Proposal: The Turbo-PLONK program syntax for specifying SNARK programs

Ariel Gabizon and Zachary J. Williamson (Aztec Protocol)

Introduction

Recent zk-SNARK constructions such as Marlin and PLONK relying on Polynomial commitment schemes are not inherently tied to R1CS constraints to specify the statement to be proven. This is roughly because the verifier equation is checked "in the clear" on a random opening of the prover polynomials, rather than "in the exponent" using a pairing. We provide a framework to capture more general and flexible constraints which we call *turbo-PLONK programs*. We give an example of how this framework allows for more concise representation of fixed-base elliptic curve scalar multiplication (only 1 turbo-PLONK gate for each two input bits), a primitive useful for constructing Pedersen hashes. We also include benchmarks of proving time for scalar multiplication on the Grumkin curve (a 255 bit curve embedded over bn254). Our results indicate a 2x improvement over Groth16.

Turbo PLONK programs

A turbo-PLONK program $\mathscr{P}$ can be thought of as sequence of states $v \in \mathbb{F}^w$, where the *state size* w is chosen by the program designer. The proof size and prover run time increase linearly in w and standard choices are 3 or 4.

A *valid execution trace* $T \in (\mathbb{F}^w)^t$ for $\mathscr{P}$ satisfies $\mathscr{P}$'s *transition gate* and *copy constraint*.

transition gate - A transition constraint of $\mathscr{P}$ is a $2w+\ell$-variate degree at most d polynomial P, where d is the *degree* of $\mathscr{P}$, and is recommended to be set to $w+1$, and ℓ is the *selector number*.

The transition gate of $\mathscr{P}$ consists of all the transition constraints together with ℓ *selector vectors* $q_1,\ldots,q_\ell \in \mathbb{F}^t$. T is said to satisfy the transition gate if for each transition

constraint $i \in [t]$, $P(T_{i,1}, \ldots, T_{i,w}, T_{i+1,1}, \ldots, T_{i+1,w}, q_1(i), \ldots, q_\ell(i)) = 0$.

copy constraint - this is a partition of $w \times t$ into distinct sets $\{S_i\}$. T is said to satisfy the copy constraint if $T_{i,j} = T_{i',j'}$ whenever (i,j) and (i',j') belong to the same set in the partition.

The copy constraint can be thought of as enabling memory access, since we can force a subsequent value in the execution trace to be equal to a preceding one.

The special case of arithmetic circuits

Let's see how an arithmetic circuit can be represented in this format. Every row will correspond to a gate, and the row values will be the incoming and outgoing wire values of the gate; so to represent fan-in t circuits we'll use $w = t + 1$. For example, for fan-in 2 the row values v_1, v_2, v_3 will correspond to the left,right and output wire values of the gate associated with the row.

The transition constraints will be somewhat "degenerate" in the sense that we won't use the "next row" values $T_{i+1,1}, \ldots T_{i+1,w}$. They will check if the correct input and output relations hold inside the row according to whether it's an addition or multiplication gate. This can be done with 3 selectors and one constraint polynomial P of degree $t + 1$. E.g., for fan-in 2 we use:

$$P(q_L, q_R, q_M, x_1, x_2, x_3) := q_L \cdot x_1 + q_R \cdot x_2 + q_M \cdot x_1 x_2 - x_3$$

By setting $q_L(i) = q_R(i) = 1, q_M(i) = 0$ for an addition gate and $q_L(i) = q_R(i) = 0, q_M(i) = 1$ for a multiplication gate, we can see the above equation checks the correct input/output relation in both cases.

The copy constraint will enforce the wiring of the circuit. For example, if the left wire of the 4th gate is the output wire of the second gate, we will enforce $T_{2,3} = T_{4,1}$.

Common techniques

Before describing our main example, we discuss two basic techniques that will be used in it.

Look Ups

We describe a basic primitive that will be useful in the next section. Suppose we have two selectors q_1, q_2. We wish to enforce that, say, the first value in row i, is equal to $q_b(i)$ where b is the second value in the same row. This can be obtained by the constraint

$$x_1 = -q_1 \cdot (x_2 - 2) + q_2 \cdot (x_2 - 1)$$

A convenient trick that we will use in the next section, is that in the special case where we wish to choose between two values, and their negations, e.g. according to the third row value being 1 or -1, we can do it using the equation
$x_1 = (-q_1 \cdot (x_2 - 2) + q_2 \cdot (x_2 - 1)) \cdot x_3$.

Interleaving PLONK Gates

After designing several PLONK transition gates (recall that a transition gate means a *set* of $2w + \ell$ - variate polynomials, together with values for ℓ selector polynomials.) , one may wish to design a program that enforces a different subset of these constraints depending on the row. For example, given two gates G_1, G_2 we might want to enforce only G_1 on some row transitions, only G_2 on other row transitions, and both constraints on the rest. The generic way to achieve this is to add two selectors q_{G_1}, q_{G_2} that will be zero or one according to whether we want the corresponding gate to be activated at a given row. Then we can multiply each constraint P in G_i by q_{G_i}, and define the new gate's constraints to be the union of these constraints from both gates.

Example: Fixed-base scalar multiplication

The NAF scalar representation

We assume our scalar s is in the range $\{1, .., M\}$ for M smaller than half the elliptic curve group order having the form $2 \cdot 4^n - 1$ for some integer n. (Allowing a general scalar requires some additional work which we omit here for simplicity of presentation.) When s is in this range, we can write $s = t + \sum_{i=0}^{n-1} b_i \cdot 4^i$, where $b_i \in \{-3, -1, 1, 3\}$, and $t \in \{4^n, 4^n + 1\}$. Thus, we think of our input as consisting of n input quads $\{b_i\}$ and one input bit t.

Our program consists of n rounds where in round i, we add either $-3[g_i], -1[g_i], 1[g_i], 3[g_i]$, where $g_i = 4^{n-i}[g]$ is a precomputed power of our actual generator. We initialize our sum as $t[g]$. An optimization we use to reduce program width, is that we only explicitly represent "intermediate sums" of our scalar, and not the actual input quads (the intermediate sums are more convenient than the quads in checking the correct scalar s was used). We define the intermediate sums as follows: $a_0 = t/4^n, a_1 = t/4^{n-1} + b_{n-1}$, and for $i \in \{2, \ldots, n\}$ $a_i = 4 \cdot a_{i-1} + b_{n-i}$. Induction shows we get $a_n = s$.

Another optimization we use is that by squaring the difference of two intermediate sums, we cancel the effect of the sign of the input quad, which is useful since the x coordinate of the point we want to select only depends on the magnitude on the difference. (And needing to explicitly represent this magnitude would again increase the width of the final program.)

2-bit NAF addition gate

We describe a width 4 program that given row values $(x_1, y_1, x_\alpha, a_{i-1}), (x_2, y_2, x_{\alpha,2}, a_i)$, checks that $(x_2, y_2) = (x_1, y_1) + (a_i - 4a_{i-1})[g_i]$.

Selecting the correct point to add

The first step is to use a variant of the lookup method from the previous section to choose the coordinates of the correct point out to add in a given round.

Let's focus on a specific round $i \in [n]$ and denote $g_i = (x_\beta, y_\beta), 3[g_i] = (x_\gamma, y_\gamma)$. Let's also denote by (x_α, y_α) the point we wish to add in this round according to the input; thus (x_α, y_α) is one of the points in the set $\{(x_\beta, y_\beta), (x_\beta, -y_\beta), (x_\gamma, y_\gamma), (x_\gamma, -y_\gamma)\}$. We can recover x_α via the following relationship:

$$\begin{aligned} &\frac{(a_i - 4a_{i-1})^2 - 9}{-8} x_\beta + \frac{(a_i - 4a_{i-1})^2 - 1}{8} x_\gamma = x_\alpha \\ \implies &(a_i - 4a_{i-1})^2 \frac{x_\gamma - x_\beta}{8} + \frac{9x_\beta - x_\gamma}{8} = x_\alpha \end{aligned}$$

We can represent the precomputed constants by selectors:

$$\begin{aligned} q_{x_{\alpha,1}} = \frac{x_\gamma - x_\beta}{8}, \quad q_{x_{\alpha,2}} = \frac{9x_\beta - x_\gamma}{8} \\ \implies (a_i - 4a_{i-1})^2 q_{x_{\alpha,1}} + q_{x_{\alpha,2}} = x_\alpha \end{aligned}$$

By storing x_α as a distinct wire value, we can extract the y-coordinate, y_α, using a cubic polynomial identity

$$\begin{aligned} &\frac{(x_\alpha - x_\gamma)(a_i - 4a_{i-1})}{(x_\beta - x_\gamma)} y_\beta + \frac{(x_\alpha - x_\beta)(a_i - 4a_{i-1})}{3(x_\gamma - x_\beta)} y_\gamma = y_\alpha \\ \implies &\left(x_\alpha \frac{3y_\beta - y_\gamma}{3(x_\beta - x_\gamma)} + \frac{x_\beta y_\gamma - x_\gamma y_\beta}{3(x_\beta - x_\gamma)} \right)(a_i - 4a_{i-1}) = y_\alpha \end{aligned}$$

We can also represent this via two precomputed selectors

$$\begin{aligned} q_{y_{\alpha,1}} = \frac{3y_\beta - y_\gamma}{3(x_\beta - x_\gamma)}, \quad q_{y_{\alpha,2}} = \frac{x_\beta y_\gamma - x_\gamma y_\beta}{3(x_\beta - x_\gamma}) \\ \implies (x_\alpha q_{y_{\alpha,1}} + q_{y_{\alpha,2}})(a_i - 4a_{i-1}) = y_\alpha \end{aligned}$$

We can recover y_α^2 by computing $x_\alpha^3 + b_{curve}$, where b_{curve} is a constant that describes the short weierstrass curve in question (for our BN254 embedded curve, $b_{curve} = -17$)

Finally, we also check that the differences of intermediate sums fall within the prescribed range via the equation

$$(a_i - 4a_{i-1} + 3)(a_i - 4a_{i-1} + 1)(a_i - 4a_{i-1} - 1)(a_i - 4a_{i-1} - 3) = 0$$

We've seen how to select the correct point to add in a given round, let's now see how to actually add it.

Adding the selected point

As we are adding in each round powers of disjoint magnitude of our generator, we can use an incomplete affine addition formula that doesn't hand repeated x coordinate. To add point (x_1, y_1) with point (x_α, y_α), to obtain output (x_2, y_2), the formula is:

$$\begin{aligned} x_2 &= \left(\frac{y_\alpha - y_1}{x_\alpha - x_1}\right)^2 - x_\alpha - x_1 \\ y_2 &= \left(\frac{y_\alpha - y_1}{x_\alpha - x_1}\right)(x_1 - x_2) - y_1 \end{aligned}$$

Expressed as an identity we can evaluate via a gate, we have

$$\begin{aligned} (x_2 + x_\alpha + x_1)(x_\alpha - x_1)^2 - (y_\alpha - y_1)^2 &= 0 \\ (y_2 + y_1)(x_\alpha - x_1) - (y_\alpha - y_1)(x_1 - x_2) &= 0 \end{aligned}$$

Affine addition using precomputed selectors

x-coordinate check

$$
\begin{gathered}
(x_2 + x_1 + x_\alpha)(x_\alpha - x_1)^2 \\
+ 2(a_i - 4a_{i-1})(x_\alpha q_{y_{\alpha,1}} + q_{y_{\alpha,2}})y_1 \\
- y_1^2 - b_{curve} = 0
\end{gathered}
$$

y-coordinate check

$$
(y_2 + y_1)(x_\alpha - x_1) - ((a_i - 4a_{i-1})(x_\alpha q_{y_{\alpha,1}} + q_{y_{\alpha,2}}) - y_1)(x_1 - x_2) = 0
$$

Initialization gate

What is left to do is to add the "offset" t. We do this by initializing the "starting point" (x_0, y_0) as either $(4^n)[g]$ or $(4^n + 1)[g]$, according to whether the value of a_0 is 1 or $1 + 1/4^n$ .We also check that a_0 is in this range using the corresponding degree two constraint.

The Final Program

We summarize the ideas above and describe the program. We will have n+1 rows each containing values labeled x_i, y_i, x_α, a . The 2-bit NAF addition gate will be active on the first n rows.In the first row the initialization gate will also be active.

$$
\begin{pmatrix}
x_0 & y_0 & x_{\alpha,0} & a_0 \\
x_1 & y_1 & x_{\alpha,1} & a_1 \\
\vdots & \vdots & \vdots & \vdots \\
x_n & y_n & x_{\alpha,n} & a_n
\end{pmatrix}
$$

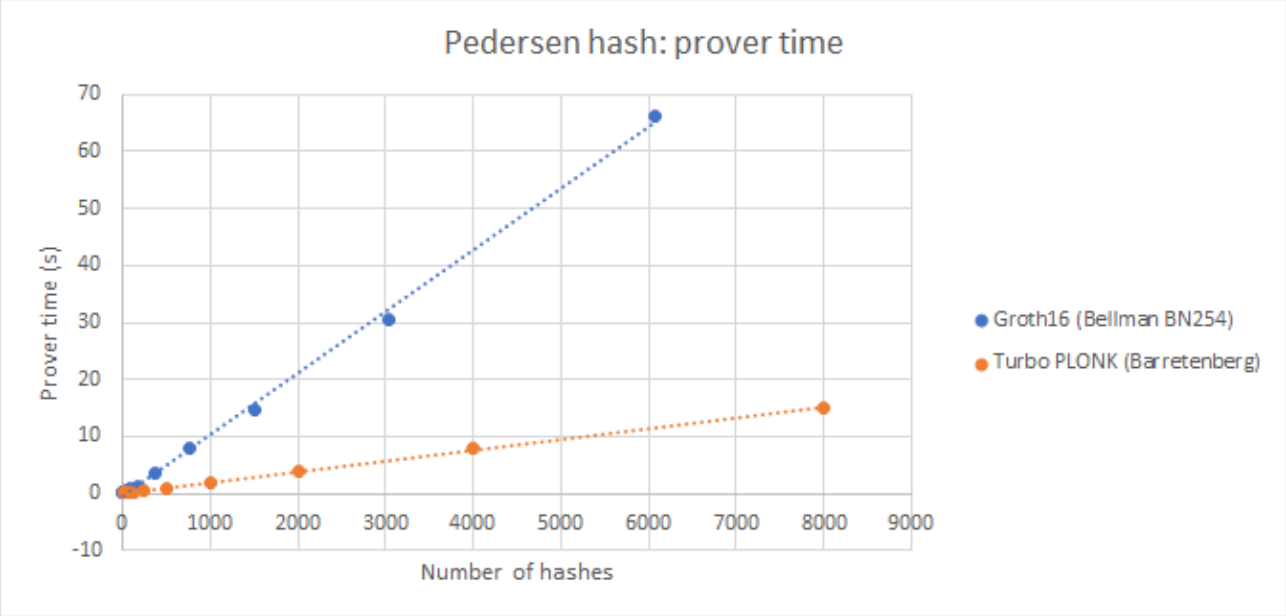
Implementation and Benchmarks

Pedersen hash benchmarks

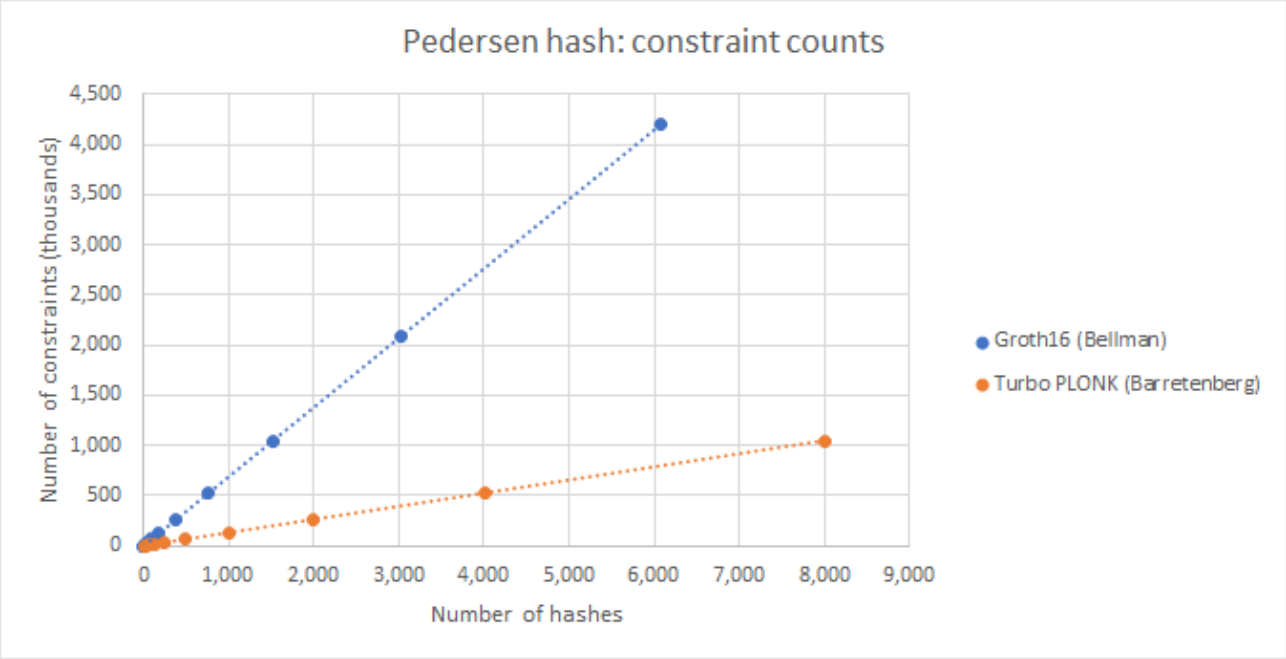
N.B. all benchmarks were measured on a Surface Pro 6, with an i7-8650U CPU at 2.1GHz, 4 physical cores and 16GB RAM.

Our open source library, barretenberg (https://github.com/AZTECProtocol/barretenberg), contains an implementation of TurboPLONK over the BN254 curve. This variant implements Pedersen hashes using our fixed-base scalar multiplication gate.

We benchmarked barretenberg against the current fastest Groth16 prover that also supports the BN254 curve, Bellman. TurboPLONK is approximately five times faster than the Groth16 implementation.



When comparing raw constraint counts, TurboPLONK requires 5 times fewer constraints than an equivalent R1CS-base SNARK

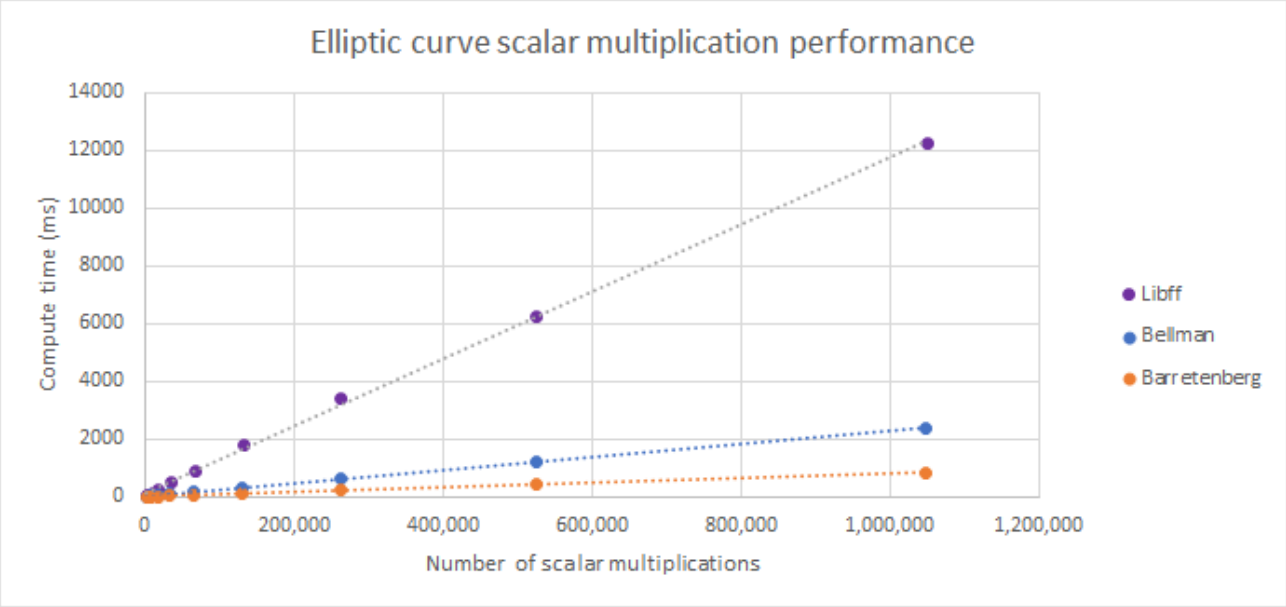


It should be noted that a TurboPLONK constraint requires approximately twice as much prover 'work' than a Groth16 constraint. 20% of the Groth16 constraints are also bit decompositions; these binary gates are approximately 4 times as expensive for the prover in TurboPLONK than Groth16. (Since PLONK doesn't have the property that the scalars in prover computation are actually equal to the witness.)

Under this model, the TurboPLONK prover should be approximately 2.25 times faster than Groth16.

New advances in multi-scalar-multiplication over elliptic curves

The remaining speed advantage between the TurboPLONK prover and the Groth16 prover can be explained by considering the relative speeds of the two libraries when computing multi-scalar-multiplications in G1, where the bulk of prover time is spent



Using original techniques for performing this algorithm, Barretenberg can compute over 1 million scalar multiplications in < 1 second on average consumer-grade hardware.

Pippenger's multi-exponentiation algorithm is the fastest known method to compute a multiple scalar-multiplication over different bases. Our new contribution is the observation that, when computing multi-products, using the *affine* point addition formula requires substantially fewer field multiplications than formulae for projective and jacobian coordinates.

When evaluating affine point addition, the formula is:

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1} \bmod p$$
$$x_3 = \lambda^2 - (x_2 + x_1) \bmod p$$
$$y_3 = \lambda(x_1 - x_3) - y_1 \bmod p$$

Computing λ requires a modular inverse, which is orders of magnitude more expensive than a field multiplication.

However, the Pippenger multi-products can be arranged to produce sequences of independent point additions. That is, the outputs of additions in the sequence are not inputs to any additions in the sequence.

This allows for the use of Montgomery's trick to compute batch modular inverses

$$\forall i \in \{1, \ldots, n\} : d_i = \prod_{j=1}^{i-1} a_j$$
$$s = (d_n a_n)^{-1}$$
$$\forall i \in \{1, \ldots, n\} : e_i = s \prod_{j=i+1}^{n} a_j$$
$$\forall i \in \{1, \ldots, n\} : r_i = d_i e_i$$

Assuming a sufficiently large n, the cost of a batched modular inverse is 3 field multiplications per inverse. This produces a cost of 6 field multiplications to compute a point addition.

For short Weierstrass curves, the traditional mixed-addition (projective) formula requires 11 field multiplications.
