Practical zk with Summa

Napoli - 25/5/2023

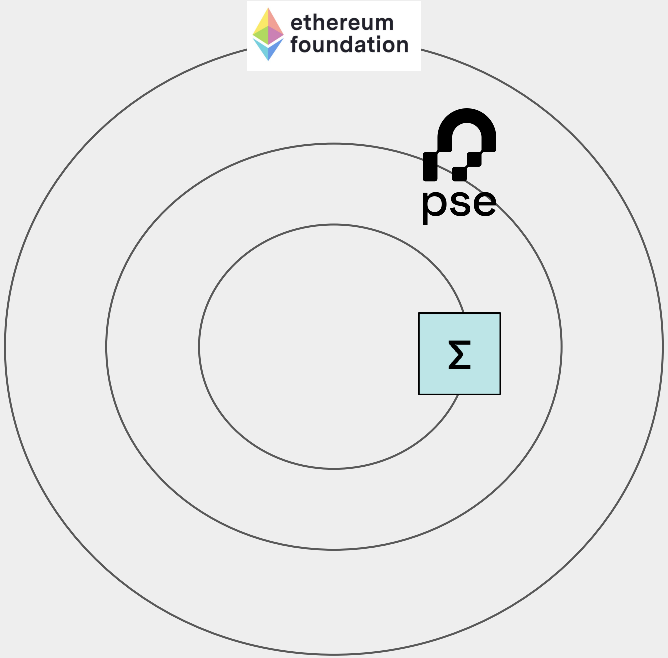
enrico.eth

Me

- Zero knowledge circuit engineer at Summa
- Previously at Polygon Hermez

Summa

- zk Proof of Solvency for CEXes



Goal : Understand zk in practice

#1 Context on ZK

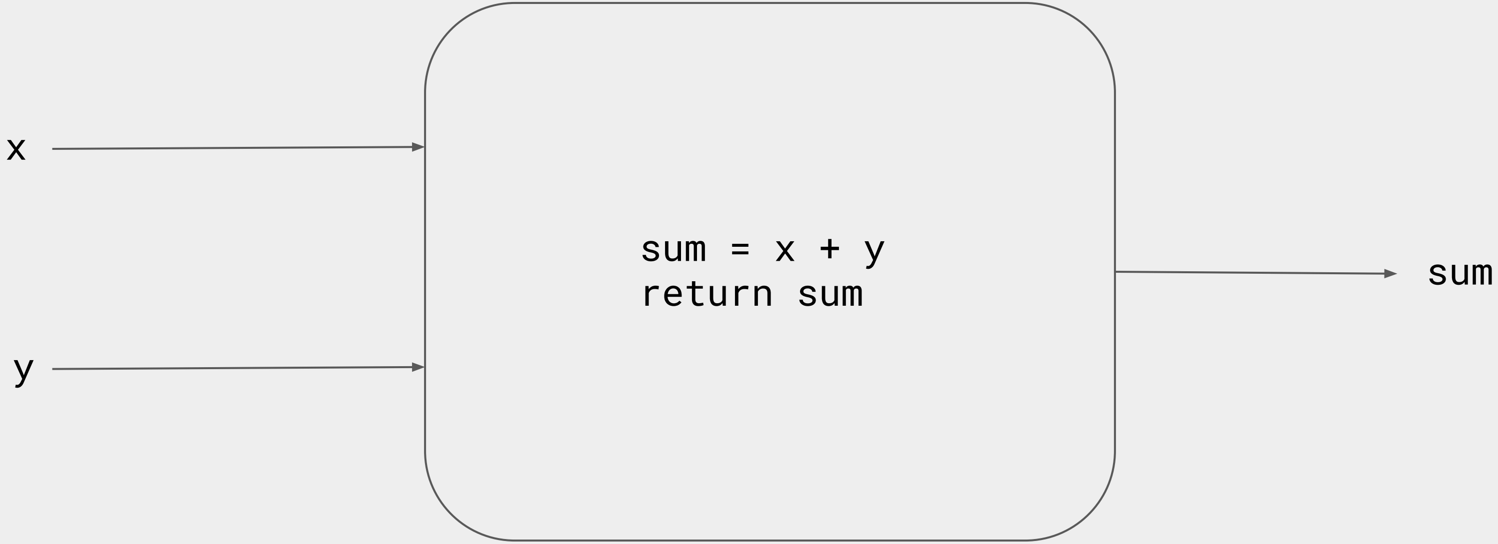
#2 Context on something practical

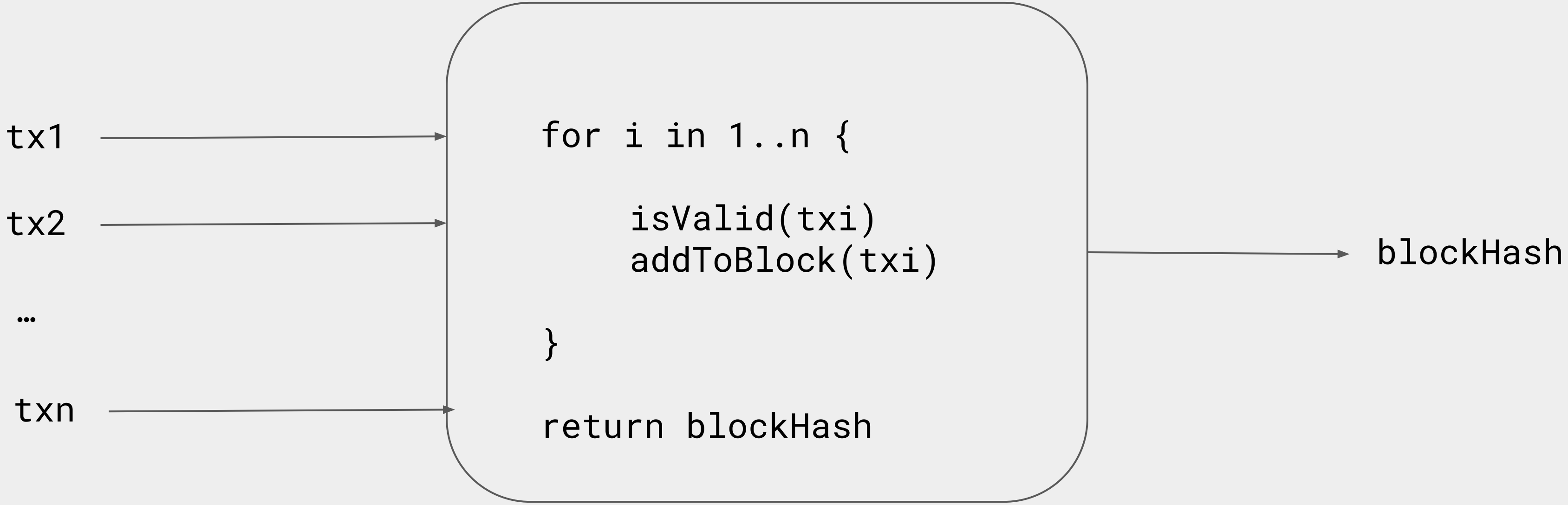
#3 Apply zk to something practical

#1 Context on ZK

Computation = set of rules

```
// Function that computes the sum of two numbers
function computeSum(x: number, y: number): number {
  // Rule 1: Compute the sum of 'x' and 'y' and return the result.
  const sum: number = x + y;
  return sum;
}
```

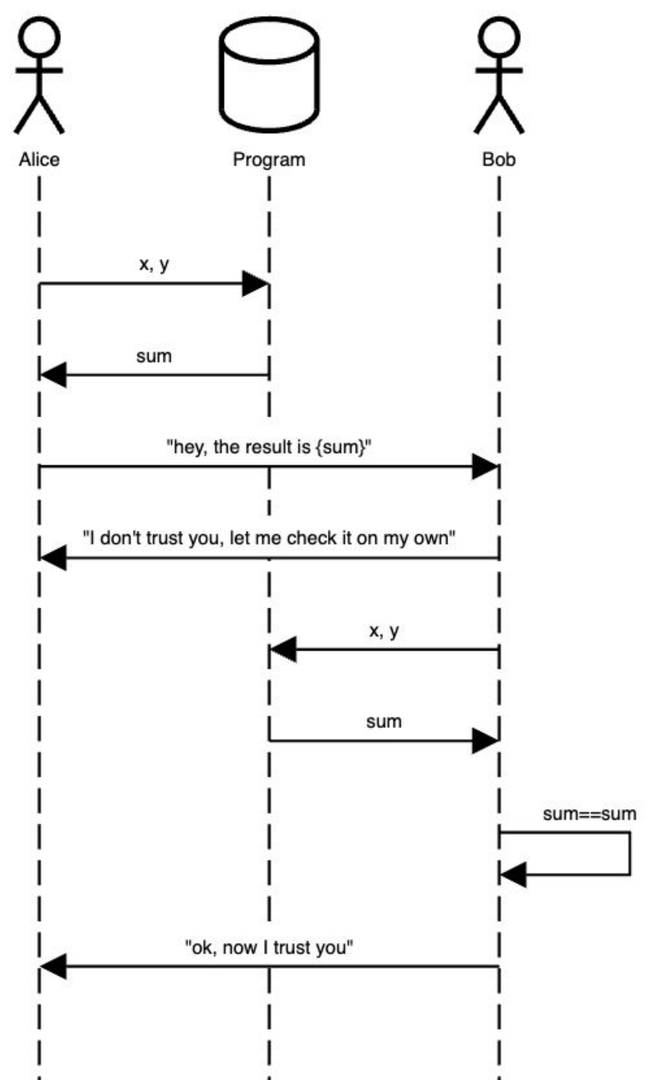

x
y
sum = x + y
return sum
sum


tx1
tx2
…
txn
for i in 1..n {
isValid(txi)
addToBlock(txi)
}
return blockHash
blockHash

Computational Integrity Guarantee

Given a computation which rules are known by everyone, a **prover** wants to prove that the output is the result of running the computation on certain inputs

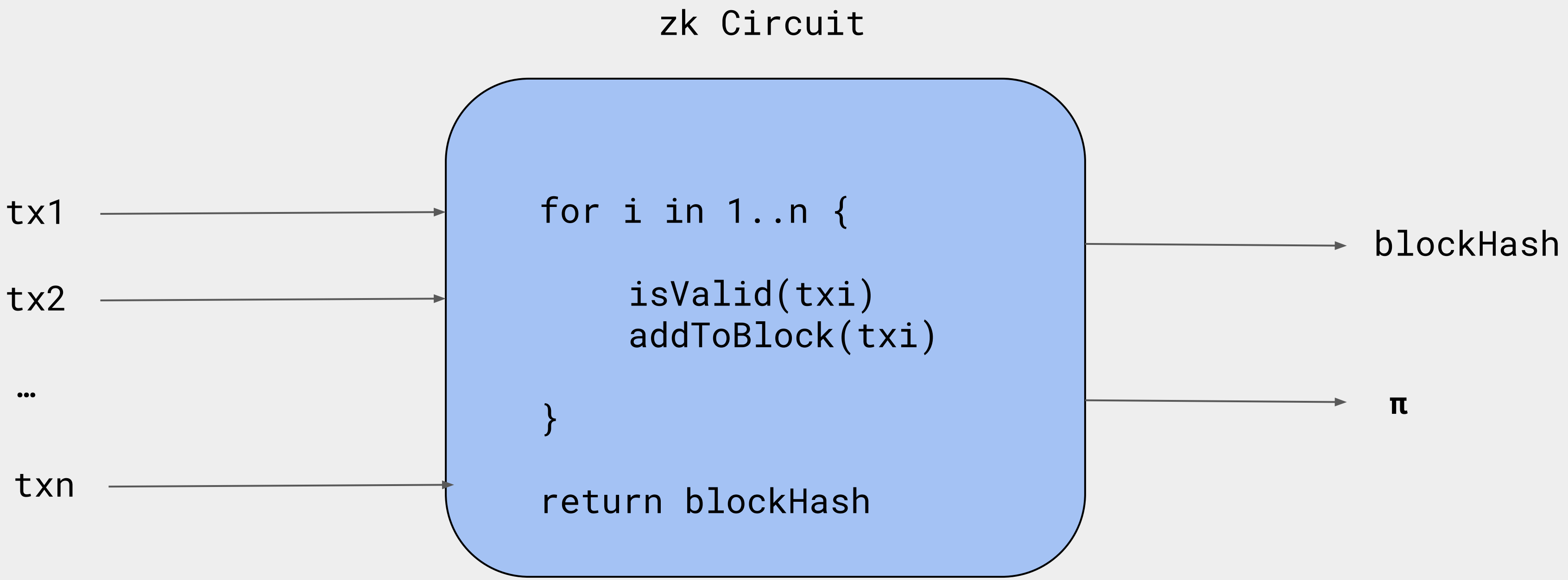
Traditional Prover Verifier dynamic

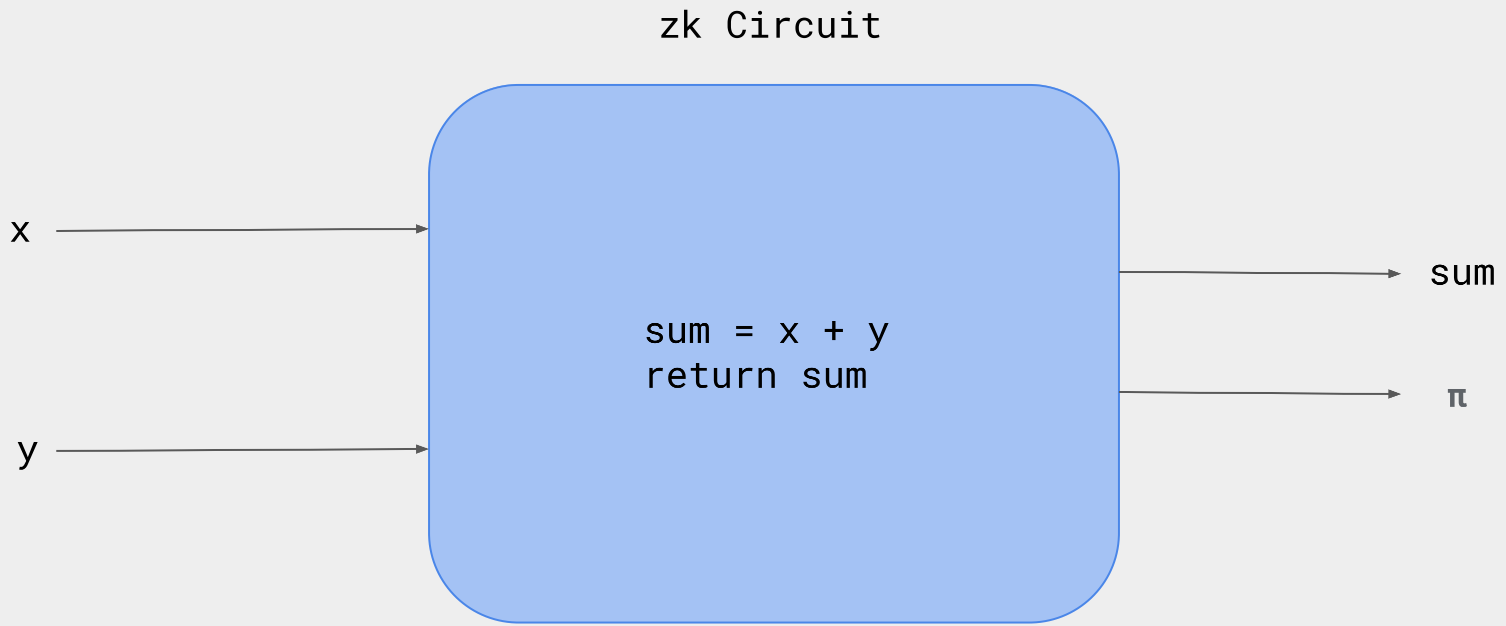


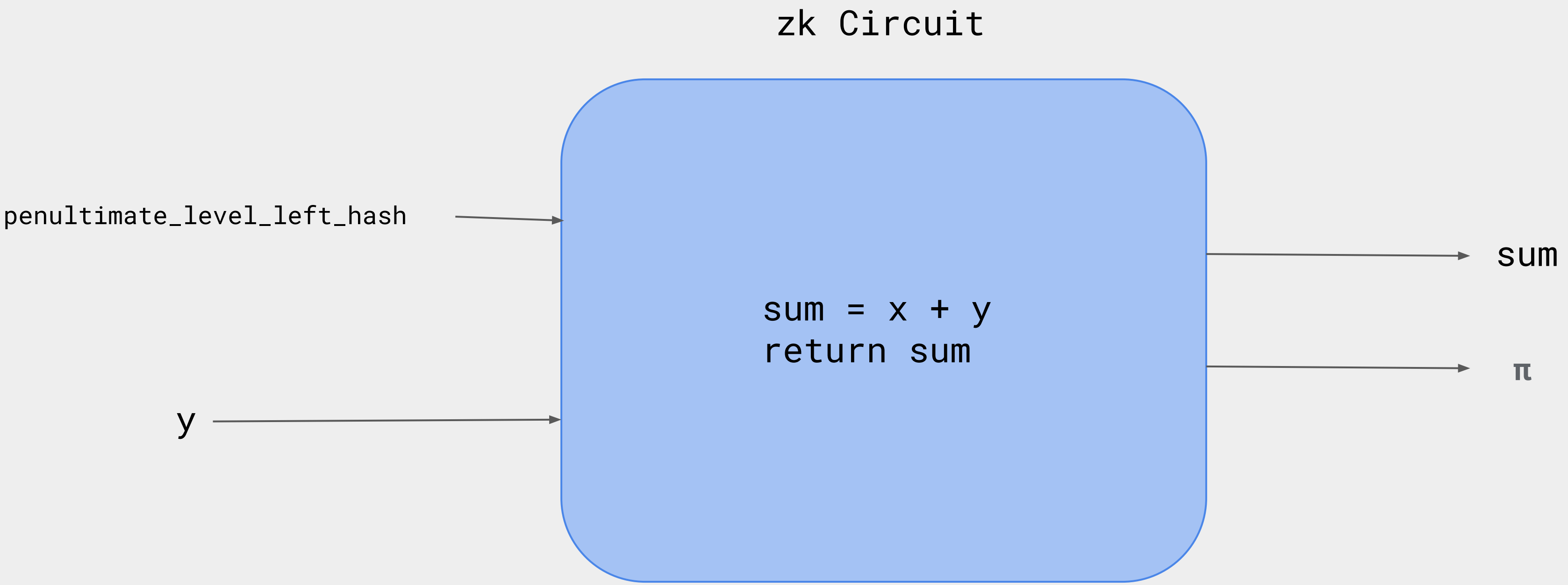
How to achieve Computational Integrity Guarantee?

- ➔ Verifier needs to rerun the computation with the same input and check that the output matches
- ➔ Issue #1: verification time is linear to the computation
- ➔ Issue #2: everything is public!

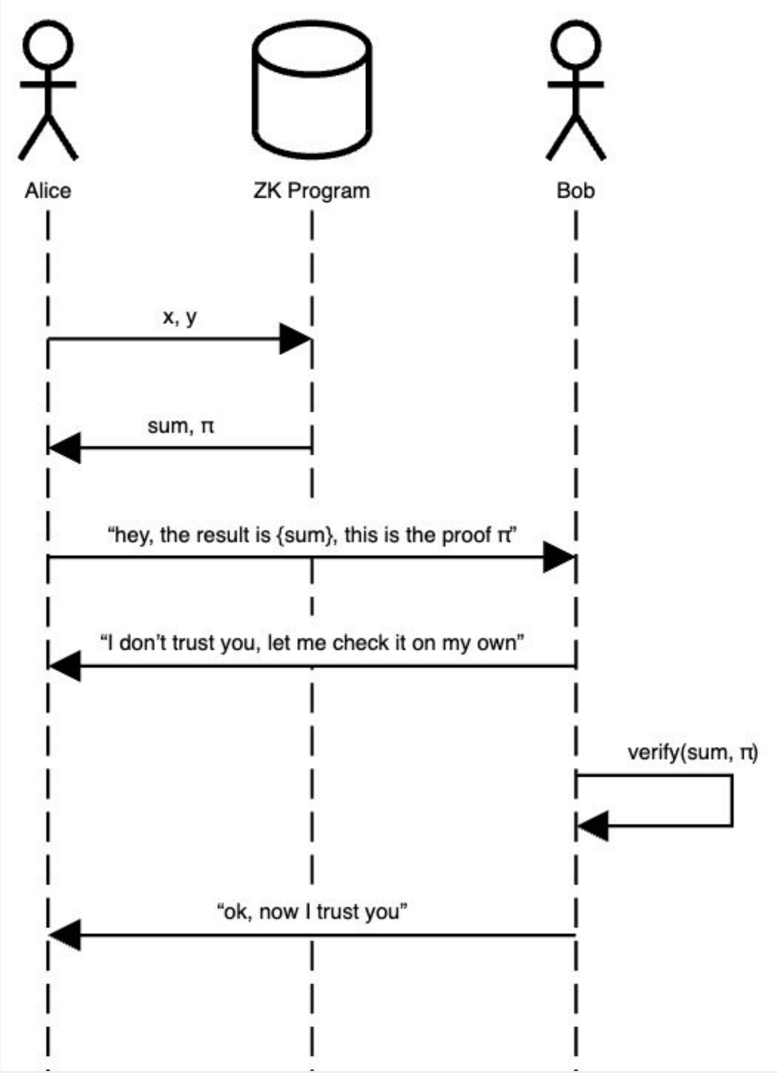
Enters ZK


zk Circuit
tx1
tx2
…
txn
for i in 1..n {
isValid(txi)
addToBlock(txi)
}
return blockHash
blockHash
π


zk Circuit
x
y
sum = x + y
return sum
sum
π


zk Circuit
penultimate_level_left_hash
y
sum = x + y
return sum
sum
π

ZK Prover Verifier dynamic

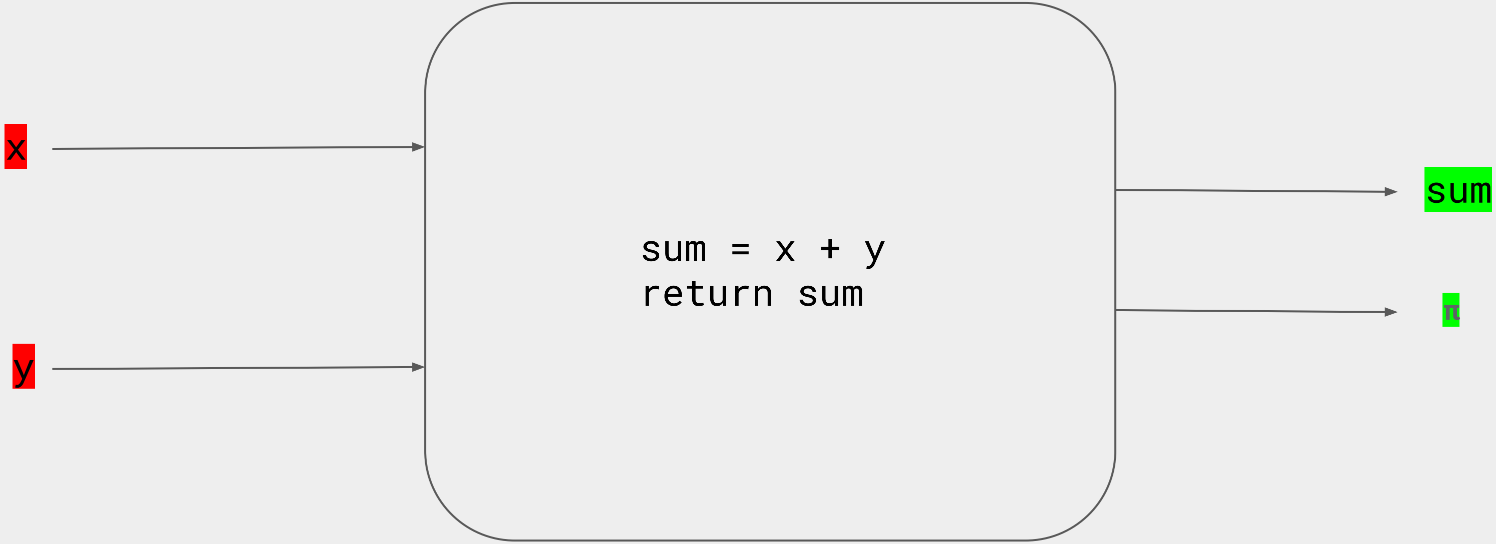


How to achieve Computational Integrity Guarantee?

- ➔ Verifier needs to run a verification algorithm on the proof π
- ➔ Solved Issue #1: verification time is constant no matter the time it took to run the computation
- ➔ Solved Issue #2: the prover can selectively decide what to keep private and what to keep public

How to achieve Computational Integrity Guarantee?

- Verifier needs to run a verification algorithm on the proof π
- Solved Issue #1: verification time is constant no matter the time it took to run the computation
- Solved Issue #2: the prover can selectively decide what to keep private and what to keep public
- New issue #1: Generating a proof for a computation is way slower than just running the computation
- New issue #2: Writing zk program is not as easy as writing a normal program


x
y
sum = x + y
return sum
sum
π

#2 Context on Something Practical

Proof of Solvency

- Cryptographic proof that a CEX is solvent at a specific moment in time

Proof of Solvency

- Cryptographic proof that a CEX is solvent at a specific moment in time

Assets >= Liabilities

LIABILITIES

- Deposits of the users
- Denominated in ETH, BTC, USDC …
- Do not live on-chain, live in the CEX’s DB

ASSETS

- Cryptographic assets (ETH, BTC, USDC…) controlled by the CEX
- Live on-chain
- Should map (at least) 1:1 the deposits of the users

LIABILITIES

- Deposits of the users
- Denominated in ETH, BTC, USDC …
- Do not live on-chain, live in the CEX’s DB

Proof Of Solvency

- Cryptographic proof that a CEX is solvent at a specific moment in time

Assets >= Liabilities

Users are confident that they can withdraw at any time

#3 Apply ZK to something practical

Summa: ZK Proof of Solvency

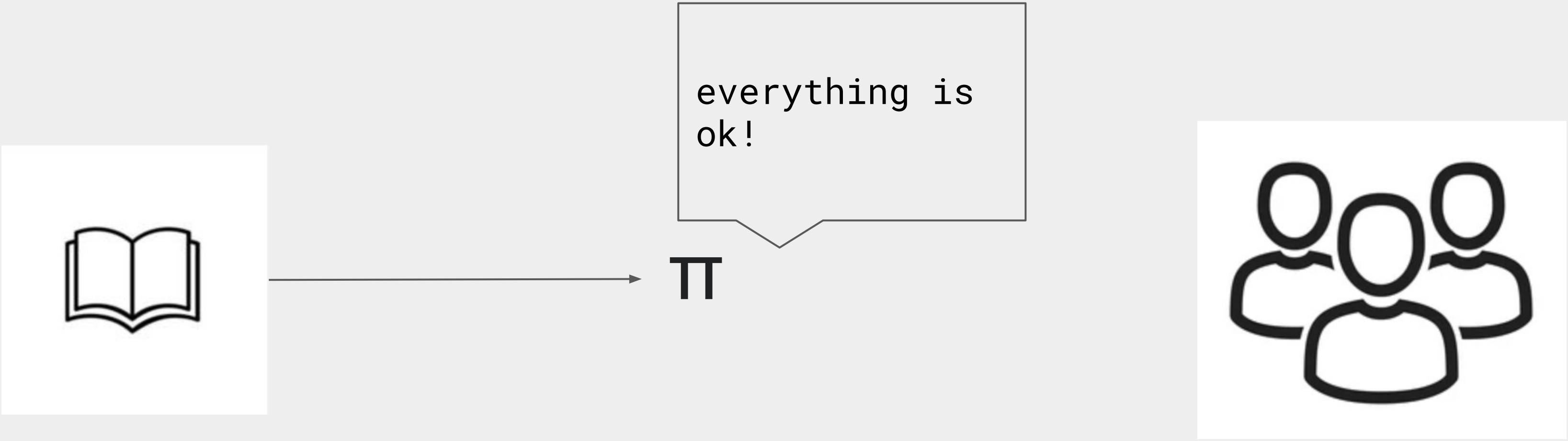
auditor-based proof of solvency

everything is ok!

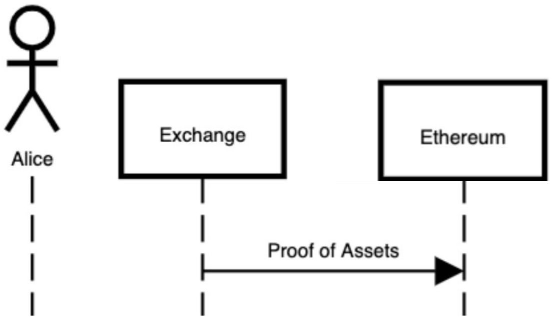
auditorless proof of solvency (naive approach)

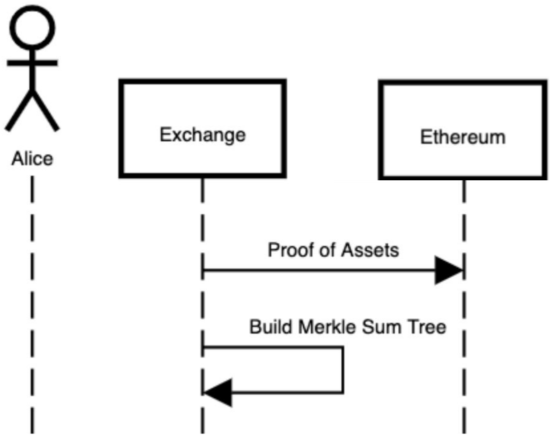
everything is ok!

auditorless proof of solvency
(ZK approach)

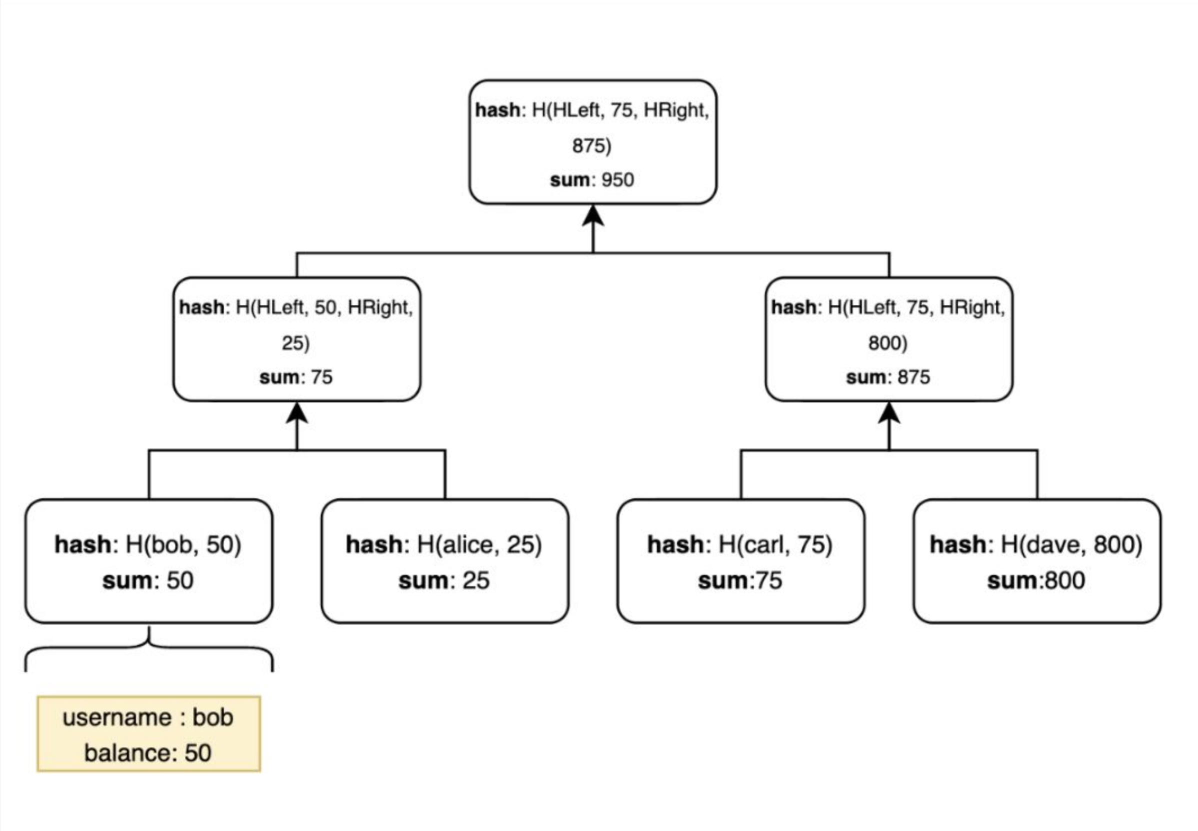


How?

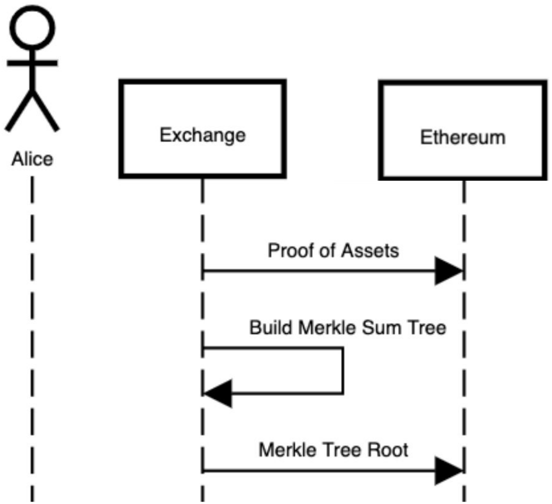

Alice
Exchange
Ethereum
Proof of Assets


Alice
Exchange
Ethereum
Proof of Assets
Build Merkle Sum Tree

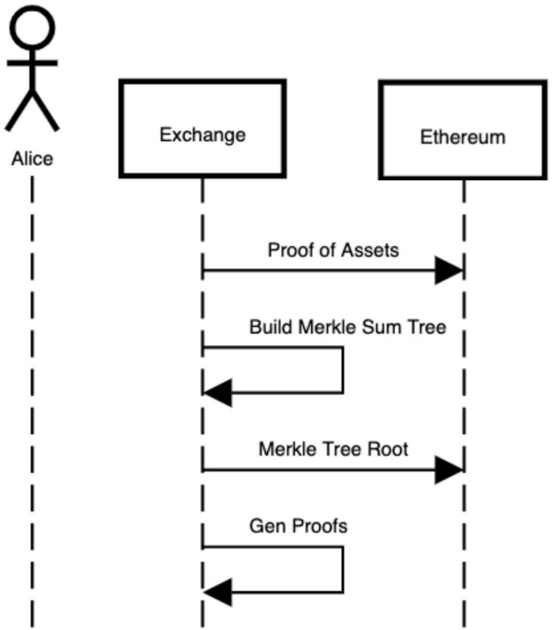
Merkle Sum Tree

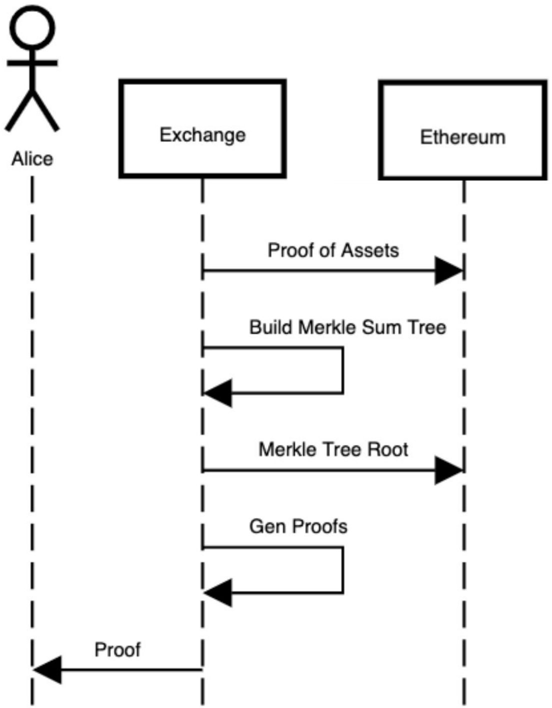


- The entries are the users' data (= liabilities)
- Lives off-chain
- Only the root-hash gets published on-chain



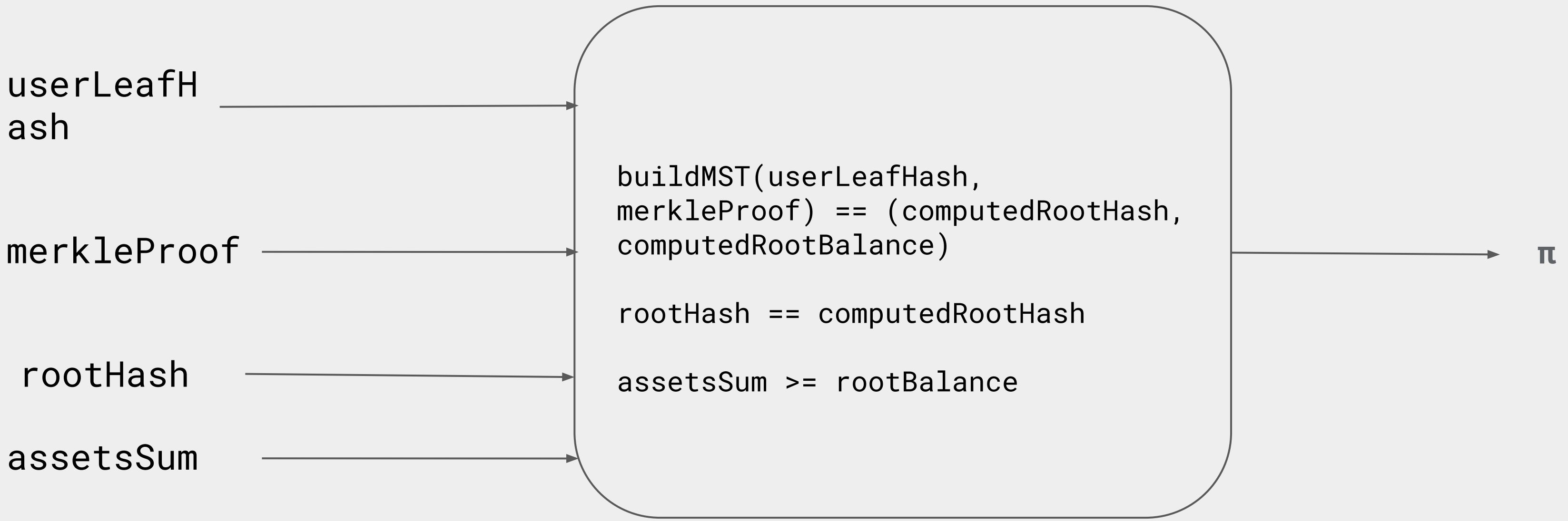
Alice
Exchange
Ethereum
Proof of Assets
Build Merkle Sum Tree
Merkle Tree Root


Alice
Exchange
Ethereum
Proof of Assets
Build Merkle Sum Tree
Merkle Tree Root
Gen Proofs


Alice
Exchange
Ethereum
Proof of Assets
Build Merkle Sum Tree
Merkle Tree Root
Gen Proofs
Proof

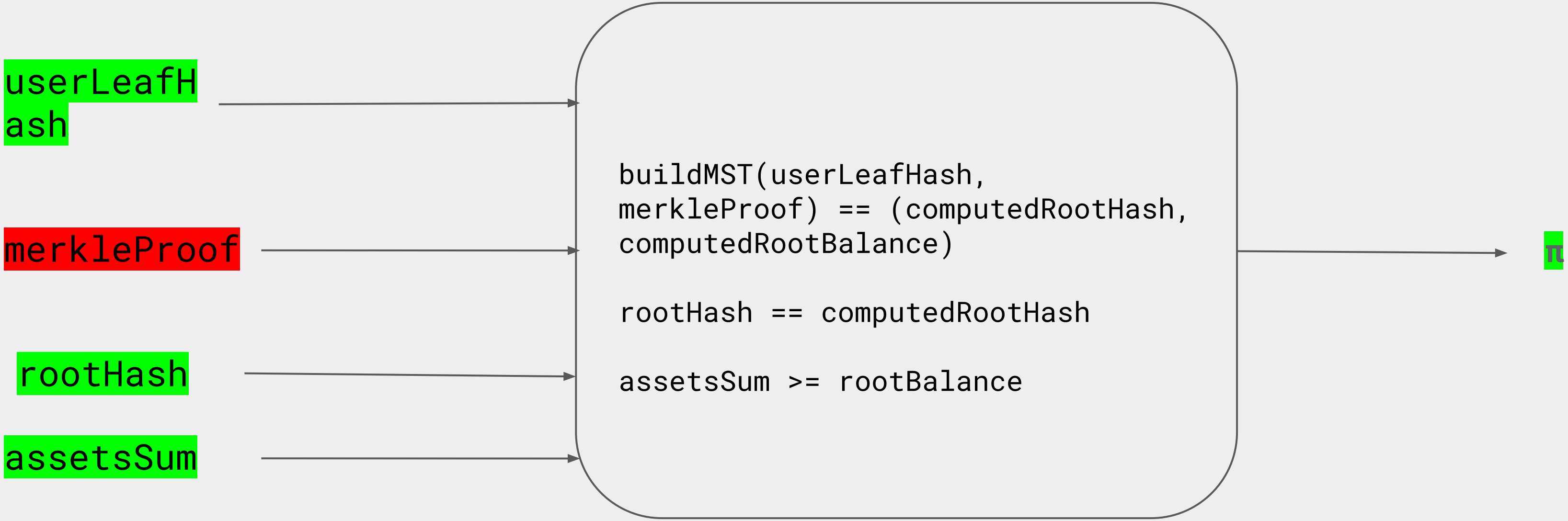
Zk Proofs - Program Rules

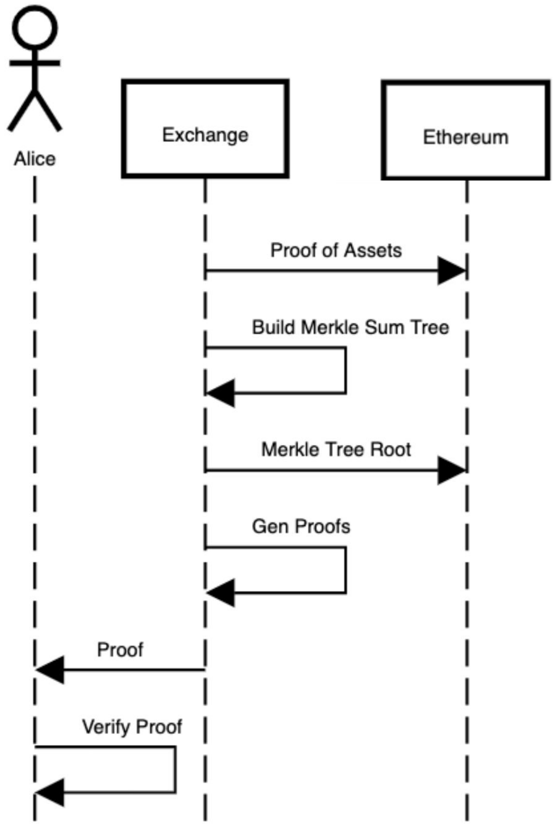
- ➔ Rule#1: The user (identified by its username) is included in the Merkle Sum Tree with the correct balance
- ➔ Rule#2: The hash of the Merkle Sum Tree matches the one committed on chain
- ➔ Rule#3: The sum of liabilities is Less Than the assets of the exchange (as committed in step 1)
- ➔ Rule#4: No sum overflow happened in the merkle sum tree computation


userLeafH
ash
merkleProof
rootHash
assetsSum
buildMST(userLeafHash,
merkleProof) == (computedRootHash,
computedRootBalance)
rootHash == computedRootHash
assetsSum >= rootBalance
π

Zk Proofs - secrecy

- Other users information such as their balances and usernames
- Total number of users
- Total amount of liabilities


userLeafHash
merkleProof
rootHash
assetsSum
buildMST(userLeafHash, merkleProof) == (computedRootHash, computedRootBalance)
rootHash == computedRootHash
assetsSum >= rootBalance
π


Alice
Exchange
Ethereum
Proof of Assets
Build Merkle Sum Tree
Merkle Tree Root
Gen Proofs
Proof
Verify Proof

Proof Verification

$F(\pi, username, balance, assetsSum, rootHash) = yes/no$

Conclusions

- How to think of zk apps: the mental model
  - Given a computation which rules are known by everyone, a prover wants to prove that the output is the result of running the computation on certain inputs, without revealing (part of) the input of such computation
- How to build zk apps
  - Building zk apps means writing circuits. Circom is the best tool to get started

Thank you!

me on github