

# Compiling to zkVMs

Alberto Centelles<sup>a</sup>

<sup>a</sup>Helix AG

\* E-Mail: [alberto@helix.dev](mailto:alberto@helix.dev)

## Abstract

With the advent of non-uniform folding schemes, the lookup singularity, generalised arithmetisations such as CCS and the application of towers of binary fields to SNARKs, many of the existing assumptions on SNARKs have been put into question, and the design space of zkVMs has opened.

Although zkVMs provide a friendly developer experience, their proving time is still significantly (around a million times) slower than direct compilation to circuits due to the overhead of their abstractions (stack, memory, execution unit, etc).

One of the causes of their poor performance is that existing zkVMs are still program agnostic; their provers haven't leveraged the structure of a program. Compilers have a long history of optimising computations by identifying patterns in their structure. We take advantage of the fact that a program is generally executed before it is proven, so the prover of a zkVM is aware of execution trace before establishing a proving strategy. We explore different ways zkVMs may benefit from identifying identical sub-circuits (data-parallel circuits) in programs by analysing techniques such as the GKR protocol, uniform compilers and proof-carrying data (PCD).

**Keywords:** zkVMs; Compilers; GKR; Data-parallel circuits; Recursion; IVC; NIVC; PCD; Folding schemes; Binius; Nova; Protogalaxy; Jolt; Mangrove; Nexus

(Received: 7 Feb 2024; Last version: 19 April 2024)

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	zkVMs	3
1.2	STARKish zkVMs	4
1.3	IVC zkVMs	5
1.4	NIVC zkVMs	7
1.4.1	NIVC zkVMs are RAM machines	7
<b>2</b>	<b>zkVM Compilers</b>	<b>8</b>
2.1	Categories of proving	11
2.1.1	Monolithic proving	12
2.1.2	Piece-wise proving	13
2.1.3	Structure-aware proving	13
2.2	Desiderata	14
2.3	Smart Block Generation	14

2.3.1	Groups of identical opcodes (GKR zkVM)	14
2.3.2	Basic Blocks (GKR zkVM Pro)	16
2.3.3	Uniform compiler (Mangrove)	17
2.3.4	Circuits as lookup tables (Jolt and Lasso)	18
2.4	Fast Provers, Small Proofs, Fast Verifiers	19
2.4.1	Small fields (Plonky2)	19
2.4.2	Smallest Fields (Binius)	21
2.4.3	Large fields, but small values (Jolt)	22
2.4.4	Large fields, but non-uniform folding (SuperNova, HyperNova, ProtoStar)	22
2.5	Modularity	23
2.5.1	Generic accumulation (Protostar)	23
2.5.2	Co-processors (Nexus)	24
<b>3</b>	<b>Conclusion</b>	<b>25</b>
<b>4</b>	<b>Acknowledgements</b>	<b>26</b>
	<b>References</b>	<b>26</b>

## 1. Introduction

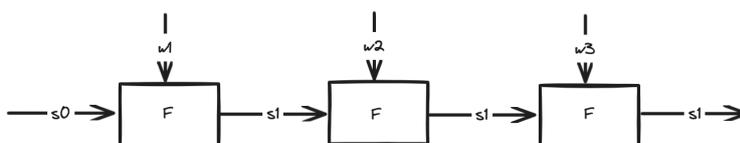
Zero-knowledge proof systems allow us to prove the validity of a statement while hiding some desired information. This statement must be written as an arithmetic circuit or, equivalently, as a set of polynomial equations. High-level languages, such as **Juvix**, must compile to this low-level ZK-friendly language before the validity of a statement can be proven in zero knowledge. This is the role of a front-end. Finally, a SNARK for circuit-satisfiability is applied to a circuit instance. This is what a SNARK backend does. The prover costs of the SNARK backend grow with the size of the circuit. Keeping a circuit small can be challenging because circuits are an extremely limited format in which to express a computation. They consist of gates connected by wires.

There are two main approaches for compiling and proving the validity of a statement written in a high-level language

- *Monolithic*: A program is executed and proven as a single (often giant) circuit.
- *Modular*: A program is potentially divided into subprograms, and each of these subsets of the program becomes a circuit. Other circuits are used to prove relationships between these circuit subsets.

The monolithic approach is naive and is the current approach taken by most SNARK compilers, such as **Vamp-IR**, until the advent of folding schemes. In this monolithic approach, the prover's memory and time requirements deriving from the circuit quickly exceed what is currently reasonable as programs get complex.

The modular approach can be seen as an arbitrary split of a program into multiple chunks or pieces that are proven separately and whose combination ensures the validity of the original statement.



**Figure 1.** State machine.

This modular approach is what recursion and, more specifically, *Incrementally Verifiable Computation (IVC)* offers, and the underlying computational paradigm of a certain type of zkVMs. The main advantage of a zkVM, whether it is STARKish or IVC, is that it allows for the verification of computations that are too large to fit in memory.

### 1.1. zkVMs

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.

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 not 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.

Generally, a zkVM is comprised of five phases:

1. **Compilation** of a program into a set of instructions
2. **Execution** and generation of the execution trace
3. **Proving**
4. (Optional) **Compression**. Not all proofs are large
5. (Optional) **Zero Knowledge**. Not all zkVMs require zero-knowledge

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

## 1.2. STARKish zkVMs

STARKish zkVMs (i.e., those SNARKs based on hashes and error-correcting codes), were the first type of zkVM that was of practical use due to:

- *Fast provers*: At the expense of large proof sizes and slower verifiers, STARK provers are *concretely* faster than other SNARKs not based on hashes and error-correcting codes. These zkVMs may leverage full recursion by wrapping a proof with another SNARK to keep proof sizes and verifier times low.
- *Tailored provers*: STARKish zkVM come with a fixed Instruction Set (IS), that allows provers to be optimised to those specific instructions, in contrast to general purpose SNARK provers, as in Halo2.

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.

Conventional STARKish VMs run over a pre-defined instruction set. Because of the generality of these instruction sets, the circuit size or number of constraints of each opcode tends to be small and zkVMs aim to minimise the amount of opcodes use (e.g., [Cairo paper](#)), since the size of the zkVM circuit is proportional to the number of opcodes. This results in a vast number of instructions that need to be proved, each instruction recurring in computational overhead due to the zkVM abstraction.

STARKish zkVMs are still the most widely deployed type of zkVM. 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.

### 1.3. 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.

An IVC zkVM is a particular instance of IVC where the inputs of a repeated step function  $F$  are a state  $s_{i-1}$  and some other private or public data  $w_i$ .

An IVC zkVM will output a proof that asserts that all states from  $s_0$  to  $s_{i-1}$  reached in  $i - 1$  steps are correct and that the application of a state transition  $F$  to  $s_{i-1}$  gives  $s_i$ . For this to work, we must augment  $F$  with the verification circuit that verifies the proof of the previous step.

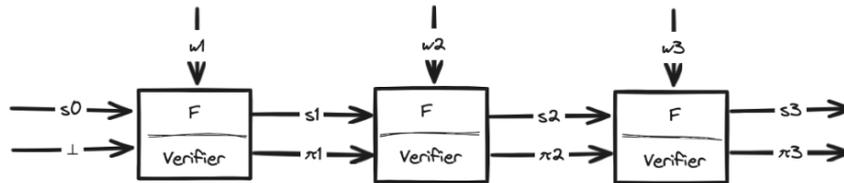


Figure 2. IVC zkVMs.

In other words, there are several properties we need to ensure, each of them encoded as a circuit:

- The previous state  $s_{i-1}$  has been correctly recognised and the input  $s_{i-1}$  to the state transition function  $F$  is correct. This is known as memory checking.
- The state transition  $F$  process itself is correct.

In short, IVC allows us to obtain proofs for long computations with relatively little memory by splitting them into iterative, verifiable, shorter computations.

While this encapsulates the essence of an IVC zkVM, two important factors render this original cryptographic primitive, as described in the original construction of IVC, inefficient:

- The function  $F$  representing a state transition is a fixed circuit that encodes *all* instructions. Moreover, if a circuit encodes different branches, the proving time is also proportional to all branches, whether they are used or not.
- The verifier algorithm encoded as a circuit that extends each state transition  $F$  is generally bigger than most opcodes. In particular, for the KZG polynomial commitment scheme, it involves checking openings in a polynomial commitment scheme using pairings and pairing operations require many constraints.

Halo [BGH19] introduced the notion of *accumulation schemes* to address the embedding of a full verifier algorithm at each step in the scheme. Their scheme defers the computationally expensive part of the verifier algorithm (i.e., the linear time polynomial commitment opening checks) and a separate party (called *decider*) later verifies the last state transition and accumulated instance, which already verifies every state transition from genesis.

The accumulation of the most expensive verifier checks to a later stage opened up the possibilities of suitable SNARKs in an IVC scheme since the performance of the verifier is no longer required to be sub-linear, and thus,

we can even choose an SNARK with an expensive verifier such as in Bulletproofs [BBB<sup>+</sup>17]. Because of the Inner Product Argument (IPA) polynomial commitment scheme introduced in Bulletproofs, Halo2 does not need a trusted setup.

Folding schemes go a step further and defer *all* verifying checks until all proofs are generated.

The original folding schemes such as Nova, were not enough to instantiate an efficient SNARK derived from a given program. In a zkVM with multiple instructions, these folding schemes require the size of the circuit  $F$  to be linear to the number of instructions.

#### 1.4. NIVC zkVMs

Arguably, the main breakthrough in IVC-based zkVMs was done by SuperNova [KS22], 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.

NIVC zkVMs allow different instruction circuits to be written without the need to use switches to toggle circuits, reducing circuit size at each state transition and achieving a “à la carte cost profile”.

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, a zkVM needs to prove that the stack connected a value between two opcodes

NIVC zkVMs in particular can potentially benefit from compiler passes. A compiler may leverage the information it gathers from a program to create the set of step functions  $F_i$  at compile time. They no longer need to be small instructions but subsets of the whole program created at compile time.

##### 1.4.1. NIVC zkVMs are RAM machines

A useful conceptual framework for handling state in a zkVM is by thinking of it as a RAM (Random Access Memory) machine that supports  $l$  instructions,  $s$  registers of width  $w$  bits and memory of size  $2^w$ .

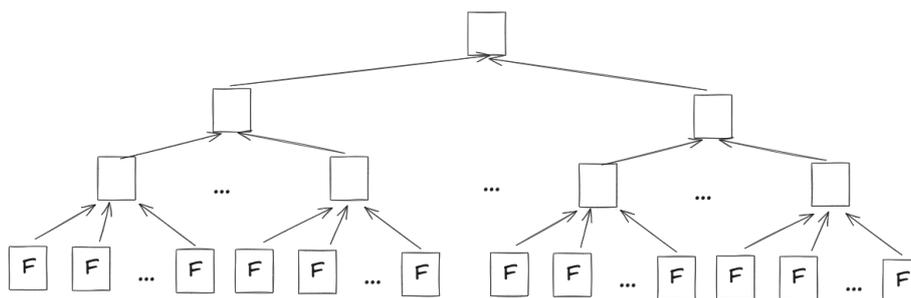
Each of the step functions  $F_i$  in  $\{F_1, \dots, F_n\}$  represents the instruction  $i$  that the machine supports. The input of this state transition  $F_i$  consists of  $s + 1$  field elements, where the first entry (the “program counter”) holds a commitment to a memory (e.g., the root of a Merkle tree with  $2^w$  leaves) that stores both a program and its state, and the remaining entries are the values of  $s$  registers. The output of each  $F_i$  consists of  $s + 1$  field elements that are updated values of the provided input.

For step  $i$ , state  $s_{i-1}$  and non-deterministic witness  $\omega_{i-1}$ , the selector function  $\phi(s_{i-1}, \omega_{i-1})$  picks the instruction in the memory (whose commitment is at  $s_{i-1}[1]$ ) at address in the program counter register  $s_{i-1}[2]$ . The initial state  $s_0[1]$  holds a commitment to the verifier's desired memory of size  $2^w$  with its program stored in it, and the rest of  $s_0$  contains the verifier's desired initial values of the machine's registers.

**Remark 1.** Designing zkVMs is as much a cryptography problem (i.e., finding the most efficient schemes or back-ends to prove a given NP statement) as it is a compilers problem (i.e., designing the right transformations of a program to improve the performance of the scheme).

## 2. zkVM Compilers

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.



**Figure 3.** Proof-carrying data

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.

While a zkVM is *not* a compiler, any end-to-end architecture for proving computation of a program involves the design of a compiler with that of a zkVM. The role of a compiler is to identify the sequence of opcodes used in a program and pass this information as an input to the zkVM circuit.

A zkVM is a universal circuit comprised by a set of instructions and other abstractions such as a stack or a memory, so these are encoded into a generally large circuit.

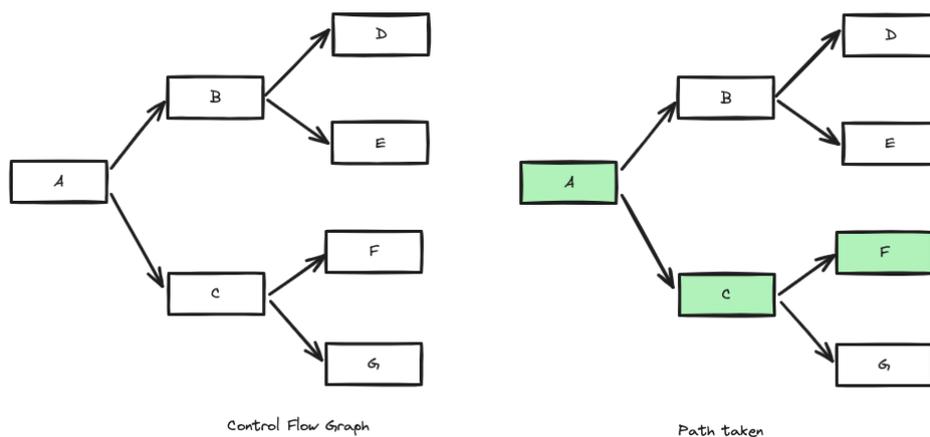
If a zkVM is defined as a universal circuit that encodes a set of instructions, what happens if we modify this set of instructions, leaving everything else unchanged? Do we still have the same zkVM or do we have a different one? If the latter, every time we modify an instruction set by adding or removing an instruction, we are compiling to a different zkVM. The same logic applies to memory checking algorithms.

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?

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.

We want to use the structure of a program to compile to a suitable zkVM for that particular program. This awareness of the structure of a program can be also applied to provers.

What is a program? One very common abstraction used in compilers is that of a control flow graph, which splits a program into a series of blocks and arrows of blocks based on jumps. A compiler takes a program written in a high-level language and outputs a circuit. The inputs and outputs of a circuit are finite field elements. A compiler should be able to *decide* which circuits or blocks are derived from a program and a prover would aim to minimise the cost of proving such subset of circuits.



**Figure 4.** Control Flow Graph.

For example, given a program  $C = \{A, B, C, D, E, F, G\}$  and inputs  $x, w$  for which the path  $P = [A, C, F]$  is taken, a compiler will output the set of circuits

while the prover's costs will only depend on this path  $P$ , not on the blocks not taken. With folding schemes, each path in  $P$  can be folded into a single instance.

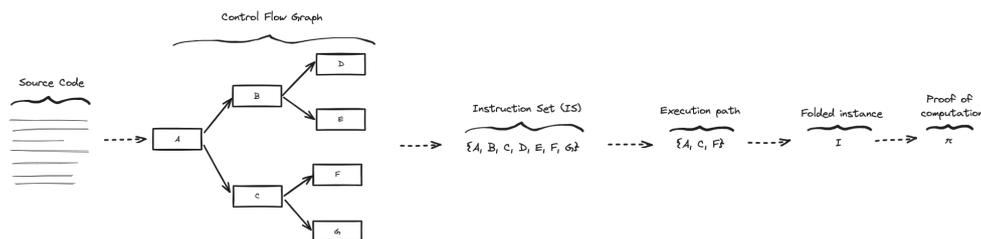
Different back-ends provide different trade-offs that will affect how the output of the compiler. This dance between compilers and proving systems is the main focus of this report.

Whatever zkVM will be suitable for a given program will be determined at compile time.

A compiler recognises patterns in a program, that is, repeated structures that are used to optimise the performance of that program.

What are these patterns in the context of a zkVM? They can be a new instruction set, or they can be blocks containing a sequence of opcodes within, or something else such as co-processors.

If the set of instructions  $\{F_1, \dots, F_N\}$  of the zkVM is set *dynamically* at compile time, we find ourselves back in the scenario that zkVMs address: compiling directly to circuits is hard. On the other hand, how can we treat each blocks dynamically while preserving the *universal* property of a zkVM?



**Figure 5.** Compiling to zkVMs.

When splitting a program into multiple chunks, we want to strike a balance between the size of each step function and the number of steps. Every step incurs in some overhead while large steps take more computational and memory resources. On one end of this spectrum, we have the whole program being one big circuit; on the other end (using a folding scheme as example), we fold of all primitive operations such as addition, equality or multiplication. There is no folding in the former case. As we have seen, this monolithic approach is impractical for the majority of programs, since they exceed both memory and computational resources that a prover generally has access to. The base overhead of folding in the latter case may be comparable to the cost of proving due to having such small circuits (and there are many of them!). Adding a small folding overhead to many small circuits removes almost all the advantages of folding. So, where is the balance?

NIVC zkVMs can benefit from compiler passes. While it is common to *fix* the set of functions  $\{F_1, \dots, F_n\}$  to a basic instruction set, it does not have to

be so. A compiler can leverage the information it gathers from a given program to create these step functions  $F_i$  at compile time. They no longer need to be small instructions, but subsets of the whole program created at compile time. Given some design constraints, the compiler, acting as a front-end to a SNARK, can split the program  $F$  into a set of  $\{F_1, \dots, F_n\}$  subprograms. The compiler must be given a heuristic of this desired balance between circuit size and number of circuits. For example, equilibrium might be found by creating bounded circuits of  $2^{12}$  gates with only one witness column. The compiler must have an educated guess of how to decompose larger circuits into smaller ones.

This holistic or dynamic approach that leverages the structure of a program using compilers is not commonly seen. We'll analyse different novel approaches in the next section.

## 2.1. Categories of proving

Let us use an example to highlight the different ways of compiling a program. Let *multi\_algorithms* be a program that serves as a database of algorithms: given a program identifier, it proves knowledge of the outcome of that algorithm. We have mixed some algorithms together to illustrate how real-world applications tend to be assembled.

For example, in the first program (i.e., *program\_id=0*), we prove both knowledge of the  $n$ -th Fibonacci number and that it is a prime number.

```

fn multi_algorithms(program_id: Field, n: Field, private answer: Field) {
  range_check(program_id, 2)
  if program_id == 0 {
    let m = fibonacci(n);
    assert_eq(answer, m)
    assert(is_prime(m))
  }
  if program_id == 1 {
    range_check(n, 16);
    let m = factorial(n);
    assert_eq(answer, m)
  }
  if program_id == 2 {
    assert(is_prime(m))
    let m = power_of_seven(n);
    assert_eq(answer, m)
  }
  if program_id == 3 {
    assert_eq(answer, collatz(n))
  }
}

```

Figure 6. The multi\_algorithms example.

Let us revisit the three different compiling and proving approaches studied so far:

### 2.1.1. Monolithic proving

This would be a direct compilation into a circuit for an arithmetisation (e.g., Halo2 Plonkish) or the approach that conventional STARKish zkVMs use.

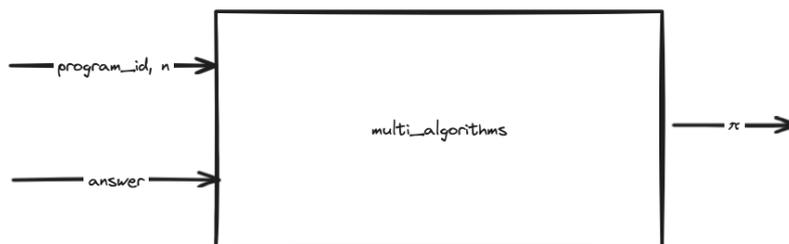
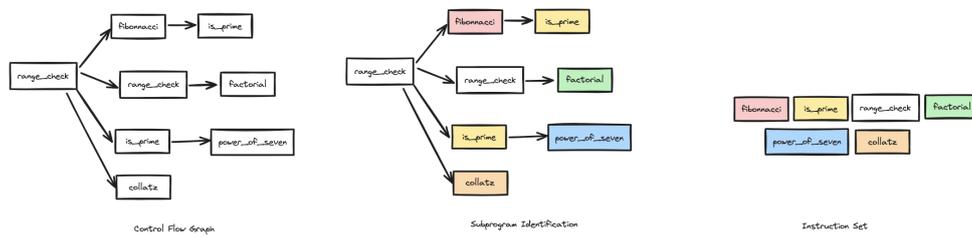


Figure 7. Monolithic circuits.

The circuit derived from this approach contains all the unused branches, since the compiler does not know the inputs of the program. For a complex application, this turns to be a large circuit with high memory requirements. Since, proving time is at least  $O(n)$ , where  $n$  is the number of gates (including



**Figure 9.** Dynamic instruction set.

all the unused branches), and memory consumption is also often linear, this approach turns out to be inefficient for large applications.

STARKish zkVMs only make this problem worse, since the abstraction of a zkVM makes the proving of most programs around a million times slower than directly proving the program without this abstraction.

### 2.1.2. Piece-wise proving

The prover do not prove the whole program at once, but proves each of these primitive operations, one at a time.



**Figure 8.** Fixed instruction sets.

Having a set of fixed instructions for any possible program turns out to be quite inefficient when the prover provides a proof for each of these opcodes, since these opcodes happen to be too small to justify the overhead of proving.

### 2.1.3. Structure-aware proving

Fixed-instruction-set zkVMs do not have knowledge of the program (so they may include instructions that a particular program might never use).

In contrast, using some heuristic, a compiler may leverage the patterns of a given program. For our naive example, a possible instruction set is depicted below. The aim of such compiler is to choose blocks of computation that are big enough to justify the overhead of folding and small enough to avoid blowing up memory resources and other issues derived from large circuits.

This set must be crafted with some considerations. Notice that functions such as *range\_check*, which checks that a value is between 0 and  $2^n$ , or *is\_prime* are independent of the rest of the computation. They are not too complex but not too small, and are used multiple times.

*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.

## 2.2. Desiderata

- **Smart Block Generation:** We want to leverage the structure of a program and provide a compiler that decomposes any program into a set of circuit blocks that can be proven efficiently using folding schemes, the GKR protocol, lookups or otherwise.
- **Fast Provers, Small Proofs, Fast Verifiers:** We want short proof generation time and fast verification, both in terms of asymptotic performance and overheads. Recursion, and specially folding schemes, allow for slow verifiers and large proofs, since they can be overcome with folding. Small fields and small value also improve the performance of the prover.
- **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.

## 2.3. Smart Block Generation

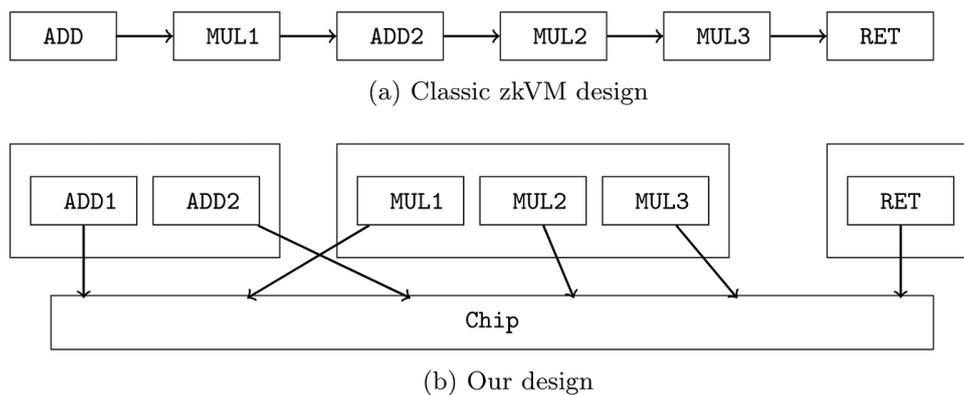
### 2.3.1. Groups of identical opcodes (GKR zkVM)

One of the main ideas behind the GKR zkVM (as introduced in the paper *Parallel Zero-Knowledge Virtual Machines* [HLZ<sup>+</sup>24]) 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:

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

- **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.



**Figure 10.** GKR zkVM

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.

### 2.3.2. Basic Blocks (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.

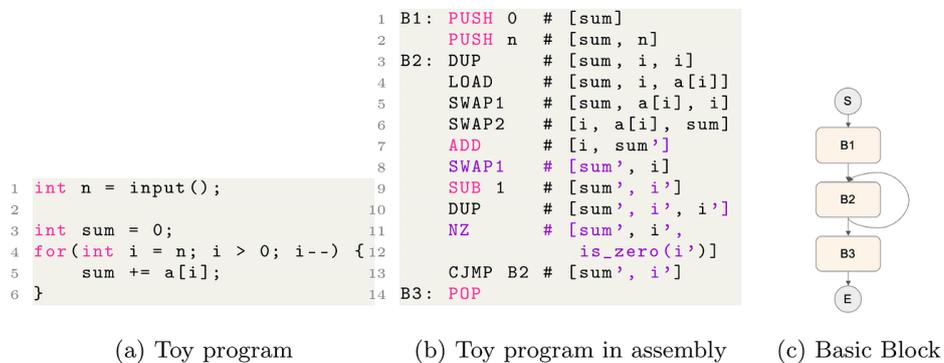


Figure 11. GKR zkVM Pro

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:

1. Compile program into assembly
2. Identify basic blocks
3. Prove each block separately
4. 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.

### 2.3.3. Uniform compiler (Mangrove)

The main idea of Mangrove [NDC<sup>+</sup>24] 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.

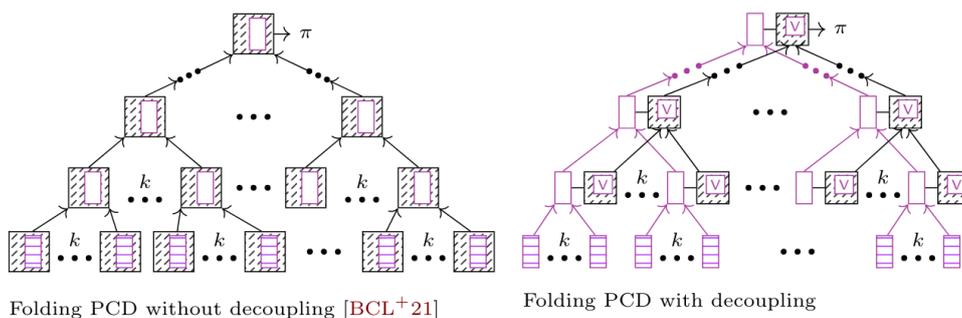


Figure 12. Mangrove PCD

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.

#### 2.3.4. Circuits as lookup tables (Jolt and Lasso)

Just One Lookup Table (Jolt) [AST23] 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  $t_1, \dots, t_l$ , each of size  $N^{1/c}$ .

For example, if an instruction takes one 64-bits-input, this would result in a  $2^{64}$  size lookup table. This is generally too large to materialise. But if instead we chunk that evaluation table 4 times, this costs goes down to  $2^{16}$ , which is entirely practical. The reduction is exponential.

Jolt claims 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-optimize 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 [STW23], 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 enor-

amous 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.

By using only lookup arguments, Jolt overcomes 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.

How can we use a compiler in this context? Jolt is already a frontend compiler that converts computer programs into a lower-level representation and then uses Lasso to generate a SNARK for circuit-satisfiability. Jolt is extensible in the sense that any custom instruction encoded as an evaluation table can be appended directly to the giant evaluation table the the Jolt zkVM represents. If a compiler wants to optimise a Jolt zkVM, it must ensure that generated lookup tables are decomposable.

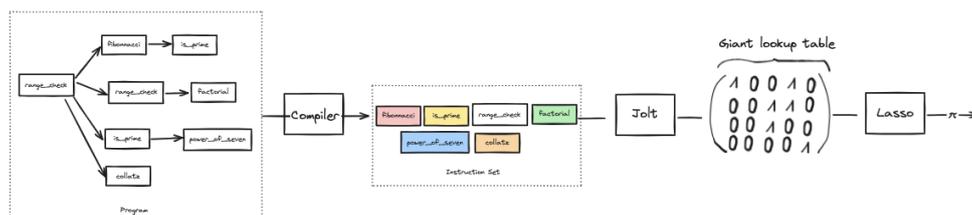


Figure 13. Compiler + Jolt pipeline.

We refer to appendix ?? for a summary of the main ideas of Jolt.

## 2.4. Fast Provers, Small Proofs, Fast Verifiers

### 2.4.1. 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 their polynomial commitment scheme, 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),

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	Bandwidth-bound

**Table 1.** Comparison of Elliptic Curve SNARKs and STARKish Protocols.

so they are *Scalable* and *Transparent*. On the other hand, STARKs today are deployed *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.

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, elliptic curves based on extension fields are likely suffering from specific attacks that do not apply to common elliptic curves constructed over large prime fields [SSS22].

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.

Plonky2 and its successor, Plonky3, take the Interactive Oracle Proof (IOP) from PLONK and mix it with the FRI Polynomial Commitment Scheme (PCS) to construct their SNARK.

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  $p$  is  $2^{31} - 1$ , thus suitable for 32-bit CPUs (it fits within a 32-bit word).

It is unclear how Plonky3 and similar approaches will benefit from folding schemes, since FRI is not an additively homomorphic PCS. However, recent work [BMNW24] introduces a technique for accumulating without a homomorphic PCS.

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

Plonky2	Plonky3	Binius
$p = 2^{64} - 2^{32} + 1$	$p = 2^{31} - 1$	$p = 2$
Goldilocks	Mersenne31	Binary

**Table 2.** Comparison of Prime Fields in Plonky2, Plonky3, and Binius.

### 2.4.2. Smallest Fields (Binius)

As we have 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 for 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  $p = 2^{64} - 2^{32} + 1$ , see [Table 2](#). 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 “Succinct Arguments over Towers of Binary Fields” (also known as *Binius*) [[DP23](#)] 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 [[CBBZ22](#)]. In particular, they leverage the [sum-check protocol](#) and “small” values protocols, where the prover commits only to small values. They revive the polynomial commitment scheme Brakedown [[GLS+21](#)], 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  $\mathbb{F}_{2^k}$ , etc.), and the gate constraints may express polynomial relations defined over particular subfields of the tower.

#### 2.4.3. Large fields, but small values (Jolt)

Although Jolt is compatible to binary fields and other small fields, their underlying lookup argument, Lasso, is designed to work mainly over small values, independently of the size of the field, by avoiding random values. The intuition is that the cost of multi-scalar multiplication depends on the size of values, not on the size of the field. Multiplying a point by a small scalar is cheaper than by a larger one.

#### 2.4.4. 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.

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  $\{F_1, \dots, F_L\}$ . Generally, the size of the circuit  $C$  is much bigger than  $L$ , so effectively the number of sub-circuits or instructions  $\{F_1, \dots, F_L\}$  do not come at any cost to the prover.

## 2.5. Modularity

### 2.5.1. 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 arithmetisation 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.

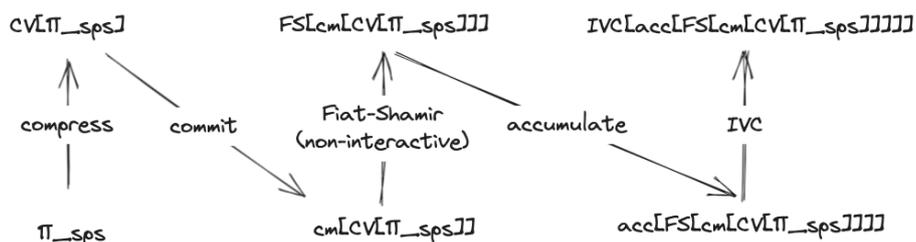


Figure 14. ProtoStar progressive blocks.

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.

### 2.5.2. Co-processors (Nexus)

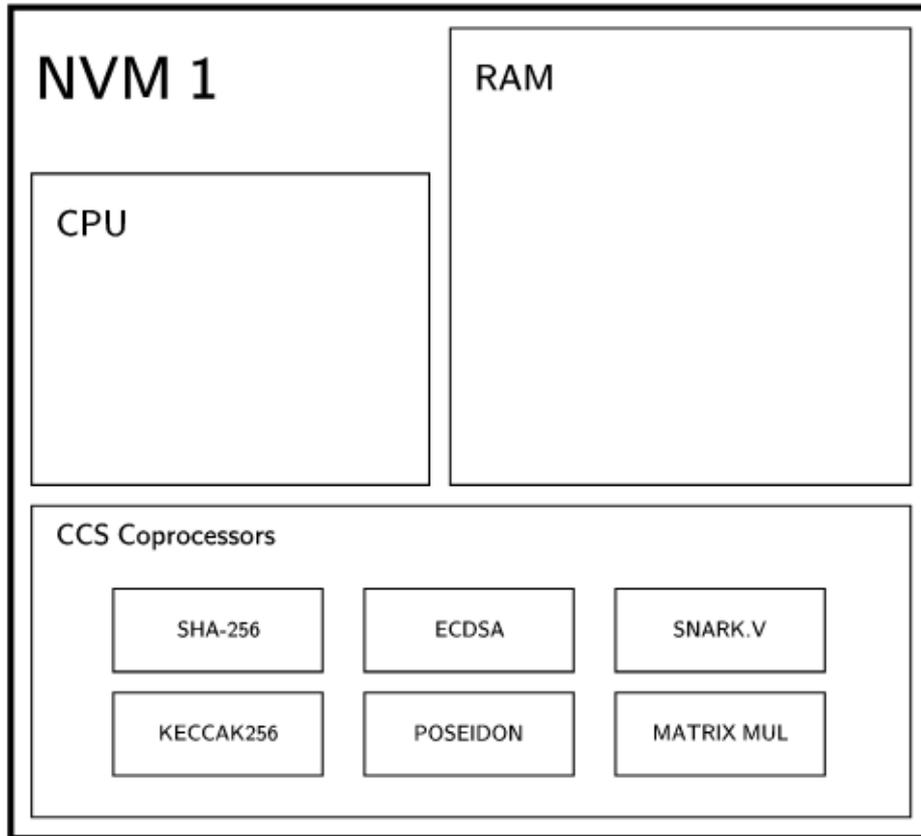
The **Nexus zkVM** is a simple, minimal, extensible and parallelisable folding-based zkVM, so it also realises PCD. It is simple in its architecture, memory model and I/O model. It is minimal in the sense that its instruction set, which is defined in the setup phase, can contain as many instructions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

$$\Pi = \text{PCD}[\text{NIVC}[\text{FS}[\text{CF}[\text{MFS}[\mathcal{R}_{\text{CCS}}]]]]]]$$

**Figure 15.** Nexus zkVM signature.

The Nexus zkVM is obtained by the following sequence of transformations:

- $\text{MFS}[\text{RCCS}]$ : Multi-folding scheme for a CCS relation
- $\text{CF}$ : CycleFold
- $\text{FS}$ : Fiat-Shamir transformation for multi-folding
- $\text{NIVC}$ : SuperNova non-uniform IVC transformation for non-interactive multi-folding schemes
- $\text{PCD}$ : Parallelization (proof-carrying-data) transformation for NIVC schemes



**Figure 16.** Nexus zkVM co-processors.

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.

### 3. Conclusion

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 simple exploration of what a uniform compiler can achieve with vanilla Plonk. While it is not presented as a zkVM, it can be easily turned into one. Mangrove is just touching the surface of what a compiler may be able to 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?

## 4. Acknowledgements

I want to thank Lukasz Czajka, Christopher Goes, Adrian Hamelink, Ferdinand Sauer and Xuyang Song for their technical feedback and fruitful discussions, and Jonathan Cubides for supporting this type of research work.

## References

- AST23. Arasu Arun, Srinath Setty, and Justin Thaler. Jolt: Snarks for virtual machines via lookups. Cryptology ePrint Archive, Paper 2023/1217, 2023. <https://eprint.iacr.org/2023/1217>. (cit. on p. 18.)
- BBB<sup>+</sup>17. Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. Cryptology ePrint Archive, Paper 2017/1066, 2017. <https://eprint.iacr.org/2017/1066>. (cit. on p. 7.)
- BGH19. Sean Bowe, Jack Grigg, and Daira Hopwood. Recursive proof composition without a trusted setup. Cryptology ePrint Archive, Paper 2019/1021, 2019. <https://eprint.iacr.org/2019/1021>. (cit. on p. 6.)
- BMNW24. Benedikt Bünz, Pratyush Mishra, Wilson Nguyen, and William Wang. Accumulation without homomorphism. Cryptology ePrint Archive, Paper 2024/474, 2024. <https://eprint.iacr.org/2024/474>. (cit. on p. 20.)
- CBBZ22. Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. Hyperplonk: Plonk with linear-time prover and high-degree custom gates. Cryptology ePrint Archive, Paper 2022/1355, 2022. <https://eprint.iacr.org/2022/1355>. (cit. on p. 21.)
- DP23. Benjamin E. Diamond and Jim Posen. Succinct arguments over towers of binary fields. Cryptology ePrint Archive, Paper 2023/1784, 2023. <https://eprint.iacr.org/2023/1784>. (cit. on p. 21.)
- GLS<sup>+</sup>21. Alexander Golovnev, Jonathan Lee, Srinath Setty, Justin Thaler, and Riad S. Wahby. Brakedown: Linear-time and field-agnostic snarks for r1cs. Cryptology ePrint Archive, Paper 2021/1043, 2021. <https://eprint.iacr.org/2021/1043>. (cit. on p. 21.)

- HLZ<sup>+</sup>24. Wenqing Hu, Tianyi Liu, Ye Zhang, Yuncong Zhang, and Zhenfei Zhang. Parallel zero-knowledge virtual machine. Cryptology ePrint Archive, Paper 2024/387, 2024. <https://eprint.iacr.org/2024/387>. (cit. on p. 14.)
- KS22. Abhiram Kothapalli and Srinath Setty. Supernova: Proving universal machine executions without universal circuits. Cryptology ePrint Archive, Paper 2022/1758, 2022. <https://eprint.iacr.org/2022/1758>. (cit. on p. 7.)
- NDC<sup>+</sup>24. Wilson Nguyen, Trisha Datta, Binyi Chen, Nirvan Tyagi, and Dan Boneh. Mangrove: A scalable framework for folding-based snarks. Cryptology ePrint Archive, Paper 2024/416, 2024. <https://eprint.iacr.org/2024/416>. (cit. on p. 17.)
- SSS22. Robin Salen, Vijaykumar Singh, and Vladimir Soukharev. Security analysis of elliptic curves over sextic extension of small prime fields. Cryptology ePrint Archive, Paper 2022/277, 2022. <https://eprint.iacr.org/2022/277>. (cit. on p. 20.)
- STW23. Srinath Setty, Justin Thaler, and Riad Wahby. Unlocking the lookup singularity with lasso. Cryptology ePrint Archive, Paper 2023/1216, 2023. <https://eprint.iacr.org/2023/1216>. (cit. on p. 18.)