nov. 2019.
- [61] S. Masson, A. Sanso, and Z. Zhang. Bandersnatch: a fast elliptic curve built over the BLS12-381 scalar field. Cryptology ePrint Archive, Report 2021/1152, 2021. <https://eprint.iacr.org/2021/1152>.
- [62] S. Micali. CS proofs (extended abstracts). In *35th FOCS*, pages 436–453. IEEE Computer Society Press, Nov. 1994.
- [63] C. Papamanthou, E. Shi, and R. Tamassia. Signatures of correct computation. In A. Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 222–242. Springer, Heidelberg, Mar. 2013.
- [64] L. Pearson, J. Fitzgerald, H. Masip, M. Bellés-Muñoz, and J. L. Muñoz-Tapia. PlonKup: Reconciling PlonK with plookup. Cryptology ePrint Archive, Report 2022/086, 2022. <https://eprint.iacr.org/2022/086>.
- [65] J. Posen and A. A. Kattis. Caulk+: Table-independent lookup arguments. Cryptology ePrint Archive, Report 2022/957, 2022. <https://eprint.iacr.org/2022/957>.
- [66] S. Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In D. Micciancio and T. Ristenpart, editors, *CRYPTO 2020, Part III*, volume 12172 of *LNCS*, pages 704–737. Springer, Heidelberg, Aug. 2020.
- [67] S. Setty and J. Lee. Quarks: Quadruple-efficient transparent zkSNARKs. Cryptology ePrint Archive, Report 2020/1275, 2020. <https://eprint.iacr.org/2020/1275>.
- [68] D. R. Stinson. Universal hashing and authentication codes. *Designs, Codes and Cryptography*, 4(3):369–380, 1994.
- [69] E. System. *Jellyfish* jellyfish cryptographic library, 2022.
- [70] J. Thaler. Time-optimal interactive proofs for circuit evaluation. In R. Canetti and J. A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 71–89. Springer, Heidelberg, Aug. 2013.
- [71] J. Thaler. Proofs, arguments, and zero-knowledge, 2020.
- [72] P. Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In R. Canetti, editor, *TCC 2008*, volume 4948 of *LNCS*, pages 1–18. Springer, Heidelberg, Mar. 2008.
- [73] R. S. Wahby, I. Tzialla, a. shelat, J. Thaler, and M. Walfish. Doubly-efficient zkSNARKs without trusted setup. In *2018 IEEE Symposium on Security and Privacy*, pages 926–943. IEEE Computer Society Press, May 2018.
- [74] B. Waters and D. J. Wu. Batch arguments for NP and more from standard bilinear group assumptions. Cryptology ePrint Archive, Report 2022/336, 2022. <https://eprint.iacr.org/2022/336>.

- [75] M. N. Wegman and J. L. Carter. New hash functions and their use in authentication and set equality. *Journal of computer and system sciences*, 22(3):265–279, 1981.
- [76] D. Wikström. Special soundness in the random oracle model. Cryptology ePrint Archive, Report 2021/1265, 2021. <https://eprint.iacr.org/2021/1265>.
- [77] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. In A. Boldyreva and D. Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 733–764. Springer, Heidelberg, Aug. 2019.
- [78] T. Xie, Y. Zhang, and D. Song. Orion: Zero knowledge proof with linear prover time. Cryptology ePrint Archive, Report 2022/1010, 2022. <https://eprint.iacr.org/2022/1010>.
- [79] A. L. Xiong, B. Chen, Z. Zhang, B. Bünz, B. Fisch, F. Krell, and P. Camacho. VERI-ZEXE: Decentralized private computation with universal setup. Cryptology ePrint Archive, Report 2022/802, 2022. <https://eprint.iacr.org/2022/802>.
- [80] A. Zapico, V. Buterin, D. Khovratovich, M. Maller, A. Nitulescu, and M. Simkin. Caulk: Lookup arguments in sublinear time. Cryptology ePrint Archive, Report 2022/621, 2022. <https://eprint.iacr.org/2022/621>.
- [81] Zcash. PLONKish arithmetization. [link](#), 2022.
- [82] J. Zhang, T. Xie, Y. Zhang, and D. Song. Transparent polynomial delegation and its applications to zero knowledge proof. In *2020 IEEE Symposium on Security and Privacy*, pages 859–876. IEEE Computer Society Press, May 2020.

A Zero Knowledge PIOPs and zk-SNARKs

In this Section, we describe a compiler that transforms a class of sumcheck-based multivariate PolyIOPs into ones that are zero knowledge. The general framework consists of two parts. The first part is to mask the oracle polynomials so that their oracle query answers do not reveal the information of the original polynomial; moreover, we require that the masking do not change evaluations over the boolean hypercube, thus the correctness of PIOPs still holds. The second part is making the underlying sumcheck PIOPs zero knowledge. For this we reuse the ZK sumchecks described in [77].

We note that in contrast with univariate PIOPs, there is a subtlety in compiling multivariate PIOPs: the zero-knowledge property is hard to achieve if the set of query points is highly structural. E.g., suppose f is 2-variate and there are 4 query points (r_1, r_2) , (r_1, r_1) , (r_2, r_1) , (r_2, r_2) . Though all of the 4 points are distinct, each dimension has at least 2 points that share the same value. This makes the adversary much easier to cancel out the masking randomness and obtain a correlation between the evaluations of f on the 4 points. We resolve the issue by restricting the set of query points to be less structured. In particular, we require that there is at least one dimension where each point has a distinct value. We also slightly modify the underlying sumcheck protocols to satisfy the restriction while the soundness is not affected.

The Section is organized as follows. We define zero knowledge PIOPs in Section A.1. In Section A.2, we describe a scheme masking the multivariate polynomials. Section A.3 reviews the ZK sumchecks in [77]. We describe the ZK compiler for PIOPs in Section A.4 and explain how to obtain a zk-SNARK from a zk-PIOP and a PCS in Section A.5.

A.1 Definition

We follow [33] and define the (honest verifier) zero-knowledge property of PIOPs. Since the provers in sumcheck PIOPs also send field elements, we slightly adapt the definition in [33].

Definition A.1. *A PIOP $\langle \mathcal{P}, \mathcal{V} \rangle$ has perfect zero-knowledge with query bound t and query checker C if there is a PPT simulator \mathcal{S} such that for every field \mathbb{F} , index i , instance \mathbf{x} , witness \mathbf{w} , and every (t, C) -admissible verifier \mathcal{V}^* , the following transcripts are identically distributed:*

$$\mathsf{View}(\mathcal{P}(\mathbb{F}, i; \mathbf{x}; \mathbf{w}), \mathcal{V}^*) \approx \mathcal{S}^{\mathcal{V}^*}(\mathbb{F}, i; \mathbf{x}).$$

Here the view consists of \mathcal{V}^* 's randomness, the non-oracle messages sent by \mathcal{P} , and the list of answers to \mathcal{V}^* 's oracle queries. A verifier is (t, C) -admissible if it makes no more than t queries, and each query is accepted by the checker C . We say that $\langle \mathcal{P}, \mathcal{V} \rangle$ is honest-verifier-zero-knowledge (HVZK) if there is a simulator for \mathcal{V} .

A.2 Polynomial Masking

Definition A.2. *A randomized algorithm msk is a (t, C, μ) -masking if*

1. *for every $d \in \mathbb{N}$ and every polynomial $f \in \mathcal{F}_\mu^{(\leq d)}$, the masked polynomial $f^* \xleftarrow{\$} \mathsf{msk}(f, t, C)$ does not change evaluations over the boolean hypercube B_μ ;*
2. *for every $d \in \mathbb{N}$ and every polynomials $f \in \mathcal{F}_\mu^{(\leq d)}$, and every list of queries $\mathbf{q} := (q_1, \dots, q_t)$ that is accepted by the checker C , let $f^* \xleftarrow{\$} \mathsf{msk}(f, t, C)$. It holds that $(f^*(q_1), \dots, f^*(q_t))$ is uniformly distributed over \mathbb{F}^t .*

Lemma A.1. *There is a (t, C_ℓ, μ) -masking algorithm $\text{msk}(f, t, \ell)$ for every $\mu, t \in \mathbb{N}$ and $\ell \in [\mu]$, where checker C_ℓ accepts a list of queries (q_1, \dots, q_t) if and only if $b_{i,\ell} \notin \{0, 1, b_{1,\ell}, \dots, b_{i-1,\ell}\}$ for every query $q_i := (b_{i,1}, \dots, b_{i,\mu}) \in \mathbb{F}^\mu$ ($1 \leq i \leq t$). For any $f \in \mathcal{F}_\mu^{(\leq d)}$ and $\ell \in [\mu]$, the degree of the masked polynomial $f^* \leftarrow \text{msk}(f, t, \ell)$ is $\max(d, t + 1)$.*

Proof. Given a polynomial $f \in \mathcal{F}_\mu^{(\leq d)}$, query bound t , and checker C_ℓ , the algorithm does follow:

- Sample a univariate polynomial $R(X) := c_0 + c_1 X + \dots + c_{t-1} X^{t-1}$ where $c_0, \dots, c_{t-1} \xleftarrow{\$} \mathbb{F}$.
- Output $f^* := f + Z(X_\ell) \cdot R(X_\ell)$, where $Z(X_\ell) := X_\ell \cdot (1 - X_\ell)$.

It is clear that f^* has degree $\max(d, t + 1)$; f^* does not change f 's evaluations over B_μ as Z evaluates to zero over B_μ . Next, we argue that $\mathbf{f}^* := (f^*(q_1), \dots, f^*(q_t)) \in \mathbb{F}^t$ is uniformly random. Denote query $q_i := (b_{i,1}, \dots, b_{i,\mu})$ ($1 \leq i \leq t$), we define \mathbf{R} to be

$$\mathbf{R} := (Z(b_{1,\ell}) \cdot R(b_{1,\ell}), \dots, Z(b_{t,\ell}) \cdot R(b_{t,\ell})).$$

Since the queries satisfy $b_{i,\ell} \notin \{0, 1\}$ for every $i \in [t]$, it holds that $z_i := Z(b_{i,\ell})$ are non-zero and thus invertible. Moreover, since R is a random univariate polynomial with degree $t - 1$ and $\{b_{1,\ell}, \dots, b_{t,\ell}\}$ are distinct, it holds that $\{R(b_{1,\ell}), \dots, R(b_{t,\ell})\}$ are uniformly random. Therefore \mathbf{R} is uniformly random, and thus $\mathbf{f}^* = \mathbf{f} + \mathbf{q}$ is also uniformly random where $\mathbf{f} := (f(q_1), \dots, f(q_t))$. \square

A.3 Zero Knowledge SumCheck

Construction. Xie et al. [77] described an efficient ZK compiler for sumchecks. For reader's convenience, we adapt Construction 1 in [77] to a PIOP.

Zero knowledge SumCheck PIOP $\langle \mathcal{P}, \mathcal{V} \rangle$:

- Input: polynomial $f \in \mathcal{F}_\mu^{(\leq d)}$ and claimed sum $H \in \mathbb{F}$.
- \mathcal{P} samples a polynomial $g := c_0 + g_1(\mathbf{x}_1) + \dots + g_\mu(\mathbf{x}_\mu)$ where $g_i(\mathbf{x}_i) := c_{i,1}\mathbf{x}_i + \dots + c_{i,d}\mathbf{x}_i^d$ and $c_{i,1}, \dots, c_{i,d}$ are uniformly random. \mathcal{P} sends oracle g and a claimed sum $G := \sum_{\mathbf{x} \in B_\mu} g(\mathbf{x})$.
- \mathcal{V} sends a challenge $\rho \xleftarrow{\$} \mathbb{F}^*$.
- \mathcal{P} and \mathcal{V} run SumCheck PIOP (Section 3.1) over polynomial $f + \rho g$ and claimed sum $H + \rho G$.
- \mathcal{V} queries g and f at point \mathbf{r} where $\mathbf{r} \in \mathbb{F}^\mu$ is the vector of sumcheck's challenges. \mathcal{V} then checks that $f(\mathbf{r}) + \rho g(\mathbf{r})$ is consistent with the last message of the sumcheck.

The completeness of the ZK PIOP holds obviously, it was shown in [32] that the PIOP also preserves soundness. The zero knowledge property is proved in [77] and we state it below.

Lemma A.2 (Theorem 3 of [77]). *For every field \mathbb{F} , verifier \mathcal{V}^* and multivariate polynomial $f \in \mathcal{F}_\mu^{(\leq d)}$, there is a simulator $\mathcal{S}_{\text{sum}}(\mathbb{F}, \mu, d, H)$ that perfectly simulates \mathcal{P} 's oracle answers except for $f(\mathbf{r})$. Here $H := \sum_{\mathbf{x} \in B_\mu} f(\mathbf{x})$.*

A.4 Zero Knowledge Compilation for SumCheck-based PIOPs

A general description to the sumcheck-based PIOPs. The multivariate PIOPs considered in this paper can all be adapted to the following format.

General sumcheck-based PIOPs:

1. Both \mathcal{P} and \mathcal{V} have oracle access to a public multilinear polynomial $p_0 \in \mathcal{F}_{\mu_0}^{(\leq 1)}$.
2. For every $i \in [k_1]$, \mathcal{P} sends a multilinear polynomial $p_i \in \mathcal{F}_{\mu_i}^{(\leq 1)}$, and \mathcal{V} sends some random challenges. p_i is a function of p_0, \dots, p_{i-1} and verifier's previous challenges.
3. \mathcal{P} and \mathcal{V} sequentially run k_2 sumcheck PIOPs. The i -th ($1 \leq i \leq k_2$) sumcheck is over a polynomial $f_i := h_i(g_1, \dots, g_{c_i}) \in \mathcal{F}_{\nu_i}^{(\leq d_i)}$, where h_i is public information and each multilinear polynomial $g_j \in \mathcal{F}_{\nu_i}^{(\leq 1)}$ ($1 \leq j \leq c_i$) is $g_j := v|_{\mathbf{X}_S=\mathbf{b}}$ for some boolean vector \mathbf{b} and some $v \in \{p_1, \dots, p_{k_1}\}$, that is, g_j is a partial polynomial of v where the variables in S are set to \mathbf{b} .
4. For every $i \in [k_2]$, \mathcal{V} queries a random point $\mathbf{r}_i \in \mathbb{F}^{\nu_i}$ to the oracle f_i , where \mathbf{r}_i are the round challenges in the i -th sumcheck. \mathcal{V} then checks that $f_i(\mathbf{r}_i)$ is consistent with the last message in the i -th sumcheck.
5. For every $i \in [k_3]$, the verifier queries a point $\mathbf{c}_i \in \mathbb{F}^{\mu_{j_i}}$ to an oracle p_{j_i} ($0 \leq j_i \leq k_1$) and checks that the evaluation is y_i . We emphasize that the evaluations $\{y_i\}_{i \in [k_3]}$ can be efficiently and deterministically derived from $\{\mathbf{c}_i, j_i\}_{i \in [k_3]}$ and the public oracle p_0 .

We note that the above description captures all of the multivariate PIOPs in this paper because

- for the case where \mathcal{P} sends an oracle $f := h(g_1, \dots, g_c) \in \mathcal{F}_{\mu}^{(\leq d)}$ for $d > 1$, we can instead let \mathcal{P} send $g_1, \dots, g_c \in \mathcal{F}_{\mu}^{(\leq 1)}$ as h is public information;
- for the case where \mathcal{P} sends multiple multilinear oracles in a round, we can merge the polynomials into a single polynomial;
- the PIOPs we consider are all finally reduced to one or more sumcheck PIOPs.

Construction. We present a generic framework that transforms any (sumcheck-based) multivariate PIOPs into zero knowledge PIOPs. For a PIOP $\langle \mathcal{P}, \mathcal{V} \rangle$, let $(\{p_i\}_{i \in [0, k_1]}, \{f_i\}_{i \in [k_2]})$ be the polynomials denoted in the above protocol. For every $i \in [k_1]$, let $t_i \in \mathbb{N}$ be the number of p_i 's partial polynomials that appear in the sumcheck polynomials f_1, \dots, f_{k_2} , and let $t^* := \max\{t_i\}_{i \in [k_1]}$. For every $i \in [k_1]$, we assume that there exists index $\ell_i \in [\mu_i]$ such that for every p_i 's partial polynomial $v|_{\mathbf{X}_S=\mathbf{b}}$ that appears in some sumcheck (where p_i 's variables in set S are boolean), it holds that ℓ_i is not in the set S . Let msk be the masking algorithm described in Lemma A.1. The compiled zero knowledge PIOP $\langle \hat{\mathcal{P}}, \hat{\mathcal{V}} \rangle$ works as follows.

The ZK-compiler for sumcheck-based PIOPs:

1. For every $i \in [k_1]$, $\hat{\mathcal{P}}$ sends an oracle $[[p_i^*]]$ where $p_i^* \xleftarrow{s} \text{msk}(p_i, t_i, \ell_i)$. $\hat{\mathcal{V}}$ sends the same challenges as \mathcal{V} does.

2. $\hat{\mathcal{P}}$ and $\hat{\mathcal{V}}$ sequentially run k_2 zero knowledge sumcheck PIOPs (Section A.3). The i -th ($1 \leq i \leq k_2$) sumcheck is over the polynomial $f_i^* := h_i(g_1^*, \dots, g_{c_i}^*) \in \mathcal{F}_{\nu_i}^{(\leq d_i t^*)}$, where h_i is the same as in $\langle \mathcal{P}, \mathcal{V} \rangle$; each $g_j^* \in \mathcal{F}_{\nu_i}^{(\leq t^*)}$ ($1 \leq j \leq c_i$) is $g_j^* := v_{|\mathbf{x}_S=\mathbf{b}}^*$ for some boolean vector \mathbf{b} and some $v^* \in \{p_1^*, \dots, p_{k_1}^*\}$.
3. For every $i \in [k_2]$, $\hat{\mathcal{V}}$ queries a random point $\mathbf{r}_i \in \mathbb{F}^{\nu_i}$ to the oracle f_i , where \mathbf{r}_i are the round challenges in the i -th ZK sumcheck. $\hat{\mathcal{V}}$ then checks that $f_i(\mathbf{r}_i)$ is consistent with the last message of the i -th ZK sumcheck. We emphasize a slight modification over the original PIOP $\langle \mathcal{P}, \mathcal{V} \rangle$: *in the i -th sumcheck, $\hat{\mathcal{V}}$ samples each round challenge $\mathbf{r}_{i,j}$ ($1 \leq j \leq \mu_i$) in the set $\mathbb{F} \setminus \{0, 1, \mathbf{r}_{1,j}, \dots, \mathbf{r}_{i-1,j}\}$ rather than in \mathbb{F} .*
4. $\hat{\mathcal{V}}$ simulates \mathcal{V} , i.e., for all $i \in [k_3]$, queries points \mathbf{c}_i to oracle p_i^* and checks the evaluation.

Theorem A.3. *Given any PIOP $\langle \mathcal{P}, \mathcal{V} \rangle$ for some relation over the boolean hypercube, the compiled PIOP $\langle \hat{\mathcal{P}}, \hat{\mathcal{V}} \rangle$ is HVZK. Moreover, $\langle \hat{\mathcal{P}}, \hat{\mathcal{V}} \rangle$ preserves perfect completeness and negligible soundness.*

Proof. **Completeness.** Completeness holds because the sumcheck relations are over boolean hypercubes and the masked polynomials' evaluations do not change over the boolean hypercubes by the property of msk .

Soundness. Compared to the sumchecks in $\langle \mathcal{P}, \mathcal{V} \rangle$, the following changes of the sumchecks in $\langle \hat{\mathcal{P}}, \hat{\mathcal{V}} \rangle$ affect soundness error:

1. The degrees of the sumcheck polynomials are increased by a factor t^* .
2. The challenge space of j -th round in the i -th sumcheck is $\mathbb{F} \setminus \{0, 1, \mathbf{r}_{1,j}, \dots, \mathbf{r}_{i-1,j}\}$ rather than \mathbb{F} .
3. The sumcheck protocols are replaced with ZK sumchecks.

Since t^* and k_2 are constants and ZK sumchecks preserves soundness [32], the compiled protocol preserves negligible soundness.

HVZK. We describe the simulator as follows.

The simulator $\mathcal{S}^{\hat{\mathcal{V}}}(\mathbb{F}, \mathbf{i}; \mathbf{x})$:

1. Honestly generate the public polynomial $p_0 \in \mathcal{F}_{\mu_0}^{(\leq 1)}$.
2. Pick arbitrary polynomial $\{\tilde{p}_i\}_{i \in [k_1]}$ conditioned on that the sumcheck relations over f_1, \dots, f_{k_2} hold. Send $\hat{\mathcal{V}}$ polynomials $\{\tilde{p}_i^*\}_{i \in [k_1]}$ where $\tilde{p}_i^* \xleftarrow{s} \text{msk}(\tilde{p}_i, t_i, \ell_i)$, obtain from $\hat{\mathcal{V}}$ the challenges in the first k_1 rounds.
3. Run the next k_2 ZK sumcheck PIOPs using p_0 and the sampled polynomials $\{\tilde{p}_i^*\}_{i \in [k_1]}$.
4. For every $i \in [k_2]$, answer query $f_i^*(\mathbf{r}_i)$ honestly using $\{\tilde{p}_i^*\}_{i \in [k_1]}$.
5. For every $i \in [k_3]$, answer query \mathbf{c}_i with value y_i , where $\{y_i\}_{i \in [k_3]}$ are deterministically derived from $\{\mathbf{c}_i, j_i\}_{i \in [k_3]}$ and the public polynomial p_0 .

Next we show that $\mathcal{S}^{\hat{\mathcal{V}}}(\mathbb{F}, \mathbf{i}; \mathbf{x}) \approx \text{View}(\hat{\mathcal{P}}(\mathbb{F}, \mathbf{i}; \mathbf{x}; \mathbf{w}), \hat{\mathcal{V}})$. We set $H_0 := \mathcal{S}^{\hat{\mathcal{V}}}(\mathbb{F}, \mathbf{i}; \mathbf{x})$ and consider following hybrid games.

- Game H_1 : identical to H_0 except that step 3 is replaced with the ZK sumcheck simulator’s output. We note that $H_1 \approx H_0$ by the ZK property of the ZK sumchecks.
- Game H_2 : identical to H_1 except that the queries in step 4 are answered with random values. (Note that $f_i^*(\mathbf{r}_i)$ ’s answer is a random value consistent with the last message of the i -th sumcheck.) We argue that $H_2 \approx H_1$: for every $i \in [k_1]$, the number of queries to oracle $\tilde{p}_i^* \leftarrow \text{msk}(\tilde{p}_i, t_i, \ell_i)$ is no more than t_i and the ℓ_i -th element in each of the query point are distinct and non-boolean, by Lemma A.1, the answers to the queries are uniformly random.
- Game H_3 : identical to H_2 except that the polynomials $\{\tilde{p}_i\}_{i \in [k_1]}$ in step 2 are replaced with $\{p_i\}_{i \in [k_1]}$. Note that $H_3 \approx H_2$ as the verifier’s view does not change at all.
- Game $H_4 := \text{View}(\hat{\mathcal{P}}(\mathbb{F}, \mathbf{i}; \mathbf{x}; \mathbf{w}), \hat{\mathcal{V}})$: identical to H_3 except that the queries in step 4 are answered honestly and the ZK sumchecks are run honestly using p_0 and the sampled polynomials $\{p_i^*\}_{i \in [k_1]}$. With similar arguments (for H_1 and H_2) we have $H_4 \approx H_3$.

Given above, it holds that $\mathcal{S}^{\hat{\mathcal{V}}}(\mathbb{F}, \mathbf{i}; \mathbf{x}) \approx \text{View}(\hat{\mathcal{P}}(\mathbb{F}, \mathbf{i}; \mathbf{x}; \mathbf{w}), \hat{\mathcal{V}})$ and we complete the proof. \square

A.5 zk-SNARKs from PIOPs

In the ZK PIOP of Section A.4, the masked polynomials sent by the prover are with the form $f^* := f + Z(\mathbf{x}_\ell) \cdot R(\mathbf{x}_\ell)$ where $f \in \mathcal{F}_\mu^{(\leq 1)}$ is multilinear and $Z(\mathbf{x}_\ell) := \mathbf{x}_\ell \cdot (1 - \mathbf{x}_\ell)$ is univariate and with degree $t + 1$. It is shown in Theorem 10 of [19] that every additive and m -spanning PCS can be compiled into a hiding PCS with a zero-knowledge **Eval** protocol, where m -spanning means that commitments to polynomials of degree at most m can already generate the commitment space \mathbb{G} . Thus we can construct a hiding PCS for f^* with ZK evaluations from any additive and spanning polynomial commitment schemes (e.g., KZG and FRI). In particular, one instantiation is to set the commitment of f^* to be $(C_1, C_2) \in \mathbb{G}$ where C_1 is the multilinear commitment to f and C_2 is the univariate commitment to $Z(X) \cdot R(X)$, then apply the ZK transformation in [19].

By combining Theorem 2.4 and Theorem A.3 we obtain the following corollary.

Corollary A.4. *Given any (non-hiding) additive and spanning polynomial commitment schemes, we can transform any (non-ZK) sumcheck-based PIOP (Section A.4) for relation \mathcal{R} to a zk-SNARK for \mathcal{R} .*

B The FRI-based multilinear polynomial commitment

In this Section, we construct a simple multilinear polynomial commitment scheme (PCS) from FRI [9]. Along the way, we also show how to generically transform a univariate PCS to a multilinear PCS using the tensor-product *univariate* PIOP from [22], which might be of independent interest. We note that Virgo [82, §3] describes another scheme constructing multilinear PCS from FRI. The main idea is to build the evaluation opening proof from a univariate sumcheck [12], which in turn uses FRI. However, the naive scheme incurs linear-time overhead for the verifier. Virgo [82, §3] resolves the issue by delegating the verifier computation to the prover. To this end, the prover needs to compute another GKR proof convincing that the linear-time verifier will accept the proof. This complicates the scheme and adds additional concrete overhead on prover time and proof size.

We refer to [9, 55] and [51] for background of FRI low-degree testing and the approach to build univariate PCS from FRI. We note that the FRI-based univariate PCS supports batch opening. The evaluation opening protocol for multiple points on multiple polynomials invokes only a *single* call to the FRI protocol. Below we present a generic approach to transforming any univariate PCS into a multilinear PCS.

Generic transformation from univariate PCS to multilinear PCS. Bootle et al. built a *univariate* PIOP for the tensor-product relation in Section 5 of [22]. The tensor-product relation $(\mathbf{x}, \mathbf{w}) = ((\mathbb{F}, n, z_1, \dots, z_\mu, y), \mathbf{f})$ states that $\mathbf{f} \in \mathbb{F}^n$ satisfies that $\langle \mathbf{f}, \otimes_j(1, z_j) \rangle = y$, where $\langle \cdot, \cdot \rangle$ denotes an inner product, and \otimes denotes a tensor product. The PIOP naturally implies an algorithm that transforms univariate polynomial commitment schemes to multilinear polynomial commitment schemes.

- The commitment to a multilinear polynomial \tilde{f} with monomial coefficients, \mathbf{f} is the commitment to a univariate polynomial f with the same coefficients.
- To open \tilde{f} at point (z_1, \dots, z_μ) that evaluates y , the prover and the verifier runs the univariate PIOP for the relation $(\mathbf{x}, \mathbf{w}) = ((\mathbb{F}, n, z_1, \dots, z_\mu, y), \mathbf{f})$, which reduces to a batch evaluation on a set of $\mu + 1$ univariate polynomials.

We provide the concrete construction below. Let $\text{PC}_u = (\text{Setup}, \text{Commit}, \text{BatchOpen}, \text{BatchVfy})$ be a univariate PCS, we construct a multilinear PCS PC_m as follows.

- $\text{PC}_m.\text{Setup}(1^\lambda, \mu) \rightarrow (\mathbf{ck}, \mathbf{vk})$. On input security parameter λ and the number of variables μ , output $\text{PC}_u.\text{Setup}(1^\lambda, n)$ where $n = 2^\mu$.
- $\text{PC}_m.\text{Commit}(\mathbf{ck}, \tilde{f}) \rightarrow c$. On input committer key \mathbf{ck} , multilinear polynomial \tilde{f} with coefficients $\mathbf{f} \in \mathbb{F}^n$, output $\text{PC}_u.\text{Commit}(\mathbf{ck}, f)$ where f has the same coefficients as \mathbf{f} .
- $\text{PC}_m.\text{Open}(\mathbf{ck}, \tilde{f}, \mathbf{z}, y) \rightarrow \pi$. On input committer key \mathbf{ck} , multilinear polynomial \tilde{f} , point $\mathbf{z} \in \mathbb{F}^\mu$ and evaluation $y \in \mathbb{F}$, the prover computes the proof as follows. Let $f_0(X) := f(X)$ be the committed univariate polynomial that has the same coefficients as \tilde{f} , consider the following PIOP for the tensor-product relation $(\mathbf{x}, \mathbf{w}) = ((\mathbb{F}, n, \mathbf{z}, y), \mathbf{f})$:

- The prover sends the verifier univariate polynomials f_1, \dots, f_μ such that for all $i \in [\mu]$,

$$f_i(X) = g_{i-1}(X) + \mathbf{z}_i \cdot h_{i-1}(X),$$

where g_{i-1}, h_{i-1} satisfies that $f_{i-1}(X) = g_{i-1}(X^2) + X \cdot h_{i-1}(X^2)$.

- The verifier samples a random challenge $\beta \leftarrow \mathbb{F}^\times$ (where \mathbb{F}^\times is a multiplicative subgroup of \mathbb{F}), and queries the oracles to obtain evaluations $\{a_i, b_i, c_i\}_{i \in \{0, \dots, \mu\}}$ such that

$$a_i := f_i(\beta), \quad b_i := f_i(-\beta), \quad c_i := f_{i+1}(\beta^2).$$

Note that we skip $f_{\mu+1}(\beta^2)$ and set $c_\mu := y$.

- The verifier checks that for all $i \in \{0, \dots, \mu\}$,

$$c_i = \frac{a_i + b_i}{2} + z_i \cdot \frac{a_i - b_i}{2\beta}.$$

The opening proof π comprises (i) the univariate commitments to f_1, \dots, f_μ , (ii) the evaluations $\{a_i, b_i, c_i\}_{i \in \{0, \dots, \mu\}}$, and (iii) the batch opening proof for polynomials (f_0, f_1, \dots, f_μ) at points $(\beta, -\beta, \beta^2)$, where the random challenge β is derived via the Fiat-Shamir transform.

- $\text{PC}_m.\text{Vfy}(\text{vk}, c, \mathbf{z}, y, \pi) \in \{0, 1\}$. On input verifier key vk , commitment c , point \mathbf{z} , evaluation y , and proof π , parse π to commitments (c_1, \dots, c_μ) , evaluations evals , and the batch opening proof π^* . Derive random challenge β via the Fiat-Shamir transform, perform the verification check in the above PIOP, and run $\text{PC}_u.\text{BatchVfy}(\text{vk}, (c, c_1, \dots, c_\mu), (\beta, -\beta, \beta^2), \text{evals}, \pi^*)$.

Efficiency. We emphasize that when instantiated PC_u with the FRI-based PCS, the multilinear polynomial commitment scheme has approximately the same complexity as that in the univariate setting. In particular, the committing phase takes only a Merkle root computation with tree depth $\log(n)$; the opening phase takes (i) μ Merkle commitment computation where the i -th ($1 \leq i \leq \mu$) Merkle tree is with size $2^{\mu-i}$, and (ii) a univariate PCS batch evaluation protocol that is simply a *single* call to the FRI protocol.

C Unrolled and optimized HyperPlonk

In Figure 11, we present an optimized and batched version of HyperPlonk. The protocol batches the zerochecks and additionally batches all evaluations using $\mathcal{R}_{\text{BATCH}}$.

Proof size analysis of compiled protocol We analyze the concrete proof size of the batched PIOP. We analyze the proof size after compilation, i.e., where the prover sends commitments and performs evaluation proofs. The analysis assumes that there are more wires than selectors, i.e., $\ell_w \geq \ell_q$. The prover sends the following elements in each round:

1. 1 multi-linear commitment to w
2. 1 multi-linear commitment to \tilde{v}
3. $\mu + \nu_w + 1$ commitments to univariate degree $d - 2$ polynomials. $2(\mu + \nu_w + 1)$ field elements (claimed evaluation of the polynomial) from the first sumcheck.
4. $8 + \ell_w + \ell_q$ claimed evaluations.
5. 1 univariate evaluation of a batched degree d polynomial.
6. $2 \cdot (\mu + \nu_w + 1 + \lceil \log_2(8 + \ell_w + \ell_q) \rceil) \leq 2\mu + 4\nu_w + 6$ field elements (claimed evaluation of the polynomial) from the second sumcheck.
7. 1 multi-linear evaluation of a batched $\mu + \nu_w + 1$ -variate polynomial.

The total proof size is thus bounded by 2 multi-linear commitments, $\mu + \nu_w + 1$ univariate commitments, 1 multi-linear evaluation proof (for a $\mu + \nu_w + 1$ -variate polynomial) and $4\mu + \ell_w + \ell_q + 6\nu_w + 16$ field elements. For KZG-based commitments, this is proportional to 2μ group elements and 4μ field elements. Concretely, for arithmetic circuits we have $\ell_w = \ell_q = 3$. Thus the proof size is $2\mu + 9 \mathbb{G}_1$ elements and $4\mu + 34$ field elements. Using BLS12-381, where \mathbb{G}_1 elements are 48 bytes and field elements are 32 bytes the proof size becomes $224 \cdot \mu + 1520$ bytes. For $\mu = 20$, this is exactly 6000 bytes.

Indexer. \mathcal{I} on an input circuit C the indexer computes the $([[s_{\text{id}}]], [[s_{\sigma}]])) \leftarrow \mathcal{I}_{\text{perm}}(\sigma)$. The oracle output is $([[q]]1, [[s_{\sigma}]], [[s_{\text{id}}]])$, and $q \in \mathcal{F}_{\mu+\nu_q}^{(\leq 1)}$, $s_{\text{id}} \in \mathcal{F}_{\mu+\nu_w}^{(\leq 1)}$, $s_{\sigma} \in \mathcal{F}_{\mu+\nu_w}^{(\leq 1)}$.

The protocol. $\mathcal{P}(\mathbf{gp}, \mathbf{i}, p, w)$ and $\mathcal{V}(\mathbf{gp}, p, [[q]], [[s_{\text{id}}]], [[s_{\sigma}]]))$ run the following protocol.

1. \mathcal{P} sends \mathcal{V} the witness oracle $[[w]]$ where $w \in \mathcal{F}_{\mu+\nu_w}^{(\leq 1)}$.
2. \mathcal{V} sends input challenge $\mathbf{r}_{IO} \xleftarrow{\$} \mathbb{F}^{\nu}$
3. \mathcal{V} sends $\mathcal{R}_{\text{MSET}}^2$ challenges $r_{mset,2}$ to reduce the instance to an $\mathcal{R}_{\text{MSET}}^1$ instance (See Section 3.4).
4. \mathcal{V} sends $\mathcal{R}_{\text{MSET}}^1$ challenge $r_{M,1}$
5. \mathcal{P} computes the product polynomial $\tilde{v} \in \mathcal{F}_{\mu+\nu_w+1}^{(\leq 1)}$ from w , s_{σ} , s_{id} and the challenges $r_{M,2}$, and $r_{M,1}$ and sends $[[\tilde{v}]]$ to \mathcal{V} (See Section 3.3)
6. Verifier sends challenge α to batch two zerochecks, one resulting from the gate identity (see Section 4.2) and one from the productcheck.
7. \mathcal{V} send zerocheck challenge $r_Z \xleftarrow{\$} \mathbb{F}^{\mu}$
8. \mathcal{P} and \mathcal{V} run sumcheck resulting from batched zerocheck. The sumcheck of size $\mu + \nu_w + 1$ and has degree $d + 1$. In each round, the prover sends an oracle to the univariate round polynomial as well as the claimed evaluation. The verifier delays querying the oracles. Similarly, in the last round, the verifier sends the claimed evaluations of all the multilinear polynomials. There are $8 + \ell_w + \ell_q$ total evaluations:
 - 2 + ℓ_w of w (one for the permutation check, 3 from the gate check, one to check the outputs)
 - 5 of \tilde{v} from the product check
 - ℓ_q of q (one per selector)
 - 1 of s_{σ} (from the product check, the verifier can evaluate s_{id} efficiently herself)
9. \mathcal{V} uses the claimed evaluations to verify all previous protocols.
10. \mathcal{P} and \mathcal{V} run the univariate batch-opening algorithm from [19] to reduce all the round polynomial queries to one.
11. \mathcal{P} and \mathcal{V} run $\mathcal{R}_{\text{BATCH}}$ on all $8 + \ell_w + \ell_q$ evaluations using a degree 2, $\mu + \nu_w + 1 + \lceil \log_2(8 + \ell_w + \ell_q) \rceil$ round sum-check. In the protocol, the prover directly transmits the round polynomial using 2 field elements. The verifier can compute the third from the claimed sum.

Figure 11: Combined and batched PIOP for $\mathcal{R}_{\text{PLONK}}$.



geometry