This is part of a larger article on “Compiling to NIVC-based ZKVMs”. Feedback is welcome

Recursion

In zero-knowledge-proofs cryptography, recursion is a technique in proving schemes in which the proof of every computation in a set of sequential computations becomes the input or witness of the next NARK (not necessarily succinct), and every proof created proves all the prior claims in the chain.

The two main properties recursion unlocks for SNARKs are compression and composability. Instead of the property “a prover shows knowledge to a verifier, without revealing the underlying fact” that a general proving system provides, recursion enables a prover to show knowledge to a verifier without fully knowing the underlying facts themselves, by taking the proof of a statement as an input. Also, a large proof can be compressed into a small one by composing two or many provers. For example, a fast prover can be run for a large circuit and then use another recursive prover to output a small proof for that smaller circuit. Composing different proof systems, although theoretically possible, is quite difficult in practice.

There are three axes that any proving system aims to optimise: proving time (i.e. the cost of proof generation), proof size and verification time. Still, the optimisation of each of these three axes seemingly comes at a cost to the other two. For example, the trade-off for having a short proof is generally having a slow prover.

[

Screenshot 2024-01-14 at 22.23.23

1326×896 50 KB

](https://europe1.discourse-cdn.com/standard20/uploads/anoma1/original/1X/04fdba930353ff756756c65e451bb30d7c0fd342.jpeg)

Recursion enables having both small proofs with short proving times. This is possible by using a fast SNARK with a large proof size for long computations and then using that large proof as a witness to another slow SNARK with short proofs. Depending on the overhead of using two SNARKs, we can potentially have a fast combined SNARK with short proof sizes. This idea of recursion lies at the heart of Incrementally Verifiable Computation (IVC).

Schemes from recursive proof composition

As its name suggests, IVC allows us to verify a potentially long computation incrementally, in batches, without doing it all at once. Since 2008, researchers have proposed different variants of IVC:

- Full recursion
- Atomic accumulation
- Split accumulation
- Folding schemes

A useful organising framework of the different recursive techniques is the position at which the prover defers recursive verification. Folding schemes defer early expensive computation to the final verifier (also called decider). The prover in these schemes has fewer computations and a smaller recursive circuit. The techniques in the later stages only defer the instantiated polynomial oracles, and they are more flexible to batch multiple instances with different circuits.

[

Screenshot 2024-01-24 at 13.26.59

1488×574 30.2 KB

](https://europe1.discourse-cdn.com/standard20/uploads/anoma1/original/1X/dcff27a186ec1cfe7b08aeec229d207c22929508.png)

Folding schemes are recursive schemes that defer verification for instances without instantiating polynomial oracles. Accumulation schemes are recursive schemes that already instantiated oracles and batch them in their instances. This is an essential difference because running recursive circuits with non-native computation of commitments is expensive, and commitments are carried along in accumulation schemes.

For example, the recursive circuit representing the folding algorithm in HyperNova only computes one group operation when folding two instances, whereas, in BCLMS21, this accumulation is linear to the size of the instances.

As a side note, both “Special Sound Protocol” and “Relaxed Relation” are two different techniques to relax the constraints of the arithmetisations to accommodate folding. We’ll discuss arithmetisations later.

Full recursion

This is the first and most obvious approach to recursion, sketched byValiant in 2008. In this scheme, the full verifier algorithm of a SNARK is turned into a circuit and appended to the circuit that represents each step in the chain of computations. At every step i

, the proof π_i

asserts that all prior computations were verified. For this scheme to be practical, the underlying proof system must have a succint verifier, that is, sublinear in time complexity. We can find Groth16 or Plonk (with KZG) among these. However, a practical scheme cannot be achieved for long computation chains, even with a succint verifier for each iteration. Ideally, we want the prover to do linear work, with just a constant factor penalty over simply running the computation.

[

Screenshot 2024-01-15 at 21.20.19

1088×266 13.2 KB

](https://europe1.discourse-cdn.com/standard20/uploads/anoma1/original/1X/75eda7d27d67155cfd4f85ed28c6065273ab841c.png)

Atomic accumulation

In their paper Recursive Proof Composition without a Trusted Setup (also known as the Halo paper

), the ZCash team noticed that the verifier algorithm is composed by a fast and a slow subalgorithms, and that for certain type of SNARKs, the verification of the linear part can be accumulated in any IPA-based SNARK. The sublinear part of the verifier algorithm is still turned into a circuit and appended to each iteration of the recursive scheme.

Following this work, Proof-Carrying Data from Accumulation Schemes coined the term accumulation schemes

to describe this particular variant of IVC, andHalo Infinite: Proof-Carrying Data from Additively Polynomial Commitments generalised the Halo construction to any additively homomorphic polynomial commitment scheme.

A polynomial commitment scheme (PCS) allows the prover to convince a verifier that, given a commitment to a polynomial f

, a challenge point x

and an evaluation y

, we have that $f(x) = y$

. In this recursion scheme, whether it uses IPA or KZG as a PCS, the verifier accumulates the linear computation and performs the sublinear check. A PCS is additively homomorphic if can take a random linear combination of polynomials $\{f_i\}$

and their commitments $\{C_i\}$

separately and ensure with high probability that $C := C_1 + \alpha C_2 + \alpha^2 C_3 + \ldots + \alpha^n C_n$

is a commitment of $f := f_1 + \alpha f_2 + \alpha^2 f_3 + \ldots + \alpha^n f_n$

.

It is important to note that this accumulation technique cannot be applied to STARKs since its polynomial commitment scheme FRI is based on hashing, and hashes are not additively homomorphic.

Inner Product Argument (IPA)

In particular, in the inner product argument (IPA) of Halo, the linear computation is an inner product $C_i = \langle\mathbf{G}, \mathbf{s}\rangle = G_1 \cdot s_1 + ... + G_n \cdot s_n$

, where $\mathbf{s}$

is the round of challenges $\{ u_1, ..., u_k\}$

of that particular recursion step and $\mathbf{G}$

is a vector of random group elements G_i

publicly given at the beginning of the protocol. The verifier needs to compute $C_i=\langle\mathbf{G},\mathbf{s}\rangle$

, which is a linear-time multiscalar multiplication, and $b = \langle\mathbf{s},\mathbf{b}\rangle = g(x, u_1,...,u_k)$

, which can be computed by the verifier in logarithmic time. This latter one is the sublinear check.

Their key observation is that C_i

is a commitment. In the final step of this PCS, the verifier performs a random linear combination of the accumulated commitments, $C = C_1 + \alpha C_2 + ... + \alpha^m C_m$

and verifies the argument in $O(m \cdot log (d))$

. Since there is only one verifier check at the end, the cost is amortised.

[

Screenshot 2024-01-15 at 21.05.01

1098×300 39.9 KB

](https://europe1.discourse-cdn.com/standard20/uploads/anoma1/original/1X/6de017b975af6a5428810069ba7e1000de1ff806.png)

Kate, Zaverucha, Goldberg (KZG)

In the case of the polynomial commitment scheme known asKZG, the verifier performs two operations:

- Creates a pair of a polynomial and a commitment to it
- Checks the polynomial-commitment pair

The first part of creating a pair is fast (sublinear) and so extends the circuit F

in the accumulation scheme; the second part of checking the polynomial-commitment pair is slow since it involves pairings and is accumulated until the end of the scheme. As with IPA, accumulating the linear check is possible because this polynomial commitment scheme is additively homomorphic, and the cost is also amortised.

Split accumulation

In the paper Proof-Carrying Data without Succint Arguments (also known as BCLSM21

), Bünz et al. realised that the expensive succint property of a SNARK is unnecessary for building accumulation schemes, that is, the proof of each iterative computation doesn't need to be succint to get the succinctness property of the overall scheme. They proposed a scheme in which each iteration is simply a NARK, and run a single SNARK at the end of the scheme. This led to the so-called Split Accumulation

technique. In this scheme, not only the verifier does not verify each proof completely, but the prover doesn't prove each iteration completely either, and it doesn't create a SNARK, but a NARK.

Folding Schemes

A natural continuation of the previous trends came with the work ofNova: Recursive Zero-Knowledge Arguments from Folding Schemes, in which the sublinear work of the verifier is also deferred.

From here, an explosion of works emerged, extending this construction of Nova. Sangria extends this with the Plonkish arithmetisation and HyperNova with Customisable Constraint System (CCS). SuperNova extended Nova with a new technique called Non-uniform IVC (NIVC) and Protostar generalised the construction of Nova, introducing a generic folding scheme.

Initial constructions

The fundamental concept behind folding schemes is batch verification, which allows checking multiple proofs in a batch with almost the same cost as checking just one proof.

The argument of folding schemes goes as follows: given a set of sequential computations $F \to F \to ... \to F$

rather than computing a SNARK proof π_i

for each iteration, we fold them into a single instance F^*

for which we produce a single SNARK proof π

.

That is, instead of providing evidence that each step function F

is computed correctly, instances are “folded” into a compressed instance (i.e. an instance that encapsulates all previous instances), and the prover outputs a proof of correct folding. Practically, this only works because folding is much cheaper (constant size) compared to verifying a SNARK.

As with accumulation schemes, folding schemes also require an additively homomorphic PCS. Thus, STARKs cannot be folded either.

The main improvements of folding over other recursion or accumulation techniques are:

- The prover performs fewer Multi-Scalar Multiplications (MSMs) and, in some folding schemes, also avoids doing Fast Fourier Transforms (FFTs), which are generally a bottleneck.
- The circuit verifying the folding iteration has fewer MSMs and hashes, leading to fewer constraints.

Multi-folding

A folding scheme for a relation R

is a protocol between a prover and a verifier that reduces the task of checking two instances in R

with the same

structure S

into the task of checking a single folded instance in R

also with structure S

.

Introduced in the HyperNova paper, multi-folding is a generalisation of folding that allows the folding of two collections of instances in relations R_1

and R_2

with the same structure S

. In layman terms, this means that we can fold two different

circuits as long as they have the same

arithmetisation. In HyperNova, this arithmetisation is CCS.

The main idea of multi-folding, as introduced in HyperNova, is to fold a CCS instance into an augmented, more restricted variant of CCS (called linearised CCS) that is carried throughout the long-running computation. This instance is called a “running instance”. That is, $F_i : CCS \times CCS_{linearised} \to CCS_{linearised}$

. The running instance is denoted by U

and the CCS instance representing the last step in the computation is denoted by u

.

Recent work, extends multi-folding to allow folding two instances of different structures.

Non-uniform IVC

In IVC, the iterative function F

must always be the same for each iteration. In this case, this function is said to be uniform

or universal

. For most use cases of IVC, we may want F

to be different, or non-uniform

.

While it is possible to combine two or more different operations into one single constraint (e.g. by using selector polynomials), the cost of proving a program's step in IVC is proportional to the size of the universal circuit, even though the corresponding program step invokes only one of the supported instructions. For NIVC to be a meaningful notion, the prover's work at each step scales only on the size of the function executed at that step.

Given the high costs imposed by universal circuits, designers of these machines aim to employ a minimal instruction set, to keep the size of the universal circuit, and thereby, the cost of proving a program step minimal. However, this doesn't work in practice. For real applications, one must execute an enormous number of iterations of the minimal circuit (e.g., billions of iterations), making the prover's work largely untenable.

SuperNova introduced the Non-uniform IVC (NIVC) construction. The subtitle of this paper reads: "Proving universal machine executions without universal circuits". This means that, instead of one universal

circuit F

, we have a bunch of different step functions $\{F_{\phi(s_{i-1}, w_i)}\}$

parameterised by $\phi : \text{some-program-data} \to \{1,...,n\}$

where only one step function F_i

is chosen per iteration. The set of step functions $\{F_1,..., F_n\}$

can be understood as an instruction set in the context of ZKVMs.

In a traditional IVC setting, the function or circuit F

that we are iterating over must always be the same, even if it comprises multiple functions. A good analogy is a custom gate with selectors, and the cost is linear to the number of functions $\{F_1, ... F_l\}$

that compose F

. With NIVC, both the size of the circuit and the cost of computation are only linear to the particular $F_{\phi(s_{i-1}, w_i)}$

that gets executed. The per-step proving cost is independent of the sizes of circuits of "uninvoked" instructions.

Formally, for a specified $(\{F_1, . . . , F_l\}, \phi)$

and (n, s_0, s_n)

, the prover proves the knowledge of a set of non-deterministic values $\{\omega_0, . . . , \omega_{n-1}\}$

and $\{s_1, . . . , s_{n-1}\}$

such that $\forall i \in \{0, . . . , n-1\}$

, we have that $s_{i+1} = F_{\phi(s_i,\omega_i)}(s_i, \omega_i)$

.

This innovation renders many applications based on folding schemes computationally feasible, and in particular ZKVMs.

Compared to all the previous techniques, proving each iteration in NIVC can be optimised to be only a multiplicative factor slower than evaluating the iterated function.

Cycle of Curves

In IVC, (part of) the verification algorithm of the first SNARK (sometimes called "the inner SNARK") is embedded into the circuit of the second ("outer") SNARK. The proof of the inner SNARK becomes the witness of the outer SNARK. For practical reasons, this construction usually requires a cycle of curves (see this post for more details).

Thus, an IVC protocol is usually instantiated over a cycle of two curves.

[

Screenshot 2024-01-14 at 22.12.12

1008×288 29.7 KB

](https://europe1.discourse-cdn.com/standard20/uploads/anoma1/original/1X/2cc797ddf464d2bdc378b60403377cd7d2e61c7b.png)

Any recursion scheme will likely need to be implemented over a cycle of curves. This has consequences on which proving systems to use and which curves work. Works like CycleFold focus on which operations are optimised on which curve. For example, in their scheme, only group operations are performed in the outer curve and then compressed.

Conclusion

In 2008, Paul Valiant proposed the first IVC scheme. Research on IVC was mostly quiet until 2020. Since then, we've seen an explosion of IVC-related papers.

Scheme	Arithmetisation	Non-uniform	Multi-folding	ZK
Nova	R1CS	No	No	No
SuperNova	R1CS	Yes	No	No
HyperNova	CCS	No	Yes	No
BCLMS21	R1CS	Yes	Yes	Yes
Protostar	Plonkish/CCS	Yes	No	No
Protogalaxy	Plonkish/CCS			

Yes

Yes

No

While IVC supports machines with a single universal instruction, NIVC supports machines with an arbitrary instruction set.

Recursion and, particularly, folding schemes are radically changing how we design SNARKs by removing some artificial limitations of today's popular SNARKs. The importance of having an efficient recursive prover, even at the expense of a large proof size or a slow verifier (which can later be folded), has revamped proving systems such as Brakedown that were neglected.

Thus, folding encourages IVC-based ZKVM designers to aim for the fastest possible prover, even if this means obtaining only slightly sublinear size proofs or linear verifiers and then applying recursion to bring the proof size down. Properties such as zero-knowledge, non-uniformity of circuits, multi-folding and the arithmetisation used in a recursive scheme will all be important in designing a ZKVM.