



网络安全创新创业实践第三次实验

用 circom 实现 poseidon2 哈希算法的电 路

学院: 网络空间安全学院

专业: 密码科学与技术

姓名: 金鑫悦

学号: 202200460042

目录

1 实验任务	2
2 实验环境	2
3 实验原理	2
3.1 Poseidon2 哈希算法原理	2
3.1.1 海绵函数框架	2
3.1.2 轮变换设计	2
3.1.3 关键组件	3
3.2 Circom 电路构造原理	3
3.2.1 电路组件分解	3
3.2.2 电路整体结构	4
3.3 Groth16 零知识证明原理	5
3.3.1 核心流程	5
3.3.2 可信设置	5
3.3.3 证明生成与验证	5
4 实验过程	5
5 结果分析	13
6 总结与感悟	14

1 实验任务

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>

2 实验环境

硬件环境:

12th Intel (R) Core (TM) i7-12700H

软件环境:

ubuntu-20.04

3 实验原理

3.1 Poseidon2 哈希算法原理

Poseidon2 是基于海绵函数 (Sponge Function) 的高效哈希算法, 核心设计目标是在零知识证明 (ZKP) 中实现低电路复杂度。其参数化结构为

$\text{POSEIDON2}^t(n, t, d)$, 其中:

n : 哈希输出位长 (与椭圆曲线标量域大小匹配, 如 256 位);

t : 状态向量长度 (本文取 $t = 3$, 即 3 个 256 位元素);

d : S 盒指数 (本文取 $d = 5$, 即 x^5 幂运算)。

3.1.1 海绵函数框架

Poseidon2 遵循海绵函数的“吸收-置换-挤压”流程:

$$\text{Hash}(M) = \text{Squeeze}(\text{Permute}(\text{Absorb}(M)))$$

吸收 (Absorb): 将输入消息 M 分块填入状态向量;

置换 (Permute): 通过多轮非线性变换 (S 盒) 和线性变换 (MDS 矩阵) 混淆状态;

挤压 (Squeeze): 从最终状态中提取哈希值。

3.1.2 轮变换设计

Poseidon2 的置换由全轮 (Full Round) 和半轮 (Partial Round) 交替组成:

$$\text{Permute}(S) = \text{Round}_{\text{full}}^4 \circ \text{Round}_{\text{partial}}^{56} \circ \text{Round}_{\text{full}}^4(S)$$

全轮 (Full Round): 所有 t 个状态元素应用 S 盒 (x^5)，后接 MDS 矩阵乘法和轮常量加法；

半轮 (Partial Round): 仅第一个状态元素应用 S 盒，其余元素直接参与 MDS 变换。

3.1.3 关键组件

- S 盒

采用幂函数 $f(x) = x^5 \bmod p$ (p 为 BN254 标量域)，实现非线性混淆：

$$S(x) = x^5 = x \times x \times x \times x \times x$$

在电路中通过多级乘法约束实现（见后续 Circom 实现）。

- MDS 矩阵

3×3 线性变换矩阵，确保状态扩散性。满足：

$$\text{MDS}(S_{\text{after S-box}} + C) \in \text{GF}(p)^{t \times t}$$

其中 C 为轮常量，确保每轮变换的唯一性。

3.2 Circom 电路构造原理

Circom 是专为零知识证明设计的电路描述语言，核心思想是通过约束系统 (R1CS) 描述计算逻辑。本文构造的电路需证明：

$$\text{Hash}(\text{preimage}) = \text{public_hash}$$

其中 preimage 为隐私输入，public_hash 为公开输入。

3.2.1 电路组件分解

- S 盒电路

通过多级乘法约束实现 x^5 运算：

$$\begin{aligned} t_1 &= x \times x, \\ t_2 &= t_1 \times x, \\ t_3 &= t_2 \times x, \\ t_4 &= t_3 \times x, \\ \text{out} &= t_4. \end{aligned}$$

对应 Circom 代码：

```
template SBox5() {
    signal input in;
    signal output out;
    signal t1, t2, t3, t4;
    t1 <== in * in;
    t2 <== t1 * in;
    t3 <== t2 * in;
```

```

t4 <= t3 * in;
out <= t4;
out === t4;
}

```

- 轮变换电路

全轮 (Full Round) :

所有状态元素过 S 盒，后接 MDS 乘法和轮常量加法：

$$\text{out}[i] = \text{MDS}[i][j] \times (\text{S-box}(\text{state}[j]) + C[j])$$

对应 Circos 代码核心逻辑：

```

template FullRound() {
    signal input state[3];
    signal input constants[3];
    signal input mds[3][3];
    signal output out[3];
    // S-box 应用
    component sboxes[3];
    for (var i = 0; i < 3; i++) {
        sboxes[i] = SBox5();
        sboxes[i].in <= state[i];
    }
    // 轮常量加法与 MDS 乘法
    for (var i = 0; i < 3; i++) {
        out[i] <= 0;
        for (var j = 0; j < 3; j++) {
            out[i] <= out[i] + mds[i][j] * (sboxes[j].out + constants[j]);
        }
        out[i] === out[i];
    }
}

```

半轮：

仅第一个元素过 S 盒，其余元素直接参与变换：

$$\text{out}[0] = \text{MDS}[0][j] \times (\text{S-box}(\text{state}[0]) + C[j]), \quad i > 0 \text{ 时} \quad \text{out}[i] = \text{MDS}[i][j] \times (\text{state}[i] + C[j])$$

3.2.2 电路整体结构

电路遵循海绵函数流程：

1. 吸收阶段：将隐私输入 preimage 填入状态向量；
2. 置换阶段：通过 8 全轮 + 56 半轮（参考 Table 1 参数）变换状态；
3. 挤压阶段：提取状态第一个元素作为哈希值，与公开输入 public_hash 绑定。

3.3 Groth16 零知识证明原理

Groth16 是高效的非交互式零知识证明 (NIZKP) 算法，核心是通过可信设置 (Trusted Setup) 生成证明密钥，实现“知识证明”而不泄露隐私输入。

3.3.1 核心流程

$$\text{Prove}(x, w) \rightarrow \pi, \quad \text{Verify}(\pi, x) \rightarrow \{0, 1\}$$

- x : 公开输入 (哈希值 public_hash);
 w : 隐私输入 (原象 preimage);
 π : 证明，验证者通过 π 确认 w 满足电路约束。

3.3.2 可信设置

生成公共参考字符串 (CRS)，需注意：

$$\text{CRS} \leftarrow \text{Setup}(\lambda, \text{Circuit})$$

其中 λ 为安全参数，本文使用 12 次幂的 Ptau 文件 ('pot12_0000.ptau')。

3.3.3 证明生成与验证

- 证明生成： $\pi \leftarrow \text{Prove}(\text{CRS}, x, w)$ ，通过电路约束生成短证明；
验证： $\text{Verify}(\text{VerificationKey}, x, \pi) \rightarrow \text{true/false}$ ，确保证明有效且未篡改。

4 实验过程

1. 准备系统环境

首先更新系统并安装基础工具：

```
1 # 更新系统包
2 sudo apt update && sudo apt upgrade -y
3
4 # 安装必要依赖
5 sudo apt install -y git wget build-essential libgmp3-dev libssl-dev
```

2. 安装 Node.js

```
1 # 安装 NodeSource 源 (用于获取 Node.js v16)
2 curl -fsSL https://deb.nodesource.com/setup_16.x | sudo -E bash -
3
4 # 安装 Node.js 和 npm
5 sudo apt install -y nodejs
6
7 # 验证版本 (确保 Node.js 16, npm 8)
```

```
8 node -v # 应输出 v16.x.x
9 npm -v # 应输出 8.x.x
```

3. 创建项目目录结构

```
1 mkdir -p poseidon2-project/{circuits,scripts,test,powersOfTau}
2 cd poseidon2-project
```

4. 安装密码学工具链

```
1 npm install -g circom@2.1.8 snarkjs@0.7.0
2 circom --version
3 snarkjs --version
```

5. 下载信任设置文件 (Ptau)

```
1 # 进入 powersOfTau 目录
2 cd powersOfTau
3
4 # 下载 12 次方 Ptau 文件 (约 1.7GB)
5 wget
       https://hermez.s3-eu-west-1.amazonaws.com/powersOfTau28_hez_final_12.ptau
      -O pot12_0000.ptau
6
7 # 验证文件完整性
8 sha256sum pot12_0000.ptau
9 #
       正确输出: a8b371131013513b90d3e93e54415d8be14cb0059057205de92033cc807961a
      pot12_0000.ptau
10 cd ..
```

6. 编写核心电路代码

circuits/constants.circom: 参数定义

```
1 cat > circuits/constants.circom << EOF
2 // Poseidon2 参数: (256,3,5), 参考文档1 Table1
3 template Poseidon2Constants() {
4     // MDS矩阵
5     signal input mds[3][3] = [
6         [0x01, 0x02, 0x03],
7         [0x04, 0x05, 0x06],
8         [0x07, 0x08, 0x09]
9     ];
10 }
```

```
11 // 轮常量 (共64轮: 4全+56半+4全)
12 signal input constants[64][3] = [
13     // 前4全轮
14     [0x10, 0x11, 0x12], [0x13, 0x14, 0x15], [0x16, 0x17, 0x18],
15     [0x19, 0x1a, 0x1b],
16     // 56半轮 (示例前4个)
17     [0x20, 0x21, 0x22], [0x23, 0x24, 0x25], [0x26, 0x27, 0x28],
18     [0x29, 0x2a, 0x2b],
19     // 后4全轮
20     [0x30, 0x31, 0x32], [0x33, 0x34, 0x35], [0x36, 0x37, 0x38],
21     [0x39, 0x3a, 0x3b]
22 ];
23 }
24 EOF
```

7. circuits/poseidon2.circom: 电路实现

```
1 cat > circuits/poseidon2.circom << EOF
2 include "constants.circom";
3 include "node_modules/circomlib/circuits/bitify.circom";
4
5 // S盒: x^5 运算
6 template SBox5() {
7     signal input in;
8     signal output out;
9     signal t1, t2, t3, t4;
10    t1 <== in * in; // in^2
11    t2 <== t1 * in; // in^3
12    t3 <== t2 * in; // in
13    t4 <== t3 * in; // in
14    out <== t4;
15    out === t4;
16 }
17
18 // 全轮变换
19 template FullRound() {
20     signal input state[3];
21     signal input constants[3];
22     signal input mds[3][3];
23     signal output out[3];
24
25     component sboxes[3];
26     for (var i = 0; i < 3; i++) {
```

```
27         sboxes[i] = SBox5();
28         sboxes[i].in <== state[i];
29     }
30
31     signal after_const[3];
32     for (var i = 0; i < 3; i++) {
33         after_const[i] <== sboxes[i].out + constants[i];
34     }
35
36     for (var i = 0; i < 3; i++) {
37         out[i] <== 0;
38         for (var j = 0; j < 3; j++) {
39             out[i] <== out[i] + mds[i][j] * after_const[j];
40         }
41         out[i] === out[i];
42     }
43 }
44
45 // 半轮变换
46 template PartialRound() {
47     signal input state[3];
48     signal input constants[3];
49     signal input mds[3][3];
50     signal output out[3];
51
52     component sbox;
53     sbox = SBox5();
54     sbox.in <== state[0];
55     signal sboxed[3];
56     sboxed[0] <== sbox.out;
57     sboxed[1] <== state[1];
58     sboxed[2] <== state[2];
59
60     signal after_const[3];
61     for (var i = 0; i < 3; i++) {
62         after_const[i] <== sboxed[i] + constants[i];
63     }
64
65     for (var i = 0; i < 3; i++) {
66         out[i] <== 0;
67         for (var j = 0; j < 3; j++) {
68             out[i] <== out[i] + mds[i][j] * after_const[j];
```

```
69         }
70         out[i] === out[i];
71     }
72 }
73
74 // 置换函数
75 template Poseidon2Permutation() {
76     signal input state[3];
77     signal input mds[3][3];
78     signal input constants[64][3];
79     signal output out[3];
80
81     signal current[3];
82     for (var i = 0; i < 3; i++) current[i] <== state[i];
83
84     // 前4全轮
85     for (var r = 0; r < 4; r++) {
86         component fr = FullRound();
87         for (var i = 0; i < 3; i++) {
88             fr.state[i] <== current[i];
89             fr.constants[i] <== constants[r][i];
90             for (var j = 0; j < 3; j++) fr.mds[i][j] <== mds[i][j];
91         }
92         for (var i = 0; i < 3; i++) current[i] <== fr.out[i];
93     }
94
95     // 56半轮
96     for (var r = 0; r < 56; r++) {
97         component pr = PartialRound();
98         for (var i = 0; i < 3; i++) {
99             pr.state[i] <== current[i];
100            pr.constants[i] <== constants[4 + r][i];
101            for (var j = 0; j < 3; j++) pr.mds[i][j] <== mds[i][j];
102        }
103        for (var i = 0; i < 3; i++) current[i] <== pr.out[i];
104    }
105
106     // 后4全轮
107     for (var r = 0; r < 4; r++) {
108         component fr = FullRound();
109         for (var i = 0; i < 3; i++) {
110             fr.state[i] <== current[i];
```

```
111         fr.constants[i] <== constants[60 + r][i];
112         for (var j = 0; j < 3; j++) fr.mds[i][j] <== mds[i][j];
113     }
114     for (var i = 0; i < 3; i++) current[i] <== fr.out[i];
115 }
116
117 for (var i = 0; i < 3; i++) out[i] <== current[i];
118 }
119
120 // 主电路
121 template Poseidon2Hash() {
122     signal public output hash;
123     signal private input preimage[2];
124
125     component constants = Poseidon2Constants();
126     signal initial_state[3];
127     initial_state[0] <== 0;
128     initial_state[1] <== preimage[0];
129     initial_state[2] <== preimage[1];
130
131     component perm = Poseidon2Permutation();
132     for (var i = 0; i < 3; i++) {
133         perm.state[i] <== initial_state[i];
134         for (var j = 0; j < 3; j++) perm.mds[i][j] <==
135             constants.mds[i][j];
136     }
137     for (var r = 0; r < 64; r++) {
138         for (var i = 0; i < 3; i++) perm.constants[r][i] <==
139             constants.constants[r][i];
140     }
141     hash <== perm.out[0];
142 }
143 component main = Poseidon2Hash();
144 EOF
```

8. scripts/compile.sh: 编译脚本

```
1 cat > scripts/compile.sh << EOF
2 #!/bin/bash
3
4 # 安装 circomlib 依赖
```

```
5  npm install circomlib
6
7  # 编译电路
8  cd circuits
9  circom poseidon2.circom --r1cs --wasm --sym
10 cd ..
11
12 # 生成 Groth16 证明密钥
13 snarkjs groth16 setup circuits/poseidon2.r1cs
    powersOfTau/pot12_0000.ptau poseidon2_0000.zkey
14
15 # 贡献到 Zkey (模拟多方参与)
16 snarkjs zkey contribute poseidon2_0000.zkey poseidon2_final.zkey
    --name="First contribution" -v -e="random text"
17
18 # 导出验证密钥
19 snarkjs zkey export verificationkey poseidon2_final.zkey
    verification_key.json
20
21 # 生成测试输入
22 node test/test_poseidon2.js
23
24 # 生成证明
25 snarkjs groth16 prove poseidon2_final.zkey input.json proof.json
    public.json
26
27 # 验证证明
28 snarkjs groth16 verify verification_key.json public.json proof.json
29 EOF
30
31 # 赋予执行权限
32 chmod +x scripts/compile.sh
```

9. test/test_poseidon2.js: 测试输入脚本

```
1 cat > test/test_poseidon2.js << EOF
2 const fs = require('fs');
3
4 // 隐私输入
5 const preimage = [1n, 2n];
6
7 // 公开输入
8 const hash = 0x1234567890abcdefn;
```

```
9
10 // 生成输入文件
11 const input = {
12     preimage: preimage.map(x => x.toString()),
13     hash: hash.toString()
14 };
15
16 fs.writeFileSync('input.json', JSON.stringify(input, null, 2));
17 console.log("生成输入文件: input.json");
18 EOF
```

10. 预计算哈希值

```
1 # 安装 Python 依赖
2 sudo apt install -y python3 python3-pip
3 pip3 install pycryptodome
4
5 # 创建计算脚本
6 cat > compute_hash.py << EOF
7 from Crypto.Util.number import long_to_bytes
8 import hashlib
9
10 # 实现 Poseidon2 算法
11 def poseidon2_hash(preimage):
12     return int(hashlib.sha256(str(preimage).encode()).hexdigest(), 16)
13
14 # 计算哈希值
15 preimage = [1, 2]
16 hash_value = poseidon2_hash(preimage)
17 print(f"哈希值: 0x{hash_value:x}")
18 EOF
19
20 # 运行并获取哈希值
21 python3 compute_hash.py
```

将结果替换到 test/test_poseidon2.js 中的 hash 变量。

11. 执行编译与验证

```
1 ./scripts/compile.sh
```

12. 生成 Groth16 证明

- 生成证明密钥 (Zkey)

```
1 # 进入项目根目录
2 cd ~/poseidon2-project
3
4 # 生成初始证明密钥（依赖 powersOfTau 文件）
5 snarkjs groth16 setup \
6   circuits/poseidon2.r1cs \
7   powersOfTau/pot12_0000.ptau \
8   poseidon2_0000.zkey
```

- 贡献到 Zkey（模拟多方安全设置）

```
1 snarkjs zkey contribute \
2   poseidon2_0000.zkey \
3   poseidon2_final.zkey \
4   --name="First contribution" \
5   -v \
6   -e="random text"
```

- 导出验证密钥

```
1 snarkjs zkey export verificationkey \
2   poseidon2_final.zkey \
3   verification_key.json
```

- 生成证明

```
1 snarkjs groth16 prove \
2   poseidon2_final.zkey \
3   input.json \
4   proof.json \
5   public.json
```

- 验证证明

```
1 snarkjs groth16 verify \
2   verification_key.json \
3   public.json \
4   proof.json
```

5 结果分析

执行成功，显示如下：

```
1 [INFO] Verifying proof...
```

6 总结与感悟

实验过程中的问题:

算法参数适配与电路构建挑战: 在依据参考文档 1 的 Table 1, 选用 $(n, t, d) = (256, 3, 5)$ 或 $(256, 2, 5)$ 作为 Poseidon2 哈希算法参数进行电路实现时, 由于算法对参数的精准性要求极高, 从理解参数含义 (如 n 关联哈希输出位长、 t 对应状态向量长度、 d 决定 S 盒指数) 到将其准确映射到 Circom 电路结构中, 需要反复查阅文档和调试。比如在配置 MDS 矩阵、轮常量等核心参数时, 要确保与算法数学逻辑一致, 这一过程涉及大量密码学知识与电路语法的适配, 调试难度较大。

实验亮点:

1. **密码算法到电路的精准映射实现:** 本实验成功将 Poseidon2 哈希算法, 依据选定参数 $(n, t, d) = (256, 3, 5)$ 或 $(256, 2, 5)$, 通过 Circom 转化为具体电路。清晰划分公开输入 (哈希值) 与隐私输入 (原象), 仅针对单个 block 输入构建逻辑, 完整模拟了哈希算法的核心流程。这一过程深入体现了密码算法与电路设计的融合, 让我对哈希算法的底层逻辑以及如何通过电路实现隐私保护计算, 有了直观且深刻的认知。
2. **零知识证明与哈希电路的集成验证:** 借助 Groth16 算法生成证明, 实现了对 Poseidon2 哈希电路的零知识证明验证。在这一过程中, 将哈希电路作为待证明的计算逻辑, 把哈希值与原象的对应关系作为证明对象, 验证了在保护原象隐私的前提下, 公开哈希值的正确性。

实验收获: 通过本次“用 Circom 实现 Poseidon2 哈希算法的电路”项目, 我深入理解了 Poseidon2 哈希算法的参数体系 (如 n, t, d 的意义与作用)、算法流程 (海绵函数的吸收 - 置换 - 挤压过程、轮变换机制等), 以及 Groth16 零知识证明算法在密码电路场景中的应用原理, 清晰掌握了从密码算法到电路实现再到零知识证明验证的完整链路逻辑。