Tornado Cash

Using SNARKs for Privacy and Scalability

Saravanan Vijayakumaran

Department of Electrical Engineering
Indian Institute of Technology Bombay

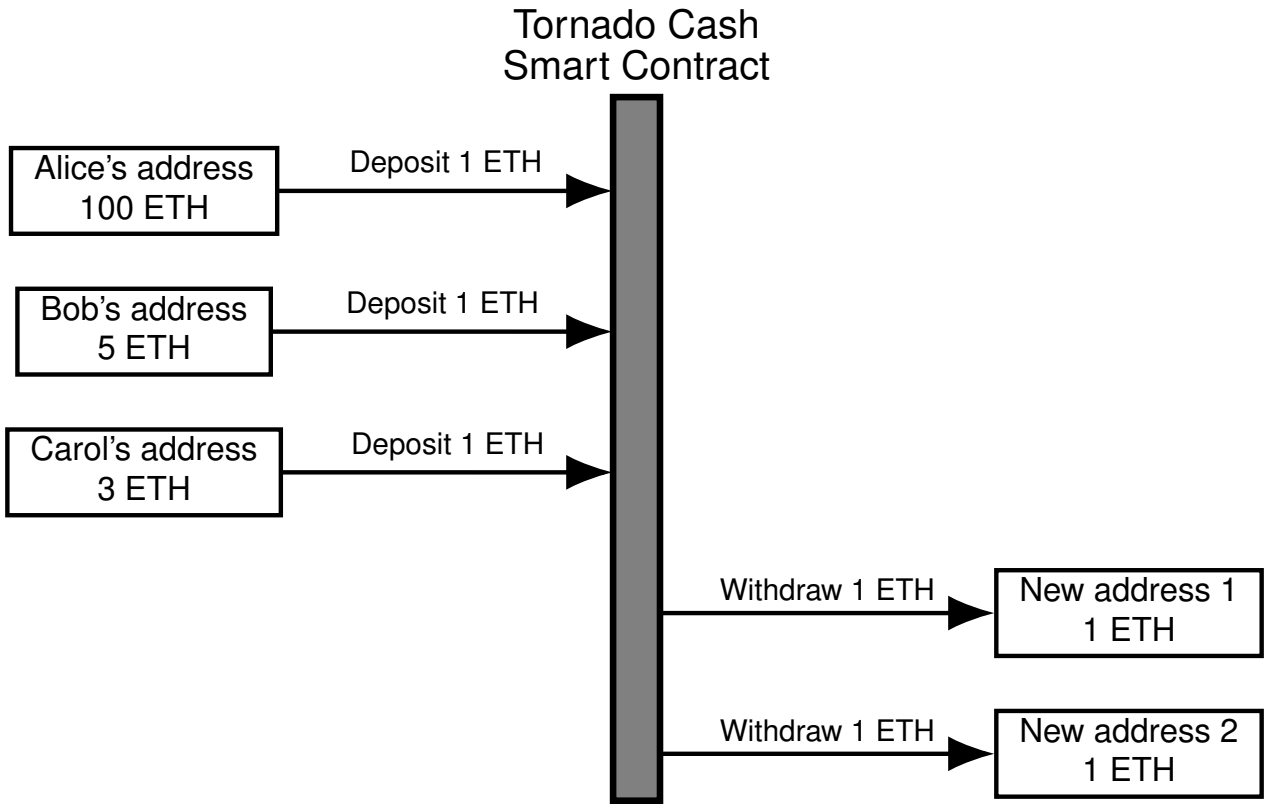
December 18, 2023

Motivation

- Consider the following scenario
  - You have 100 ETH stored in a self-custodial wallet
  - You take your family on a vacation to an exotic country
  - The hotel accepts ETH as a mode of payment
  - You pay the room rent of 1 ETH while checking in
  - The front desk clerk notices that you love your family and that your ETH address has 99 ETH
  - He has friends in the kidnapping industry
- How can you prevent leaking the total amount of ETH you hold?
  - **Option A:** You could store your ETH on an exchange and pay using their interface.
    - You risk losing funds due to exchange hacks
    - Hackers can steal customer data and sell it to their kidnapper friends
  - **Option B:** You could send 1 ETH to a fresh address from your 100 ETH address and use that to pay the room rent
    - Now suppose you decide to extend your stay
    - You make another 1 ETH transfer from your main ETH address
    - The clerk can now infer that you control a large amount of ETH
- Tornado Cash is a better **Option B**
  - It is a smart contract on Ethereum which implements a **mixer**

Tornado Cash Overview

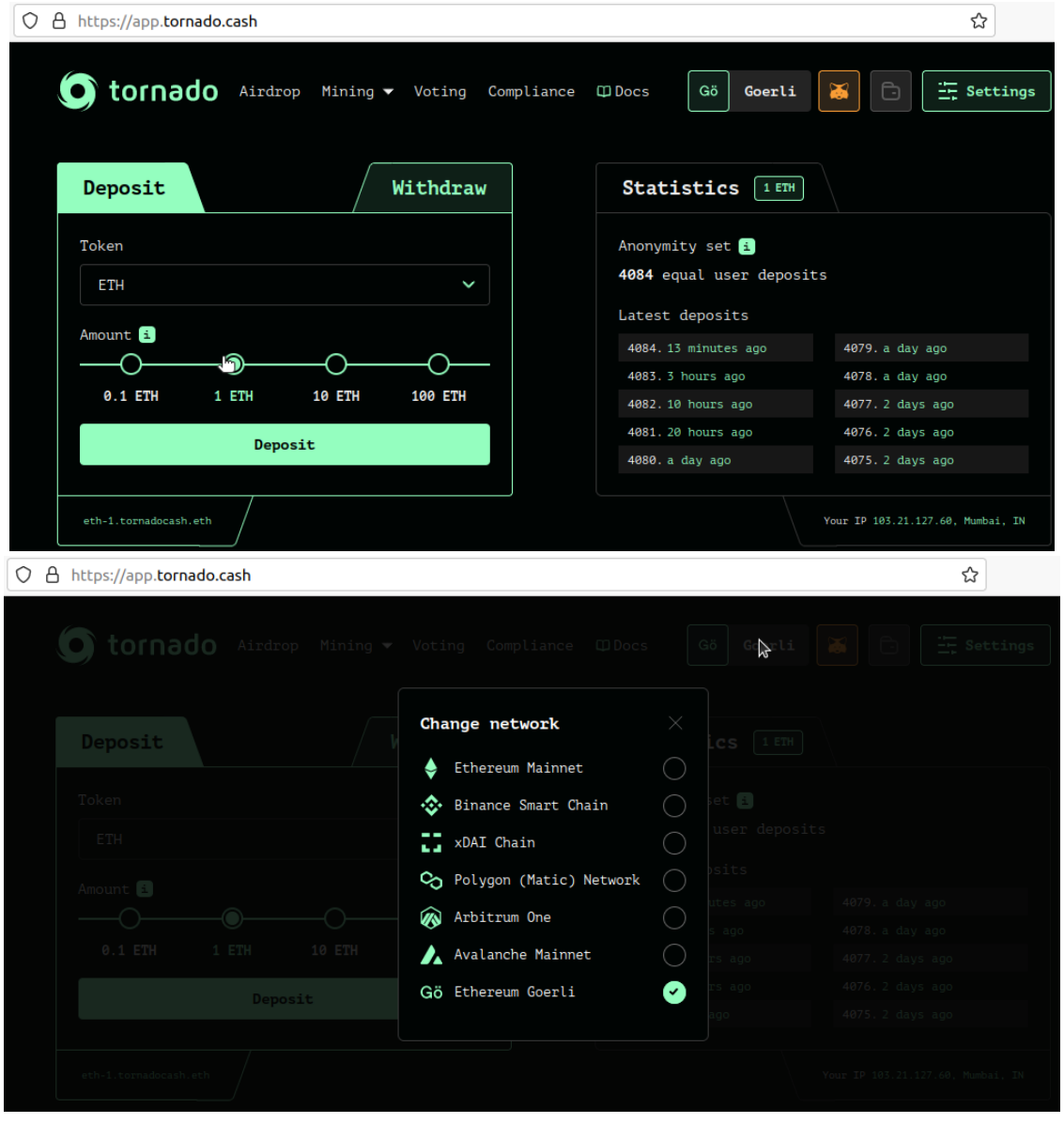
Pre-Nova Version



- Desired functionality
  - **Soundness**
    - Only past depositors should be able to withdraw
    - No double withdrawal (only one withdrawal per deposit)
  - **Privacy**: A withdrawal should not be linkable to a particular past deposit

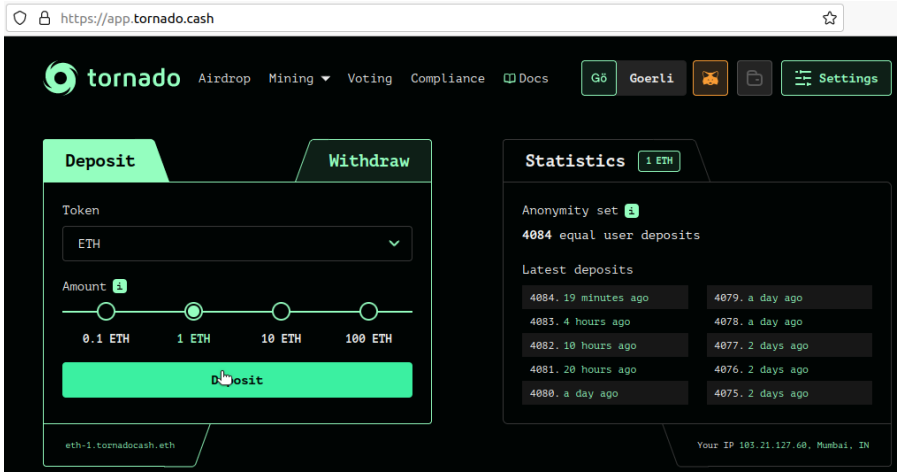
Deposit Workflow (1/2)

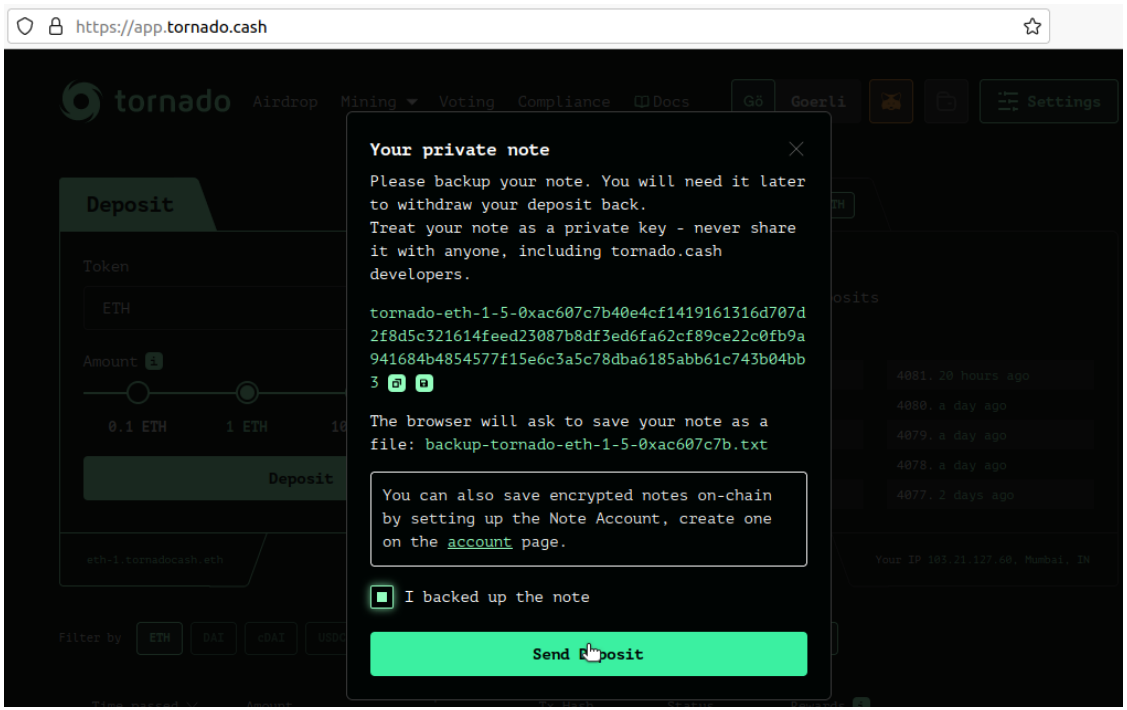
Choose amount and chain



Deposit Workflow (2/2)

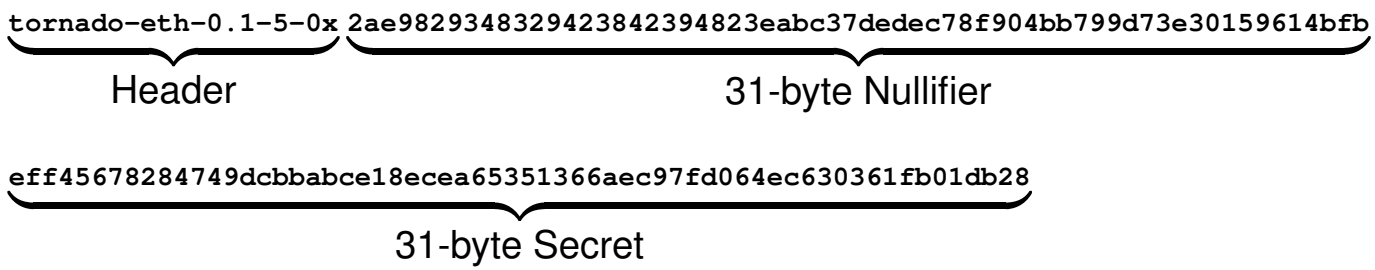
Connect wallet, save note, and deposit





Deposit Steps (1/2)

- Anatomy of a Tornado Cash private note



- The 62 bytes in the nullifier and secret are randomly generated on the user's computer
- A **commitment** (Pedersen hash of the 62 bytes) is calculated and submitted to the contract

$$\text{Pedersen hash of bitstring } b_1 b_2 \ldots b_n = g_1^{b_1} g_2^{b_2} \ldots g_n^{b_n}.$$

- Contract checks that `_commitment` has not been seen before

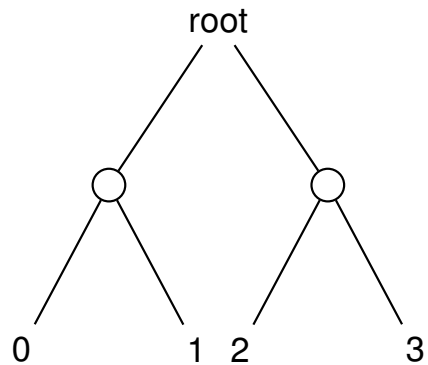
```
mapping(bytes32 => bool) public commitments;
// <snip>
require(!commitments[_commitment], "The commitment has been submitted");
```

Deposit Steps (2/2)

- Contract inserts `_commitment` into a Merkle tree

```
uint32 insertedIndex = _insert(_commitment);
```

  - Tree has 20 levels
  - `insertedIndex` is the index of new leaf



  - No leaf deletions allowed $\implies$ Maximum of 2^{20} deposits

- Stores the fact that `_commitment` has been seen

```
commitments[_commitment] = true;
```

- Checks that ETH being sent equals contract denomination

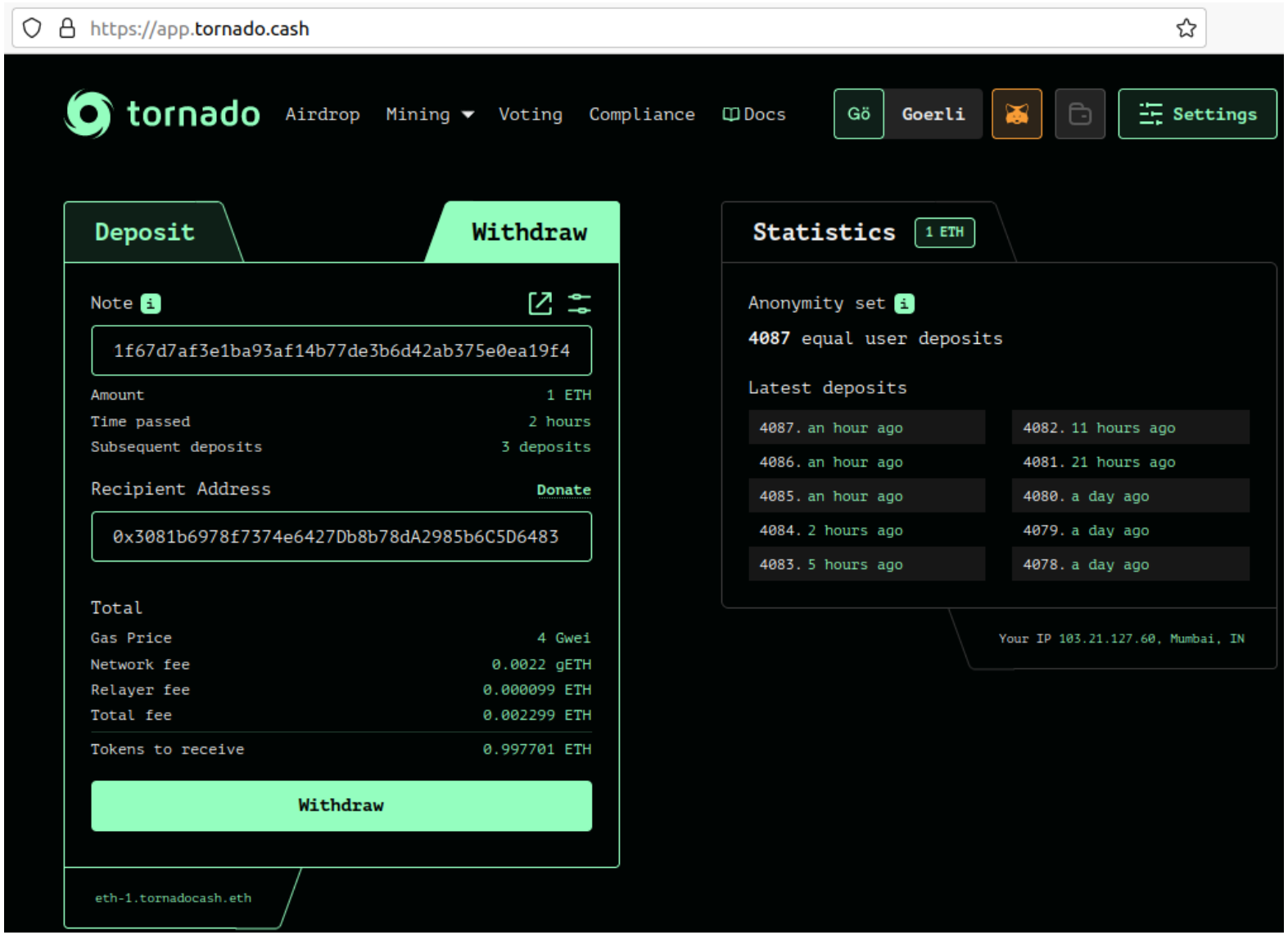
```
require(msg.value == denomination, "Please send 1 ETH with transaction");
```

- Emits an event

```
emit Deposit(_commitment, insertedIndex, block.timestamp);
```

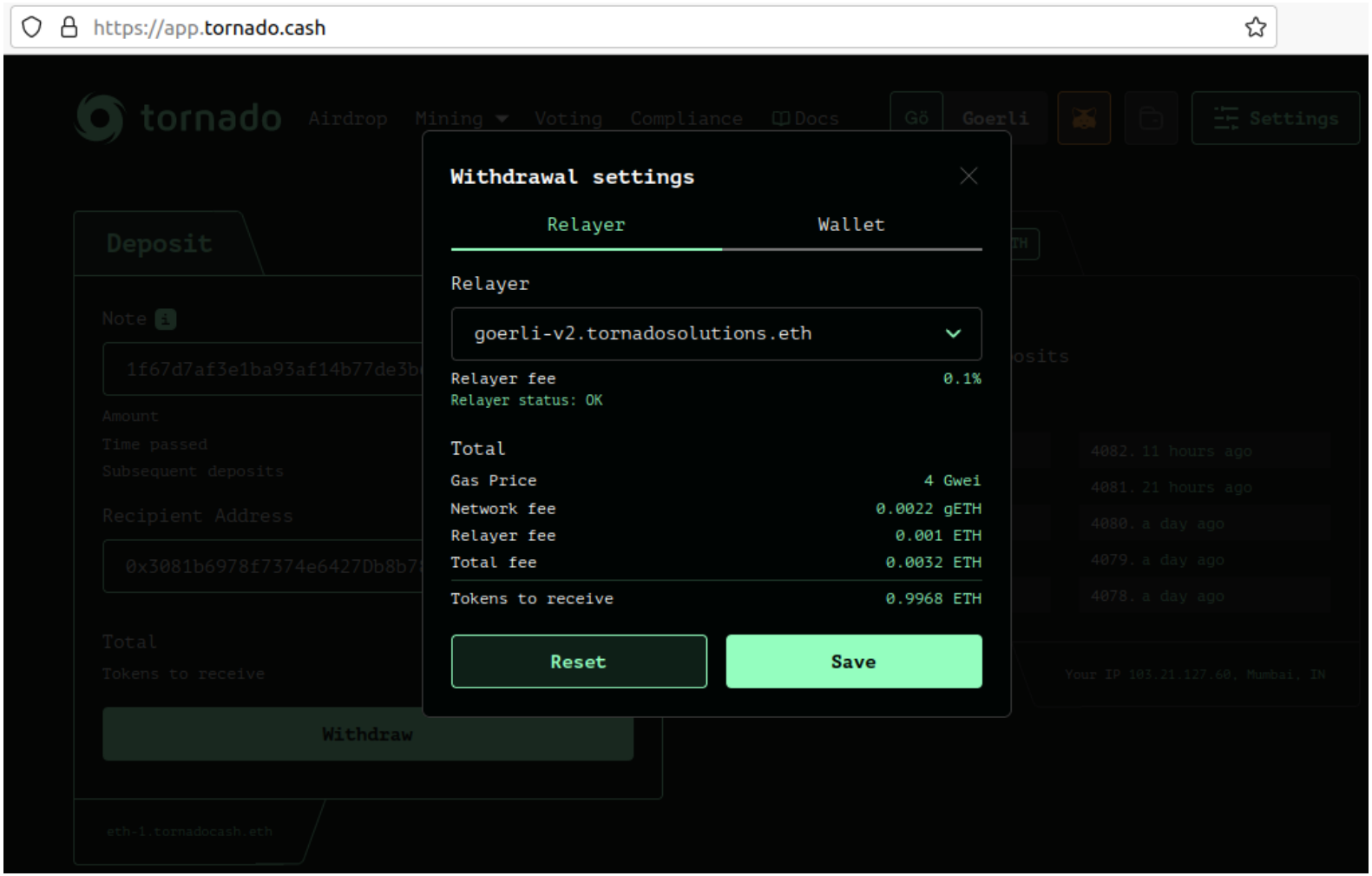
Withdrawal Workflow (1/3)

Enter note string and recipient address



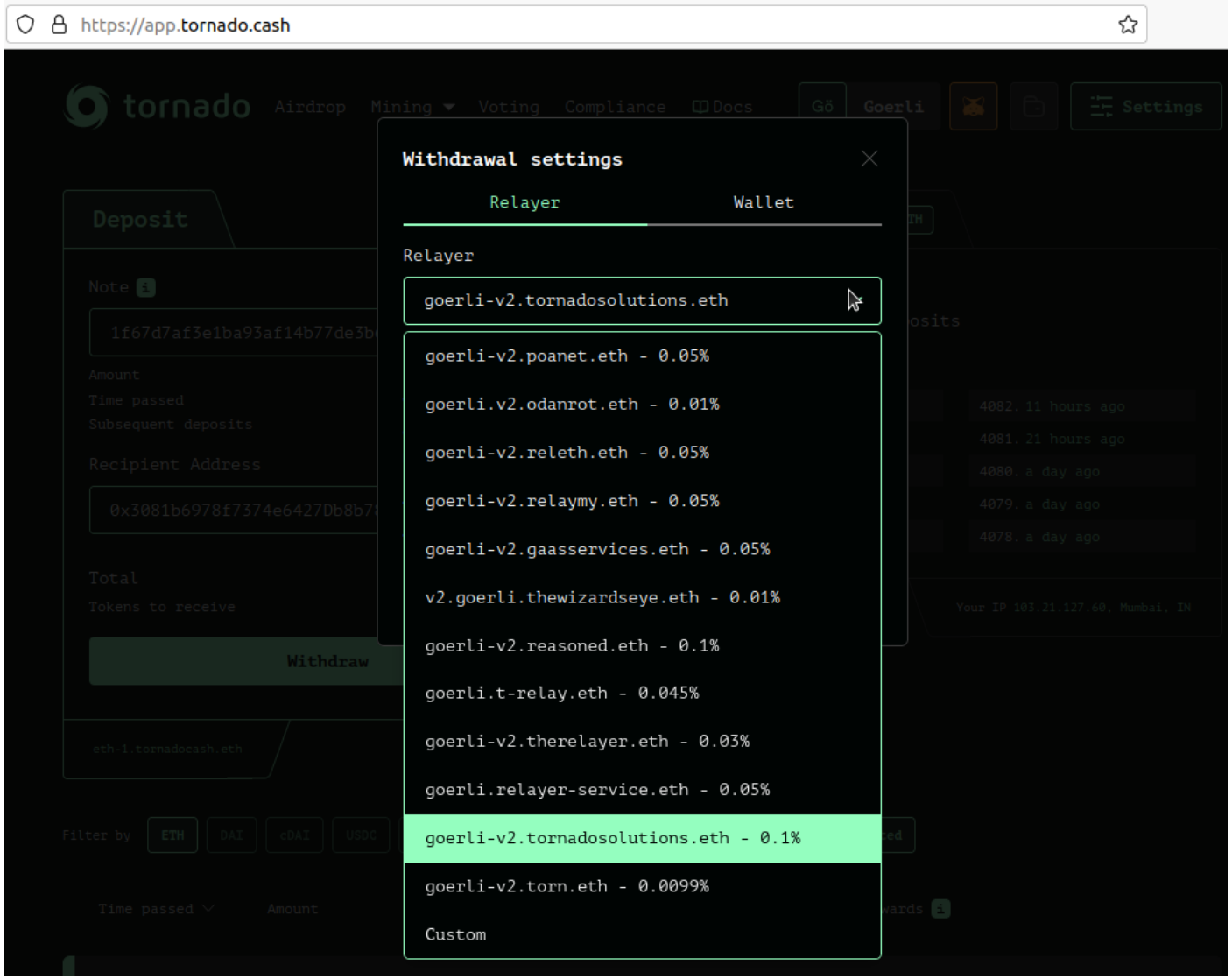
Withdrawal Workflow (2/3)

Choose relayer



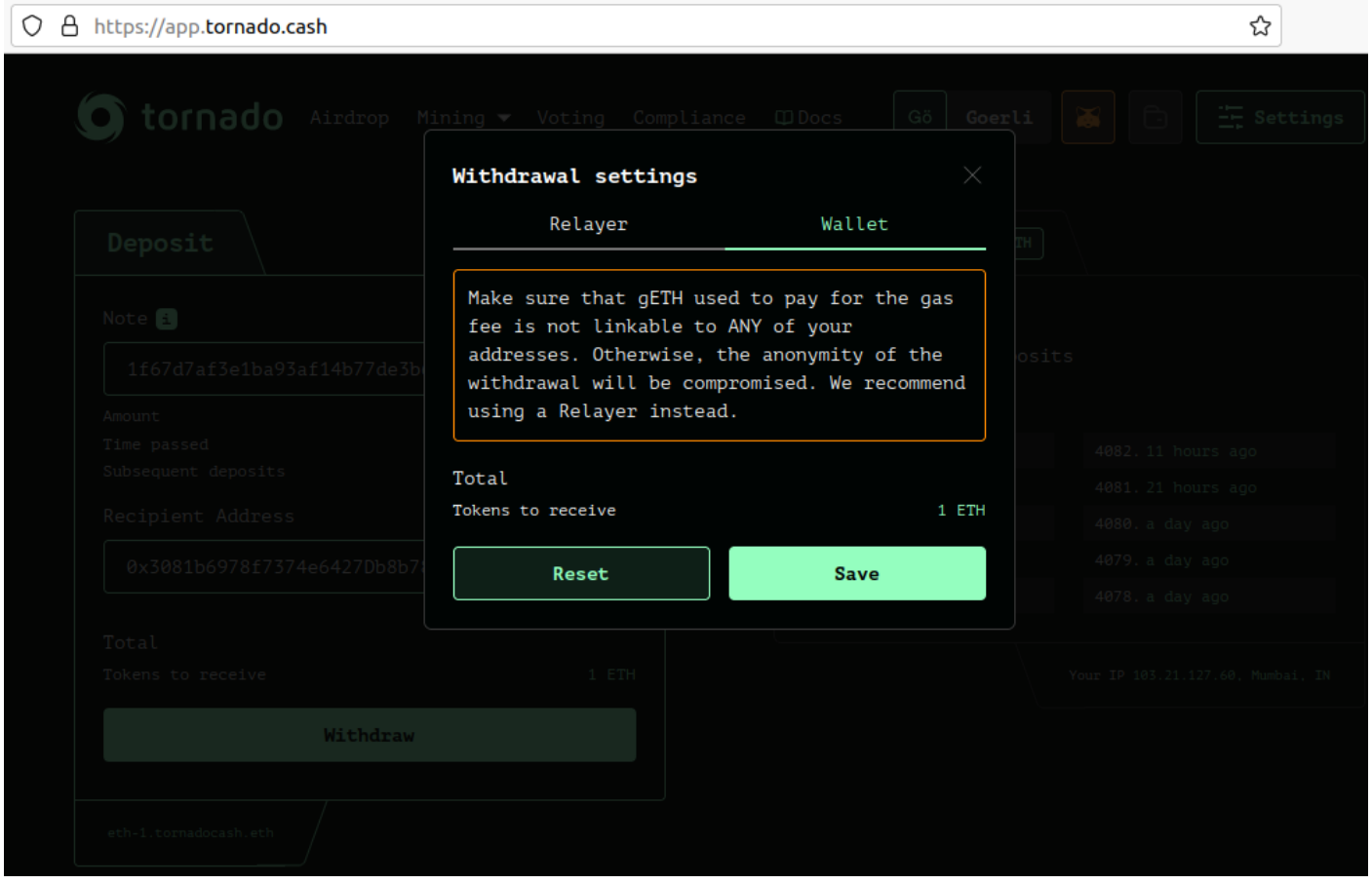
Withdrawal Workflow (2/3)

Choose relayer



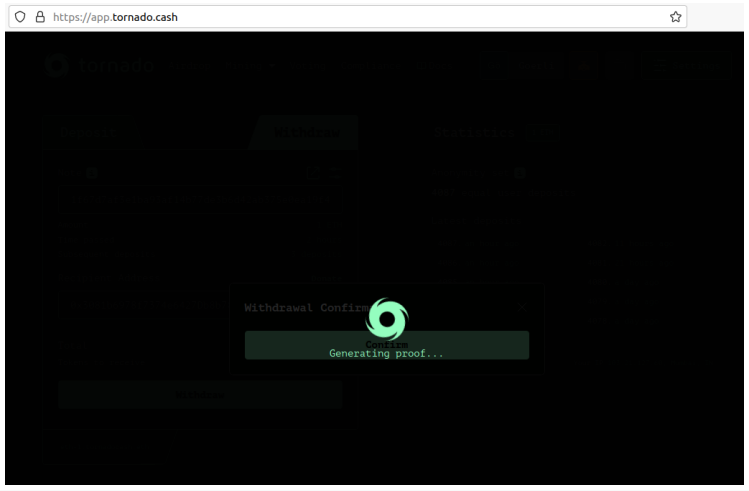
Withdrawal Workflow (2/3)

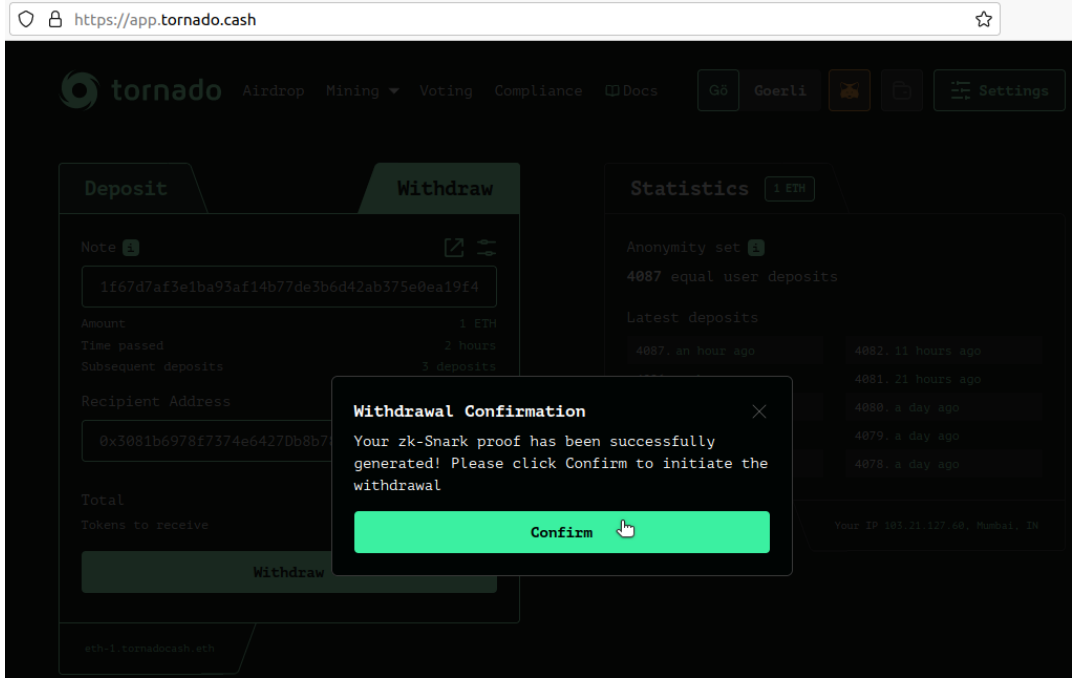
Choose wallet if you have an unlinkable address with ETH



Withdrawal Workflow (3/3)

Generate proof and confirm withdrawal





Withdrawal Steps (1/2)

- Recall our requirements
  - **Soundness**
    - Only past depositors should be able to withdraw
    - No double withdrawal (only one withdrawal per deposit)
  - **Privacy**: A withdrawal should not be linkable to a particular past deposit
- The `withdraw` method is executed

```
function withdraw(
  bytes calldata _proof,
  bytes32 _root,
  bytes32 _nullifierHash,
  address payable _recipient,
  address payable _relayer,
  uint256 _fee
  // <snip>
)
```

- `_proof` is a SNARK proof for the following statement:
  I know the secret and nullifier for a commitment which is included in the Merkle tree with root `_root`.
  Furthermore, `_nullifierHash` *is the Pedersen hash of the commitment's nullifier.*

Withdrawal Steps (2/2)

- Contract checks that `_nullifierHash` has not been seen before.

```
mapping(bytes32 => bool) public nullifierHashes;
// <snip>
require(!nullifierHashes[_nullifierHash], "Note already spent");
```

  This prevents double withdrawal
- Checks that `_root` is any of the last 100 Merkle roots

```
require(isKnownRoot(_root), "Cannot find your merkle root");
```

- It then verifies the SNARK proof on-chain

```
require(
  verifier.verifyProof(_proof,
    [uint256(_root), uint256(_nullifierHash), ...]
  ),
  "Invalid withdraw proof"
);
```

- Stores the fact that `_nullifierHash` has been seen

```
nullifierHashes[_nullifierHash] = true;
```

- Sends relevant amounts to `_recipient` and `_relayer`

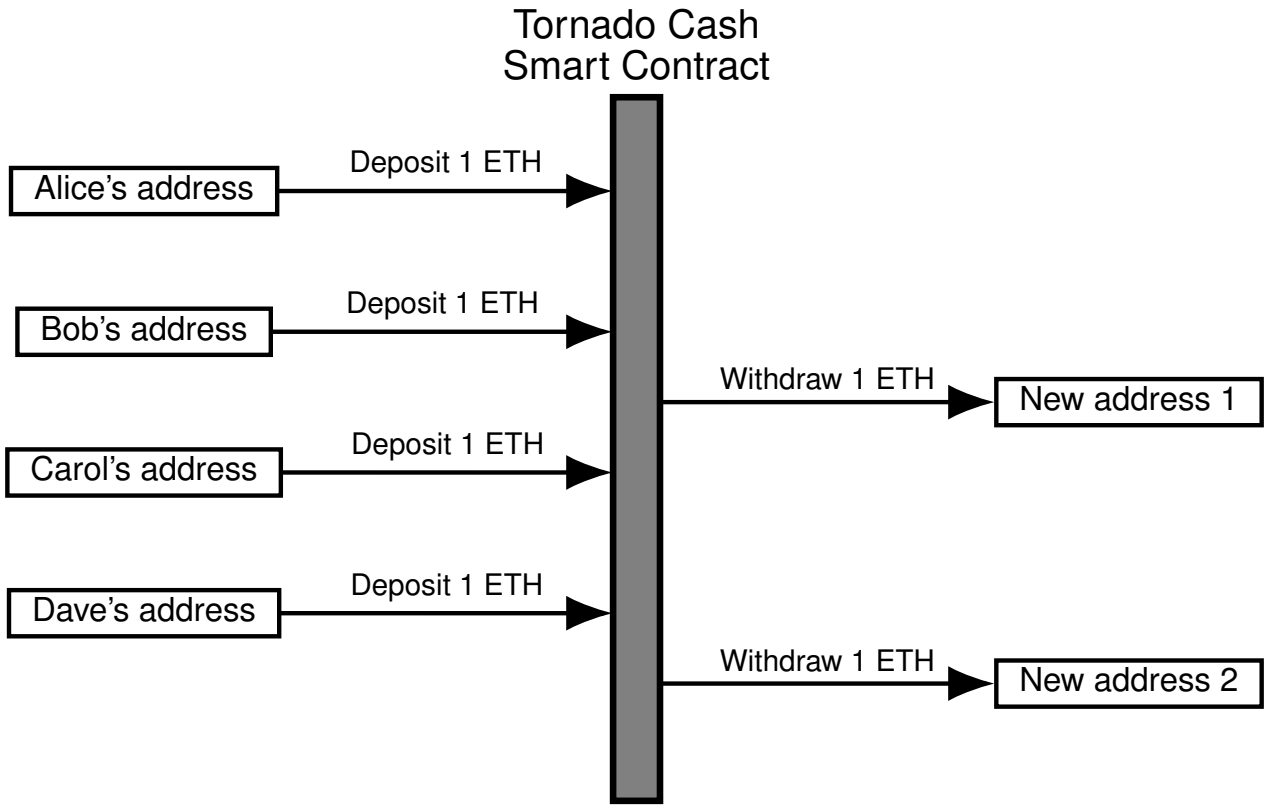
```
_recipient.call.value(denomination - _fee)("");
_relayer.call.value(_fee)("");
```

- The SNARK proof also “signs” the `_recipient`, `_relayer`, `_fee` fields to prevent tampering
- The verifier contract is generated using the `circom` compiler.

withdraw.circom

```
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 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;
    recipientSquare <== recipient * recipient;
    feeSquare <== fee * fee;
    relayerSquare <== relayer * relayer;
}

component main = Withdraw(20);
```

Contract Liquidity Increases Privacy

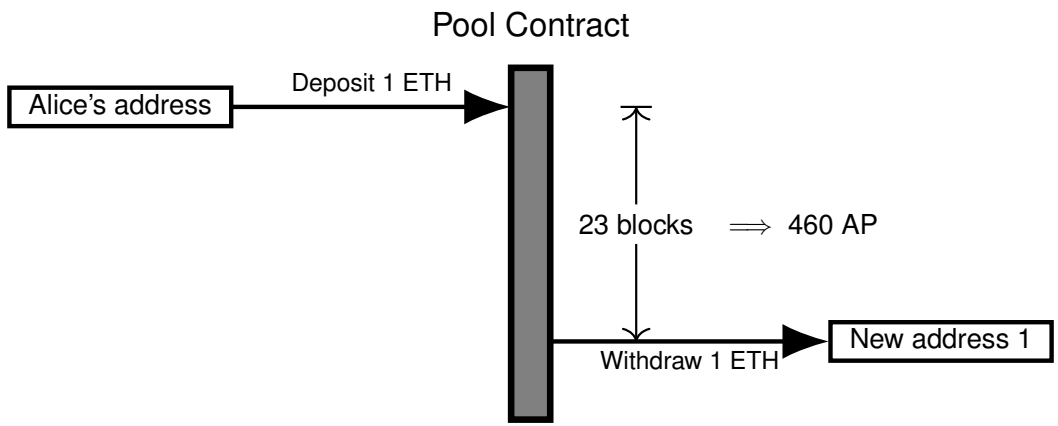


- The first withdrawal has an anonymity set size of 2 while the second one has an anonymity set size of 4
- To increase liquidity, Tornado Cash developers introduced **anonymity mining**

Anonymity Mining

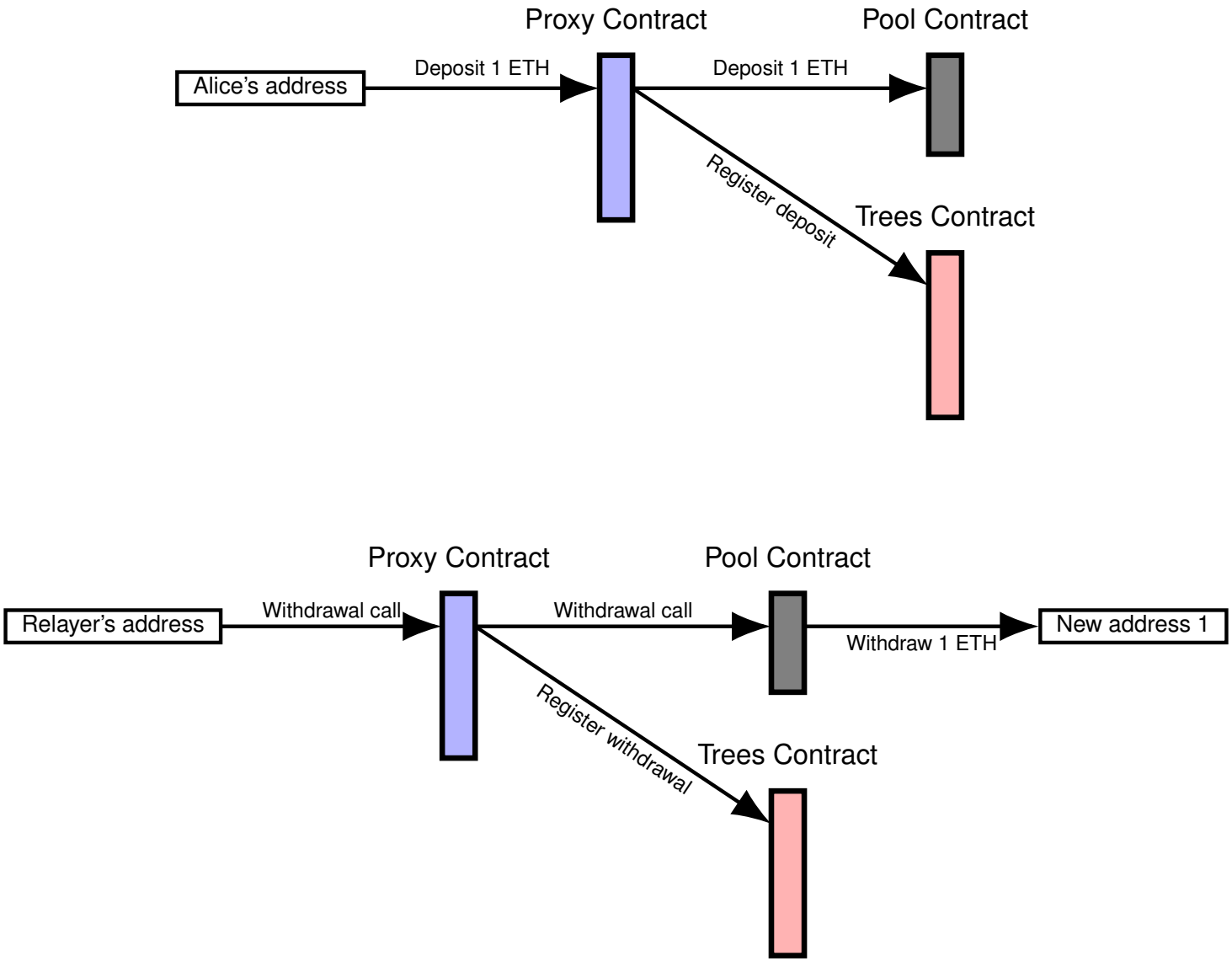
- Depositors get rewarded in proportion to the amount and duration of liquidity they provide
- They earn anonymity points (AP) which can be swapped for TORN tokens

Pool	AP per block
100 ETH	400
10 ETH	50
1 ETH	20
0.1 ETH	4

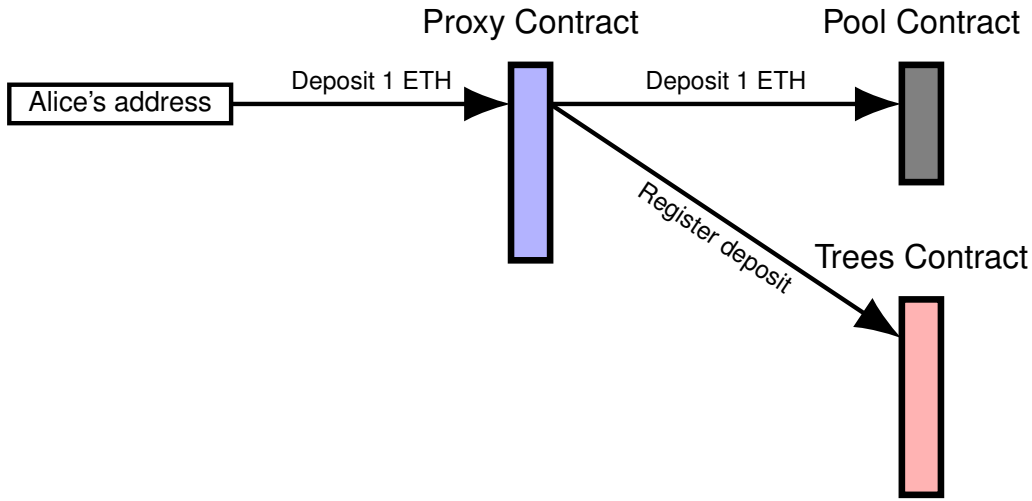


What about privacy?

Anonymity Mining Contract Call Flow



Register Deposit Steps



- Trees contract stores hashes of deposit data in a mapping

```
mapping(uint256 => bytes32) public deposits;
```

- Current length of deposits is read

```
uint256 _depositsLength = depositsLength;
```

- Hash of deposit data is stored in the mapping

```
deposits[_depositsLength] =
  keccak256(abi.encode(_instance, _commitment, blockNumber()));
```

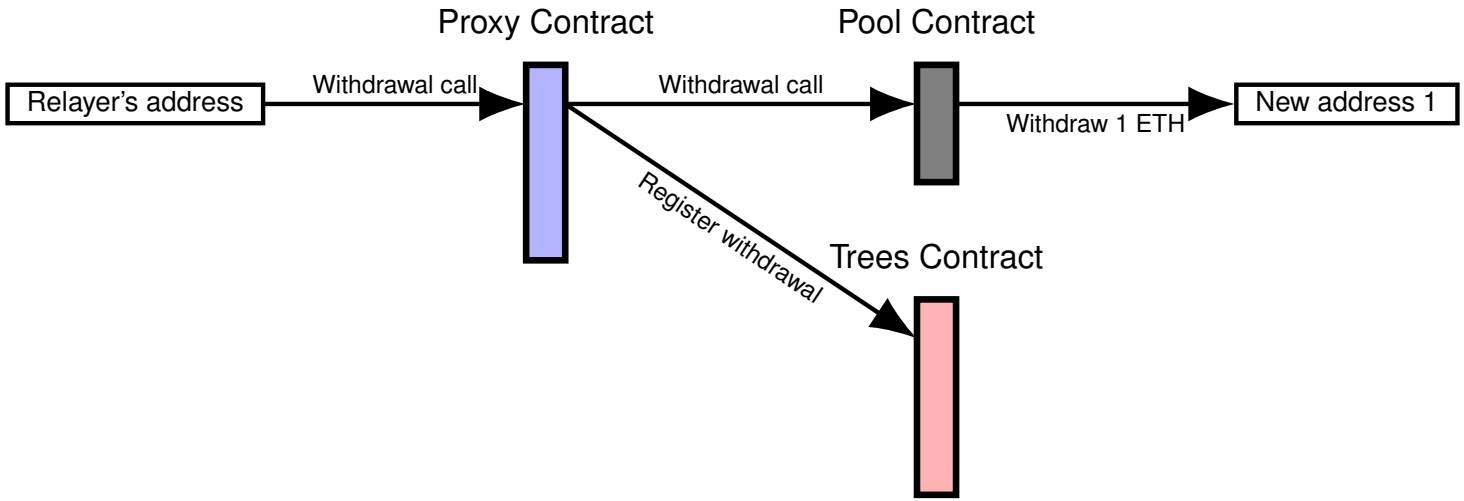
- Emit event with hash pre-image

```
emit
DepositData(_instance, _commitment, blockNumber(), _depositsLength);
```

- Increment number of deposits

```
depositsLength = _depositsLength + 1;
```

Register Withdrawal Steps



- Trees contract stores hashes of withdrawal data in a mapping

```
mapping(uint256 => bytes32) public withdrawals;
```

- Current length of withdrawals is read

```
uint256 _withdrawalsLength = withdrawalsLength;
```

- Hash of withdrawal data is stored in the mapping

```
withdrawals[_withdrawalsLength] =
  keccak256(abi.encode(_instance, _nullifierHash, blockNumber()));
```

- Emit event with hash pre-image

```
emit WithdrawalData(_instance, _nullifierHash,
                    blockNumber(), _withdrawalsLength);
```

- Increment number of withdrawals

```
withdrawalsLength = _withdrawalsLength + 1;
```

Why Trees contract?

- Contract has two Merkle trees, one each for deposit and withdrawal hashes

```
OwnableMerkleTree public immutable depositTree;
OwnableMerkleTree public immutable withdrawalTree;
```

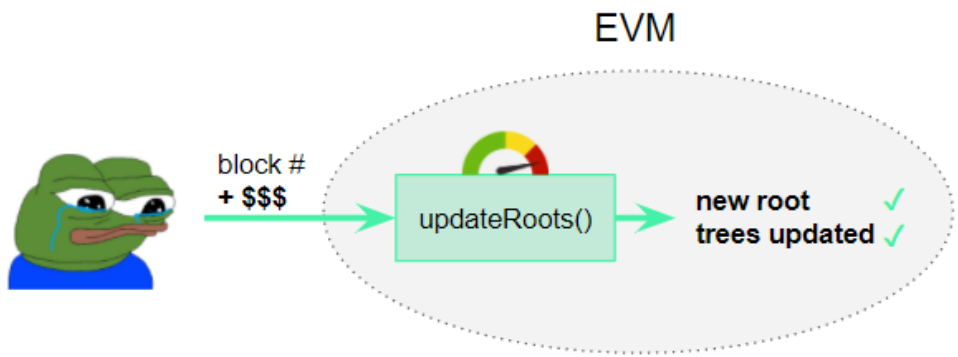
- To collect AP privately, a miner gives a SNARK proof proving the following statements
  - *I performed a deposit which is included in the Merkle tree with root* `depositRoot` *and a withdrawal which is included in the Merkle tree with root* `withdrawalRoot`.
  - *I am withdrawing an AP amount which equals* $\textbf{\textit{rate}} \times (\textbf{\textit{withdrawalBlockHeight}} - \textbf{\textit{depositBlockHeight}}) - \textbf{\textit{fee}}$.
  - *Furthermore,* `rewardNullifier` *is the* ***Poseidon*** *hash of the commitment's nullifier.*
- For cost reasons, each deposit and withdrawal does not update the tree roots

```
depositTree.bulkInsert(leaves);
withdrawalTree.bulkInsert(leaves);
```

- From time to time, the roots had to be updated by an altruistic party

Tree updates became expensive in 2021

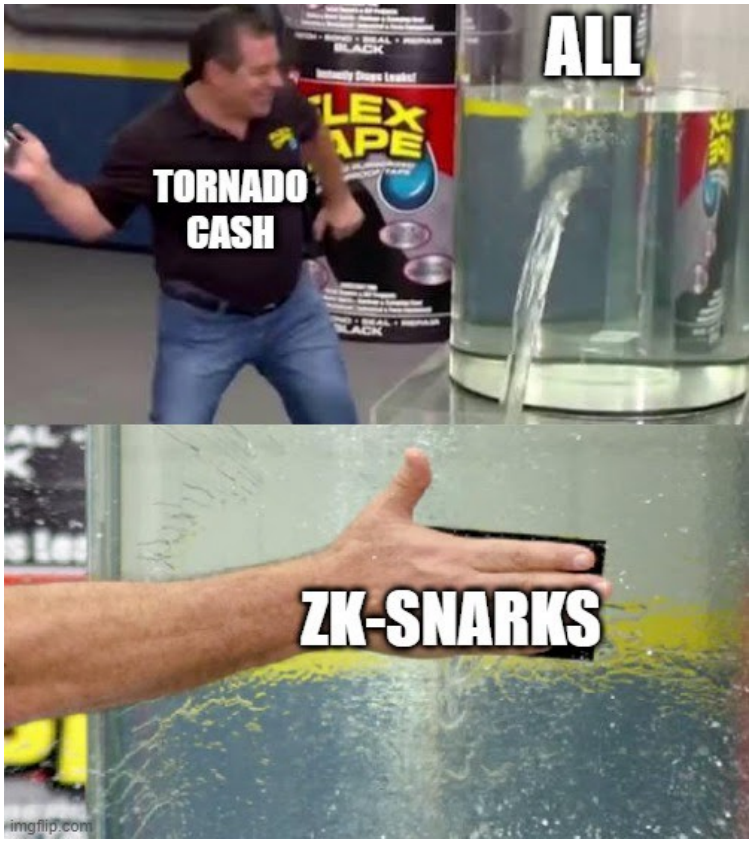
Under the mad gas fee hike in the beginning of the year, it soon became too expensive for the average crypto-Joe to altruistically update the data structures for everyone's benefit. By the end of March, it cost roughly $16 in gas to register the deposit-withdrawal of a single note. With the backlog growing slowly to over 40 days, and more than 25,000 deposits and withdrawals pending, it would cost ~300 ETH to update the Merkle trees. Anonymity mining was stuck for a good reason.



Expensive operation = no execution = no AP claims

Screenshot from `https://tornado-cash.medium.com/gas-price-claimed-anonymity-mining-a-victim-but-now-everyone-can-claim-ap-5441aaa32a1a`

Workaround (circa March 2021)



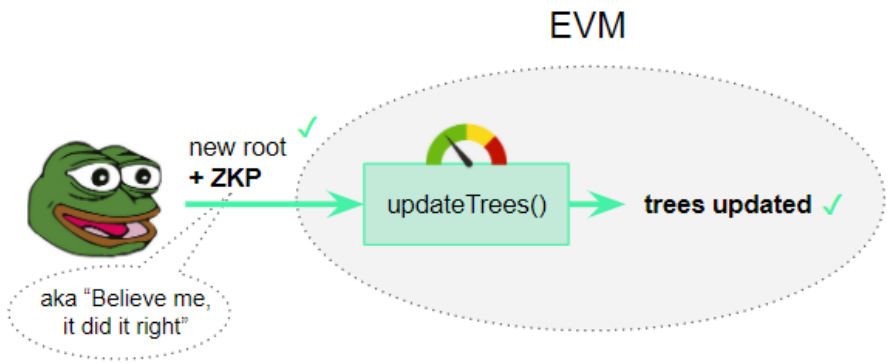


Image credits: `https://tornado-cash.medium.com/gas-price-claimed-anonymity-mining-a-victim-but-now-everyone-can-claim-ap-5441aaa32a1a`

SNARKs for Cheaper Tree Updates

- Batches of 256 deposits or withdrawals are used to update the trees

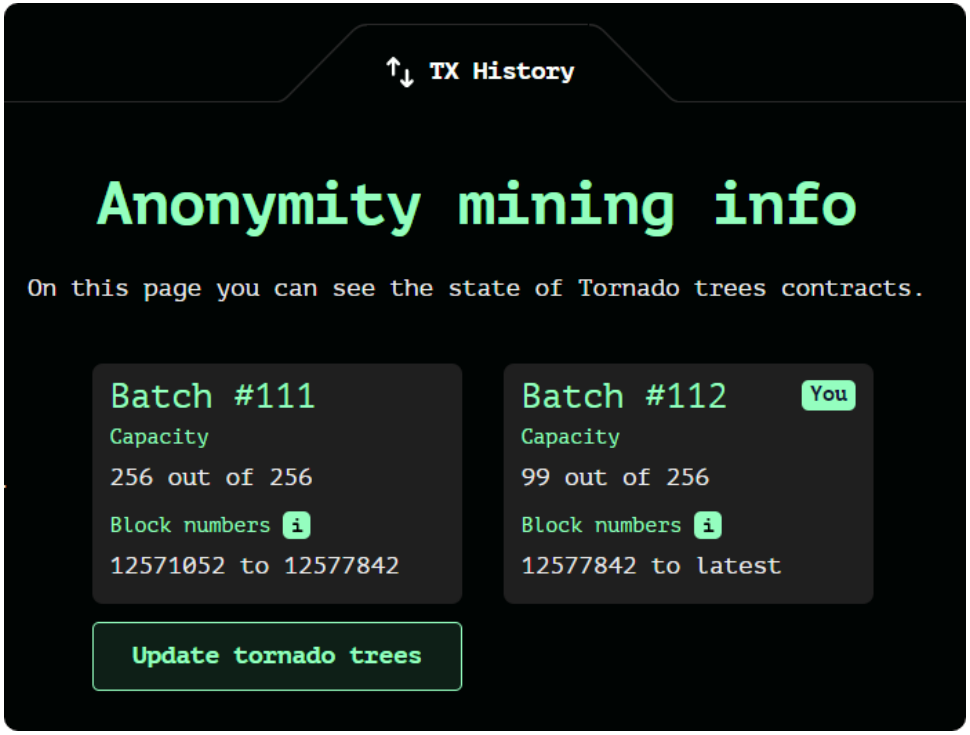


Image credit: Tornado Cash Medium post

- Root updater gives a SNARK proof proving the following statement:
  When the 256 leaves I am showing you are added to the Merkle tree, its root changes from `_currentRoot` *to* `_newRoot`.

Tree Updater is a Validity Rollup

- Off-chain SNARK proof generation reduces on-chain computation
- All data needed by prover is present in event logs

```
emit DepositData(_instance, _commitment,
                 blockNumber(), _depositsLength);
emit WithdrawalData(_instance, _nullifierHash,
                    blockNumber(), _withdrawalsLength);
```

- Prover role is decentralized but without proper incentives

OFAC Sanctions

- On Aug 8, 2022, the US Office of Foreign Assets Control placed Tornado Cash addresses on a sanction list
- US residents/businesses cannot interact with entities on the list
- Allegations include facilitating money laundering by ransomware operators and smart contract attackers
- Github removed source repos and three contributors had Github accounts suspended
- Due to the efforts of Prof. Matthew Green and EFF, OFAC allowed use of code for educational purposes
- Github repositories and accounts restored in 2023
- Developer Alexey Pertsev arrested in Netherlands in Aug 2022; released on bail in April 2023
- Developer Roman Storm arrested in US on Aug 23, 2023
- Court hearings still pending

References

- Tornado Cash App `https://tornado.ws/`
- Tornado Cash Docs `https://docs.tornado.ws/`
- Circom `https://docs.circom.io/`
- Medium post introducing anonymity mining
- Medium post on how SNARKs make anonymity mining cheaper
- `https://github.com/tornadocash/tornado-core`
- `https://github.com/tornadocash/tornado-anonymity-mining`
- `https://github.com/tornadocash/tornado-trees`
- EFF article on OFAC sanctions
- EFF update in April 2023
- DAO to fund court case against TC developers `https://wewantjusticedao.org/`

Thanks for your attention