Anoma Research Topics | TECHNICAL REPORT

Compiling to zkVMs

Alberto Centelles[a]

[a]Heliax AG

* *E-Mail: alberto@heliax.dev*

Abstract

With the advent of non-uniform folding schemes, 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.
We explore the concept of a zkVM as an efficient alternative to the existing naive circuit-building and proving approach to large computations. While there are different types of zkVMs, we focus on the ones based on Incrementally Verifiable Computation, which allow us to prove incrementally small steps and accumulate its proofs.
Lastly, we discuss the role of a compiler in optimising zkVMs, where the instruction set is not fixed but determined at compile time and suggest different compilation pipelines for future zkVMs.

Keywords: Protostar ; Protogalaxy ; Nova ; Halo2 ; Jolt ; Lasso ; Juvix ; Compilers ; zkVM ; Folding Schemes ; Accumulation ; Recursion

(Received: 7 Feb 2024)

Contents

1. Introduction

1.1. Recursion

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

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

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

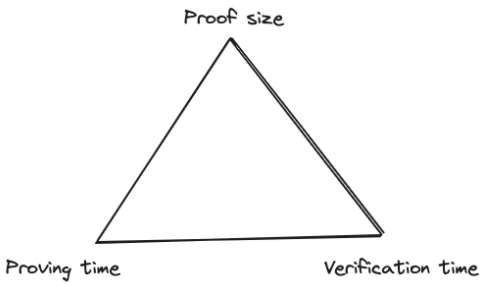

Figure 1. SNARK trade-offs triangle.

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

1.2. Schemes from recursive proof composition

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

1. Full recursion,
2. Atomic accumulation,
3. Split accumulation, and
4. Folding schemes.

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

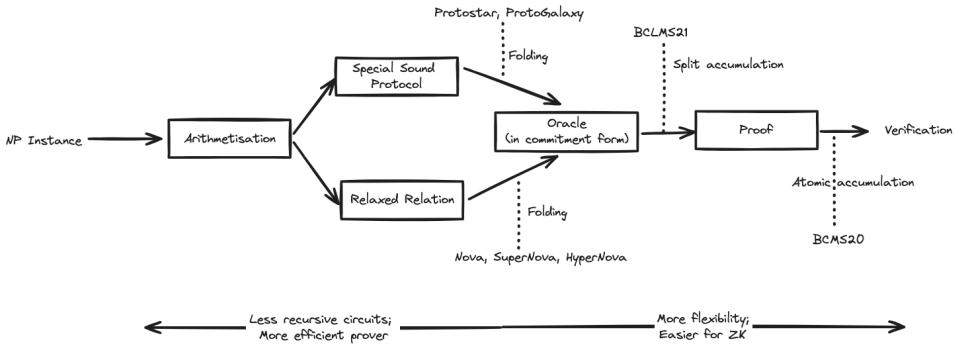

Figure 2. Credit: KiloNova paper.

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

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

As a side note, both "Special Sound Protocol" and "Relaxed Relation" are two different techniques to relax the constraints of the arithmetisations to accommodate folding. We will discuss arithmetisations later in Section 3.

1.3. Full recursion

This is the first and most obvious approach to recursion, sketched by Valiant [Val08] in 2008. In this scheme, the full verifier algorithm of a SNARK is turned into a circuit and appended to the circuit that represents each step in the chain of computations. At every step i, the proof π_i asserts that all prior computations were verified. For this scheme to be practical, the underlying proof system must have a succinct verifier, that is, sublinear in time complexity. We can find Groth16 [Gro16], Plonk [GWC19] and KZG [KZG10]. However, a practical scheme cannot be achieved for long computation chains, even with a succinct verifier for each iteration. Ideally, we want the prover to do linear work, with just a constant factor penalty over simply running the computation.

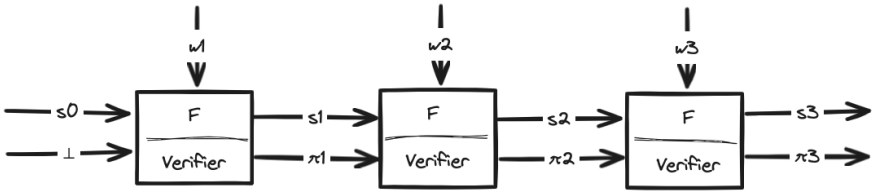

Figure 3. Full recursion.

1.4. Atomic accumulation

In their paper "Recursive Proof Composition *without a Trusted Setup*" [BGH19] (also known as the *Halo paper*), the ZCash team noticed that the verifier algorithm is composed of fast and slow sub algorithms and that for certain type of SNARKs, the verification of the linear part can be accumulated in any IPA-based SNARK. The sublinear part of the verifier algorithm is still turned into a circuit and appended to each iteration of the recursive scheme.

Following this work, "Proof-Carrying Data *from Accumulation Schemes*" [BCMS20] coined the term *accumulation schemes* to describe this particular variant of IVC, and "Halo Infinite: Proof-Carrying Data from Additively Polynomial Commitments" [BDFG20] generalised the Halo construction to any additively homomorphic polynomial commitment scheme.

A polynomial commitment scheme (PCS) allows the prover to convince a verifier that, given a commitment to a polynomial f, a challenge point x and an evaluation y, we have that $f(x) = y$. In this recursion scheme, whether it uses IPA or KZG as a PCS, the verifier accumulates the linear computation and performs the sublinear check. A PCS is additively homomorphic if can take a random linear combination of polynomials $\{f_i\}$ and their commitments $\{C_i\}$ separately and ensure with high probability that $C := C_1 + \alpha C_2 + \alpha^2 C_3 + \ldots + \alpha^n C_n$ is a commitment of $f := f_1 + \alpha f_2 + \alpha^2 f_3 + \ldots + \alpha^n f_n$.

It is important to note that this accumulation technique cannot be applied

to STARKs since its polynomial commitment scheme FRI is based on hashing, and hashes are not additively homomorphic.

1.4.1. Inner Product Argument (IPA)

In particular, in the inner product argument (IPA) of Halo, the linear computation is an inner product $C_i = \mathbf{G}, \mathbf{s} = G_1 \cdot s_1 + \cdots + G_n \cdot s_n$, where $\mathbf{s}$ is the round of challenges $\{u_1, \cdots, u_k\}$ of that particular recursion step, and $\mathbf{G}$ is a vector of random group elements G_i publicly given at the beginning of the protocol. The verifier needs to compute $C_i = \mathbf{G}, \mathbf{s}$, which is a linear-time multiscalar multiplication, and $b = \mathbf{s}, \mathbf{b} = g(x, u_1, \cdots, u_k)$, which can be computed by the verifier in logarithmic time. This latter one is the sublinear check.

Their key observation is that C_i is a commitment. In the final step of this PCS, the verifier performs a random linear combination of the accumulated commitments, $C = C_1 + \alpha C_2 + \cdots + \alpha^m C_m$ and verifies the argument in $O(m \cdot log(d))$. Since there is only one verifier check at the end, the cost is amortised.

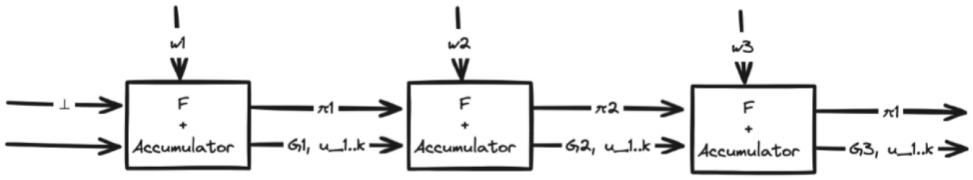

Figure 4. Halo2 IPA recursion.

1.4.2. Kate, Zaverucha, Goldberg (KZG)

In the KZG polynomial commitment scheme [KZG10], the verifier performs two operations:

- Creating a pair of a polynomial and a commitment to it.
- Checking the polynomial-commitment pair.

The first part of creating a pair is fast (sublinear) and so extends the circuit F in the accumulation scheme; the second part of checking the polynomial-commitment pair is slow since it involves pairings and is accumulated until the end of the scheme. As with IPA, accumulating the linear check is possible because this polynomial commitment scheme is additively homomorphic, and the cost is also amortised.

1.4.3. Split accumulation

In the paper "Proof-Carrying Data *without Succinct Arguments*" [BCL+20] (also known as *BCLSM21*), Bünz et al. realised that the expensive succinct property of a SNARK is unnecessary for building accumulation schemes, i.e.,

the proof of each iterative computation do not need to be succinct to get the succinctness property of the overall scheme. They proposed a scheme in which each iteration is simply a NARK and run a single SNARK at the end of the scheme. This led to the so-called *Split Accumulation* technique. In this scheme, the verifier and prover do not need to verify or prove each iteration completely, respectively, and it generates a NARK instead of a SNARK.

1.5. Folding schemes

A natural continuation of the previous trends came with the work of "Nova: Recursive Zero-Knowledge Arguments from Folding Schemes" [KST21], in which the sublinear work of the verifier is also deferred.

From here, an explosion of works emerged, extending this construction of Nova. Sangria extends this with the Plonkish arithmetisation and HyperNova [KS23b] with Customisable Constraint System (CCS) [STW23a]. SuperNova [KS22] extended Nova with a new technique called **Non-uniform IVC (NIVC)** and Protostar [BC23] generalised the construction of Nova, introducing a generic folding scheme.

1.5.1. Initial constructions

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

The argument of folding schemes goes as follows: given a set of sequential computations $F \rightarrow F \rightarrow \cdots \rightarrow F$ rather than computing a SNARK proof π_i for each iteration, we fold them into a single instance F^* for which we produce a single SNARK proof π.

That is, instead of providing evidence that each step function F is computed correctly, instances are "folded" into a compressed instance (i.e., an instance that encapsulates all previous instances), and the prover outputs a proof of correct folding. Practically, this only works because folding is much cheaper (constant size) compared to verifying a SNARK.

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

The main improvements of folding over other recursion or accumulation techniques are the following.

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

1.5.2. Multi-folding

A folding scheme for a relation R is a protocol between a prover and a verifier that reduces the task of checking two instances in R with the *same* structure S into the task of checking a single folded instance in R also with structure S.

Introduced in the HyperNova paper, multi-folding is a generalisation of folding that allows the folding of two collections of instances in relations R_1 and R_2 with the same structure S. In layman's terms, this means that we can fold two *different* circuits as long as they have the *same* arithmetisation. In HyperNova, this arithmetisation is CCS.

The main idea of multi-folding, as introduced in HyperNova, is to fold a CCS instance into an augmented, more restricted variant of CCS (called linearised CCS) that is carried throughout the long-running computation. This instance is called a "running instance". That is, $F_i : CCS \times CCS_{linearised} \longrightarrow CCS_{linearised}$. The running instance is denoted by U, and the CCS instance representing the last step in the computation is denoted by u.

The work of "KiloNova: Non-Uniform PCD with Zero-Knowledge Property from Generic Folding Schemes" [ZGGX23] extends multi-folding to allow folding two instances of different structures.

1.5.3. Non-uniform IVC

In IVC, the iterative function F must always be the same for each iteration. In this case, this function is said to be *uniform* or *universal*. For most use cases of IVC, we may want F to be different or *non-uniform*.

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

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

SuperNova introduced the Non-uniform IVC (NIVC) construction. The subtitle of this paper reads: "Proving universal machine executions without universal circuits". This means that, instead of one *universal* circuit F, we have a bunch of different step functions $F_{\phi(s_{i-1},w_i)}$ parameterised by ϕ : some-program-data $\longrightarrow \{1, \cdots, n\}$ where only one step function F_i is chosen per iteration. The set of step functions $\{F_1, \cdots, F_n\}$ can be understood

as an instruction set in the context of zkVMs.

In a traditional IVC setting, the function or circuit F that we are iterating over must always be the same, even if it comprises multiple functions. A good analogy is a custom gate with selectors, and the cost is linear to the number of functions $\{F_1, \cdots F_l\}$ that compose F. With NIVC, both the size of the circuit and the cost of computation are only linear to the particular $F_{\phi(s_{i-1}, w_i)}$ that gets executed. The per-step proving cost is independent of the sizes of circuits of "uninvoked" instructions.

Formally, for a specified $(\{F_1, \cdots, F_l\}, \phi)$ and (n, s_0, s_n), the prover proves the knowledge of a set of non-deterministic values $\{\omega_0, \cdots, \omega_{n-1}\}$ and $\{s_1, \cdots, s_{n-1}\}$ such that $\forall i \in 0, \cdots, n-1$, we have that $s_{i+1} = F_{\phi(s_i, \omega_i)}(s_i, \omega_i)$. This innovation renders many applications based on folding schemes computationally feasible, particularly zkVMs.

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

1.6. Cycle of Curves

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

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

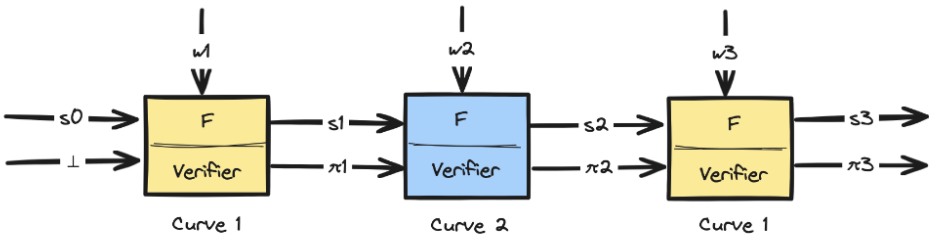

Figure 5. IVC over a cycle of curves.

Any recursion scheme will likely need to be implemented over a cycle of curves. This has consequences on which proving systems to use and which curves work. Works like "CycleFold: Folding-scheme-based recursive arguments over a cycle of elliptic curves" [KS23a] focus on which operations are optimised on which curve. For example, in their scheme, only group operations are performed in the outer curve and then compressed.

1.7. Remarks

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

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

Table 1. Recursive schemes comparison.

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

Recursion and, particularly, folding schemes are radically changing how we design SNARKs by removing some artificial limitations of today's popular SNARKs. The importance of having an efficient recursive prover, even at the expense of a large proof size or a slow verifier (which can later be folded), has revamped proving systems such as "Brakedown: Linear-time and field-agnostic SNARKs for R1CS" [GLS+21] that were neglected.

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

2. zkVMs

2.1. Motivation

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 whole program becomes a single (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 a state machine where every state transition (or chunk of the program) outputs a proof of valid computation. Ideally, at each step, in addition to proving that the state transition is correct, it is proven that all state transitions are correct from genesis up to the current state.

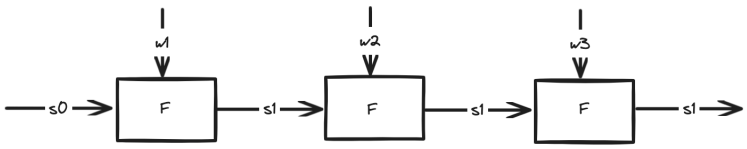

Figure 6. State machine.

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 STARK-based or IVC-based, is that it allows for the verification of computations that are too large to fit in memory.

2.2. Description

One way to think about a zkVM is as a proving system that generates a proof of correct execution from an input program and data. Generally, they emulate the von Neumann architecture and prove relations between a program's execution and its use of Random Access Memory. These proving systems or SNARKs are optimised for circuits that are decomposable into a given set of instructions and target machine abstractions, also known as Instruction Set Architectures (ISA). Thus, this instruction set is a fundamental component of a zkVM.

The other main component of a zkVM is the Virtual Machine (VM). A VM emulates physical hardware, but compared to physical hardware, the

virtualised hardware in a zkVM can change as long as the prover and verifier agree on it. This gives designers fewer constraints and more choices and trade-offs.

2.3. STARK-based zkVMs

STARK zkVMs 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 faster than other non-FRI-based SNARKs. These zkVMs leverage full recursion to keep proof sizes and verifier times low.
- *Tailored provers*: STARK-based zkVM come with a fixed Instruction Set (IS), that allows provers to be optimised to those specific instructions, in contrast to general purpose provers, as in Halo2.
- *Full recursion*: In STARKs, a program is typically arithmetised as a matrix, where every row is a constraint. The prover in a STARK zkVM provides a proof for every row in a recursive manner.

A conventional STARK VM runs over a pre-defined IS or a set of fixed opcodes given. Because of the generality of these instruction sets, the number of opcodes in an ISA tends to be small (especially in CairoVM), and thus the circuit size or number of constraints of each opcode is small, too. This implies that the number of instructions that we need to prove and verify for a given program is likely gigantic in such architectures. STARK zkVMs only work with full recursion, that is, they need to run a full verifier circuit inside each iteration, so the computational overhead of proving and verifying every single small instruction render these zkVM designs suboptimal. Folding schemes do not work with any non-homomorphic PCS, so they do not work with STARKs since they use FRI, a hash-based PCS. We cannot apply any proof batching in those zkVMs.

With over five years of engineering work, STARK-based zkVMs are still the most widely deployed type of zkVM.

2.4. IVC-based zkVMs

This is the type of zkVM we are most interested in this report, both due to its novelty and potential. It 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-based zkVM will output a proof that asserts that all states from s_0 to s_{i-i} 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.

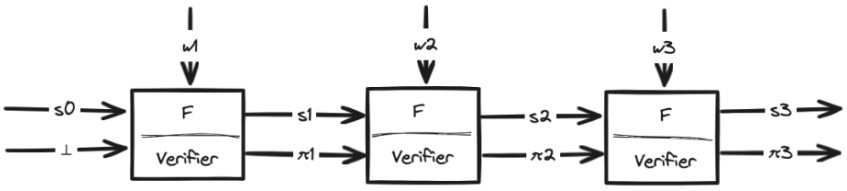

Figure 7. IVC-based zkVMs.

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-based 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 circuit that extends F is computationally expensive since it involves checking openings in a polynomial commitment scheme.

Earlier in our discussion of recursive proof systems, we saw that Halo introduced the notion of accumulation schemes to address the problem of having an expensive verifier in the scheme, in which the computationally expensive part of the verifier algorithm (i.e., the linear time polynomial commitment opening checks) was deferred and later combined. This accumulation of checks opened up the possibilities of suitable SNARKs in an IVC scheme since the performance of the verifier is no longer required to be sublinear, and thus, we can even choose an SNARK with an expensive verifier such as in Bulletproofs [BBB+17]. Because of the Inner Product Argument (IPA) polynomial commitment scheme introduced in Bulletproofs, Halo2 does not need a trusted setup. But this was still not enough to construct a competitive zkVM.

While folding schemes improve the efficiency of IVC and accumulation schemes by going a step further and deferring *all* verifying checks until all proofs are generated, they still incur in some overhead and must be taken into account for designing an optimal zkVM.

The original folding schemes (understood as a common single function F that gets executed repeatedly, deferring all verification checks), 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.

2.5. NIVC-based zkVMs

Fortunately, SuperNova introduced the concept of Non-uniform IVC (NIVC). This innovation renders IVC zkVMs based on folding schemes computationally feasible. While it is common to think of these $\{F_1, \cdots, F_n\}$ as pre-determined instructions (e.g., addition, equality, multiplication, range checks, etc.), they do not have to be fixed. In fact, we can leverage the work of compilers to tailor them to a given program.

NIVC-based zkVMs significantly reduce the hardware requirements for provers, enhance parallel efficiency and reduce circuit sizes. Different instruction circuits can be written without the need to use switches to toggle circuits, reducing circuit size and achieving a "à la carte cost profile". Furthermore, different circuit inputs can be folded together.

NIVC-based zkVMs can benefit from compiler passes. 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 $\{F_1, \cdots, F_n\}$ subprograms.

The compiler must be given a heuristic of this desired optimal balance between circuit size and the number of circuits. For example, an optimal equilibrium might create bounded circuits of 2^{12} gates with only one witness column. That is, a compiler must have an educated guess of how to decompose larger circuits into smaller ones.

2.5.1. 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 $\{F_1, \cdots, F_n\}$ 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 ω_{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 for a given arithmetisation) as it is a compiler problem (i.e., designing the right transformations of a program that we want to efficiently prove its computation).

3. Arithmetisations

An arithmetisation describes how computation is structured to make it easier to prove properties about that computation. While the simplest way of expressing a computation using fields is with arithmetic circuits, circuits are unstructured. This lack of structure makes it challenging to implement SNARKS and motivates the quest for a suitable abstraction or arithmetisation.

When designing circuits, knowing which types of operations are required and how many times they are used in the circuit guides the design of the prover for most proving systems. In other words, the performance of a proof system may be constrained by the chosen arithmetisation. Some representations such as R1CS are more fixed, others like Plonkish are more granular. This granularity offers control and potential optimisation at the expense of the overall complexity of the proving system as a whole.

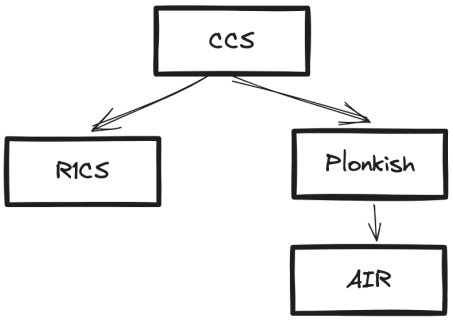

Figure 8. Arithmetisations.

In contrast to a general-purpose proving system, a zkVM with a fixed set

of instructions allows us to tune the proving system to a particular circuit or set of circuits. This restriction opens the door for further optimisation, as we have seen with STARK zkVMs, where each prover is tailored to a particular zkVM.

Recursive schemes offer a much larger room for optimising their provers. Two factors must be taken into account when choosing an arithmetisation for a recursive scheme: flexibility and complexity.

Since we are designing a compiler to zkVMs, this compiler must understand the constraints of the IR or arithmetisation to optimise the prover's efficiency.

3.1. R1CS

Rank-One Constraint System (R1CS) uses constraints of the form $Az \times Bz = Cz$ where A, B, and C are *linear combinations* of variables and constants encoded in witness z.

Folding schemes such as Nova (and its successor SuperNova) only work over R1CS instances, while the HyperNova scheme is defined over CCS. As we will see, CCS is a generalisation of both R1CS and Plonkish. The folding scheme ProtoStar is itself a generalisation of Nova defined over CCS and thus over Plonkish and R1CS.

Compared to other arithmetisations, the main property of R1CS is that it is simple. R1CS-based folding schemes such as Nova achieve most of the benefits of folding, but they cannot have gates. Custom gates are powerful but complex. However, lookups can arguably be used as an alternative to custom gates.

The simplicity of these folding schemes on top of R1CS derives from the simplicity of R1CS. From an engineering perspective, they are likely to be more efficient than Plonkish-based folding schemes on the get-go in a naive implementation, but they have less room for optimisation.

3.2. AIR

Algebraic Intermediate Representation (AIR) is a particularly structured kind of Plonkish circuit, generally associated with STARKs.

AIR is depicted by an execution trace, i.e., a matrix of field elements where each column represents a register and each row denotes a moment in time. The values of registers over time are recorded in the execution trace. To ensure that a computation has been performed correctly, it is necessary to prove that the execution trace satisfies certain properties.

The main two constraints in AIR are transition and boundary constraints. Transition constraints represent the evolution of the trace, that is, that a each time step, a certain relation holds between the current state and the next state.

Boundary constraints assert that a register takes on a certain value at some point in time.

While AIR is the intermediate representation that STARKWARE uses, most SNARKs can be easily tweaked to support both Plonkish and AIR.

3.3. Plonkish

Plonkish is a generalisation of AIR that includes a new column type of selectors.

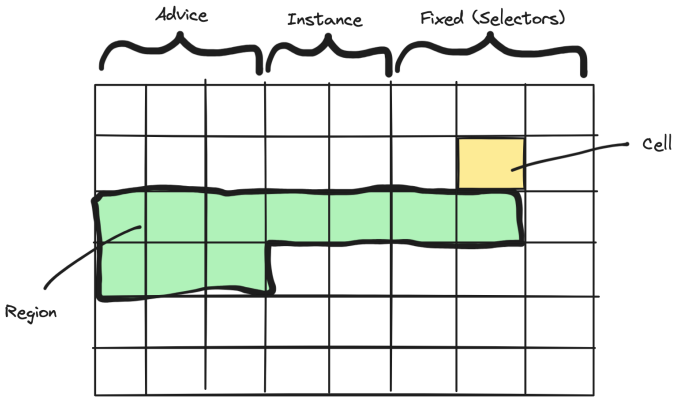

Figure 9. Plonkish arithmetisation.

Both, Plonkish and AIR are conceptualised over the idea of an execution trace. However, Plonkish circuits have selectors, i.e., columns of boolean values that allow the proving system to merge several constraints into a single one, while AIR arithmetisation does not. This makes Plonkish circuits more general.

A Plonkish circuit-satisfiability relation consists of three modular relations, namely:

- A *high-degree gate* relation checking that each custom gate is satisfied.
- A *permutation* relation checking that different gate values are consistent if the same wire connects them.
- A *lookup* relation checking that a subset of gate values belongs to a pre-processed table.

Unlike R1CS, which has a fixed structure, Plonkish circuits are a family of circuits defined in terms of a rectangular matrix of values. We refer to rows, columns, and cells of this matrix with the conventional meanings.

Among other proving systems, Halo2 uses this arithmetisation, although it provides a higher-level API with abstractions such as gates and chips. A gate in Halo2 is a collection of linearly independent constraints which must all be

satisfied in a given row. A chip may apply multiple gates to a region of the trace matrix, and the gates may have been defined with the assumption that they will always be applied in a certain order. This is achieved by using the "rotation" of columns to reference values from neighbouring rows. This level of abstraction unfortunately discards a lot of valuable information about the potential relationship between different gates.

ProtoStar was also designed with Plonk as a target backend. Due to their modular architecture, their folding scheme proved to be flexible enough to be implementable over CCS when it came out.

3.3.1. Formal description

Formally, a Plonkish structure S, $S_{Plonkish} = (m, n, l, t, q, d, e, g, T, s)$, consists of:

- a multivariate polynomial g in t variables where g is expressed as a sum of q monomials and each monomial has a total degree at most d. That is,
$$g(X_1, \cdots, X_t) = \sum_{i=0}^{q} k_i \cdot X_1^{i_1} \cdots X_t^{i_t},$$
where $i_1 + \cdots + i_t \leq d$ for all i.
- a vector of constants called *selectors* $s \in \mathbb{F}^e$.
- a set of m constraints. Each constraint i is specified via a vector T_i of length t, with entries in the range $\{0, \cdots, n + e - 1\}$.
$$T_i := [T_{i_0}, \cdots, T_{i_{t-1}}] \text{ (constraint)}$$
That is, if $j < n - l$, then $T_i[j]$ will be a private input w_j. If $j \geq n - l$ and $j < n$, then $T_i[j]$ will be a private input x_j. Otherwise, $T_i[j]$ will be a selector.
- size bounds $m, n, l, t, q, d, e \in \mathbb{N}$:
  - m is the number of constraints (i.e., rows in the Plonkish matrix representation).
  - n is the number of private and public inputs (i.e., columns in the Plonkish matrix representation, excluding selectors).
  - l is the number of public inputs ($n - l$ is the number of private inputs).
  - t is the number of variables in the polynomial g.

- q is the number of monomials that compose g.
- d is the maximum degree of g.
- e is the number of selectors.

A Plonkish instance consists of public input $x \in \mathbb{F}^l$. A Plonkish witness consists of a vector $w \in \mathbb{F}^{n-l}$. A Plonkish structure-instance tuple (S, x) is satisfied by a Plonkish witness w if

$$g(z[T_i[1]], \cdots, z[T_i[t]]) = 0 \tag{1}$$

for all $i \in \{0, \cdots, m-1\}$, where $z = (w, x, s) \in \mathbb{F}^{n+e}$.

3.4. CCS

Customisable Constraint System (CCS) [STW23a] is a generalisation of R1CS and Plonkish that enables the use of high-degree gates while not requiring permutation arguments. There are costless reductions from instances of R1CS and Plonkish instances to equivalent CCS instances.

There has recently been a steady effort of abstracting out components in proving systems and thus enabling the decoupling between arithmetisation and backend. CCS is a consequence of these efforts.

However, there is no practical implementation of the CCS arithmetisation. This renders folding schemes such as HyperNova only theoretical.

3.4.1. Formal description

Formally, a CCS instance consists of public input $x \in \mathbb{F}^l$. A CCS witness consists of a vector $w \in \mathbb{F}^{n-l-1}$. A CCS structure-instance tuple (S, x) is satisfied by a CCS witness w if

$$\sum_{i=0}^{q-1} c_i \cdot \circ_{j \in S_i} M_j \cdot z = \mathbf{0} \tag{2}$$

where $z = (w, 1, x) \in \mathbb{F}^n$.

$M_j \cdot z$ denotes matrix-vector multiplication, $\circ$ denotes the Hadamard (entry-wise) product between vectors, and 0 is an m-sized vector with entries equal to the additive identity in $\mathbb{F}$.

Expanded, this equation looks like:

$$c_0 \cdot \overbrace{(M_{j_0} \cdot z \circ \cdots \circ M_{j_{|S_0|-1}} \cdot z)}^{j_i \in S_0} + \cdots + c_{q-1} \cdot \overbrace{(M_{j_0} \cdot z \circ \cdots \circ M_{j_{|S_{q-1}|-1}} \cdot z)}^{j_i \in S_{q-1}} = \mathbf{0}$$

A CCS structure S consists of:

- a sequence of matrices $M_0, \cdots, M_{t-1} \in \mathbb{F}^{m \times n}$ with at most $\Omega(max(m,n))$ non-zero entries in total (think of these matrices as a generalisation of the A, B, and C matrices in R1CS that encode the constraints).
- a sequence of q multisets $[S_0, \cdots, S_{q-1}]$, where an element in each multiset is from the domain $0, \cdots, t-1$ and the cardinality of each multiset is at most d (think of S_i as containing the pointers to the matrices M_i for each of the addends in the CCS equation).
- a sequence of q constants $[c_0, \cdots, c_{q-1}]$, where each constant is from $\mathbb{F}$
- size bounds $m, n, N, l, t, q, d \in N$ where $n \geq l$
  - m is the number of constraints (i.e., rows in the Plonkish matrix representation)
  - n is the number of private and public inputs (i.e., columns in the Plonkish matrix representation, excluding selectors)
  - N is the total number of non-zero entries in $M_0, \cdots, M_{t-1}$
  - l is the number of public inputs (thus $n - l$ is the number of private inputs)
  - t is the number of M matrices. (e.g., in R1CS, which can be seen as an instance of CCS, there are 3 M matrices: A, B and C)
  - q is the number of addends in the CCS equation (i.e., $q = |\{S_i\}|$)
  - d is the upper bound of the cardinality of each S_i

3.4.2. Representing R1CS in CCS

As an example, the R1CS equation, $(A \cdot z) \circ (B \cdot z) - C \cdot z = 0$ can be represented with CCS as $S_{CCS} = (n, m, N, l, t, q, d, [M_0, M_1, M_2], [S_1, S_2], [c_1, c_2])$, where m, n, N, l are from R1CS and $t = 3$, $q = 2$, $d = 2$, $M_0 = A$, $M_1 = B$, $M_2 = C$, $S_1 = \{0, 1\}$, $S_2 = \{2\}$, $c_0 = 1$, $c_1 = -1$. The relation then becomes:

$$1 \cdot \overbrace{(A \cdot z \circ B \cdot z)}^{S_0=\{0,1\}} + (-1) \cdot \overbrace{(C \cdot z)}^{S_1=\{2\}} = 0$$

3.4.3. Representing Plonkish in CCS

How does the structure of CCS compare to Plonkish?

$S_{Plonkish} = (m, n, l, t, q, d, e, g, T, s)$

$S_{CCS} = (m, n, N, l, t, q, d, [M_0, \cdots, M_{t-1}], [S_0, \cdots, S_{q-1}], [c_0, \cdots, c_{q-1}])$

To derive $M_0, \cdots, M_{t-1}$ in S_{CCS} from $S_{Plonkish}$, each polynomial constraint in $S_{Plonkish}$ corresponds to some row in M_i, so, it suffices to specify how the i-th row of these matrices is set.

For all $j \in \{0, 1, \cdots, t-1\}$, let $k_j = T_i[j]$ (recall that t is the number of variables in g and T_i represents a constraint in vector form). So k_j is one of the values in a constraint.

If $k_j \geq n$ (i.e., if k_j is greater than the number of public and private inputs - it points to a selector), then we set $M_j[i][0] = s[k_j - n]$. Otherwise, we set $M_j[i][k_j] = 1$. We set $S_{CCS}.N$ to be the total number of non-zero entries in $M_0, \cdots, M_{t-1}$.

Each c_i in $[c_0, \cdots, c_{q-1}]$ is the coefficient of the i-th monomial of g. For $i \in \{0, 1, \cdots, q-1\}$, if the i-th monomial contains a variable j, where $j \in \{0, 1, \cdots, t-1\}$, add j to multiset S_i with multiplicity equal to the degree of the variable.

4. zkVM Compilers

4.1. Motivation

While a zkVM is *not* a compiler, any end-to-end architecture for proving computation of a program likely involves the design of a compiler with that of a zkVM. A zkVM is often regarded as "something" that can prove the computation of a generic CPU program. Since a program is written in a high-level language, this seems to imply that a compiler is a component in a zkVM, but it is not. For a given program, a compiler may output different circuits for the same instruction set of a zkVM or, more interestingly, generate different instruction sets or zkVMs that are optimised for that compiled program.

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 computing such subset of circuits that gets executed from a given input.

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

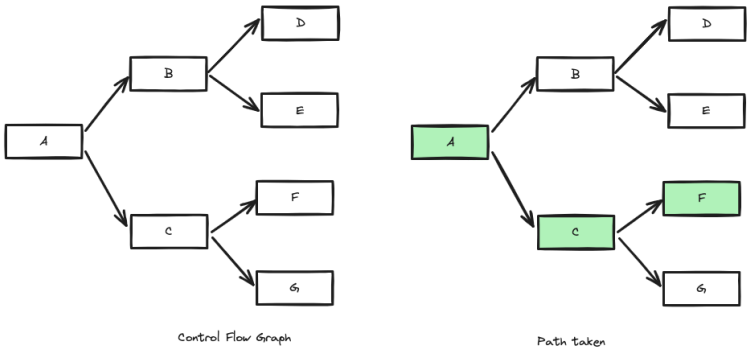

Figure 10. Control Flow Graph.

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 will be the focus of this report.

In fact, we are not designing a zkVM, but a compiler to a zkVM. Whatever zkVM will be suitable for a given program will be determined at compile time. Another way of seeing this, is that the set of instructions $\{F_1, \cdots, F_N\}$ of the zkVM will be set *dynamically* at compile time. A question may arise: what is really a zkVM? Does a different instruction set render a different zkVM, or can the same zkVM hold different instruction sets?

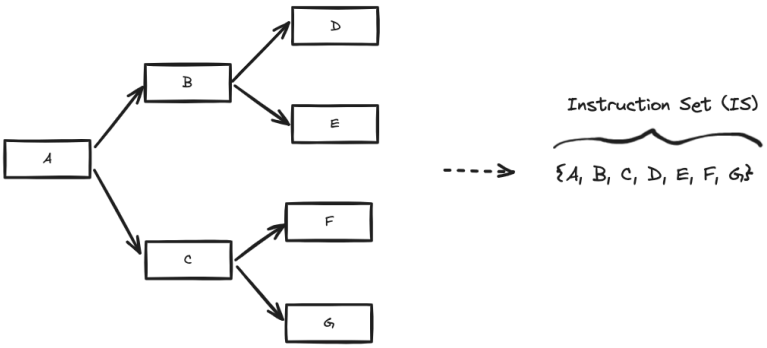

Figure 11. Dynamic Instruction Set.

We want to design a compiler that leverages the state-of-the-art work on folding schemes, thus outputting a NIVC-based zkVM.

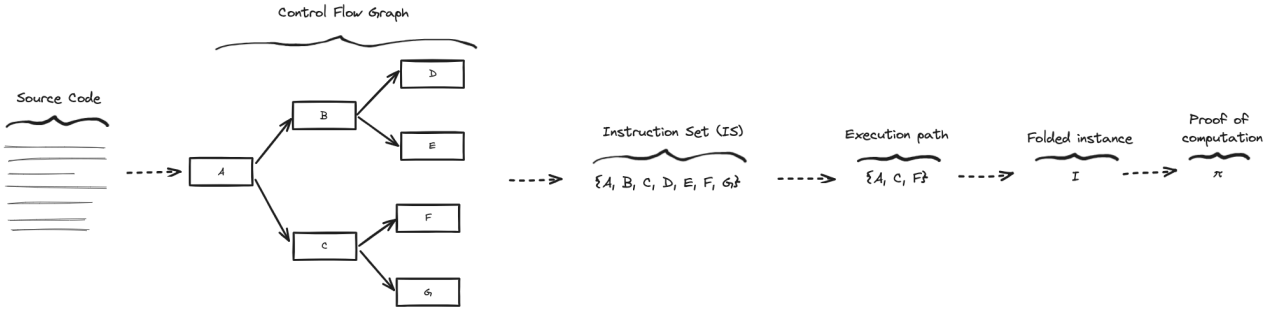

Figure 12. Compiling to zkVMs.

In the design of a zkVM compiler, it is important to strike a balance between the size of each step function F and the overhead induced by recursion, accumulation or folding. On one end of this spectrum, we have the whole program being one big circuit; on the other end, we fold of all primitive operations such as addition, equality or multiplication. There is no folding in the former case, rendering in the majority of cases a circuit that exceeds the current memory limits 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-based zkVMs can benefit from compiler passes. While it is common to *fix* the set of functions $\{F_1, \cdots, F_n\}$ 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 $\{F_1, \cdots, F_n\}$ 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 zkVM compiler approach is not commonly seen, since most projects are focused on building on top of fixed instruction sets (IS) such as RISC-V or one that is compatible with the EVM. However, we do find this integral frontend-backend approach in projects like "Jolt: SNARKs for Virtual Machines via Lookups" [AST23] and "Unlocking the lookup singularity with Lasso" [STW23b].

4.2. Example

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 Fibonnacci 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 = fibonnacci(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 13. The `multi_algorithms` example.

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

4.2.1. Monolithic circuits

This would be a direct compilation into a circuit for an arithmetisation (e.g., Halo2 Plonkish).

Figure 14. 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 all the unused branches), and memory consumption is also often linear, this approach turns out to be inefficient for large applications.

4.2.2. Fixed instruction sets

This is the approach taken by STARK zkVMs: the set of (primitive) instructions is pre-determined, that is, it is independent of the program. Underlying any computation on finite fields, there are primitives operations such as addition or multiplication, as well as other common and more complex operations such as range checks.

The prover do not prove the whole program at once, but proves each of these primitive operations, one at a time. In the case of STARKs, they use full recursion.

Figure 15. 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.

4.2.3. Dynamic instruction sets

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

In contrast, using some heuristic, a compiler may determine the set of instructions for a given program. For our 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.

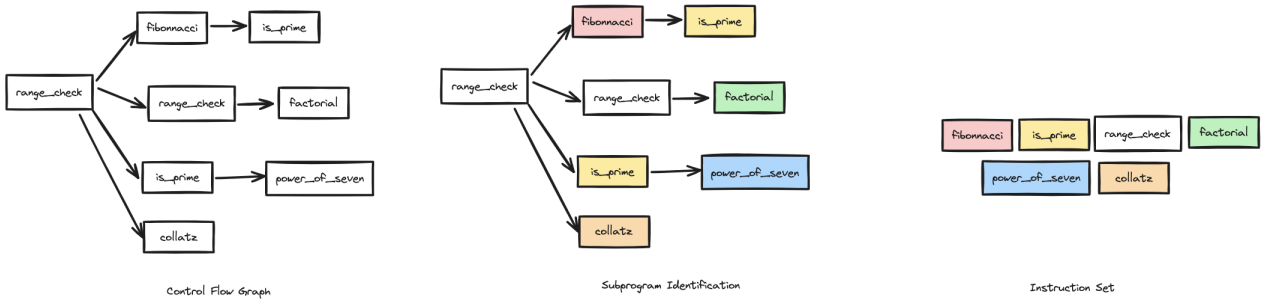

Figure 16. Dynamic instruction set.

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. This means they can be treated as separate "opcodes" in this dynamic set of instructions. They are not too complex but not too small, and are used multiple times.

The different ways of splitting a program into subprograms that serve as an instruction set for an NIVC zkVM is more an art than it is a science.

4.3. Design Goals

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

- **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.
- **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 will 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.
- **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.
- **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.

4.4. Smart Block Generation

4.4.1. Circuits as lookup tables (Jolt and Lasso)

Just One Lookup Table (Jolt) [AST23] is an exciting theoretical innovation in the zkVM space, despite it not having a stable open-source implementation yet. It springs from the realisation that the key property that makes an instruction "SNARK-friendly" is *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 [STW23b], 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 an 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 two things occur. First, all generated circuit blocks are decomposable. Second, the generated blocks or instruc-

tions are lookup tables.

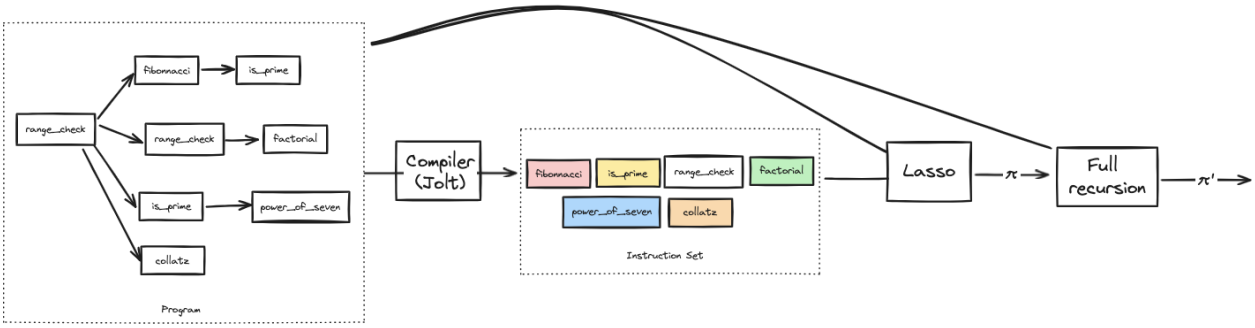

Figure 17. Compiler + Jolt pipeline.

The line between such a 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:

- *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, Lasso is applied 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}.

- *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.

- *LDE-structure.* LDE is short for a technical notion called a Low-Degree Extension polynomial. A Multilinear 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 with other lookup arguments since *Lasso pays only for the table elements it actually accesses.*

- *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.

- *Auditing.* Jolt encourages zkVM designers to simply specify the evaluation table of each instruction. It is easier to audit lookup arguments than many lines of hand-coded constraints.

- *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.

4.5. Fast Provers, Small Proofs, Fast Verifiers

4.5.1. 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 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 Knowl-

edge (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 *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.

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 2. Comparison of Elliptic Curve SNARKs and STARKish Protocols.

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 suffer 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.

In other words, for any sort of uniform computation, the prover's time of STARKish protocols seems unbeatable by elliptic curve SNARKs. At this time, the benefits of elliptic curve SNARKs 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 have 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 is $2^{31} - 1$, 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.

4.5.2. Smallest Fields (Binius)

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

Table 3. Comparison of Prime Fields in Plonky2, Plonky3, and 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 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 "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 $\mathbb{F}_{2^4}$, 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.

4.5.3. 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.

	PROTOSTAR	**HyperNova**	**SuperNova**
Language	Degree d Plonk/CCS	Degree d CCS	R1CS (degree 2)
Non-uniform	yes	no	yes
P native	$\|w\|G$ $O(\|w\|d\log^2 d)F$	$\|w\|G$ $O(\|w\|\log^2 d)F$	$\|w\|G$
extra P native	$O(\|\ell_k\|)G$	$O(T)F$	N/A
w/ lookup			
P recursive	$3G$ $(d+O(1))H + H_{in}$ $(d+O(1))F$	$1G$ $d\log nH + H_{in}$ $O(d\log n)F$	$2G$ $H_{in} + O_\lambda(1)H) + 1H_G$
extra P recursive	$1H$	$O(\log T)H$	N/A
w/ lookup		$O(\ell_k \log T)F$	

Table 4. Comparison among IVCs (credit: Protostar paper).

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

4.6. Modularity

4.6.1. 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 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.

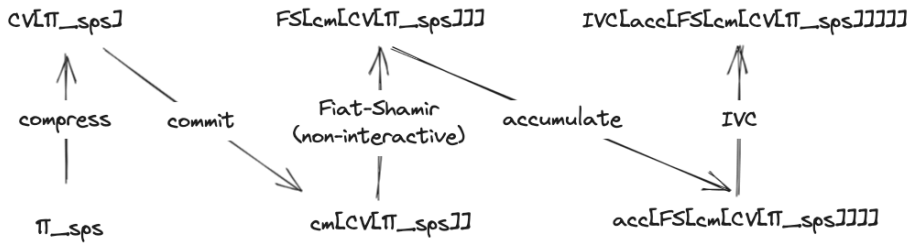

Figure 18. ProtoStar progressive blocks.

The diagram above can be read as follows:

- We start from a special-sound protocol Π_{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 Π_{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 Π_{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.

4.7. Function privacy

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

Function privacy is a property of some execution environments such as Taiga or Zexe [BCG+18] of hiding which function or application is called at any

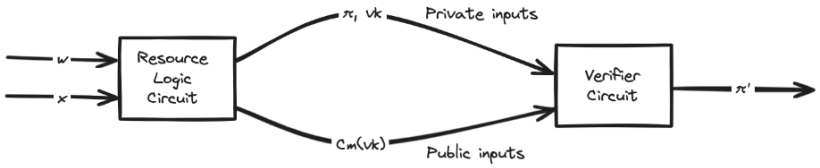

Figure 19. Full recursion in Taiga

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 is 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*, this circuit encodes the verifier algorithm and represents the recursive step that achieves function privacy. It takes both the proof π 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 π' 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 do not contain the verifying key of the resource logic, but its commitment and a proof that the logic is satisfied, hence achieving function privacy. It is 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.

5. Conclusion

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

While it is common to fix a set of functions $\{F_1, \cdots, F_n\}$ to a basic instruction set (like STARKish zkVMs do), it doesn't have to be so. A compiler can use the information it gathers from a program to create tailored step functions or instruction set (IS) $\{F_i\}$ at compile time. These fundamental instructions no longer need to be small enough opcodes that satisfy all programs, but subsets of the compiled program.

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

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

Some of the main questions we want to answer are:

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

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

5.1. Compiling to STARKish zkVMs

STARKish zkVMs use full recursion. Their polynomial commitment schemes are based on hash functions and error-correcting codes. Since these SNARKs are not based on elliptic curves, they are able to use smaller fields. This makes the prover much faster since the polynomial commitment scheme is

generally the bottleneck for any SNARK and committing to small fields is more efficient. The work on towers of binary fields pioneered by Binius promises to improve dramatically the efficiency of these STARKish protocols. In a recent presentation, they run a benchmark for committing to a polynomial with 2^{28} coefficients.

	1-bit elements	8-bit elements	32-bit elements	64-bit elements
Hyrax BN254 (Lasso)	85.02 s	118.6 s	286.9 s	605.5 s
FRI, Poseidon (Plonky2)	98.718 s	98.718 s	98.817 s	98.817 s
FRI, Keccak (Plonky2)	29.36 s	29.36 s	29.36 s	29.36 s
Binius	0.586 s	5.173 s	29.36 s	58.62 s

Table 5. Your caption here.

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

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

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

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

5.2. Compiling to NIVC zkVMs

On the other hand, the polynomial commitment schemes of elliptic-curve-based SNARKs are generally *additively homomorphic* and thus these SNARKs can be folded, i.e., $Com(x) + Com(y) = Com(x + y)$.

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

Elliptic-curve-based SNARKs allow for both full recursion and folding. KZG is the polynomial commitment scheme that outputs the smallest proof, but it requires a trusted setup. IPA, though less efficient (technically is not *succinct*), removes this limitation. SNARKs that use HyperPlonk avoid FFT during proof generation, which brings down memory requirements and makes them more parallelisable.

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

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

However, some of these works are still merely theoretical. Works built solely on CCS such as HyperNova do not have an implementation, which makes them hard to consider from an engineering perspective. Others, such as ProtoStar, are not only the state-of-the-art performance-wise, but are also the most generic and have already Plonkish implementations and adaptations on other systems such as Halo2. In particular, ProtoStar is a generic folding scheme. This means that future works in arithmetisations, polynomial commitment schemes or even folding techniques promise to be easily adopted in ProtoStar with ease.

5.3. Compiling to Jolt zkVMs

(J)ust - (O)ne - (L)ookup -(T)able (i.e., Jolt) is simply a compiler that, given a set of *decomposable* instructions, generates a zkVM that is solely based on lookups. A Jolt zkVM is just a giant lookup table. The proving system that performs these lookups is Lasso.

One of the costs in Lasso is proportional to the number of permutations of the lookup table. If the lookup table is 64-bits this would be 2^{64}. This is too big. If instead we chunk that 4 times, this costs goes down to 2^{16}. Which is entirely practical. The reduction is exponential.

If a compiler is able to provide a set of decomposable instructions to Jolt, then Jolt can do the remaining work and use Lasso for proving. Decomposable means that one lookup into the evaluation table t of an instruction, which has size N , can be answered with a small number of lookups into much smaller tables $t_1, \cdots, t_l$, each of size $N^{1/c}$.

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

In the future, Binius PCS can and likely will work with Jolt and Lasso.

There will be folding with Lasso like lookups possibly aggregating hyper efficient Binius proofs.

As of today, Lasso is stable[1], Jolt is in development[2].

5.4. Final Remarks

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

	Pros	Cons
STARKish zkVMs	Performance of a single instance (fast prover). Maturity	Performance of the scheme as a whole (no folding)
NIVC zkVMs	Performance of the scheme as a whole (folding). Small proofs	Large fields
Jolt zkVMs	Conceptually simple. Benefits from small fields	Experimental

Table 6. Comparison of zkVMs.

Arguably, the "STARKish" approach is not separate from Jolt. Jolt can use any multilinear polynomial commitment scheme. Once the initial implementation of Jolt is done (using curve-based commitments) they will likely replace the commitment scheme with Binius-commitment, expecting at least a 5x speedup from that.

One of the main criticisms of IVC schemes is that doing everything incrementally is limiting. They require to break big computations into tiny chunks, handle each chunk separately and then efficiently combine the results. For example, lookup arguments like Lasso have a non-trivial "fixed cost" that gets amortized over many lookups, and that kind of amortization is not possible unless one operates on a decent-sized chunk "all at once". So IVC schemes currently use different, less efficient lookup arguments than Jolt.

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

As this report suggests, a compiler may overcome the IVC limitation of IVC schemes by generating an instruction set of bigger chunks, tailored to a given program. This will in turn proportionately increase the prover memory requirement. Since IVC naturally supports iterative computations (i.e., computing $F(F(F(\cdots(F(F(x))))))$ for a function F), for computations naturally expressed that way it's hard to beat IVC. Thus a compiler may leverage this property to provide a circuit abstraction of pure functions, moving away from a fully-fledged VM abstraction.

[1] https://github.com/a16z/Lasso.

[2] https://github.com/a16z/Lasso/tree/jolt/jolt-core/src/jolt.

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

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

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

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

6. 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 pp. 23 and 27.)

BBB+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. 13.)

BC23. Benedikt Bünz and Binyi Chen. Protostar: Generic efficient accumulation/folding for special sound protocols. Cryptology ePrint Archive, Paper 2023/620, 2023. https://eprint.iacr.org/2023/620. (cit. on p. 7.)

BCG+18. Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. Zexe: Enabling decentralized private computation. Cryptology

ePrint Archive, Paper 2018/962, 2018. https://eprint.iacr.org/2018/962. (cit. on p. 34.)

BCL+20. Benedikt Bünz, Alessandro Chiesa, William Lin, Pratyush Mishra, and Nicholas Spooner. Proof-carrying data without succinct arguments. Cryptology ePrint Archive, Paper 2020/1618, 2020. https://eprint.iacr.org/2020/1618. (cit. on p. 6.)

BCMS20. Benedikt Bünz, Alessandro Chiesa, Pratyush Mishra, and Nicholas Spooner. Proof-carrying data from accumulation schemes. Cryptology ePrint Archive, Paper 2020/499, 2020. https://eprint.iacr.org/2020/499. (cit. on p. 5.)

BDFG20. Dan Boneh, Justin Drake, Ben Fisch, and Ariel Gabizon. Halo infinite: Recursive zk-snarks from any additive polynomial commitment scheme. Cryptology ePrint Archive, Paper 2020/1536, 2020. https://eprint.iacr.org/2020/1536. (cit. on p. 5.)

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. 5.)

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. 31.)

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. 31.)

GLS+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 pp. 10 and 32.)

Gro16. Jens Groth. On the size of pairing-based non-interactive arguments. Cryptology ePrint Archive, Paper 2016/260, 2016. https://eprint.iacr.org/2016/260. (cit. on p. 5.)

GWC19. Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Paper 2019/953, 2019. https://eprint.iacr.org/2019/953. (cit. on p. 5.)

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.)

KS23a. Abhiram Kothapalli and Srinath Setty. Cyclefold: Folding-scheme-based recursive arguments over a cycle of elliptic curves. Cryptology ePrint Archive, Paper 2023/1192, 2023. https://eprint.iacr.org/2023/1192. (cit. on p. 9.)

KS23b. Abhiram Kothapalli and Srinath Setty. Hypernova: Recursive arguments for customizable constraint systems. Cryptology ePrint Archive, Paper 2023/573, 2023. https://eprint.iacr.org/2023/573. (cit. on p. 7.)

KST21. Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. Cryptology ePrint Archive, Paper 2021/370, 2021. https://eprint.iacr.org/2021/370. (cit. on p. 7.)

KZG10. Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications, 2010. https://www.iacr.org/archive/asiacrypt2010/6477178/6477178.pdf. (cit. on pp. 5 and 6.)

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. 30.)

STW23a. Srinath Setty, Justin Thaler, and Riad Wahby. Customizable constraint systems for succinct arguments. Cryptology ePrint Archive, Paper 2023/552, 2023. https://eprint.iacr.org/2023/552. (cit. on pp. 7 and 19.)

STW23b. 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 pp. 23 and 27.)

Val08. Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. IACR, 2008. (cit. on p. 5.)
ZGGX23. Tianyu Zheng, Shang Gao, Yu Guo, and Bin Xiao. Kilonova: Non-uniform pcd with zero-knowledge property from generic folding schemes. Cryptology ePrint Archive, Paper 2023/1579, 2023. https://eprint.iacr.org/2023/1579. (cit. on p. 8.)