Cross-chain bridges via backwards-compatible SNARKs

Sergio Juárez[1], Mark Blunden[1], Joris Koopman[1]
Anish Mohammed[1], Kapil Shenvi Pause[2], Steve Thakur[2]

[1]Panther Protocol [2]Mozak

Abstract

In recent years, SNARKs have shown great promise as a tool for building trustless bridges to connect the heterogeneous ecosystem of blockchains. Unfortunately, the parameters hardwired for many of the widely used blockchains are incongruous with the conventional SNARKs, which results in unsatisfactory performance. This bottleneck necessitates new proof systems tailored for efficiency in these environments.

The primary focus of this paper is on succinct bridges from Cosmos to Ethereum, which largely boils down to efficient proofs of multiple Ed25519 signatures. However, these techniques can be ported to settings that require succinct proofs of multiple secp256k1 or BLS12-381 signatures.

We describe our succinct validity bridging scheme *Overfly*, which uses the field-agnostic SNARK [Th23a] to circumvent the huge overhead of non-native field arithmetic arising from Ed25519 scalar multiplications in the circuit. We also explore the schemes deVirgo and zkTree ([DD23]), which exploit the parallelization of proof generation and the subsequent aggregation of proofs.

Our benchmarks indicate that it is crucial to sidestep non-native arithmetic to the extent that it is possible[1]. We also found that two or more proof systems need to be securely amalgamated[2] to optimize a succinct validity bridging scheme.

1 Introduction

Since the genesis of the first distributed public ledger (the Bitcoin network) in 2009, many diverse and distinct blockchain ecosystems have emerged. These ecosystems have provided elegant and valuable solutions to preexisting challenges while also addressing issues that have arisen within their own contexts. Web3 is in a constant state of evolution, perpetually demanding further improvements and innovation.

As blockchain ecosystems gained widespread adoption, the concerns of privacy, security, and scalability became prominent. Given the intricacies and potential vulnerabilities of bridging mechanisms, malicious entities have capitalized on security weaknesses to exploit funds or manipulate data during the transfer process. In 2022, multiple cross-chain bridges were compromised, and bridge hacks accounted for 52 percent of all the crypto losses and 64 percent of all the DeFi protocol losses. Besides these security challenges, the existing cross-chain bridges are often centralized and have to deal with poor performance. While addressing the challenges related to privacy, security, and scalability in blockchain networks is vital, an equally important goal is

[1]The need for EVM compatibility makes it impossible to avoid non-native arithmetic *entirely*

[2]The schemes customized for Ed25519 scalar multiplications are different from those well-suited for SHA-512 hashes

connecting the diverse and heterogeneous ecosystems that currently coexist. As things stand, the developers of these ecosystems face challenges in finding secure ways to interact and transact with other ecosystems. The key to achieving scalability within the multi-chain ecosystem lies in succinct, verifiable computation. The objective is to transform these ecosystems from siloed entities into an interconnected network, much like the internet, which evolved from a paradigm where various computer networks lacked standardized ways to communicate with each other. Cross-chain bridges offer a path to reducing fragmentation and aggregating liquidity across different blockchain ecosystems.

Many of the challenges faced by blockchain networks can be addressed via the cryptographic primitives of *zero-knowledge proofs* (ZKPs) and *verifiable computation*. ZKPs enable a *Prover* to convince a *Verifier* of the validity of a statement without revealing any additional information about the data. Verifiable computation allows a Verifier to attest to the validity of an expensive computation without having to run that computation or download a large amount of data. Instead of conducting direct verification on the blockchain, consensus can be demonstrated off-chain and then efficiently validated on-chain at a minimal cost, utilizing succinct proofs of computation through SNARKs. This innovation empowers cross-chain bridges based on SNARKs to amplify the versatility of light client bridges. As a result, chains no longer need to inherently support each other's signature schemes to attain affordable interoperability. These *succinct bridges* leverage reduced verification expenses without introducing new trust assumptions.

In theory, SNARKs enable the design of bridges between different blockchain ecosystems, thus improving the decentralization and security of inter-blockchain communication. In practice, however, such schemes are of little practical value if the proof generation or the verification is prohibitively expensive. The proof generation time for the SNARKs that offer cheap verification is at least linear in the size of the arithmetic circuit, which is a measure of the complexity of the NP-statement to be succinctly proven. Unfortunately, these SNARKs typically require working with finite fields different[3] from those in which the relevant NP-statements are natively defined for many of the widely used blockchains. This results in a mismatch of fields and consequently, in the huge overhead of *non-native* or *foreign-field* arithmetic when conventional SNARKs are used for this purpose, making the resulting NP-statements far too complex and the proof generation prohibitively expensive. Despite several fruitful attempts at mitigating this issue in recent years, non-native arithmetic remains a bottleneck.

While all bridging schemes acknowledge this non-native overhead as a major hurdle to overcome, there are fundamental differences between their attempts to address or mitigate this problem. Some schemes such as *zkTree* (Polymer Labs) and *zk-Bridge* (Polyhedra) (Section 3) tackle this obstacle head-on rather than sidestepping it, relying on massive parallelization and aggregation with SNARKs that work inherently well with these techniques. Others - including ours (Section 3) - attempt to minimize the amount of non-native arithmetic by using field-agnostic[4] SNARKs compatible with arithmetic circuits in the finite fields wherein the relevant NP-statements are natively defined.

We note that we have used the term *succinct validity bridges* rather than the more commonly used term *zk bridges* since we believe the former to be more accurate.

1.1 Succinct proofs of multiple Cosmos signatures

Cosmos uses the EdDSA signature scheme, a variant of Schnorr's proof of knowledge. It is instantiated with the twisted Edward curve Ed25519 with base field $\mathbb{F}_{2^{255}-19}$. For a fixed base

[3] more precisely, FFT-friendly fields

[4] instantiable with any large enough prime finite field

point B in Ed25519, public key P, message M and a signature (R, s), the verification equation is given by

$$[8 \cdot s] * B \stackrel{?}{=} [8] * R + [8 \cdot \text{SHA-512}(R||A||M)] * P.$$

This entails two ED25519 scalar multiplications and one SHA-512 hash within the circuit. These signatures are not aggregable and the EVM does not support Ed25519 pre-compiles, which makes the verification quite expensive. Each Ed25519 verification consumes 500K gas, and verifying an entire light client header would require a minimum of 50 million gas for 100 validators. This cost could escalate to 500 million for larger, more decentralized Cosmos chains featuring 1000 validators.

Hence, it is necessary to get a succinct and cheaply verifiable proof of the validity of these Ed25519 signatures. The naïve approach would be to use a pairing-based SNARK such as Groth16 or PlonK[5], which would yield a small proof that can be efficiently verified on-chain. Unfortunately, the only Snark-friendly curve with EVM precompiles is BN254.

1.2 The overhead of non-native field arithmetic

The primary impediment to succinct bridges from Cosmos is the lack of traditional SNARKs compatible with the field $\mathbb{F}_{2^{255}-19}$ wherein the Cosmos signatures and, hence, the relevant NP-statements are defined. Some of the initial attempts at succinct validity bridges used Groth16 for proofs of Ed25519 signatures, resulting in impractical proof generation times. Simulating the arithmetic of a finite field - in this case $\mathbb{F}_{2^{255}-19}$ - in a circuit defined over a field with a different characteristic is infamously expensive.

In recent years, there has been a concerted effort to devise efficient ways to deal with non-native arithmetic in the circuit. This includes lookup arguments to optimize range checks and the components of conventional hashes. But despite the promising advances along these lines, the non-native arithmetic remains the paramount hurdle for efficient succinct validity bridges from Cosmos. This is also the case for a few other problems with vital real-world applications. These include:

- Succinct proofs of multiple secp256k1 ECDSA signatures
- One-layer recursion with BN254, for aggregation of multiple BN254 SNARK proofs
- Aggregation of multiple BLS signatures instantiated with the curve BLS12-381

The techniques explored in this paper can be efficiently ported to these other settings. However, in many ways, the problem of Ed25519 signatures appears to be less tractable than the others in this family. The curve secp256k1 forms a (non-pairing) 2-cycle with the curve secq256k1, thus supporting schemes based on inner product arguments. BN254 forms a curve 2-cycle with the (non pairing-friendly) curve Grumpkin and supports the GoblinPlonK proof system. On the other hand, the group of Ed25519 points value over its prime field $\mathbb{F}_{2^{255}-19}$ has a non-prime order (co-factor 8) and hence, does not support a curve 2-cycle. Furthermore, unlike BLS signatures, EdDSA signatures are not efficiently aggregable. This leaves us with a few good options for Cosmos bridges, aside from field-agnostic SNARKs compatible with the field $\mathbb{F}_{2^{255}-19}$ and proof systems that inherently work well with massive parallelization.

[5] allows for a universal updateable trusted setup, but is less efficient than Groth16

1.3 Types of approaches

The mismatch between Ed25519's base field and BN254's scalar field entails a huge overhead of non-native field arithmetic, resulting in a prohibitively expensive Prover time with naïve approaches. A far more pragmatic approach is as follows:

Step 1. Use a proof system with a substantially faster Prover, either

(i) by parallelization and proof aggregation or

(ii) by circumventing the huge overhead of non-native arithmetic.

The schemes zkTree ([DD23]) and deVirgo ([XZC+S22]) - explored in Seection 3- use the proving systems Plonky2 and deVirgo, respectively, and fall into the first category. These schemes exploit massive parallelization and aggregation of proofs. zkTree aggregates multiple Plonky2 proofs via recursion. deVirgo exploits the parallelizability of the Virgo proof system.

On the other hand, our approach (Section 2) is decidedly of the latter type. It hinges on the field-agnostic KZG-based SNARK constructed in [Th23a] instantiated with a pairing-friendly outer curve to Ed25519.

Step 2. Simulate the verification within a BN254 circuit with either Groth16 or *Fflonk*[6]. This step will require non-native arithmetic and will result in a blowup of the circuit size. But the Verifier circuit is much smaller than the original circuit, and hence, this is a far smaller price to pay than having the non-native arithmetic blowup *upfront*.

We note that in our scheme, unlike the approaches hinging on hash-based SNARKs, Step 1 results in a reasonably small proof size of 1.4 KB. However, the verification requires a pairing check and a few scalar multiplications in a curve different from BN254. Unless EIP-1962 is approved to provide support for a wider class of curves on the EVM, Step 2 will remain necessary for our succinct validity bridging scheme.

While the crux of this paper is mechanisms to sidestep non-native arithmetic for Cosmos signatures, we note that avoiding non-native arithmetic *entirely* does not appear to be possible. The two components of a signature's verification algorithm - namely, the SHA-512 hash and Ed25519 scalar multiplications - are disparate and require fundamentally different proof systems for optimized efficiency. Thus, the two proof systems need to be securely combined, which entails some non-native arithmetic.

For instance, the SNARK [Th23a] which our succinct validity bridge uses for Ed25519 scalar multiplications is not well-suited for SHA-512 hashes. It requires working with a curve with a base field of bitsize larger than 512, which is disadvantageous compared to schemes over smaller fields when it comes to SHA-512 hashes. Furthermore, although it supports an efficient lookup protocol, the Prover time for lookups depends on the size of the preprocessed table. This makes it infeasible to use lookup tables of size 2^{32} for SHA-512 hashes, which is possible with the lookup protocol **cq** ([EFG22]) instantiated with the curve BN254.

1.4 Structure of the paper

In section 2, we delve into our bridging scheme hinging on the SNARK constructed in ([Th23a]). This approach does not rely on massive parallelization or aggregation via recursion. Instead, it tries to sidestep the overhead of non-native arithmetic through a field-agnostic SNARK. We describe the use of large lookup tables and efficient lookup protocols to make the SHA-512

[6] a version of PlonK optimized for the Verifier

hashes cheaper in the circuit. This section also includes a description of our Rust implementation and some benchmarks.

The third section focuses on recursive SNARKs and the aggregation of SNARK proofs as a mechanism for computing proofs of multiple Ed25519 signatures. We explore the schemes zkTree and deVirgo.

We conclude with section 4, which summarizes our findings and discusses possible directions for future work.

2 Overfly

We have been exploring backwards-compatible proof systems that can create succinct proofs of Ed25519 signatures. More precisely, we have been exploring SNARKs that could work in this setting without incurring the enormous overhead of non-native field arithmetic. Unfortunately, the curve Ed25519 is not well-suited to the traditional SNARKs for the following reasons:

1. It does not admit a curve cycle, since the number of points on this curve is not a prime. This makes recursion based on inner product arguments prohibitively expensive in this specific setting. We note that this is the primary reason the problem of Ed25519 signatures is considered less tractable than that of secp256k1 signatures.

2. The integer $p - 1 := 2^{255} - 20$ has low 2-adicity, which precludes efficient FFTs. Furthermore, it also lacks large smooth divisors, which makes it difficult to construct suitable vanishing polynomials, i.e. high degree sparse polynomials that split completely over $\mathbb{F}_p$. The SNARKs that rely on univariate PCS's and Lagrange bases - which constitute the majority of known SNARKs - explicitly need a large subset of $\mathbb{F}_p$ with a sparse vanishing polynomial. Thus, the structure of p makes these SNARKs all but incompatible with this setting.

There has been an impressive body of work in recent years that avoids FFTs altogether and achieves linear time Provers by using multilinear polynomial commitment schemes as opposed to univariate PCS's. But all of these schemes appear to have the downsides of larger proof sizes and more expensive verification costs. This also makes self-recursion with these SNARKs less efficient. Furthermore, such schemes often entail substantially more MSMs or Merkle-hashes, which is not necessarily a good tradeoff for fewer FFTS. In fact, in most proof systems that hinge on univariate PCS's, benchmarks for circuits of reasonable size (2^{25} or lower) indicate that it is, in fact, the linear time operations such as the MSMs and the Merkle-hashes that dominate the proof generation time, rather than the superlinear FFTs.

In recent work ([Th23a]), with Ed25519 signatures as a primary use case, we constructed a field-agnostic pairing-based SNARK that is meant to be backwards-compatible with hardwired and widely used parameters. In other words, the SNARK can be instantiated with arbitrary pairing-friendly curves endowed with sufficiently large prime scalar fields. In particular, it can be instantiated with pairing-friendly outer curves to Ed25519[7], which are straightforward to construct using the Cocks-Pinch algorithm. Ed25519 allows for outer curves with embedding degrees dividing 12, while the other three allow for outer curves with embedding degrees dividing 6.

For use cases that require EVM compatibility, non-native arithmetic cannot be completely sidestepped unless EIP-1962[8] is approved. The curve BN254 is presently the only curve with EVM precompiles. However, we have had some success deferring the non-native blowup arising

[7] and secp256k1, BN254, BLS12-381 etc.

[8] EVM support for a wider class of pairing-friendly curves

from the scalar multiplications to the very end. Thus, the overhead of non-native arithmetic affects the constant-sized Verifier circuit rather than the initial circuit, which is linear in the number of validator signatures.

2.1 Overview of the SNARK

The scheme [Th23a] is a KZG-based SNARK and, as such, requires a universal updateable common reference string. The CRS is of length linear in the circuit size and is computed via a one-time MPC. The scheme uses Plonkish arithmetization, which allows for custom gates and efficient lookup arguments for subsets and subsequences. It uses the monomial basis and sidesteps the need for smooth-order multiplicative subgroups in the field where the arithmetic circuit is defined. In particular, it is compatible with pairing-friendly outer curves to Ed25519.

The two subprotocols that form the core of this scheme are:

1. The Hadamard product protocol, which shows that three committed univariate polynomials $f_1(X)$, $f_2(X)$, $f_{1,2}(X)$ satisfy the relation

$$f_{1,2}(X) \ = \ f_1 \odot f_2(X) \ := \sum_{j=0}^{\min(\deg(f_1),\deg(f_2))} \texttt{Coef}(f_1, j) \cdot \texttt{Coef}(f_1, j) \cdot X^j.$$

The protocol hinges on the simple observation that for any integer $N \geq \deg(f_2)$ and for any randomly generated challenge γ, the coefficient of the twisted product

$$f_{\Pi,\gamma}(X) := \ f_1(\gamma \cdot X) \cdot X^N \cdot f_2(X^{-1})$$

at X^N coincides with the evaluation $f_1 \odot f_2(\gamma)$.

Unlike PlonK ([GWC19]), this scheme also requires a protocol for degree upper bounds on committed univariate polynomials. We use the one from [Th19] which hinges on the simple observation that $\deg(f) \leq k$ if and only if the rational function $X^k \cdot f(X^{-1})$ is a polynomial.

2. A permutation argument akin to PlonK's ([GWC19]), but on the coefficients.

This argument hinges on the observation that for a committed univariate polynomial $f(X)$ of degree $\leq N-1$ and a permutation $\boldsymbol{\sigma} : \ [0, \ N-1] \ \longrightarrow \ [0, \ N-1]$ committed via the polynomial $S_{\boldsymbol{\sigma}}(X) \ := \ \sum_{j=0}^{N-1} \boldsymbol{\sigma}(j) \cdot X^j$, the following are equivalent:

(i) $\boldsymbol{\sigma}(f) = f$ in the sense that $\texttt{Coef}(f \ , \ j) \ = \ \texttt{Coef}(f \ , \ \boldsymbol{\sigma}(j))$ for every index j.

(ii) For randomly generated challenges $\beta, \gamma \in \mathbb{F}_p$, the Prover knows a polynomial $F_{\beta,\gamma}(X)$ of degree $\leq N-1$ with cyclic right shift

$$F_{\beta,\gamma}^{\texttt{RShift}}(X) := F_{\beta,\gamma}(X) \cdot X \ (\text{mod } X^N - 1)$$

such that the polynomials obtained from the Hadamard products

$$\Big[f(X) + \beta \cdot \sum_{j=0}^{N-1} j \cdot X^j + \gamma \cdot \sum_{j=0}^{N-1} X^j \Big] \ \odot \ F_{\beta,\gamma}(X) \quad \text{and}$$

$$\Big[f(X) + \beta \cdot \sum_{j=0}^{N-1} \boldsymbol{\sigma}(j) \cdot X^j + \gamma \cdot \sum_{j=0}^{N-1} X^j \Big] \ \odot \ F_{\beta,\gamma}^{\texttt{RShift}}(X)$$

coincide.

The multiple Hadamard products in the protocol are batched. The resulting randomized sum of twisted polynomial products - computed using multimodular FFTs - is the only superlinear computation in the proof generation algorithm.

2.1.1 Proof size and Prover time

The SNARK uses the KZG polynomial commitment scheme over a 9-limb curve outer curve E to Ed25519. In other words, E is a curve over a prime field $\mathbb{F}_q$ of bit-size between 512 and 576 such that the group $E(\mathbb{F}_q)$ of the $\mathbb{F}_q$-valued points on E is divisible by $2^{255} - 19$. We describe the construction of this curve and its optimal ate-paring in the next subsection.

The proof size in the version of the SNARK optimized for the proof size and the verification time consists of 10 $\mathbb{G}_1$, 20 $\mathbb{F}_p$ elements. The Prover time is dominated by:

- the 10 $\mathbb{G}_1$-MSMs in the curve E, with a combined MSM length of $22 \cdot |\text{Circuit}|$
- (to a lesser extent) a single sum of six polynomial products over the field $\mathbb{F}_{2^{255}-19}$, some of them of degree $3 \cdot |\text{Circuit}|$

On the other hand, the proof size in the version optimized for the Prover time consists of 14 $\mathbb{G}_1$, 24 $\mathbb{F}_p$ elements. The Prover time is dominated by:

- the 14 $\mathbb{G}_1$-MSMs in the curve E,, each of length $|\text{Circuit}|$
- (to a lesser extent) a single sum of ten polynomial products over the field $\mathbb{F}_{2^{255}-19}$, each of them of degree $|\text{Circuit}|$.

For comparison, the widely used version of PlonK ([GWC19]) optimized for the Prover time has a proof size of 9 $\mathbb{G}_1$, 7 $\mathbb{F}_p$.

The SNARK [Th23a] does not have a linear time Prover, as is the case with most of the schemes based on multilinear PCS's. However, the only superlinear computation the Prover performs is a single sum of polynomial products over the scalar field. We found that the simple multimodular FFT[9] algorithm outperforms the other alternatives such as Schönhage-Strassen ([SS71])) or the ECFFT ([BCKL21]) for this computation.

We empirically found that the MSMs are the bottleneck rather than the (multimodular) FFTs for the use cases and the circuit sizes we care about. This is not surprising, perhaps, since the base fields in curves constructed via the Cocks-Pinch or the Brezing-Weng algorithms are at least twice as large as the scalar fields (in bitsize).

2.1.2 Verifier costs

The verification consists of a few scalar multiplications and a single pairing check of the form

$$\mathbf{e}(\mathbf{Q}\ ,\ \mathbf{g}_2) \stackrel{?}{=} \mathbf{e}(\mathbf{A}\ ,\ \mathbf{g}_2^{\mathbf{s}})$$

where $\mathbf{g}_2$ is the fixed generator of $\mathbb{G}_2$, $\mathbf{s}$ is the trapdoor, and $\mathbf{Q}$, $\mathbf{A}$ are $\mathbb{G}_1$-elements constructed from those sent by the Prover. In particular, the verification does not involve pairings with Prover defined $\mathbb{G}_2$ points, which is helpful for aggregation of Verifier checks - both on bare metal or in a recursive circuit. This property is particularly helpful when multiple proofs need to be aggregated within a non-native BN254 circuit.

2.2 SHA-512 hashes

The other key part of Ed25519 signatures is the set of SHA-512 hashes. While some of the newer implementations of EdDSA use SNARK-friendly hashes such as Poseidon or Rescue, the same cannot be said of Cosmos's deployed version, which uses the older and more battle-tested hash SHA-512. These hashes are infamously expensive in the circuit and naïvely, would account

[9]The Prover uses ordinary FFTs when the scalar field has high enough 2-adicity

for around 50K gates per hash. Although dwarfed by the cost of a single Ed25519 scalar multiplication in the non-native circuit, this is still quite expensive.

With large lookup tables, this can be brought down to far fewer gates. While the scheme [Th23a] supports lookups for subsets and subsequences, it offers no advantage over PlonK instantiated with BN254 when it comes to the SHA-512 hashes. PlonK in the smaller curve BN254 is substantially faster as a result of the cheaper field operations in the 4-limb base field and the Plonk proof having fewer MSMs than [Th23a]. Furthermore, KZG Plonk supports the lookup protocol **cq** ([EFG22]) whereby the Prover time is independent[10] of the sizes of the lookup tables, which is something the scheme [Th23a] currently lacks. To this end, our bridging design uses:

- [Th23a] instantiated with an outer curve to Ed25519 for the subcircuit defined by the scalar multiplications from the signatures

- PlonK ([GWC19]) instantiated with BN254 for the SHA-512 hashes. The protocol **cq** ([EFG22]) allows us to use large lookup tables for the operators XOR, Maj, Choose, which greatly reduces the size of this circuit

The two proof systems are then securely combined to show that the scalars in the first circuit coincide with the SHA-512 hashes computed in the second. It is necessary to show that the SHA-512 hashes in this circuit coincide with the corresponding scalars the public keys are multiplied within the other circuit.

2.2.1 Lookups for SHA-512

The SHA-512 hash is infamously expensive in a circuit, costing around 50K gates per hash without any optimizations. A commonly used technique is reducing this circuit size using lookup tables for the binary operators such as $\oplus$ (XOR), $\wedge$ (AND), $\texttt{Maj}$ etc. We slightly adapt the techniques from Aztec's Barratenberg library for the SHA-512 hash.

We fix a base $\mathbf{b} \in \mathbb{F}_p$, which in practice will be 7 or 11. For an element $a = \sum_{j=0}^{k-1} 2^j \cdot a_j$ ($a_j \in \{0,1\}$), we call the element $\texttt{sp}(a) = \sum_{j=0}^{k-1} \mathbf{b}^j \cdot a_j \in \mathbb{F}_p$ the *sparse representation* of a. We denote the two-column table of elements $[0,\ 2^k - 1]$ and their sparse representations by $\mathbf{T}_{\text{sparse}}$. Furthermore, set

$$\mathbf{T}_{\text{3XOR}} := \Big\{ \sum_{j=0}^{k-1} \mathbf{b}^j \cdot a_j :\ a_j \in \{0,2\} \Big\} \ ,\ \ \mathbf{T}_{\texttt{Maj}} := \Big\{ \sum_{j=0}^{k-1} \mathbf{b}^j \cdot a_j :\ a_j \in \{0,1\} \Big\} \ .$$

To work with these relatively small tables, we exploit the following equations for elements A, B, C, O in $[0,\ 2^k - 1] \subseteq\ \mathbb{F}_p$:

$$A \oplus B \oplus C = O \iff \texttt{sp}(A) + \texttt{sp}(B) + \texttt{sp}(C) - \texttt{sp}(O) \in \mathbf{T}_{\text{3XOR}}$$

$$\texttt{Maj}(A,B,C) = O \iff \texttt{sp}(A) + \texttt{sp}(B) + \texttt{sp}(C) - 2 \cdot \texttt{sp}(O) \in \mathbf{T}_{\texttt{Maj}}$$

$$A \wedge B = O \iff \texttt{sp}(A) + \texttt{sp}(B) - \texttt{sp}(O) \in \mathbf{T}_{\texttt{Maj}}$$

The tables $\mathbf{T}_{\text{sparse}}$, $\mathbf{T}_{\texttt{Maj}}$, $\mathbf{T}_{\text{3XOR}}$ each consist of 2^k rows, whereas the naïve approach would entail tables with 2^{2k} rows.

With lookup tables of size 2^{16}, the circuit size resulting from the SHA-512 hashes can be brought down to around 12K gates per hash. Furthermore, with lookup tables of size 2^{32}, it can

[10] after preprocessing

be brought down to 6K gates per hash. We note that in the latter case, the lookup tables result in an optimized SNARK Prover time only if the lookup protocol used is one where the Prover time is independent of the circuit size after preprocessing. Arguably, the most efficient protocol for this purpose is **cq** ([EFG22]) in conjunction with PlonK instantiated with the curve BN254.

The circuit size can be further optimized using the weighted sum protocol from [Th23b]. While this protocol falls short of making addition gates "free" as in Groth16, it eliminates the *intermediate* gates arising from high fan-in additions in Plonkish arithmetic circuits. In particular, this reduces the cost of the several decompositions into k-bit chunks that are necessary for the SHA-512 hashes.

2.3 Simulating the Verifier in a BN254 circuit

EVM compatibility requires our Verifier to be simulated within a BN254 circuit for on-chain verification. Unlike the hash-based schemes, the proof size (1.4 KB[11]) and the verification times (10 milliseconds) of our scheme are quite efficient. However, unless/until EIP-1962 is approved, the pairing check and the scalar multiplications that our scheme requires will not be compatible with the EVM. As things stand, the only pairing-friendly curve supported by the EVM is BN254, and hence, our Verifier needs to be simulated in a BN254 circuit, which entails non-native field arithmetic. However, we note that this circuit is constant-sized, and hence, the overhead of non-native arithmetic is far lower than that in the approach where non-native field arithmetic is handled *upfront*, where the circuit size is linear in the number of validator signatures.

The verification cost in [Th23a] is dominated by a single pairing check, i.e. two pairings. As usual, the two final exponentiations can be batched. The verification does not involve pairings with Prover-defined $\mathbb{G}_2$ points, which makes recursive aggregation of proofs convenient. This seems particularly important for this use case since multiple pairings in the *non-native* BN254 circuit would be prohibitively expensive.

We intend to use one of the following options, each of which has certain strengths and weaknesses:

1. Groth16 in BN254

This approach requires a per-circuit trusted setup, which is not updateable. It also does not support efficient lookups, which is not ideal when dealing with non-native field arithmetic. However, the RapidSnark implementation of Groth16 is substantially faster than any other alternative. The fact that the addition gates are free is a particularly important advantage in this setting.

2. PlonK instantiated with the curve BN254; lookup tables of size 2^{32}; the lookup protocol **cq** ([EFG22])

This allows for a universal updateable trusted setup. Furthermore, the lookup protocol **cq** greatly reduces the circuit size with large lookup tables. However, PlonK is significantly slower than Groth16. The addition gates no longer being free is a major downside, especially for circuits that involve a high amount of non-native field arithmetic and, thus, have a lot of high fan-in weighted sums.

While we have been unable to completely neutralize the edge that R1CS has over Plonkish arithmetization regarding the cost of addition gates, we have had some partial success along these lines via the weighted sum protocol described in [Th23b]. An inherent limitation of this

[11] with a pairing-friendly 9-limb outer curve to Ed25519

protocol is that if a wire appears in k linear relations, it needs to appear k times in the circuit. In this sense, it falls short of making addition gates free, as in Groth16. However, it mitigates the problem by removing the intermediate gates arising from the several high fan-in linear relations in Plonkish circuits. In particular, this seems useful for circuits that involve a high amount of non-native field arithmetic or matrix multiplications.

2.4 Overview of the implementation

We now describe our Rust implementation of [Th23a]. We start out with the curve construction, followed by the ate pairing in the curve. We also briefly describe the multimodular FFTs used to compute the superlinear part of the proof generation, namely the randomized sum of twisted polynomial products arising from the batched Hadamard products. Lastly, we summarize the benchmarks.

2.4.1 Curve construction

The first step towards using [Th23a] for Cosmos signatures was the construction of a pairing-friendly outer curve to ED25519. This was followed by the Rust implementation of the ate pairing in this curve. We used the ECFactory to find such a curve using the Cocks-Pinch algorithm. This well-known algorithm yielded a curve E over a 9-limb prime field $\mathbb{F}_q$ with the following properties:

• the embedding degree with respect to the prime $p := 2^{255} - 19$ is 6.

We originally constructed an outer curve with embedding degree 12, with a view toward 128-bit security. However, EVM compatibility presently requires simulating our Verification algorithm within a BN254 circuit, which has a security level of roughly 100-bits. To that end, a curve over a 9-limb prime field $\mathbb{F}_q$ with embedding degree 6 with respect to the prime p suffices for the targeted security level.

• The curve has equation $y^2 = x^3 + 4$ in the affine form.

An affine equation $y^2 = x^3 + a$ ensures that the curve has j-invariant 0 and discriminant -3. The endomorphism algebra is given by $\mathbb{Q}(\sqrt{-3})$, which contains the primitive sixth roots of unity and hence, allows for twists of degrees 2, 3 and 6 over the ground field $\mathbb{F}_q$.

• We ensured that $q - 1$ was easy to factorize and consequently, finding a generator of the multiplicative group $\mathbb{F}_q^\times$ was not too expensive.

We now describe the constructions of the cyclic groups $\mathbb{G}_1$ and $\mathbb{G}_2$ of order p. Group G1 is straightforward to construct since it is a subgroup of E(Fq) order p=225519. The group $\mathbb{G}_2$ is typically chosen to be an order p subgroup of some *twist* of E, i.e. a curve E' over some extension $\mathbb{F}_{q^k}$ such that:

- The product $k \cdot d$ is the embedding degree of E with respect to p (in this case, 6).

- the curves E, E' are isogenous after base changes to the extension $\mathbb{F}_{q^{k \cdot d}}$.

To minimize the number of operations over extension fields of $\mathbb{F}_q$, we opted for $k = 1$ and $d = 6$, constructing a sextic twist E' over the ground field $\mathbb{F}_q$. The output of the pairing lies in the order p multiplicative subgroup of $\mathbb{F}^*_{q^6}$, which necessitates constructing and working with this field extension. Rather than constructing it directly as the splitting field of any irreducible degree 6 polynomial, it's beneficial to do so via a tower of extensions. To exploit Arkworks's optimizations, we chose the cubic extension $\mathbb{F}_{q^3}$ as the intermediate field. For any element $u \in \mathbb{F}^*_q$

that is neither a square nor a cube, the polynomials $X^6 u$ and $X^3 u$ are irreducible and have splitting fields $\mathbb{F}_{q^6}$ and $\mathbb{F}_{q^3}$, respectively. For an element $\alpha \in \mathbb{F}_{q^6}$ such that $\alpha^6 = u$, the extension fields are given by

$$\mathbb{F}_{q^6} = \mathbb{F}_q[\alpha] \ , \ \mathbb{F}_{q^3} = \mathbb{F}_q[\alpha^2].$$

The first part of the pairing computation is the Miller loop. The ate pairing has a Miller loop with length $(q-1) \pmod p$. Highly optimized curves families such as BLS, BN and MNT enjoy the benefit of this integer being small, which makes the Miller loops highly efficient. Unfortunately, that does not hold for curves constructed via the Cocks-Pinch algorithm.

As in [Ver08], we compute relatively small integers $a_0,\ a_1$ such that $a_0 + a_1 \cdot q \equiv 0 \mod r$ using the LLL reduction algorithm. We then compute two Miller loops[12] with lengths a_0 and a_1 respectively. The extra line functions are vertical, which makes the Miller loop computation cheaper.

Once the Miller loops have been computed with output $f \in \mathbb{F}_{q^6}^*$, we need to compute the final exponentiation $f^{(q^6-1)/p}$, which lands in the subgroup of order p. The "easy" part

$$f_{\text{easy}} := f^{(q^3-1)(q+1)} \ \in \ \mathbb{F}_{q^6}^*$$

- as the name suggests - is straightforward to compute using the Frobenius of $\mathbb{F}_q$. We then decompose the exponent as

$$\frac{q^2-q+1}{p} \ = \ c_0 + c_1 \cdot q \quad , \ c_0,\ c_1 \ \in \ [0,\ q-1]$$

and compute the "hard" part

$$f^{\frac{q^6-1}{p}} \ = \ \left(f_{\text{easy}}\right)^{\frac{q^2-q+1}{p}} \ \in \ \mu_p(\mathbb{F}_{q^6}^*)$$

by performing the exponentiations to the c_i. The unique subgroup of order $\Phi_6(q) = q^2 - q + 1$ in $\mathbb{F}_q^*$ is called the *cyclotomic subgroup*, and it is well-known that squarings in this group can be optimized using techniques such as those of [Kar10].

With this implementation, a pairing takes less than 4.7 milliseconds on an 8-core laptop. The SNARK verification is dominated by a single pairing check, i.e. two pairings. The Miller loops in these two pairings can be parallelized, and the two final exponentiations can be batched into one.

2.4.2 Multimodular FFTs

A consequence of using the monomial basis as in [Th23a] is that the only superlinear computations the Prover performs are products of polynomials in $\mathbb{F}_p[X]$. The simplest and the most efficient way to do so is to use multimodular FFTs (terminology as in the NTL library).

We fix highly 2-adic primes $p_1,\ p_2$ such that $p < \min(p_1, p_2)$ and the product $p_1 \cdot p_2$ is larger than $p^2 \cdot M$ where M is an upper bound on the degrees of any polynomials to be multiplied. Given polynomials $f_1(X),\ f_2(X) \ \in \ \mathbb{F}_p[X]$, a the Prover computes the product $f_1(X) \cdot f_2(X)$ as follows.

1. Lift the polynomials $f_1(X),\ f_2(X)$ to characteristic zero. Denote these polynomials by $\widetilde{f}_1(X),\ \widetilde{f}_2(X) \ \in \ \mathbb{Z}[X]$

2. Compute the polynomials $\widetilde{f}_1(X) \cdot \widetilde{f}_2(X) \pmod{p_i \cdot \mathbb{Z}[X]}$ using FFTs in $\mathbb{F}_{p_i}$ for $i = 1,\ 2$.

[12] this can be parallelized

3. Use the Chinese remainder theorem on the coefficients of these polynomials to recover the polynomial $\widetilde{f}_{1,2}(X) := \widetilde{f}_1(X) \cdot \widetilde{f}_2(X) \in \mathbb{Z}[X]$.

4. Reduce the coefficients of $\widetilde{f}_{1,2}(X)$ modulo p to retrieve the $\mathbb{F}_p$-polynomial $f_1(X) \cdot f_2(X)$.

Computing a sum $\sum_{j=1}^{k} f_{j,1}(X) \cdot f_{j,2}(X)$ entails k (parallelizable) FFTs and a single inverse FFT *per prime modulus used*, followed by the Chinese remainder theorem and reduction of the coefficients modulo p.

2.4.3 SNARK implementation and benchmarks

As described in the preceding subsection, Panther's implementation uses a 570-bit outer curve to Ed25519 constructed via the Cocks-Pinch algorithm. The curve has embedding degree 6 with respect to the prime $2^{255} - 19$ and has discriminant -3. At the moment, a million gate circuit has the following Prover times on a 64 vCPU AWS machine:

- 32 seconds for the version optimized for the proof size and the verification time
- 22 seconds for the version optimized for the Prover time.

With a 5-ary (4-inputs, 1-output) circuit and custom gates, each Ed25519 doubling/addition entails 2 gates. Furthermore, the scalar multiplications corresponding to the Ed25519 signature verifications can be batched together using a randomly generated challenge. With Pippenger's algorithm within the circuit, this brings the number of point additions/doublings for N signatures down to around $(512 \cdot N)/\log_2(N)$ and the resulting circuit size to around $(1024 \cdot N)/\log_2(N)$ in addition to the bit-decompositions of the SHA-512/256 hashes. Thus, the scalar multiplications arising from 8192 validator signatures entail a circuit size of roughly 1.7 million gates with a 5-ary circuit.

3 Succinct validity bridges via parallelization and aggregation

As outlined in the introduction, one way to tackle the challenge of verifying multiple Ed25519 signatures is through extensive parallelization followed by aggregation. In this section, we will delve into the schemes zkTree and deVirgo. The former uses Plonky2, a FRI-based SNARK with a low recursion threshold that allows for efficient proof aggregation and compression via self-recursion. To that end, we commence this section with a brief overview of recursive SNARKs. Subsequently, we examine deVirgo's approach, which utilizes aggregation by a centralized entity, albeit without recursion.

3.1 Recursive SNARKs

SNARKs allow a Verifier to succinctly verify the correctness of a computation made by an untrusted Prover. Verifying the SNARK proof constitutes a computation in its own right. Consequently, a SNARK can generate a proof proving that the proof generated by another SNARK was valid, i.e. a proof of a proof. The verification of this second proof legitimizes the first one. This concept is known as a recursive SNARK. We now describe two broad categories of ways recursion can be a vital tool.

The first type is *one-by-one* sequential proof generation (aka incrementally verifiable computation), where for each index k, the proof π_k verifies the preceding proof π_{k-1} and by induction, attests to the validity of all preceding proofs. The left side of Figure 1 shows a simple two-level recursion, where the Prover takes a *public input* x and a *witness* w to produce a proof π, then $Prover'$ takes

public input x' and proof π to produce proof π'. Finally, verification of π' implies verification of π.

The second type of batched proof generation is the aggregation of multiple proofs into a single proof. This has the structure of a binary tree, where each *parent proof* (node) verifies a *left child proof* and a *right child proof*, see Right side Figure 1 which shows that the verification of proof π implies verification of both proof π_1 and proof π_2.

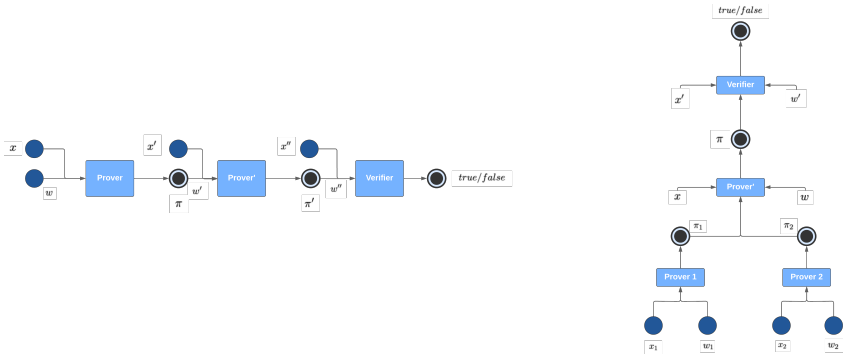

Figure 1: Left) Sequential recursive SNARK scheme of two layers. Right) Recursive aggregation SNARK scheme

Recursive schemes based on elliptic curves suffer from the mismatch of the scalar and base fields of the curves, which entails some non-native arithmetic. FRI-based recursive schemes sidestep this problem since the circuits in each recursive layer are defined over the same field.

3.1.1 Aggregating SNARK proofs via recursion

The natural data structure for aggregation via recursion is a tree, as suggested by figure **??** and as it is depicted by figure 2, where each node represents a SNARK proof, and each parent node recursively verifies the SNARK proofs of its children. Verifying the "root proof" suffices to determine that all leaf proofs were valid. Subsequently, the Prover can provide a Merkle membership proof for the corresponding leaf to show that a specific valid proof was contained in the tree.

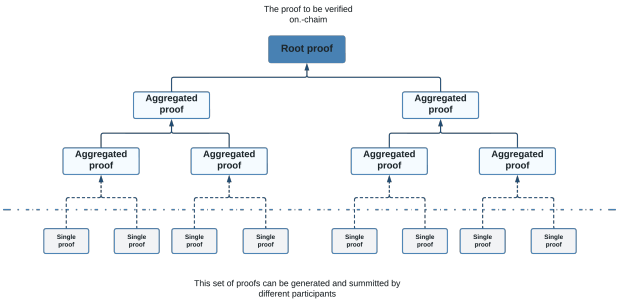

Figure 2: Structure of a tree of zk-proofs.

A model to construct a tree consisting of SNARK proofs is presented in [DD23]. It involves three types of proofs: *user proof* π, *leaf proof* υ, and *node proof* ω.

A delegated proof π_i may be generated from various circuits using different SNARK schemes

and configurations. Each π_i is associated with a unique $\texttt{VD}_i$[13]. Distinct υ_i proofs are produced by different Leaf circuits of the same SNARK scheme, while separate ω_i proofs are generated by the same Node circuit in the same SNARK scheme as υ_i. The tree is constructed using the sets π, υ, and ω, subject to the following constraints:

- Proofs are initially generated by undelegated Provers/Nodes. To generate a proof π_i for inclusion in a tree with public inputs x_i and Verifier data $\texttt{VD}_i$, Prover i executes:

$$P_i(x_i) \longrightarrow (\pi_i, \texttt{VD}_i).$$

  As mentioned earlier, these proofs can be generated independently. Furthermore, they can be generated using different proof systems and configurations. The key requirement is to ensure that each π_i can be verified using the corresponding $\texttt{VD}_i$. This flexibility allows for the integration of proofs generated from diverse sources and configurations into the tree, enabling efficient verification of a wide range of proofs while maintaining compatibility between the proofs and their respective Verifier data.

- Leaves of the tree are built from the user proofs. The tree Leaf builder takes a user proof along with the public input x_i and the verified data $\texttt{VD}_i$ to yield a 3-tuple:

$$L_i(\pi_i, x_i, \texttt{VD}_i) \longrightarrow (\upsilon_i, h_i, c_i)$$

  to verify π_i with other inputs and generate the leaf proof υ_i. Here $c_i = H(\texttt{VD}_\ell || \texttt{VD}_i)$, where $\texttt{VD}_\ell$ is the hash of Verifier data of the Leaf circuit, and $h_i = H(x_i)$ is the hash of all the Prover proof's public inputs.

- Nodes arise from various potential combinations. A node can be constructed by combining two leaves, a leaf, and a node, or two nodes. These combinations facilitate the formation of the hierarchical tree structure, with nodes symbolizing the aggregation of the children elements. So, regarding the aforementioned possibilities, the node builder looks like any of the following expressions:

$$N(\upsilon_i, h_i, c_i, \upsilon_j, h_j, c_j) \longrightarrow (\omega_k, h_k, c_k)$$
$$N(\upsilon_i, h_i, c_i, \omega_j, h_j, c_j) \longrightarrow (\omega_k, h_k, c_k)$$
$$N(\omega_i, h_i, c_i, \omega_j, h_j, c_j) \longrightarrow (\omega_k, h_k, c_k)$$

Concisely, a finite set of delegated Prover proofs is transformed into leaves. Subsequently, nodes are constructed recursively from these leaves by combining leaves with nodes or nodes with nodes, culminating in the creation of the tree's root. This sequential process gives rise to the tree's structure.

The recursive tree structure for SNARK proofs facilitates parallel proof generation at each level. The time for generating the SNARK tree, considering n delegated Provers, is given by log(n)t, where t is the time required for proving a node (dependent on the specific proof system), alongside the time needed for transmitting a node proof between workers.

[13] Verifier Data

3.1.2 zkTree: aggregating Cosmos signatures via recursion

Aggregation of SNARK proofs via recursion is an extremely useful tool when a larger circuit can be decomposed into several subcircuits with several Provers working in parallel. While this approach does not reduce the total computation, it results in a lower *effective* runtime by making the computation much more parallelizable. This is made possible by schemes such as Plonky2 with a low recursion threshold. Rather than proving N Ed25519 signatures in a single non-native circuit, a Prover may compute the N distinct proofs of valid signatures on N distinct machines in parallel. These proofs can then be aggregated into a single proof via recursion.

The EdDSA digital signature scheme used by Tendermint is quite expensive to express as an arithmetic circuit in a field other than the field $\mathbb{F}_{2^{255}-19}$ (which the EVM does not support). A single signature verification can be over two million gates in a non-native circuit. Furthermore, the EdDSA signatures are not efficiently aggregable and even with optimizations such as Pippenger's algorithm in the circuit, N of these signatures in a circuit cost at least $\frac{N}{\log(N)}\times$ compared to the cost of a single signature. This is where aggregation of proofs via recursion offers a possible solution.

To generate a proof of N Ed25519 signatures from a Cosmos block header, we may generate (in parallel) N proofs $\{\pi_1, \cdots, \pi_N\}$ of the validity of those individual signatures. We may then use recursion to aggregate these proofs as follows:

- The N proofs of valid signatures will be aggregated into a single proof via repeated self-recursion.
- The verification of the proof obtained in the first step will be simulated within a BN254 circuit (either Groth16 or Fflonk). This constant-sized proof can then be verified on-chain.

In theory, recursive SNARKs are relatively simple since we only need to compose proof systems in such a way that the output of one SNARK becomes the input of another. In practice, however, this is only optimal when the recursion threshold is low. For instance, the recursive schemes based on amicable pairs of elliptic curves do not yield a good performance in this setting. The mismatch between the base field and the scalar field of a curve entails a fair amount of non-native arithmetic. Furthermore, this requires working with large curves with expensive MSMs, which results in a high Prover time.

3.1.3 Plonky2

The recursive SNARK used in zkTree is Plonky2. This is a proof system with a low recursion threshold and supports arbitrarily deep recursion. It combines Plonkish arithmetization - which allows for custom gates - with an FRI-based polynomial commitment scheme. As such, it is compatible with any FFT-friendly finite field of a suitable size[14].

Plonky2's polynomial commitment scheme operates within a small characteristic field, allowing witnesses to be encoded in 64-bit field elements. The prime field, referred to as the Goldilocks field, has a size of $2^{64} - 2^{32} + 1$. This design choice enhances Prover performance by using native CPU operations.

Plonky2 typically uses circuit-friendly hashes such as Poseidon since the conventional hashes such as SHA or Keccak would be expensive in the recursive circuits. The Poseidon configuration in Plonky2 has 8 full rounds and 22 partial rounds, with a width of 12 field elements. Since 7 is the smallest prime that does not divide $2^{64} - 2^{32}$, the S-box used is given by $x \mapsto x^7$.

[14] using suitable field extensions when necessary

There is a trade-off between the proof size and the Prover performance of Plonky2. A larger blowup factor yields a larger proof size but a faster prover time. Nevertheless, Plonky2 can compress extensive proofs into a constant size, typically around 43 KB. The Plonky2 paper reports that it takes approximately 300 ms to generate a recursive proof on a MacBook Air 2021. The recursion threshold for Plonky2 is 2^{12} gates.

Naïvely, a proof of 100 Ed25519 signatures would take $100 \cdot 30 = 3000$ seconds. However, the total time required for recursively composing Plonky2 proofs with parallelism exploited to the fullest consists of 1 time unit for generating the initial Plonky2 proof, in addition to $\log_2(N)$ time units for generating the recursive Plonky2 proof. Leveraging Plonky2's rapid recursion time of 1 second per proof, the overall time needed to prove the Tendermint light client would be approximately 30 seconds $+ \log(100) * 1$ second $\approx$ 37 seconds .

3.2 deVirgo

Consider a scenario where a particular circuit $\mathcal{C}$ comprises numerous instances of identical *sub-circuits*. Such circuits are known as data-parallel circuits due to their nature. Assuming that a data-parallel circuit is composed by exactly k copies of a sub-circuit, each copy can be processed independently due to the lack of connections or dependencies. Therefore, assuming the availability of k machines $\{\mathcal{M}_0, \cdots, \mathcal{M}_{k-1}\}$, we can distribute the workload among these machines and process the sub-circuits in parallel. The deVirgo proof system presents a construction that builds upon the Virgo proof system while harnessing parallel and aggregation processing capabilities. Hence, each node or machine produces a Virgo proof for a sub-circuit, accelerating the Virgo scheme by a factor of k.

Instead of recursion, deVirgo introduces a protocol that enables machines to operate autonomously and communicate with each other. At the culmination of this process, the *master node* $\mathcal{M}_0$ aggregates all proofs into a single proof. Essentially, this proof consists of the concatenation of the k Virgo proofs. However, a notable drawback is that the size of the proof increases linearly with the number of sub-circuit copies in the original circuit.

To address this challenge and to mitigate the effect of the factor k on the final proof size, deVirgo implements modifications to the GKR protocol within the Virgo proof system. These protocols are also adapted into distributed versions. As a result, the master node $\mathcal{M}_0$ transmits the aggregated message to the Verifier in each round, rather than individual machines sending messages directly to the Verifier. This strategy notably expedites the proof generation process for circuits divisible into sub-circuits.

In summary, a master node aggregates all individual proofs into a unified proof, yielding a proof size akin to what Virgo produces for a single proof. The resulting proof is then compressed via Groth16 or PlonK in the EVM-compatible curve BN254. The ultimate result is a distributed proof system that operates k times faster than Virgo while preserving the same compact proof size.

A key drawback is that the process relies heavily on the master node. Although it distributes and parallelizes the proof generation, the designated master node is responsible for receiving all proofs generated by all other nodes and subsequently aggregating the entire set of proofs. On the other hand, the Plonky2 scheme only needs a coordinating entity to orchestrate the participation of nodes and to prevent the simultaneous generation of proofs for the same signature by two or more nodes. In conclusion, both schemes demonstrate a certain degree of centralization, with deVirgo substantially more centralized than zkTree. This communication complexity between the nodes participating in the protocol can be a bottleneck for settings with a large number of

validators. For instance, as noted in [DD23], aggregating 5000 proofs would entail a prohibitively expensive communication complexity of around 5TB.

4 Conclusion and future directions

Despite the concerted effort and several promising advances in this direction, an efficient proof of multiple Ed25519 signatures that is cheap to both compute and to verify on the EVM remains an elusive goal. Reasons for this include the low 2-adicity of the prime $2^{255} - 19$ and the lack of support for a wider class of pairing-friendly curves on the EVM. As outlined in this paper, there are two broad classes of approaches:

Approach 1: Massive parallelization with several machines, enabled either by proof aggregation using a recursive scheme or by the scheme being fully distributed.

Approach 2: Using a field-agnostic SNARK that allows the Prover to sidestep the huge overhead of non-native arithmetic arising from the Ed25519 scalar multiplications.

In both cases, it is currently necessary to simulate the verification within a non-native BN254 circuit with Groth16 or Fflonk. For approach 1, the proof is too large to be verified on-chain except in the case of Pianist which arguably has a proof small enough for on-chain verification. For approach 2, the proof size can be far smaller (1.4 KB using our scheme). But unless EIP-1962 is approved, EVM compatibility requires simulating the verification within a BN254 circuit. With either of these two approaches, the BN254 simulation of the Verifier is far from insignificant on account of the non-native arithmetic. However, the higher the number of signatures of blocks batched together, the less dominant this part of the end-to-end proof generation would be.

Our benchmarks suggest that in settings with a reasonably high number of validators, the second approach yields a better performance for the Prover and entails far lower total CPU time. Aggregation (via recursion or via other means) is relatively less essential - albeit still extremely beneficial - when the circuit size is far smaller by virtue of minimizing the non-native arithmetic.

The question that poses itself here is whether we could combine the strengths of both approaches. In other words, it would be eminently desirable to have an efficient field-agnostic SNARK that also supports massive parallelization, either through efficient aggregation via recursion as in Plonky2 or a fully distributed approach similar to *Pianist* ([LXZ+23]). Work in progress from the Mozak team describes a fully distributed SNARK instantiable with any pairing-friendly curve with a large enough scalar field. This allows us to simultaneously circumvent non-native arithmetic and achieve massive parallelization to generate a succinct proof of the validity of multiple Ed25519 scalar multiplications.

Within the context of field-agnostic SNARKs for Ed25519 signatures, a path worthy of further exploration is that of SNARKs hinging on multilinear polynomial commitment schemes rather than univariate ones. While our benchmarks suggest that univariate PCS's perform better *at the moment*, this could quite plausibly cease to be the case in the near future.

The other component of Ed25519 signatures is the set of SHA-512 hashes within the circuit. Although dwarfed by the cost of Ed25519 scalar multiplications in a non-native circuit, these hashes do entail a significant cost. The proof systems best suited for Ed25519 scalar multiplications are not as well-suited for the SHA-512 hashes, which necessitates using two different proof systems and securely combining them. Currently, our proposed solution is using lookups to make these hashes cheaper within the circuit. In particular, a lookup protocol such as **cq** that allows for a Prover time independent of the table size[15] is useful in this setting. An intriguing path yet to be

[15] after preprocessing

explored is using the binary field SNARK *Binius* ([DP23]) for these hashes, greatly reducing the non-native arithmetic arising from them.

Lastly, unless EIP-1962 comes to fruition, it will remain necessary to simulate the Verifier of some SNARK within a non-native BN254 circuit for on-chain verification. To that end, it is highly desirable to optimize operations such as non-native pairings and the Goldilocks Plonky2 verification algorithm in a BN254 circuit with Groth16 and Fflonk. The former largely boils down to optimizing non-native arithmetic and field extension arithmetic within a circuit. The recent work [TSP24] makes these operations cheaper in Plonkish circuits, but the costs and the efficiency of that protocol with non-KZG polynomial commitment schemes have yet to be studied.

Acknowledgements

We thank Roman Melnikov for fruitful conversations about bridging and for helpful advice regarding the Rust implementation of [Th23a]. We thank Sai Deng for answering our questions regarding [DD23].

References

[AC22] R. Akeela, W. Chen, *Yafa-108/146: Implementing ed25519-embedding Cocks-Pinch curves in Arkworks-rs*, https://eprint.iacr.org/2022/1145

[BCKL21] E. Ben-Sasson, D. Carmon, S. Kopparty, D. Levit, *Elliptic Curve Fast Fourier Transform* (*ECFFT*) *Part I: Fast Polynomial Algorithms over all Finite Fields*, https://arxiv.org/abs/2107.08473

[Bl22] R. Bloemen, *NTT transform argument* (blogpost), https://xn–2-umb.com/22/ntt-argument/

[CBBZ22] B. Chen, B. Bünz, D. Boneh, Z. Zhang, *HyperPlonK: PlonK with Linear-Time Prover and High-Degree Custom Gates*, https://eprint.iacr.org/2022/1355

[CHMMVW20] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely and N.P. Ward. *Marlin: Preprocessing zk-SNARKs with universal and updatable SRS*. Eurocrypt 2020, Part I, volume 12105 of LNCS

[DD23] S. Deng, B. Du, *zkTree: A Zero-Knowledge Recursion Tree with ZKP Membership Proofs*, Polymer Labs, 2023. https://eprint.iacr.org/2023/208

[DP23] B. Diamond, J. Posen, *Succinct Arguments over Towers of Binary Fields*, https://eprint.iacr.org/2023/1784

[EFG22] L. Eagen, D. Fiore, and A. Gabizon. *cq: Cached quotients for fast lookups*, https://eprint.iacr.org/2022/1763

[EG23] L. Eagen and A. Gabizon, *cqLin: Efficient linear operations on KZG commitments with cached quotients*, https://eprint.iacr.org/2023/393

[GWC19] A. Gabizon, Z. Williamson, O. Ciobotoru, *PlonK: Permutations over Lagrange-bases for Oecumenical Non-interactive arguments of Knowledge*, https://eprint.iacr.org/2019/953

[GW20] A. Gabizon, Z. Williamson, *Plookup: A simplified polynomial protocol for lookup tables*, https://eprint.iacr.org/2020/315

[GW21] A. Gabizon, Z. Williamson, *fflonk: a Fast-Fourier inspired verifier efficient version of PlonK*, https://eprint.iacr.org/2021/1167

[G23] C. Goes, *The Interblockchain Communication Protocol: An Overview*, Interchain GmbH, Berlin, Germany, 2023 https://arxiv.org/abs/2006.15918v1

[HG21] E. Housni, A. Gullevic, *Families of Snark-friendly 2-chains of elliptic curves*, https://eprint.iacr.org/2021/1359

[Kar10] K. Karabina, *Squaring in cyclotomic subgroups*, https://eprint.iacr.org/2010/542

[KZG10] A. Kate, G. Zaverucha, and I. Goldberg. *Constant-size commitments to polynomials and their applications.* In Masayuki Abe, editor, Asiacrypt 2010, volume 6477 of LNCS, pages 177–194. Springer, Heidelberg, December 2010.

[Gab21] A. Gabizon, *Aztec emulated field and group operations*, https://hackmd.io/@arielg/B13JoihA8#Aztec-emulated-field-and-group-operations

[Inb22] K. Inbasekar, *Bridging the Multichain Universe with Zero Knowledge Proofs* (Blogpost)

[KPS18] A. Kosba, C. Papamanthou and E. Shi, *xJsnark: A Framework for Efficient Verifiable Computation*, 2018 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 2018, pp. 944-961, doi: 10.1109/SP.2018.00018.

[Kub22] I. Kubjas, *Notes about optimizing emulated pairing* (*part 1*), https://hackmd.io/@ivokub/SyJRV7ye2

[LXZSZ23] Liu et al, *Pianist: Scalable zkRollups via Fully Distributed ZK Proofs*, https://eprint.iacr.org/2023/1271

[Pol21] Polygon, *Plonky* 2*: Fast recursive arguments with PlonK and FRI*

[SS71] A. Schönhage, V. Strassen. *Schnelle multiplikation großer zahlen.* Computing, 7(34):281–292, 197.

[Sh01] V. Shoup, *The NTL Library*, https://libntl.org/

[Sh20] V. Shoup *Arithmetic Software Libraries*, https://www.shoup.net/papers/akl-chapter.pdf

[Th23a] S. Thakur,, *A flexible Snark via the monomial basis*, https://eprint.iacr.org/2023/1255

[Th23b] ———, *An optimization of the addition gate count in Plonkish circuits*, https://eprint.iacr.org/2023/1264

[TSP24] S. Thakur, K. Shenvi Pause, *Polynomial products in Plonkish circuits*, https://hackmd.io/@3WHkK2CBQrKfvT5B6u-pog/Skv-OoUpT (Blogpost)

[Thl] J. Thaler, *A Note on the GKR Protocol*, https://people.cs.georgetown.edu/jthaler/GKR

[Ver08] F. Vercauteren, *Optimal pairings*, https://eprint.iacr.org/2008/096