This is a continuation to the postZKVM Compilers: Motivation. I reached the maximum number of words in that post, so I'm splitting it.

Compiler design goals

When designing an end-to-end compiler for ZKVMs, these are our desiderata:

- \textbf{Smart Block Generation}

: We want a compiler that decomposes any program into a set of circuit blocks that can be proven efficiently, using folding schemes or otherwise. The Instruction Set (IS) of the generated ZKVM consists of these circuit blocks.

- \textbf{Fast Provers, Small Proofs, Fast Verifiers}

: We want short proof generation time and fast verification, both in terms of asymptotic performance and overheads. As we'll see, the advent of folding schemes and other works have reframed some assumptions of existing SNARKs, where slow verifiers and large proofs can be overcome with folding.

- \textbf{Modularity}

: We want to reason about a specific proving system without tying ourselves to a concrete Intermediate Representation (IR), specific arithmetisation or fixed finite field. Most novel proving systems such as ProtoStar are agnostic to their arithmetisation, and others such as Jolt and Lasso also allow us to be flexible with the field of choice.

- \textbf{Function privacy}

: We want a proving system that proves that some program is satisfied, without knowing which program, by only publishing its commitment. This is not strictly a property of the compiler but of an operating system such as Taiga.

We now survey the work that has been done in relation to these goals:

Smart Block Generation

Circuits as lookup tables (Jolt and Lasso)

Jolt (Just One Lookup Table) is an exciting theoretical innovation in the ZKVM space, despite itnot having a stable open-source implementation yet. It springs from the realisation that the key property that makes an instruction "SNARK-friendly" is \textit{decomposability}

, that is, that an instruction can be evaluated on a given pair of inputs (x, y)

by breaking x

and y

up into smaller chunks by evaluating first a small number of functions on each chunk and then combining the results.

They claim that the other ZKVMs are wrongly designed from focusing on artificial limitations of existing SNARKs . That is, all other proving systems have been hand-designing VMs to be friendly to the limitations of today's popular SNARKs, but these assumed limitations are not real.

Jolt eliminates the need to hand-design instruction sets for ZKVMs or to hand-optimise circuits implementing those instruction sets because it replaces those circuits with a simple evaluation table of each primitive instruction

. This modular and generic architecture makes it easier to swap out fields and polynomial commitment schemes and implement recursion, and generally reduces the surface area for bugs and the amount of code that needs to be maintained and audited.

Jolt's companion work and backend, Lasso, is a new family of sum-check-based lookup arguments that supports gigantic (decomposable) tables. As we mentioned, Jolt is a ZKVM technique that avoids the complexity of implementing each instruction's logic with a tailored circuit. Instead, primitive instructions are implemented via one lookup into the entire evaluation table of the instruction. The key contribution of Jolt is to design these enormous tables with a certain structure that allows for efficient lookup arguments using Lasso. Lasso differs from other lookup arguments in that it explicitly exploits the cheapness of committing to small-valued elements.

In their paper, Jolt demonstrates that all operations in a complex instruction set such as Risc-V are decomposable, thus efficiently convertible into lookup tables. So, already in their paper, Jolt and Lasso theoretically provide a better alternative to existing STARK ZKVMs. With a MSM-based polynomial commitment, Jolt prover costs are roughly that of committing to 6 arbitrary field elements (256-bit) per CPU step. This compares to RISC-Zero, where prover costs are roughly equivalent to 35 (256-bit) field elements per CPU step, and Cairo VM, where prover commits to over 50 field elements per step of the

Cairo CPU.

If a compiler wants to generate a suitable instruction set for a program to be proved in Lasso, it must ensure that all generated circuit blocks are decomposable. And these blocks or instructions must be lookup tables.

[

Screenshot 2024-02-06 at 21.22.51

1916×482 71.8 KB

](https://europe1.discourse-cdn.com/standard20/uploads/anoma1/original/1X/60a66127e0a4bebe6944d137dea639d2f40bcce0.png)

The line between such compiler and Jolt is blurry. Jolt is already a frontend that converts computer programs into a lower-level representation and then uses Lasso to generate a SNARK for circuit-satisfiability. In their paper, they adapt a program to a given set of decomposable instructions, and they prove it a la STARK ZKVMs. A compiler can leverage the understanding of decomposable functions and replace Jolt in some sense, while using the techniques in Jolt to improve the overall performance.

Note that unlike STARKs implementations, Lasso's is not yet optimised. So a detailed experimental comparison of Jolt to existing ZKVMs will have to wait until a full implementation is complete.

(Optional). Below is a summary of the main ideas of this work:

- \textit{Lookup singularity}

. Jolt uses lookups instead of tailored circuits. It can represent any opcode or instruction in a separate table, and these tables can be gigantic. That is, for each instruction, the table stores the entire evaluation table of the instruction. Then it applies Lasso to prove these lookups. The key insight here is that most instructions can be computed by composing lookups from a handful of small tables. For example, if an instruction operates on a 64-bit input (and a 64-bit output), the table representing this function has size 2^{128}

.

- \textit{Decomposability}

. The purpose of Jolt is to design gigantic tables with a certain structure that allows for efficient lookup arguments using Lasso. Decomposability roughly means that a single lookup into the table can be answered by performing a handful of lookups into much smaller tables. One lookup into an evaluation table, which has size N

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

, each of size $N^{1/c}$

. For most instructions, l

will equal one or two, and about c

lookups are performed into each table.

- \textit{LDE-structure}

. LDE is short for a technical notion called a Low-Degree Extension polynomial. A Multi Linear Extension (MLE) is a specific low-degree extension of a polynomial. For each of the small tables, the multilinear extension can be evaluated at any point, using just $O(\log(N)/c)$

field operations.

If the table is structured, no party needs to commit to all of the values in the table. This contrasts to other lookup arguments, since Lasso pays only for the table elements it actually accesses

.

- \textit{Sum-check protocol}

. The cost of committing to data is a key bottleneck for SNARK prover performance. The sum-check protocol minimises the amount of data (and size of each piece of data) that the prover has to commit to. Lasso is a lookup argument where the prover commits to few and small values, compared to other proving systems. That is, all committed values are small.

- \textit{Auditing}

. Jolt encourages ZKVM designers to simply specify the evaluation table of each instruction. It's easier to audit lookup arguments than many lines of hand-coded constraints.

- \textit{Offline memory checking}

. Lasso's core contribution is, arguably, its single-table lookup procedure: a virtual polynomial protocol which uses offline memory-checking. Any ZKVM has to perform memory-checking. This means that the prover must be able to commit to an execution trace for the VM (that is, a step-by-step record of what the VM did over the course of its execution), and the verifier has to find a way to confirm that the prover maintained memory correctly throughout the entire execution trace. In other words, the value purportedly returned by any read operation in the execution trace must equal the value most recently written to the appropriate memory cell.

Fast Provers, Small Proofs, Fast Verifiers

Small fields (Plonky2)

STARKs pioneered in 2018 an alternative design of proving systems, based on linear error-correcting codes and collision-resistant hash functions, and characterised by the use of smaller fields (specifically of 64-bit-sized prime fields instead of the usual 128, 192 or 256 bits). They were able to use a smaller field because the Polynomial Commitment Scheme (PCS) they use, FRI, does not require a large-characteristic field (in fact, FRI was originally designed to work over towers of binary fields).

Plonky2 claims to achieve the performance benefits of both SNARKs and STARKs, although the difference between Succinct Non-interactive ARgument of Knowledge (SNARKs) and Scalable Transparent ARgument of Knowledge (STARKs) is blurry. Most modern SNARKs do not require trusted setup and their proving time is quasilinear (i.e. linear up to logarithmic factors), so they are \textit{Scalable}

and \textit{Transparent}

. On the other hand, STARKs today are deployed \textit{Non-interactively}

and thus they are SNARKs.

Instead, we define a STARK as the specific construction from the STARKWARE team. We define a STARKish protocol as a SNARK from linear codes and hash functions, in contrast to elliptic-curve-based SNARKs. FRI-based or Brakedown-based SNARKs are STARKish protocols. Thus Plonky2 is a STARKish protocol.

Elliptic curve SNARKs

STARKish protocols

Smaller proofs (~1 Kb)

Larger proofs (~100 Kb)

Fast verifier

Slower verifier

Slow prover

Faster prover

Aggregation & Folding friendly

Native-field full recursion friendly

Big fields (256 bits)

Small fields support (32 bits)

Compute-bound

Bandwitdth-bound

The main mathematical idea behind using arithmetic over smaller fields in a SNARK is field extensions or towers of fields. That is, using smaller fields operations while employing their field extensions when necessary (e.g. for elliptic curve operations) promises to improve performance and maintain security assumptions.

However, Research suggests that elliptic curves based on extension fields are likely suffer from specific attacks that do not apply to common elliptic curves constructed over large prime fields.

STARKish protocols leverage the relative efficiency of small-field arithmetic, and achieve state-of-the-art proving performance for naive proving, mainly because collision resistant hash functions are much faster than elliptic curve primitives.

In other words, for any sort of uniform computation, the prover’s time of STARKish protocols seems unbeatable by elliptic curve SNARKs. The benefits of elliptic curve SNARKs at this time are mainly their smaller proofs and efficient verifiers. Folding schemes seem like they can bridge this gap in prover performance for long computations, as we’ve seen above.

Plonky2 and its successor Plonky3 can be seen as implementations of PLONK with FRI. That is, they take the Interactive Oracle Proof (IOP) from PLONK and mix it with the FRI Polynomial Commitment Scheme (PCS) to construct their SNARK.

The polynomial commitment scheme FRI has a very fast prover and this allows Plonky to play with the spectrum between fast proving and large proofs and slow proving and small proofs. Plonky2 use smaller field than in other elliptic-curve-based SNARKs, called the Goldilocks field, which is of size 2^{64}

, making native arithmetic in 64-bit CPUs efficient. Plonky3 utilises an even smaller prime field F_p

called Mersenne31, where $p = 2^{31} - 1$

, thus suitable for 32-bit CPUs (it fits within a 32-bit word).

Despite being one of the most performant implementations of a SNARK, it is unclear how Plonky3 will benefit from folding schemes, since FRI is not an additively homomorphic PCS.

Research on combining operations in small fields with other operations in their extension fields for SNARK protocols is currently very active.

In summary, Plonky3 promises a very performant STARKish ZKVM based on full recursion. Thus it is not folding friendly. While still in progress, the Valida ZKVM is an implementation of a Plonky3 ZKVM.

Smallest Fields (Binius)

Plonky2

Plonky3

Binius

$p = 2^{64} - 2^{32} + 1$

(Goldilocks)

$p = 2^{31} - 1$

(Mersenne31)

$p = 2$

(Binary)

As we’ve seen, one of the main disadvantages of SNARKs over elliptic curves compared to STARKish protocols is that elliptic-curve-SNARKs require a bigger field in their circuits, which affects negatively the performance of their provers.

In the case of a SNARK, an element of their field of choice generally decomposes into 256 bits, whereas STARKish protocols leverage the fact that the characteristic of a field $\mathbb{F}_p$

is equal to the characteristic of any of its extension fields $\mathbb{F}_{p^n}$

, allowing to have small overheads for certain operations and then using extension fields to achieve the desired cryptographic security. The most widely used field in STARKs is $\mathbb{F}_p$

where $p=2^{64} - 2^{32} + 1$

. This field is called the Goldilocks field. Among other properties, every element in this field fits in 64 bits, allowing for more efficient arithmetic on CPUs working on 64-bit integers

The question that Binius raised and addressed was: “what is the optimal field to use in any arithmetisation?”. The obvious answer is binary fields, since arithmetic circuits are essentially additions and multiplications, and these operations over binary fields are ideal. They propose using towers of binary fields to overcome the overhead of embedding $\mathbb{F}_2 \hookrightarrow \mathbb{F}_p$

that are a waste of resources specially in SNARKs (compared to STARKs), since they require 256-bit prime fields and many gates take 0 or 1 values.

They remark that the FRI polynomial commitment scheme that lies at the heart of STARKs was designed to work over binary fields. They apply techniques from other works such as Lasso and Hyperplonk. In particular, they leverage thesum-check protocol and “small” values protocols, where the prover commits only to small values. They revive the polynomial commitment scheme Brakedown, which was mainly discarded because of its slow verifier and the large proofs it produces, despite having an incredibly efficient prover O(N)

. Their SNARK, based on HyperPlonk, makes Plonkish constraint systems a natural target.

The main consequences of using towers of binary fields are:

- Efficient bitwise operations like XOR or logical shifts, which are heavily used in symmetric cryptography primitives like SHA-256. This turns “SNARK-unfriendly” operations into friendly.
- Small memory usage from working with small fields.
- Hardware-friendly implementations. This means they can fit more arithmetic and hashing accelerators on the same silicon area, and run them at a higher clock frequency.

This work on towers on binary fields advocate hash-based polynomial commitment schemes such as FRI or Brakedown because they allow using smaller fields, which in turn reduces storage requirements and more efficient CPU operations, flexibility of fields that enables modular reduction, and cheaper cryptographic primitives (hash functions are faster than elliptic curve primitives).

In particular, Binius adapts HyperPlonk to the multivariate setting and is not fixed to a single finite field. They partition the representative Plonkish trace matrix into columns, each corresponding to different subfields in the tower (e.g. some columns will be defined over \mathbb{F}_2

, others over \mathbb{F_{2^8}}

, etc.), and the gate constraints may express polynomial relations defined over particular subfields of the tower.

In summary, like Plonky3, Binius is also a STARKish protocol, thus not folding friendly. It has re-configured the way we understand SNARKs performance and we are likely to see significant improvements in many schemes from applying the techniques stated in this work, as they did on hash functions (they yield faster SNARKs for these hash functions than anything previously done). For example, this work has already improved the performance of other works such as Lasso

Large fields, but non-uniform folding (SuperNova, HyperNova, ProtoStar)

Despite the benefits of using small fields, the commitment schemes used in STARKish protocols are not additively homomorphic. In contrast, elliptic-curves-based polynomial commitment schemes such as KZG or IPA are additively homomorphic and thus folding is possible. Not only that, but the work on Non-Uniform Incrementally Verifiable Computation (NIVC) introduced by SuperNova renders these folding schemes as a promising alternative to construct zkVMs.

[

Screenshot 2024-01-25 at 17.51.42

838×396 29.1 KB

](https://europe1.discourse-cdn.com/standard20/uploads/anoma1/original/1X/982cace950aa497358b5d8a6961f528a6e5be89f.png)

As we’ve seen, NIVC enables us to select any specific “instruction” (or generated block) F_i

at runtime without having a circuit whose computation is linear in the entire instruction set. NIVC reduces the cost of recursion from O(N\cdot C \cdot L)

to O(N(C+L))

, where N

is the number of instructions actually called in a given program, C

is the number of constraints or size of the circuit (upper bound) and L

is the number of sub-circuits or size of the instruction set { F_1, ..., F_L }

. Generally, the size of the circuit C

is much bigger than L

, so effectively the number of sub-circuits or instructions { F_1, ..., F_L } do not come at any cost to the prover.

Modularity

Generic accumulation (Protostar)

ProtoStar is a folding scheme built with a generic accumulation compiler. In their paper, they show the performance of an instance of this protocol that uses Plonk as a backend. As ProtosStar was conceived, the work of Customisable Constraint Systems (CCS), providing an alternative, more generic arithmetisation capable of expressing high-degree gates. In an appendix, ProtoStar took the opportunity to show how their general compiler can adopt a different arithmesation such as CCS while remaining the most efficient folding scheme to date.

So, modularity means that each step in the workflow below for building an IVC can be implemented in different ways, that is, one could change any component, from the arithmetisation to the commitment scheme in isolation, as long as they preserve certain properties.

For example, the commitment scheme in this recipe requires the commitment function to be additively homomorphic. As we've seen above, this renders the works around STARKish protocols not directly applicable here.

[

Screenshot 2024-01-25 at 17.50.29

852×306 22.7 KB

](https://europe1.discourse-cdn.com/standard20/uploads/anoma1/original/1X/05e87c0c11567b7274ef4b0598d3cccdc261bbb5.png)

The diagram above can be read as follows:

- We start from a special-sound protocol \Pi_{sps}

for a relation \mathcal{R}

. A special-sound protocol is a simple type of interactive protocol where the verifier checks that all of equations evaluate to 0. The inputs to these equations are the public inputs, the prover's messages and the verifier's random challenge.

- Then, we transform \Pi_{sps}

into a compressed verification version of it (CV[\Pi_{sps}]

), i.e. a special-sound protocol for the same relation \mathcal{R}

that compresses the l

degree-d

equations checked by the verifier into a single degree-(d+2

) equation using random linear combinations and 2\sqrt{l}

degree-2

equations.

- We construct a commit-and-open scheme from this sound-protocol CV[\Pi_{sps}]

that renders another special-sound protocol \Pi_{cm}

for the same given relation.

- A special-sound protocol is an interactive protocol. Thus we can apply the Fiat-Shamir transform to make it non-interactive, and FS[\Pi_{cm}]

becomes a NARK.

- Next, we accumulate the verification predicate V_{sps}

of the NARK scheme FS[\Pi_{cm}]

and so we have an accumulation scheme acc[FS[\Pi_{cm}]]

.

- From a given accumulation scheme acc[FS[\Pi_{cm}]]

, there exists an efficient transformation that outputs an IVC scheme, assuming that the circuit complexity of the accumulation verifier V_{acc}

is sub-linear in its inputs.

Function Privacy

Universal Circuits + Full Recursion (Taiga, Zexe, VeriZexe)

Function privacy is a property of some execution environments such asTaiga or Zexe of hiding which function or application is called at any given time from some parties; function privacy is always defined with respect to a specific verifier. This verifier does not learn anything about a particular function, other than that (a) it's commitment (hiding + binding) and (b) that the function was satisfied over some inputs.

In broad terms, one of the main ideas behind achieving function privacy consists of fixing a single universal function that takes user-defined functions as inputs. The other main idea behind function privacy is using a fully recursive scheme to hide a function, or in particular a verifying key of a circuit, via a commitment to that verifying key.

The architecture of Taiga comprises three circuits:

- Resource logic circuits: They represent the application-specific circuits, that is, the encoding of an application into a circuit

- Verifier circuit: An instance of [full recursion

](https://research.anoma.net/t/schemes-for-recursive-proof-composition/440), this circuit encodes the verifier algorithm and represents the recursive step that achieves function privacy. It takes both the proof \pi

and the verifying key vk

or a resource logic as private inputs, and the commitment of this verifying key Cm(vk)

as a public input. On the one hand, it verifies that the resource logic proof is correct. On the other hand, it hashes the verifying key and checks that the commitment to the verifying key is correctly computed.

It outputs another proof \pi'

that states that "I know of a valid proof of a resource logic whose verifying key is hidden under this commitment Cm

".

So, the resource that gets included in a transaction doesn't contain the verifying key of the resource logic, but its commitment and a proof that the logic is satisfied, hence achieving function privacy.

[

Screenshot 2024-01-18 at 21.20.47

936×232 13.5 KB

](https://europe1.discourse-cdn.com/standard20/uploads/anoma1/original/1X/05c1e37670b491884814fe3e9368a2b2b18ae7e0.png)

It's very important to note that the verifier circuit is run by the prover

to blind the verifying key vk

.

- Compliance circuit: This circuit proves that some resources are created and some are consumed correctly in a transaction (i.e. the proposed state transitions follow the Taiga rules) with regards to the commitment of the verifying key Cm(vk)

of a certain resource logic.

Both verifier and compliance circuit are fixed (i.e. they do not depend on resource logics), and the resource logic commitments are both blinding and binding, so we do not reveal any information about the resource logics involved, and achieve function privacy.

Next: Conclusion