zk-snark in real world

: Tornado Cash with circom code

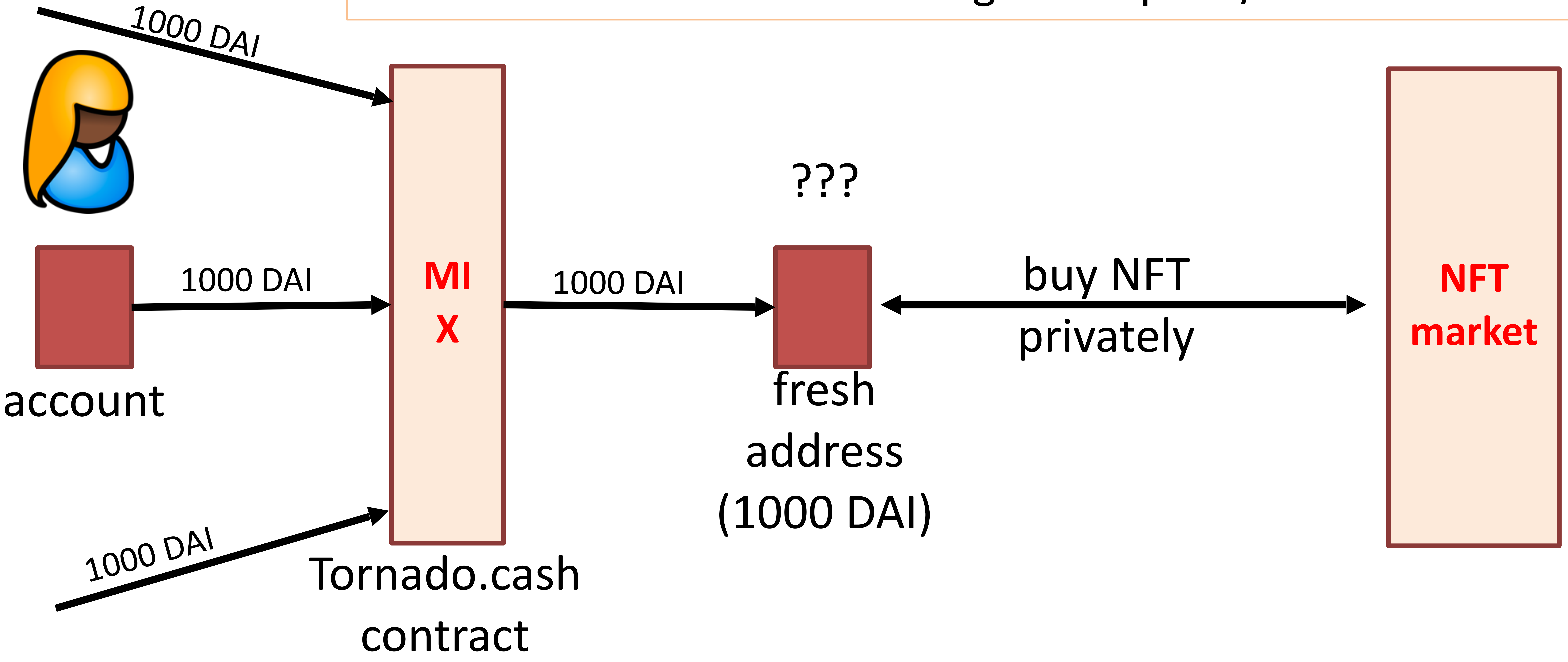
23.04.06
zk-school: biginner class
Seohee Park

Tornado cash: a zk-based mixer

Launched on the Ethereum blockchain on May 2020 (v2)

Tornado Cash: a ZK-mixer

A common denomination (1000 DAI) is needed to prevent linking
Alice to her fresh address using the deposit/withdrawal amount

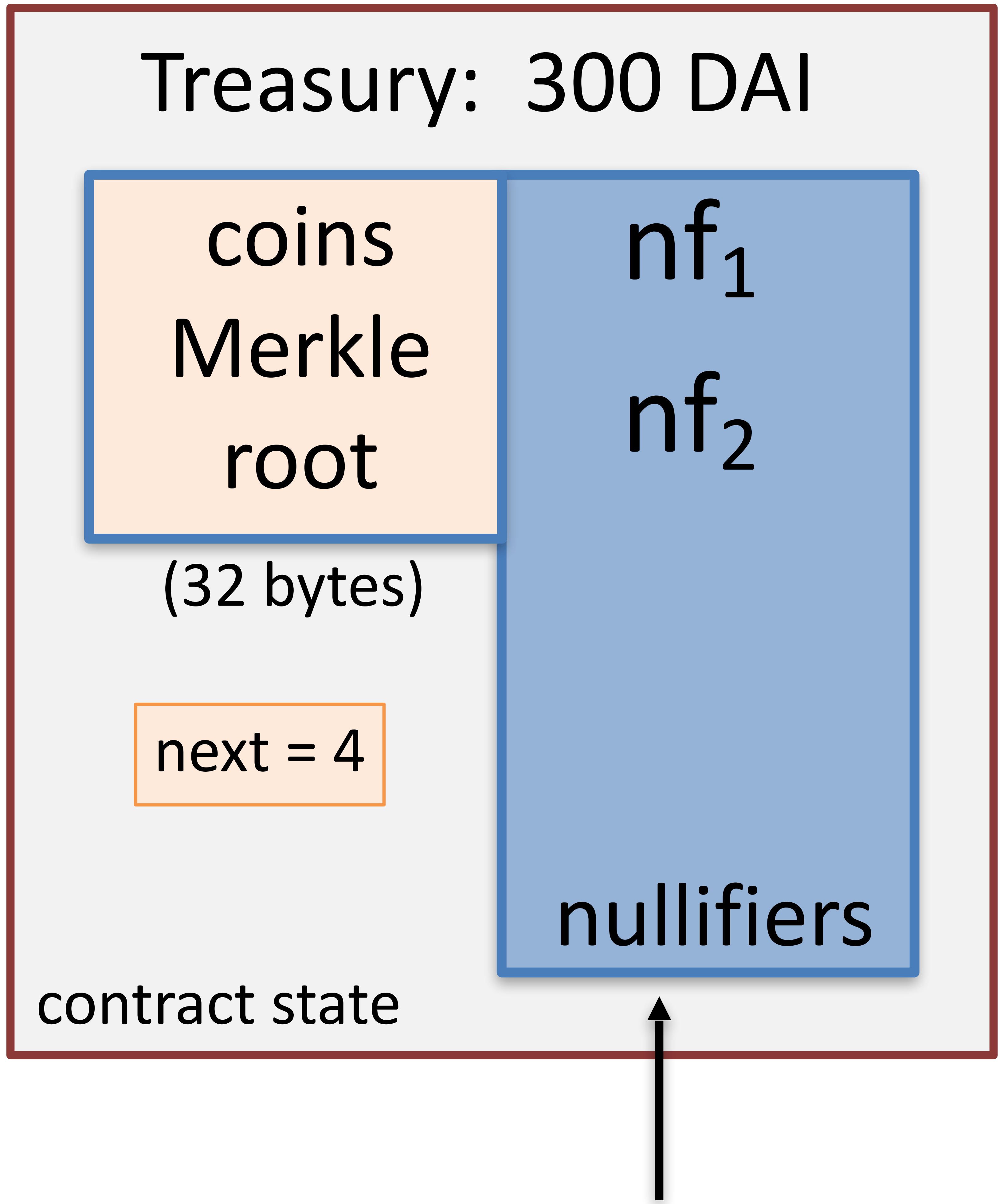


The tornado cash contract (simplified)

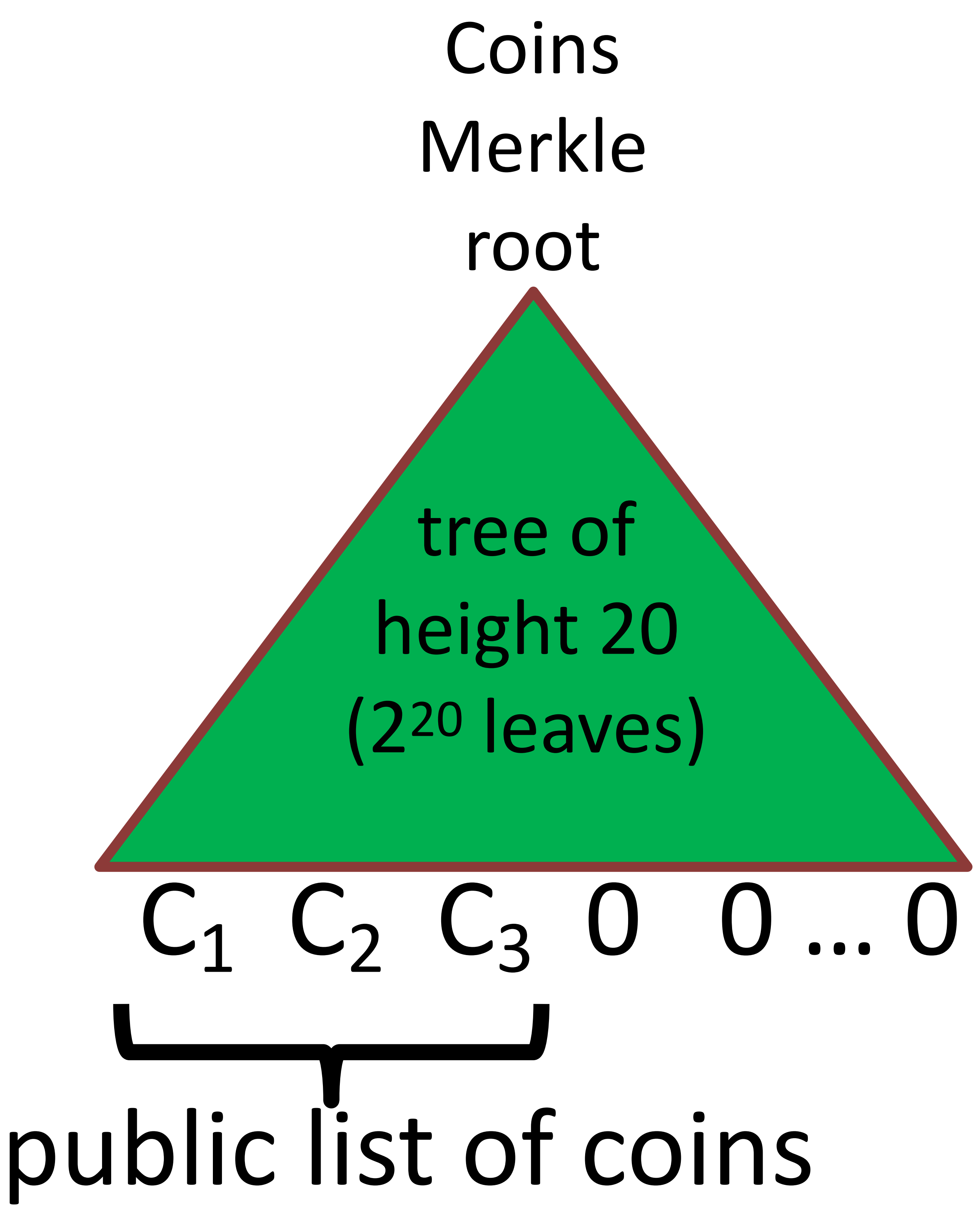
H_1, H_2: $R \rightarrow \{0,1\}^{256}$ CRHF

100 DAI pool:
each coin = 100 DAI

Currently:
- three coins in pool
- contract has 300 DAI
- two nullifiers stored



explicit list:
one entry per **spent coin**

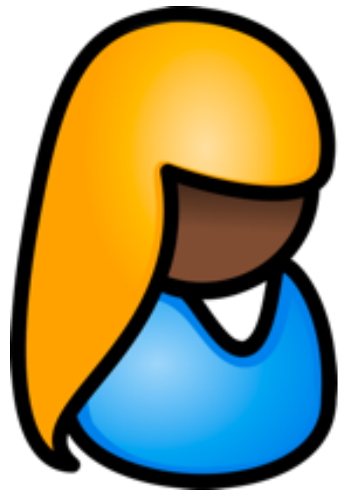


Tornado cash: deposit (simplified)

<u>100 DAI pool</u>:

each coin = 100 DAI

<u>Alice deposits 100 DAI:</u>



100 DAI →

C_4 , MerkleProof(4)

Build Merkle proof for leaf #4:
MerkleProof(4) (leaf=0)

choose random k, r in R

set $C_4 = H_1(k, r)$

Treasury: 300 DAI

coins Merkle root	nf_1 nf_2 nullifiers

(32 bytes)

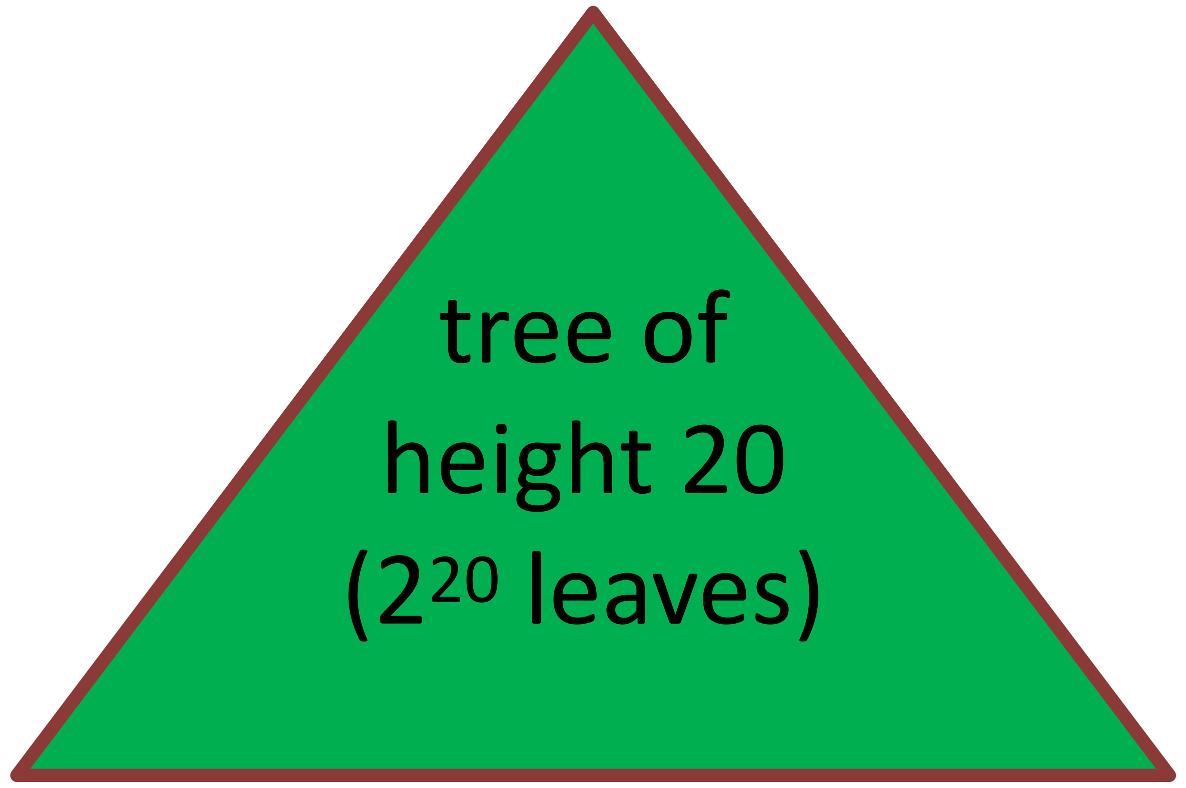
next = 4

contract state

explicit list:
one entry per **spent coin**

H_1, H_2: $R \rightarrow \{0,1\}^{256}$

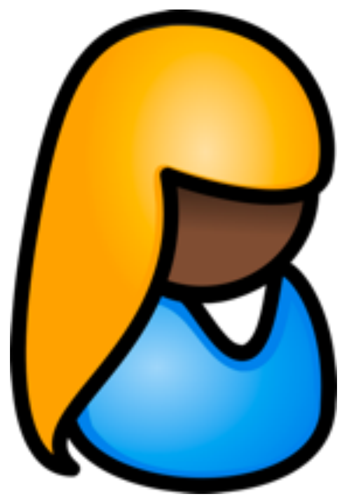
Coins Merkle root



C_1 C_2 C_3 0 0 … 0

$\underbrace{C_1 \; C_2 \; C_3}_{\text{public list of coins}}$

Tornado cash: deposit (simplified)



100 DAI

C_4 , MerkleProof(4)

coins Merkle root

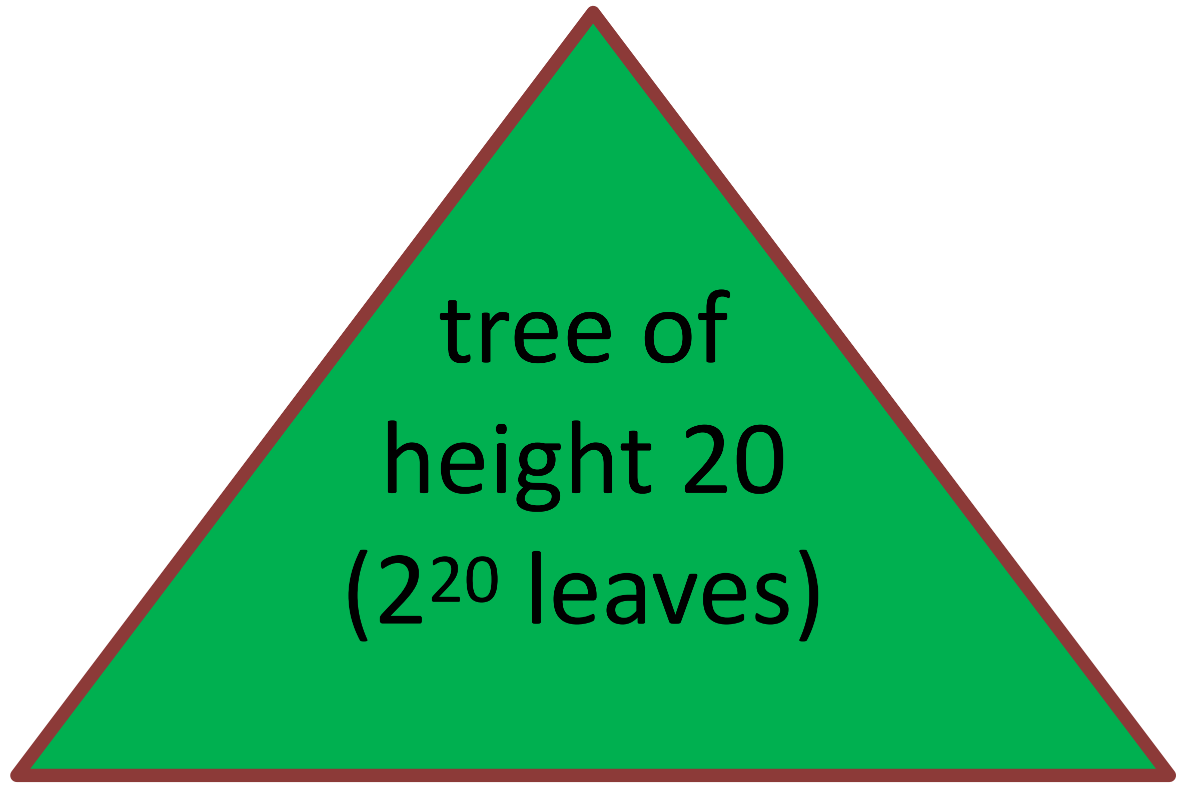
(32 bytes)

next = 4

Tornado contract

H_1, H_2: $R \rightarrow \{0,1\}^{256}$

Coins Merkle root



C_1 C_2 C_3 0 0 … 0

public list of coins

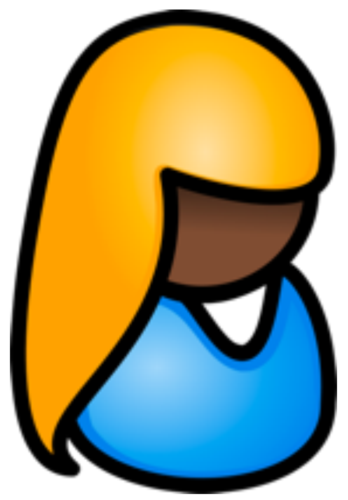
Tornado contract does:

(1) verify MerkleProof(4) with respect to current stored root

(2) use C_4 and MerkleProof(4) to compute updated Merkle root

(3) update state

Tornado cash: deposit (simplified)



100 DAI

C_4 , MerkleProof(4)

coins Merkle root

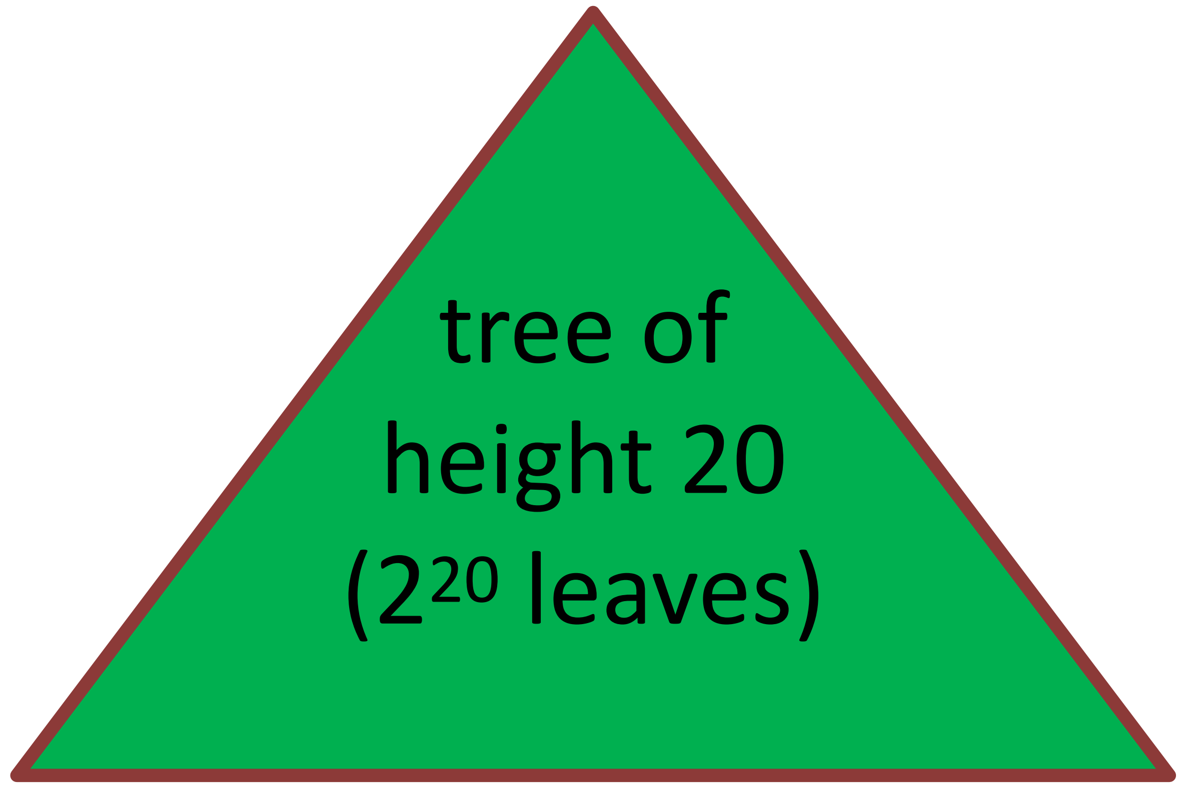
(32 bytes)

next = 4

Tornado contract

H_1, H_2: $R \rightarrow \{0,1\}^{256}$

updated Merkle root



$\underbrace{C_1 \;\; C_2 \;\; C_3 \;\; C_4}_{\text{public list of coins}} \;\; 0 \ldots 0$

Tornado contract does:

(1) verify MerkleProof(4) with respect to current stored root

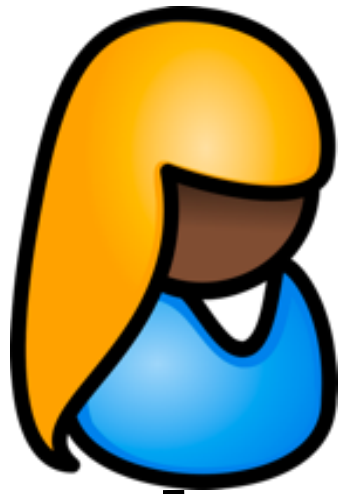
(2) use C_4 and MerkleProof(4) to compute updated Merkle root

(3) update state

Tornado cash: deposit (simplified)

100 DAI pool:
each coin = 100 DAI

Alice deposits 100 DAI:



100 DAI →
C_4 , MerkleProof(4)

note: (k, r)
Alice keeps secret
(one note per coin)

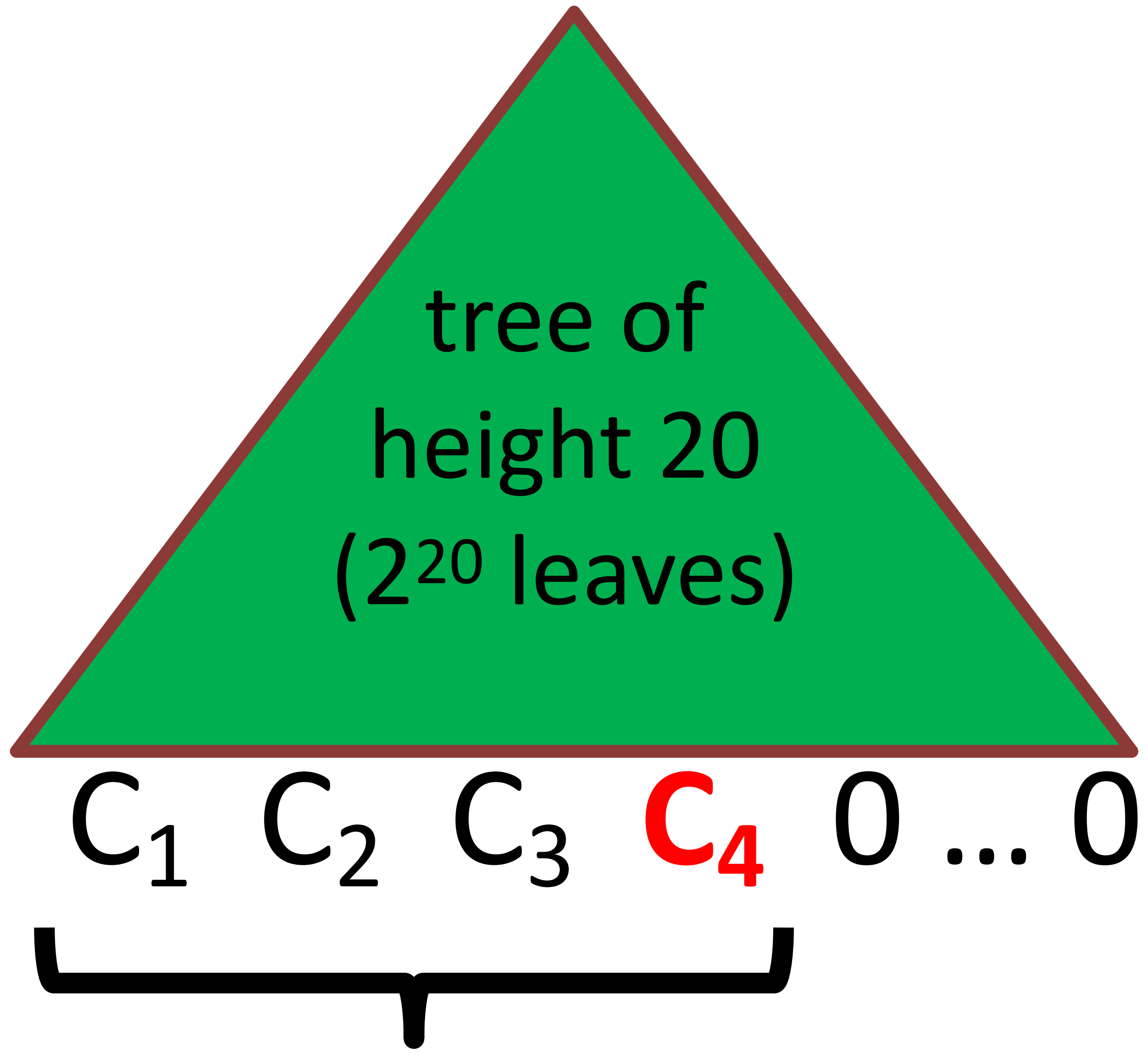
Treasury: **400** DAI

updated Merkle root (32 bytes)	nullifiers
updated Merkle root	nf_1
(32 bytes)	nf_2
next = **5**	

updated contract state

Every deposit: new Coin added sequentially to tree

updated Merkle root



public list of coins

an observer sees who owns which leaves

Tornado cash: withdrawal (simplified)

100 DAI pool:
each coin = 100 DAI

Withdraw coin #3 to addr A:

has note= (k’, r’)

set **nf** = $H_2(k')$

Treasury: **400** DAI

coins Merkle root

(32 bytes)

next = 5

nf_1

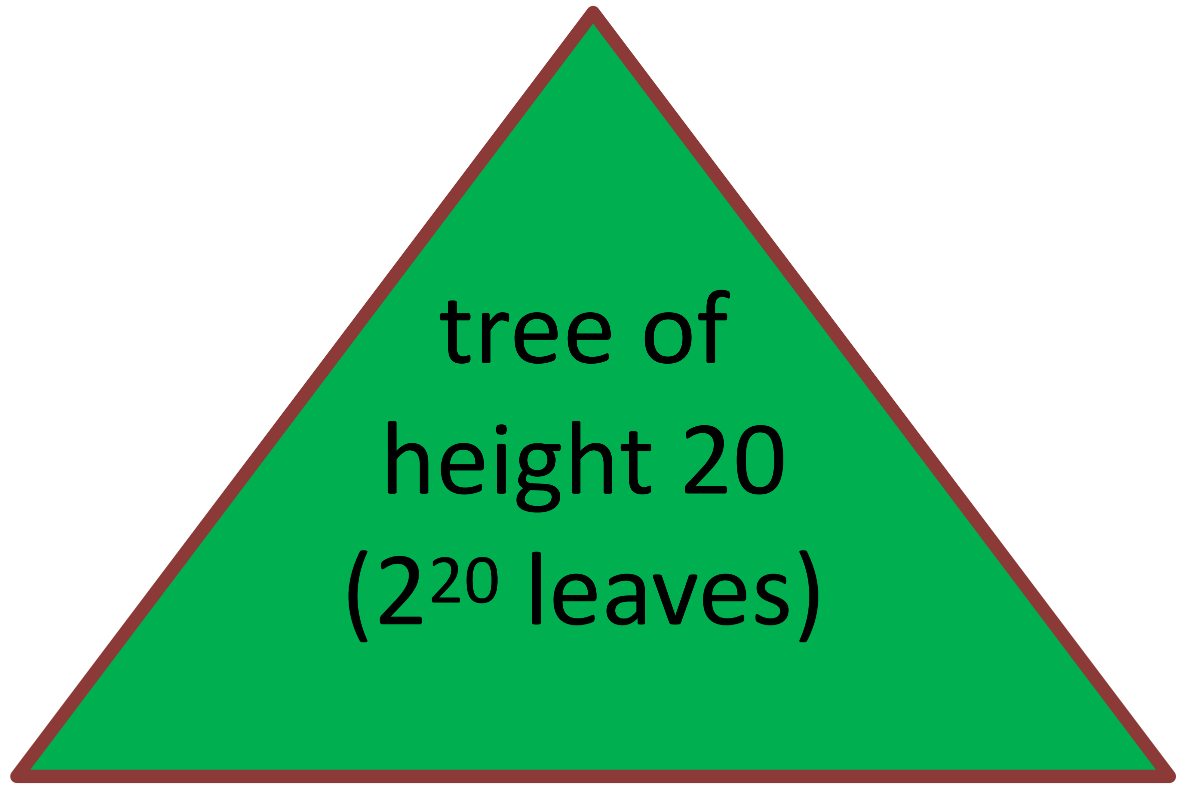
nf_2

nullifiers

contract state

H_1, H_2: $R \rightarrow \{0,1\}^{256}$

Merkle root



C_1 C_2 **C_3** C_4 0 … 0

public list of coins

Bob proves “I have a note for some leaf in the coins tree, and its nullifier is **nf**”
(without revealing which coin)

Tornado cash: withdrawal (simplified)

H_1, H_2: $R \to \{0,1\}^{256}$

Withdraw coin #3 to addr A:

has note= (k’, r’)   set **nf** = H_2(k’)

Bob builds zk-SNARK proof π for

public statement x = (**root**, **nf**, **A**)

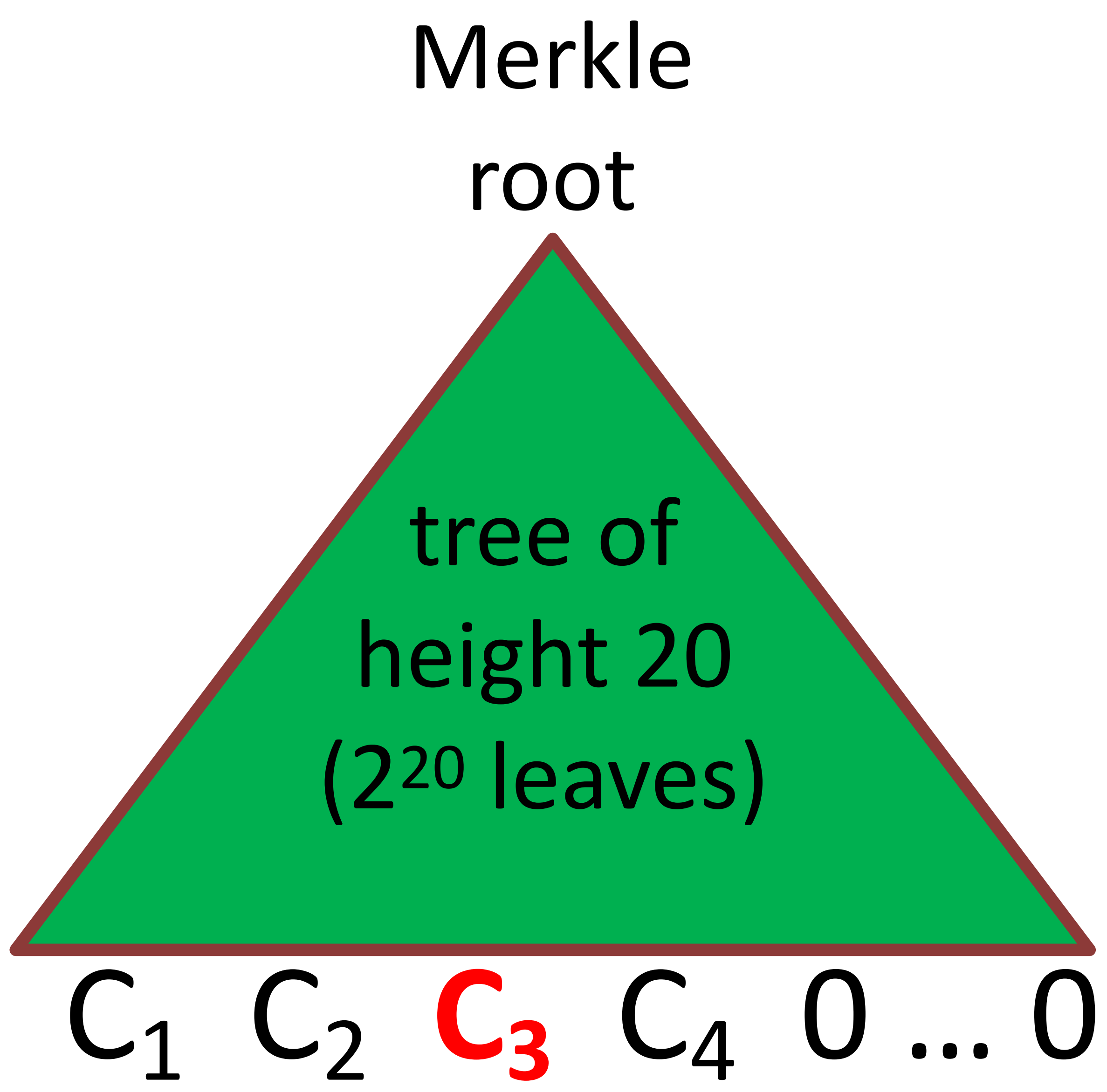
secret witness w = (k’, r’, C_3, MerkleProof(C_3))

where Circuit(x,w)=0 iff:

(i) C_3 = (leaf #3 of **root**), i.e. MerkleProof(C_3) is valid,

(ii) $C_3 = H_1$(k’, r’), and

(iii) **nf** = H_2(k’).



(address A not used in Circuit)

Tornado cash: withdrawal (simplified)

$H_1, H_2\colon R \rightarrow \{0,1\}^{256}$

Withdr

The address A is part of the statement to ensure that a miner cannot change A to its own address and steal funds

Assumes the SNARK is **non-malleable**:

adversary cannot use proof π for x to build a proof π' for some “related” x’

(e.g., where in x’ the address A is replaced by some A’)

C_1 C_2 C_3 C_4 0 … 0

Bob builds zk-SNARK proof π for

public statement x = (**root**, **nf**, **A**)

secret witness w = (k’, r’, C_3, MerkleProof(C_3))

Tornado cash: withdrawal (simplified)

100 DAI pool:
each coin = 100 DAI

Withdraw coin #3 to addr A:

nf, proof π, **A**

(over Tor)

Bob's ID and coin C_3 are not revealed

Treasury: **400** DAI

coins Merkle root

(32 bytes)

next = 5

nf_1

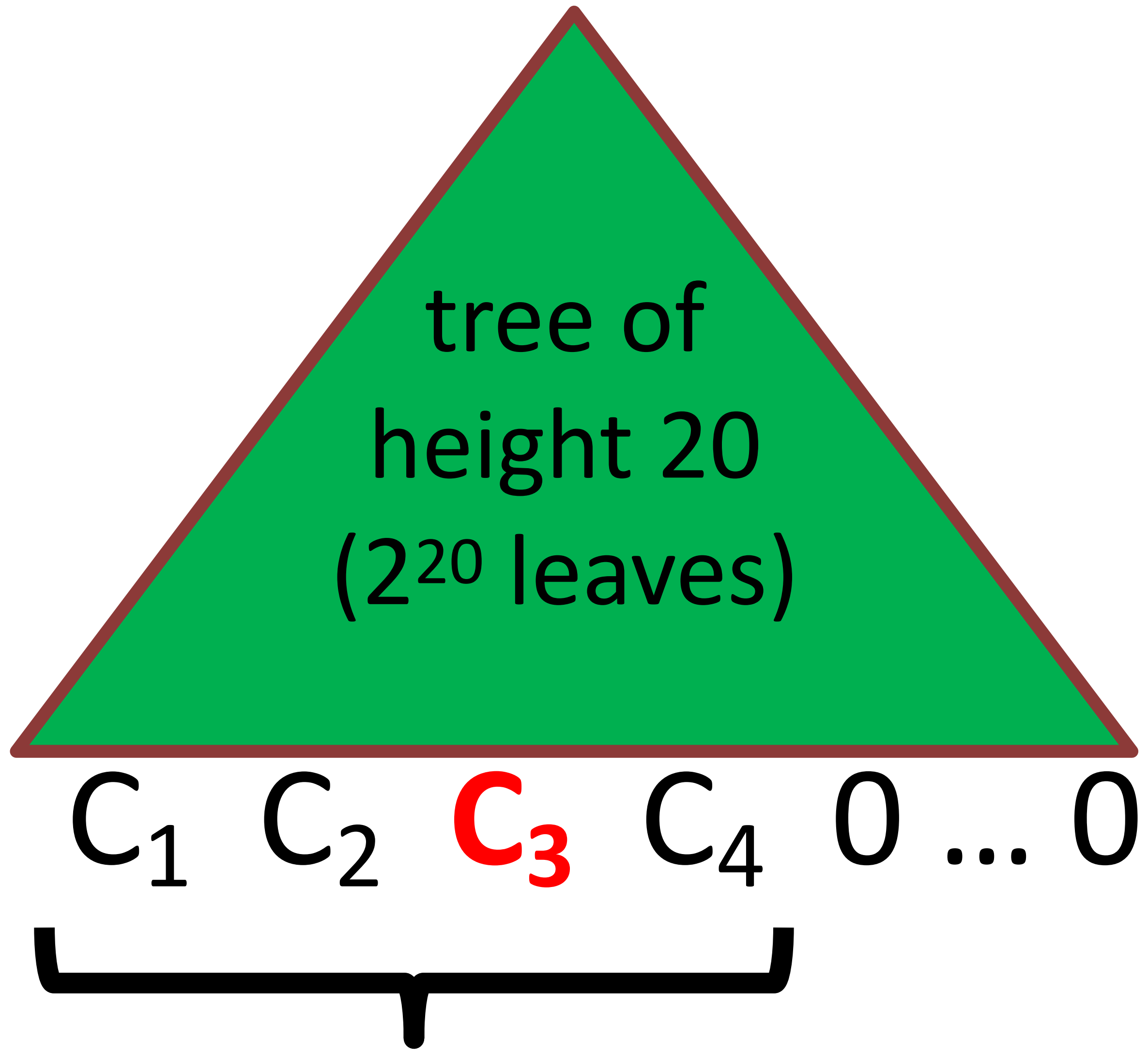
nf_2

nullifiers

contract state

H_1, H_2: $R \rightarrow \{0,1\}^{256}$

Merkle root



public list of coins

Contract checks (i) proof π is valid for (root, **nf,** **A**), and (ii) **nf** is not in the list of nullifiers

Tornado cash: withdrawal (simplified)

100 DAI pool:
each coin = 100 DAI

Withdraw coin #3 to addr A:

nf, proof π, **A**
(over Tor)

100 DAI
to address **A**

Treasury: **300** DAI

coins Merkle root
(32 bytes)

nf_1
nf_2
nf

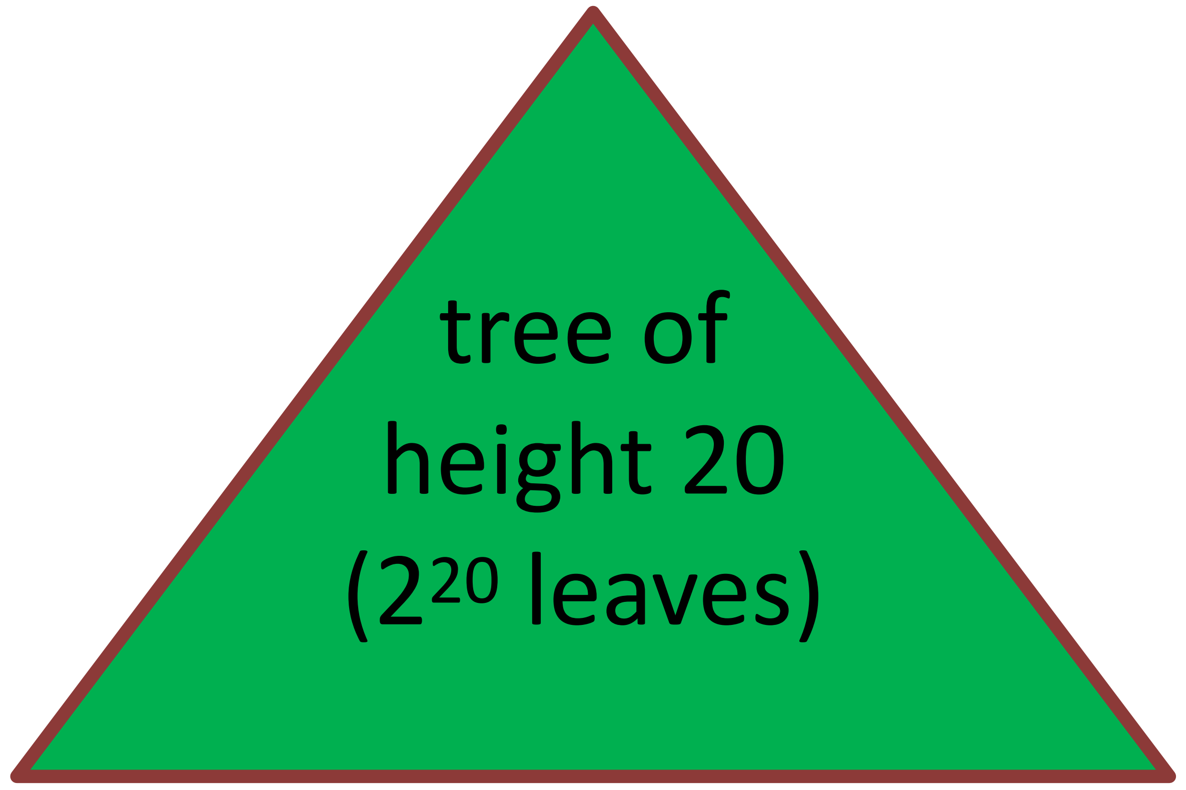
nullifiers

next = 5

contract state

H_1, H_2: $R \rightarrow \{0,1\}^{256}$

Merkle root



C_1 C_2 C_3 C_4 0 … 0

public list of coins
… but observer does not know which are spent

nf and π reveal nothing about which coin was spent.

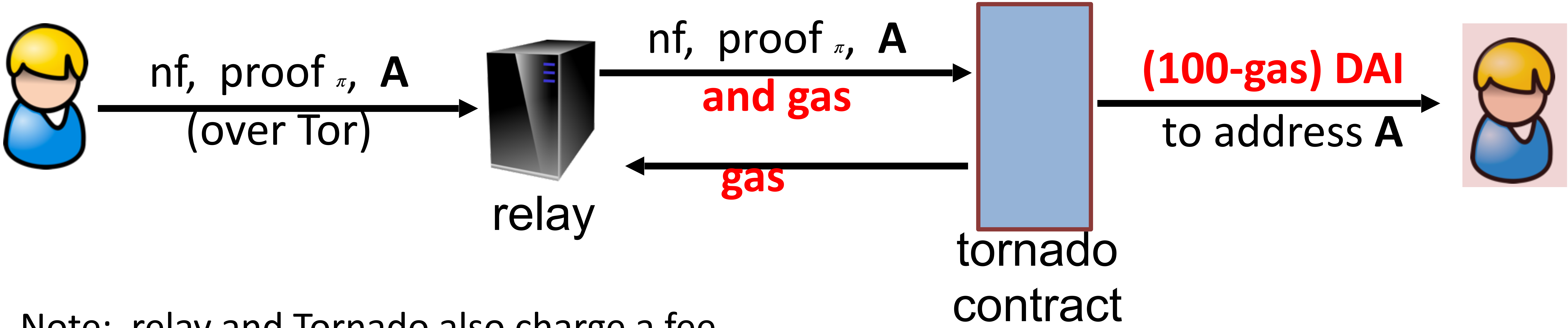
But, coin #3 cannot be spent again, because **nf = $H_2(k')$** is now nullified.

Who pays the withdrawal gas fee?

Problem: how does Bob pay for gas for the withdrawal Tx?

- If paid from Bob's address, then fresh address is linked to Bob

Tornado's solution: **Bob uses a relay**



Note: relay and Tornado also charge a fee

Withdraw coin #3 to addr A:

has note= (k', r') set **nf** $= H_2(k')$

Bob builds zk-SNARK proof π for

public statement $x = (\textbf{root}, \textbf{nf}, \textbf{A})$

secret witness $w = (k', r', C_3, \text{MerkleProof}(C_3))$

where Circuit(x,w)=0 iff:

(i) C_3 = (leaf #3 of **root**), i.e. MerkleProof(C_3) is valid,

(ii) $C_3 = H_1(k', r')$, and

(iii) **nf** $= H_2(k')$.

withdrow.circom

```
// computes Pedersen(nullifier + secret)
template CommitmentHasher() {
    signal input nullifier;
    signal input secret;
    signal output commitment;
    signal output nullifierHash;

    component commitmentHasher = Pedersen(496);
    component nullifierHasher = Pedersen(248);
    component nullifierBits = Num2Bits(248);
    component secretBits = Num2Bits(248);
    nullifierBits.in <== nullifier;
    secretBits.in <== secret;
    for (var i = 0; i < 248; i++) {
        nullifierHasher.in[i] <== nullifierBits.out[i];
        commitmentHasher.in[i] <== nullifierBits.out[i];
        commitmentHasher.in[i + 248] <== secretBits.out[i];
    }

    commitment <== commitmentHasher.out[0];
    nullifierHash <== nullifierHasher.out[0];
}
```

withdraw.circom

public statement

secret witness

```
// Verifies that commitment that corresponds to given secret and nullifier is included in the merkle tree of deposits
template Withdraw(levels) {
    signal input root;
    signal input nullifierHash;
    signal input recipient; // not taking part in any computations
    signal input relayer;  // not taking part in any computations
    signal input fee;      // not taking part in any computations
    signal input refund;   // not taking part in any computations
    signal private input nullifier;
    signal private input secret;
    signal private input pathElements[levels];
    signal private input pathIndices[levels];

    component hasher = CommitmentHasher();
    hasher.nullifier <== nullifier;
    hasher.secret <== secret;
    hasher.nullifierHash === nullifierHash;

    component tree = MerkleTreeChecker(levels);
    tree.leaf <== hasher.commitment;
    tree.root <== root;
    for (var i = 0; i < levels; i++) {
        tree.pathElements[i] <== pathElements[i];
        tree.pathIndices[i] <== pathIndices[i];
    }

    // Add hidden signals to make sure that tampering with recipient or fee will invalidate the snark proof
    // Most likely it is not required, but it's better to stay on the safe side and it only takes 2 constraints
    // Squares are used to prevent optimizer from removing those constraints
    signal recipientSquare;
    signal feeSquare;
    signal relayerSquare;
    signal refundSquare;
    recipientSquare <== recipient * recipient;
    feeSquare <== fee * fee;
    relayerSquare <== relayer * relayer;
    refundSquare <== refund * refund;
}

component main = Withdraw(20);
```

https://github.com/tornadocash/tornado-core/tree/master/circuits

Withdraw coin #3 to addr A:

has note= (k′, r′)   set **nf** = H_2(k′)

Bob builds zk-SNARK proof π for
public statement x = (**root**, **nf**, **A**)
secret witness w = (k′, r′, C_3, MerkleProof(C_3))

where Circuit(x,w)=0 iff:

(i) C_3 = (leaf #3 of **root**), i.e. MerkleProof(C_3) is valid,

(ii) C_3 = H_1(k′, r′), and

(iii) **nf** = H_2(k′).

withdraw.circom

```
// Verifies that commitment that corresponds to given secret and nullifier is included in the merkle tree of deposits
template Withdraw(levels) {
    signal input root;
    signal input nullifierHash;
    signal input recipient; // not taking part in any computations
    signal input relayer;  // not taking part in any computations
    signal input fee;      // not taking part in any computations
    signal input refund;   // not taking part in any computations
    signal private input nullifier;
    signal private input secret;
    signal private input pathElements[levels];
    signal private input pathIndices[levels];

    component hasher = CommitmentHasher();
    hasher.nullifier <== nullifier;
    hasher.secret <== secret;
    hasher.nullifierHash === nullifierHash;

    component tree = MerkleTreeChecker(levels);
    tree.leaf <== hasher.commitment;
    tree.root <== root;
    for (var i = 0; i < levels; i++) {
        tree.pathElements[i] <== pathElements[i];
        tree.pathIndices[i] <== pathIndices[i];
    }

    // Add hidden signals to make sure that tampering with recipient or fee will invalidate the snark proof
    // Most likely it is not required, but it's better to stay on the safe side and it only takes 2 constraints
    // Squares are used to prevent optimizer from removing those constraints
    signal recipientSquare;
    signal feeSquare;
    signal relayerSquare;
    signal refundSquare;
    recipientSquare <== recipient * recipient;
    feeSquare <== fee * fee;
    relayerSquare <== relayer * relayer;
    refundSquare <== refund * refund;
}

component main = Withdraw(20);
```

public statement

secret witness

mercleTree.circom

```
template MerkleTreeChecker(levels) {
    signal input leaf;
    signal input root;
    signal input pathElements[levels];
    signal input pathIndices[levels];

    component selectors[levels];
    component hashers[levels];

    for (var i = 0; i < levels; i++) {
        selectors[i] = DualMux();
        selectors[i].in[0] <== i == 0 ? leaf : hashers[i - 1].hash;
        selectors[i].in[1] <== pathElements[i];
        selectors[i].s <== pathIndices[i];

        hashers[i] = HashLeftRight();
        hashers[i].left <== selectors[i].out[0];
        hashers[i].right <== selectors[i].out[1];
    }

    root === hashers[levels - 1].hash;
}
```

https://github.com/tornadocash/tornado-core/tree/master/circuits

Withdraw coin #3 to addr A:

has note= (k’, r’)   set **nf** = H_2(k’)

Bob builds zk-SNARK proof π for

public statement x = (**root**, **nf**, **A**)

secret witness w = (k’, r’, C_3, MerkleProof(C_3))

where Circuit(x,w)=0 iff:

(i) C_3 = (leaf #3 of **root**), i.e. MerkleProof(C_3) is valid,

(ii) C_3 = H_1(k’, r’), and

(iii) **nf** = H_2(k’).

withdraw.circom

public statement

secret witness

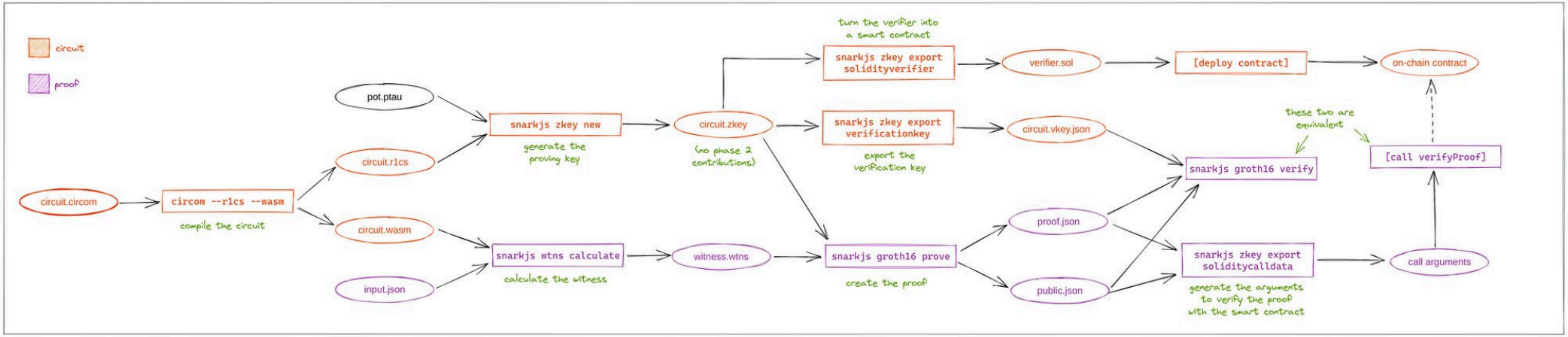
```
// Verifies that commitment that corresponds to given secret and nullifier is included in the merkle tree of deposits
template Withdraw(levels) {
    signal input root;
    signal input nullifierHash;
    signal input recipient; // not taking part in any computations
    signal input relayer;  // not taking part in any computations
    signal input fee;      // not taking part in any computations
    signal input refund;   // not taking part in any computations
    signal private input nullifier;
    signal private input secret;
    signal private input pathElements[levels];
    signal private input pathIndices[levels];

    component hasher = CommitmentHasher();
    hasher.nullifier <== nullifier;
    hasher.secret <== secret;
    hasher.nullifierHash === nullifierHash;

    component tree = MerkleTreeChecker(levels);
    tree.leaf <== hasher.commitment;
    tree.root <== root;
    for (var i = 0; i < levels; i++) {
        tree.pathElements[i] <== pathElements[i];
        tree.pathIndices[i] <== pathIndices[i];
    }

    // Add hidden signals to make sure that tampering with recipient or fee will invalidate the snark proof
    // Most likely it is not required, but it's better to stay on the safe side and it only takes 2 constraints
    // Squares are used to prevent optimizer from removing those constraints
    signal recipientSquare;
    signal feeSquare;
    signal relayerSquare;
    signal refundSquare;
    recipientSquare <== recipient * recipient;
    feeSquare <== fee * fee;
    relayerSquare <== relayer * relayer;
    refundSquare <== refund * refund;
}

component main = Withdraw(20);
```

mercleTree.circom

```
template MerkleTreeChecker(levels) {
    signal input leaf;
    signal input root;
    signal input pathElements[levels];
    signal input pathIndices[levels];

    component selectors[levels];
    component hashers[levels];

    for (var i = 0; i < levels; i++) {
        selectors[i] = DualMux();
        selectors[i].in[0] <== i == 0 ? leaf : hashers[i - 1].hash;
        selectors[i].in[1] <== pathElements[i];
        selectors[i].s <== pathIndices[i];

        hashers[i] = HashLeftRight();
        hashers[i].left <== selectors[i].out[0];
        hashers[i].right <== selectors[i].out[1];
    }

    root === hashers[levels - 1].hash;
}
```

https://github.com/tornadocash/tornado-core/tree/master/circuits

snarkjs



- made by fvictorio

Tornado.sol

```
/**
  @dev Withdraw a deposit from the contract. `proof` is a zkSNARK proof data, and input is an array of circuit public inputs
  `input` array consists of:
    - merkle root of all deposits in the contract
    - hash of unique deposit nullifier to prevent double spends
    - the recipient of funds
    - optional fee that goes to the transaction sender (usually a relay)
*/
function withdraw(
  bytes calldata _proof,
  bytes32 _root,
  bytes32 _nullifierHash,
  address payable _recipient,
  address payable _relayer,
  uint256 _fee,
  uint256 _refund
) external payable nonReentrant {
  require(_fee <= denomination, "Fee exceeds transfer value");
  require(!nullifierHashes[_nullifierHash], "The note has been already spent");
  require(isKnownRoot(_root), "Cannot find your merkle root"); // Make sure to use a recent one
  require(
    verifier.verifyProof(
      _proof,
      [uint256(_root), uint256(_nullifierHash), uint256(_recipient), uint256(_relayer), _fee, _refund]
    ),
    "Invalid withdraw proof"
  );

  nullifierHashes[_nullifierHash] = true;
  _processWithdraw(_recipient, _relayer, _fee, _refund);
  emit Withdrawal(_recipient, _nullifierHash, _relayer, _fee);
}
```

Verifier.sol

https://github.com/tornadocash/tornado-core/tree/master/contracts

Thank you