网络空间安全创新创业实践

Project4

姓　　名: 张治瑞
学　　号: 202200210078
班　　级: 网安 22.1 班
学　　院: 网络空间安全学院

2025 年 8 月 10 日

目录

1 实验环境

处理器	Intel(R) Core(TM) i9-14900HX 2.20 GHz
机载 RAM	16.0 GB (15.6 GB 可用)
Windows 版本	Windows 11

2 实验题目

Project 3: 用 circom 实现 poseidon2 哈希算法的电路

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 实验目的

1. 深入理解 Poseidon2 哈希算法的工作原理和数学基础
2. 掌握 Circom 电路描述语言的使用方法
3. 实现符合 BN254 椭圆曲线参数的 Poseidon2 哈希电路
4. 构建完整的 Groth16 零知识证明系统
5. 验证零知识证明的正确性和可靠性

4 实验原理

4.1 Poseidon2 哈希算法

Poseidon2 是专为零知识证明系统设计的密码学哈希函数，相比于传统的 SHA 系列哈希函数，它在算术电路中具有更高的效率。该算法基于置换网络 (Substitution-Permutation Network) 结构，主要特点如下：

4.1.1 核心参数

- $t = 3$: 状态大小，表示内部状态包含 3 个域元素
- $d = 5$: S-Box 的次数，使用 x^5 作为非线性变换
- $R_F = 8$: 外部轮数，所有位置都应用 S-Box
- $R_P = 22$: 内部轮数，仅对第一个位置应用 S-Box
- $n = 256$: 安全级别，对应 BN254 椭圆曲线的安全强度

4.1.2 算法结构

Poseidon2 算法包含三个主要阶段：

1. **初始外部轮** ($R_F/2 = 4$ 轮): 对所有状态位置应用 S-Box 和外部矩阵 M_E
2. **内部轮** ($R_P = 22$ 轮): 仅对第一个状态位置应用 S-Box，使用简化的内部矩阵
3. **最终外部轮** ($R_F/2 = 4$ 轮): 再次对所有位置应用 S-Box 和外部矩阵 M_E

4.1.3 数学运算

- **S-Box 变换**: $S(x) = x^5 \bmod p$，其中 p 为 BN254 标量域的素数
- **外部矩阵**: $M_E = \begin{pmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 3 \end{pmatrix}$
- **内部矩阵**: $M_I(s) = (s_0 + \sum s_i, s_1 + \sum s_i, s_2 + \sum s_i)$
- **轮常数**: 每轮添加预定义的轮常数以增强安全性

4.2 Circom 电路设计

Circom 是一种用于构造算术电路的领域特定语言，特别适用于零知识证明。本实验中的电路设计遵循以下原则：

4.2.1 约束系统

电路中的每个约束都必须是二次的，即形式为 $A \cdot B - C = 0$。这要求我们将复杂的运算分解为简单的乘法和加法约束。

4.2.2 信号分类

- **公开信号**: `hash` - Poseidon2 哈希值，验证者已知
- **隐私信号**: `preImage[2]` - 哈希原象，证明者私有
- **辅助信号**: 轮常数和矩阵参数

4.3 Groth16 证明系统

Groth16 是目前最高效的零知识证明系统之一，具有恒定大小的证明和快速验证特性：

- **可信设置**: 生成证明密钥和验证密钥
- **证明生成**: 基于见证和约束系统生成简洁证明
- **证明验证**: 验证者使用验证密钥检查证明的有效性

5 代码实现与分析

5.1 核心电路实现

5.1.1 S-Box 模板

Listing 1:

```
template Sbox() {
    signal  input x;
    signal output out;
    signal x2 <== x * x;
    signal x4 <== x2 * x2;
    out <== x4 * x;
}
```

该模板实现 x^5 变换，通过两次平方运算和一次乘法实现，确保所有约束都是二次的。

5.1.2 外部矩阵乘法

Listing 2:

```
template ExternalMatrix(t) {
    signal  input state[t];
    signal  input M[t][t];
    signal output out[t];

    signal products[t][t];
    signal acc[t][t + 1];

    for (var i = 0; i < t; i++) {
        acc[i][0] <== 0;
        for (var j = 0; j < t; j++) {
            products[i][j] <== state[j] * M[i][j];
            acc[i][j + 1] <== acc[i][j] + products[i][j];
        }
        out[i] <== acc[i][t];
    }
}
```

该模板通过显式的中间信号实现矩阵乘法，每个运算都对应一个二次约束，确保电路的正确性。

5.1.3 主要电路结构

Listing 3:

```
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];

    // 状态链接信号
    signal state[NUM_ROUNDS + 1][t];
    signal sbox_outputs[NUM_ROUNDS][t];

    // 初始状态
```

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

    // 轮次处理逻辑...
}
```

5.2 算法实现

5.2.1 JavaScript 参考实现

Listing 4:

```
function poseidon2(inputs) {
    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++) {
        state = state. map((a, j) =>
            Scalar.mod(Scalar.add(a, C[ round][j]), p));
        state = state. map(a => sbox(a));
        state = externalMatrixMul(state);
         round++;
    }

    // 内部轮和最终外部轮...
    return state[0];
}
```

该实现提供了电路外的参考计算，用于生成测试用例和验证电路正确性。

5.3 见证生成

Listing 5:

```
const preImage = ["12345", "67890"];
const calculatedHash = poseidon2(preImage);

const circuitInputs = {
    "preImage": preImage,
    "hash": calculatedHash.toString(),
    "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()))
};
```

见证生成器自动计算哈希值并准备电路所需的所有输入信号。

5.4 链下输入与哈希计算

Listing 6:

```
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
```

```
17 function toBigInt(value) {
18     if (typeof value === 'string') {
19         // 处理没有0x前缀的十六进制字符串
20         if (value.match(/^[0-9a-fA-F]+$/) && !value.startsWith('0x')) {
21             return Scalar.fromString('0x' + value);
22         }
23         return Scalar.fromString(value);
24     }
25     return Scalar.e(value);
26 }
27
28 // 将常量和矩阵转换为 Scalar
29 const C = BN254_POSEIDON2_C. map(row => row. map(c => toBigInt(c)));
30 const M_E = BN254_POSEIDON2_M_E. map(row => row. map(m => toBigInt(m)));
31
32 // S-Box函数 - 计算x^d mod p
33 function sbox(x) {
34     // 正确用法: Scalar.mod(Scalar. pow(x, d), p)
35     // 而不是 Scalar. pow(x, d).mod(p)
36     return Scalar.mod(Scalar. pow(x, d), p);
37 }
38
39 function externalMatrixMul(state) {
40     const newState = new Array(t).fill(Scalar.e(0));
41     for (let i = 0; i < t; i++) {
42         for (let j = 0; j < t; j++) {
43             // 正确用法: 累加 Scalar.add(current, newValue)
44             newState[i] = Scalar.add(newState[i], Scalar.mul(M_E[i][j],
                    state[j]));
45         }
46         // 取模操作
47         newState[i] = Scalar.mod(newState[i], p);
48     }
49     return newState;
50 }
51
52 function internalMatrixMul(state) {
```

```
53      let  sum = Scalar.e(0);
54      for (let i = 0; i < t; i++) {
55           sum = Scalar.add( sum, state[i]);
56      }
57       sum = Scalar.mod( sum, p);
58
59      return state. map(x => Scalar.mod(Scalar.add(x,  sum), p));
60  }
61
62  function poseidon2(inputs) {
63      if (inputs.length !== t - 1) {
64          throw new Error(`Expected ${t - 1} inputs, got ${inputs.length}`);
65      }
66
67      // 状态初始化
68      let state = [Scalar.e(0)];
69      for (let i = 0; i < inputs.length; i++) {
70          state.push(Scalar.fromString(inputs[i]));
71      }
72      let  round = 0;
73
74      // 初始外部轮
75      for (let i = 0; i < R_F / 2; i++) {
76          // AddRoundConstants
77          state = state. map((a, j) => Scalar.mod(Scalar.add(a, C[
                round][j]), p));
78          // S-Box
79          state = state. map(a => sbox(a));
80          // Matrix
81          state = externalMatrixMul(state);
82           round++;
83      }
84
85      // 内部轮
86      for (let i = 0; i < R_P; i++) {
87          // AddRoundConstants
88          state = state. map((a, j) => Scalar.mod(Scalar.add(a, C[
                round][j]), p));
```

```
89          // S-Box (on first element)
90          state[0] = sbox(state[0]);
91          // Matrix
92          state = internalMatrixMul(state);
93           round++;
94      }
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 };
```

6 实验结果与分析

Listing 7:

```
1 #!/bin/bash
```

```
2
3  # 脚本将在遇到任何错误时退出
4   set -e
5
6  # --- 1. 清理和准备 ---
7  echo "--- Cleaning up old files ---"
8  rm -f poseidon2.r1cs poseidon2.sym poseidon2_js/* witness.wtns proof.json
       public.json
       input.json *.zkey verification_key.json
9
10 # --- 2. 编译电路 ---
11 echo "--- Compiling circuit (poseidon2.circom) ---"
12 # 这会生成 poseidon2.r1cs (约束系统) 和 poseidon2_js 目录 (包含WASM和JS代码
       )
13 circom poseidon2.circom --r1cs --wasm --sym
14
15 # --- 3. 查看电路信息 ---
16 echo "--- Circuit Info ---"
17 snarkjs r1cs info poseidon2.r1cs
18
19 # --- 4. 可信设置 (Groth16) ---
20 # 这部分需要一个 Powers of Tau 文件。如果本地没有，snarkjs会尝试下载。
21 # 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。
22 echo "--- Performing trusted setup (Groth16) ---"
23 if [ ! -f pot14_final.ptau ]; then
24     echo "Downloading Powers of Tau file..."
25     wget https://hermez.s3-eu-west-1.amazonaws.com/
           powersOfTau28_hez_final_14.ptau -O pot14_final.ptau
26 fi
27
28 # 4.1 Phase 1: 生成初始 .zkey 文件
29 snarkjs groth16 setup poseidon2.r1cs pot14_final.ptau poseidon2_0000.zkey
30 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! ---"
```

```
--- 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

图 2

6.1 电路编译结果

根据实验截图显示的编译信息：

- **模板实例**: 6 个
- **非线性约束**: 210 个
- **线性约束**: 228 个
- **公开输入**: 1 个（哈希值）
- **隐私输入**: 101 个
- **信号线**: 540 条
- **标签**: 1262 个

6.2 电路规模分析

1. **约束复杂度**: 总计 438 个约束，其中 210 个非线性约束主要来自 S-Box 运算
2. **信号效率**: 540 条信号线合理地实现了 30 轮 Poseidon2 运算
3. **隐私保护**: 101 个隐私输入包含原象和所有中间计算状态

6.3 可信设置结果

实验成功完成了 Groth16 的两阶段可信设置：

1. **Phase 1**: 使用 Powers of Tau 生成初始参数
2. **Phase 2**: 电路特定的参数生成和贡献
3. **密钥导出**: 成功生成验证密钥

6.4 证明生成与验证

实验数据显示：

- **输入**: preImage = [12345, 67890]
- **哈希输出**: 17544152195513999953959538191872051464671201218741774639073381503831739013
- **见证生成**: 成功
- **证明生成**: 成功生成 proof.json 和 public.json
- **验证结果**: 验证成功

6.5 性能评估

表 1: 电路性能指标

指标	数值	评价
总约束数	438	高效
非线性约束比例	47.9%	合理
S-Box 使用数	42 个	符合理论值
矩阵运算次数	30 次	精确匹配

7 实验心得与感悟

7.1 技术收获

1. **算法理解**: 通过实现 Poseidon2，深入理解了面向零知识证明的哈希函数设计原理，特别是如何在保证安全性的同时最小化电路复杂度。
2. **电路设计**: 掌握了 Circom 的核心概念，包括信号、约束、模板等，学会了如何将数学运算转化为二次约束系统。
3. **系统集成**: 构建了从电路编译到证明验证的完整工具链，体验了现代零知识证明系统的工程实践。

7.2 挑战与解决

1. **约束优化**: 初期实现中约束数量过多，通过重新设计 ExternalMatrix 模板，使用显式中间信号减少了冗余约束。

2. **常数准确性**: Poseidon2 的安全性高度依赖于轮常数的正确性，通过从 Neptune 库提取标准参数确保了实现的可靠性。

3. **数值精度**: BN254 域的大数运算需要特别小心，使用 ffjavascript 库的 Scalar 模块确保了数值计算的准确性。

7.3 应用前景

1. **隐私计算**: Poseidon2 哈希在隐私保护的身份验证、投票系统等场景中有广阔应用前景。

2. **区块链技术**: 作为 zk-SNARK 的基础组件，可用于构建高效的 Layer 2 扩容方案。

3. **学术研究**: 为密码学哈希函数的电路优化研究提供了实践基础。

7.4 改进方向

1. **参数扩展**: 支持更多参数配置如 $(256, 2, 5)$，增强系统的灵活性。

2. **性能优化**: 进一步优化电路结构，减少约束数量和证明时间。

3. **安全增强**: 添加更多的输入验证和边界检查，提高系统的健壮性。

8 结论

本实验成功实现了基于 Circom 的 Poseidon2 哈希算法零知识证明电路，验证了理论算法在实际零知识证明系统中的可行性。电路具有 438 个约束，能够高效地证明哈希原象的知识而不泄露具体值。实验结果表明，该实现在保证安全性的同时实现了良好的性能表现，为后续的隐私计算应用奠定了坚实基础。

通过本实验，深入理解了现代零知识证明技术的核心原理和工程实践，掌握了从算法设计到系统实现的完整流程，为在隐私计算、区块链等前沿领域的进一步研究和应用积累了宝贵经验。