ZK Circuits Workshop

Outline

- Introduction to zk-SNARK
- Introduction to zk Circuits
- Introduction to Noir
- Exercise

zk-SNARK

The acronym zk-SNARK stands for Zero-Knowledge Succinct Non-Interactive Argument of Knowledge and refers to a proof construction where one can prove possession of certain information, e.g., a secret key, without revealing that information, and without any interaction between the prover and verifier.

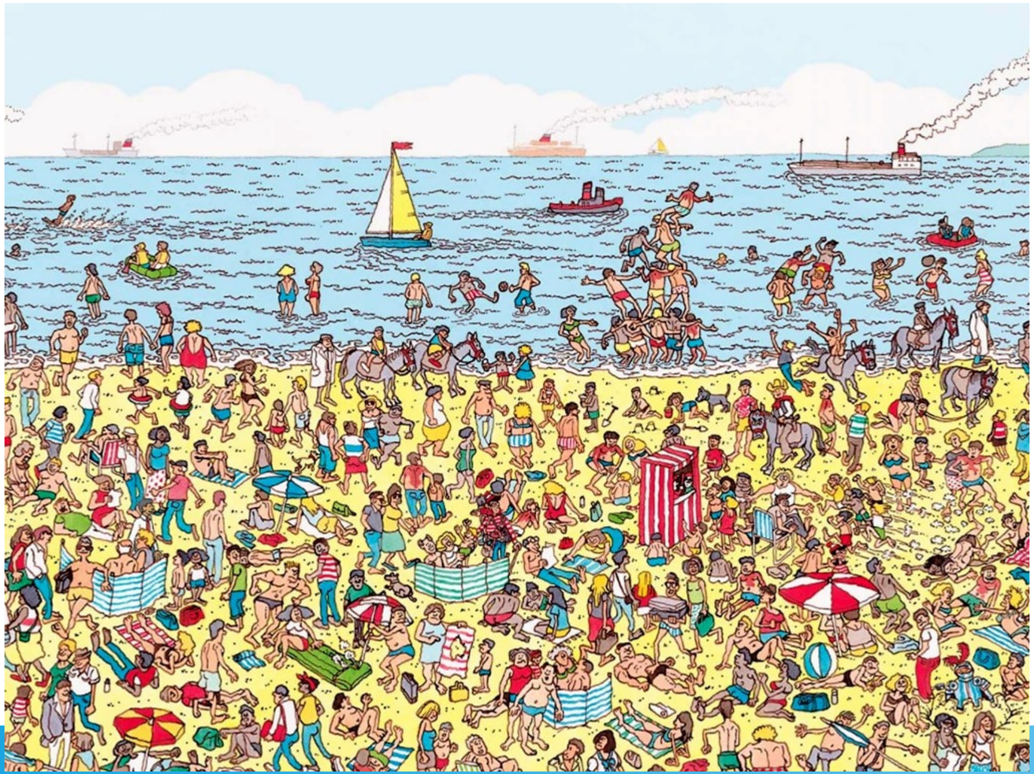
- Source: https://z.cash/learn/what-are-zk-snarks/

zk-SNARK

- Zero Knowledge: The prover convinces the verifier about the truth of a statement without revealing any information.
- Argument of Knowledge: The prover convinces the verifier that they   know the solution to a problem (not only that it exists)
    - Example 1: I convince you that I know the solution to a Sudoku without telling you the solution numbers

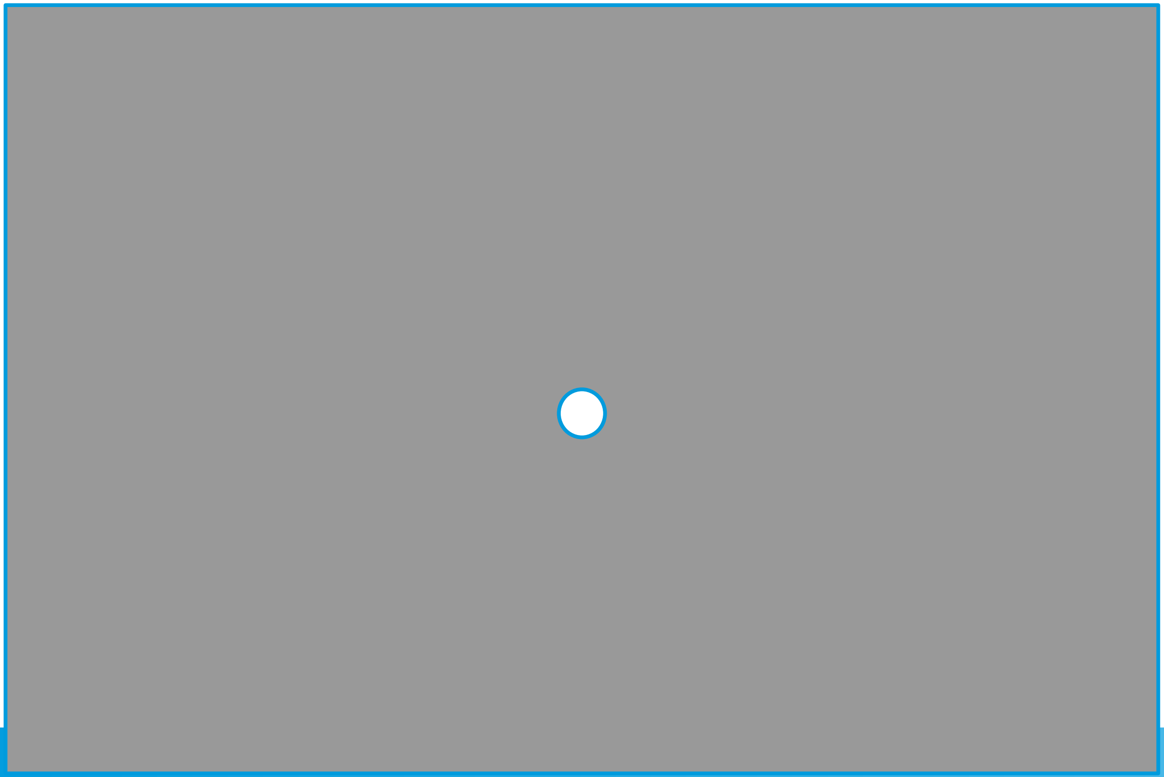
zk-SNARK

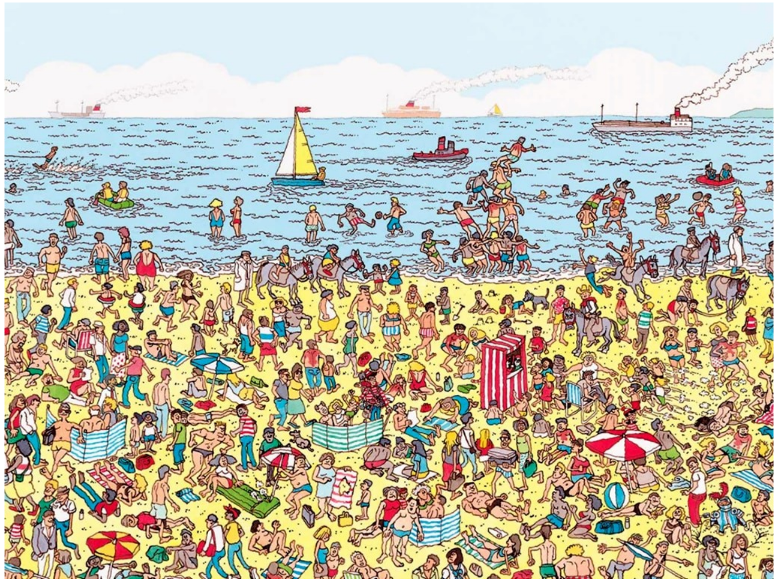
- Example 2: Where's Waldo?



zk-SNARK

- Example 2: Where's Waldo?





zk-SNARK

zk-SNARK

- Non-Interactive: The prover can generate the proof on its own, without interaction with the verifier.
- Succint: The proof can be verified very quickly and is small, even if the problem behind it is big.

zk-SNARK

- More examples:
  - I know the private key to this public key
    - By showing that I can derive the public key from it
  - I know the preimage of this hash
    - By showing that I get this hash by applying the hash function to some value
  - I'm over 18 years old
    - By showing that the government has signed my birth date, and elapsed(current date, my birth date) >= 18 years

Zk Circuits

- A zk-SNARK allows us to convince a 3rd party that we know the solution to a computational problem.
- Moreover we can choose which data we make public and which data is kept private.

Zk Circuits: Flow

- Setup
  1) Choose a problem that can be encoded into a function that returns a boolean.
  2) Encode the function as a circuit (this encoding depends on the zkSNARK system)
  3) Perform a setup using the circuit to generate auxiliary verifying keys that bind the circuit

Zk Circuits: Flow

- Proving & Verifying
  1) Prover has a solution to the problem (some inputs to the function that make it return True)
  2) Prover uses this inputs to generate a proof. This takes a long time.
  3) Prover sends the proof and some of the inputs (the public inputs) to the verifier
  4) Verifier takes the proof and keys from the setup, and runs a check that returns true if the proof is valid

Zk Circuits: Flow

```
func f(publicInputs: []Int, privateInputs: []Int) bool
```

- Prover:
  - Knows \`publicInputs\` and \`privateInputs\` such that:
    - \`f(publicInputs, privateInputs) == true\`
  - Uses these inputs to generate a \`proof\`
- Verifier:
  - Runs a verification: \`verification(proof, verifyingKey, publicInputs)\` which is only \`true\` if the \`proof\` was generated with \`publicInputs\` and \`privateInputs\` such that:
    - \`f(publicInputs, privateInputs) == true\`

Zk Circuits: Aritmetization

- The process of encoding the function is called arithmetization
- Different proof systems use different arithmetizations:
  - Groth16: R1CS
  - Halo2, Plonky2: Plonk(ish)
  - Plonky3: AIR
  - GKR: arithmetic circuits

Zk Circuits: Aritmetization

- Most arithmetizations consist of a way to create a list of polynomial expressions (with different restrictions) that when evaluated on the witness (the “problem” solution) become 0.
- Only* if all those expressions evaluate to 0 the proof verification succeeds.

Zk Circuits: Aritmetization

- Define a list of arithmetic expressions that use `publicInputs` and `privateInputs` as variables, and evaluate to 0 only if `f(publicInputs, privateInputs) == true`
- Example1:

```
func f(x: Int, y: Int, z: Int) bool {
    return x * y == z
}


p(x, y, z) = x * y - z
```

Zk Circuits: Aritmetization

- Example2:

```
func f(opt: bool, x: Int, y: Int, z: Int) bool {
    if (opt) {
        return x * y == z
    } else {
        return x + y == z
    }
}

p(opt, x, y, z) = (1-opt)*(x + y - z) + opt*(x * y - z)
```

Zk Circuits: Aritmetization

- Example2:

```
func f(opt: bool, x: Int, y: Int, z: Int) bool {
    if (opt) {
        return x * y == z
    } else {
        return x + y == z
    }
}


p(opt, x, y, z) = (1-opt)*(x + y - z) + opt*(x * y - z)
```

Zk Circuits: Difference from CPU execution

- In a CPU we have a set of instructions that take variable time, and in each cycle we run one instruction.
  - When we have an if/else the execution only follows one path
- In an arithmetic circuit we don't have cycles! We must "flatten" all the possible execution paths, and "run" them all at the same time and discard the paths that we don't need by multipying by 0.

Zk Circuits: Difference from CPU execution

- Example3:

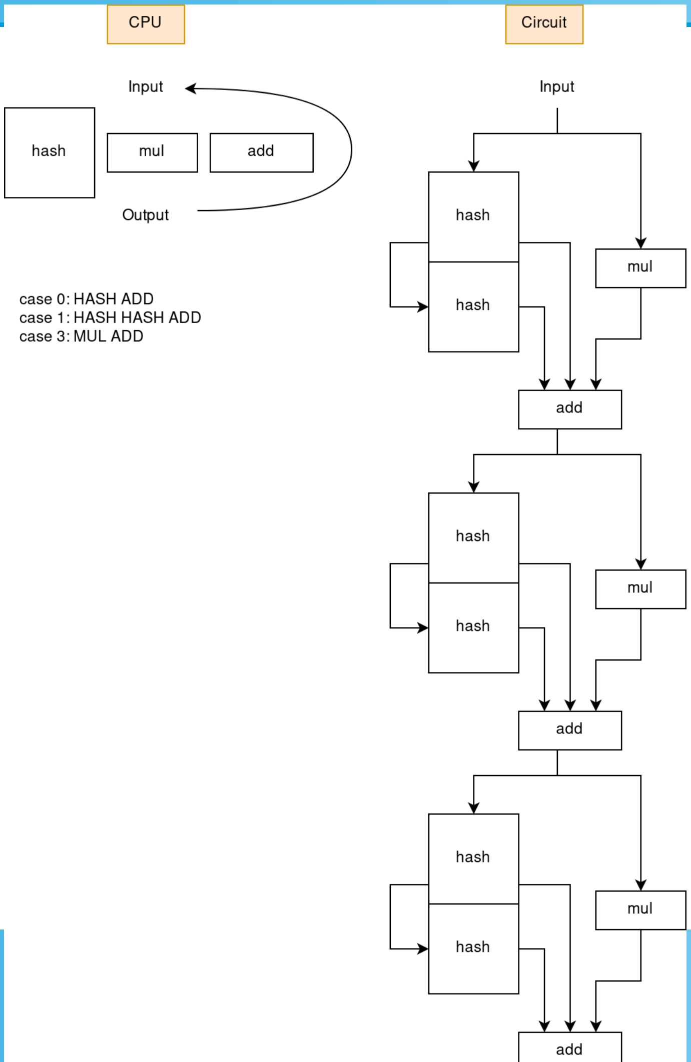
```
func f(n: Int, x: Int, y: Int) bool {
    r := 0
    for (i = 0; i < n; i++) {
        r = r + x
    }
    return r == y
}
```

- For an arithmetic circuit we need to unroll the loop to the max value of `n`!

Zk Circuits: Difference from CPU execution

- Example4:

```
func f(opts: [3]Int, x: Int, y: Int) bool {
    r := 0
    for opt in opts {
        switch (opt) {
        case 0:
            r += hash(x)
        case 1:
            r += hash(hash(x))
        case 2:
            r += x * x
        }
    }
    return r == y
}
```



case 0: HASH ADD
case 1: HASH HASH ADD
case 3: MUL ADD