网络空间安全创新创业实践 Project3

吴扬

2025 年 7 月 28 日

目录

1 实验环境

处理器	Intel(R) Core(TM) i9-14900HX 2.20 GHz
机带 RAM	16.0 GB (15.6 GB 可用)
Windows 版本	Windows 11 家庭中文版
操作系统版本	22631.4169

2 问题重述

用 circom 实现 poseidon2 哈希算法的电路。

要求:

1. poseidon2 哈希算法参数参考参考文档 1 的 Table1,用 (n,t,d)=(256,3,5) 或 (256,2,5)
2. 电路的公开输入用 poseidon2 哈希值，隐私输入为哈希原象，哈希算法的输入只考虑一个 block 即可。
3. 用 Groth16 算法生成证明

参考文档:

1. poseidon2 哈希算法 https://eprint.iacr.org/2023/323.pdf
2. circom 说明文档 https://docs.circom.io/
3. circom 电路样例 https://github.com/iden3/circomlib

3 实验原理

Poseidon2 是一种高效的哈希函数，专为零知识证明友好型设计。其核心思想是利用有限域上的 S-Box 非线性变换以及矩阵混合来实现高安全性和高性能。Poseidon2 算法参数可参考文献 [1] 的 Table 1，本实验选用 $(n, t, d) = (256, 3, 5)$。在零知识证明场景下，通常需要对哈希函数实现证明电路，以便对哈希前像进行隐私保护。电路设计采用 Circom 语言，输入分为公开输入（哈希值）和隐私输入（哈希前像）。证明系统选用 Groth16，能够在 BN254 曲线下高效生成和验证证明。

4 实现过程

4.1 环境准备

- 安装 Node.js, circom, snarkjs 等工具。

- 准备参考文献和相关 Circom 电路样例。

4.2 电路编写

参考 Poseidon2 算法文档，编写 Poseidon2 哈希电路，主要包括：

- S-Box 模板：实现 x^5 非线性变换。
- AddRoundConstants 模板：每轮加常数。
- ExternalMatrix/InternalMatrix 模板：分别实现外部和内部矩阵混合。
- Poseidon2 主模板：根据参数 (t, R_F, R_P) 实现多轮哈希变换。
- Main 模板：设定公开输入为 hash，隐私输入为 preImage，常量输入为 round_constants 和 M_E。

```
pragma circom 2.1.5;

/*
 * ============================================================
 * Helper Templates (Top Level)
 * ============================================================
 */

// S-Box (x^5)
template Sbox() {
    signal input x;
    signal output out;
    signal x2 <== x * x;
    signal x4 <== x2 * x2;
    out <== x4 * x;
}

// Add Round Constants
template AddRoundConstants(t) {
    signal input state[t];
    signal input C[t];
    signal output out[t];

    for (var i = 0; i < t; i++) {
        out[i] <== state[i] + C[i];
```

```
    }
}

// External Matrix Multiplication (M_E)
// This version explicitly breaks down the computation into simple quadratic constraints.
template ExternalMatrix(t) {
    signal input state[t];
    signal input M[t][t];
    signal output out[t];

    // Pre-declare all intermediate signals needed for the calculation.
    signal products[t][t];
    signal acc[t][t + 1];

    for (var i = 0; i < t; i++) {
        // Initialize accumulator for this row
        acc[i][0] <== 0;
        for (var j = 0; j < t; j++) {
            // 1. Create a simple quadratic constraint for each product.
            // This is of the form: products - state * M = 0
            products[i][j] <== state[j] * M[i][j];

            // 2. Create a simple linear constraint for each accumulation.
            // This is of the form: acc[j+1] - acc[j] - products = 0
            acc[i][j + 1] <== acc[i][j] + products[i][j];
        }
        // 3. The final assignment is a simple linear constraint.
        // This is of the form: out - acc = 0
        out[i] <== acc[i][t];
    }
}


// Internal Matrix Multiplication (M_I)
template InternalMatrix(t) {
```

```
    signal input state[t];
    signal output out[t];

    var sum = 0;
    for (var i = 0; i < t; i++) {
        sum += state[i];
    }

    for (var i = 0; i < t; i++) {
        out[i] <== state[i] + sum;
    }
}


/*
 * ==============================================================
 * Main Poseidon2 Hasher Template
 * ==============================================================
 */
template Poseidon2(t, R_F, R_P) {
    signal input in[t-1];
    signal output out;

    signal input round_constants[R_F + R_P][t];
    signal input M_E[t][t];

    var NUM_ROUNDS = R_F + R_P;

    // --- Pre-declare all components and signals for all rounds ---
    component arc[NUM_ROUNDS];
    component sboxes[R_F * t + R_P]; // Total S-Boxes needed
    component mat_ext[R_F];
    component mat_int[R_P];

    // --- State chaining signals ---
    signal state[NUM_ROUNDS + 1][t];
    signal sbox_outputs[NUM_ROUNDS][t];


```

```
    // --- Initial State ---
    state[0][0] <== 0;
    for (var i = 1; i < t; i++) {
        state[0][i] <== in[i-1];
    }

    var r = 0; // round counter
    var sbox_c = 0; // s-box counter
    var mat_ext_c = 0; // external matrix counter
    var mat_int_c = 0; // internal matrix counter

    // --- Initial External Rounds ---
    for (var i = 0; i < R_F / 2; i++) {
        arc[r] = AddRoundConstants(t);
        arc[r].state <== state[r];
        for(var k=0; k<t; k++) arc[r].C[k] <== round_constants[r][k
           ];

        for (var k = 0; k < t; k++) {
            sboxes[sbox_c] = Sbox();
            sboxes[sbox_c].x <== arc[r].out[k];
            sbox_outputs[r][k] <== sboxes[sbox_c].out;
            sbox_c++;
        }

        mat_ext[mat_ext_c] = ExternalMatrix(t);
        mat_ext[mat_ext_c].state <== sbox_outputs[r];
        for(var k=0; k<t; k++) for(var l=0; l<t; l++) mat_ext[
           mat_ext_c].M[k][l] <== M_E[k][l];

        state[r+1] <== mat_ext[mat_ext_c].out;
        mat_ext_c++;
        r++;
    }

    // --- Internal Rounds ---
    for (var i = 0; i < R_P; i++) {
        arc[r] = AddRoundConstants(t);
        arc[r].state <== state[r];
```

```
137         for(var k=0; k<t; k++) arc[r].C[k] <== round_constants[r][k
                ];
138
139         sboxes[sbox_c] = Sbox();
140         sboxes[sbox_c].x <== arc[r].out[0];
141         sbox_outputs[r][0] <== sboxes[sbox_c].out;
142         sbox_c++;
143         for (var k = 1; k < t; k++) {
144             sbox_outputs[r][k] <== arc[r].out[k];
145         }
146
147         mat_int[mat_int_c] = InternalMatrix(t);
148         mat_int[mat_int_c].state <== sbox_outputs[r];
149         state[r+1] <== mat_int[mat_int_c].out;
150         mat_int_c++;
151         r++;
152     }
153
154     // --- Final External Rounds ---
155     for (var i = 0; i < R_F / 2; i++) {
156         arc[r] = AddRoundConstants(t);
157         arc[r].state <== state[r];
158         for(var k=0; k<t; k++) arc[r].C[k] <== round_constants[r][k
                ];
159
160         for (var k = 0; k < t; k++) {
161             sboxes[sbox_c] = Sbox();
162             sboxes[sbox_c].x <== arc[r].out[k];
163             sbox_outputs[r][k] <== sboxes[sbox_c].out;
164             sbox_c++;
165         }
166
167         mat_ext[mat_ext_c] = ExternalMatrix(t);
168         mat_ext[mat_ext_c].state <== sbox_outputs[r];
169         for(var k=0; k<t; k++) for(var l=0; l<t; l++) mat_ext[
                mat_ext_c].M[k][l] <== M_E[k][l];
170
171         state[r+1] <== mat_ext[mat_ext_c].out;
172         mat_ext_c++;
```

```
        r++;
    }

    out <== state[NUM_ROUNDS][0];
}


/*
 * ============================================================
 * Main Circuit Template
 * ============================================================
 */
template Main() {
    signal input hash;
    signal input preImage[2];
    signal input round_constants[30][3];
    signal input M_E[3][3];

    component hasher = Poseidon2(3, 8, 22);

    hasher.in[0] <== preImage[0];
    hasher.in[1] <== preImage[1];

    for (var i=0; i<30; i++) {
        for (var j=0; j<3; j++) {
            hasher.round_constants[i][j] <== round_constants[i][j];
        }
    }
    for (var i=0; i<3; i++) {
        for (var j=0; j<3; j++) {
            hasher.M_E[i][j] <== M_E[i][j];
        }
    }

    hash === hasher.out;
}

/*
 * ============================================================
```

```
 * Instantiate the Main Component
 * ============================================================
 */
component main { public [ hash ] } = Main();
```

4.3 链下输入与哈希计算

通过 Node.js 脚本，使用 ffjavascript 库链下实现 Poseidon2 哈希算法，并生成电路所需 input.json 文件。输入包括：

- preImage（哈希前像，作为隐私输入）
- hash（哈希值，作为公开输入）
- round_constants 和 M_E（电路常量）

```
const { Scalar } = require("ffjavascript");

// Poseidon2 参数 (t=3, d=5, R_F=8, R_P=22 for BN254)
// 从 Neptune 库中获取: https://github.com/filecoin-project/neptune/blob/master/src/parameters/neptune_params.rs
const {
    BN254_POSEIDON2_C,
    BN254_POSEIDON2_M_E,
} = require('./poseidon2_constants.js');

const t = 3;
const R_F = 8;
const R_P = 22;
const d = 5;
const p = Scalar.fromString("21888242871839275222246405745257275088548364400416034343698204186575808495617");

// 正确处理十六进制字符串的辅助函数
function toBigInt(value) {
    if (typeof value === 'string') {
        // 处理没有0x前缀的十六进制字符串
        if (value.match(/^[0-9a-fA-F]+$/) && !value.startsWith('0x')) {
            return Scalar.fromString('0x' + value);
```

```
        }
        return Scalar.fromString(value);
    }
    return Scalar.e(value);
}

// 将常量和矩阵转换为 Scalar
const C = BN254_POSEIDON2_C.map(row => row.map(c => toBigInt(c)));
const M_E = BN254_POSEIDON2_M_E.map(row => row.map(m => toBigInt(m))
   );

// S-Box函数 - 计算x^d mod p
function sbox(x) {
    // 正确用法: Scalar.mod(Scalar.pow(x, d), p)
    // 而不是 Scalar.pow(x, d).mod(p)
    return Scalar.mod(Scalar.pow(x, d), p);
}

function externalMatrixMul(state) {
    const newState = new Array(t).fill(Scalar.e(0));
    for (let i = 0; i < t; i++) {
        for (let j = 0; j < t; j++) {
            // 正确用法: 累加 Scalar.add(current, newValue)
            newState[i] = Scalar.add(newState[i], Scalar.mul(M_E[i][
               j], state[j]));
        }
        // 取模操作
        newState[i] = Scalar.mod(newState[i], p);
    }
    return newState;
}

function internalMatrixMul(state) {
    let sum = Scalar.e(0);
    for (let i = 0; i < t; i++) {
        sum = Scalar.add(sum, state[i]);
    }
    sum = Scalar.mod(sum, p);

```

```
    return state.map(x => Scalar.mod(Scalar.add(x, sum), p));
}

function poseidon2(inputs) {
    if (inputs.length !== t - 1) {
        throw new Error(`Expected ${t - 1} inputs, got ${inputs.
           length}`);
    }

    // 状态初始化
    let state = [Scalar.e(0)];
    for (let i = 0; i < inputs.length; i++) {
        state.push(Scalar.fromString(inputs[i]));
    }
    let round = 0;

    // 初始外部轮
    for (let i = 0; i < R_F / 2; i++) {
        // AddRoundConstants
        state = state.map((a, j) => Scalar.mod(Scalar.add(a, C[round
           ][j]), p));
        // S-Box
        state = state.map(a => sbox(a));
        // Matrix
        state = externalMatrixMul(state);
        round++;
    }

    // 内部轮
    for (let i = 0; i < R_P; i++) {
        // AddRoundConstants
        state = state.map((a, j) => Scalar.mod(Scalar.add(a, C[round
           ][j]), p));
        // S-Box (on first element)
        state[0] = sbox(state[0]);
        // Matrix
        state = internalMatrixMul(state);
        round++;
    }
```

```
95
96      // 最终外部轮
97      for (let i = 0; i < R_F / 2; i++) {
98          // AddRoundConstants
99          state = state.map((a, j) => Scalar.mod(Scalar.add(a, C[round
                ][j]), p));
100         // S-Box
101         state = state.map(a => sbox(a));
102         // Matrix
103         state = externalMatrixMul(state);
104         round++;
105     }
106
107     return state[0];
108 }
109
110 // 导出函数
111 module.exports = {
112     poseidon2,
113     BN254_POSEIDON2_C,
114     BN254_POSEIDON2_M_E,
115     t,
116     R_F,
117     R_P
118 };
```

4.4 Groth16 证明生成与验证

参考如下流程脚本：

1. 编译电路生成约束系统（.r1cs）、WASM 和符号文件。
2. Trusted Setup（Groth16），生成 zkey 和验证密钥。
3. 生成 input.json，计算 witness。
4. 生成 proof 和 public.json。
5. 验证 proof，输出“Verification successful!”。

```
const fs = require('fs');
const {
    poseidon2,
    BN254_POSEIDON2_C,
    BN254_POSEIDON2_M_E
} = require('./poseidon2.js');

// 定义电路的输入
// 这是隐私输入，即哈希原象
const preImage = [
    "12345",
    "67890"
];

// 使用链下代码计算对应的哈希值
const calculatedHash = poseidon2(preImage);

// 准备要写入 input.json 的完整输入对象
const circuitInputs = {
    // 隐私输入
    "preImage": preImage,
    // 公开输入
    "hash": calculatedHash.toString(),

    // --- 只包含电路中实际定义的输入信号 ---
    "round_constants": BN254_POSEIDON2_C.map(row => row.map(c =>
        BigInt(c).toString())),
    "M_E": BN254_POSEIDON2_M_E.map(row => row.map(m => BigInt(m).
        toString()))
};

// 将输入写入 input.json 文件
fs.writeFileSync(
    'input.json',
    JSON.stringify(circuitInputs, null, 2),
    'utf-8'
);

console.log('Witness input file (input.json) generated successfully
```

```
    .');
console.log(`Pre-image: [${preImage[0]}, ${preImage[1]}]`);
console.log(`Hash: ${calculatedHash.toString()}`);
```

```
#!/bin/bash

# 脚本将在遇到任何错误时退出
set -e

# --- 1. 清理和准备 ---
echo "--- Cleaning up old files ---"
rm -f poseidon2.r1cs poseidon2.sym poseidon2_js/* witness.wtns proof.json public.json input.json *.zkey verification_key.json

# --- 2. 编译电路 ---
echo "--- Compiling circuit (poseidon2.circom) ---"
# 这会生成 poseidon2.r1cs (约束系统) 和 poseidon2_js 目录 (包含WASM和JS代码)
circom poseidon2.circom --r1cs --wasm --sym

# --- 3. 查看电路信息 ---
echo "--- Circuit Info ---"
snarkjs r1cs info poseidon2.r1cs

# --- 4. 可信设置 (Groth16) ---
# 这部分需要一个 Powers of Tau 文件。如果本地没有，snarkjs会尝试下载。
# 对于真实应用，需要一个安全的多方计算仪式。这里我们使用一个现成的文件。
echo "--- Performing trusted setup (Groth16) ---"
if [ ! -f pot14_final.ptau ]; then
    echo "Downloading Powers of Tau file..."
    wget https://hermez.s3-eu-west-1.amazonaws.com/powersOfTau28_hez_final_14.ptau -O pot14_final.ptau
fi

# 4.1 Phase 1: 生成初始 .zkey 文件
snarkjs groth16 setup poseidon2.r1cs pot14_final.ptau poseidon2_0000.zkey
echo "Generated poseidon2_0000.zkey"
```

```
31
32 # 4.2 Phase 2: 贡献 (这里我们只做一个虚拟贡献)
33 snarkjs zkey contribute poseidon2_0000.zkey poseidon2_final.zkey --
       name="Test Contribution" -v
34 echo "Generated poseidon2_final.zkey"
35
36 # 4.3 导出验证密钥
37 snarkjs zkey export verificationkey poseidon2_final.zkey
       verification_key.json
38 echo "Exported verification_key.json"
39
40 # --- 5. 生成见证 (Witness) ---
41 echo "--- Generating witness ---"
42 # 5.1 首先, 用JS脚本生成 input.json
43 node generate_witness.js
44
45 # 5.2 然后, 使用WASM计算器生成 witness.wtns
46 # 进入JS目录执行
47 cd poseidon2_js
48 node generate_witness.js poseidon2.wasm ../input.json ../witness.
       wtns
49 cd ..
50 echo "Generated witness.wtns"
51
52 # --- 6. 生成证明 ---
53 echo "--- Generating proof ---"
54 snarkjs groth16 prove poseidon2_final.zkey witness.wtns proof.json
       public.json
55 echo "Generated proof.json and public.json"
56
57 # --- 7. 验证证明 ---
58 echo "--- Verifying proof ---"
59 snarkjs groth16 verify verification_key.json public.json proof.json
60
61 echo "--- Verification successful! ---"
```

```
sherlock@ubuntu:~/poseidon2_zkp_project$ ./run.sh
--- Cleaning up old files ---
--- Compiling circuit (poseidon2.circom) ---
template instances: 6
non-linear constraints: 210
linear constraints: 228
public inputs: 1
private inputs: 101
public outputs: 0
wires: 540
labels: 1262
Written successfully: ./poseidon2.r1cs
Written successfully: ./poseidon2.sym
Written successfully: ./poseidon2_js/poseidon2.wasm
Everything went okay
--- Circuit Info ---
[INFO]  snarkJS: Curve: bn-128
[INFO]  snarkJS: # of Wires: 540
[INFO]  snarkJS: # of Constraints: 438
[INFO]  snarkJS: # of Private Inputs: 101
[INFO]  snarkJS: # of Public Inputs: 1
[INFO]  snarkJS: # of Labels: 1262
[INFO]  snarkJS: # of Outputs: 0
--- Performing trusted setup (Groth16) ---
[INFO]  snarkJS: Reading r1cs
[INFO]  snarkJS: Reading tauG1
[INFO]  snarkJS: Reading tauG2
[INFO]  snarkJS: Reading alphatauG1
[INFO]  snarkJS: Reading betatauG1
[INFO]  snarkJS: Circuit hash:
                9dcc321b f6576cb1 13520edd d2850459
                48a94439 fc78c39e bcd16ad5 9f03303e
                99b4db1b 137745f3 94802286 ebe3e227
                c9c6ab18 58b57eb7 4386ebb7 d55bf6aa
Generated poseidon2_0000.zkey
Enter a random text. (Entropy): l
[DEBUG] snarkJS: Applying key: L Section: 0/538
[DEBUG] snarkJS: Applying key: H Section: 0/512
[INFO]  snarkJS: Circuit Hash:
                9dcc321b f6576cb1 13520edd d2850459
                48a94439 fc78c39e bcd16ad5 9f03303e
                99b4db1b 137745f3 94802286 ebe3e227
                c9c6ab18 58b57eb7 4386ebb7 d55bf6aa
[INFO]  snarkJS: Contribution Hash:
                bcd9cb00 15be82f4 ab4479ec 86d09bb8
                33db38d1 edeef65d 856b87a2 cdb6199d
                506dcb51 1010344b cc63ce73 e16a2413
                589fed7e 7cc3a4f3 5b6db53e 34aa99a7
Generated poseidon2_final.zkey
[INFO]  snarkJS: EXPORT VERIFICATION KEY STARTED
[INFO]  snarkJS: > Detected protocol: groth16
[INFO]  snarkJS: EXPORT VERIFICATION KEY FINISHED
Exported verification_key.json
--- Generating witness ---
Witness input file (input.json) generated successfully.
Pre-image: [12345, 67890]
Hash: 17544152195513999953959381918720514646712012187417746390733815038317390135028
Generated witness.wtns
--- Generating proof ---
Generated proof.json and public.json
--- Verifying proof ---
[INFO]  snarkJS: OK!
--- Verification successful! ---
```

图 1: 实验终端运行结果

```
--- Generating witness ---
Witness input file (input.json) generated successfully.
Pre-image: [12345, 67890]
Hash: 17544152195513999953959381918720514646712012187417746390733815038317390135028
Generated witness.wtns
--- Generating proof ---
Generated proof.json and public.json
--- Verifying proof ---
[INFO]  snarkJS: OK!
--- Verification successful! ---
```

图 2: 验证结果

5 心得与感悟

本实验让我深入理解了 Poseidon2 哈希电路的结构和零知识证明的流程。Poseidon2 的电路实现对矩阵混合、常量轮次管理等细节要求较高，且参数配置需严格对照文献标准。通过 Circom 和 snarkjs 工具链，可高效完成电路编译、信号生成和证明验证过程。实验过程中遇到的主要问题有：

- ffjavascript 库的 API 使用与原生 BigInt 的区别。
- 电路输入信号与 input.json 字段的严格匹配。
- 常量输入的格式（十六进制前缀、数组结构）需与电路一致。

最终电路通过 Groth16 算法验证成功，证明了电路和链下哈希计算的一致性。Circom 的灵活性和 Groth16 的高效性为实现安全可信的零知识应用提供了强有力支持。

参考文献

[1] Dmitry Khovratovich, et al. *Poseidon2: Improved Hash Functions for Zero Knowledge Proof Systems*. `https://eprint.iacr.org/2023/323.pdf`

[2] Circom Documentation. `https://docs.circom.io/`

[3] Circomlib Examples. `https://github.com/iden3/circomlib`