Plonky2:
Fast Recursive Arguments with PLONK and FRI

Polygon Zero Team

DRAFT
February 6, 2022

Abstract

We describe Plonky2, an implementation of cryptographic arguments with an emphasis on fast recursive composition. On a commodity laptop, Plonky2 takes about 300 milliseconds to generate a recursive proof.

Plonky2's arithmetization is based on TurboPLONK, but it replaces PLONK's polynomial testing scheme with a different one based on FRI. Since FRI does not require a large-characteristic field, we encode the witness in a 64-bit field, which greatly improves prover performance.

We also demonstrate how recursion can be used to shrink arbitrarily large proofs to a constant size. Plonky2 can shrink any proof to about 43 kilobytes, depending on desired security and latency.

Contents

1 Introduction

While the idea of proof composition dates back to the PCP literature [1], achieving practical recursion has been a major challenge. Recursive composition was first realized in practice using cycles of elliptic curves [2]. While this was an important milestone, its use of large MNT curves limited its practicality.

More recently, Halo [3] demonstrated how recursion can be realized without pairings, enabling much smaller elliptic curves to be used. Subsequent works like Nova [4] build upon Halo's accumulation strategy, reduce certain incremental verification costs.

One challenge with schemes like Halo is that, while most of the work to verify a proof involves that proof's base field, constraint evaluation must be done in its scalar field. Since that is not the native field of the verification circuit, these checks are typically deferred to the next proof in a recursion chain. However, this deferral strategy relies on public inputs, and verifying public inputs itself requires some use of non-native arithmetic.

To avoid the difficulties associated with elliptic curve cycles, we turn to FRI [5], which supports any prime field with smooth subgroups. While Fractal [6] previously demonstrated FRI-based recursion, its recursion threshold was 2^{21} R1CS constraints, resulting in proof times on the order of minutes.

We achieve a smaller recursion threshold of 2^{12} gates using a PLONK [7] based arithmetization, with custom gates tailored to the verifier's bottlenecks. We also use a smaller field which results in much faster proving times.

2 Field selection

We encode witness data in a prime field $\mathbb{F}_p$, with $p = 2^{64} - 2^{32} + 1$. This field was chosen for speed of computation: its elements fit within a 64-bit word, and the structure of of p gives rise to an efficient reduction method.

To see why reduction modulo p is uniquely efficient, observe that

$$2^{64} \equiv 2^{32} - 1 \pmod{p},$$

and consequently

$$\begin{aligned} 2^{96} &\equiv 2^{32}\left(2^{32} - 1\right) \pmod{p} \\ &\equiv 2^{64} - 2^{32} \pmod{p} \\ &\equiv -1 \pmod{p}. \end{aligned}$$

To reduce a 128-bit number n, we first rewrite n as $n_0 + 2^{64}n_1 + 2^{96}n_2$, where n_0

is 64 bits and n_1, n_2 are 32 bits each. Then

$$\begin{aligned} n &\equiv n_0 + 2^{64} n_1 + 2^{96} n_2 \pmod{p} \\ &\equiv n_0 + (2^{32} - 1) n_1 - n_2 \pmod{p}. \end{aligned}$$

To perform this in computer arithmetic, we first compute $(2^{32} - 1)n_1$, which may be done with a bitshift and subtraction. We add the first two terms modulo 2^{64}, subtracting p if overflow occurs.[1] We then subtract n_2 modulo 2^{64}, adding p if underflow occurs.[2] At this point we have reduced n to a 64-bit integer. This partial reduction is adequate for most purposes, and the canonical form can be obtained by a final conditional subtraction.

Our choice of p also reduces the need to keep constants in the registers. In particular, an overflowing addition requires us to subtract p as a correction. Subtracting p is equivalent to adding $2^{64} - p = 2^{32} - 1$ in 64-bit arithmetic. Conveniently, many architectures have an instruction to set a register to either 0 or $2^{32} - 1$ depending on the carry flag; for example, x86 has `SBB` and ARM64 has `CSETM`. Therefore, even though p appears as a constant in the reduction algorithm, it does not need to be kept in a register; this reduces register pressure and the overhead of loading constants.

2.1 Extension field

In certain parts of our protocol, a larger field is required for soundness. Following [8], we use an extension field in those cases, specifically $\mathbb{F}_p[X]/(X^2 - 7)$.

3 PLONK modifications

3.1 Custom gates

Following prior work like TurboPLONK [9], Plonky2 makes extensive use of custom gates. To illustrate the model, suppose we were designing a gate for (field) division, $q = x/y$, or equivalently, $(qy = x) \wedge (y \neq 0)$. To enforce $y \neq 0$, we ask the prover to supply a purported inverse, $i = 1/y$. We then enforce the constraints

$$\begin{aligned} qy &= x, \\ yi &= 1. \end{aligned}$$

More concretely, we would map these variables to wire polynomials. If we pick $w_1(x), \ldots, w_4(x)$ for x, y, q and i respectively, then after rearranging, our con-

[1] Such subtraction must underflow, thus cancelling the error.

[2] Similarly, such addition must overflow.

straints become

$$w_3(x)w_2(x) - w_1(x) = 0,$$
$$w_2(x)w_4(x) - 1 = 0.$$

Note that these constraints should be enforced only on rows of our trace which correspond to division gates. A simple solution is to preprocess a polynomial $d(x)$, defined by

$$d(g^i) = \begin{cases} 1 & \text{if the } i\text{th gate is a division gate,} \\ 0 & \text{otherwise,} \end{cases}$$

and use $d(x)$ to "filter" our division constraints, giving

$$d(x)(w_3(x)w_2(x) - w_1(x)) = 0,$$
$$d(x)(w_2(x)w_4(x) - 1) = 0.$$

3.1.1 Tree based filtering

If we have k custom gates, introducing k of these filter polynomials would substantially increase the cost of our opening protocol. We can achieve the same effect with $\lceil \log(k) \rceil$ filter polynomials if we organize our gates into a binary tree, then introduce one filter polynomial $f_i(x)$ for each depth of the tree.

Each gate is then filtered according to its path from the root. For example, a gate with a path of $(1, 0, 1)$ would be given a filter expression of $f_1(x)(1 - f_2(x))f_3(x)$.

Note that incorporating these filter expressions increases each gate's constraint degree based on its distance to the root of the tree. To minimize the degree of our overall constraint system, it is desirable to place higher-degree gates closer to the root. Plonky2 handles this automatically, doing an exhaustive search for the tree arrangement that minimizes constraint degree and tree height.

3.2 Advice wires

Certain trace elements do not need to be wired to any other trace elements. For example, it is common to implement division nondeterministically, by allocating a column for a purported inverse. We refer to these as *advice wires*, and we exclude them from PLONK's permutation argument.

This reduces the number of σ_i polynomials in our protocol, which reduces the degree of the permutation argument. Since there are fewer polynomials to open, it also reduces proof sizes and saves the verifier some work.

3.3 Cumulative products

Suppose our circuit has r routed wires. PLONK's permutation argument imposes the following degree $r + 1$ constraint:

$$Z(x) \prod_{i=1}^{r} f'_i(x) = Z(gx) \prod_{i=1}^{r} g'_i(x).$$

In our case, r is large, and such a high-degree constraint would be undesirable.

We can rewrite the above as

$$Z(gx) = Z(x) \prod_{i=1}^{r} f'_i(x)/g'_i(x).$$

To obtain lower-degree constraints, we will split the product above into chunks of 8 terms, and introduce a prover polynomial $c_i(x)$ to hold each cumulative product. Let $s = \lfloor r/8 \rfloor$; then we have

$$\begin{aligned}
c_1(x) &= Z(x) \prod_{i=1}^{8} f'_i(x)/g'_i(x) \\
c_2(x) &= c_1(x) \prod_{i=9}^{16} f'_i(x)/g'_i(x) \\
&\vdots \\
Z(gx) &= c_s(x) \prod_{i=8s+1}^{r} f'_i(x)/g'_i(x).
\end{aligned}$$

These equations are suitable for deriving the values of $c_i(x)$ and $Z(gx)$ at each $x \in H$, but they cannot directly be used as polynomial constraints, since they contain rational functions which are not necessarily polynomials. To obtain polynomial constraints, we multiply by each $g'_i(x)$ term, giving constraints like

$$c_1(x) \prod_{i=1}^{8} g'_i(x) = Z(x) \prod_{i=1}^{8} f'_i(x),$$

and so on for the other cumulative products.

3.4 Soundness analysis

The PLONK paper [7] assumes a large field, so any soundness errors inversely proportional to the field size are treated as negligible. Since our field size is much smaller than our security target, we must do a more detailed analysis, and modify parts of the protocol to boost soundness.

3.4.1 Permutation argument

We begin with a direct analysis of PLONK's permutation argument. Suppose that $g \neq \sigma(f)$; hence there exists some j such that $g(g^j) \neq f(g^{\sigma(j)})$. We would like to bound the probability that, for random $\beta, \gamma \in \mathbb{F}^2$,

$$\prod_{i \in [n]} (f(g^i) + \beta i + \gamma) = \prod_{i \in [n]} (g(g^i) + \beta \sigma(i) + \gamma).$$

We can treat both sides as factored polynomials with variables β and γ. Consider the factor of the left hand polynomial corresponding to $i = \sigma(j)$, namely

$$f(g^{\sigma(j)}) + \beta \sigma(j) + \gamma.$$

If the right hand polynomial contained the same factor, it must also have $\sigma(j)$ as the coefficient of β, so it must be the factor associated with $i = j$, namely

$$g(g^j) + \beta \sigma(j) + \gamma.$$

But since $g(g^j) \neq f(g^{\sigma(j)})$, these factors are not in fact equal, so the polynomials are not equal. Since their total degree is n, the Schwartz-Zippel lemma implies that our constraint holds with probability at most $n/|\mathbb{F}|$.

3.4.2 Combining constraints

Like many IOPs, PLONK combines a set of constraints, $c_0, \ldots, c_{l-1}$, into a single constraint using randomness. In particular, the verifier selects a random $\alpha \in \mathbb{F}$, and the combined constraint is

$$C(x) = \sum_{i=0}^{l-1} \alpha^i c_i(x).$$

Suppose some constraint is not satisfied: $c_j(g^k) \neq 0$ for some j, k. We would like to show that with high probability, $C(g^k) \neq 0$. Consider the polynomial

$$p(\alpha) = \sum_{i=0}^{l-1} \alpha^i c_i(g^k)$$

Since the coefficient of $p(\alpha)$ associated with α^j is nonzero, $p(\alpha)$ is not the zero polynomial, so it has at most l roots by the Schwartz-Zippel lemma. Thus, $C(g^k) \neq 0$ except with probability at most $l/|\mathbb{F}|$ over α.

3.4.3 Boosting soundness

The soundness errors mentioned above are too large for practical use. To boost a subprotocol's soundness, we simply repeat it k times in parallel. By the tight parallel repetition theorem, this results in a soundness error of ϵ^k, where ϵ is the soundness error of the original subprotocol.

For example, repeating the permutation argument three times gives a soundness error of $(n/|\mathbb{F}|)^3$. This is less than 2^{-128} for any $n \leq 2^{21}$.

3.5 Public inputs

In Plonky2, any routable[3] trace element can be marked as public. The circuit itself hashes these public elements, and routes the hash output $h = (h_1, \ldots, h_4)$[4] to a `PublicInputGate`. This `PublicInputGate` simply imposes the constraints

$$\begin{aligned} w_1(x) &= h_1, \\ w_2(x) &= h_2, \\ w_3(x) &= h_3, \\ w_4(x) &= h_4. \end{aligned}$$

By contrast, the constraint system in the original PLONK protocol involved $\mathrm{PI}(x)$, a polynomial obtained by interpolating l public input values. Since hashing lets us "compress" l to four, we can encode public input data as four constants in our constraint system, rather than defining polynomials to encode an arbitrary quantity of public input data.

3.6 Zero-knowledge

PLONK achieves zero-knowledge by adding random multiples of $Z_H(x)$ to trace polynomials, as well as to the permutation polynomial $Z(x)$. This increases the degree of these polynomials just beyond a power of two, which is undesirable in our setting, as both the FFT and FRI algorithms deal with polynomials defined over smooth multiplicative subgroups.

Instead, we blind prover polynomials *before* padding to a power of two. To blind trace polynomials, we add rows filled with random elements to the trace. To blind $Z(x)$, we add pairs of randomized rows to the trace, with copy constraints between each pair of columns. We refer to [10] for more details and a proof of statistical

[3] See Section 3.2 regarding routability.

[4] This assumes a standard configuration, where hash outputs are comprised of four field elements, offering approximately 128 bits of collision resistance.

zero-knowledge. Note that more blinding factors must be adjusted when using PLONK with FRI instead of a PCS.[5]

We then use standard techniques to make the FRI protocol zero-knowledge:

1. Since sending evaluations of prover polynomials on H would directly reveal witness information, we instead send evaluations on a coset of H.

2. As suggested in [11], we transform Merkle trees into hiding vector commitments by wrapping each leaf in a hash-based commitment.

4 Hashing

In most FRI-based proof implementations, hashing is the main bottleneck for both the prover and the verifier. Since the cost to generate a recursive proof is a function of both proving costs and verification costs, the efficiency of FRI-based recursion is highly dependent on our choice of hash function.

To this end, we decided to use Poseidon$^{\pi}$ [12] in a sponge. We use a width of 12 $\mathbb{F}_p$ elements, and x^7 as the S-box. For 128 bit security with the recommended security margin, we use 8 full rounds and 22 partial rounds, for a total of 118 S-boxes.

Plonky2 also supports GMiMC$_{\text{erf}}$ [13] as an option, but although it is more efficient, we avoid using it in practice due to security concerns.

4.1 Hashing in the circuit

In Plonky2, about 75% of the recursive circuit is devoted to verifying Merkle proofs. Thus to obtain a small recursive circuit, it is critical to arithmetize hashes efficiently.

In most other SNARK implementations, a single evaluation of an arithmetic hash takes place over several rows of the trace. For example, an arithmetization of Rescue might use one row per round of the cipher. Round constants may be passed in using preprocessed polynomials, or other techniques like periodic columns [8]. Note that both options have some drawbacks—more preprocessed polynomials means more openings, and periodic columns impose alignment requirements on the placement of rows.

By contrast, Plonky2 uses a single `PoseidonGate` to evaluate an entire instance of Poseidon$^{\pi}$. This allows us to specify round constants in the constraint system itself. To keep our constraints low-degree, we introduce "intermediate value" wires for each S-box input, resulting in constraints of degree 7.

[5] FRI lacks the zero-knowledge property of a PCS opening, TODO.

This design does result in somewhat wide traces; Plonky2 uses 135 columns. There are tradeoffs between narrower, longer traces and wider, shorter traces, but we find this width to be a reasonable balance for minimizing both proof sizes and FRI verification costs.

5 Polynomial testing

A low-degree test such as FRI may be used to construct a polynomial commitment scheme. To commit to a polynomial $p(x)$, the prover simply sends a vector commitment to a low-degree extension of $p(x)$. To open $p(x_0) = y_0$, the FRI protocol is run on $(p(x) - y_0)/(x - x_0)$, which is a polynomial if and only if $p(x_0) = y_0$.

However, this construction assumes that FRI is used with a δ parameter within the decoding radius, $(1 - \rho)/2$. For efficient recursion, we would like a larger δ, in which case the prover is not bound to a single polynomial. Still, after sending a vector commitment to $u \in \mathbb{F}_p^n$, the prover is bound to the set of polynomials in the Hamming ball of radius δ centered around u.

We can then argue that if no polynomial satisfies our constraints, it is unlikely that any polynomial in this Hamming ball satisfies the check at a random ζ. We refer to the DEEP-ALI protocol [14] for a soundness analysis. While the original DEEP-ALI analysis assumed a single witness polynomial, this was later generalized in [15] and [16].

6 Optimizations

6.1 Structure of the trace

It is natural to think of a trace as a matrix, $\mathbb{F}^{r \times c}$, where each row corresponds to the wires of one gate. However, PLONK's copy constraints imply that certain trace elements will always have the same value. These constraints induce a partition of trace elements, where two elements reside in the same set if and only if there is a path between them in the wiring graph.

With this in mind, another natural structure to consider is a disjoint-set forest. When building a circuit, we begin with a singleton set for each trace element. When a copy constraint is added between two elements, we perform a union.

Later, when generating the trace values, we store one value per representative rather than one for each trace cell. This has the benefit that when a value is populated, there is no need for it to be copied to neighboring cells in the wiring graph.

6.2 FRI optimizations

While these techniques are not novel, we apply several optimizations to the FRI protocol:

1. We use the minimal number of Merkle trees needed to carry out the protocol. For example, we combine all preprocessed polynomials, such as PLONK's σ_i and selector polynomials, into a single Merkle tree.

2. When optimizing for proof size, we perform an exhaustive search for the sequence of FRI reduction arities that minimizes proof sizes. This sometimes results in higher arities in earlier FRI rounds, followed by lower arities in later rounds. When optimizing for recursion costs, we used a fixed arity of 8 to make the verifier's work more uniform.

3. Wherever a contiguous block of oracle data would be queried together, we place the entire block in a leaf of the Merkle tree. In Merkle trees from the FRI commit phase, for example, a leaf consists of evaluations over an entire coset rather than a single point.

4. As a size optimization, we prune overlapping Merkle paths. When the same query index occurs twice in the same FRI layer, we omit the response altogether.

5. In the recursive setting, we would not want the variable-length Merkle paths that pruning would create. Instead, we have the prover send "Merkle caps" [17] in place of Merkle roots. This lets us prune a few hashes from each Merkle path, while still keeping path lengths fixed.

6. In the original description of FRI, reductions are performed until the reduced codeword has degree 1. In practice, we terminate the reduction protocol once we reach some reasonably small degree, and the prover sends this reduced polynomial rather than a constant.

7. Following [8], we use grinding to boost security.

6.3 Poseidon

In the MDS layer, we use the circulant MDS matrix whose first row is given by $[1, 1, 2, 1, 8, 32, 2, 256, 4096, 8, 65536, 1024]$. This choice of MDS matrix enables numerous optimizations. Firstly, the MDS matrix is composed of powers of two. This permits implementations to eschew multiplication for bitshifts, which are faster on most hardware. Additionally, the entries of the matrix are relatively low. Integer matrix multiplication, performed with inputs in $0, ..., 2^{64} - 1$,[6] will always

[6]The inputs are not necessarily in canonical form.

yield vector entries smaller than 2^{81}. This bound simplifies the final reduction.

In addition to these two properties, our x86-64 and ARM64 implementations accelerate hashing by making heavy use of SIMD operations and hand-tuned assembly. The circulant property simplifies SIMD implementations—by appropriately permuting the input vector at each step, we can ensure that all entries are multiplied by the same constant. This enables us to use instructions that multiply all elements of a vector by a scalar or by an immediate, reducing register pressure.

7 Evaluation

Plonky2's recursion threshold depends on various settings. When optimizing for prover speed, we typically use a codeword rate of 1/8. Then, when configured for 100 bits of conjectured[7] security, Plonky2 has a threshold of 2^{12} gates. These recursive proofs take approximately 300ms to generate on a 2021 Macbook Air.

When optimizing for size, we use larger codeword rates. Using a rate of 1/256, Plonky2 is able to compress any proof to approximately[8] 43 kilobytes. These proofs take approximately 11.6 seconds to generate, again on a 2021 Macbook Air.

References

[1] S. Arora and S. Safra, "Probabilistic checking of proofs: A new characterization of NP," *Journal of the ACM (JACM)*, vol. 45, no. 1, pp. 70–122, 1998.

[2] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Scalable zero knowledge via cycles of elliptic curves." Cryptology ePrint Archive, Report 2014/595, 2014. https://ia.cr/2014/595.

[3] S. Bowe, J. Grigg, and D. Hopwood, "Recursive proof composition without a trusted setup." Cryptology ePrint Archive, Report 2019/1021, 2019. https://ia.cr/2019/1021.

[4] A. Kothapalli, S. Setty, and I. Tzialla, "Nova: Recursive zero-knowledge arguments from folding schemes." Cryptology ePrint Archive, Report 2021/370, 2021. https://ia.cr/2021/370.

[5] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Fast Reed-Solomon Interactive Oracle Proofs of Proximity," in *45th International Colloquium on*

[7] This is based on the soundness conjecture in [8].

[8] The exact size depends on the verifier's random tape, since the path-pruning optimization results in different compression ratios for different query index sets.

Automata, Languages, and Programming (ICALP 2018), vol. 107 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 14:1–14:17, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018.

[6] A. Chiesa, D. Ojha, and N. Spooner, "Fractal: Post-quantum and transparent recursive proofs from holography." Cryptology ePrint Archive, Report 2019/1076, 2019. https://ia.cr/2019/1076.

[7] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "PLONK: Permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge." Cryptology ePrint Archive, Report 2019/953, 2019. https://ia.cr/2019/953.

[8] StarkWare, "ethSTARK documentation." Cryptology ePrint Archive, Report 2021/582, 2021. https://ia.cr/2021/582.

[9] A. Gabizon and Z. J. Williamson, "Proposal: The Turbo-PLONK program syntax for specifying SNARK programs," 2020. https://docs.zkproof.org/pages/standards/accepted-workshop3/proposal-turbo_plonk.pdf.

[10] D. Lubarov, "Adding zero knowledge to PLONK-Halo," 2020. https://mirprotocol.org/blog/Adding-zero-knowledge-to-Plonk-Halo.

[11] E. Ben-Sasson, A. Chiesa, and N. Spooner, "Interactive oracle proofs." Cryptology ePrint Archive, Report 2016/116, 2016. https://ia.cr/2016/116.

[12] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger, "Poseidon: A new hash function for zero-knowledge proof systems." Cryptology ePrint Archive, Report 2019/458, 2019. https://ia.cr/2019/458.

[13] M. R. Albrecht, L. Grassi, L. Perrin, S. Ramacher, C. Rechberger, D. Rotaru, A. Roy, and M. Schofnegger, "Feistel structures for MPC, and more." Cryptology ePrint Archive, Report 2019/397, 2019. https://ia.cr/2019/397.

[14] E. Ben-Sasson, L. Goldberg, S. Kopparty, and S. Saraf, "DEEP-FRI: Sampling outside the box improves soundness." Cryptology ePrint Archive, Report 2019/336, 2019. https://ia.cr/2019/336.

[15] A. Kattis, K. Panarin, and A. Vlasov, "RedShift: Transparent SNARKs from list polynomial commitment IOPs." Cryptology ePrint Archive, Report 2019/1400, 2019. https://ia.cr/2019/1400.

[16] E. Ben-Sasson, D. Carmon, Y. Ishai, S. Kopparty, and S. Saraf, "Proximity gaps for Reed-Solomon codes." Cryptology ePrint Archive, Report 2020/654, 2020. https://ia.cr/2020/654.

[17] A. Chiesa and E. Yogev, "Subquadratic SNARGs in the random oracle model." Cryptology ePrint Archive, Report 2021/281, 2021. https://ia.cr/2021/281.