What is a zkVM?

A zkVM is generally defined as an emulation of a CPU architecture

into a universal

circuit that allows to compute

and prove

any program from a given set of opcodes

.

One of the main goals when designing a zkVM is to maximise prover's performance. Surprisingly, not all zkVMs are designed to maximise this.

Why zkVMs?

Whether circuits are written manually, through libraries such as arkworks or through DSLs such as Noir, writing circuits is hard, meaning that it requires expertise and errors can be easily made.

An arguably lesser issue is that different circuits require different verifiers, thus involving a preprocessing step per circuit.

The main raison d'être of zkVMs is that of giving a nice developer experience without any exposure to cryptography. Any program can be compiled to a defined set of opcodes. Since this set is obviously finite and generally not very large, it can easily be audited.

Developers can thus leverage an arsenal of tooling and compiler infrastructure. One single circuit can suffice for running all programs up to a certain bound and only a single verifier is needed, since the zkVM circuit is universal.

However, zkVMs as they currently exist are not the holy grail. They have a much poorer performance compared to the manual approach due to the extra overhead of adding abstractions such as a memory or a stack, or embedding a set of instructions in this universal circuit.

Implementing certain important operations in a zkVM is extremely expensive. For example, a SHA-256 circuit is 1000 times faster if it is proven outside the abstraction of a zkVM.

Last and least, and despite their longer history, compiling a high-level program into assembly may incur into some bugs.

If programs directly compiled to circuits suffer principally from memory use and proving time, this problem is only accentuated with the use of zkVMs.

Why this title?

If a zkVM is defined as a universal circuit that encodes a set of instructions, what happens if we modify this set of instructions, leaving everything else unchanged? Do we still have the same zkVM or do we have a different one?

Similarly, since a main goal of a zkVM is maximising the efficiency of the prover, and different proving systems benefit from different designs, does a backend determine a zkVM? For example, how does a logarithmic time verifier of a particular proving system affect the design of a zkVM? We can pose a similar questions around memory checking.

These questions may be too technically nuanced, but it is also here where we enter the realm of compilers. There is a long history of compilers where optimisations are made depending of the structure of the program. For example, if a program has a repeated structure, a compiler may be able to optimise it.

This awareness

of the structure of a program can be also applied to provers, as we will see.

What has been done?

Generally, a zkVM is comprised of five phases:

- Compilation

of a program into a set of instructions

- Execution

and generation of the execution trace

- Proving
- (Optional) Compression

. Not all proofs are large

- (Optional) Zero Knowledge

. Not all zkVMs require zero-knowledge

The proving step is the one determining the different categories in which zkVMs are divided.

STARKish zkVMs

These zkVMs apply a SNARK to the constraint system to generate a proof and only leverage proof recursion to reduce the proof size and verification costs.

In particular, they use a STARKish protocol, that is a protocol whose commitment scheme is based on hashes and linear error-correction codes, instead of multi-scalar multiplications (MSMs) and homomorphic commitments as elliptic curve SNARKs do.

Despite their poor asymptotic performance (e.g. their prover time is super-linear and verifier time is logarithmic), the concrete efficiency of STARKish protocols currently surpasses that of elliptic curve SNARKs. This is because hashes are fast compared to MSM and they use smaller fields, sometimes at the expense of their security level.

Even after the continuous engineering work of the past years and the most recent breakthroughs, STARKish zkVMs such as the Cairo zkVM or STARKNET still need to delegate the prover computation to machines with large memory and high computation power, and are only able to prove practically small programs.

IVC zkVMs

An alternative approach for proving large statements while reducing memory costs consists of chunking the statement itself, compared to only chunking the prover's computation as STARKish zkVMs do. This approach requires each piece to be proven separately, store intermediate results and combine the proofs somehow into a single, final one.

The naive approach of instantiating an IVC zkVM involves embedding a verifier circuit into each chunk and proving each chunk in a sequential manner. Each chunk in this sequence verifies all prior computation. This approach is called "full recursion".

Usually, a chunk in this model is an opcode from the zkVM instruction set, or more specifically, all the opcodes with a pointer to the right one. Since an opcode is often just a few constraints, the verifier algorithm may surpass the number of constraints of an opcode by many orders of magnitude, rendering this construction highly impractical.

[

Screenshot 2024-04-02 at 15.11.16

986×202 22.8 KB

](https://europe1.discourse-cdn.com/standard20/uploads/anoma1/original/1X/b6ca610febc0a96c8e3048bf57aa5a7b2f40e1b9.png)

While full recursion is potentially "non-uniform" (meaning that each step in the IVC model can be different), this would also mean that a different verifier circuit is required at each step.

A more promising approach involves minimising the work of the verifier in this IVC construction by delegating the verification of previous statements to a later stage, called decider, and simply verify on each step that the "folding" is done correctly. This IVC instantiation is known as a "folding scheme".

Arguably, the main breakthrough in IVC-based zkVMs was done bySuperNova, where they achieved "non-uniform" folding, meaning that the time to prove each iteration of the IVC scheme does no longer depend on the size of the instruction set. They achieve it by carrying an instance of each instruction in their accumulator.

A question that may arise is, do we need a zkVM for folding? Not necessarily, but we need at least a way to prove that the gluing of the chunks was done correctly, that is, that the chunks in this IVC sequence corresponds to the original NP statement. For example, the stack in a zkVM connects a value between two opcodes. As we'll see, there are different systems that allow us to prove a program using a folding scheme.

What can be done?

One of the main disadvantages of the IVC-based scheme sketched above is that computation is sequential. Proof-Carrying Data (PCD) is a generalisation of IVC that enables parallel proving by structuring computation as a tree, where the proof of each node is only dependent on its children.

[

Screenshot 2024-04-02 at 15.24.41

1272×414 21.5 KB

](https://europe1.discourse-cdn.com/standard20/uploads/anoma1/original/1X/60754716dcda720e9650a15f071cb1938e9cb23c.png)

Notice that before proving a program, this must be executed in full, to generate its witness or trace. Then, knowing which opcodes were used and their inputs and outputs, the prover generates a proof for correct execution.

However, the above schemes are fixed

, or program agnostic

; they don't take into account the structure of a program or the information encoded in the trace before proving it.

Data-parallel circuits

are circuits that contain a repetitive pattern, or identical copies of smaller sub-circuits. These are also referred as "single instruction, multiple data" (SIMD) computations.

The GKR protocol is particularly suited to leverage data parallelism. It consists of d

sumcheck protocols for a layered circuit of depth d

, each layer linked via a chain of reductions. The first and last layers are dedicated to the circuit outputs and inputs.

Compared to some of the previous approaches, one of the many advantages of the GKR protocol is that the prover doesn't need to commit to intermediate data (i.e. the trace or witness), but only to the inputs and outputs of the circuit. The prover runs in linear time, since it consists on reductions via the sumcheck protocol.

So, given a data-parallel layered circuit, the GKR proving time is linear in the size of the circuit and it can be reduced by a factor of M

if the proving is distributed over M

machines.

GKR zkVMs

One of the main ideas behind the GKR zkVM (as introduced in the paperParallel Zero-Knowledge Virtual Machines) is identifying repeated opcodes in a program (patterns), grouping them together and proving them in batches using the GKR protocol.

The GKR zkVM consists of two main circuits:

1. Opcode circuit

: A data-parallel circuit that proves correct execution of each opcode. They use a global state to track the state transitions.

1. Chip circuit

: A circuit consisting on mainly set equality checks (i.e. permutation arguments) and lookup arguments that proves that the opcode circuits are executed in the correct order and global states are updated correctly.

[

Screenshot 2024-04-02 at 16.32.35

1464×642 22.9 KB

](https://europe1.discourse-cdn.com/standard20/uploads/anoma1/original/1X/cdb0943035c80ca6a092cbc9d83a68fd22e10d26.png)

One of the advantages of this zkVM is that the prover is dynamic

; it adapts to the execution trace. In other words, the number of opcodes in each group varies depending on the program.

Although the branching overhead in sequential branching is addressed in recent folding schemes such as SuperNova with their application of non-uniformity

, the step function still needs to assert the type of opcode used. In contrast, with this two-step

proving system (i.e. first proving opcodes, then “gluing” them in the chip circuit), GKR zkVMs avoid such overheads when proving an opcode, since opcodes within the same group are of the same type and they can be proven in parallel. This renders an extremely fast prover.

However, the prover is no longer creating a proof for a universal circuit, since each program is now a different circuit. As mentioned, one of the biggest advantages of a zkVM is having a single verifier for all programs.

A workaround proposed consists of having a smaller zkVM as a second layer that verifies

these non-universal proofs (i.e. the proofs corresponding to different programs) and outputs a universal one. The intuition behind this seemingly incredibly expensive approach is that the verifier runs logarithmically on the size of the circuit.

While the GKR zkVM design managed to leverage the structure of a program to provide a faster prover, it grouped together basic instructions from the instruction set to prove in parallel. These shallow circuits don’t take full advantage of the GKR protocol, in which the prover only needs to commit to the inputs. In a shallow circuit, the ratio between inputs and total number of gates is large.

A compiler may find more complex patterns in a program to parallelise.

GKR zkVM Pro*

The main idea behind this variant of the GKR zkVM is the distinction between basic blocks

and opcodes. A basic block

is a chunk of a program that doesn’t contain branches, so opcodes are always executed sequentially. Branches are what makes circuits dynamic. Within a basic block, the stack will behave identically for different programs.

[

Screenshot 2024-04-02 at 16.34.26

1578×654 97.3 KB

](https://europe1.discourse-cdn.com/standard20/uploads/anoma1/original/1X/19bc7b0154ff86bcb9544f7cf97d216c2af798da.png)

If a program contains multiple identical basic blocks, these can be seen as data-parallel

circuits and proved in parallel, as in the case of a loop.

Compared to the previous GKR zkVM design, opcodes are now grouped in basic blocks, instead of by type. The conceptual division between opcode circuits

and chip circuits

remains the same.

To summarise, these are the steps that take place in the GKR zkVM Pro:

- Compile program into assembly
- Identify basic blocks
- Prove each block separately
- Prove the chip circuit

This basic block abstraction enables removing some control overhead at the opcode level within, such as handling the stack, checking the timestamp or handling global states.

Mangrove SNARKs

The main idea of Mangrove is compiling

a program into identical simple steps

and applying a tree-based PCD

construction. This approach disregards the instruction set abstraction and is not considered a zkVM.

As with the previous case, Mangrove's compiler also uses the patterns in a program to produce chunks to fold. In this case, these chunks are identical or uniform

. Thus, Mangrove's scheme does not require a universal circuit. By creating identical chunks, these become data-parallel circuits

and can be proven in parallel.

However, the uniform circuit they produce for folding incurs into some overhead as it must be big enough to accommodate the different computations that the chunk generalises. To avoid the high overhead in the form of constraints for opening commitments they use a commit-and-fold

optimisation. In short, they introduce a generalised foldable relation that supports proving over committed values without encoding the commitment opening constraints.

One advantage of this approach, compared to the dynamic

prover of the GKR zkVM, is that the verifier is naturally fixed

(since chunks are identical).

Another significant innovation of Mangrove's tree-based folding construction (PCD) is the decoupling

of the core leaf computation from the recursive control merging computation, removing the unnecessary work on the leaves (i.e. the merging logic). Notice that when the arity of the tree is high, the majority of the work is performed at the leaves.

[

Screenshot 2024-04-02 at 16.36.56

1302×448 77.5 KB

](https://europe1.discourse-cdn.com/standard20/uploads/anoma1/original/1X/bc64ebc730c7a91eef113646ce1c3528250c4050.png)

Using different techniques to the GKR zkVMs, Mangrove SNARKs are also well-suited for both streaming

(memory efficiency) and distributed computing

(parallelism efficiency). The proving time of their construction is comparable to leading monolithic SNARKs, while being able to prove much larger programs with much fewer resources.

Although Mangrove's SNARK is also IVC-based, it is not a zkVM, since it is not a universal circuit and does not require a set of instructions, in contrast to the IVC zkVMs described above. Their uniform

compiler generates one single instruction that is folded. One would expect future work in which a non-uniform

compiler uses non-uniform

IVC techniques to improve the performance of the scheme.

IVC zkVMs (Nexus zkVM)

The Nexus zkVM is a simple, minimal, extensible and parallelisable folding-based zkVM, so it also realises PCD. It is simple in its architecture, memory model and I/O model. It is minimal in the sense that its instruction set, which is defined in the setup phase, can contain as many instructions as desired and it is empty by default. This setup phase can be seen as a compiler to zkVMs.

It is extensible

in the sense that its fixed instruction set can be extended with custom instructions, or co-processors

. It decouples instructions (inside the CPU abstraction) and user-defined co-processors (outside the CPU abstraction) without affecting per-cycle prover performance and minimising the size of the zkVM circuit being proven. Leveraging the non-uniform

techniques introduced in Supernova, the prover only pays for those instructions when they are actually executed.

Jolt zkVMs

Jolt is a compiler that takes a program and generates giant structured matrices consisting of all the evaluations of the different opcodes used in the zkVM abstraction. These structured matrices are called decomposable

, and allows the circuits produced by Jolt to only

perform lookups to these lookup tables that never materialise in full. Jolt produces a universal

circuit from combining all the opcode evaluation tables into one.

The main techniques they use to avoid the materialisation of these gigantic lookup tables are multi-linear extensions and the sumcheck protocol. Jolt decomposes

the lookup computations into chunks and glues the results together, achieving surprising results. 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}

.

For example, 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.

[

Screenshot 2024-04-02 at 16.45.18

1658×204 13.7 KB

](https://europe1.discourse-cdn.com/standard20/uploads/anoma1/original/1X/ea32efc8d9fcde6c6994e200b8dde3fea9cc64d3.png)

As with the Nexus zkVM, the Jolt zkVM is also extensible, and custom instructions can be added by users at will.

By using only lookup arguments, they overcome one of the main issues of handwritten circuits or even in zkVMs programs: the large attack surface that compilers represent. The simplicity of the Jolt design allows for improved auditability.

What does the future look like?

Compared to STARKish zkVMs, using a compiler to chunk the statement itself and combine the pieces as IVC or GKR approaches do, seems to be a more promising avenue.

Since the goal of a zkVM is to prove efficiently a program, there is still work to do in removing some of the overhead that this abstraction brings. Non-uniformity

was the key innovation that enabled IVC zkVMs to be practical. Abstractions such as co-processors

help to reduce the size of the zkVM circuit.

Parallelism as an alternative to sequential proving may finally render IVC zkVMs comparable to monolithic SNARKs, as Mangrove hints. For this, a compiler that discerns patterns in programs and creates data-parallel

circuits seems to be key in achieving an optimal parallelisation of the prover.

We are seeing that decoupling

is a promising approach to optimisation, whether it is applied to PCD in decoupling the leaf and the merging computation, or to blocks

and co-processors

in contrast to fixed opcodes.

Mangrove SNARKs offered a sane, efficient alternative to zkVMs by providing a uniform

compiler. While it is a refreshing approach, it is just touching the surface of what a compiler may do if it were non-uniform

.

Is there anything else that can be decoupled

, extended

, or make it non-uniform

? Will zkVM stand as the right abstraction in the long run? Will we realise the lookup singularity

? How else can compilers be used?