Proof Batching Report

Decentriq AG

Motivation

As part of the Filecoin system, each storage provider needs to continuously generate zero-knowledge proofs for retrievability of the data that they are storing. Those proofs ensure that the provider is indeed storing the data throughout the duration of the contract. Crucially, these proofs need to be publicly verifiable, such that anyone can ensure the provider is abiding by the agreed upon terms. This leads to a problem, as the proofs need to be stored on-chain, resulting in significant blockchain bloat in the long-run. One solution to this problem is proof batching. Instead of recording the proofs on-chain, each storage provider keeps the proofs locally and only generates a single proof at a given time interval, which verifies all the retrievability proofs that have been generated during this period. This approach drastically reduces the amount of information need to be stored on the blockchain.

Types of batching

The main idea behind batching is that the storage provider verifies the proofs of retrievability *inside a new SNARK*, by encoding the verification equations in the SNARK. Consequently, this means that the provider should have access to a proof system, which allows efficient pairing computations *inside the SNARK*. This requirement is satisfied by using a proof system with special elliptic curves, which are detailed in the next section.

We discuss two main approaches for proof batching, which we dub “naive” and “non-naive” batching. In naive batching, the storage provider iterates over each individual retrievability proof and verifies its correctness. In non-naive batching, the storage provider leverages properties of the proof system used to produce the retrievability proofs to “combine” the verification equations for the batch, thus reducing the amount of computation that needs to be done for their verification inside the SNARK.

Elliptic curves for batching

For a pairing-based proof system, such as **Groth16** or **GM17**, we need a pairing-friendly elliptic curve in order to be able to efficiently verifier the proof. If we say that F_p is the base field of the curve and that is has a subgroup of large prime order *r*, then:

- the SNARK supports proving statements about arithmetic circuits over F_r
- proof verification requires statements over F_p

This means that any SNARK verifying proofs for this proof system needs to be encoded as statements over F_r. However, this is highly inefficient, as it requires simulating statements for F_p

over F_r. To allow recursive composition of proofs to an arbitrary depth, cycles of elliptic curves have been suggested, e.g. in **BCTV17**. However, those curves have very large fields sizes and are thus not efficient. Furthermore, to do proof batching we don't need arbitrary recursive depth, we only need one layer of recursion. Thus, it suffices to have a second curve, which *has a curve order matching the size of the base field of the curve used to produce original proof.* In other words, we need a curve which has a group order of *p*.

Such a pair of curves were sampled in **Zexe** (see the Section on Zexe for more details). We provide a summary of the parameters for these curves in the table below:

Name	**Curve type**	**Embedding degree**	**Prime order subgroup size**	**Base field size**
BLS12-377	BLS	12	r	p
SW6	Short Weierstrass	6	p	q

with:

- r = 0x12ab655e9a2ca55660b44d1e5c37b00159aa76fed00000010a11800000000001
- p = 0x1ae3a4617c510eac63b05c06ca1493b1a22d9f300f5138f1ef3622fba094800170b5d44300000008508c00000000001
- q = 0x3848c4d2263babf8941fe959283d8f526663bc5d176b746af0266a7223ee72023d07830c728d80f9d78bab3596c8617c579252a3fb77c79c13201ad533049cfe6a399c2f764a12c4024bee135c065f4d26b7545d85c16dfd424adace79b57b942ae9

With this setup, proofs of retrievability can be generated inside a SNARK over BLS12-377 (using Groth16, e.g.) and those proofs could be efficiently batch verified inside a SNARK over SW6 (again using Groth16, e.g.).

We note that an additional technical requirement for this to work is that both *r* and *p* need to have high 2-adicity in order to make proof generation efficient over both curves. 2-adicity is defined as the largest number a s.t. 2^a divides $r-1$ (resp. $p-1$). A large 2-adicity ensures that the finite field over which the prover is doing polynomial interpolations using FFTs, has a large (2^a) root of unity[1], i.e. $\exists x \in F_p$ s.t. $x^{2^a} = 1$.

[1] A n^{th}-root of unity in a finite field is a field element x, s.t. x^n = 1.

To see why this is the case, note that:

- 2^a dividing $p-1$ is equivalent to $2^a * m = p - 1$ for some $m \in F_p$
- Following Fermat's Little Theorem, for any element $x \in F_p$ $x^{p-1} = 1$.

Re-writing the first expression we obtain $m = (p-1)/2^a$. It is now trivial to see that x^m is a 2^a root of unity. For this to be true, we need $(x^m)^{2^a} = 1$. Replacing m with $(p-1)/2^a$ in this equation, we arrive at $(x^{(p-1)/2^a})^{2^a} = x^{p-1}$, which is equal to one following Fermat's little theorem.

Having a large 2^a root of unity is a prerequisite for the efficient computation of the FFT, similarly to the standard computation of FFT over the complex numbers, where the computation relies on the complex roots of unity. With a 2^a root of unity, the computation of the FFT can be done with a simple (and standard) radix-2 FFT algorithm [p. 24 of **https://eprint.iacr.org/2013/507.pdf**][p. 16 of **https://eprint.iacr.org/2013/879.pdf**].

The 2-adicity of *r* is 47, while the one of *p* is 46, which implies that it is efficient to generate proofs of up to 2^47 (resp. 2^46) multiplicative constraints over BLS12-377 and SW6.

Proof systems

In this section, we look at two proof systems and evaluate the number of constraints necessary for proof batching.

GM17

The GM17 proof was designed to address a malleability issue with Groth16, namely that anyone who has seen a valid proof for some relation can generate another valid proof without knowing a valid witness. The proof consists of the same number of elements as in Groth16, but the verification requires checking two equations, which makes it slightly less efficient than Groth16.

We denote elements of the verification key using **the notation from Zexe**, as:

- $h \in \mathbb{G}_2$
- $g\_\alpha \in \mathbb{G}_1$
- $h\_\beta \in \mathbb{G}_2$
- $g\_\gamma \in \mathbb{G}_1$
- $h\_\gamma \in \mathbb{G}_2$
- $query \in$ Vec<$\mathbb{G}_1$>

We denote elements of the proof as:

- $A : \mathbb{G}_1$
- $B : \mathbb{G}_2$

- $C : \mathbb{G}_1$

The two verification equations, written in multiplicative notation, with $e(x, y)$ denoting a pairing, are:

- $e(A * g\_\alpha, B * h\_\beta) = e(g\_\alpha, h\_\beta) * e(g\_\psi, h\_\gamma) * e(C, h)$
- $e(A, h\_\gamma) = e(g\_\gamma, B)$

We note the following:

- g_ψ is a linear combination of the public inputs with the $query$ elements of the verification key. As such, the computation of g_ψ requires a scalar multiplication (circa 3.2K constraints) and a point addition (7 constraints) (both in $\mathbb{G}_1$) for each public input
- the verification equations consist of 5 pairing operations, however in theory the $e(g\_\alpha, h\_\beta)$ pairing can be pre-computed from the verification key outside of the SNARK, so we only need to do 4 pairings inside the SNARK. The implementation of the GM17 proof verification gadget in Zexe, however, computes all the pairings inside the gadget.

The total number of constraints for verifying one GM17 proof inside a SNARK using the **GM17 proof verification gadget** in Zexe is **57543**, with the main contributions coming from:

- g_ψ (ca. 3200 constraints per public input): ca. 12800 constraints in total
- Multi-Miller loop for the 4 pairing operations in the 1st verification equation: 13206 constraints
- Multi-exponentiation for the 1st verification equation: 7857
- Multi-Miller loop for the 2 pairing operations in the 2nd verification equation: 7548
- Multi-exponentiation for the 2nd verification equation: 7857
- Enforcing bit-ness of the public inputs: 1 constraint per bit (the public inputs in our test run were 4 each of 256 bits, so a total of 1024 constraints)
- Check if the proof elements are on the curve and in the correct sub-group: 599 constraints for $\mathbb{G}_1$ elements and 3477 constraints for $\mathbb{G}_2$ elements, for a total of 4675 constraints

The majority of the remaining constraints come from a coordinate transformation of $\mathbb{G}_2$ elements, which makes them more efficient to work with inside the Miller loop.

Groth16

In this section we carry out a similar analysis for the Groth16 proof system, as we did for GM17.

We denote elements of the verification key as:

- $\alpha \in \mathbb{G}_1$
- $\beta\_g1 \in \mathbb{G}_1$
- $\beta\_g2 \in \mathbb{G}_2$

- $\gamma \in \mathbb{G}_2$
- $\delta\_g1 \in \mathbb{G}_1$
- $\delta\_g2 \in \mathbb{G}_2$
- $IC \in$ Vec<$\mathbb{G}_1$>

We denote elements of the proof as:

- $A \in \mathbb{G}_1$
- $B \in \mathbb{G}_2$
- $C \in \mathbb{G}_1$

Constraints for naive batching

The verification equation, written in multiplicative notation, with $e(x, y)$ denoting a pairing, is:

$$e(A, B) * e(ic, -\gamma) * e(C, -\delta\_g2) = e(\alpha, \beta\_g2)$$

Similarly to the GM17 case, we note that:

- ic is a linear combination of IC (from the verification key) and the public inputs, so it requires one scalar multiplication and one addition per public input to compute
- The verification equation consists of 4 pairing operations, however the $e(\alpha, \beta\_g2)$ pairing can be precomputed from the verification key and so in theory only 3 pairing operations are required in the SNARK. Our implementation of the Groth16 verification gadget in Zexe follows the implementation of the GM17 verification gadget and computes all 4 pairings inside the SNARK.

The total number of constraints for verifying one GM17 proof inside a SNARK using the **Groth16 proof verification gadget** in Zexe is **41087**, with the main contributions coming from:

- ic (ca. 3200 constraints per public input): ca. 12800 constraints in total
- Multi-Miller Loop for 4 pairing operations: 13206
- Multi-Exponentiation: 7548
- Enforcing bit-ness of the public inputs: 1 constraint per bit (the public inputs in our test run were 4 each of 256 bits, so a total of 1024 constraints)
- Check if the proof elements are on the curve and in the correct sub-group: 599 constraints for $\mathbb{G}_1$ elements and 3477 constraints for $\mathbb{G}_2$ elements, for a total of 4675 constraints

Similarly to GM17, the majority of the remaining constraints come from a coordinate transformation of $\mathbb{G}_2$ elements, which makes them more efficient to work with inside the Miller loop.

Constraints for non-naive batching

The non-naive batching for Groth16 proofs eliminates the need for applying the final exponentiation to the output of each Miller loop and instead does a single final exponentiation for the verification of the entire batch, thus reducing the overall number of constraints necessary for verifying a batch of proofs, compared to the naive approach. The implementation of the non-naive batching is following **the specification** provided by Ariel Gabizon.

We provide two implementations for the non-naive Groth16 batching: **a native one** (done outside the SNARK) and a **SNARK gadget**. Verifying a batch of 100 proofs requires 1989549 constraints, or approximately 20K constraints per proof. *Note that our implementation currently does not produce a randomized linear combination of the individual proofs, as described in the specification above. Adding this would require an additional two variable-base scalar multiplications per proof in the batch, approximately 3200 constraints each, which would increase the number of constraints per proof to 26K for the non-naive batching.*

Zexe

Protocol Labs so far used **bellman** as a library for building zk-SNARKs. The library is one of several tools published by Zcash / Electric Coin Company to build zk-SNARKS applications in Rust. In particular there is also **pairing** for efficient computation of pairing-friendly elliptic curves and **ff** to define finite fields in a declarative fashion.

Zexe is an academic project out of UC Berkeley to use zkSNARKs for attesting to the correctness of the offline computations in ledger-based systems. In that context, the library provides an implementation for the BLS-377 curve mentioned above, as well as a gadget for naive GM17 proof batching.

The implementations for finite field arithmetic, elliptic curve arithmetic and FFT have been combined in a new crate called *algebra,* which is only partially adapted from the code in the crates mentioned above.

Porting code between Bellman and Zexe

Curves

As a result of the refactoring of the finite field and elliptic curve arithmetic implementation, the code in Zexe is much more structured and new interfaces have been established. That makes it a non-trivial task to port a curve from Bellman to Zexe, as implementation details regarding efficient field arithmetics are needed.

Proof System

The underlying data structures and traits used to represent the zk-SNARK structure and proof system, like *Variable, Index, LinearCombination, SynthesisError,* are identical. Same holds for the API exposed by the proof system implementation like: *create_random_proof, generate_random_parameters, prepare_verifying_key, verify_proof.*

Consequently, it is possible to port a proof system to Zexe with reasonable effort. We have shown this by porting **Groth16 to Zexe**.

Please note that the goal was to have an implementation for Groth16 in Zexe for testing and developing purposes. Hence, we wanted to make as few changes to the very optimised parameter generation and prover implementation as possible. Bellman uses a more generic version of Pippenger's algorithm (tracking the density) for the variable base multi-scalar multiplication than Zexe. Consequently, we also re-use the multi-exponentiation code from bellman also in Zexe. For a production implementation it is strongly recommended to reduce this code duplication and therefore use the *VariableBaseMSM* implementation provided by Zexe for multi-exponentiation. Note that some further investigation on the performance effects of crossbeam vs rayon might be necessary for finalizing this decision. According to the author of Zexe (Pratyush Mishra), they measured a 10% performance hit for using crayon, however, internal tests at Protocol Lab have shown a discrepancy of at least 20%.

Circuits

The traits which need to be implemented for circuits, *Circuit* and *ConstraintSystem,* are almost identical between the two implementation. As a result, most circuits that have been designed using bellman can be trivially ported to zexe. The reader can track the changes needed for porting an example circuit, the *blake2s gadget,* **here**. We note that most API changes can be covered by aliasing of imports. The most involved change is the renaming of the function to create a new namespace from *cs.namespace()* to *cs.ns()* as well as the change that the assignment now needs to be passed into the function as a *Result*. As namespace allocations are used all the time, fixing this can be time-intensive and one can think about either: a) reverting the changes in a fork of Zexe or b) writing a script to automate these changes.

Much more evolved for porting are circuits which directly depend on the embedded JubJub curve, as this means the curve has to be ported and this results in the complexity described in the corresponding section above.

Zexe run-time results

Below we provide a matrix of results for test runs done using different proving systems and curves on Zexe. The goal is to analyze the relative performance of the prover in Groth16 and GM17 over the BLS12-381, BLS12-377 and SW6 curves and to compare the prover running time for Groth16 over BLS12-381 in Zexe and bellman.

https://docs.google.com/spreadsheets/d/1IjgT--1iTXeA_7kQeThOzwHtlzMmu1t728lS7rFZt2g/edit#gid=0

Backend	Proof System	Curve	Blake2s bytes	Circuit Size	Metric	Time	**Comments**
							gm17/zexe parameter generation much more efficient (time and memory)
Bellman	Groth16	BLS381	3072	1M	Proof Generation	10.57	
Bellman	Groth16	BLS381	6144	2M	Proof Generation	20	
zexe	Groth16	BLS381	3072	1M	Proof Generation	10.97	
zexe	Groth16	BLS381	6144	2M	Proof Generation	**21.21**	similar performance between bellman and zexe
zexe	GM17	BLS381	3072	1M	Proof Generation	21.92	
zexe	GM17	BLS381	6144	2M	Proof Generation	**43.98**	2x due to proof system
zexe	GM17	BLS377	3072	1M	Proof Generation	21.996	
zexe	GM17	BLS377	6144	2M	Proof Generation	**43.65**	BLS curves similar
zexe	GM17	SW6	3072	1M	Proof Generation	**84.484**	**3.84088016**
zexe	GM17	SW6	6144	2M	Proof Generation	168.612	
zexe	GM17	SW6	6144	4M	Proof Generation	**OOM**	out of memory on 32gb machine
zexe	Groth16	BLS381	384	128K	Proof Generation	1.388	
zexe	Groth16	SW6	384	128K	Proof Generation	5.9	**4.25072046**

We note the following:

- Zexe Relative performance on BLS12-381 is comparable to Bellman.
- GM17 proof system is 2x slower compared with Groth16 over the same curves
- Groth16 and GM17 perform identically over BLS12-381 and BLS12-377
- The prover over SW6 is about ~4x slower compared with BLS curves (with both GM17 and Groth16)

- The maximum number of constraints we were able to run using SW6 + GM17 before getting OutOfMemory exceptions on a machine with 32GB RAM was 4M.

Proof size for SW6

The proof (for both Groth16 and GM17) is two $\mathbb{G}_1$ elements and one $\mathbb{G}_2$ element. According to the Zexe paper, for SW6 the size of a $\mathbb{G}_1$ element is 104 bytes, while the size of a $\mathbb{G}_2$ element is 312 bytes. This would yield a total of $2 * 104 + 312 = 520$ bytes per proof, which has to be stored on-chain. For comparison, the proof size over BLS12-381 is 192 bytes, so the overall difference is ~2.6x. Thus, when batching N proofs, the overall reduction in the size of the proofs that will have to be stored on-chain will be $N/2.6$.