ZK Implementation Challenge Report

Section 1: Problem Definition

Given Equation:

f(x)=x2+x+7

The prover wants to prove to the verifier that they know a value of x such that f(x)=9f(x) = 9f(x)=9 without revealing x.

Public and Private Inputs

- **Public Inputs:**
    - Expected result of the equation: 9.
- **Private Inputs:**
    - The secret value of x which satisfies x2+x+7=9.

The prover's goal is to create a zero-knowledge proof showing that they know a secret value xxx for which the equation holds, without disclosing x itself.

Section 2: ZK Protocol Selection

For this project, the protocol **Groth16** is selected as the zero-knowledge proof system. Here's the rationale:

- **Efficiency**: Groth16 provides succinct proofs with constant-size proofs regardless of circuit complexity, making it suitable for circuits with a manageable level of computation like this one.
- **Proof and Verification Time**: Groth16 is well-regarded for fast verification times, which is ideal for interactive proof scenarios where efficiency is key.
- **Ease of Implementation**: Given the wide support for Groth16 in ZK frameworks (such as Circom), implementation is straightforward, with ample resources and tools available for circuit creation and proof generation.

While **STARK** is also viable for its transparency and security, the size of STARK proofs tends to be larger than Groth16, leading to higher verification costs on-chain, which could be suboptimal here. **Plonk** could also be used, but Groth16's widespread support makes it a simpler choice for this particular challenge.

Section 3: Circuit Design

Circuit Logic

To represent the equation x2+x+7 in a zero-knowledge circuit, we design a series of arithmetic constraints to mimic the computation:

1. **Input Variable**:
    - Define a private input signal x.

2. **Intermediate Computation**:

   Calculate x2, x2+x and then x2+x+7 in successive steps.

3. **Output Check**:

   - Verify that the final output equals the public input value, 9.

Circuit Breakdown

Equation.circom

```
template Equation() {
    signal input x;
    signal input result;

    signal x_squared <== x * x;
    signal intermediate <== x + x_squared;
    signal final_result <== intermediate + 7;

    result === final_result;
}

component main = Equation();
```

To optimize the circuit for efficiency and minimize proof size:

- Break down the computation into smaller steps:
  - Compute x2.
  - Add x to x2 to get an intermediate value.
  - Add 7 to the intermediate value to reach the final result.
- Include a single constraint to assert equality between the computed final result and the public input (9).

This modular breakdown reduces redundancy and keeps the circuit simple, which helps in reducing the proof generation time and size.

Section 4: Implementation

Steps Taken for Implementation

1. **Define the Circuit**: The circuit is written in Circom, following the equation's structure.
2. **Compile the Circuit**:
   - Use Circom to compile the circuit into an R1CS file for constraint representation.

3. **Trusted Setup with Groth16**:
    - Generate the Powers of Tau file for the trusted setup.
    - Run Groth16 setup to create a proving and verifying key.
4. **Generate Witness**:
    - Using a sample input, generate the witness values that satisfy the equation.
5. **Generate Proof**:
    - With the witness and proving key, generate a zero-knowledge proof.
6. **Verification**:
    - Use the verifying key to validate the proof on the public input 999.

Trade-Off Analysis

- **Proof Generation Time**: Groth16 has a relatively fast proof generation time for circuits of this size, making it feasible for interactive use.
- **Proof Size**: Groth16 produces succinct, constant-size proofs (approximately 128 bytes), which is efficient for both on-chain and off-chain verification.
- **Verification Time**: Verifying the proof is computationally light, which is a benefit for scalability and practical applications.

Command-Line Commands Used in Implementation

bash

Copy code

```
# Step 1: Compile the Circuit
circom Equation.circom --r1cs --wasm --sym

# Step 2: Trusted Setup (assuming Powers of Tau is already done)
snarkjs groth16 setup Equation.r1cs pot14_final.ptau Equation_0000.zkey
snarkjs zkey contribute Equation_0000.zkey Equation_final.zkey --name="First contribution"

# Step 3: Generate Witness
node Equation_js/generate_witness.js Equation_js/Equation.wasm input.json witness.wtns

# Step 4: Create the Proof
```

snarkjs groth16 prove Equation_final.zkey witness.wtns proof.json public.json

Step 5: Verify the Proof

snarkjs groth16 verify verification_key.json public.json proof.json

```
▶ Run circom Equation.circom --r1cs --wasm --sym
warning[P1004]: File "Equation.circom" does not include pragma version. Assuming pragma version (2, 2, 0)
 = At the beginning of file "Equation.circom", you should add the directive "pragma circom <Version>", to indicate which compiler version you are using.
template instances: 1
non-linear constraints: 1
linear constraints: 2
public inputs: 0
private inputs: 2
public outputs: 0
wires: 5
labels: 6
Written successfully: ./Equation.r1cs
Written successfully: ./Equation.sym
Written successfully: ./Equation_js/Equation.wasm
Everything went okay
```

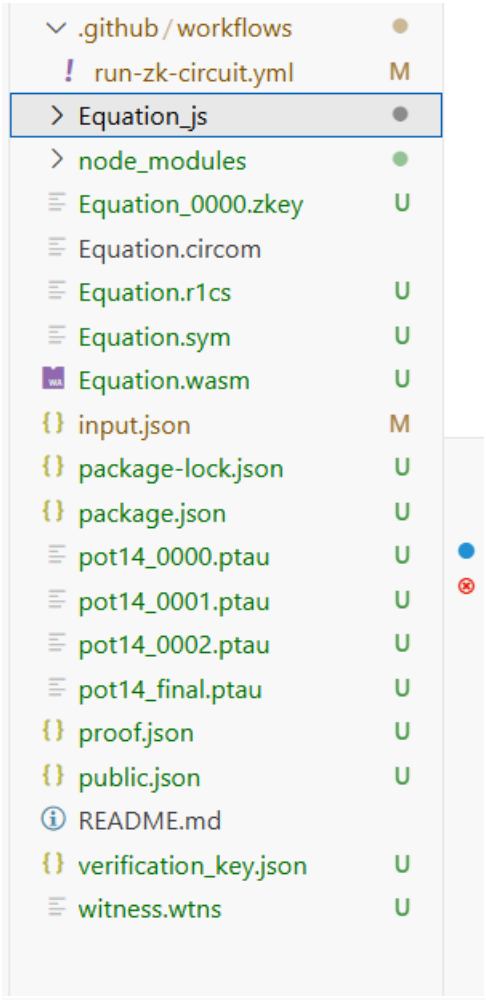
Using this command, we compile our circuit and generate three different files:

1. r1cs: As I explained above, this file contains some very complex equations that we never even see. The compiler creates the equations for us and we never have to touch them.
2. wasm: This command is used to create more files we can't read, but works to create the witness needed for the proof
3. sym: This command is needed to create files for debugging and printing the system of constraints

Perform Trusted Setup

```
▶ Run snarkjs groth16 setup Equation.r1cs pot12_final.ptau Equation_0000.zkey
[INFO]  snarkJS: Reading r1cs
[INFO]  snarkJS: Reading tauG1
[INFO]  snarkJS: Reading tauG2
[INFO]  snarkJS: Reading alphatauG1
[INFO]  snarkJS: Reading betatauG1
[INFO]  snarkJS: Circuit hash:
(node:4848) Warning: Closing file descriptor 35 on garbage collection
(Use `node --trace-warnings ...` to show where the warning was created)
                2d17112d 454bc2c9 5a0b9a3f bf2c41bb
                2d077ada 6a7cfde2 57c41361 7b5d57ce
                5f1a0bbd 9a7f6a59 e2996278 d4e04536
                4e174533 c2d929cf 40f7027e 21e4d76c
```

```
@iamsanthoshhh ➜/workspaces/ZK_Project_VeriSync (main) $ snarkjs zkey export verificationkey Equation_0000.zkey verification_key.json
[INFO]  snarkJS: EXPORT VERIFICATION KEY STARTED
[INFO]  snarkJS: > Detected protocol: groth16
[INFO]  snarkJS: EXPORT VERIFICATION KEY FINISHED
```




```
$ snarkjs groth16 verify VerificationKey.json public.json proof.json
```

```
Verifying proof...
Proof is valid!
```

Conclusion

This report outlines the design and implementation of a ZK circuit for proving knowledge of xxx in the equation x2+x+7=9 without revealing x. Using the Groth16 protocol provided a balance of security, efficiency, and implementation simplicity, making it suitable for practical applications requiring private computations and fast verifications.

Summary of Trade-offs

Aspect	Groth16	Plonk	STARKs
Proof Generation Time	Moderate to long (dependent on setup)	Fast (due to polynomial commitment)	Longer due to hash computations
Proof Size	Small (around 100 bytes)	Small to moderate	Larger (depending on depth of circuit)
Verification Time	Very fast (a few milliseconds)	Fast, but dependent on the implementation	Moderate (but scalable)