Testudo: Linear Time Prover SNARKs with Constant Size Proofs and Square Root Size Universal Setup

Matteo Campanelli[1], Nicolas Gailly[1], Rosario Gennaro[2], Philipp Jovanovic[3], Mara Mihali[4]*, and Justin Thaler[5]

[1] Protocol Labs, {matteo,nikkolasg}@protocol.ai
[2] Protocol Labs & CCNY, rosario.gennaro@protocol.ai
[3] UCL, p.jovanovic@ucl.ac.uk
[4] Aztec Labs mara@aztecprotocol.com
[5] Georgetown & a16z crypto research, justin.thaler@georgetown.edu

Abstract. We present Testudo, a new FFT-less SNARK with a near linear-time prover, constant-time verifier, constant-size proofs and a square-root-size universal setup. Testudo is based on a variant of Spartan [30]–and hence does not require FFTs–as well as a new, fast multivariate polynomial commitment scheme (PCS) with a square-root-sized trusted setup that is derived from PST [27] and IPPs [9]. To achieve constant-size SNARK proofs in Testudo we then combine our PCS openings proofs recursively with a Groth16 SNARK. We also evaluate Testudo and its building blocks: to compute a PCS opening proof for a polynomial of size 2^{25}, our new scheme opening procedure achieves a 110x speed-up compared to PST and 3x compared to Gemini [6], since opening computations are heavily parallelizable and operate on smaller polynomials. Furthermore, a Testudo proof for a witness of size $2^{30} (\approx 1\,GB)$ requires a setup of size only 2^{15} ($\approx$ tens of kilobytes). Finally, we show that a Testudo variant for proving data-parallel computations is almost 10x faster at verifying 2^{10} Poseidon-based Merkle tree opening proofs than the regular version.

* Work done mainly while the author was affiliated with UCL and Protocol Labs.

Table of Contents

1 Introduction

Succinct Non-Interactive Arguments of Knowledge (SNARKs) have been a prolific area of research in the last decade: a SNARK allows a prover to prove to a verifier that a certain (non-deterministic) computation F has been performed correctly, or more specifically that there exists a *witness* w such that $y = F(x, w)$ where x is a public input. The crucial property of SNARKs is that the size of the proof and the verification time should be *short*, i.e., sublinear in the size of the computation F and of the witness w. Otherwise a simple proof would be to send w and have the verifier recompute $F(x, w)$. Additionally, SNARKs can be zero-knowledge, i.e., they do not reveal any information about w.

SNARKs are evaluated according to various performance metrics where the three most important ones are (1) the time it takes the prover to generate a proof, (2) the size of the proof, and (3) the time it takes the verifier to validate a proof. There are various design trade-offs that can be explored to optimize those metrics and an essential distinguishing factor that impacts performance are the preprocessing phases of SNARK systems.

On the one end of the spectrum, there is, for example, the Groth16 SNARK [19] which produces proofs consisting of only 3 group elements and which has a verification that is independent of the complexity of F, only requiring the evaluation of a few pairings. This makes Groth16 essentially optimal in terms of proof size and verifier time but it comes at the expense of a superlinear overhead in prover time and a function-specific trusted setup, the latter of which is a serious drawback in practice. On the other end are (*transparent*) SNARKs that do not require a trusted setup [5, 8, 38] at all but they tend to have larger proofs and verifier times, e.g., at least logarithmic in the size of the witness. Finally, there are (*universal*) SNARKS which offer a compromise, as they permit smaller proofs in comparison to transparent systems at the cost of a single setup that is universal enabling them to prove any circuit up to a certain size [20]. However, these schemes are still suboptimal as they have slower provers than Groth16 [12], which becomes particularly evident for large circuits [40], and they do not have constant size proofs or verification times. Finally, all of the schemes with a trusted setup produce a linear-size common reference string (CRS) which is particularly problematic for large circuits (again), as the CRS has to be downloaded, stored, and moved into RAM at proving time. For example, Filecoin [23] uses Groth16 for circuits of size $\approx 2^{30}$ but producing a trusted setup for this size was practically infeasible. As a workaround, provers work with (sub-)circuits of size $\approx 2^{27}$ and generate ≈ 10 proofs per (large) circuit.

In summary, all of these observations led us to the following research question:

Can we design a SNARK with a small universal trusted setup, constant size proofs and verification time, and a fast prover? [6]

1.1 Contributions

In this section we present our main contributions together with an informal overview of the techniques used to achieve them.

Testudo: **Near-Linear time prover with succinct verification.** To achieve this goal, we start our design from Spartan [30] a sumcheck-based argument requiring only field arithmetic (e.g. much faster that its point counterpart) and a single multilinear polynomial opening. The original Spartan is a transparent scheme which relies only on discrete based log curves and transparent polynomial commitment schemes, giving substantially larger proof sizes and verification times ($\approx \sqrt{N}$). To improve on those, the main idea is to have a Groth16 prover verifying the Spartan proof. Below we describe the technical challenges of this approach, but one important thing to note here is that in order to obtain a ZK-SNARK it is sufficient to run the (much simpler and more efficient) non-ZK version of Spartan since the outer Groth16 proof will hide any information possibly leaked by the non-ZK inner Spartan proof.

Embedding the verification of a Spartan-based SNARK in a Groth16 circuit presents various implementation challenges. The sumcheck component of the proof operates only on field elements and requires the use of a hash function to make it non-interactive via Fiat Shamir. Field operations can be natively encoded in R1CS

[6] To maximize backwards compatibility to already deployed systems, we require that our SNARK system works with R1CS-based circuits.

constraints and we adapted our codebase to use Poseidon, a SNARK friendly hash function, having the advantage of a more efficient representation in a circuit. However, things get more complicated for writing the R1CS constraints for the verification of the polynomial commitment opening, since it requires point arithmetic which, if naively encoded in the circuit, would massively increase the number of constraints by multiple orders of magnitude. This issue would be amplified by the relative large size of Spartan's original commitment's proof size and verification – $O(\sqrt{N})$ – which can potentially be reduced if we leverage using a different PCS with a trusted setup.

Why the name Testudo? Testudo was a type of battle formation that ancient Rome adopted, where its soldiers operated "under the hood" of their shields. Testudo, the proof scheme, is similar: a Spartan prover woking under the hood of Groth16.

Testudo-Comm: New Multivariate Polynomial Commitment Scheme. To reduce the size of the PCS opening, we devise a new polynomial commitment scheme based on PST [27] and inner pairing product [9] that avoids the use of FFTs. Testudo, as Spartan, considers circuits of size $N = 2^n$ where the polynomial representation of the circuit has n variables, n is logarithmic and so N will be linear in the size of the circuit. The high level idea is to express the coefficients of the witness multivariate polynomial from Spartan as a square matrix of size $\sqrt{N} \times \sqrt{N}$. To commit to this polynomial, the prover commits to each row of the matrix using PST, leading to a vector $\vec{A}$ of size $\sqrt{N}$. Then the prover commits to $\vec{A}$ using the MIPP commitment in [9] (e.g. a pairing product between commitments and random base) to create the final commitment T - a single group element. To open, the prover carefully performs PST and MIPP opening on $\sqrt{N}$ sized polynomials with many operations in paralell. Both the MIPP and PST part operate on $\sqrt{N}$ sized polynomials.

Since the 2 opening operations can be done in parallel we obtain a considerable speedup in practice (about 2 orders of magnitude faster) than PST, even though it requires heavier operations like $\log N$ pairings to create the combined commitment. Moreover, when comparing with Gemini [6], we estimate our opening procedure to be 3x faster for large N such as 2^{25}, where [6] is likely faster for small sizes, with the additional cost that it requires to perform FFTs for practical deployment.

Usage of 2-chain curves for efficient verification. To enable verifying group operations in a circuit, we use the standard approach pf running Spartan over a 2-chain curve (such as BLS12-377) where group elements can be encoded as field elements in a companion curve over which the Groth16 prover is then implemented. For backwards compatibility reason, we also explored the possibility to run Testudo on curves without a companion curve (such as BLS12-381, which is currently used by Filecoin proofs).

Aggregation. Additionally, we show how Testudo proofs can be aggregated. Since the outer layer is a Groth16 proof, we can use standard aggregation tools such as SnarkPack [16] to aggregate several Groth16 proofs together. Another option is to aggregate proofs at the inner level and then run a single Groth16 proof on top of the aggregated inner proofs.
Note that an interesting point of this design is that the Groth16 prover can be outsourced to more powerful machines or a "prover-as-a-service" infrastructure. Indeed, if the "Spartan" prover ran the sumcheck and the commitment opening, then resulting proof already hides the witness thanks to the zero knowledge property of Spartan. As a consequence, more powerful machine can aggregate many of these "semi" proofs inside one Groth16 proof, without ever seeing the witness, similar to how Snarkpack works. This settings has practical implications in terms of deployment that we believe are worth exploring.

Analysis and Experimental Results. As explained in the body of the paper, our SNARK avoids the use of FFT altogether and obtains a nearly linear-time prover[7]. In practice, we show that:

– Our polynomial commitment scheme has a commitment time comparable to PST while producing opening proofs at two order of magnitude faster (at the cost of larger proof sizes).
– Our experimental results show that for data parallel circuits, we can estimate Testudo to run more than $\approx$ 5x to $\approx$ 10x faster than the fastest Groth16 implementation (i.e., Bellperson [4]), depending on the size of the small subcircuit. For example, if the sub-circuit is of size 2^{15}, as an upper bound to a circuit verifying

[7] Our prover runs N multi-exponentiations of size N, which is roughly $O(N\frac{\lambda}{\log N})$ group operations with $\lambda > \log N$ for security reason.

a Poseidon based Merkle Tree opening proofs with 32 layers, then Testudo can verify 2^{10} such proofs $\approx$ 9.7x faster than the Groth16 equivalent.

1.2 Related Work

The literature on SNARKs is very large and we refer the reader to Thaler's monograph [34] for a comprehensive survey. In this section we focus on a few works that are relevant to Testudo.

We were inspired to use a 1-level recursion with Groth16 verifying a faster inner SNARK by the work of Belling et al. [3] where the verification of a GKR proof [18] for hash computations is outsourced to a Groth16 prover. Concurrently to our efforts, a similar approach was also taken in the ZKBridge paper [37], where the verification of a Virgo [38] proof is outsourced to a Groth16 prover. Because Virgo is also GKR-based, the underlying SNARK in either case is known to be efficient for large "parallel" computations. We believe we are the first to apply this approach to a general purpose SNARK like Spartan.

When it comes to universal trusted setup proofs, many systems today do not use R1CS but rather "custom gates" (sometimes also called Plonkish arithmeization), and apply SNARKs such as Plonk [15] (or alternatives such as Hyperplonk [11]) to the resulting constraint systems. The use of "custom gates" makes a comparison to pure R1CS-based schemes not immediate. We are still working on achieving meaningful comparisons but we estimate that Testudo is competitive with approaches that do use custom gates. We point out that many applications (including our main motivating one – Filecoin proofs) are already encoded as R1CS systems, and therefore it is very useful to have an efficient SNARK with universal trusted setup that can be used off-the-shelves.

Our new PCS Testudo-Comm leverages ideas from [16] and [9] to reduce the size of the trusted setup for the KZG univariate polynomial commitment [21] to square-root size from linear. We adapted them to achieve the same reduction for the PST commitment. We note that, as far as we know, we are the first to implement these techniques. We also point out that the reduction of in the trusted setup size comes at the expense of larger opening proofs: however in our case that drawback is "absorbed" by the outer Groth16 proof, which compresses the final proof down to constant.

[6] presents a generic transformation to turn a univariate polynomial commitment into a multilinear one. In Section 4 we discuss why we believe using Testudo-Comm is a better choice for us.

2 Preliminaries

We assume the reader is familiar with the definitions of R1CS, Polynomial Commitment Schemes and SNARKS, which we however recall in the Appendix.

2.1 Notation

We assume we have cyclic groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ of order q generated by g and equipped with a bilinear pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. We denote by $p(x_1, \dots, x_n)$ a multilinear polynomial with n variables. For $s_1, \dots, s_n \in \mathbb{Z}_q$ we write $\vec{s} = (s_1, \dots, s_n) \in \mathbb{Z}_q^n$. Let $i \in \{0,1\}^n$, we can denote $i = (i_1 \dots i_n)$ as $i_j \in \{0,1\}$. We denote the value $\prod_j s_j^{i_j}$ by $\vec{s}^{\,i}$

2.2 Cryptographic Assumptions

The security of our constructions holds in the Generic Group Model (GGM) [32]. In Section 4 and Section 5 we rely on the security of the underlying building to claim that of our protocols. The security of these building blocks can be argued from assumptions implied by the GGM. In more detail:

- for PST we require the $(\mu+1)\delta$–Strong Diffie-Hellman and the (δ, μ)–Extended Power Knowledge of Exponent assumption (see [39] and discussion in [10, E.1]).
- for MIPP we require a variant of the (q, m)-Auxiliary Structured Single Group Pairing (see [9]).

Trusted setup: Let $t_1, \ldots, t_n$ be random values in $\mathbb{Z}_q$. The CRS consists of the N values $g_1^{\chi_i(t)}$ where $\chi_i(X_1, \ldots, X_n) = [\Pi_{j:i_j=1} X_j][\Pi_{j:i_j=0}(1 - X_j)]$ for $i \in \{0,1\}^n$ the multilinear Lagrange polynomial. The CRS also includes the values $g_2^{t_j}$ for $j = 1, 2, \ldots, n$.
Commitment: $Com(p) = g_1^{p(\vec{t})} = C$ which can be evaluated given the values of p on the Boolean hypercube and the CRS.
Opening: To prove that $y = p(\vec{a})$ where $\vec{a} = (a_1, \ldots, a_n)$ the prover computes the polynomials $q_i(\vec{x})$ such that

$$p(\vec{x}) - y = \Sigma_i (x_i - a_i) q_i(\vec{x})$$

by repeated polynomial division (note that in reality $q_i(\vec{x}) = q_i(x_i, \ldots, x_n)$, i.e., only of the last $n - i$ variables). The proof is then the vector $\vec{w} = (w_1, \ldots, w_n)$ where $w_i = g_1^{q_i(\vec{t})}$ which can be computed given the polynomials q_i and the CRS.
Verification: To verify that $y = p(\vec{w})$ given $\vec{w}$, the verifier checks that

$$e(Cg^{-y}, g) = \Pi_i e(w_i, g_2^{t_i - a_i}) \ .$$

Fig. 1: The PST commitment scheme in the Lagrange basis.

P sends the polynomial $p_1(x) = \sum_{i \in \{0,1\}^{n-1}} p(x, i)$.
V checks that $p_1(0) + p_1(1) = a$ and sends back $r_1 \in_R \mathbb{F}$.
P sends the polynomial $p_2(x) = \sum_{i \in \{0,1\}^{n-2}} p(r_1, x, i)$.
V checks that $p_2(0) + p_2(1) = p_1(r_1)$ and sends back $r_2 \in_R \mathbb{F}$.
...
At round j P sends the polynomial $p_j(x) = \sum_{i \in \{0,1\}^{n-j}} p(r_1, \ldots, r_{j-1}, x, i)$.
V checks that $p_j(0) + p_j(1) = p_{j-1}(r_{j-1})$ and sends back $r_j \in_R \mathbb{F}$.
...
At the last round P sends the polynomial $p_{n-1}(x) = p(r_1, \ldots, r_{n-1}, x)$.
V checks that $p_{n-1}(0) + p_{n-1}(1) = p_{n-2}(r_{n-2})$, selects $r_n \in_R \mathbb{F}$ and checks that $p_{n-1}(r_n) = p(r_1, \ldots, r_n)$ via a single query to p.

Fig. 2: The Sumcheck Protocol

2.3 PST Polynomial Commitments

We refer the reader to Section 2.1 for the notation we use in this section. In Fig. 1 we describe the PST polynomial commitment modified to work over the Lagrange basis [27].

Note that if $n = \log N$ where N is the size of the R1CS, then the trusted setup is linear in the size of the circuit, and that verification of the opening requires $O(n)$ (i.e., logarithmic in the size of the circuit) work.

2.4 Sumcheck

Let $p(x_1 \ldots, x_n)$ be a multilinear[8] polynomial in n variables defined over a field $\mathbb{F}$. Consider the value $a = \sum_{i \in \{0,1\}^n} p(i)$, i.e., the sum of the value of p on all the vertices of the Boolean hypercube. This computation takes $N = 2^n$ time and the sumcheck protocol [24] described in Figure 2, is a way for a Prover to convince a Verifier that a is correct in $O(n)$ time, plus a *single* query to the polynomial p on a random point in $\mathbb{F}^n$.

[8] We only care about multilinear polynomials for Testudo but the sumcheck protocol can be run on any multivariate polynomial.

2.5 Spartan Overview

In this section we review Spartan [30], a transparent SNARK for R1CS. For space reasons, ours is a very high level review and the reader is referred to [30] for details.

Recall that a R1CS instance $(\mathbb{F}, A, B, C, x, N, m)$ is satisfiable if there exists a witness $w \in \mathbb{F}^{N-|x|-1}$ such that

$$(A \cdot z) \circ (B \cdot z) = (C \cdot z)$$

where $z = (x, 1, w)$, $\cdot$ is the matrix-vector product, and $\circ$ is the Hadamard (entry-wise) product.

The first step in Spartan is to encode the matrices A, B, C and the vector z via their multilinear polynomial extensions. Let $n = \log N$. For the matrix A consider the unique multilinear polynomial in $2n$ variable $\tilde{A}(t_1, \dots, t_n, u_1, \dots, u_n)$ such that $\tilde{A}(i_1, \dots, i_n, j_1, \dots, j_n) = A(i, j)$ where $(i_1, \dots, i_n)$ is the binary expansion of i and $(j_1, \dots, j_n)$ is the binary expansion of j. The polynomials $\tilde{B}, \tilde{C}$ are defined similarly, as well as the polynomial $\tilde{Z}(u_1, \dots, u_n)$ where $Z(i_1, \dots, i_n) = z(i)$.

The satisfiability condition is then equivalent to the following polynomial $F(t_1, \dots, t_n)$ being zero on all the points of the Boolean hypercube

$$F(\vec{t}) = \left(\sum_{\vec{u} \in \{0,1\}^n} \tilde{A}(\vec{t}, \vec{u}) \tilde{Z}(\vec{u}) \right) \cdot \left(\sum_{\vec{u} \in \{0,1\}^n} \tilde{B}(\vec{t}, \vec{u}) \tilde{Z}(\vec{u}) \right) - \sum_{\vec{u} \in \{0,1\}^n} \tilde{C}(\vec{t}, \vec{u}) \tilde{Z}(\vec{u})$$

Consider now the *multilinear extension* [9]of $F(\cdot)$, that is the polynomial $Q(\vec{s}) = \sum_{\vec{t} \in \{0,1\}^n} F(\vec{t}) eq(\vec{t}, \vec{s})$ where $eq(\vec{t}, \vec{s}) = \prod_{i=1}^n s_i t_i + (1 - s_i)(1 - t_i)$ is the multilinear polynomial which is equal to 1 if and only if $\vec{t} = \vec{s}$ and otherwise is equal to 0.

Since $F(\vec{t})$ is zero on the Boolean hypercube, $Q(\vec{s})$ is then identical to the zero polynomial by Schwartz-Zippel lemma. This condition can be verified by testing $Q(\vec{s})$ on a random point. Spartan is a way to check this evaluation in an efficient way. More precisely, to verify the satisfiability of the original R1CS Spartan performs the following steps:

1. Proves that $Q(\vec{r}) = 0$ for a random point $\vec{r} \in \mathbb{F}^n$. Note that due to the definition of $Q(\cdot)$ this can be done via a sumcheck protocol.
2. The above sumcheck protocol reduces to proving that $\sigma = F(\vec{\rho})$ for a random $\vec{\rho} \in \mathbb{F}^n$. Due to the definition of F this reduces to proving the value of three summations $\sum_{\vec{u} \in \{0,1\}^n} \tilde{A}(\vec{\rho}, \vec{u}) \tilde{Z}(\vec{u})$, $\sum_{\vec{u} \in \{0,1\}^n} \tilde{B}(\vec{t}, \vec{u}) \tilde{Z}(\vec{u})$, and $\sum_{\vec{u} \in \{0,1\}^n} \tilde{C}(\vec{t}, \vec{u}) \tilde{Z}(\vec{u})$. Each one of them can also be proven via a sumcheck, and in Spartan these 3 sumchecks are aggregated into a single one.
3. Finally the above sumchecks reduce to proving the values of the multilinear extensions on random points, i.e., the values of $\tilde{A}(\vec{r_x}, \vec{r_y})$, $\tilde{B}(\vec{r_x}, \vec{r_y})$, $\tilde{C}(\vec{r_x}, \vec{r_y})$, and $\tilde{Z}(\vec{r_y})$.

The final point is achieved via the use of *polynomial commitments*. The prover commits to the polynomials $\tilde{A}, \tilde{B}, \tilde{C}$ (these are called *computation commitments* since they encode the computation), and $\tilde{Z}$ (*witness commitment*, since it encodes the witness).

A major contribution of Spartan is to show how to efficiently commit to $\tilde{A}, \tilde{B}, \tilde{C}$ to leverage their sparseness (recall that in R1CS matrices have N^2 entries but only m are non-zero). This requires a non-trivial use of memory checking techniques, and introduces a substantial overhead which can be avoided in practice for uniform circuits where the Verifier can evaluate $\tilde{A}, \tilde{B}, \tilde{C}$ on their own.

Spartan's focus was to obtain a transparent SNARK, and therefore it uses a multidimensional Pedersen's commitment together with an inner product proof to implement the polynomial commitment. Because we are already using a trusted setup for the Groth16 layer, we changed the polynomial commitment to a different one which also has a trusted setup.

3 A Generalized MIPP Protocol

In order to obtain a multilinear PCS with $O(\sqrt{N})$ trusted setup, in this section we show how to adapt ideas from Section 6 of [9] which were applied to the KZG univariate PCS. We generalize it to work with multivariate polynomials and the PST commitment.

[9] Such a polynomial of degree at most 1 in each variable always exists for any function f mapping $\{0,1\} \to \mathbb{F}$ [34].

– If $M = 1$, the Prover sends A_1, y_1, and the verifier checks that $T = e(A_1, h_1)$ and $U = A_1^{y_1}$.
– Assume now that $M \geq 2$ and is a power of 2. Let $M' = M/2$.

- The prover sets the following:
  – $\vec{A_L} = [A_1 \dots A_{M'}]$ and $\vec{A_R} = [A_{M'+1} \dots A_M]$
  – $\vec{y_L} = [y_1 \dots y_{M'}]$ and $\vec{y_R} = [y_{M'+1} \dots y_M]$
  – $\vec{h_L} = [h_1 \dots h_{M'}]$ and $\vec{h_R} = [h_{M'+1} \dots h_M]$
- The Prover computes and sends to the Verifier the following values $U_L = \langle \vec{A_L}, \vec{y_R} \rangle$; $U_R = \langle \vec{A_R}, \vec{y_L} \rangle$; $T_L = CM(\vec{A_L}, \vec{h_R})$; $T_R = CM(\vec{A_R}, \vec{h_L})$
- The Verifier chooses a random field element x and sends it to the prover
- They recurse on the following values (note that the vectors have now size $M' = M/2$): $\vec{A'} = \vec{A_L} * \vec{A_R}^{x}$;$\vec{y'} = \vec{y_L} * \vec{y_R}^{x^{-1}}$; $\vec{h'} = \vec{h_L} * \vec{h_R}^{x^{-1}}$;$T' = T * T_L^{x^{-1}} * T_R^{x}$;$U' = U * U_L^{x^{-1}} * U_R^{x}$

Fig. 3: Generalized MIPP Protocol.

Changes from original MIPP. The protocol in this section has two changes when compared to the one in [9] (we are referring specifically to MIPP_k which we recall in Appendix C.4). First, we generalize MIPP to work on multivariate rather than univariate polynomials. Second, we show that the techniques also work when the polynomial is represented in the Lagrangian basis.

The Generalized MIPP protocol: Given a vector $\vec{A} = [A_1, \dots, A_M]$ of group elements in $\mathbb{G}_1$, the IPP commitment to $\vec{A}$ with a CRS $\vec{h} = [h_1, \dots, h_M]$ of group elements in $\mathbb{G}_2$ is

$$T = CM(\vec{A}, \vec{h}) = \prod_{i=1}^{M} e(A_i, h_i).$$

Let $m = \log M$. In our case, $h_i = h^{\chi_i(\vec{t})}$ for $i \in \{0,1\}^m$, where $\vec{t} = [t_1, \dots, t_m]$ is a random *secret* vector of field elements and h is a generator of $\mathbb{G}_2$.

Our generalized MIPP protocol allows a prover to prove that given a public vector of field elements $\vec{b} = [b_1, \dots, b_m]$, we have that

$$U = \langle \vec{A}, \vec{y} \rangle = \vec{A}^{\,\vec{y}} = \prod_{i=1}^{M} A_i^{y_i},$$

where the vector $\vec{y}$ is defined as $\vec{y} = [y_1, \dots, y_M]$ with $y_i = \chi_i(\vec{b})$ for $i \in \{0,1\}^m$, where $\chi_i(X)$ is the i^{th} Lagrange polynomial defined as

$$\chi_i(X_1, \dots, X_m) = \prod_{j: i_j = 1} X_j \cdot \prod_{j: i_j = 0} (1 - X_j).$$

This proof has size and verification time $O(m)$, which means that the verifier needs only to read the vector $\vec{b}$ and not construct the entire vector $\vec{y}$, which is only implicitly defined.

The protocol is described in Fig. 3.

Note that there will be m levels of recursion. Also note that the Verifier cannot compute the vectors $\vec{A'}, \vec{y'}, \vec{h'}$ since they are too big. Only the prover will compute those and provide the final value at the end of the recursion to the Verifier. We show later how the Verifier can check that they are correct. The Verifier can compute T', U'.

Properties of the construction We make the following claims about the construction above which are easily proven by induction.

Claim: $T' = CM(\vec{A'}, \vec{h'})$

Claim: $U' = \langle \vec{A'}, \vec{y'} \rangle$

Claim: The Verifier will work only in $O(m) = O(\log M)$ time

Table 1: Comparison of Prover Efficiency

Scheme	**Setup Size**	**Committing**	**Opening**
Testudo	$O(\sqrt{N}), \mathbb{G}_1$, $O(\sqrt{N})\mathbb{G}_2$	$O(\sqrt{N})\mathbb{G}_1$, $O(\sqrt{N})$ pairings, $O(\sqrt{N})\mathbb{G}_t$	$6\sqrt{N}\mathbb{G}_1$, $O(\sqrt{N})\mathbb{G}_2$, $4\sqrt{N}$ pairings, $4\sqrt{N}\mathbb{G}_t$
PST	$O(N)\mathbb{G}_1$	$O(N)\mathbb{G}_1$	$2N\mathbb{G}_1$ $O(N)$ poly division
Gemini	$O(N)\mathbb{G}_1$	$O(N)\mathbb{G}_1$	$4N\mathbb{G}_1$

Table 2: Comparison of Verifier Efficiency

Scheme	**Proof Size**	**Verification Time**
Testudo	$\approx \log(N)/2(\mathbb{G}_t + \mathbb{G}_1 + \mathbb{G}_2)$	$\approx \log(N)/2(\mathbb{G}_t + \mathbb{G}_1 + \mathbb{G}_2)$
PST	$O(\log N)\mathbb{G}_2$	$O(\log N)$
Gemini	$3n\mathbb{G}_2$	$8n$ pairings$, 3n\mathbb{G}_2, 3n\mathbb{G}_1$

How can the verifier compute the vectors $\vec{y'}, \vec{h'}$ without reading them? The trick is that they are "structured". It is easy to see by induction that at the end of the recursion the value $\hat{y}$ (the collapsed version of $\vec{y'}$ at the end of the recursion) is equal to $(1 - b_1 + x_1^{-1}b_1), \ldots, (1 - b_m + x_m^{-1}b_m)$ which the verifier can compute in $O(m)$ time on their own.

Similarly the value $\hat{h}$ (the collapsed version of $\vec{h'}$ at the end of the recursion) can be seen to be equal to $\hat{h} = h^{\prod_i (1-t_i+x_i^{-1}t_i)}$.

Note that $\hat{h}$ is a PST commitment of a multilinear polynomial in m variables. The Verifier does not compute it itself (it would be too expensive) but receives it at the end of the recursion from the Prover. To check that it is correct, the verifier computes the polynomial in a random point and it asks the prover to open this PST commitment. The verification time of this construction is $O(m)$.

4 Testudo-Comm: Our PCS with Square Root Trusted Setup

Now we show how to reduce the size of the PST trusted setup to $O(\sqrt{N})$ using the generalized MIPP in Section 3. See Fig. 4.

Theorem 1. Testudo-Comm *(Fig. 4) is secure in the GGM.*

Efficiency. Our commitment scheme improves in proving time trading against proof size and verification time. The key observation for proving efficiency is that, even though prover has to do more expensive operations (pairings, $\mathbb{G}_t$ multiplications etc), it does them on a $\sqrt{N}$ sized polynomial, which makes a large difference in practice for large N. For example, in Gemini [6] (see Appendix A), for 2^{25}, it takes *at least* 36s to create an opening proof[25] while we evaluate it takes only 11s using Testudo's commitment (see evaluation section7.1). However, on smaller N, the Gemini transformation is likely to outperform Testudo's commitment because of the time required to perform the pairings and $\mathbb{G}_t$ comutations in our case.

We summarize the efficiency properties of the prover in Table 1 assuming a circuit of size $N = 2^n$ and security parameter λ. We then compare the efficiency for the verifier in Table 2.

- **PST**: To open, the prover computes for each of the n rounds, a polynomial division of size 2^{n-1} leading to a $O(2^{n-1})$-sized polynomial division complexity. While this operates on field elements, we found out that this division, because it doesn't use FFTs, is actually a bottleneck on large sizes (such as 2^{25}).

Trusted Setup We perform the PST trusted setup for $m = n/2$ variables. Let $t_1, \ldots, t_m$ be random values in $\mathbb{Z}_q$. The CRS consists of the $M = \sqrt{N}$ values $g_1^{\chi_i(\vec{t})}$.

There is also the trusted setup to generate the vector $\vec{h}$ for the MIPP proof also of size $M = \sqrt{N}$.

Commitment Let $p(x_1, \ldots, x_n)$ be a multilinear polynomial with n variables. We split the variables in two sets X and Y each of size $m = n/2$, and denote $p(X, Y)$ accordingly. Let $p_i(X) = \left(\sum_{j\in\{0,1\}^m} p(j, i) \cdot \chi_j(X)\right)$, so that

$$p(X,Y) = \sum_{i\in\{0,1\}^m} \left(\sum_{j\in\{0,1\}^m} p(j,i) \cdot \chi_j(X) \right) \chi_i(Y)$$

The prover "in its own head" PST-commits to each $p_i(X)$ (it can do this because each p_i is multilinear and in $m = n/2$ variables). This yields a vector of M group elements in G_1, say $A = (A_1, ..., A_M)$, one commitment A_i for each p_i.

The actual commitment the prover sends to the verifier is a MIPP commitment T to the vector A, defined as above i.e. $T = CM(\vec{A}, \vec{h}) = \Pi_i e(A_i, h_i)$

Opening Now suppose the verifier asks for $p(\vec{a}, \vec{b})$ where $\vec{a}, \vec{b} \in \mathbb{F}^m$. This can be written as:

$$p(\vec{a}, \vec{b}) = \sum_{i\in\{0,1\}^m} p_i(\vec{a}) * \chi_i(\vec{b})$$

Let us define the m-variate multilinear polynomial

$$q(X) := \sum_{i\in\{0,1\}^m} p_i(X)\chi_i(\vec{b})$$

Note that q is multilinear, and $p(\vec{a}, \vec{b}) = q(\vec{a})$.

Let $\vec{y}$ denote the length $\sqrt{N}$ vector consisting of $y_i = \chi_i(\vec{a}) \in \mathbb{F}_r$ as i ranges over $\{0,1\}^m$.

To prove that $v = p(\vec{a}, \vec{b})$ the prover proceeds in three steps as in Section 6 of the IPP paper [9]:

1. P sends U a PST-commitment to q using the same SRS as that used to commit to each p_i. Note that U is the inner product of $\vec{A}$ and $\vec{y}$ i.e., $U = \Pi_i A_i^{y_i}$.
2. Second, P proves using the generalized MIPP protocol that U is indeed the inner product of the vector $\vec{A}$ and the vector $\vec{y}$, where $\vec{A}$ is the opening vector to T.
3. Third, P uses the PST-evaluation protocol to prove that given the commitment U the opening polynomial q evaluated at $\vec{a}$ yields v (i.e. $q(\vec{a}) = v$).

Verification The verifier receives v claimed to be $v = p(\vec{a}, \vec{b})$, the value U, the generalized MIPP proof π_1 and the PST proof π_2 and it performs the MIPP and PST verifications. Note that all verification steps are $O(m) = O(\log N)$.

Fig. 4: Testudo-Comm

– **Gemini**: To open, the prover computes for each of the n rounds, 1 KZG openings of size 2^{n-i} and 2 of size 2^{n-i-1}, leading to a complexity of $O(2N + N + N) = O(4N)\mathbb{G}_1$ scalar multiplications. For verification, it therefore requires perfoming $O(8n)$ pairings (or $4n$ pairing checks).
– **Testudo**: To open, the prover must compute:
  - A PST commitment to the $q(X)$ polynomial, so $O(\sqrt{N})\mathbb{G}_1$
  - A MIPP opening proof, consisting of n rounds where prover computes (a) $O(2\sqrt{N}/2^i)\mathbb{G}_1$ scalar multiplication to compute the reduced vectors, and (b) $O(2\sqrt{N}/2^i)\mathbb{G}_t$ and pairings operations to compute the commitment to each reduced vectors. This leads to $O(4\sqrt{N})\mathbb{G}_1$ and $O(4\sqrt{N})\mathbb{G}_t$ + pairings.
  - Two PST opening proofs, each of size $O(\sqrt{N})$, one on $\mathbb{G}_1$ and one on $\mathbb{G}_2$

Remark 1 (Distributed trusted setup). Our construction requires a trusted setup for the polynomial commitment of a specific form. It needs to encode in particular a secret tuple of points and their (multivariate) monomial evaluation. We can obtain an MPC for such a setup by straightforwardly adapting the techniques from [7]. We will detail these techniques in the full version of the paper.

Remark 2 (Proof Size). The proof size for our commitment scheme are 8x bigger at 2^{25} than PST. To reduce the size, we can compress the G_t elements on the torus as in [26]. This could potentially reduce by half the proof size, bringing it to the same order of magnitude as a PST opening proof. Note however, proof size don't matter much in the Testudo SNARK as they are verified by another Groth16 proof on top.

5 Testudo: Our Construction

At this point we recap the general structure of Testudo. Let A, B, C be the input R1CS of size N.

Trusted Setup. We assume that a trusted party (or a distributed multiparty computation protocol, aka ceremony) generates the trusted setup for Testudo-Comm (which is of size $\sqrt{N}$) and the Groth16 trusted setup for an R1CS correspoding to the verification algorithm of the Spartan sumchecks and the Testudo-Comm opening proofs (this R1CS has size $O(\log N)$. This trusted setup is independent of A, B, C and therefore universal.

Computation Commitments. As in Spartan, in a preprocessing stage, the prover encodes A, B, C as sparse polynomials $\tilde{A}, \tilde{B}, \tilde{C}$ and commits to them via polynomial commitments (*computational commitments*). We note that for uniform circuits (e.g., data-parallel, with many sub-circuits repeating in regular patterns), this step is not necessary or much reduced in complexity, since the verifier can efficiently compute $\tilde{A}, \tilde{B}, \tilde{C}$ on their own or is only required to compute the computational commitment for the subcircuit.

Witness Commitments. In the online phase, the prover computes $\tilde{w}$, a multilinear extension of the witness w and commits to it using Testudo-Comm. Note that the polynomials are of size $O(N)$ here, corresponding to the number of R1CS constraints.

Prover. The Testudo prover:

– Executes the Spartan prover to prove the satisfiability of A, B, C (see Section 2.5), with the only difference that it uses Testudo-Comm as the underlying polynomial commitment.
– Produces the appropriate openings of the Testudo-Comm PCS.
– Produces and outputs a Groth16 proof that it knows the above modified Spartan proof.

Verifier. The verifier checks the output Groth16 proof and accept/rejects accordingly.

Theorem 2. *Assuming that* Testudo-Comm *is an extractable PCS, and Groth16 is a SNARK, then* Testudo *is a SNARK.*

Informally the proof follows from the fact that if Groth16 is a SNARK we can extract a "modified Spartan" proof – modified to use Testudo-Comm ass the underlying PCS. But if Testudo-Comm is extractable, then we know that we can extract the witness (Spartan is a SNARK as long as the underlying PCS is extractable).

As with all recursive SNARKs we have to heuristically assume that we can instantiate the random oracle in Spartan to a very specific hash function (in our case Poseidon) and not lose security. This is because the code of the hash function has to be embedded in the outer Groth16 proof.

6 Practical considerations

Choice of curves. The original version of Spartan (the starting point for the Testudo) uses a custom version of `curve25519-dalek`, which provides an efficient implementation of a prime-order Ristretto group [28], an abstraction that facilitates implementations of prime-order groups with strong security guarantees. However, as this elliptic curve does not support pairings, composing the original Spartan with Groth16 is not possible and, thus, we had to find a pairing-friendly alternative. We opted for BLS12-377 combined with BW6-761 because they represent the most efficient pair that further supports *2-chaining of pairing-equipped elliptic curves* [13, 14, 1] which is required for our design. See Appendix D for more details.

Testudo for data-parallel computation. We can make Testudo particularly efficient for data-parallel computation. Consider a relation R^* composed of several repetitions of the same relation $R(\vec{x}^{(1)}, \vec{w}^{(1)}) \wedge \cdots \wedge R(\vec{x}^{(K)}, \vec{w}^{(K)})$ $\left(\vec{x}^{(1)}, \ldots, \vec{x}^{(K)}\right)$ We are able to amortize the proving costs related to the wiring of the circuit whenever the circuit is of this form.

In the Spartan lingo the building block for proving the wiring of the circuit refers to a *computation commitment.* A computation commitment is a polynomial commitment opening to polynomials encoding the structure of the circuit. If we apply Testudo naively to such a relation we would need to open a computation commitment of size roughly $K|w|$.Instead, we modify our building blocks appropriately to leverage the structure of the circuit and we require computation commitments whose opening grows only linearly in the size of the small subrelation R. We expand on this construction in Appendix B.

6.1 Parallelization and aggregation of Testudo proofs

We observe that this framework enables aggregation of proofs at different levels, each with their pros and cons, but all being compatible with each other, resulting in a system that can scale to large instances in practice because it enables *parallelization* of the proof generation.

Aggregation at Spartan level: Assume a prover is running different sumchecks + PCS openings in parallel using different witnesses, *on different machines* (otherwise, the prover should use the data parallel version that requires the whole witness to be present). In this setting, aggregating the verification of the sumcheck can be done either via (a) aggregating the different Groth16 sumcheck-verifier proofs together using Snarkpack like constructions, or (b) having one Groth16 proof that verifies multiple instances of the sumcheck. Aggregation for the polynomial commitment scheme could be done by the prover (a) at the beginning, by committing to a random linear combination of the different multilinear extensions of the witnesses and (b) by opening at a random point this combined polynomial. This would require communication between the machines to aggregate the polynomials together; given it's a single round of communication, we believe it can still be a useful for a practical deployement inside a cluster of machines.

Aggregation at the Groth16 level: Instead of verifying a single sumcheck instance Groth16 proof and a single PCS opening proof, the outer proof can verify *multiple of those.* We need further work to estimate the complexity of the final outer circuit but our current estimation (4M constraints for the outer circuit) seem to indicate that it is possible to verify in the order of 5-10 proofs together in a reasonable timeframe, depending on the application.
Note that this aggregation *does not require knowledge of the witness* and therefore can be done by more expensive prover machines.

Aggregation on top: Because Testudo's final proof is a Groth16 proof, one can simply use snarkpack to efficiently pack thousands of such proofs together. Similar to previous category, this aggregation level *does not require knowledge of the witness* and therefore can be done by more expensive prover machines.

7 Implementation and Evaluation

7.1 Implementation

We have a working Testudo implementation[10] that features the sumcheck verifier proof and our Testudo commitment scheme. We based our work on the Spartan [30] codebase, which has been adapted to use the

[10] The current version of the repository is available at https://github.com/cryptonetlab/testudo .

Arkworks [2] framework to enable support for any pairing based curves. We also have started an effort on parallelizing the Spartan codebase, although there are still many low hanging fruits to optimize for. On top of this, we implemented a wrapper around a BLS12-381 library that supports GPU operations and released it open source on Github at `https://github.com/nikkolasg/ark-blst`.

7.2 Testudo Commitment

We first evaluate our new multivariate commitment scheme compared to the standard PST algorithm. This has been ran over a c5a.12xlarge AWS instance, i.e., 48 cores with 96 threads. We note that the structure of the commitment allows for heavy use of parallelism, which we exploited in our implementation.

Figure 5 shows that the Testudo commitment maintains the performance of the PST commitment and, for large circuit instances, **it is 2 orders of magnitude faster** for opening by significantly reducing the size of the MSM required. Indeed, it operates on $\sqrt{N}$ size MSM. However, verification is slower, due to the logarithmic number of pairings required to verify the inner pairing product argument proof. There are still many low-hanging fruits in the codebase to speed up verification such as batching the pairing operations in MIPP and PST. Moreover, we continue to avoid the use of FFT due to the usage of multilinear polynomials
Verification speed: As mentionned above, there are many low hanging fruits for optimizing the codebase. For example, to speed up verification, one can bundle the MIPP and the PST part together (e.g. run the pairings check all at once). Currently both MIPP and PST codebase are quite separate.

Proof Size: The Testudo Commitment brings an increase to the proof size in comparison to the simple PST by a factor of 3 but we ensure this is not an issue for the communication cost through recursion. Using the BLS12-377, we are able to efficiently verify commitment openings inside a Groth16 circuit as outlined in section 6.1.

7.3 Testudo Groth16 Constraints

In this section, we estimate the number of R1CS constraints necessary to verify the core of Testudo (the sumchecks, the PCS opening and the computation commitment). Figure 6 shows the number of constraints for each parts according to the user circuit size (e.g. the circuit that the user writes on Testudo). In this estimation, we:

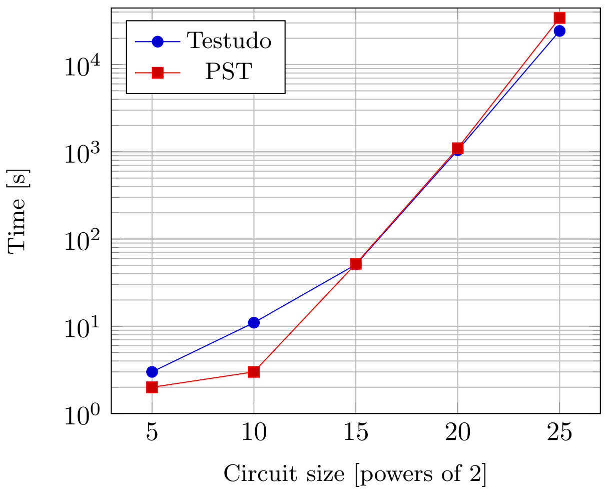
- Use Testudo Commitment as the multilinear PCS scheme for the computation commitment part of Spartan. In the original design, it uses Hyrax.
- Thanks to the previous point, we can now do a random linear combination of all the polynomials the prover must perform and having the prover only compute a single Testudo Commitment opening proof
- We verify the core Spartan sumchecks (steps 2 and 3 in Section 2.5) and the grand product sumchecks ([29, bottom of pg. 27]) from the computation commitment inside the same Groth16 verifier that checks the sumcheck in the satisfiability proof. This is possible since both are operating on the same fields.
- Note that we need to verify a PST opening both on $\mathbb{G}_1$ and $\mathbb{G}_2$ during the Testudo commitment (for the PST part and for the MIPP part respectively).

The biggest contributor of the R1CS constraints number is the MIPP part, because it requires to compute $log(N)$ $\mathbb{G}_t$ operations (exponentiation is almost 40k in the library we used). We expect this number to drastically go down by roughly 30–50% with optimizations on $\mathbb{G}_t$ computations, such as using the torus arithmetic version and endomorphisms optimizations [26].

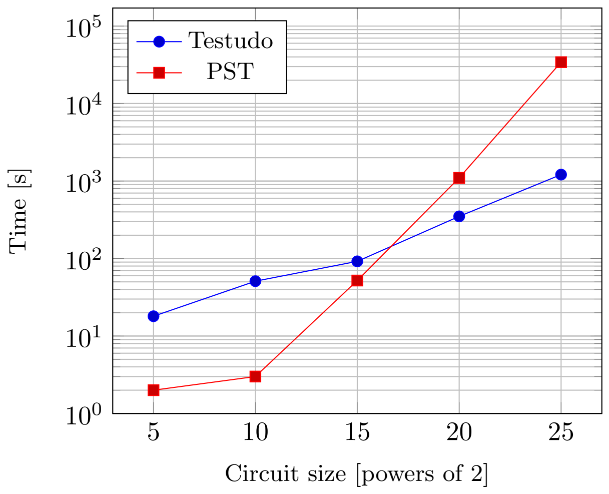
7.4 Testudo on data-parallel circuits

We have the necessary building blocks to estimate accurately the proving time of a uniform circuits (even though the implementation does not yet offer that feature). Specifically, to estimate the time of proving for uniform circuits, we need to add the time for

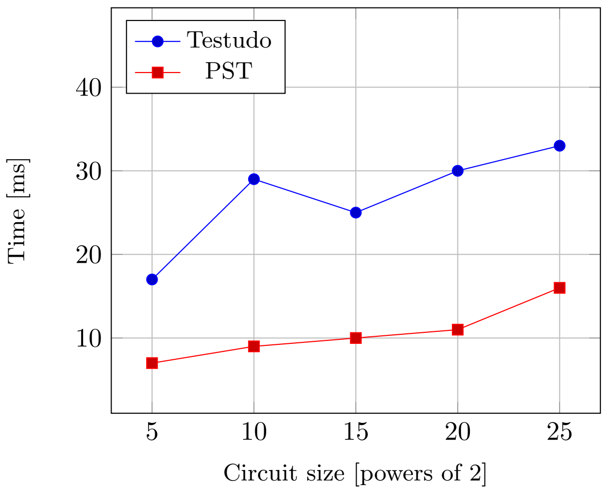
- The first sumcheck on the full R1CS matrix (SC1)
- The second sumcheck on the small subcircuit (SC2)
- The Testudo commitment (TC) times on the full witness size - commitment and opening combined



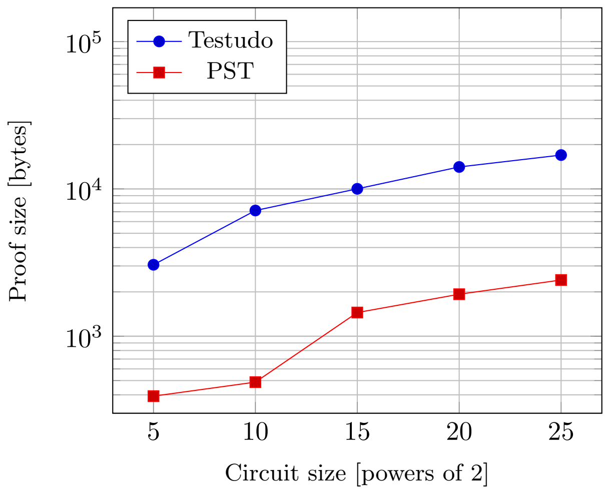
(a) Commitment computation time



(b) Commitment opening time



(c) Opening proof verification time



(d) Opening proof sizes

Fig. 5: Comparison between the Testudo and PST commitment schemes. While the time to compute a commitment is similar for Testudo and PST, see (a), we can see in (b) that Testudo outperforms PST when opening a commitment for large circuit sizes by almost two orders of magnitude. On the other hand, graph (c) shows that PST is faster than Testudo by a factor of 2x which, however, can likely be addressed as there are various straightforward ways to further optimize the Testudo code. Finally, graph (d) shows that Testudo opening proofs are about one order of magnitude larger than PST proofs, which, however, will not have any practical impact on the overall sizes of Testudo SNARK proofs, as the opening proof will be ultimately verified by a constant-size Groth16 proof.

Fig. 6: Constraints for Groth16 verifier of Testudo Proof

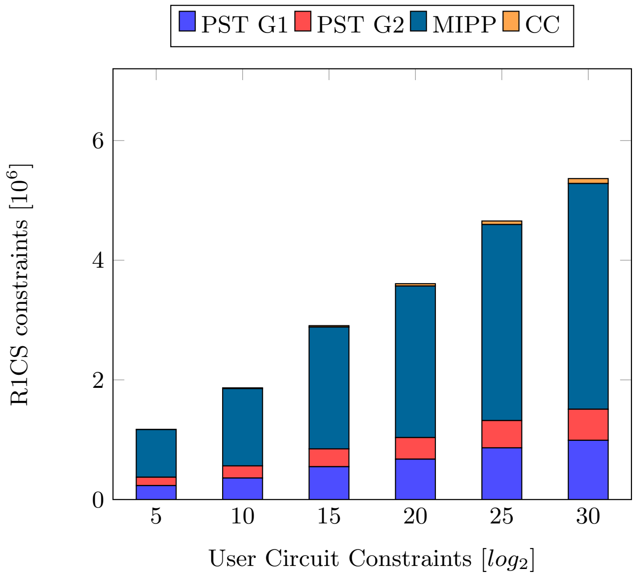


Fig. 7: Individual cost of data parallel version of Testudo

Constraints	2^{15}	2^{20}	2^{25}
Commit (s)	0.072	0.723	11
Sumcheck 1 (s)	0.04	0.903	28
Sumcheck 2 (s)	0.036	0.941	29
TC Opening (s)	0.134	0.471	1.5
CC Opening (s)	2	32	939

– The computation commitment time on the small subcircuit (CC)

We have benchmarked these data for two different subcircuit of size (a) 2^{15} and (b) 2^{20}. The subcircuit corresponds roughly to the size of a Merkle Tree proof verification circuit for 32 layers using either (a) Poseidon for 2^{15} or (b) SHA256 for 2^{20} as the hash function. We're estimating the proving time to repeat these circuits to achieve overall a circuit of 2^{25} constraints: subcircuit (a) is repeated 1024 times (i.e. verify 1024 Merkle Tree proofs), and subcircuit (b) is repeated 32 times. For these parameters, we show that our data parallel Testudo version can be **9.7x faster** for (a) and **4.3x faster** for (b) than their Groth16 equivalent. The improvement is expected to be even larger as the gap of constraints between the subcircuit and the bigger circuit grows. As further work, we will compare these improvement to SNARKs based on plonkish arithmetization, although we are not aware of such speedups for other proof systems at this time.

7.5 Future Work

Here we list additional points of improvement to make Testudo even more practical. Some of these points are implementation-related, some are more cryptographically-flavored design choices.

– **Circuit building**: The original codebase did not feature an API to build circuits, and was constructing the R1CS manually for testing. By having an API for transforming circuits in the R1CS representation, we could make "product oriented" comparison with other proof system in a meaningful way. For example, answering "what is the proving time to verify a Merkle Tree opening proof using SHA256 on 32 layers".
– **Data Parallel Circuits**: The performance of Testudo on data parallel circuits can be easily estimated with our codebase, however we plan to make it an off-the-shelf feature which mainly requires modifying sumcheck according to the construction described in section 6.2. This would further provide directions for future improvements of the codebase.

Fig. 8: Cost of the data parallel version of Testudo vs Groth16

Subcircuit Size	**Proving Time**	**Speed up vs Groth16**
2^{15} constraints (1024 repetitions)	33s	9.7x
2^{20} constraints (32 repetitions)	73s	4.3x
Groth16 Proving (2^{25})	322s	—

– **Computation Commitment**: The computation commitment is the biggest bottleneck in the codebase currently, by far. While such cost is expected from this type of SNARK constructions, we note that the Spartan Computational Commitment and implementation is constructed with a transparent setup in mind. We plan to revisit this part using better polynomial commitment scheme e.g., the curve we run our SNARK on supports pairings and a trusted setup is acceptable in our setting. Another way to alleviate the costs is changing the polynomial batching strategy for commitment and proving evaluation in the CC.
– **CCS language and recursion**: CCS [31] is a generalization of R1CS, Plonkish and AIR languages. This work further leads to Hypernova [22], a folding scheme which could efficiently be used for recursion in such based languages. Implementing the CCS generalization and the folding scheme could show Testudo to be a viable framework for various optimization down the line. For example, integrating lookup table is also described in this paper, and they could provide great improvement speed for some computation.

Acknowledgements.

References

1. D. F. Aranha, Y. El Housni, and A. Guillevic. A survey of elliptic curves for proof systems. Cryptology ePrint Archive, Report 2022/586, 2022. https://eprint.iacr.org/2022/586.
2. arkworks contributors. arkworks zksnark ecosystem, 2023.
3. A. Belling, A. Soleimanian, and O. Bégassat. Recursion over public-coin interactive proof systems; faster hash verification. Cryptology ePrint Archive, Report 2022/1072, 2022. https://eprint.iacr.org/2022/1072.
4. bellperson contributors. The bellperson zk-SNARK library, 2023.
5. 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.
6. J. Bootle, A. Chiesa, Y. Hu, and M. Orrù. Gemini: Elastic SNARKs for diverse environments. In *EUROCRYPT 2022, Part II*, LNCS. Springer, Heidelberg, May / June 2022.
7. 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.
8. 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*. IEEE Computer Society Press, May 2018.
9. B. Bünz, M. Maller, P. Mishra, N. Tyagi, and P. Vesely. Proofs for inner pairing products and applications. In *ASIACRYPT 2021, Part III*, LNCS. Springer, Heidelberg, Dec. 2021.
10. M. Campanelli, D. Fiore, and A. Querol. LegoSNARK: Modular design and composition of succinct zero-knowledge proofs. In *ACM CCS 2019*. ACM Press, Nov. 2019.
11. B. Chen, B. Bünz, D. Boneh, and Z. Zhang. HyperPlonk: Plonk with linear-time prover and high-degree custom gates. Cryptology ePrint Archive, Report 2022/1355, 2022. https://eprint.iacr.org/2022/1355.
12. A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. P. Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In *EUROCRYPT 2020, Part I*, LNCS. Springer, Heidelberg, May 2020.
13. Y. El Housni and A. Guillevic. Optimized and secure pairing-friendly elliptic curves suitable for one layer proof composition. In *CANS 20*, LNCS. Springer, Heidelberg, Dec. 2020.
14. Y. El Housni and A. Guillevic. Families of SNARK-friendly 2-chains of elliptic curves. In *EUROCRYPT 2022, Part II*, LNCS. Springer, Heidelberg, May / June 2022.
15. 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.
16. N. Gailly, M. Maller, and A. Nitulescu. SnarkPack: Practical SNARK aggregation. In *FC 2022*, LNCS. Springer, Heidelberg, May 2022.
17. R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *EUROCRYPT 2013*, LNCS. Springer, Heidelberg, May 2013.

18. S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating computation: interactive proofs for muggles. In *40th ACM STOC*. ACM Press, May 2008.
19. J. Groth. On the size of pairing-based non-interactive arguments. In *EUROCRYPT 2016, Part II*, LNCS. Springer, Heidelberg, May 2016.
20. J. Groth, M. Kohlweiss, M. Maller, S. Meiklejohn, and I. Miers. Updatable and universal common reference strings with applications to zk-SNARKs. In *CRYPTO 2018, Part III*, LNCS. Springer, Heidelberg, Aug. 2018.
21. A. Kate, G. M. Zaverucha, and I. Goldberg. Constant-size commitments to polynomials and their applications. In *ASIACRYPT 2010*, LNCS. Springer, Heidelberg, Dec. 2010.
22. A. Kothapalli and S. Setty. HyperNova: Recursive arguments for customizable constraint systems. Cryptology ePrint Archive, Paper 2023/573, 2023.
23. P. Labs. Filecoin: A Decentralized Storage Network, 2023.
24. C. Lund, L. Fortnow, H. J. Karloff, and N. Nisan. Algebraic methods for interactive proof systems. In *31st FOCS*. IEEE Computer Society Press, Oct. 1990.
25. G. K. Michele Orrù. zka.lc, 2023.
26. M. Naehrig, P. S. L. M. Barreto, and P. Schwabe. On compressible pairings and their computation. In *AFRICACRYPT 08*, LNCS. Springer, Heidelberg, June 2008.
27. C. Papamanthou, E. Shi, and R. Tamassia. Signatures of correct computation. In *TCC 2013*, LNCS. Springer, Heidelberg, Mar. 2013.
28. ristretto contributors. The Ristretto Group, 2023.
29. S. Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. Cryptology ePrint Archive, Report 2019/550, 2019. `https://eprint.iacr.org/2019/550`.
30. S. Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In *CRYPTO 2020, Part III*, LNCS. Springer, Heidelberg, Aug. 2020.
31. S. Setty, J. Thaler, and R. Wahby. Customizable constraint systems for succinct arguments. Cryptology ePrint Archive, Paper 2023/552, 2023.
32. V. Shoup. Lower bounds for discrete logarithms and related problems. In *EUROCRYPT'97*, LNCS. Springer, Heidelberg, May 1997.
33. J. Thaler. Time-optimal interactive proofs for circuit evaluation. In *CRYPTO 2013, Part II*, LNCS. Springer, Heidelberg, Aug. 2013.
34. J. Thaler. Proofs, Arguments, and Zero-Knowledge, 2015–2023.
35. I. Tzialla, A. Kothapalli, B. Parno, and S. Setty. Transparency dictionaries with succinct proofs of correct operation. Cryptology ePrint Archive, Report 2021/1263, 2021. `https://eprint.iacr.org/2021/1263`.
36. 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*. IEEE Computer Society Press, May 2018.
37. T. Xie, J. Zhang, Z. Cheng, F. Zhang, Y. Zhang, Y. Jia, D. Boneh, and D. Song. zkBridge: Trustless cross-chain bridges made practical. In *ACM CCS 2022*. ACM Press, Nov. 2022.
38. 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*. IEEE Computer Society Press, May 2020.
39. Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. A zero-knowledge version of vSQL. Cryptology ePrint Archive, Report 2017/1146, 2017. `https://eprint.iacr.org/2017/1146`.
40. zk Harness contributors. zk-Harness, 2023.

Supplementary Material

A The Gemini Transformation

Here we review the Gemini transformation [6] that allows to commit to a multilinear polynomial using a commitment to a univariate polynomial in a black-box fashion.

Let $F(x_1, \ldots, x_n)$ be a multilinear polynomial with $F(\vec{x}) = \sum_{i=0}^{2^n-1} \phi_i \; \vec{x}^i$ where $\vec{x}^i = x_1^{i_1} x_2^{i_2} \ldots x_n^{i_n}$ and $[i_1, \ldots, i_n]$ is the binary representation of i.

We can associate to F the univariate polynomial $f(z) = \sum_{i=0}^{2^n-1} \phi_i \; z^i$ which has the same coefficients as F

Assume we want to prove that $F(\alpha_1, \ldots, \alpha_n) = y$. The first step is to show how to express y as $O(n)$ points on univariate polynomials. Define polynomials $f_0, \ldots, f_n$ recursively as follows

- $f_0 = f$
- $f_i(z) = g_{i-1}(z) + \alpha_i h_{i-1}(z)$ where g_{i-1}, h_{i-1} are polynomial defined as

$$f_{i-1}(z) = g_{i-1}(z^2) + z \; h_{i-1}(z^2)$$

  that is $g_{i-1}(z)$ [resp. $h_{i-1}(z)$] is the polynomial with the even [resp. odd] degree terms in f_{i-1} – the same decomposition used in the recursive FFT algorithm
- Note that f_i has degree half that of f_{i-1} and therefore f_n is a constant

Claim: $F(\alpha_1, \ldots, \alpha_n) = y = f_n$.
Easily seen by induction.

To commit to F we use any univariate polynomial commitment to commit to $f = f_0$

To open $F(\alpha_1, \ldots, \alpha_n) = y$:

- the prover sends commitments to $f_1, \ldots, f_n$ (the polynomials f_i are a function of $\alpha_1, \ldots, \alpha_n$)
- The verifier asks for a random β
- The prover opens $a_i = f_i(\beta), b_i = f_i(-\beta), c_i = f_{i+1}(\beta^2)$ for the univariate poly commitments.

Note that

$$c_i = f_i(\beta^2) = g_{i-1}(\beta^2) + \alpha_i h_{i-1}(\beta^2)$$

by definition of f_{i+1} and

$$a_i = f_i(\beta) = g_i(\beta^2) + \beta h_i(\beta^2)$$

$$b_i = f_i(-\beta) = g_i(\beta^2) - \beta h_i(\beta^2)$$

by definition of g_i, h_i.

Therefore

$$c_i = \frac{a_i + b_i}{2} + \alpha_i \frac{a_i - b_i}{2\beta}$$

The verifier checks all the above equations for $i = 0, \ldots, n$ and that $f_n = y$

To make this non-interactive compute β via Fiat-Shamir on the commitments of the f_i.

B More on Testudo for Data-Parallel Computations

Additional formal details See figure Fig. 9.

Below are adaptations of some of the equations in Spartan for the data-parallel setting. For further reference on these equations, see Spartan paper, page 20, and our figure for reference. Below s, u, t are formal (tuples of) variables.

$$
\begin{aligned}
\tilde{F}(s,u) &:= \overbrace{\left(\sum_v \tilde{A}(u,v)\cdot\tilde{Z}(s,v)\right)}\cdot\overbrace{\left(\sum_v \tilde{B}(u,v)\cdot\tilde{Z}(s,v)\right)} - \overbrace{\sum_v \tilde{C}(u,v)\cdot\tilde{Z}(s,v)} \\
&= \bar{A}(s,u)\cdot\bar{B}(s,u) - \bar{C}(s,u)
\end{aligned}
$$

$$
Q^*(t) := \sum_{s,u} \chi_t(s||u)\cdot\tilde{\mathcal{F}}(s,u)
$$

At the end of the first sumcheck we “fix” (s,u) on a random challenge point $r_x = (\nu,\rho)$. Second sumcheck is on the following claim:

$$
y^* = \sum_v \tilde{Z}_{\text{tot}}(\rho,v)\cdot\Big(r_A\cdot\tilde{A}(\nu,v) + r_B\cdot B(\nu,v) + r_C\cdot C(\nu,v)\Big) \qquad (1)
$$

At the end of the second sumcheck we “fix” v on a random challenge point r_y.

Notice that the protocol in Fig. 9 does not include checks for public input; these checks are straightforward and are being ignored here for simplicity only.

Comparison with Phalanx Our approach for data-parallel Spartan has in common with the approached used in Phalanx [35]. Both works, ours and Phalanx, observe that the starting equation of the Spartan IOP can roughly retain the same shape (see (1)) and thus can be proved as in the original Spartan, simply incurring checks for additional variables (logarithmic in the number of circuit repetitions). The second observations both works use is that proper terms rearrangement allows us to amortize the cost of many computational commitments into a constant number (three).

Our approach was developed independently of that in Phalanx and took explicit inspiration from the observations in [33] and [36] which even predate Spartan. The Phalanx and the Testudo approach, moreover, differ in that our current protocol for data-parallelism deals with a “simpler” setting: *(i)* we do not require any input/output consistency; *(ii)* Phalanx additionally requires to include some “folding”-related polynomials which are helpful for their “running instance” setting.

C Additional Preliminaries

C.1 Rank-1 Constraint Systems (R1CS)

As mentioned earlier, *Rank-1 Constrained Systems (R1CS)* is a popular way to encode computations to be proven via a SNARK. R1CS were implicitly defined as *Quadratic Arithmetic Programs* in [17] where it is proven that they are NP-complete (and therefore can express any arbitrary non-deterministic polynomial computation).

An R1CS instance is a tuple $(\mathbb{F}, A, B, C, x, N, m)$ where x denotes the public input of the instance, $A, B, C \in \mathbb{F}^{N\times N}$ are matrices defined over a field $\mathbb{F}$, with $N \geq |x| + 1$, and there are at most m non-zero entries in each matrix.

An R1CS instance $(\mathbb{F}, A, B, C, x, N, m)$ is satisfiable if there exists a witness $w \in \mathbb{F}^{N-|x|-1}$ such that

$$
(A\cdot z)\circ(B\cdot z) = (C\cdot z)
$$

where $z = (x, 1, w)$, $\cdot$ is the matrix-vector product, and $\circ$ is the Hadamard (entry-wise) product.

C.2 Polynomial Commitments for Multilinear Polynomials

A polynomial commitment scheme [27] *for multilinear polynomials* is a tuple of four algorithms $\mathsf{PC} = (\mathsf{Setup}, \mathsf{Commit}, \mathsf{Eval}, \mathsf{VerEval})$:

- $pp \leftarrow \mathsf{Setup}(1,\lambda,n)$: takes as input the number of variables in a multilinear polynomial n and produces public parameters pp.

$\underline{\mathsf{Setup}\left(1^{\lambda}, 1^{\ell}, 1^{N_{\text{sub}}}, 1^{K}\right)}$

$\underline{P\left(\left(\vec{x}^{(1)}, \ldots, \vec{x}^{(K)}\right), \left(\vec{w}^{(1)}, \ldots, \vec{w}^{(K)}\right)\right)}$

$\underline{V\left(\left(\vec{x}^{(1)}, \ldots, \vec{x}^{(K)}\right)\right)}$

Let $Z_{\text{tot}} = (\vec{x}^{(1)}, \vec{w}^{(1)}, \ldots, \vec{x}^{(K)}, \vec{w}^{(K)})$

$C_{\text{tot,z}} \leftarrow \mathsf{MippPST.Commit}(\tilde{Z}_{\text{tot}})$

$\xrightarrow{C_{\text{tot,Z}}}$

$\tau \leftarrow_{\$} \mathbb{F}^{\log K + \log N_{\text{sub}}}$

$\xleftarrow{\tau}$

// Claim $Q^*(\tau) = \sum_{s,u} \chi_\tau(s||u) \cdot \tilde{\mathcal{F}}(s,u) = 0$

Invoke SC for $\sum_{s,u} \chi_\tau(s||u) \cdot \tilde{\mathcal{F}}(s,u) = 0$

(Sample SC challenge $r_x = (\nu, \rho) \in \mathbb{F}^{\log K + \log N_{\text{sub}}}$)

[Invoke step 6-11 in Spartan, pg 20,

including SC on Eq. (1) and sampling of r_y]

$v \leftarrow \tilde{Z}_{\text{tot}}(\nu||r_y)$

Same final steps as in Spartan, but:

- use $\mathsf{MippPST}$opening for$\tilde{Z}_{\text{tot}}$(on $(\nu||r_y)$)
- each computation commitment $\tilde{M}$ is evaluated on $(\rho||r_y)$

Fig. 9: Interactive version of our protocol for batch (data-parallel) relations. Given sub-relation R, above we prove batch relation $R(\vec{x}^{(1)}, \vec{w}^{(1)}) \wedge \cdots \wedge R(\vec{x}^{(K)}, \vec{w}^{(K)})$ $\left(\vec{x}^{(1)}, \ldots, \vec{x}^{(K)}\right)$ are the public inputs, each of size ℓ. $\left(\vec{w}^{(1)}, \ldots, \vec{w}^{(K)}\right)$ are the witnesses, each of size N_{sub}. We define the total witness size N_{tot} as $N_{\text{tot}} := N_{\text{sub}} \cdot K$. Notation $\tilde{v}(\vec{X})$ is the multi-linear extension of vector $\vec{v}$, i.e., $\tilde{v}(\vec{X}) := \sum_i v_i \cdot \chi_{(\vec{X})}(\vec{X})(i)$. "SC" stands for sumcheck. We use commas and concatenation interchangeably.

- $(C; D) \leftarrow \mathsf{Commit}(pp; P)$: takes as input a n-variate multilinear polynomial P over a finite field $\mathbb{F}$ and produces a public commitment C and a secret decommitment information D.
- $y, \pi \leftarrow \mathsf{Eval}(pp, \vec{x}, C, D)$: takes as input the commitment/decommitment information for a polynomial P and an input point $\vec{x} \in \mathbb{F}^n$ and returns $y = P(\vec{x})$ and a proof π that y is correct.
- $b \leftarrow \mathsf{VerEval}(pp, C, \vec{x}, y, \pi)$ takes as input a commitment C, an input value $\vec{x} \in \mathbb{F}^n$ an output value $y \in \mathbb{F}$ and a proof π and returns a bit $b \in \{0, 1\}$.

Correctness. There is an obvious *correctness* condition which says that if C, D are generated correctly using Commit on input P, and y, π are generated correctly on input $C, D, \vec{x}$ using Eval, then $\mathsf{VerEval}$ accepts on input $C, \vec{x}, y, \pi$.

Binding. A polynomial commitment is *binding* if for every efficient adversary $\mathcal{A}$ we have that

$$\Pr\begin{bmatrix} pp \leftarrow \mathsf{Setup}(1, \lambda, n) & \\ (C, \vec{x}, y, \pi, y', \pi') \leftarrow \mathcal{A}(pp) & \\ \mathsf{VerEval}(pp, C, \vec{x}, y, \pi) = 1 & : \ y \neq y' \\ \mathsf{VerEval}(pp, C, \vec{x}, y', \pi') = 1 & \end{bmatrix} = \epsilon(\lambda) \tag{2}$$

where $\epsilon(\cdot)$ is a negligible function.

Extractability. We describe this property only informally. The complete formal definition is quite involved; we refer the reader, e.g., to [12, Definition 6.2] for details. This property states that from any adversary that can satisfactorily prove evaluations $(\vec{x}, y)$ over polynomial commitments C we should be able to extract a polynomial p consistent with the proofs.

C.3 Succinct Non-Interactive Arguments of Knowledge

A SNARK with universal trusted setup is a tuple of algorithms $(\mathsf{KGen}, \mathsf{Derive}, \mathsf{Prove}, \mathsf{Verify})$ that work as described below.

- $\mathsf{KGen}(1^\lambda, 1^N) \to \mathsf{srs}_{\text{univ}}$ is a probabilistic algorithm that takes as input a security parameter and a bound N and outputs a universal structured reference string $\mathsf{srs}_{\text{univ}}$.
- $\mathsf{Derive}(\mathsf{srs}_{\text{univ}}, \mathcal{R}) \to (\mathsf{ek}, \mathsf{vk})$ is a *deterministic* algorithm that takes as input a universal SRS $\mathsf{srs}_{\text{univ}}$ and a specific relation $\mathcal{R}$ and outputs an evaluation key ek and a verification key vk.
- $\mathsf{Prove}(\mathsf{ek}, x, w) \to \pi$ takes as input an evaluation key ek, a statement x, and a witness w such that $\mathcal{R}(x, w)$ holds, and returns a proof π.
- $\mathsf{Verify}(\mathsf{vk}, x, \pi) \to b$ takes as input a verification key vk, a statement x, and a proof π and either accepts $(b = 1)$ or rejects $(b = 0)$ the proof π.

A SNARK satisfies the following properties: **Completeness and Succinctness.** For all $\lambda, N \in \mathbb{N}$ for all relations $\mathcal{R}$ (described as an R1CS of at most N constraints), for all x, w such that $\mathcal{R}(x, w) = 1$ it holds that $\mathsf{Verify}(\mathsf{vk}, x, \mathsf{Prove}(\mathsf{ek}, x, w)) = 1$ where $(\mathsf{ek}, \mathsf{vk}) \leftarrow \mathsf{Derive}(\mathsf{KGen}(1^\lambda, 1^N), \mathcal{R})$. Succinctness holds if the running time of Verify is $\mathsf{poly}(\lambda)(1 + |x| + \log |w|)$ and the proof size is $\mathsf{poly}(\lambda)(1 + \log |w|)$. **Extractability.** This property[11] states we can efficiently "extract" a valid witness from a proof that passes verification. This is modeled through the existence of an efficient machine, an extractor, returning the witness. As done in other works, we assume that the code of the extractor may depend on that of the adversary. **Zero-knowledge.** This property states that a proof leaks nothing about the witness. This is modeled through a simulator that can output a valid proof for an input in the language without knowing the witness.

[11] We describe extractability and zero-knowledge only informally. We refer the reader to [12] for a formal definition.

C.4 MIPP Commitments and proofs

Given a vector $\vec{A} = [A_1, \dots, A_M]$ of group elements in $\mathbb{G}_1$, the IPP commitment to $\vec{A}$ with a CRS $\vec{h} = [h_1, \dots, h_M]$ of group elements in $\mathbb{G}_2$ is

$$T = CM(\vec{A}, \vec{h}) = \prod_{i=1}^{M} e(A_i, h_i).$$

In [9] a suite of *Inner Pairing Product Proofs* are presented and we are interested specifically in MIPP_k which allows to efficiently prove the following statement:

– Assume the CRS is defined as $h_i = h^{s^i}$ for a random secret element in $s \in_R Z_q$
– Given T defined as above for an arbitrary vector of $\mathbb{G}_1$ group elements $\vec{A}$ and the above structured CRS
– Given a vector $\vec{b} = [b_1, b_2, ...b_M]$ with $b_i = b^i \bmod q$ for an arbitrary $b \in Z_q$
– Prove that
$$U = \langle \vec{A}, \vec{b} \rangle = \vec{A}^{\,\vec{b}} = \prod_{i=1}^{M} A_i^{b_i},$$

The MIPP_k proof has size and verification time $O(\log M)$, which means that the verifier needs only to read the value b and not construct the entire vector $\vec{b}$, which is only implicitly defined. We do not describe the protocol here since we describe a generalization of it in Section 3.

D More on Curve Choices

Recall how an elliptic curve has both a scalar field and a base field. In the case of BLS12-377, we denote the scalar field as $\mathbb{F}_r$, the base field as $\mathbb{F}_q$ and $|E(\mathbb{F}_q)| = r$. On the other hand, BW6-761 has scalar field $\mathbb{F}_q$ (where this corresponds exactly to the base field of BLS12-377) and base field $\mathbb{F}_t$ with $|E(\mathbb{F}_t)| = q$. Normally, if we want to design a constraint system using BLS12-377, this is built on the scalar field $\mathbb{F}_r$ and then proven by Groth16 in $\mathbb{F}_q$ through an application of the billinear map e to points in base groups $\mathbb{G}_0$ and $\mathbb{G}_1$. However, this approach does not work if the constraint system includes operations on points (which is the case in Testudo for the polynomial commitment). This is where 2-chaining of pairing-equipped elliptic curves aid our design because they allow us to write a constraint system in $\mathbb{F}_q$, including point operations, and produce a Groth16 proof using BW6-761 on $\mathbb{F}_t$. This way, the Testudo Verifier only has to perform the verification of a Groth16 proof which is constant. We name this Testudo77.

On the other hand, Filecoin proof are constructed using BLS12-381 which is arguably a more popular curve in the ecosystem. We were interested to see how Testudo can be adapted to work in this setting, which has the additional advantage that Filecoin storage provider would not have to re-encode their storage but rather can use Testudo off-the-shelf. One main reason why the re-encoding is required is because the hash functions usrf in Filecoin is Poseidon which, unlike SHA256, requires field operations and produces outputs depending on the field. The main issue with this curve is that it has no *"outer curve"* sister with high 2-adicity in its scalar field, required to enable efficient Fast-Fourier Transforms which is one of the most expensive operations in Groth16. Hence, we cannot use Groth16 naively as for Testudo77. We considered several theoretical alternatives to also enable Testudo81; one example is considering a dlog-based SNARK system, such as Spartan (!) building on a curve whose order would match the one of BLS12-381. This is however left as future work.