Sangria: A PLONK-ish folding scheme

Crypto Fairy

Follow

--

Listen

Share

Folding schemes are optimization techniques used by Zero Knowledge Proofs. They target two main areas: memory usage and proof generation time reduction in zkVMs (Zero Knowledge Virtual Machines) and recursive computations, also known as Incrementally Verifiable Computation (IVC). Understanding and addressing these in the context of VMs can be quite complex for someone encountering this topic for the first time. Therefore, in this article, we will focus more on the folding scheme outside of zkVMs before delving into VM-specifics.

In ZKPs, we can divide the prover's execution into two phases: "pre-proof" and proof generation. The "pre-proof" phase is the period where all the necessary data is gathered (witness vectors, public inputs, etc.). This is when the actual computation takes place. After this phase, we have a state that we want to prove. The proof generation phase is where this state is "arithmetized", converted into polynomials, so that we can generate a proof using PLONK or another protocol. Folding schemes for IVC fit between these two phases.

How it works?

Previously, we have identified where folding schemes fit in the proof generation cycle. Now, we will discuss what folding is and how it works.

We are already familiar with arithmetic circuits that are required to generate a proof for a given statement or computation. Consider a very primitive circuit consisting of 3 gates (two multiplication gates and one addition gate), which generates the proof for the following equation: a

×b

+c

×d

=e

.

The wiring and gate locations are fixed; the variable part here is the data (input/output parameters and intermediate results between gates). This means we can actually reuse this circuit to generate a proof for another dataset.

So, to generate the proof for these two datasets, we would typically need to run the proof generation process twice. However, what we can do is generate one proof for both datasets. The datasets have to be "folded" into one instance: the 'a' variable from the first dataset is "folded" with the 'a' variable from the second dataset, 'b' with 'b', and so on.

Folding" two variables (a

′ and a

″) in mathematics is represented as a linear combination of those variables and a random value rr

.

This operation is much cheaper and faster than generating proofs for those two dataset instances separately. However, folding is not without its costs; there will be an overhead, as the new proof must also account for the data folding. Despite this, the overhead is significantly less than the cost of generating two separate proofs. By applying this pattern, we can fold the following variables similarly:

However, if we return to the original equation the circuit aims to prove (a

·b

+c

·d

=e

), we encounter a problem when applying our folding technique. When substituting real numbers into the equation, it becomes clear that this approach does not work as intended:

This demonstrates that the folding approach, as initially outlined, misses an essential element. We will address this problem later in the article. For now, we have identified where in the proving cycle folding schemes are used and how they generally work.

Why to fold?

Before delving into the technical challenges of making folding work, let's first address a more obvious question: Why should we fold two independent dataset instances in the first place? The answer lies in the context of larger computations, where you can divide the task into smaller, reusable circuits. This concept slightly resembles what zkVMs are designed for.

Consider the need to prove a very extensive computation. There are two approaches. The first is to build a circuit specifically for generating the proof for that computation. Unfortunately, this circuit cannot be reused for other tasks. The second approach involves using a zkVM, which, besides performing computations, produces related proofs. In the first scenario, significant time must be spent to develop, compile, and test the circuit. In the second, a zkVM is powered by a set of opcodes (commands) that execute programs written in a language compiled into these opcodes. A prime example is a zkEVM, where programs written in Solidity are translated into EVM-compatible opcodes and executed on the zkEVM.

In a zkVM, each command is a separate circuit, and executing them in a specific order according to a program allows us to generate the proof. It is well understood that the larger the program, the longer it takes to execute, a principle that also applies to proof generation: the larger the state we wish to prove, the more memory and time are required to generate it. Consequently, with every new step of the program, the memory used by the prover grows, and by the end of program execution, the zkVM must generate a proof for this enormous state.

The authors of the Nova paper propose a novel approach to implementing Incrementally Verifiable Computations using a folding scheme. At a higher level, the idea is as follows: during execution time, it is more efficient to generate the proof for the folded data and, at the end of the program, generate the remaining proofs. For instance, consider a program consisting of the following opcodes executed in order: A', C', B', A'', A''', C'', B''. First, the zkVM executes an opcode A with some data and accumulates the witness data, the same happens with C and B. When the next opcode A is executed with a new dataset, we can take the witness data from the first step and fold it with the new data, accumulating the result. A similar process occurs with C and B. In the end, the zkVM needs to generate the proof only for the accumulated "folded" instances of A, B, and C. After every folding, the prover generates a proof, which is then passed to the next step where the folding proof is verified.

Sangria

We have now reached the discussion on the implementation of folding. Earlier, we identified that merely using a linear combination to fold two instances is insufficient, as it results in an imbalance in the equation (a

·b

+c

·d

≠e

). We noted that a crucial piece is missing from the equation, necessitating a more sophisticated approach than previously used. In ZKP, when arithmetising statements, we construct gate constraints (algebraic equations equal to zero).

The Nova paper introduced a folding scheme to R1CS arithmetization. In this work, I decided to experiment by integrating a Plonk-like folding scheme, which is named Sangria, into my original implementation of the Plonk protocol.

Let's consider folding in the context of R1CS with the following formula:

Here, A

, B

, and C

are matrices acting as gate selectors, and vector ww

is our witness vector. The multiplication between A

×w

and B

×w

represents a Hadamard product (element-wise multiplication). For more information on R1CS and their use, you can refer to my previous articles on the Groth16 protocol:

Under the hood of zkSNARK Groth16 protocol (part 1)

Zero Knowledge Proofs (ZKP) is a concept that has garnered much attention in the blockchain industry. Many, including…

medium.com

This formula includes two parts: the fixed matrices A

, B

, and C

that can be reused for generating specific proofs, and the witness vector w

, the variable part. When we have two different witness vector instances w

′ and w

″, folding them using a linear combination results in a witness vector that does not satisfy the R1CS equation:

To correct this imbalance, the authors introduced the concept of “relaxing” the equation by incorporating a “slack” or error vector e

and a scalar value u

:

This relaxed equation accommodates both single and folded witness instances. For a single instance, the parameters are as follows: u

=1 and vector e

must contain all zeros. For a folded instance, parameters u

and e

must be calibrated to maintain equation balance.

Returning to a relaxed PLONK arithmetization — Sangria (aptly named for the idea of having relaxed constraints, what can be more relaxing than drinking Sangria?), let’s consider the original gate constraint equation:

The selector polynomials, denoted as q

, are fixed, and the polynomialsaa

, b

, and c

represent our witness — the data that can be altered to generate proofs for multiple data instances. Yet, folding instances a

, b

, c

using a linear combination results in an unbalanced equation, not equal to zero.

To reintroduce balance, we must incorporate a new scalar variable u

and a “slack” or error vector encoded into a polynomial:

Code

All my previous articles have been accompanied by working code examples, and this article is no exception. The Sangria paper focuses on constraining gates and incorporating a relaxed version into the PLONK protocol, necessitating several modifications elsewhere.

For those interested in the original implementation of the Plonk protocol, you can find the articles here:

Under the hood of zkSNARKs — PLONK protocol: Part 1

PLONK is one of the zero-knowledge proving systems that belong to the SNARK group. Compared to Groth16, which is an…

medium.com

The complete code for the relaxed PLONK protocol is available here:

zkSNARK-under-the-hood/plonk_sangria.ipynb at main · tarassh/zkSNARK-under-the-hood

Implementation of zero knowledge proof protocol - Groth16, Plonk. For education purposes. Not a production ready code…

github.com

In this article, I am focusing on the main areas of modification:

We have two different witness instances, and as you can see, only the a

, b

, c

parts are different; the q

vectors remain the same. Witness vectors a

, b

, and c

are encoded into polynomials and folded together using a linear combination with a random value r

.

Now that we understand what the folded instance is, we can calculate the u

and e

parameters to obtain our gate polynomial:

Fortunately, the permutation part does not need any changes, so rounds 2 and 3 of the protocol do not require updates. However, rounds 4 and 5 need modifications. In round 4, we evaluate the E

polynomial at the ζ

(zeta) value and use it as part of the transcript:

Round 5 must be updated with the E

polynomial evaluation over the ζ

value and the u

parameter, and the polynomial Wζ

must also contain both the E

polynomial and its evaluation:

The verifier must be updated accordingly:

What is missing here is the notion of witness folding represented as a separate polynomial constraint. Implementing such a constraint would embed this polynomial in the same manner as the E

polynomial across all places.

References

:

https://github.com/geometryresearch/technical_notes/blob/main/sangria_folding_plonk.pdf

https://eprint.iacr.org/2021/370.pdf

https://github.com/tarassh/zkSNARK-under-the-hood/blob/main/plonk_sangria.ipynb