Circom 2

单签名 signature

poseidon.circom

```rust
include "circomlib/poseidon.circom";

template SecretToPublic() {
    signal input sk;
    signal output pk;

    component poseidon = Poseidon(1);
    poseidon.inputs[0] <== sk;
    pk <== poseidon.out;
}

template Sign(){
    signal input m;
    signal input sk; // private
    signal input pk;

    // check that we know the secret ket corresponding to the public key.
    component checker = SecretToPublic();
    checker.sk <== sk;
    pk === checker.pk;

    // dummy constraint
    signal mSquared;
    mSquared <== m * m;
}

component main { public [ pk, m ] } = Sign();

/* INPUT ={
     "sk": "5",
     "pk":
"19065150524771031435284970883882288895168425523179566388456001105768498065277",
     "m": "1"
} */
```

此时, 只有 "sk" 和 "pk" 匹配成功 , 才能通过这个电路的验证 ;
比如随便修改 pk 中的一个字符, "sk" 便 无法和 "pk" 匹配

群签名 group signature

群签名（Group signatures）是一种加密技术，它允许一个成员集合中的某个成员使用匿名身份在该集合内签署消息，同时不透露其真实身份。与传统数字签名不同，群签名不需要签名者透露其身份，因此在一些应用场景中具有重要的作用。

在群签名中，一个群成员可以使用自己的私钥签署一条消息，生成一个群签名。其他人可以验证签名，但无法确定签名的具体来源是哪个成员。与传统数字签名不同的是，群签名允许签名者在不暴露自己身份的情况下进行签名，这意味着任何一个成员都有可能是签名者。

群签名有很多应用场景，比如匿名投票、匿名投诉和匿名举报。在这些场景中，需要确保成员的身份不会被暴露，同时需要保证签名的有效性。群签名技术能够满足这些需求。

然而，群签名技术也存在一些安全性问题，如群成员可能会伪造签名、签名者可能会被追踪等。因此，研究人员一直在致力于解决这些问题，并提出了许多改进的方案来增强群签名的安全性和可靠性。

generate a bunch of different Keys

自己搜索默克尔树的构造 , 一言以蔽之: 在二叉树上，每个(父)节点都是它的子节点的哈希值的哈希，这样构造出来的树叫做 Merkle 树

下面代码：

- 目的: 验证给定的私钥 `sk` 是否对应于公钥数组 `pk[n]` 中的某公钥， `pk[n]` 中的公钥属于同一 group
- `signal input sk`: 输入 1 个私钥 sk
- `signal input pk[n]`: 输入 1 组公钥数组 `pk[n]`， `n` 是常数，表示组内公钥的数量
- `signal input m`: 待被签名的消息
- `component computePk = SecretToPublic()`: 上面已经定义了，用于计算私钥 `sk` 对应的公钥 `pk`
- `computePk.sk <== sk`: 将输入的私钥 `sk` 传递给计算计算组件 `computePk`
- `computePk.pk <== sk`: 私钥对应的公钥 (计算结果)
- `signal zeroChecker[n+1]`: 中间变量数组，会被用于检查计算出的公钥是否隶属于公钥数组 `pk[n]`
- `zeroChecker[0] <== 1`: 将 `zeroChecker[0]` 的值设置为 `1` , 类似于一个 flag 标志
- `pk[i] - computePk.pk` : 如果私钥 `sk` 对应的公钥 `computePk.pk` 在这个数组中, 则该计算结果为 0 , 则下一位的 `zeroChecker` 就会被记录为 0 ; 该标志位会被传递下去 , 直到遍历结束
- `zeroChecker[n] === 0`: 检查被传递的标志位 :
  - 如果为仍为 1 , 说明公钥数组 Group 中没有任何一个 PK 能与该私钥 `sk` 的公钥匹配 , 此时程序会 `Assert` 报错 ;
  - 如果该标志位为 0, 说明 Group 中曾有一个 `PK'` 与 `sk` 的公钥匹配 , 且将该标志位传递了下来 ;

代码详情：

```
include "circomlib/poseidon.circom";

template SecretToPublic() {
    signal input sk;
    signal output pk;

    component poseidon = Poseidon(1);
    poseidon.inputs[0] <== sk;
    pk <== poseidon.out;
}

template GroupSign(n) {
    signal input sk;
    signal input pk[n];

    // even though m is not involved in the circuit,
    // it is still constrained and cannot be changed after it is set.
    signal input m;

    // get the public key
    component computePk = SecretToPublic();
    computePk.sk <== sk;

    signal zeroChecker[n+1];
    zeroChecker[0] <== 1;
    for (var i = 0; i < n; i++) {
        zeroChecker[i+1] <== zeroChecker[i] * (pk[i] - computePk.pk);
    }
    zeroChecker[n] === 0;
}

component main { public [ pk, m ] } = GroupSign(5);
/* INPUT ={
     "sk": "5",
     "pk": ["19065150524771031435284970883882288895168425523179566388456001105768498065277",
"1", "2", "3", "4"],
     "m": "12345678"
} */
```

Larger groups (with Merkle Trees)

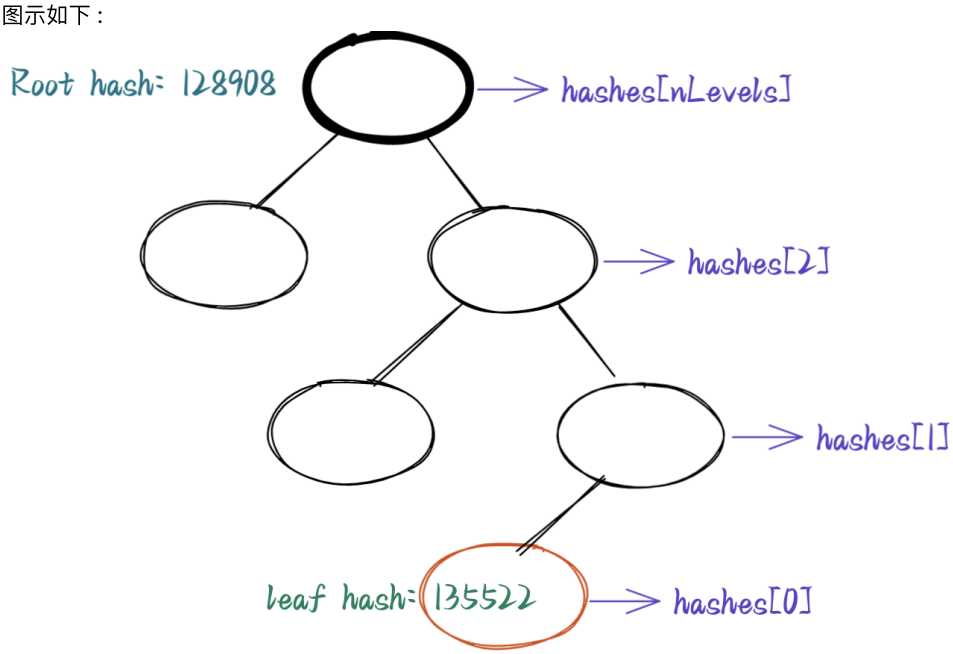
上面的群签名方案需要输入**所有的公钥**作为输入，对于非常大的数组来说很笨拙。

使用 Merkle Tree (默克尔树)，则只需要向电路中输入 `log(n)` 个元素即可证明成员资格：

```
include "circomlib/poseidon.circom";

// if s == 0 returns [in[0], in[1]]
// if s == 1 returns [in[1], in[0]]
template DualMux() {
    signal input in[2];
    signal input s;
    signal output out[2];

    s * (1 - s) === 0;
    out[0] <== (in[1] - in[0])*s + in[0];
    out[1] <== (in[0] - in[1])*s + in[1];
}

template MerkleTreeInclusionProof(nLevels) {
    signal input leaf;
    signal input pathIndices[nLevels];
    signal input siblings[nLevels];
    signal input root;

    component mux[nLevels];
    component poseidons[nLevels];

    signal hashes[nLevels+1];
    hashes[0] <== leaf;

    for (var i = 0; i < nLevels; i++) {
        mux[i] = DualMux();
        mux[i].in[0] <== hashes[i];
        mux[i].in[1] <== siblings[i];
        mux[i].s <== pathIndices[i];

        poseidons[i] = Poseidon(2);
        poseidons[i].inputs[0] <== mux[i].out[0];
        poseidons[i].inputs[1] <== mux[i].out[1];
        hashes[i+1] <== poseidons[i].out;
    }

    root === hashes[nLevels];
}

component main { public [ leaf, root ] } = MerkleTreeInclusionProof(15);


/* INPUT = {
    "root": "12890874683796057475982638126021753466203617277177808903147539631297044918772",
    "leaf": "135522435269582748397508080717826040336574853040 7",
    "siblings": [
        "1",
        "21723437734888465469187937751879432385729494715149027879071080937632563980 9",
        "18624361856574916496058203820366795950790078780687078257641649903530959943449",
        "19831903348221211061287449275113949495274937755341117892716020320428427983768",
        "51013616581647838001629502779649470865223843652071512830799097453625461778 17",
        "11552819453851113656956689238827707323483753486799384854128595967739676085386",
        "10483540708739576660440356112223782712680507694971046950485797346645134034053",
        "73899295642479071652218177429238034675665522739180716304422193444968521418 97",
        "63734674040374221986968505919612701979482593937357565053501733024607613915 61",
        "14340012938942512497418634250250812329499499250184704496617019030530171289909",
        "10566235887680695760439252521824446945750533956882759130656396012316506290852",
        "14058207238811178801861080665931986752520779251556785412233046706263822020051",
```

```
        "18418048571463388765026032114737954825675744290389480824064702827977101122 30",
        "60689746712777519469413563303146253359245229737075043162172019138313932583 19",
        "10344803844228993379415834281058662700959138333457605334309913075063427817 480"
    ],
    "pathIndices": [
        "1",
        "1",
        "1",
        "1",
        "1",
        "1",
        "1",
        "1",
        "1",
        "1",
        "1",
        "1",
        "1",
        "1",
        "1"
    ]
} */
```

代码实现的功能图示如下：



下面我们来一段一段看代码

DualMux

`DualMux` 是一个 **双路复用器**，用于实现路径索引选择（path index selector），用于选择两个输入信号中的一个

其中，输入端口 `in[0]` 和 `in[1]` 分别连接两个输入信号，选择信号 `s` 用于控制复用器选择哪一个输入信号，输出信号 `out[0]` 和 `out[1]` 分别输出被选中的信号。

注释里也提到了：

1. 如果 `s == 0` , 则 return `[in[0], in[1]]`
2. 如果 `s == 1` , 则 return `[in[1], in[0]]`

```
// if s == 0 returns [in[0], in[1]]
// if s == 1 returns [in[1], in[0]]
template DualMux() {
    signal input in[2];
    signal input s;
    signal output out[2];
```

```
    s * (1 - s) === 0;
    out[0] <== (in[1] - in[0])*s + in[0];
    out[1] <== (in[0] - in[1])*s + in[1];
}
```

MerkleTreeInclusionProof

核心变量定义：

- 四个输入信号：`leaf`、`pathIndices[]`、`siblings[]` 和 `root`。
  - `leaf` 表示待验证的叶子节点；
  - `pathIndices[nLevels]` 是一组节点-路径索引
    - 从输入可以看到，`pathIndices[] = [1,1,1, .., 1]` 是由 `[0, 1]` 构成的，`[0, 1]` 表示在该层中选择左子节点还是右子节点。因此，`pathIndices[0] == 0` 表示从根节点到第一层时选择右子节点，`pathIndices[1] == 0` 表示从第一层到第二层时选择了右子节点 ……
    - 在本例的输入中，由于所有 `pathIndices` 中的元素都是 `1`，这意味着从根节点到叶子节点的路径上只选择了右侧子节点。这是 Merkle 树中一种可能的情况，因为 Merkle 树可以有不同的结构，因此路径可能是不同的 .
  - `siblings[nLevels]` 表示路径上的兄弟节点
  - `root` 表示该树的根节点。
  - `nLevels` 是树的高度
- `component mux[nLevels]` :
  - 定义 `mux` 双路复用器**数组**，数组中每个元素都将被赋值为 1 个 `DualMux` 组件。其 length 是 `nLevels`，表示树的高度。
  - `DualMux` 组件的作用是根据 路径索引值 选择哈希值或者兄弟节点，以便计算下一级的哈希值
-

核心流程：

1. `signal hashes[nLevels+1];` 中间临时数组, 用来储存当前计算层的节点 Hash , 最终计算到顶层节点时, 计算该节点的 Hash 是否等于 Root 节点的 Hash ;
2. 进入 `for` 循环，方向是从下向上(从 Leaf 向 Root) 遍历树的每一层 :
   1. `mux[i]` 的输入 0 为 `hashes[i]`，输入 1 为 `siblings[i]`，s 为 `pathIndices[i]` → 选择右侧节点
   2. `poseidons[i] = Poseidon(2)` 表示初始化一个接受 2 个 input 的 Hash 函数, 然后将 `mux[i]` 的输出 0 和输出 1 分别作为 `poseidons[i]` 的输入 0 和输入 1，最后将 `poseidons[i]` 的输出作为 `hashes[i+1]`
   3. `hashes[i]` 更新为 2 个子节点的父节点的哈希值, 进入 next loop ...
   4. 最终循环计算到最顶层节点时, 观察该节点计算出的 Hash 是否等于 Root 节点的 Hash : 如果相等则无事发生, 如果不等, 说明该节点不是 Merkle 树的一部分, 电路 Assert Error !
   5. 总结下, 这个 `for` 循环的作用是由叶到根逐级计算 `Merkle` 树的哈希，以构建 `Merkle` 树的完整结构并最终验证一个给定的叶子节点是否属于该 Merkle 树。

```
template MerkleTreeInclusionProof(nLevels) {
    signal input leaf;
    signal input pathIndices[nLevels];
    signal input siblings[nLevels];
    signal input root;

    component mux[nLevels];
    component poseidons[nLevels];

    signal hashes[nLevels+1];
    hashes[0] <== leaf;

    for (var i = 0; i < nLevels; i++) {
        mux[i] = DualMux();
        mux[i].in[0] <== hashes[i];
        mux[i].in[1] <== siblings[i];
        mux[i].s <== pathIndices[i];

        poseidons[i] = Poseidon(2);
        poseidons[i].inputs[0] <== mux[i].out[0];
        poseidons[i].inputs[1] <== mux[i].out[1];
```

```
        hashes[i+1] <== poseidons[i].out;
    }

    root === hashes[nLevels];
}
```

代码部分

poseidon.circom / poseidon hash:

- https://github.com/0xPARC/zkrepl/blob/eddb8a84ada7eb6242933f08d3ce36596fb1a16f/src/data/example.circom

https://github.com/0xPARC/circom-starter/tree/master/circuits :

- division.circom
- hash.circom
- simple-polynomial.circom