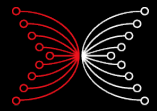


INPUT | OUTPUT

Alex Chepurnoy and Amitabh Saxena

ZeroJoin: Combining Zerocoin and CoinJoin

Privacy in the UTXO model

- ZeroJoin works only in the UTXO model
- Similar to other protocols it is based upon
  - Zerocoin
  - CoinJoin

What do we mean by privacy?

- Informally: unlinkability

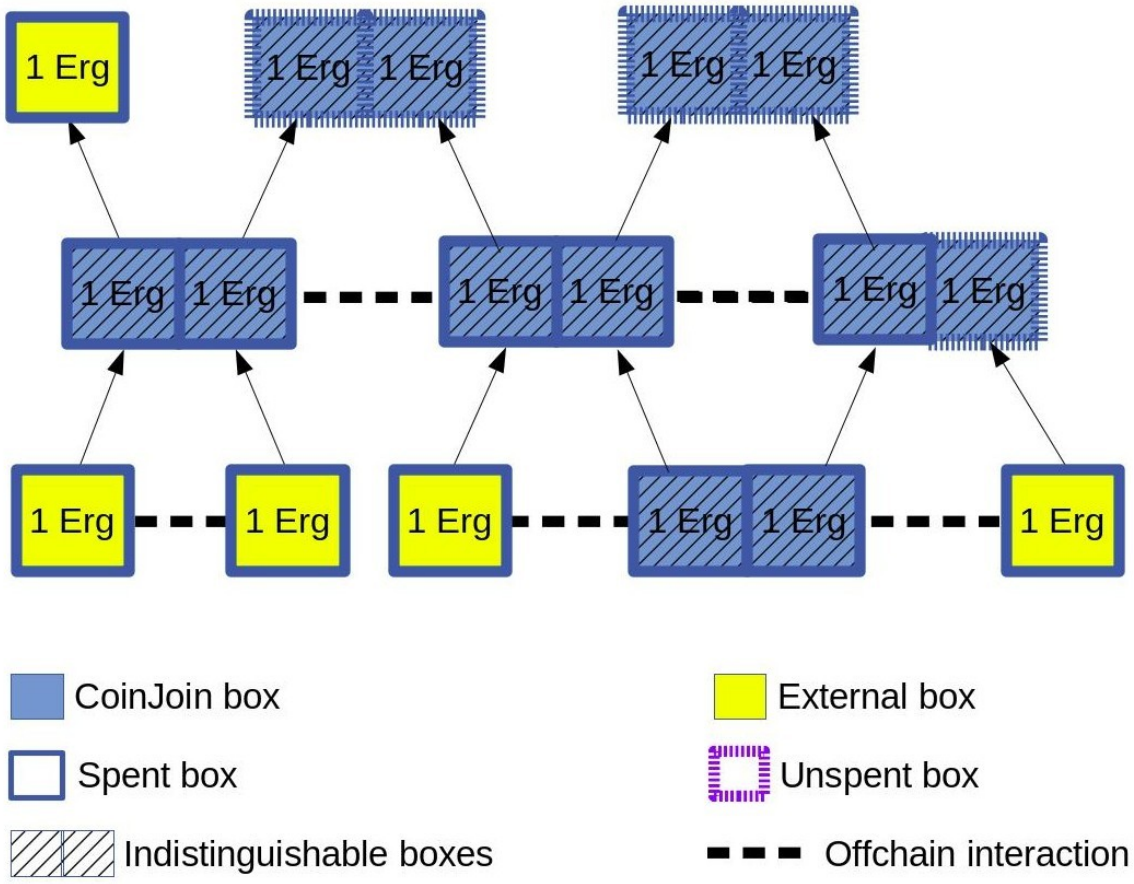
- For security parameter ***k***

  Pr[guessing link] < $1/2^{k}$

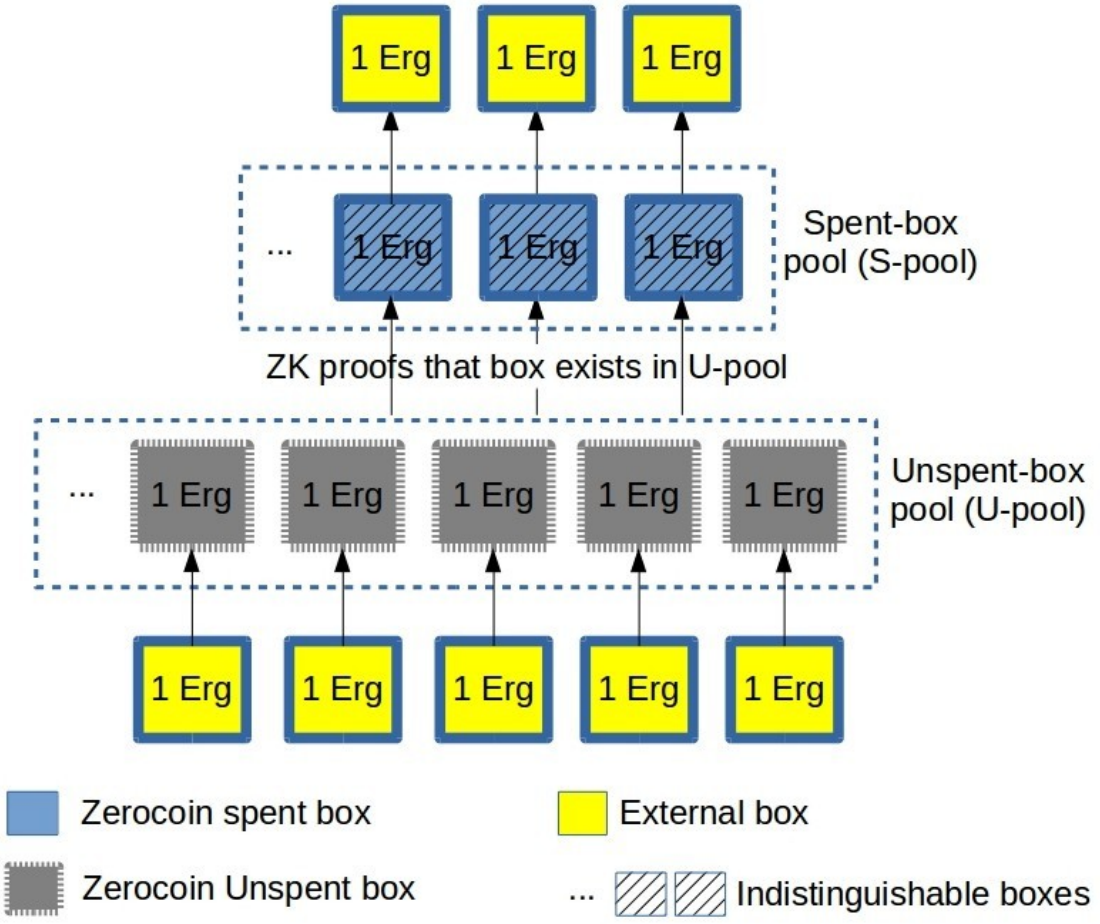
State of the art

	Monotonic UTXO set	***Interaction needed***	***Generic NIZKs/ range proofs***	***Eavesdropping attack***
CoinJoin	no	**yes**	no	**yes**
Zerocoin	**yes**	no	**yes**	no
Zcash	**yes**	no	**yes**	no
Monero	**yes**	no	no	no
MimbleWimble	no	no	**yes**	**yes**
OWAS (CS)	no	no	no	**yes**
Quisquis	no	no	**yes**	no
Confidential Tx	no	no	**yes**	**yes**
ZeroJoin	no	no	no	no

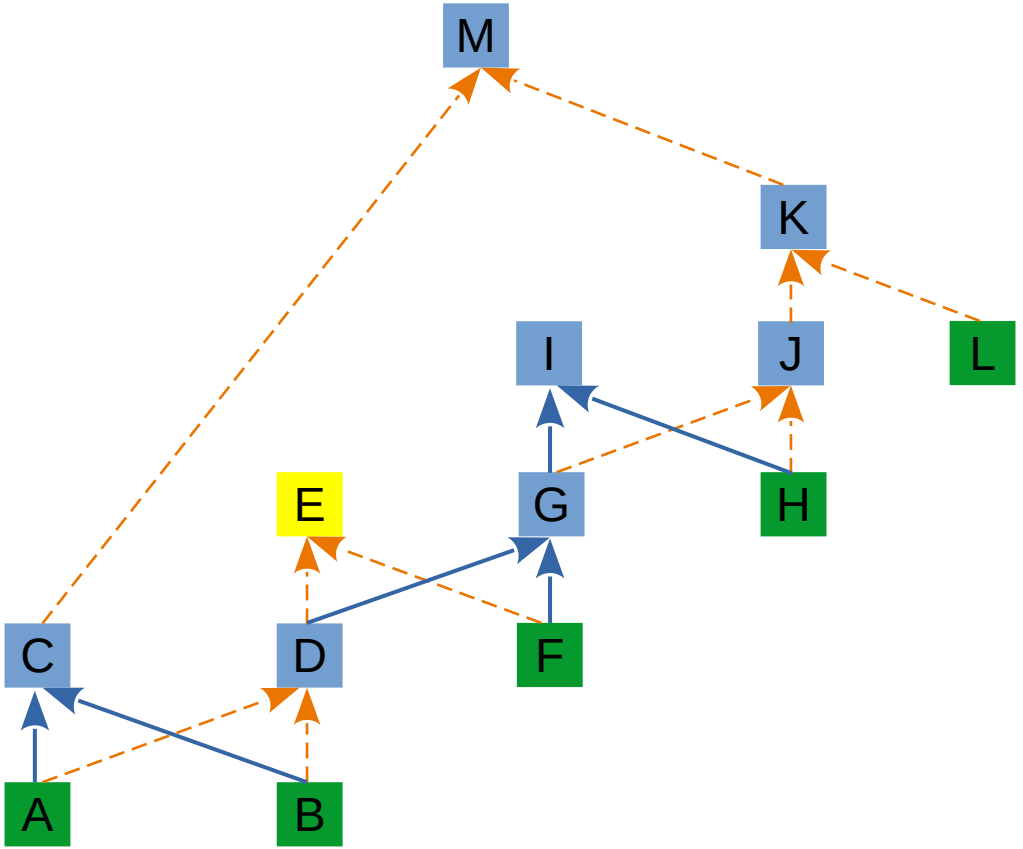
Canonical CoinJoin



Zerocoin


1 Erg
1 Erg
1 Erg
...
1 Erg
1 Erg
1 Erg
Spent-box
pool (S-pool)
ZK proofs that box exists in U-pool
...
1 Erg
1 Erg
1 Erg
1 Erg
1 Erg
Unspent-box
pool (U-pool)
1 Erg
1 Erg
1 Erg
1 Erg
1 Erg
Zerocoin spent box
External box
Zerocoin Unspent box
...
Indistinguishable boxes

2-Coin (hypothetical)

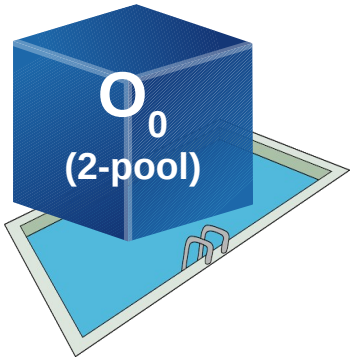


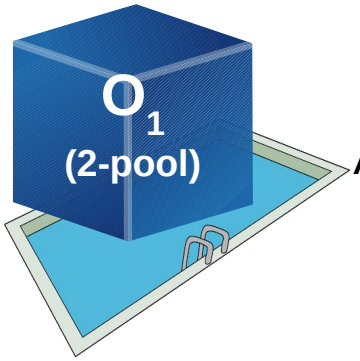
- Users can add boxes to pool
- Each box in pool can be spent twice
  - First is called half-spent
  - Second is called fully spent
- Fully spent boxes removed from pool
- Of the two spents
  - One is by owner
  - Other is by random partner

Half spent (1st)
Fully spent (2nd)
Pool to pool (internal)
Deposits (from external)
Withdraws (to external)

2-Coin primitive

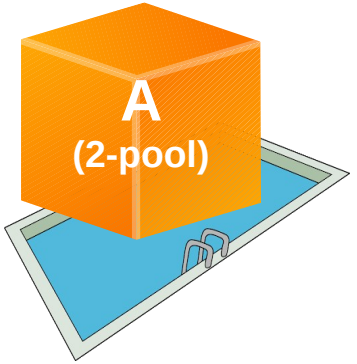
Both boxes are indistinguisble to outsiders

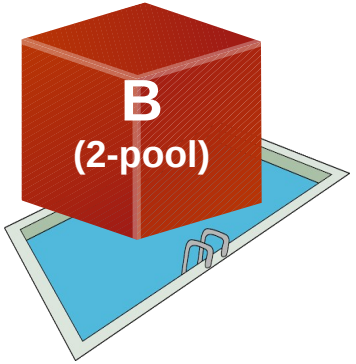




Alice spends her and Bob's boxes to create a box spendable by her

Bob spends his and Alice's boxes to create a box spendable by him



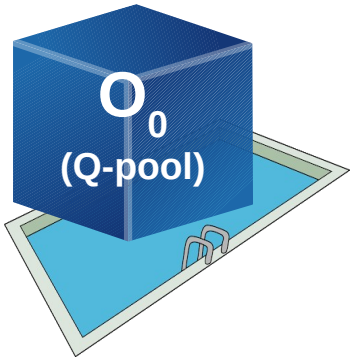


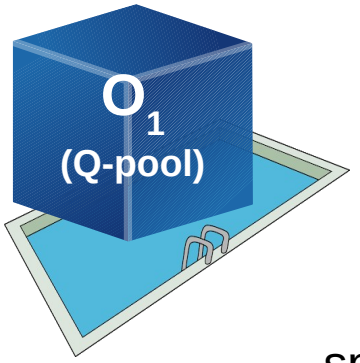
Bob adds box to pool

Alice adds box to pool

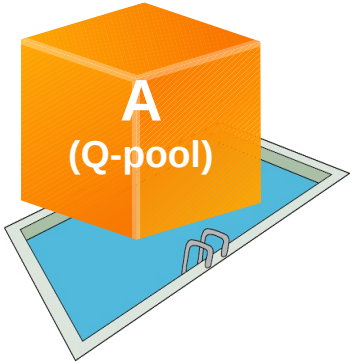
Quisquis primitive

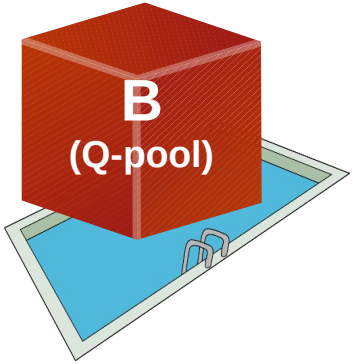
Both boxes are indistinguisble to outsiders





Bob spends his and Alice's boxes to create two new boxes, one spendable by Alice and other by Bob



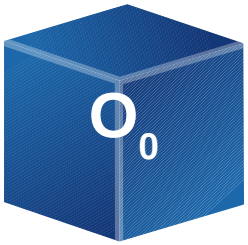


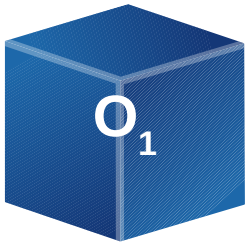
Bob adds box to pool

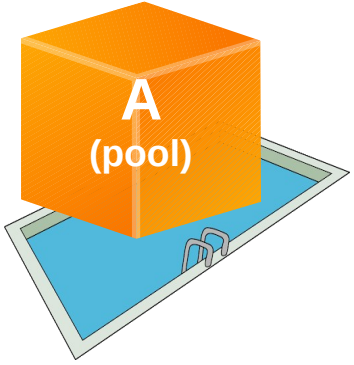
Alice adds box to pool

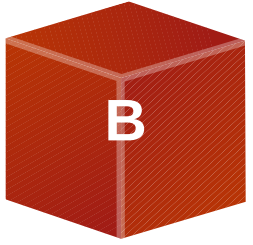
ZeroJoin primitive

Both boxes are indistinguisble to outsiders









Compared to 2-coin/Quisquis:

New boxes not part of pool
(need another tx to add to pool)

Only one input of tx from pool

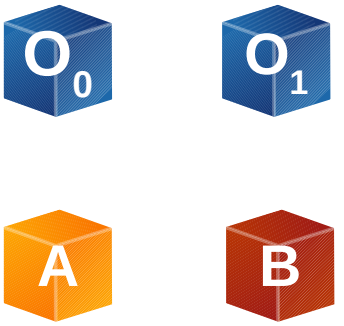
Bob spends his and Alice's boxes
to create two new boxes, one
spendable by Alice and other by Bob

Alice adds box to pool

Quisquis vs ZeroJoin

Similarities

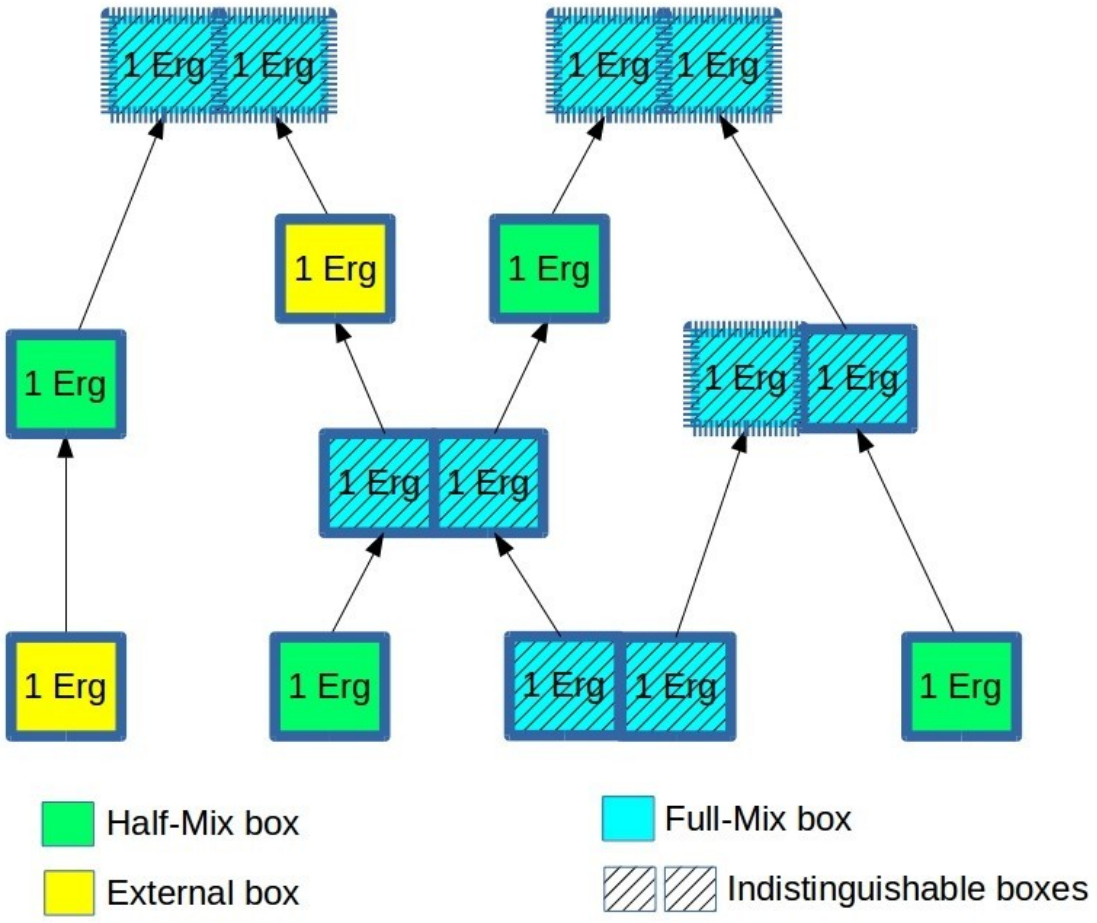
- Alice adds a box A to pool
- A teqhnique for Bob to spend A with one of his own box B to create two boxes s.t:
  - One spendable only by Alice and other only by Bob
  - Outsiders cannot distinguish which belongs to whom
- Above works assuming Bob follows certain rules (“Honest-but-curious”)
  - Because boxes are indistinguishable, not possible to publicly verify if Bob is misbehaving
- Rest of the protocol designed to ensure that Bob follows rules (via ZK proofs)
  - Proof that boxes are constructed properly without leaking any other info



Differences in underlying ZK proofs

- Generic NIZKs in Quisquis (several kilobytes and few hundred exponentiations to verify)
- Sigma proofs in ZeroJoin (several bytes and 8 exponentiations to verify)

ZeroJoin


1 Erg
1 Erg
1 Erg
1 Erg
1 Erg
1 Erg
1 Erg
1 Erg
1 Erg
1 Erg
1 Erg
1 Erg
1 Erg
1 Erg
Half-Mix box
External box
Full-Mix box
Indistinguishable boxes

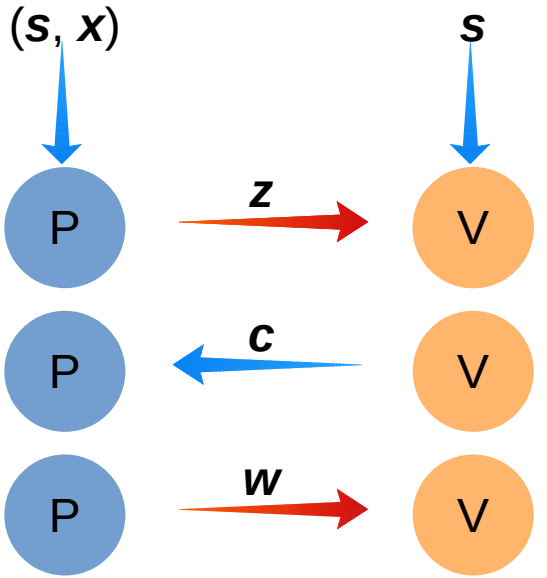
Notation

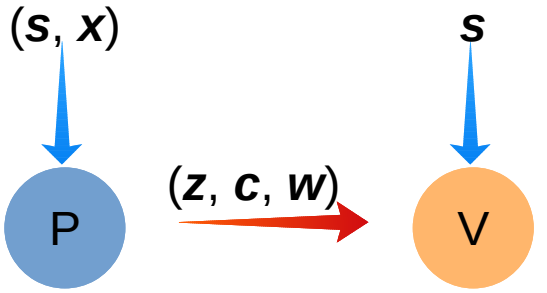
- Setup: ***s*** is a “statement”, ***x*** is the secret witness for ***s***
  - Prover (P) is given (***s***, ***x***)
  - Verifier (V) is given ***s***
- Sigma **proof of knowledge** of ***x*** such that ***s***(***x***) is true:
  - Prover sends a comitment ***z*** to Verifier (commit)
  - Verifier sends a challenge ***c*** to Prover (challenge)
  - Prover sends a response ***w*** to Verifier (response)

  Verifier accepts if (***z***, ***c***, ***w***) is an “accepting transcript”

  We call this **PK3(x) s.t. s(x)** because its a 3-round protocol
- We also denote **PK(x) s.t. s(x)** as a protocol:
  - Prover sends a tuple (***z***, ***c***, ***w***) to Verifier

    Verifier accepts if (***z***, ***c***, ***w***) is an “accepting transcript”





Primitives

- [Schnorr] **PK3**(***x***) s.t: $u = g^x$ — proveDlog
- [Schnorr] **PK3**(***x***) s.t: $u = g^x$ **and** $v = h^x$ — proveDHTuple
- [CDS94]
  PK3(***x***) s.t: ***foo***(***x***)
  + **PK3**(***x***) s.t: ***bar***(***x***)

  = **PK3**(***x***) s.t: (***foo***(***x***) **or** ***bar***(***x***))
- [Fiat-Shamir]
  PK3(***x***) s.t: ***foo***(***x***) => **PK**(***x***) s.t: ***foo***(***x***)

What we finally have

- Let (g, u , h , v) be a tuple of form
  (g, g^x, g^y, g^{xy}) [Alice knows x, Bob knows y]

  - **PK**(x) s.t. $u = g^x$ — ***ProveDlog(g, u)***
  - **PK**(y) s.t. $h = g^y$ — ***ProveDlog(g, h)***
  - **PK**(x) s.t. $u = g^x$ and $v = h^x$ — ***ProveDHTuple(g, h, u, v)***
  - **PK**(y) s.t. $h = g^y$ and $v = u^y$ — ***ProveDHTuple(g, u, h, v)*** ***('Dual')***
  - **PK**(x or y) s.t. ($u = g^x$ and $v = h^x$) or ($h = g^y$) — ***ProveDHTuple(g, h, u, v)*** || ***ProveDlog(g, h)***

- More clearly written as:
  - **PK**($?$) s.t. $h = g^?$ and $v = u^?$ — **Bob**
  - **PK**($?$) s.t. ($u = g^?$ and $v = h^?$) or ($h = g^?$) — **Alice and Bob**

Basic protocol

- Alice selects secret ***x*** publishes coin with ***u*** = g^x
- Bob selects secrets ***y*** and bit ***b***
- Bob picks Alice's coin and spends it to generate two equal value coins O_0, O_1 such that:
  - O_b has two registers (***alpha***, ***beta***) = (g^y, g^{xy})
  - O_{1-b} has two registers (***alpha***, ***beta***) = (g^{xy}, g^y)
- Both O_0, O_1 are protected by:

  PK(***?***) s.t. ((***u*** = $g^?$ and ***beta*** = ***alpha***$^?$) or (***beta*** = $g^?$))
- Thus, the statements become:
  - O_b : **PK**(***?***) s.t. (($g^x = g^?$ and $g^{xy} = g^{y?}$) or ($g^{xy} = g^?$))
  - O_{1-b} : **PK**(***?***) s.t. (($g^x = g^?$ and $g^y = g^{xy?}$) or ($g^y = g^?$))

(only) Alice can spend O_b using ***x*** and (only) Bob can spend O_{1-b} using ***y***.

Complete Protocol

Enforce Bob to behave correctly

- Bob picks Alice's coin and spends it to generate two equal value coins O_0, O_1 such that:
  - $O_{\boldsymbol{b}}$ has two registers (***alpha***, ***beta***) = ($\boldsymbol{g^y}$, $\boldsymbol{g^{xy}}$)
  - $O_{1-\boldsymbol{b}}$ has two registers (***alpha***, ***beta***) = ($\boldsymbol{g^{xy}}$, $\boldsymbol{g^y}$)
- For Bob's correct behavior. We need to ensure that
  - One of (***alpha***, ***beta***) or (***beta***, ***alpha***) is of the form ($\boldsymbol{g^y}$, $\boldsymbol{g^{xy}}$)
- This can be done by requiring Bob to prove:
  PK(***?***) s.t.((***alpha*** = $\boldsymbol{g^?}$ and ***beta*** = $\boldsymbol{alpha^?}$) or (***beta*** = $\boldsymbol{g^?}$ and ***alpha*** = $\boldsymbol{beta^?}$))

Ergo Platform

- UTXO based general-purpose blockchain (launched 2019)
  - Smart contract in Scala-like language (ErgoScript)
  - Nonoutsourceable proof-of-work (Autolykos)
  - Storage rent (on-chain garbage collection) – prevent blockchain bloat in the long term
  - Data inputs (use other boxes without spending them)
- Advanced context information for smart contracts
  - Entire transaction, with inputs and outputs
- ErgoMix: ZeroJoin at smart contract layer with fee
  - Implemented as a 2-stage protocol
  - 1st stage (Alice's box) encodes rules for 2nd stage (mixed boxes)
  - Mining fee using secondary tokens and "emission boxes"

Fee in ErgoMix

- Solution 1: suitable for many use-cases
  - Altrustic approach: fee free if remixing
  - No free-loaders: free fee only if remixing
  - Fee-emission box: fee emitted if transaction conforms to above
    - Multiple fee-emission boxes to allow parallel usage
- Solution 2: long-term solution with actual fee
  - In any mix tx, each party must provide some non-zero number of **fee-tokens**
  - For any mix tx, fee emission box requires destruction of one fee-token
  - Remining tokens distributed equally among two outputs
  - One party could pay less fee as long as:
    - All parties start with fixed number of fee-tokens, say 1000
    - Parties can use up fee-tokens only in successive remixes
  - If some box has less tokens, then it has higher privacy

ErgoScript code

https://tinyurl.com/ergomix

```
val $halfMixScriptSource =
  """{
    |  val g = groupGenerator
    |  val gX = SELF.R4[GroupElement].get
    |
    |  val c1 = OUTPUTS(0).R4[GroupElement].get
    |  val c2 = OUTPUTS(0).R5[GroupElement].get
    |
    |  OUTPUTS(0).value == SELF.value &&
    |  OUTPUTS(1).value == SELF.value &&
    |  OUTPUTS(0).R6[GroupElement].get == gX &&
    |  OUTPUTS(1).R6[GroupElement].get == gX &&
    |  blake2b256(OUTPUTS(0).propositionBytes) == fullMixScriptHash &&
    |  blake2b256(OUTPUTS(1).propositionBytes) == fullMixScriptHash &&
    |  OUTPUTS(1).R4[GroupElement].get == c2 &&
    |  OUTPUTS(1).R5[GroupElement].get == c1 && {
    |    proveDHTuple(g, gX, c1, c2) ||
    |    proveDHTuple(g, gX, c2, c1)
    |  } && SELF.id == INPUTS(0).id
    |}""".stripMargin
```

```
val $fullMixScriptSource =
  """{
    |  val g = groupGenerator
    |  val c1 = SELF.R4[GroupElement].get
    |  val c2 = SELF.R5[GroupElement].get
    |  val gX = SELF.R6[GroupElement].get
    |  proveDlog(c2) ||           // either c2 is g^y
    |  proveDHTuple(g, c1, gX, c2) // or c2 is u^y = g^xy
    |}""".stripMargin
```

Handling fee: Summary

- Approximate Fairness via fee tokens (ErgoMix)
  - Bob may pay less fee if he uses “highly mixed” coin
- Altruistic fee (ErgoMix)
  - Free if remixing (fee paid by sponsors)
- Range proofs to hide amounts (but inefficient)

Conclusion

- ZeroJoin
  - Like CoinJoin but non-interactive
    - Secure from eveasdropping attacks
  - Like Zerocoin but with non-monotonic UTXO set
    - ZK proofs to spend unlinkably
  - Like Quisquis but with much shorter and faster proofs
    - Sigma protocols instead of NIZKs
- Open problems
  - Can we have 2-coin structure (pool-to-pool transactions)?
    - Right now we need an extra tx to add to half-mix pool
  - Better fee approach than approximate fairness?