**  
  
  
  
  
  
  
How Ark works at tech level:**

The Ark Protocol is a Layer 2 solution for Bitcoin that enables off-chain transactions while maintaining the ability to settle on-chain when necessary. The protocol revolves around Virtual Transaction Outputs (VTXOs), which are virtual representations of Bitcoin UTXOs that can be exchanged off-chain.  
  
  
Key architectural components include:

* Boarding outputs for fund entry
* Virtual transaction outputs (VTXOs) for off-chain transactions
* Round-based settlement process
* Unilateral and collaborative exit mechanisms

**Boarding Transaction:**

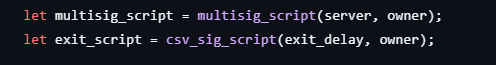
1 . Boarding output creation:

A **Boarding Output** is a Taproot output created during the process of entering

The ark ecosystem.

Taproot output builds using : Two scripts;

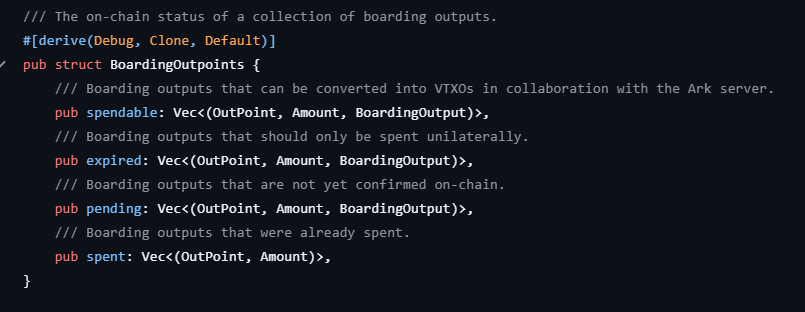
1. A **multisig script** (server + owner) for cooperative spends.
2. A **CSV script** (CheckSequenceVerify) that allows unilateral claim by the user **after a time delay**



* The boarding output's address is derived from this multisig construction, with funds sent to this address entering the Ark ecosystem.

After Sending Funds to Boarding Output’address : USER CAN MONITOR ONCHAIN STATUS

Given a list of [`BoardingOutput`]s, determine their on-chain status.:

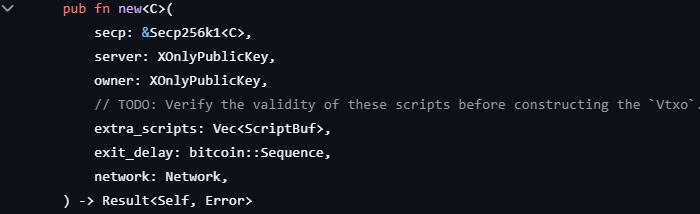


**Join the Ark: Creation of VTXOs (Virtual UTXOs)**

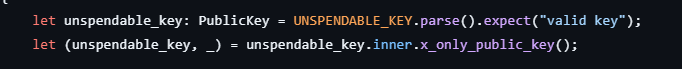
Once a boarding output is confirmed on-chain, the Ark client works with the ark-server to transform this output into VTXOs, which are spendable assets within the Ark ecosystem.

Build a vtxos ;

The `extra\_scripts` argument allows for additional spend paths;



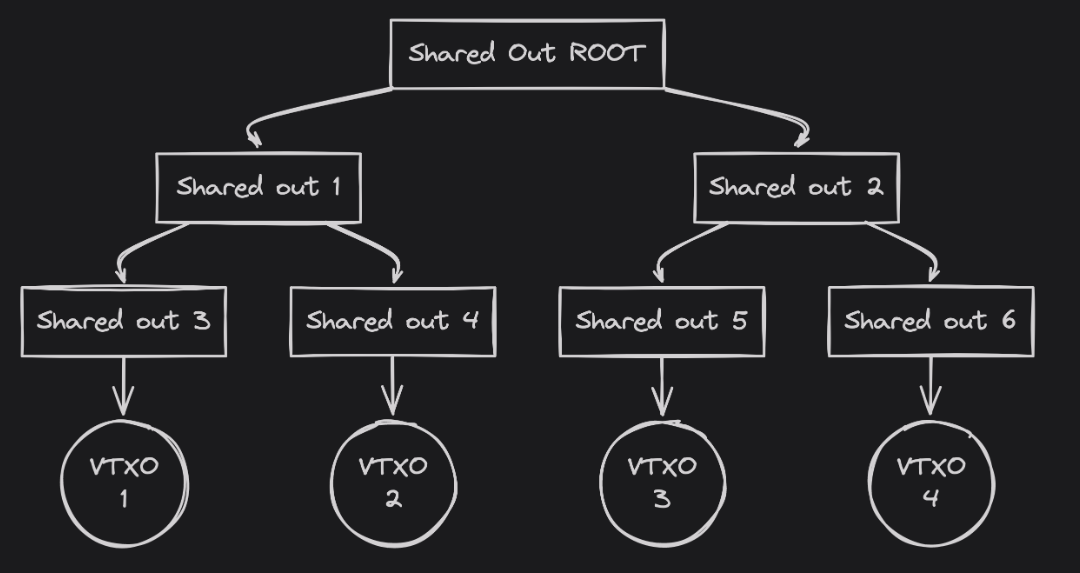
A VTXO is locked by a [taproot](https://bips.dev/341/) script that must contain the following spending conditions:

* unspendable key path

* any script path must be either a **collaborative** or a **exit**

**VTXO Tree**[**​**](https://arkdev.info/docs/learn/concepts#vtxo-tree)

VTXOs are created by a shared outputs. The root transaction of the VTXO tree spends the shared output and splits it into 2 other shared outputs, which are respecively split into other 2 shared outputs, etc.  
At the leaf level of this binary tree we find the 1-input-1-output transactions. The outputs of these transactions are the VTXOs of the users.



Each **VTXO** (Virtual Transaction Output) is associated with a **Shared Output** –funded by the server and encumbered by a set of script paths. These allow controlled spending under different conditions

**i) Unspendable Key Path:**



**ii) there must be only 2 script paths, unroll and sweep:**

1. **UNROLL PATH**



* This is the **Unroll path**, where **both server and user must collaborate** to spend the output.
* This allows a spend only if both parties provide valid signatures.

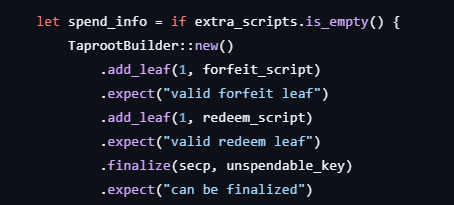
1. **SWEEP PATH**

****

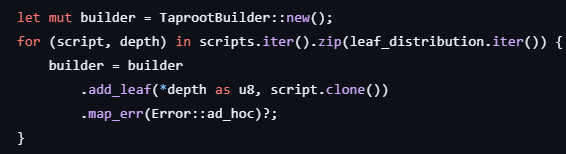
<Delay> CHECKSEQUENCEVERIFY DROP <Server pub\_Key> CHECKSIG

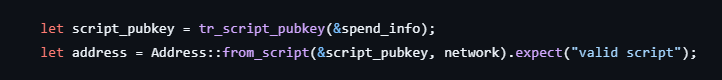
This script allows the server to sweep after exit\_delay time has passed since confirmation.

**Building the Shared output Tree:**

** If no extra scripts:**

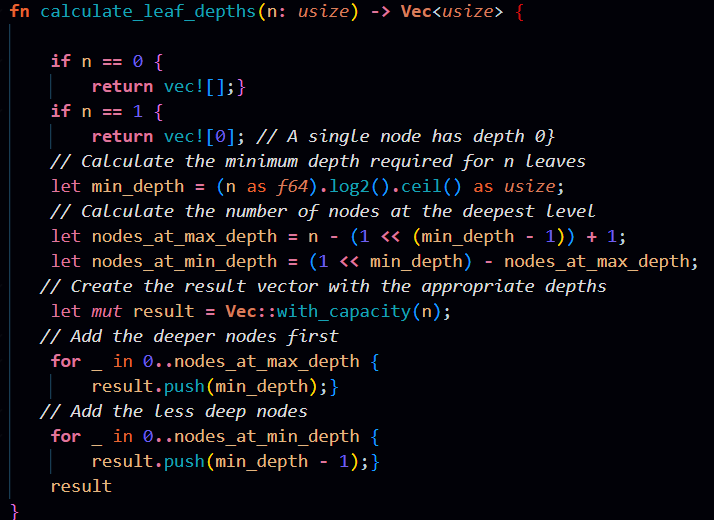
** else :**



Once the shared output built :

This Taproot address is where the **shared output** is sent

**Optimization of VTXO tree**

**Why? ANS** **: To minimize witness sizes if an unilateral exit happens, the tree should be:**

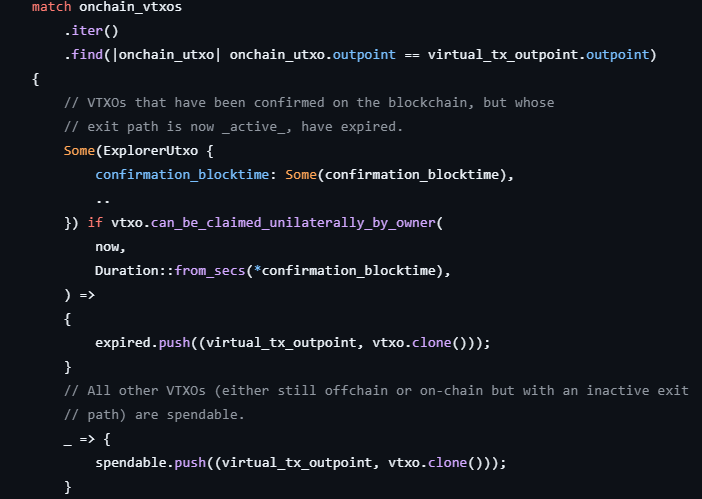
**Balanced as possible**

**Now , we have the VTXOs : we need to classify them**

**CLASIFICATION OF VTXOs :**

**These server classifies VTXOs based on their on-chain status and timelock expiration:**

* **Expired: The unilateral claim condition is met**
* **Spendable: Still within the cooperative use period.**

****

**Whenever the user’s VTXOs close to its expiry date , USER can swap them to new ones to extend their liveness: BY entering into ROUNDS:**

**Preparing Inputs for the Round**

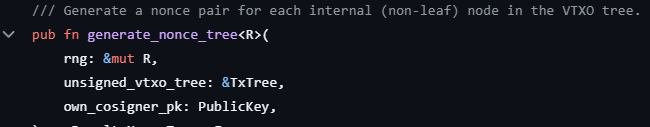
****

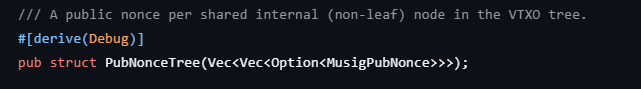
**Server Initiates New Round BY creating two outputs :**

* **Shared Output (SO): Commits to the root of a new VTXO tree**
* **Connector Output (C): Commits to the root of a tree of connectors**

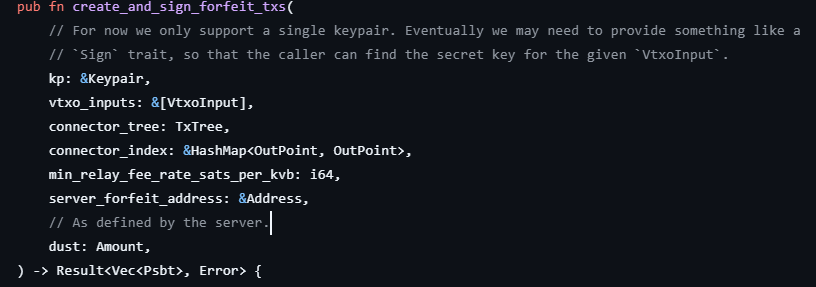
**Generating the Nonce Tree:**

**For each internal node in the VTXO tree, participants generate cryptographic nonces:**

****

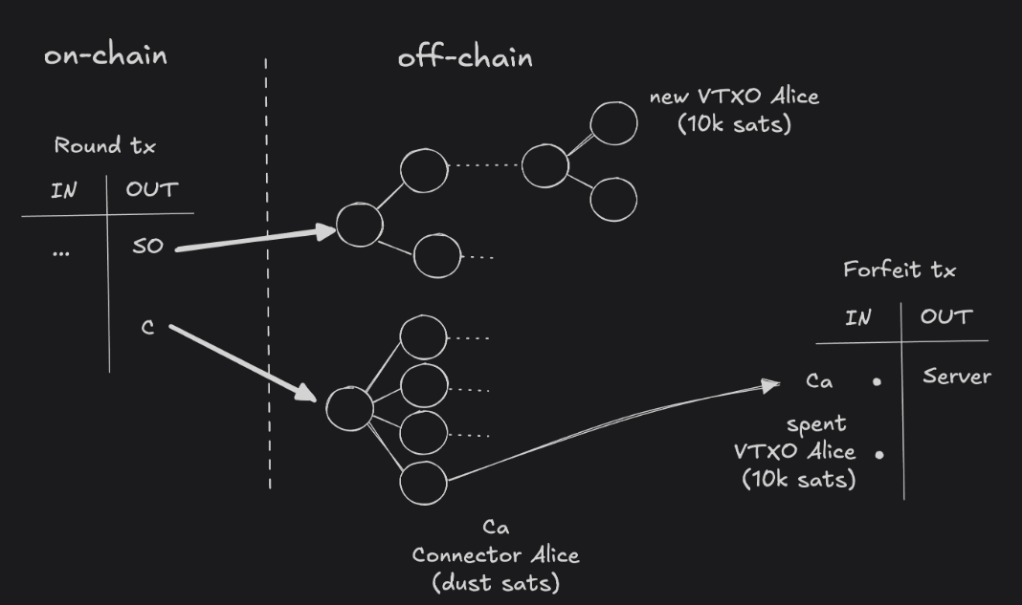
**Convert to public form (public\_nonce\_tree):**

Creating and Signing Forfeit Transactions:

**Users create forfeit transactions that connect their old VTXOs to connectors:**

**The forfeit transaction:**

* **Has two inputs:** 
  + 1. **A connector output (dust sats)**
    2. **The user's spent VTXO (e.g., 10k sats)**

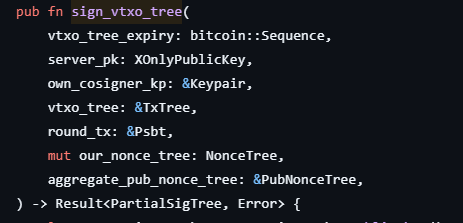
****

The connector is crucial for atomicity. Since the connector output only exists after the round transaction is confirmed on-chain, the forfeit transaction cannot be valid until the round transaction is confirmed. This ensures the user doesn't lose their funds if the round transaction isn't broadcast

Sign each shared internal (non-leaf) node of the VTXO tree with `own\_cosigner\_kp` and using our\_nonce\_tree` to provide our share of each aggregate nonce

Users provide two categories of signatures:

* **MuSig Partial Signatures** for the VTXO tree:
* **Direct PSBT Signatures** for round inputs:



Each internal node in the VTXO tree is a **MuSig aggregate public key**, involving:

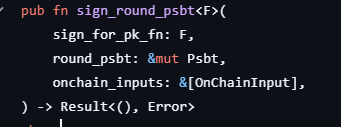
a) server\_pk (Ark server)

b) own\_cosigner\_kp (user's keypair)

Each node is signed using:

* The user's private key
* Their nonce from our\_nonce\_tree
* The aggregated public nonces from aggregate\_pub\_nonce\_tree

**Signing the Round PSBT Inputs:**

* **Sign every input of the `round\_psbt` which is in the provided `onchain\_inputs` list.**

Users sign the inputs that they control in the PSBT:

* onchain\_inputs: These are regular UTXOs (boarding inputs)
* Each UTXO must be signed by its owner (the user)

**Connector Mechanism:**

By introducing **connector** , Ark makes it *cryptographically impossible* for the server to steal or mishandle VTXOs during a round. This is enforced via **forfeit transactions** that are **pre-signed**, but only valid once the round transaction confirms.

The round transaction creates a connector tree through its Connector Output (C)

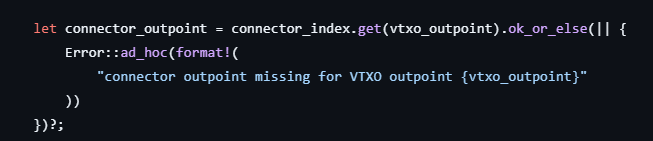
* Each connector is a dust-valued output signed only by the server
* The connector serves as one input to the forfeit transaction
* The spent VTXO serves as the other input to the forfeit transaction
* The forfeit transaction can only be valid if the connector exists on-chain

**Forfeit Transaction:** Each user pre-signs a forfeit transaction:  
 Inputs:

* 1. VTXO spending USER + Server
  2. (Connector from next Round transaction)

This forfeit TX is:

* Submitted along with round participation
* Only valid once the connector is on-chain



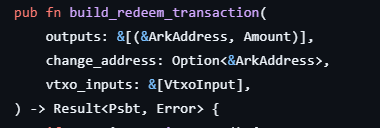
This code ensures:

Every VTXO to be refreshed is **mapped to a connector**

Redeem Transaction:

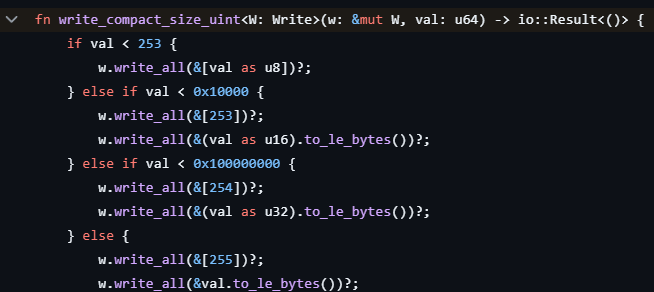
Transactions to be passed between parties off-chain before eventually settling on-chain

Input: VTXO spending  USER+ Server  
  
output : (reciever and Server) or (reciever after 24 hours)   
  
  
**Build\_redeem\_transaction:**

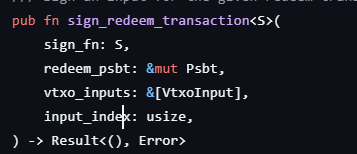


The core function build\_redeem\_transaction creates a transaction that spends VTXOs to send funds to other addresses  
 steps :

* **Create Outputs**: Converts recipient ArkAddresses into Taproot script pubkeys
* Computes the fee using compute\_redeem\_tx\_fee() based on VTXO witness size, number of inputs/outputs
* Builds an unsigned Transaction ( LockTime::ZERO) using the VTXO inputs and outputs
* For each input, inserts a custom PSBT field (under key taptree) encoding taproot tree structure using write\_compact\_size\_uint

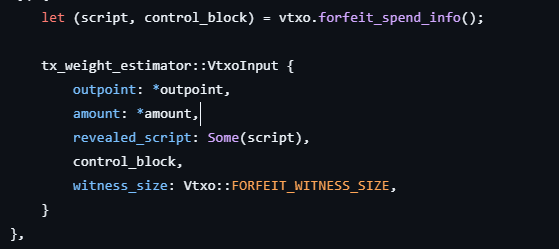


**sign\_redeem\_transaction:**

****

Steps:

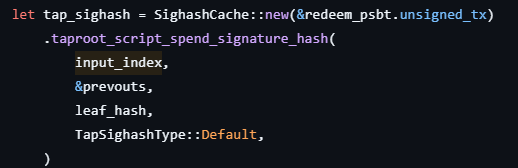
* Fetches the script and control block for the input using vtxo.forfeit\_spend\_info()



* Sets witness\_utxo to the expected input prevout.



* Configures tap\_scripts for script path spend.
* Uses sighashCache::taproot\_script\_spend\_signature\_hash() to calculate a BIP341-compatible digest for signing.



* Calls provided sign\_fn with the secp256k1::Message to obtain a (signature, x\_only\_pk) pair
* Stores signature in tap\_script\_sigs PSBT map keyed by public key and tapleaf hash