Tornado Cash

Using SNARKs for Privacy and Scalability

Saravanan Vijayakumaran

Department of Electrical Engineering
Indian Institute of Technology Bombay

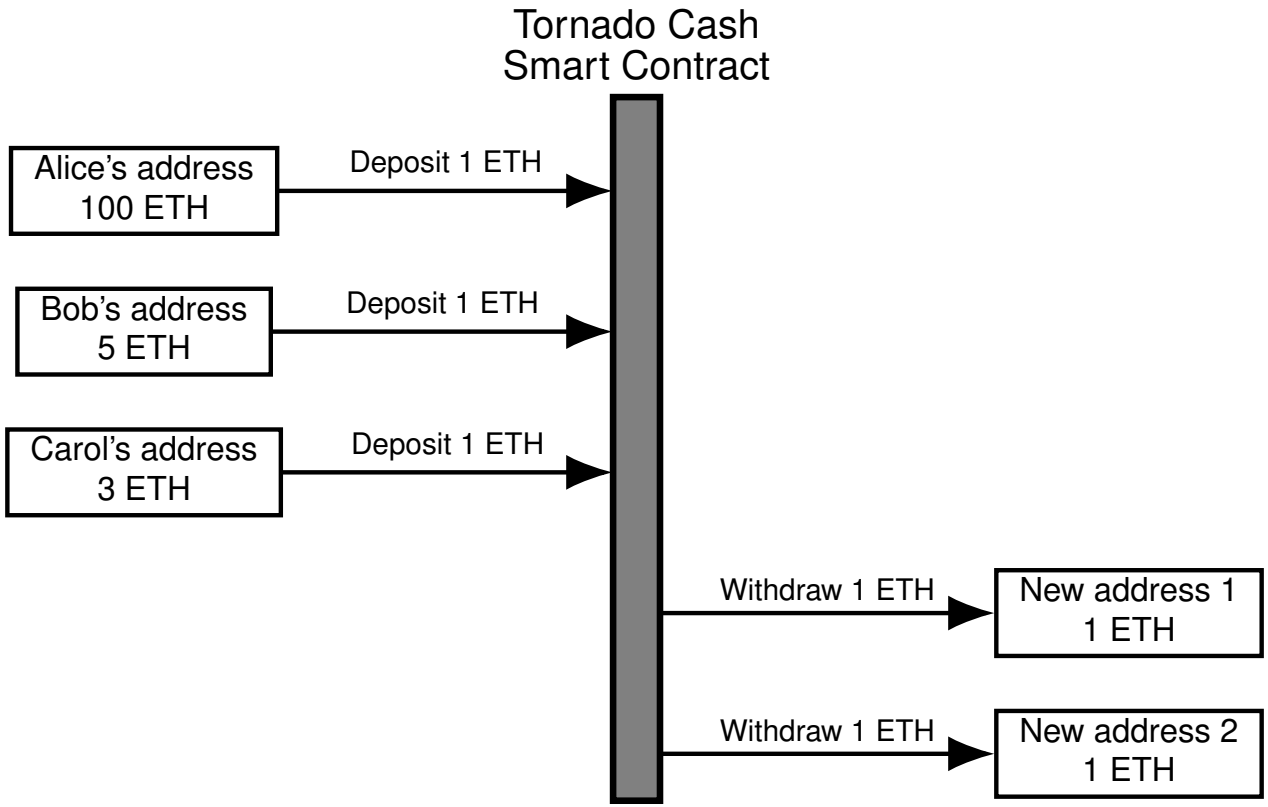
April 1, 2024

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

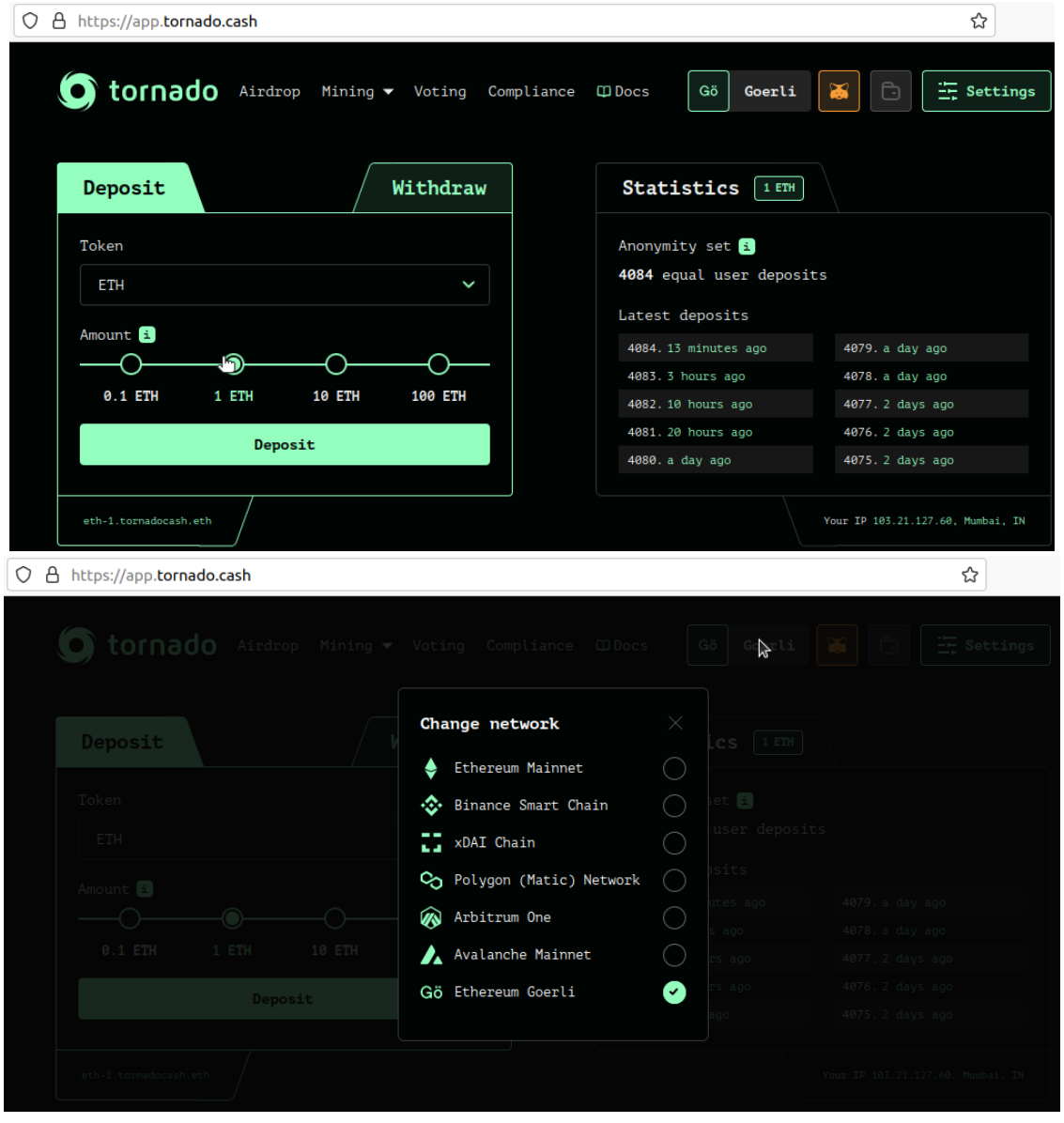
Pre-Nova Version



- 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

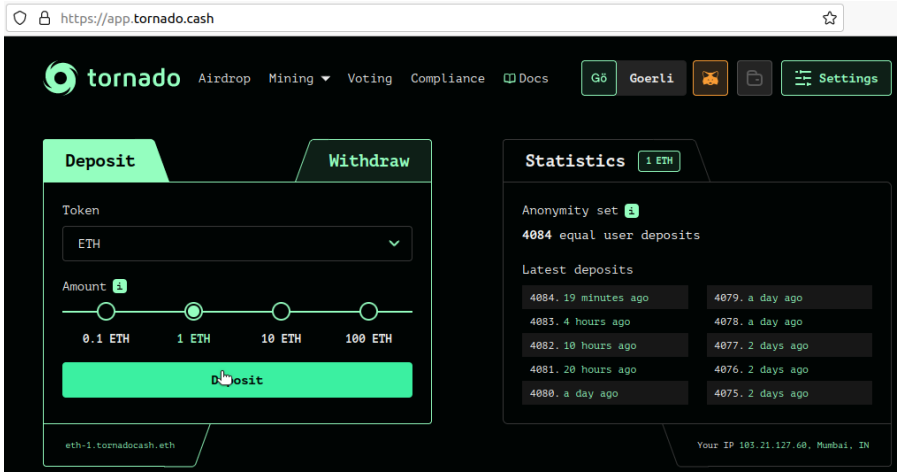
Deposit Workflow (1/2)

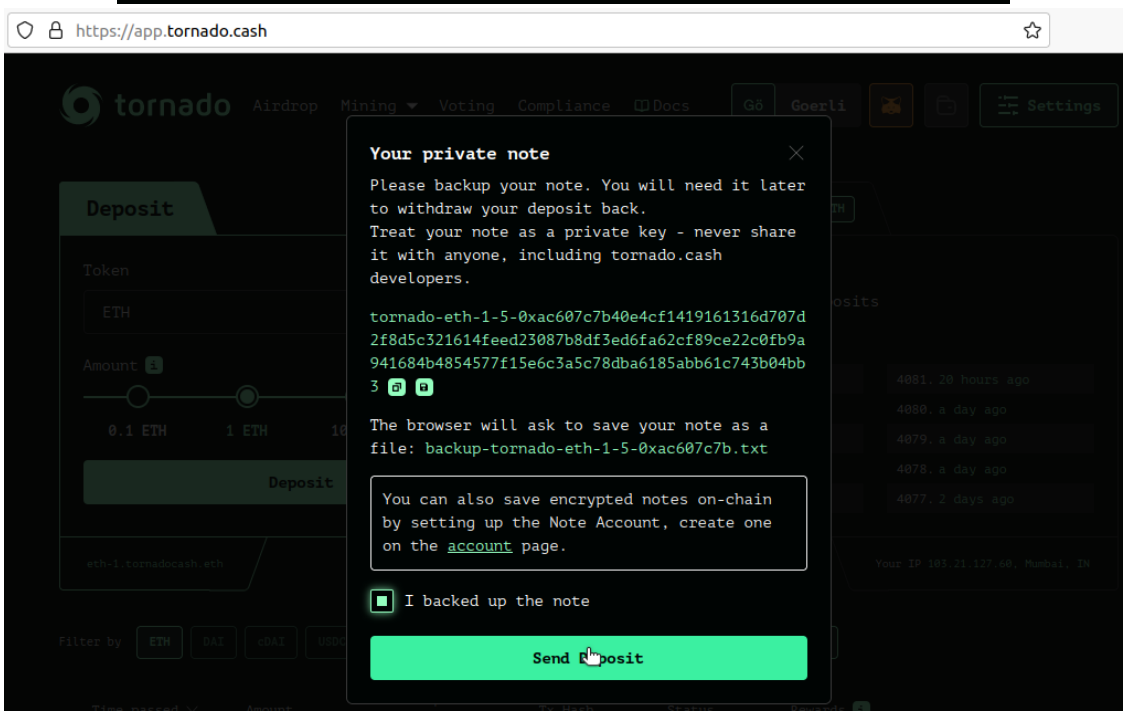
Choose amount and chain



Deposit Workflow (2/2)

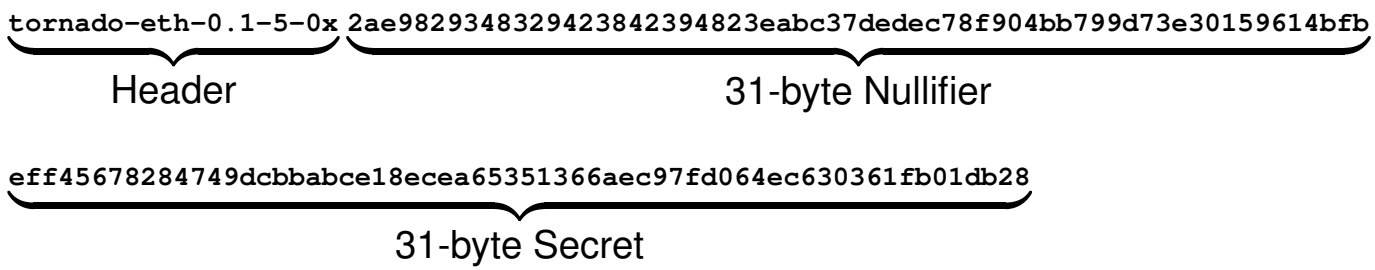
Connect wallet, save note, and deposit





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

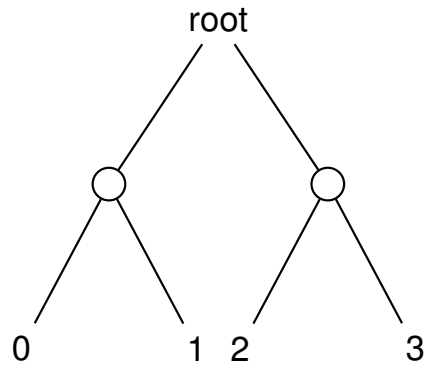
```
mapping(bytes32 => bool) public commitments;
// <snip>
require(!commitments[_commitment], "The commitment has been submitted");
```

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

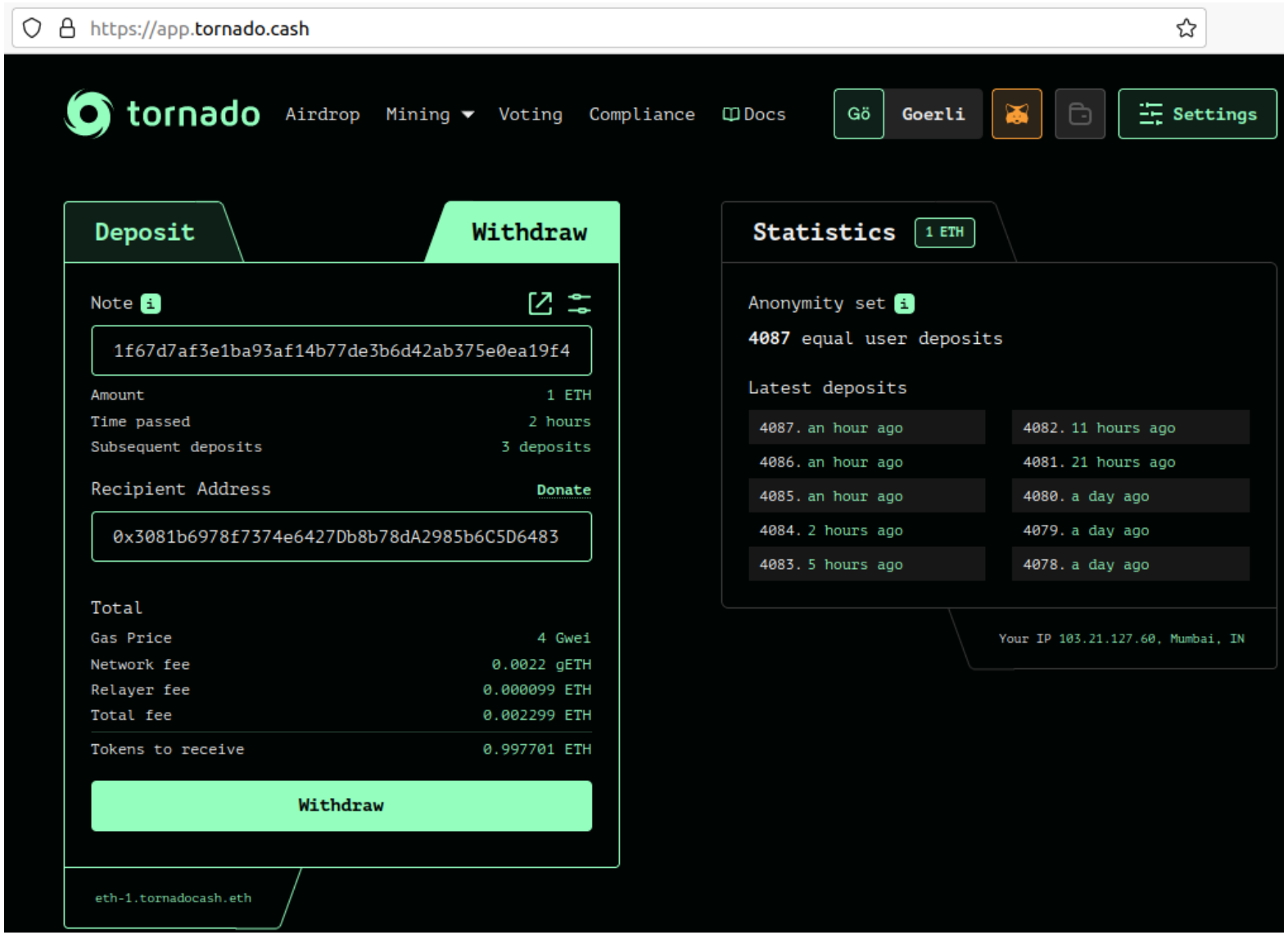
```
require(msg.value == denomination, "Please send 1 ETH with transaction");
```

- Emits an event

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

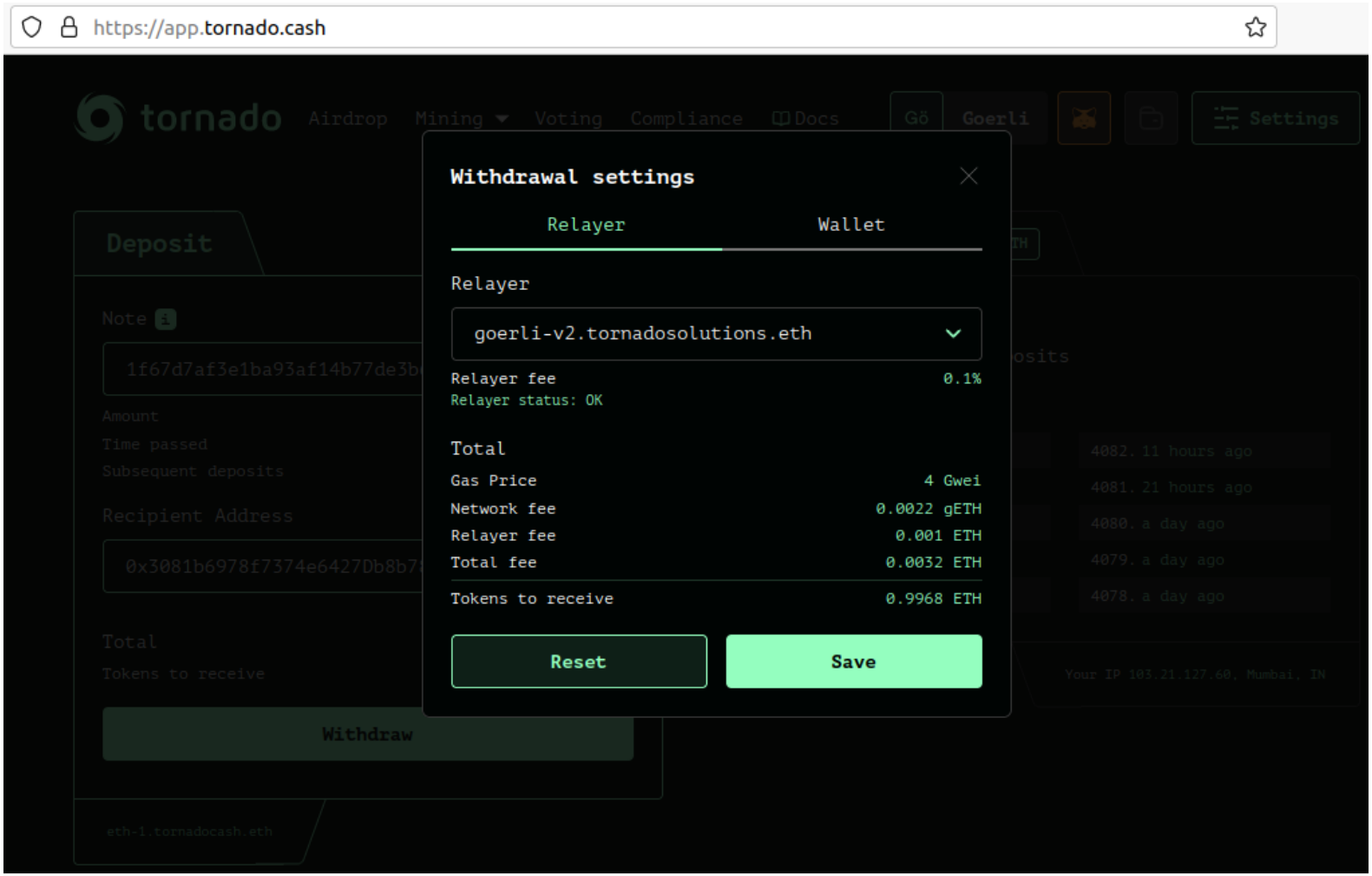
Withdrawal Workflow (1/3)

Enter note string and recipient address



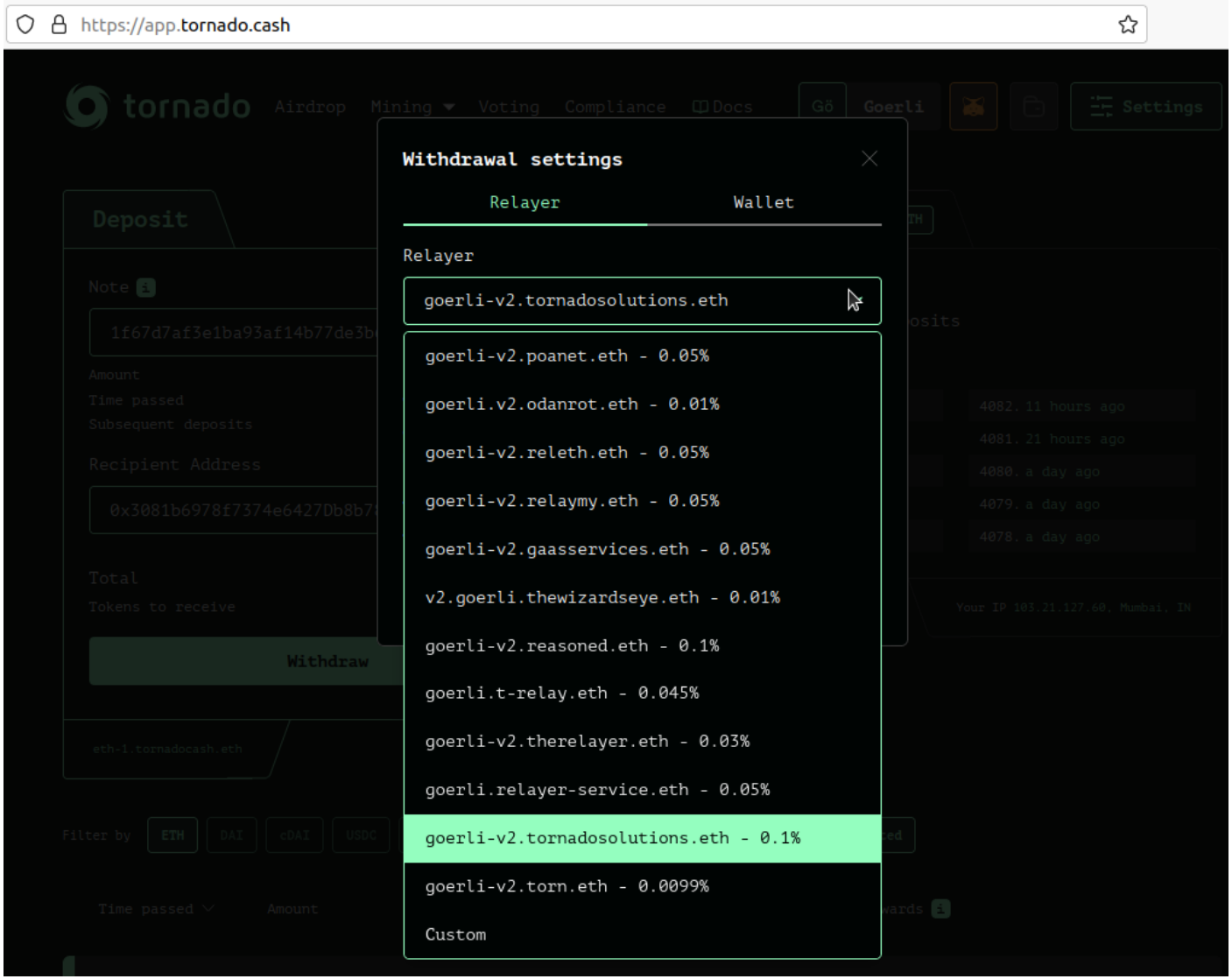
Withdrawal Workflow (2/3)

Choose relayer



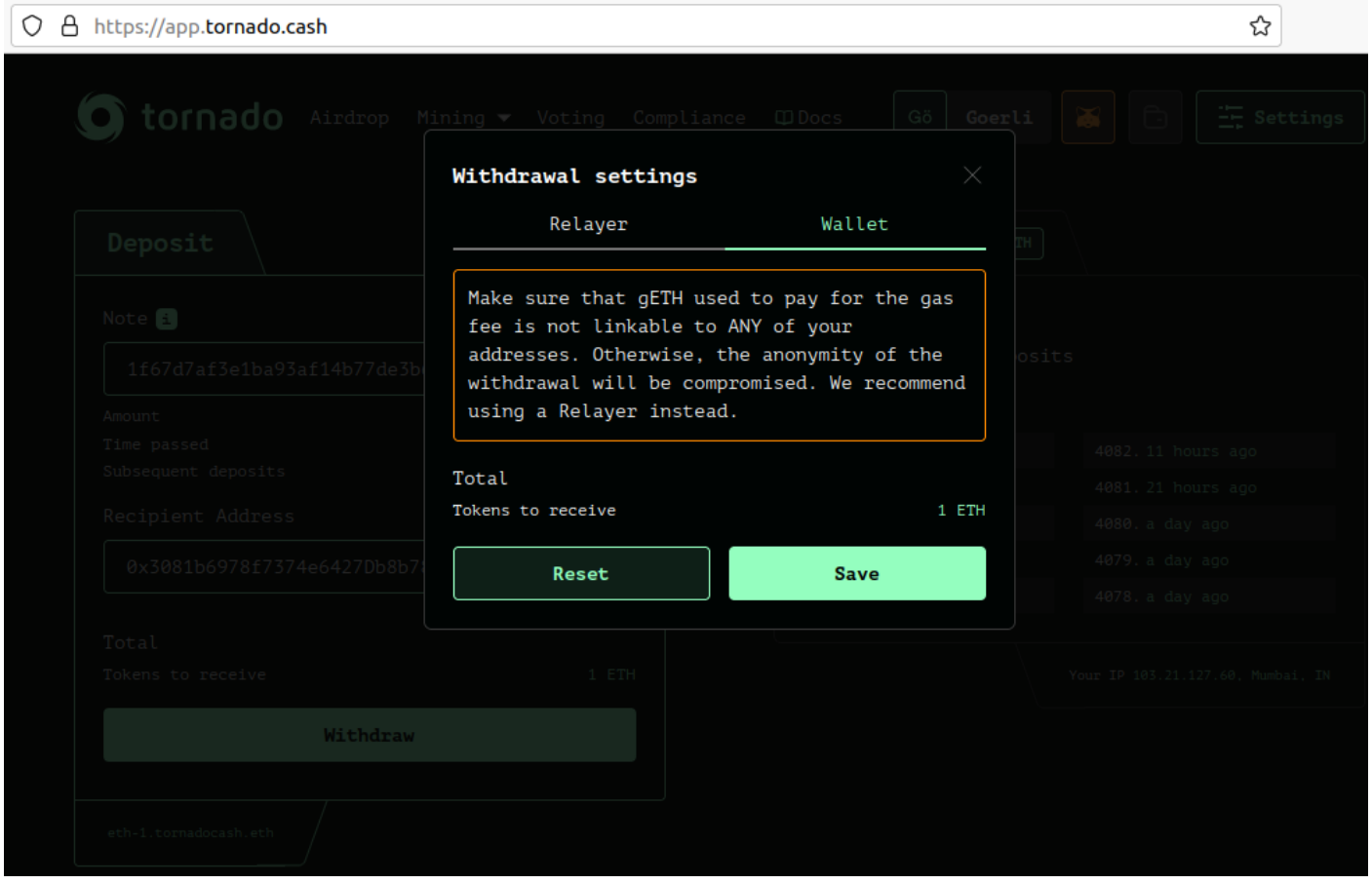
Withdrawal Workflow (2/3)

Choose relayer



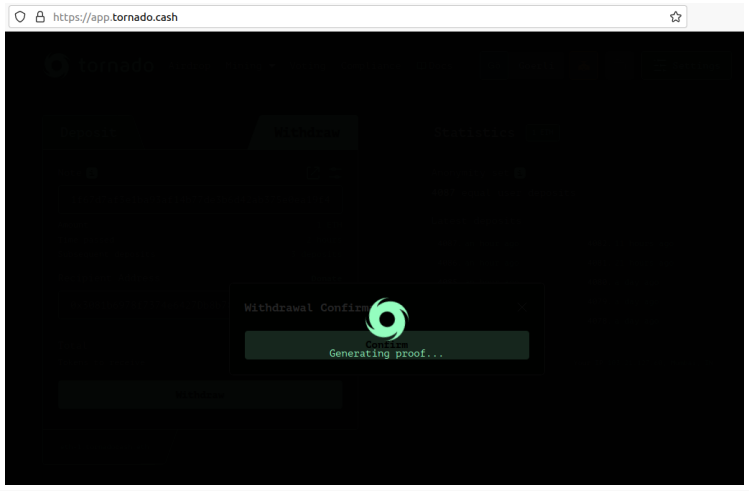
Withdrawal Workflow (2/3)

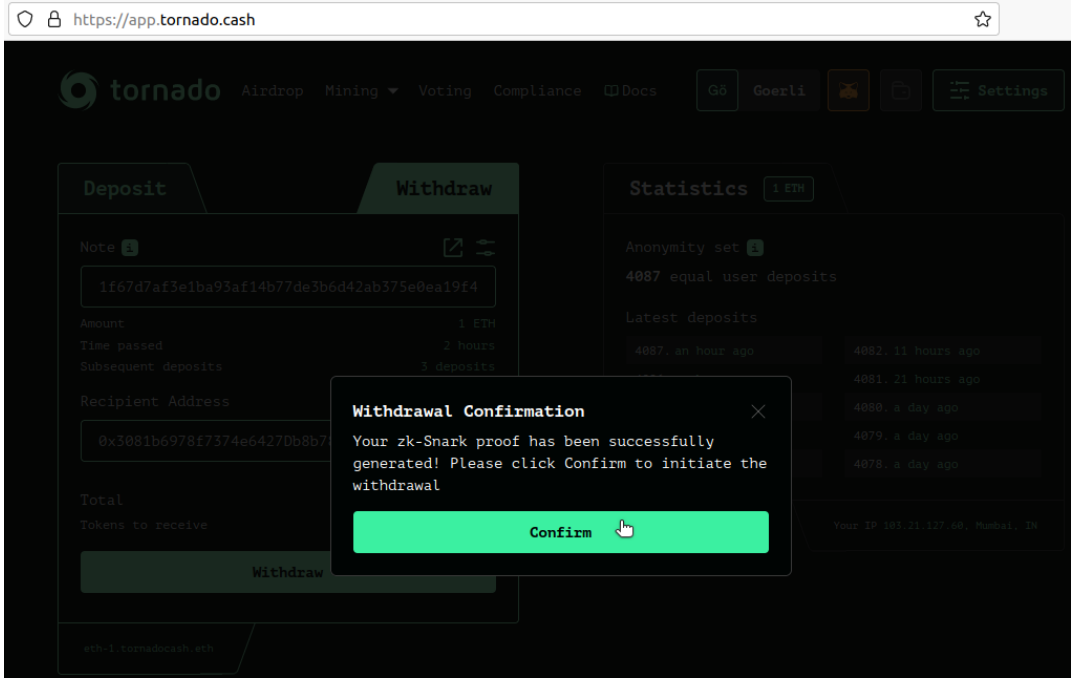
Choose wallet if you have an unlinkable address with ETH



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);
```

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 and later released on bail
- Pertsev's trial began on March 26, 2024. Verdict expected on May 14

References

- Tornado Cash App `https://tornadoeth.cash/`
- Tornado Cash Docs `https://docs.tornadoeth.cash/`
- Circom `https://docs.circom.io/`
- `https://github.com/tornadocash/tornado-core`
- EFF article on OFAC sanctions
- EFF update in April 2023