This is a continuation of Schemes for recursive proof composition and part of the Compiling to NIVC-based ZKVMs report. Feedback is welcome

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 a 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 asVamp-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 we not only prove that the state transition is correct, but also that all state transitions are correct from genesis up to the current state.

[

Screenshot 2024-01-16 at 21.59.59

1024×298 27.5 KB

](https://europe1.discourse-cdn.com/standard20/uploads/anoma1/original/1X/189f6e5231b9d8546b9b98e58472bf7c74a14507.png)

This modular approach is what recursion and more specifically \textit{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.

ZKVMs

One way to think about a ZKVM is as a proving system that receives an input program and some data and generates a proof of correct execution. 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 prover and verifier agree on it. This gives designers fewer constraints and more choices and trade-offs.

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 tend 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 5 years of engineering work, STARK-based ZKVMs are still the most widely deployed type of ZKVM.

IVC-based ZKVMs

This is the type of ZKVM we are mostly 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 are required to augment F

with the verification circuit that verifies the proof of the previous step.

[

Screenshot 2024-01-16 at 22.00.49

966×240 26.3 KB

](https://europe1.discourse-cdn.com/standard20/uploads/anoma1/original/1X/52caf6e6c4a00307f1d6e610c8752628e4d415f9.png)

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 do proofs for long computations with relatively little memory by splitting it 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 \textit{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.

As we've seen earlier in our discussion of recursive proof systems, 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) were 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 a SNARK with an expensive verifier such as in Bulletproofs. Because of the IPA polynomial commitment scheme introduced in Bulletprooofs, Halo2 doesn't 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 \textit{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.

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,..., F_n }

as pre-determined instructions (e.g. addition, equality, multiplication, range checks, etc.), they don't 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, ..., F_n }

subprograms.

The compiler must be given a heuristic of this desired optimal balance between circuit size and number of circuits. For example, an optimal equilibrium might be creating 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.

NIVC ZKVMs as 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,...,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}, ω_{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.

Conclusion

So, 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 compilers problem (i.e. designing the right transformations of a program that we want to efficiently prove its computation).

We’ll discuss compilers to ZKVMs next.