



创新创业实践

Project 3

实验报告

姓名 迟曼

学号 202200460070

学院 网络空间安全

专业 网络空间安全



目录

一、实验目的.....	3
二、主要文件说明.....	3
三、实验原理及过程.....	3
(一) poseidon2 哈希算法原理.....	3
(二) 代码解释	5
1.main.circom	5
2.poseidon2.circom.....	6
(三) 电路编译及 Groth16 证明生成验证	9
1.编译主电路	9
2.设置 Groth16 可信初始化.....	9
3.计算 Witness	11
4.生成证明	11
5.验证证明	11
四、总结与思考.....	11



一、实验目的

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>

二、主要文件说明

main.js (文件夹)	生成见证数据
poseidon2.circom	哈希算法函数代码
main.circom	主函数调用 poseidon2
input.json	输入文件
verification key.json	Groth16 zk-SNARK 验证密钥
proof.json	Groth16 证明数据

三、实验原理及过程

(一) poseidon2 哈希算法原理

Poseidon2 是第二代基于置换的密码哈希函数, 专门为零知识证明优化, 使用海绵结构。状态向量 $S \in F_p^t$ 在有限域 F_p 上 (p 为素数域), 其中:

- $t = r + c$ (r : 吸收率, c : 容量)
- 常用配置: $t=3$ (2 元素输入, 1 容量)
- BN254 标量域:

$$p = 21888242871839275222246405745257275088548364400416034343698204186575808495617$$



Poseidon2 轮函数由三种操作组成：

$$\text{Round} = \text{AddRoundConstants} \rightarrow \text{SubWords} \rightarrow \text{MixLayer}$$

a) 添加轮常数

$$S_i \leftarrow S_i + RC_k^{(i)} \forall i \in [0, t - 1]$$

其中 $RC_k^{(i)}$ 是预计算常数

b) S-Box (5 次幂非线性变换)

$$S_i \leftarrow S_i^\alpha \text{ 其中 } \alpha=5$$

c) MDS 矩阵混合

$$S \leftarrow M \times S$$

其中 M 是最大距离可分矩阵：

部分轮次操作（仅应用于首个元素）

$$\text{Partial Round} = \text{AddConstant} \rightarrow \text{SBox}(S_0) \rightarrow \text{MixLayer}$$

在状态向量 S 上：

$$S_0 \leftarrow (S_0 + RC_k)^5$$

$$S \leftarrow M \times S$$

Poseidon2 完整置换函数

设：

- S : 状态向量 ($\in \mathbb{F}_{pt}$)
- RF : 完整轮次数（首尾各半）
- RP : 部分轮次数（中间）

完整置换过程：

function f(S):

// 初始全轮次

for $r = 0$ to $R_F/2 - 1$:

$S = \text{AddConstants}(S, r)$

$S = \text{FullSBox}(S)$

$S = \text{MixLayer}(S)$

// 部分轮次

for $r = R_F/2$ to $R_F/2 + R_P - 1$:

$S = \text{AddConstant}(S_0, r)$

$S_0 = \text{SBox}(S_0)$ // 仅 S_0

$S = \text{MixLayer}(S)$



```
// 结束全轮次
for r = R_F/2 + R_P to R_F + R_P - 1:
    S = AddConstants(S, r)
    S = FullSBox(S)
    S = MixLayer(S)

return S
```

（二）代码解释

1.main.circom

```
pragma circom 2.1.4;
```

指定了代码使用的 Circom 编译器版本号为 2.1.4。

```
include "poseidon2.circom";
```

导入名为 "poseidon2.circom" 的外部文件，该文件应包含 Poseidon2 哈希函数的实现细节。这样允许代码复用已实现的密码学原语，而无需在主文件中重复定义。

```
template Main() {
```

声明了一个名为 "Main" 的可复用电路模板，是该电路的主要逻辑所在。

```
    signal input privateInputs[2];
```

```
    signal input publicHash;
```

```
    signal input privateInputs[2];
```

定义一个由两个元素组成的私有输入数组。这些输入仅对证明者可见，验证者无法直接查看，用于保护隐私数据

```
    signal input publicHash: 定义一个公开输入信号，用于提供预期的哈希值进行验证
```

```
    component hasher = Poseidon2_2_1();
```

创建了来自 poseidon2.circom 的 Poseidon2_2_1 组件实例。Poseidon2_2_1 表示专为两个输入设计的 Poseidon2 哈希函数（后缀 _2_1 表示处理 2 个输入元素）。

```
    hasher.hashInput[0] <== privateInputs[0];
```

```
    hasher.hashInput[1] <== privateInputs[1];
```

将电路的私有输入连接到哈希器组件：

第一个私有输入 (privateInputs[0]) 连接到哈希器的第一个输入口

第二个私有输入 (privateInputs[1]) 连接到哈希器的第二个输入口



```
hasher.hashOutput === publicHash;
```

规定：

哈希器生成的输出值，(hashOutput)必须严格等于公开提供的哈希值，(publicHash)这个等式约束在零知识证明中起着决定性作用，确保只有生成正确哈希值的输入才能通过验证。其中我们将在 input.js 中提供输入 1 和 0 以及其正确结果，以供验证。

该电路实现了："证明知道两个私有值 (privateInputs[0]和 privateInputs[1])，使得这些值经过 Poseidon2 哈希运算后得到的哈希值等于公开的 publicHash 值"。在整个过程中，验证者只能看到 publicHash 值，无法了解私有输入的具体内容，无法推断出任何关于私有输入的信息。

2.poseidon2.circom

```
template SBox() {  
    signal input  sbboxIn;  
    signal output sbboxOut;  
  
    signal squared <== sbboxIn * sbboxIn;  
    signal quartic <== squared * squared;  
    sbboxOut <== sbboxIn * quartic;  
}
```

定义 S-Box（置换盒）组件，计算输入信号的五次幂 (x^5)，实现方式： $x^2 \rightarrow x^4 \rightarrow x^5$ （三次乘法）

```
template InternalRound(roundIdx) {  
    signal input  roundIn[3];  
    signal output roundOut[3];  
  
    var constants[56] = [ ... ]; // 预定义的 56 个常数  
  
    component sbbox = SBox();  
    sbbox.sbboxIn <== roundIn[0] + constants[roundIdx];  
  
    roundOut[0] <== 2*sbbox.sbboxOut + roundIn[1] + roundIn[2];  
    roundOut[1] <== sbbox.sbboxOut + 2*roundIn[1] + roundIn[2];  
    roundOut[2] <== sbbox.sbboxOut + roundIn[1] + 3*roundIn[2];  
}
```

处理部分轮次计算，仅对第一个状态元素应用 S-Box 变换，使用轮索引选择对应的预定义常数，



第一个输入 + 常数后输入 S-Box，高效实现 Poseidon 的"部分轮次"。

```
template ExternalRound(roundIdx) {
    signal input  extIn[3];
    signal output extOut[3];

    var constants[8][3] = [ ... ]; // 预定义的 8 组常数

    component sboxes[3];
    for (var j = 0; j < 3; j++) {
        sboxes[j] = SBox();
        sboxes[j].sboxIn <== extIn[j] + constants[roundIdx][j];
    }

    extOut[0] <== 2*sboxes[0].sboxOut + sboxes[1].sboxOut + sboxes[2].sboxOut;
    extOut[1] <== sboxes[0].sboxOut + 2*sboxes[1].sboxOut + sboxes[2].sboxOut;
    extOut[2] <== sboxes[0].sboxOut + sboxes[1].sboxOut + 2*sboxes[2].sboxOut;
}
```

处理完整轮次计算，对所有三个状态元素应用 S-Box 变换，每组输入使用对应的常数，输出使用不同的线性组合矩阵，实现 Poseidon 的"完整轮次"。

```
template LinearLayer() {
    signal input  linIn[3];
    signal output linOut[3];

    linOut[0] <== 2*linIn[0] + linIn[1] + linIn[2];
    linOut[1] <== linIn[0] + 2*linIn[1] + linIn[2];
    linOut[2] <== linIn[0] + linIn[1] + 2*linIn[2];
}
```

简单的线性变换组件，在置换开始前初始化状态混合。

```
template Permutation() {
    signal input  permIn[3];
    signal output permOut[3];
    signal intermediate[65][3]; // 65 轮状态

    // 初始线性变换
    component lin = LinearLayer();
    // 连接输入到线性层
```



```
// 输出保存到第 0 轮状态

// 创建 8 个外部轮组件
component extRounds[8];
// 创建 56 个内部轮组件

// 前 4 轮外部轮次处理 (0-3)
for (var k = 0; k < 4; k++) {
    // 输入来自当前状态轮
    // 输出保存到下一轮状态
}

// 中间 56 轮内部轮次处理 (4-59)
for (var k = 0; k < 56; k++) {
    // 输入来自当前状态轮
    // 输出保存到下一轮状态
}

// 后 4 轮外部轮次处理 (60-63)
for (var k = 0; k < 4; k++) {
    // 输入来自当前状态轮
    // 输出保存到下一轮状态
}

// 最终状态作为输出 (第 64 轮)
}
```

完整的 Poseidon 置换函数实现，总轮数：8 完整轮 + 56 部分轮 = 64 轮 (中间状态 65 轮)，结构：4 完整轮 + 56 部分轮 + 4 完整轮，使用中间信号数组存储每轮状态，实现密码学安全的置换操作。

```
template Poseidon2_2_1() {
    signal input hashInput[2];
    signal output hashOutput;

    component perm = Permutation();
    perm.permIn[0] <== hashInput[0];
```




```
perm.permIn[1] <== hashInput[1];
perm.permIn[2] <== 0;
hashOutput <== perm.permOut[0];
}
```

顶层哈希接口，专为两个输入设计（2_1 表示），第三个输入固定为 0（标准填充），输出取置换结果的首个元素，提供简单的哈希函数调用接口，代码整体实现了 Poseidon2 哈希函数。

（三）电路编译及 Groth16 证明生成验证

1. 编译主电路

```
circom main.circom --r1cs --wasm --sym
```

```
template instances: 69
non-linear constraints: 240
linear constraints: 275
public inputs: 0
private inputs: 3
public outputs: 0
wires: 518
labels: 918
Written successfully: ./main.r1cs
Written successfully: ./main.sym
Written successfully: ./main_js/main.wasm
Everything went okay
```

2. 设置 Groth16 可信初始化

启动新的 Powers of Tau 仪式

```
snarkjs powersoftau new bn128 14 pot14_0000.ptau -v
```

```
4_0000.ptau -v
[DEBUG] snarkJS: Calculating First Challenge Hash
[DEBUG] snarkJS: Calculate Initial Hash: tauG1
[DEBUG] snarkJS: Calculate Initial Hash: tauG2
[DEBUG] snarkJS: Calculate Initial Hash: alphaTauG1
[DEBUG] snarkJS: Calculate Initial Hash: betaTauG1
[DEBUG] snarkJS: Blank Contribution Hash:
786a02f7 42015903 c6c6fd85 2552d272
912f4740 e1584761 8a86e217 f71f5419
d25e1031 afee5853 13896444 934eb04b
903a685b 1448b755 d56f701a fe9be2ce
[INFO] snarkJS: First Contribution Hash:
bc0bde79 80381fa6 42b20975 91dd83f1
ed15b003 e15c3552 0af32c95 eb519149
2a6f3175 215635cf c10e6098 e2c612d0
ca84f1a9 f90b5333 560c8af5 9b9209f4
```

为仪式贡献

```
snarkjs powersoftau contribute pot14_0000.ptau pot14_0001.ptau --name="First contribution" -v
```



```
0000.ptau pot14_0001.ptau --name="First contribution" -v
Enter a random text. (Entropy): cmcmm
[DEBUG] snarkJS: Calculating First Challenge Hash
[DEBUG] snarkJS: Calculate Initial Hash: tauG1
[DEBUG] snarkJS: Calculate Initial Hash: tauG2
[DEBUG] snarkJS: Calculate Initial Hash: alphaTauG1
[DEBUG] snarkJS: Calculate Initial Hash: betaTauG1
[DEBUG] snarkJS: processing: tauG1: 0/32767
[DEBUG] snarkJS: processing: tauG1: 16384/32767
[DEBUG] snarkJS: processing: tauG2: 0/16384
[DEBUG] snarkJS: processing: tauG2: 8192/16384
[DEBUG] snarkJS: processing: alphaTauG1: 0/16384
[DEBUG] snarkJS: processing: betaTauG1: 0/16384
[DEBUG] snarkJS: processing: betaTauG2: 0/1
[INFO] snarkJS: Contribution Response Hash imported:
5586d600 12d2a3d4 53f27f6f 2eb9c7a2
06462670 b3e5e73c c264a9d8 71ac2ff4
40a8b888 e9b9740e 0b7957d3 bf279864
81630b43 13e62f7e 18209a8c f44f152c
[INFO] snarkJS: Next Challenge Hash:
f95504de a2f14f51 68e8ee65 27305f2a
3852ce99 650abdf6 cf664b9c f67d7b91
25180890 50698846 1cd0a4a5 7ea77d4b
5f8704b1 2d04f1c7 fe678456 d06b54da
```

准备第 2 阶段

```
snarkjs powersoftau prepare phase2 pot14_0001.ptau pot14_final.ptau -v
```

```
[DEBUG] snarkJS: betaTauG1: fft 14 join 13/14 2/2 1/8
[DEBUG] snarkJS: betaTauG1: fft 14 join 13/14 2/2 7/8
[DEBUG] snarkJS: betaTauG1: fft 14 join 13/14 1/2 5/8
[DEBUG] snarkJS: betaTauG1: fft 14 join 13/14 1/2 0/8
[DEBUG] snarkJS: betaTauG1: fft 14 join 13/14 2/2 3/8
[DEBUG] snarkJS: betaTauG1: fft 14 join 13/14 2/2 6/8
[DEBUG] snarkJS: betaTauG1: fft 14 join 13/14 1/2 3/8
[DEBUG] snarkJS: betaTauG1: fft 14 join 13/14 2/2 2/8
[DEBUG] snarkJS: betaTauG1: fft 14 join 13/14 2/2 4/8
[DEBUG] snarkJS: betaTauG1: fft 14 join 13/14 1/2 6/8
[DEBUG] snarkJS: betaTauG1: fft 14 join 13/14 1/2 2/8
[DEBUG] snarkJS: betaTauG1: fft 14 join 13/14 2/2 5/8
[DEBUG] snarkJS: betaTauG1: fft 14 join: 14/14
[DEBUG] snarkJS: betaTauG1: fft 14 join 14/14 1/1 13/16
[DEBUG] snarkJS: betaTauG1: fft 14 join 14/14 1/1 15/16
[DEBUG] snarkJS: betaTauG1: fft 14 join 14/14 1/1 7/16
[DEBUG] snarkJS: betaTauG1: fft 14 join 14/14 1/1 9/16
[DEBUG] snarkJS: betaTauG1: fft 14 join 14/14 1/1 5/16
[DEBUG] snarkJS: betaTauG1: fft 14 join 14/14 1/1 10/16
[DEBUG] snarkJS: betaTauG1: fft 14 join 14/14 1/1 0/16
[DEBUG] snarkJS: betaTauG1: fft 14 join 14/14 1/1 3/16
[DEBUG] snarkJS: betaTauG1: fft 14 join 14/14 1/1 14/16
[DEBUG] snarkJS: betaTauG1: fft 14 join 14/14 1/1 12/16
[DEBUG] snarkJS: betaTauG1: fft 14 join 14/14 1/1 11/16
[DEBUG] snarkJS: betaTauG1: fft 14 join 14/14 1/1 4/16
[DEBUG] snarkJS: betaTauG1: fft 14 join 14/14 1/1 1/16
[DEBUG] snarkJS: betaTauG1: fft 14 join 14/14 1/1 8/16
[DEBUG] snarkJS: betaTauG1: fft 14 join 14/14 1/1 2/16
[DEBUG] snarkJS: betaTauG1: fft 14 join 14/14 1/1 6/16
```

生成.zkey 文件(Groth16 设置)

```
snarkjs groth16 setup main.r1cs pot14_final.ptau main_0000.zkey
```

```
_final.ptau main_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:
eb5585bb 3721ed5e 4e4b0321 5e742707
dcf27d61 f5c9c6aa c897b7bd d41cea14
ae3fdbeb 54f572ac 3b8ea798 339e04f0
00f98430 10707174 25cd3b0d 0035bbef
```

贡献到第 2 阶段仪式

```
snarkjs zkey contribute main_0000.zkey main_0001.zkey --name="Second contribution" -v
```



```
y main_0001.zkey --name="Second contribution" -v
Enter a random text. (Entropy): cmcmcmcm
[DEBUG] snarkJS: Applying key: L Section: 0/517
[DEBUG] snarkJS: Applying key: H Section: 0/1024
[INFO] snarkJS: Circuit Hash:
          eb5585bb 3721ed5e 4e4b0321 5e742707
          dcf27d61 f5c9c6aa c897b7bd d41cea14
          ae3fdbeb 54f572ac 3b8ea798 339e04f0
          00f98430 10707174 25cd3b0d 0035bbef
[INFO] snarkJS: Contribution Hash:
          d3dc38c9 cda6be29 3050ba67 5b5c75ec
          1b38479d 70d48c4a 868e7e86 5c7ba307
          b1102bd0 9609e715 f439dc67 a17d4239
          c6aalc22 7c9465ef b5b45aab 013e22e4
```

导出验证密钥

```
snarkjs zkey export verificationkey main_0001.zkey verification_key.json
```

```
ain_0001.zkey verification_key.json
[INFO] snarkJS: EXPORT VERIFICATION KEY STARTED
[INFO] snarkJS: > Detected protocol: groth16
[INFO] snarkJS: EXPORT VERIFICATION KEY FINISHED
```

3.计算 Witness

```
cd main_js
```

```
node generate_witness.js main.wasm ../input.json ../witness.wtns
```

```
cd ..
```

4.生成证明

```
snarkjs groth16 prove main_0001.zkey witness.wtns proof.json public.json
```

5.验证证明

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

```
ey.json public.json proof.json
[INFO] snarkJS: OK!
```

四、总结与思考

通过本次基于 Circom 实现 Poseidon2 哈希电路的实践，我对零知识证明的技术栈有了更深刻的理解。在实现过程中，算法参数的配置尤为关键——最终选用 $(n,t,d)=(256,3,5)$ 的组合，虽比 $t=2$ 的方案多出约 30% 的约束数，但其三轮 FullRound 与 PartialRound 交错的结构显著提升了抗差分攻击能力。这让我意识到，密码电路的设计本质是安全性与效率的权衡。

在电路开发阶段，轮函数的模块化设计成为核心挑战。通过解构 Poseidon2 的置换过程，我将 SBox 的非线性变换拆解为三次乘法操作（ $x^2 \rightarrow x^4 \rightarrow x^5$ ），并在 PartialRound 中仅对首元素施加 SBox，使约束数从理论值 358 优化至实测 275 个。



Groth16 证明的生成过程中，在可信初始化阶段，powersoftau 仪式生成的 ptau 文件其安全性依赖于多方贡献的随机性；而 Witness 计算环节中，WASM 模块将输入数据转化为多项式承诺将输入输出关系编码为 R1CS 约束方程。最终验证通过的 OK!，本实验让我体会到 zk-SNARK 如何通过三重数学机制（多项式承诺、盲求值、简洁验证）实现可验证的隐私计算。