zkVMs are great to prove large computations but they are not perfect. In particular, zkVMs don’t have knowledge of the program, so they include instructions that a particular program might never use and the instructions used may not be optimal; for any given program and backend, there is a most suitable set of instructions.

While it is common to fix a set of functions {F_1,..., F_n}

to a basic instruction set (like STARKish zkVMs do), it doesn’t have to be so. A compiler can use the information it gathers from a program to create tailored step functions or instruction set (IS) {F_i}

at compile time. These fundamental instructions no longer need to be small enough opcodes that satisfy all programs, but subsets of the compiled program.

Given some design constraints and using some heuristics, a compiler (acting as a front-end to a SNARK) may split a program into a set of subprograms and strike a balance between circuit size and number of circuits for a specific backend.

In recent zkVMs, this backend can either be a folding scheme, a fully recursive scheme or a giant lookup table, each providing very different properties and in turn affecting the design of the zkVM compiler.

Some of the main questions we want to answer are:

- What is the right approach of designing a zkVM?
- How different is the work of the compiler for each of these types of backends?
- How much do these schemes benefit from such a compiler?
- How will existing research potentially influence these zkVMs?

[

Screenshot 2024-01-29 at 19.55.31

2164×488 87.9 KB

](https://europe1.discourse-cdn.com/standard20/uploads/anoma1/original/1X/587d79fee81c79026edb4f9652431ff8bf370f7f.png)

After analysing the state-of-the-art works on IVC, arithmetisation, recursion schemes, finite field arithmetic, lookups, etc. we have identified three distinct, promising directions in future zkVMs: STARKish, NIVC and Jolt. In the sections below, we also hint the role of a compiler in such constructions.

Compiling to STARKish zkVMs

STARKish zkVMs use full recursion. Their polynomial commitment schemes are based on hash functions and error-correcting codes. Since these SNARKs are not based on elliptic curves, they are able to use smaller fields. This makes the prover much faster since the polynomial commitment scheme is generally the bottleneck for any SNARK and committing to small fields is more efficient. The work on towers of binary fields pioneered by Binius promises to improve dramatically the efficiency of these STARKish protocols. In a recent presentation, they run a benchmark for committing to a polynomial with 2^{28}

coefficients.

1-bit elements

8-bit elements

32-bit elements

64-bit elements

Hyrax BN254 (Lasso)

85.02 s

118.6 s

286.9 s

605.5 s

FRI, Poseidon (Plonky2)

98.718 s

98.718 s

98.817 s

98.817 s

FRI, Keccak (Plonky2)

29.36 s

29.36 s

29.36 s

29.36 s

Binius

0.586 s

5.173 s

29.36 s

58.62 s

Binius benefits from using towers of binary fields in the sense that 1-bit element operations are much cheaper than 64-bit operations. They can extend their binary field as their wish for certain operations. In contrast, Plonky2 uses a 64-bit prime field, so any element must be embedded into this field, no matter how small the element is. Elliptic-cuve-based SNARKs use a 256-bit so every operand must be embedded in this large prime field, no matter how small its value is.

The work of Binius in particular is redefining some of the (mis)-conceptions of SNARK friendliness, and traditional hash functions such as SHA-3 are now “SNARK-friendly”.

However, STARKish protocols can’t be folded, since the sum of their commitments is not homomorphic. Since a zkVM is an instance of IVC, any STARKish protocol is limited to full recursion. On one hand, they offer a much faster prover. On the other hand, they can’t accumulate or delay any verification.

A compiler such as the one described in this report can still improve greatly the performance of these zkVMs. Fewer but larger operations means that the overhead of having a full verifier on every operation (i.e. full recursion) can be made potentially negligible, relative to the cost of proving such operation.

Compiling to NIVC zkVMs

On the other hand, the polynomial commitment schemes of elliptic-curve-based SNARKs are generally additively homomorphic

and thus these SNARKs can be folded, i.e. Com(x) + Com(y) = Com(x+y)

.

Folding was impractical until SuperNova because the prover time of such a zkVM or IVC instance (which already is more expensive than the prover is a STARKish protocol) was linear to the number of instructions in its instruction set. SuperNova introduced Non-uniform IVC, which removes this limitation and renders such an IVC-based zkVM practical.

Elliptic-curve-based SNARKs allow for both full recursion and folding. KZG is the polynomial commitment scheme that outputs the smallest proof, but it requires a trusted setup. IPA, though less efficient (technically is not succinct

), removes this limitation. SNARKs that use HyperPlonk avoid FFT during proof generation, which brings down memory requirements and makes them more parallelisable.

Although the recursive overhead is much less for folding than for full recursion, even a small folding overhead. such as in Protostar (500 constraints) or in HyperNova (10.000 constraints), is expensive if the operation is as small as an addition.

A compiler that generates an ideal set of instructions for a NIVC zkVM renders this approach not only practical, but the most promising one.

However, some of these works are still merely theoretical. Works built solely on CCS such as HyperNova don’t have an implementation, which makes them hard to consider. Others, such as ProtoStar, are not only the state-of-the-art performance-wise, but are also the most generic and have already Plonkish implementations and adaptations to other

systems such as Halo2. In particular, ProtoStar is a generic folding scheme. This means that future works in arithmetisations, polynomial commitment schemes or even folding techniques promise to be easily adopted in ProtoStar with ease.

Compiling to Jolt zkVMs

(J)ust - (O)ne - (L)ookup -(T)able (i.e. Jolt) is simply a compiler that, given a set of decomposable

instructions, generates a zkVM that is solely based on lookups. A Jolt zkVM is just a giant lookup table. The proving system that performs these lookups is Lasso.

One of the costs in Lasso is proportional to the number of permutations of the lookup table. If the lookup table is 64-bits this would be 2^{64}

. This is too big. If instead we chunk that 4 times, this costs goes down to 2^{16}

. Which is entirely practical. The reduction is exponential.

If a compiler is able to provide a set of decomposable instructions to Jolt, then Jolt can do the remaining work and use Lasso for proving. Decomposable means that one lookup into the evaluation table t

of an instruction, which has size N

, can be answered with a small number of lookups into much smaller tables t_1, ... , t_l

, each of size N^{1/c}

.

What is promising about this approach is its simplicity. Performing lookups is conceptually much simpler than designing circuits by hand and arguably less error-prone and easier to audit. Because the prover's algorithm is simple (most of the prover's work is pushed to a lookup), R1CS turns out to be suitable for Lasso.

In the future, Binius PCS can and likely will work with Jolt and Lasso. There will be folding with Lasso like lookups possibly aggregating hyper efficient Binius proofs.

As of today, Lasso is stable, Jolt is in development.

Final words

2023 was a great year for multivariate, sum-check based zero-knowledge proofs. These innovations led to the works described in this article.

Pros

Cons

STARKish zkVMs

Performance of a single instance (fast prover). Maturity

Performance of the scheme as a whole (no folding)

NIVC zkVMs

Performance of the scheme as a whole (folding). Space efficient. Small proofs

Large fields

Jolt zkVMs

Conceptually simple. Easier to reason & audit. Benefits from small fields

Experimental

Arguably, the "STARKish" approach is not separate from Jolt. Jolt can use any multilinear polynomial commitment scheme.

Once the initial implementation of Jolt is done (using curve-based commitments) they will likely replace the commitment scheme with Binius-commitment, expecting at least a 5x speedup from that.

One of the main criticisms of IVC schemes is that doing everything incrementally is limiting. They require to break big computations into tiny chunks, handle each chunk separately and then efficiently combine the results. For example, lookup

arguments like Lasso have a non-trivial “fixed cost” that gets amortized over many lookups, and that kind of amortization is not possible unless one operates on a decent-sized chunk “all at once”. So IVC schemes currently use different, less efficient lookup arguments than Jolt.

On the other hand, zkVMs based on IVC are poised to be much more space efficient for the prover than Jolt (at least in the short-to-medium-term) because they can break big computations up into much smaller pieces than non-IVC schemes, without recursion overhead being a bottleneck.

As this report suggests, a compiler may overcome the IVC limitation of IVC schemes by generating an instruction set of bigger chunks, tailored to a given program. This will in turn proportionately increase the prover memory requirement. Since IVC naturally supports iterative computations (i.e., computing F(F(F(...(F(F(x))))))

for a function F

), for computations naturally expressed that way it’s hard to beat IVC. Thus a compiler may leverage this property to provide a circuit abstraction of pure functions, moving away from a fully-fledged VM abstraction.

Currently, there are some gaps to make the combination of Jolt and folding schemes fully work but we expect it will be resolved in the near future. There are intermediate design points between the high-developer-friendliness-but-high-overhead of a VM abstraction and hand-optimising a circuit for a particular particular step function F

.

So, given the current state of research and engineering, NIVC seems to be the most promising approach to building a zkVM. By avoiding full recursion and uniform circuits, proving each iteration in NIVC can be optimised to being only a multiplicative factor slower than actually evaluating the iterated function. This is a great advantage over the other approaches. Specially for long computations, folding offers a unique advantage and the final proof will also be small.

Existing efforts to integrate ProtoStar in Halo2 makes this NIVC approach more tractable engineering-wise for systems that are already implemented in Halo2, such as Taiga. Elliptic-curve-based IVC can also apply full recursion for function privacy (as Taiga requires).

A continuation to this work would be to determine exactly what work a compiler needs to do in order to optimise the chosen type of zkVM. For example, blocks in STARKish zkVMs are likely to be larger than in NIVC zkVMs since full recursion is more expensive than folding. As mentioned, a compiler may want to have a purely functional abstraction to leverage the properties of NIVC. In contrast, Jolt zkVMs requires decomposable blocks, independently of the size, since they will compose a giant table in the end. A simple heuristic such as a fixed number of constraints would be a good starting point.