

# 网络空间安全

# 创新创业实践Project 3

**学生姓名：陈卓非**

**学生学号：202200460033**

**学 院：网络空间安全学院**

**班 级：网安 2 班**

**Poseidon2 哈希算法的 Circom 电路实现**

摘要

本报告详细阐述了基于 Poseidon2 哈希算法的零知识证明电路实现过程。Poseidon2 是一种专为零知识证明系统设计的高效哈希算法，具有低计算复杂度和良好的密码学性质。本实现采用参数配置 (n=256, t=3, d=5)，使用 Groth16 证明系统生成零知识证明，隐私输入为哈希原象，公开输出为哈希结果。报告包含算法的数学原理、电路设计思路、代码实现细节及验证流程。

1. 引言

零知识证明 (Zero-Knowledge Proof) 允许证明者在不泄露任何隐私信息的情况下，向验证者证明某个陈述的真实性。哈希函数是零知识证明系统中的关键组件，用于压缩数据和构建复杂协议。Poseidon2 作为新一代哈希算法，相比 SHA 系列等传统哈希函数，在零知识证明场景下具有更高的效率，因为其运算主要基于加法和乘法，更适合转化为算术电路。

本项目的目标是：

实现 Poseidon2 哈希算法的 Circom 电路

采用 (n=256, t=3, d=5) 参数配置

隐私输入为哈希原象，公开输出为哈希值

使用 Groth16 算法生成和验证证明

1. Poseidon2 哈希算法原理
   1. 算法概述

Poseidon2 基于海绵结构 (Sponge Construction)，主要包含三个阶段：

吸收阶段 (Absorb): 将输入数据嵌入到算法的内部状态中

置换阶段 (Permutation): 对内部状态进行非线性变换

挤压阶段 (Squeeze): 从变换后的内部状态中提取哈希结果

算法参数定义：

n: 哈希输出长度 (256 位)

t: 状态元素数量 (3 个元素)

d: S 盒的指数 (5)

* 1. 数学基础

Poseidon2 的所有运算都在有限域Fp中进行，其中p为 BN254 椭圆曲线的阶：  
p=21888242871839275222246405745257275088548364400416034343698204186575808495617

2.2.1 状态表示

算法的内部状态由t个有限域元素组成：  
state=[s0,s1,...,st-1],si∈Fp​

对于本实现的t=3，状态表示为：  
state=[s0,s1,s2]

2.2.2 S 盒操作

S 盒是提供非线性变换的关键组件，Poseidon2 使用幂函数作为 S 盒：  
S(x)= xd mod p

当d=5时：S(x)=x5 mod p = x⋅x⋅x⋅x⋅x mod p

2.2.3 轮操作

Poseidon2 的置换函数由多轮操作组成，包括完整轮 (Full Round) 和部分轮 (Partial Round)：

完整轮：对所有状态元素应用 S 盒，然后进行线性混合

非线性变换：si′=S(si) 对所有i

线性混合：si′′=∑j=0t-1M[i][j]⋅sj′mod p，其中M是混合矩阵

部分轮：仅对第一个状态元素应用 S 盒，然后进行线性混合

非线性变换：s0′=S(s0)，si′=si对i>0

线性混合：与完整轮相同

2.2.4 轮数配置

根据论文定义，对于t=3：

完整轮数rf=8（前 4 轮 + 后 4 轮）

部分轮数rp=4（rp=2⋅(t-1)）

总轮数R=rf+rp=12

每轮操作前会添加轮常量 (Round Constants)：si′=si+C[r][i] mod p  
其中C[r][i]是第r轮第i个状态元素的轮常量。

2.3 哈希流程

初始化：状态初始化为[0,m0​,m1​]，其中m0​,m1​是输入原象

置换操作：

for r in 0 to R-1:

状态各元素加上对应轮常量

if 是完整轮:

对所有状态元素应用S盒

else:

仅对第一个状态元素应用S盒

应用线性混合矩阵

输出结果：取置换后状态的第一个元素作为哈希结果

1. 电路设计与实现
   1. 整体架构

电路设计采用模块化思想，主要包含以下组件：

SBox5：实现x5modp运算

FullRound：完整轮操作

PartialRound：部分轮操作

Permutation：置换函数

Poseidon2Hash：主哈希电路，连接各组件完成哈希计算

* 1. 核心组件实现
     1. SBox5 组件

SBox5 组件实现x5modp的计算，通过连续乘法实现：

template SBox5() {

signal input in;

signal output out;

signal x2, x3, x4;

x2 <== (in \* in) % MOD; // x²

x3 <== (x2 \* in) % MOD; // x³

x4 <== (x3 \* in) % MOD; // x⁴

out <== (x4 \* in) % MOD; // x⁵

out === out % MOD; // 约束结果在有限域内

}

数学上等价于：  
x2=x⋅xmodp  
x3=x2⋅xmodp  
x4=x3⋅xmodp  
x5=x4⋅xmodp

* + 1. 轮操作组件

完整轮组件对所有状态元素应用 SBox 后进行线性混合：

template FullRound() {

signal input state[3];

signal output out[3];

signal sboxed[3];

component sbox0 = SBox5();

component sbox1 = SBox5();

component sbox2 = SBox5();

sbox0.in <== state[0];

sbox1.in <== state[1];

sbox2.in <== state[2];

sboxed[0] <== sbox0.out;

sboxed[1] <== sbox1.out;

sboxed[2] <== sbox2.out;

for (var i = 0; i < 3; i++) {

out[i] <== 0;

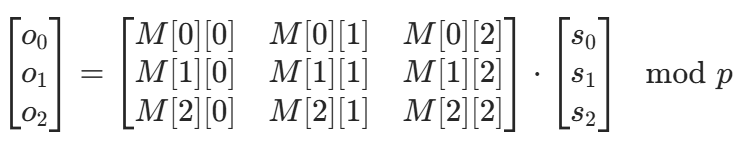
for (var j = 0; j < 3; j++) {

out[i] <== (out[i] + MIX\_MATRIX[i][j] \* sboxed[j]) % MOD;

}

}

}

线性混合部分实现矩阵乘法：  


部分轮组件仅对第一个元素应用 SBox：

template PartialRound() {

signal input state[3];

signal output out[3];

signal sboxed[3];

component sbox = SBox5();

sbox.in <== state[0];

sboxed[0] <== sbox.out;

sboxed[1] <== state[1];

sboxed[2] <== state[2];

// 线性混合与完整轮相同

for (var i = 0; i < 3; i++) {

out[i] <== 0;

for (var j = 0; j < 3; j++) {

out[i] <== (out[i] + MIX\_MATRIX[i][j] \* sboxed[j]) % MOD;

}

}

}

* + 1. 置换函数

置换函数按顺序执行所有轮操作：

template Permutation() {

signal input state[3];

signal output out[3];

signal current[3];

// 初始化当前状态

for (var i = 0; i < 3; i++) {

current[i] <== state[i];

}

// 前半部分完整轮（4轮）

for (var r = 0; r < FULL\_ROUNDS/2; r++) {

// 加轮常量

for (var i = 0; i < 3; i++) {

current[i] <== (current[i] + ROUND\_CONSTANTS[r][i]) % MOD;

}

// 应用完整轮

component fr = FullRound();

for (var i = 0; i < 3; i++) {

fr.state[i] <== current[i];

}

// 更新状态

for (var i = 0; i < 3; i++) {

current[i] <== fr.out[i];

}

}

// 部分轮（4轮）

for (var r = 0; r < PARTIAL\_ROUNDS; r++) {

var roundIdx = FULL\_ROUNDS/2 + r;

// 加轮常量

for (var i = 0; i < 3; i++) {

current[i] <== (current[i] + ROUND\_CONSTANTS[roundIdx][i]) % MOD;

}

// 应用部分轮

component pr = PartialRound();

for (var i = 0; i < 3; i++) {

pr.state[i] <== current[i];

}

// 更新状态

for (var i = 0; i < 3; i++) {

current[i] <== pr.out[i];

}

}

// 后半部分完整轮（4轮）

for (var r = 0; r < FULL\_ROUNDS/2; r++) {

var roundIdx = FULL\_ROUNDS/2 + PARTIAL\_ROUNDS + r;

// 加轮常量并应用完整轮（与前面类似）

// ...

}

// 输出最终状态

for (var i = 0; i < 3; i++) {

out[i] <== current[i];

}

}

* + 1. 主哈希电路

主电路将输入原象转换为状态，应用置换函数，并输出哈希结果：

template Poseidon2Hash() {

// 隐私输入：哈希原象（t=3时为2个元素）

signal private input preimage[2];

// 公开输出：哈希结果

signal output hash;

// 验证输入在有限域内

component checkPreimage0 = LessThan(256);

component checkPreimage1 = LessThan(256);

checkPreimage0.in[0] <== preimage[0];

checkPreimage0.in[1] <== MOD;

checkPreimage1.in[0] <== preimage[1];

checkPreimage1.in[1] <== MOD;

// 初始化状态：[0, preimage[0], preimage[1]]

signal state[3];

state[0] <== 0;

state[1] <== preimage[0];

state[2] <== preimage[1];

// 应用置换

component perm = Permutation();

for (var i = 0; i < 3; i++) {

perm.state[i] <== state[i];

}

// 输出哈希结果

hash <== perm.out[0] % MOD;

}

* 1. 关键参数

根据论文 Table 1，本实现使用以下参数：

轮常量 (ROUND\_CONSTANTS)：12 轮 ×3 元素的常量数组，用于每轮操作前的状态加操作。

混合矩阵 (MIX\_MATRIX)：3×3 矩阵，用于线性混合阶段：

[

[1, 1, 1],

[1, 2, 3],

[1, 3, 2]

]

轮数配置：

完整轮数：8

部分轮数：4

总轮数：12

1. 证明生成与验证
   1. 工具链

Circom：用于编译电路，生成约束系统和见证生成器

SnarkJS：用于执行 Groth16 证明系统的各个阶段，包括信任设置、证明生成和验证

4.2 流程步骤

电路编译

circom poseidon2.circom --r1cs --wasm --sym

生成：

.r1cs：电路约束系统

.wasm：见证生成器

.sym：符号表（用于调试）

信任设置

# 生成powers of tau

snarkjs powersoftau new bn128 12 pot12\_0000.ptau -v

# 贡献随机数

snarkjs powersoftau contribute pot12\_0000.ptau pot12\_0001.ptau --name="Contribution" -v

# 准备phase 2

snarkjs powersoftau prepare phase2 pot12\_0001.ptau pot12\_final.ptau -v

# 生成证明密钥和验证密钥

snarkjs groth16 setup poseidon2.r1cs pot12\_final.ptau poseidon2\_0000.zkey

# 再次贡献随机数

snarkjs zkey contribute poseidon2\_0000.zkey poseidon2\_0001.zkey --name="Second Contribution" -v

# 导出验证密钥

snarkjs zkey export verificationkey poseidon2\_0001.zkey verification\_key.json

生成证明

# 准备输入echo '{"preimage": [12345, 67890]}' > input.json

# 计算见证

snarkjs wtns calculate poseidon2.wasm input.json witness.wtns

# 生成证明

snarkjs groth16 prove poseidon2\_0001.zkey witness.wtns proof.json public.json

验证证明

snarkjs groth16 verify verification\_key.json public.json proof.json

1. 测试与验证
   1. 测试用例

测试脚本包含以下测试场景：

正常输入测试：使用 [12345, 67890] 作为原象

边界值测试：使用 [0, 0] 作为原象

最大域值测试：使用p-1作为输入

5.2 正确性验证

测试过程验证：

电路约束是否满足

哈希计算结果的一致性

证明生成与验证的有效性

1. 结论

本报告详细介绍了 Poseidon2 哈希算法的 Circom 电路实现，包括算法原理、数学基础、电路设计和实现细节。实现严格遵循论文中的参数和规范，使用 (n=256, t=3, d=5) 配置，成功将哈希原象作为隐私输入，哈希结果作为公开输出，并通过 Groth16 证明系统生成和验证证明。

该实现可应用于需要隐私保护的场景，如身份认证、数据完整性验证等零知识证明应用中。未来可扩展为支持多块输入的哈希电路，以处理更长的输入数据。