keccak256-gnark

keccak256-gnark is a circuit for the Keccak256 hash function, written in golang using the gnark API.

gnark

gnark is a library for building circuits/constraint systems for use in succinct noninteractive arguments of knowledge (SNARKs), written in golang. We chose to write our Keccak256 circuit in gnark since there is already an implementation in circom by vocdoni, and gnark has a fast backend.

Keccak256

Keccak256 is a hash function that takes as input a bitstring of any length, and outputs a string of 256 bits. More generally, the Keccak algorithm can output a bitstring of any length, which is why "256" is added as a suffix to this version of Keccak, which truncates the output at 256 bits. A full specification of the Keccak algorithm, written in pseudocode by the Keccak Team, can be found here.

Essentially, Keccak is made of two components: a permutation and a sponge construction. The permutations used in the various versions of Keccak are called the Keccak-f family of permutations. These, as well as the sponge construction, depend on parameters called the bitrate r and capacity c. In order to make our circuit compatible with Ethereum, we modeled our circuit on the go-ethereum Keccak256 program. This uses the Keccak-f[1600] permutation, with a rate of 1088 and capacity of 512.

Fortunately, there is already a circuit to perform the Keccak-f[1600] permutation as part of the gnark library. This takes in an array of 25 frontend.Variables, considered as uint64s, and returns an array of 25 frontend.Variables. While the Keccak-f[1600] permutation can be defined at the level of bits, in fact the bits are grouped into a 5x5 grid of lanes, which are 64 bits long each (5x5x64 = 1600): thus the use of 64-bit integers. What remained for us was to build a circuit for the sponge construction, using the Keccak-f[1600] permutation circuit. To support working with 64 bit integers, we copied `uint64api.go` from gnark to a local package becuse it doesn't export functionswe use.

Implementing the Sponge Construction

The sponge construction consists of three phases: 1. Padding 2. Absorbing 3. Squeezing

Padding

In the original Keccak specification, any number of bits can be used as input, called the message. The message must then be padded to a number of bits

which is a multiple of r. In the version of Keccak256 used by Ethereum, the input is actually an array of bytes. The Keccak-f[1600] permutation circuit we used worked on uint64s, or blocks of eight bytes each. In order to closely model Ethereum's Keccak256, we needed to assume that the prover would input their message as an array of bytes. Which meant that for the first phase, we had to do the following: 1. Pad the message with enough bytes to equate to a multiple of 1088 bits, i.e. 136 bytes or 17 uint64s. 2. Convert the message into an array of uint64s, for use in the Keccak-f[1600] permutation circuit.

Absorbing

Once the message is padded and grouped into uint64s, it is chopped into blocks of 17 uint64s each. The first block is XORed with the 17x64 initial segment of the 25x64 sponge, the sponge is permuted using the Keccak-f[1600] circuit, then the second block is XORed with the sponge, the sponge is permuted, and so on. The GNARK API ensures that the appropriate constraints are generated for each XOR and permutation in the sponge construction.

Squeezing

After the absorbing phase is complete, the squeezing phase outputs the first four uint64s (=256 bits).

Why label taking the first four entries as the squeezing phase? In other Keccak implementations with longer output length, the squeezing phase is more complicated and occurs in multiple rounds.

Testing the Circuit

To test that the circuit generated the appropriate constraints, we set up a Groth16 proof using the gnark API, using the examples on https://play.gnark.io/ as a template. We designed tests in `keccak_test.go`, in which our `TestAllBackends` function uses fuzzing.

Install dependencies by running:

```
go get
```

Use

```
go run .
```

to run a test circuit.

To run unit tests, execute

```
go test -v
```

Berkeley RDI ZKP/Web3 Hackathon 2023

This project has been done as a submission to Berkeley RDI ZKP/Web3 Hackathon 2023, zkBridge Track, Category 1 (Circuit), Designated Task 1.8.

While working on this project, we

- Learned about differences in Keccak that Ethereum/BNB uses and SHA3 algorithm
- Got first hand experience with Golang and gnark framework
- Performed unit testing with fuzzing, different backends and serialization tests, e.g. Gnark can instantiate "randomized" witnesses and cross check execution result between constraint system solver
- Learned about optimizing circuits by packing bytes into integers

Future work:

- Compile Keccak256 gnark circuit and execute proving in a browser as WASM module
- Verify a proof on a Solidity and Move smart contracts
- Benchmark against Circom and Halo2 implementations