I have implemented an early prototype of a zero-knowledge VM based on the ideas fromA sketch for a STARK-based VM post. The project repo is here.

The VM consumes a program and inputs for the program, executes the program against these inputs, and returns program output together with a STARK proof of program execution. This proof can then be used by anyone to verify that the program was executed correctly without the need for re-executing it or even knowing what the program was.

Performance

The instruction set is still pretty limited, but the prototype is fully functional, and the performance is much better than I expected. I benchmarked the performance using a simple program which calculates n-th Fibonacci number. The results are below (and some more results are here):

Operation Count

Execution time

Execution RAM

Verification time

Proof size

210

150 ms

negligible

2 ms

86 KB

214

2.2 sec

130 MB

3 ms

142 KB

218

44 sec

2.6 GB

3 ms

212 KB

This was run on Intel Core i5-7300U @ 2.60GHz (single thread) with 8 GB of RAM.

One interesting observation: the proof overhead is about 2000x. That is, out of the execution time shown above, about 0.05% is spent on executing the program itself, and 99.95% is spent on generating the proof for it.

I expect the performance to decrease somewhat as more instructions are added to the VM, but at the same time, this is not a particularly well optimized implementation running in a single thread. So, even with new instructions, once optimized for multi-threaded execution, the performance will improve significantly.

For example, once hashing instructions are implemented, I expect the VM to be able to do over 1000 Rescue hashes/second on a 4 core machine.

Example

Here is a example of how to execute a simple program which pushes two numbers onto the stack and computes their sum (Distaff VM is a stack machine):

use distaff::{ ProofOptions, processor, processor::opcodes };

```
// this is our program let program = [ opcodes::PUSH, 1, opcodes::PUSH, 2, opcodes::ADD ];

// let's execute it let (outputs, program_hash, proof) = processor::execute( &program, &[], // we won't initialize the stack with any inputs 1, // a single item form the stack will be returned &ProofOptions::default()); // we'll be using default options

// the output should be 3 assert_eq!(vec![3], outputs);
```

A slightly more sophisticated example of a Fibonacci number calculator is here.

Current limitations/issues

- As I mentioned above, the instruction set is currently very limited. But I'm planning to add many more instructions to manipulate the stack, perform hashing, compare values etc. For some possible future instructions see here.
- Distaff VM has no random access memory - all values live on the stack. I'd like to add a memory module, if it doesn't introduce too much computational overhead.
- To generate program hash I'm using a modified version of Rescue hash function. This modification likely makes the function insecure. An un-modified version can be used, though, it would add some complexity (something I'd like to avoid).
- STARK proof generation is currently single-threaded. Though, I've built a lot of plumbing to make implementing multi-threaded execution simpler in the future.
- I'm using the original FRI algorithm in STARK proofs. I'd like to swap it out for DEEP-FRI.

If you'd like to get involved and help in solving some of the issues (or many others) - let me know. Any contributions and feedback are welcome!