Ark: A UTXO-based Transaction Batching Protocol

Marco Argentieri[1], Zeta Avarikioti[2], Andrew Camilleri[1], Pim Keer[2], and Matteo Maffei[2]

[1]Ark Labs
[2]TU Wien

Abstract

Ark is a novel scaling protocol for Bitcoin that enables offchain transaction execution while allowing users to keep full custody of their funds. This is made possible through the introduction of Virtual UTXOs (VTXOs), which allow users to transact offchain while retaining the ability to unilaterally exit to the Bitcoin mainchain. Coordinated by an operator who batches user activity into onchain commitments, Ark achieves high transaction throughput with a minimal onchain footprint, offering a simple and accessible scaling solution for Bitcoin, and can accommodate layer-2 constructions that would be too inefficient or costly to execute on the Bitcoin mainchain.

Contents

1 Introduction

Bitcoin's base layer offers strong security guarantees but limited throughput, making it unsuitable for everyday transactional use at scale. To address this, multiple scaling solutions [8, 6, 9] have been proposed to facilitate offchain transactions with lower latency and reduced fees. There seems to be, however, an inherent trade-off between the solution's ability to move transaction processing offchain, minimising its onchain footprint, providing a favourable user experience, and being fully non-custodial.

For example, the Lightning Network [8] is a network of payment channels, where each channel is essentially an onchain output held by the two channel parties. Many operations in the Lightning Network, and other payment channel constructions [4, 2] for that matter, such as user onboarding and liquidity management, require onchain activity. Moreover, channel participants need to constantly monitor the blockchain in order to protect against a potentially misbehaving counterparty.

Sidechains like Liquid [6] can move arbitrary amounts of execution off the Bitcoin mainchain, but rely on a federation of custodians that hold on to the bitcoins of users operating within the sidechain.

Finally, shared UTXO protocols like CoinPool [5]—apart from requiring Bitcoin protocol updates—require onchain activity for onboarding, as well as all pool participants being available to update the state of the pool.

Ark aims to strike a balance between the different extremes of this trade-off by offering an efficient way to batch many transactions into a single onchain output. This is achieved by having a single coordinating entity—an operator—who aggregates user activity into onchain *commitment transactions*. These transactions compactly encode offchain transactions, user entries and exits, and settlement logic, resulting in a verifiable system with a minimal onchain footprint. This mechanism is, moreover, entirely trustless due to the use of *connectors*. The offchain transactions follow a similar logic as onchain transactions, through the use of a novel abstraction called *virtual UTXOs (VTXOs)*, which represent offchain, spendable balances with unilateral exit options. This design enables flexible and scalable offchain transactions with significantly reduced coordination overhead. In particular, Ark enables the *lifting* of aforementioned constructions to an offchain setting, complementing existing scaling solutions by allowing them to operate with a minimal onchain footprint.

In this litepaper, we present an overview of the Ark protocol, describe its core components and transaction flow, and highlight the design principles that enable efficient transaction batching on Bitcoin. Finally, we explore some of the use cases that Ark enables, and conclude with a number of directions for future research.

1.1 Contributions

This litepaper introduces the Ark protocol and outlines its key design features:

- **Virtual UTXOs (VTXOs):** A novel abstraction enabling offchain transactions while allowing holders to keep custody of their funds at all times.
- **Minimal onchain footprint:** Ark scales by batching user transactions into onchain commitment transactions, reducing the onchain cost to a constant in the optimistic case, regardless of the number of offchain transactions.
- **Unilateral exit guarantees:** Every user can independently exit the protocol and reclaim their funds. The exit cost is logarithmic in the number of offchain transactions in the batch containing the user's funds.

Together, these features position Ark as a simple, scalable, and user-friendly transaction batching protocol that enables the efficient and secure aggregation of multiple offchain transactions into one onchain output, opening the doors to many scaling solutions previously infeasible to run on Bitcoin.

2 Overview

In this section, we give a high-level overview of the Ark protocol and what it enables. Also, we discuss in which model, and under which assumptions, we study the protocol and its security and scalability properties.

2.1 Assumptions

Throughout this text, we operate under the following assumptions.

Cryptographic assumptions. We assume there are secure communication channels between participants, as well as cryptographically secure hash functions, signature schemes and encryption schemes. Moreover, all protocol participants are computationally bounded.

Blockchain assumptions. The Ark protocol runs on top of the Bitcoin blockchain. Users of Bitcoin can submit transactions as inputs to this blockchain, which then outputs an immutable ledger in which the inputted transactions are ordered. We model the Bitcoin blockchain according to the Bitcoin backbone protocol [3].

Network model. We assume a synchronous network. We distinguish between the Ark users and the Ark operator. The users are assumed to come online periodically with period T_o, whereas the operator is assumed to be online at all times.

2.2 Protocol Overview

Ark is a transaction batching protocol for Bitcoin that enables multiple users to transact offchain with a minimal onchain footprint. To facilitate this, at a high level, a user can first *enter the Ark* by spending bitcoin via a specific *boarding transaction*, receiving one or multiple virtual transaction outputs (VTXOs) in return, enabling the user to transact within the system. These offchain transactions, facilitated by the Ark operator, are very similar to onchain transactions, spending VTXOs and creating new VTXOs. Finally, users can *leave the Ark* by redeeming their funds back to the Bitcoin blockchain.

A crucial role in most of these operations is played by the Ark operator, who has three main responsibilities:

- **Liquidity provision**: Supplying the funds needed to operate the Ark by effectively lending capital to users and earning fees in return.
- **Batching**: Collecting requests from current and prospective users to create *commitment transactions*, which include *batches* that facilitate user onboarding, transfers, and exits.
- **Signing**: Collaborating with users to finalise transactions within the Ark, including the signing of *batches*.

We showcase all the components of the Ark protocol in Figure 1. In this example, a user, Alice, boards the Ark, makes a simple offchain transaction to another user, Bob, and later exits the Ark collaboratively with the operator. Some time later, Bob also wishes to exit, but the Ark operator has crashed, requiring Bob to exit unilaterally. Keep in mind that Ark's scaling potential comes from the—in theory—arbitrarily large number of VTXOs that can be contained within a single output (batch) of a commitment transaction. We distinguish the following protocol components:

Running the Ark. The Ark operator can run an Ark by broadcasting so-called *commitment transactions*. These transactions are mostly funded by the Ark operator's own funds and specify, through the use of *batches* as its transaction outputs, a set of VTXOs created within the Ark. We discuss in detail the structure and spending conditions of these transaction outputs in Section 4.4.

Transacting within the Ark. New VTXOs can be created by Ark transactions. This is done by collaborating with the Ark operator to construct an offchain (i.e., virtual) transaction that spends existing VTXOs and outputs new VTXOs. Such Ark transactions are simply Bitcoin transactions, with specific locking scripts on the outputs.

Batch swapping. Users can swap VTXOs from previous batches or created by Ark transactions into fresh VTXOs encapsulated in a batch of a new onchain commitment transaction. To do so, the user will forfeit the current VTXO, meaning that the Ark operator can claim it in case VTXO ever goes onchain and becomes a UTXO. In return, the operator will include a fresh VTXO in the next commitment transaction. In Section 4.3, we describe how both steps can happen atomically through the use of *connectors*.

Boarding the Ark. To board the Ark, a user Alice creates a boarding transaction, which is funded by Alice's own funds, and has one UTXO that can only be spent by Alice together with the Ark operator or alone by Alice after a timelock. As soon as this transaction is included onchain, Alice will contact the Ark operator, who can then include the UTXO to (partially) fund the next commitment transaction. Alice will now have a VTXO, which she can use to transact within the Ark. This VTXO can be turned into an actual UTXO at all times, but will be held offchain as long as necessary.

Leaving the Ark. After a while, Alice wants to retrieve a VTXO from the Ark and turn it into a UTXO. Just as if she would batch swap her VTXO, Alice forfeits her current VTXO, but instead of constructing a new set of VTXOs, the Ark operator will add an additional output to the next commitment transaction that can be spent by Alice only. If the Ark operator is unresponsive whenever, say, Bob wants to exit the Ark, he can still do so by spending from the corresponding output of the commitment transaction his VTXO is in. We elaborate on this in Section 4.5.

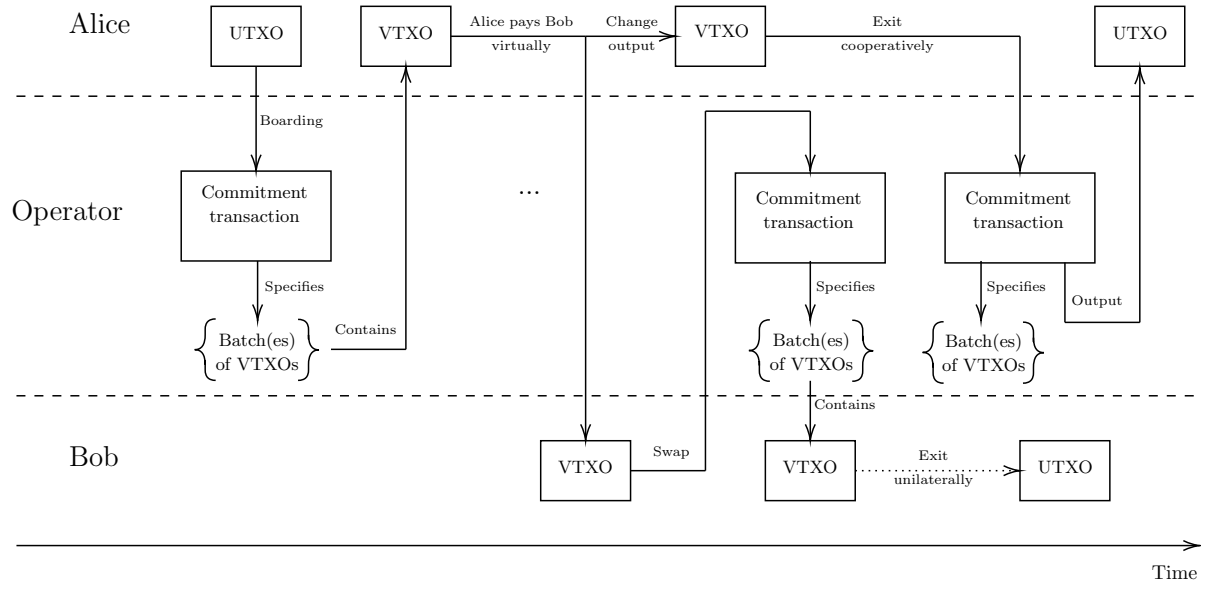


Figure 1: A general flow within the Ark protocol.

2.3 Security and Scalability Properties

As with Bitcoin, Ark transactions achieve finality only upon inclusion in an onchain block. Once a transaction is incorporated into a batch and anchored via a corresponding Commitment Transaction, the Ark protocol can be shown to satisfy a number of security and scalability properties under certain assumptions on the operator's behaviour. In particular, we consider both a *rational* operator, who will only deviate from the protocol if it increases its profits, and a *malicious* operator, who may arbitrarily deviate from the protocol. Below, we list the six properties the Ark protocol satisfies under the assumption of a rational operator.

(NS) **Ark Onramp Safety**: One cannot obtain a VTXO without spending onchain funds through a boarding transaction.

(NL) **Ark Onramp Liveness**: If one spends onchain funds through a boarding transaction, then either a VTXO is obtained, or the onchain funds are returned.

(AS) **Ark Safety**: Ark participants cannot double-spend confirmed VTXOs.

(AL) **Ark Liveness**: Any valid transaction submitted by an Ark user to the Ark operator will be confirmed within some time.

(FS) **Ark Offramp Safety**: One cannot claim Ark funds onchain without forfeiting or publishing onchain the corresponding VTXO.

(FL) **Ark Offramp Liveness**: If one correctly forfeits[1] or publishes[2] an unspent VTXO that has not yet expired, then one will claim the corresponding Ark funds onchain or retain the VTXO until expiry.

Assuming a malicious operator, the Ark protocol only satisfies (NL) and (FL). However, the violations of (NS), (AS) and (FS) will never lead to a loss of funds for Ark users that follow the protocol. This is also summarised in Table 1.

Table 1: Security properties of the Ark protocol under a rational and malicious operator assumption. A star $*$ indicates that although the property is not fulfilled, the violations of the property come only at the cost of the operator, and not of users following the protocol.

Operator	(NS)	(NL)	(AS)	(AL)	(FS)	(FL)
Rational	✓	✓	✓	✓	✓	✓
Malicious	$*$	✓	$*$		$*$	✓

Under the assumption of a rational operator, we could eventually show that the six properties mentioned above imply that the Ark protocol is *balance secure*, as long as $T_o < T_e$, where T_e is the batch expiry set by the operator. That is, any participant who follows the Ark protocol, being a user or an operator, is able to exit the protocol without losing funds (other than fees) that have been confirmed to be theirs.

Regarding scalability, and again, assuming a rational Ark operator, the Ark protocol has a:

- **Constant execution cost**: Executing the Ark protocol has an $\mathcal{O}(1)$ onchain footprint.
- **Constant exit cost (optimistic)**: Assuming that the operator is online and behaves according to the protocol, an Ark user can exit the Ark with $\mathcal{O}(1)$ transactions.
- **Logarithmic exit per-VTXO cost (pessimistic)**: In the worst case, an Ark user can exit a VTXO from the Ark with $\mathcal{O}(\log t)$ transactions, where t is the number of VTXOs contained in the corresponding batch.

3 Background

Already in the previous section, we used the terms UTXO and VTXO several times. By construction, the Ark protocol inherits Bitcoin's UTXO model. In this section, we elaborate on the UTXO model and introduce the necessary notation. We mostly follow the notation of Aumayr et al. [1]. Afterwards, we introduce within this UTXO model several components that are essential to the construction of the Ark protocol.

3.1 The Unspent Transaction Output (UTXO) Model

The Bitcoin blockchain follows an unspent transaction output (UTXO) model. Let us denote the Bitcoin ledger by $\mathcal{L}$, and let U be a user of the ledger. This user has a secret-public key pair (sk_U, pk_U), and can use its secret key to produce, for an arbitrary message $m \in \{0,1\}^*$, a signature $\sigma_U(m)$.

The ledger $\mathcal{L}$ is made up of transactions. In the UTXO model, each transaction `tx` maps a non-empty list of existing unspent transaction outputs, UTXOs for short, to a list of new UTXOs. Such a transaction output is defined as a tuple $\texttt{out} := (\texttt{value}, \texttt{lockScript})$ with $\texttt{out.value} \in 10^{-8}\mathbb{N} \cup \{0\}$[3] the amount of bitcoins (₿) held by `out` and `out.lockScript` the locking script. This locking script specifies the conditions that need to be satisfied in order to use this output in a new transaction to move the coins to a new output. We can then view the old output as an input $\texttt{in} := (\texttt{prevTx}, \texttt{ind}, \texttt{lockScript})$, where `prevTx` is the already existing transaction containing the old output and `ind` the index of the output within `prevTx`. We slightly abuse notation and write `in.value` to denote the value of the corresponding output, i.e., $\texttt{in.value} = \texttt{in}_i\texttt{.prevTx.outputs}[\texttt{in}_i\texttt{.ind}]\texttt{.value}$, where we denote the i-th element of a list `list` by $\texttt{list}[i]$.

[1] That is, first verifies that the commitment transaction that contains the connector also has a UTXO with the correct amount and locking script.

[2] That is, the corresponding virtual transactions.

[3] Keep in mind that the smallest amount of Bitcoin is a satoshi: $1₿ = 10^8$ satoshis. We ignore any technicalities on the maximum value of an output.

Hence, we can formally write a transaction as a tuple $\texttt{tx} := (\texttt{inputs}, \texttt{witnesses}, \texttt{outputs})$. Here, $\texttt{tx.inputs} := [\texttt{in}_1, \ldots, \texttt{in}_n]$ is the list of n transaction inputs, $\texttt{tx.outputs} := [\texttt{in}_1, \ldots, \texttt{in}_m]$ the list of m transaction outputs, and $\texttt{tx.witnesses} := [\texttt{w}_1, \ldots, \texttt{w}_n]$ the list of n witnesses. For $i \in \{1, \ldots, n\}$, $\texttt{w}_i$ is the witness data that serves as an input to the locking script $\texttt{in}_i\texttt{.lockScript}$. Only if the locking script outputs `True`, i.e. $\texttt{in}_i\texttt{.lockScript}(\texttt{w}_i) = \texttt{True}$, do we consider the spending condition fulfilled and can the UTXO $\texttt{in}_i$ be spent. We say that the transaction is *valid* only if every transaction input is unspent, every witness leads to a successful execution of the corresponding locking script, and the sum of coins in the inputs is greater than or equal to the sum of coins in the outputs, i.e., $\sum_{i=1}^{n} \texttt{in}_i\texttt{.value} \geq \sum_{j=1}^{m} \texttt{out}_j\texttt{.value}$.

The locking scripts are expressed in the stack-based Bitcoin scripting language. These scripts can become quite complex, but we can distinguish a number of fundamental building blocks used throughout this paper.

- **Signature locks.** An output can be locked by $\texttt{checkSig}_{pk_U}$, meaning that it can only be spent if the transaction spending this output is signed with the corresponding secret key sk_U.
- **Multi-signature locks.** In order to spend an output locked with a k-of-n multi-signature lock $\texttt{checkMultiSig}_{k,n;pk_{U_1},pk_{U_2},\ldots,pk_{U_k}}$, one needs to provide k out of n signatures for the spending transaction.
- **Timelocks.** Outputs can be locked temporarily. This can either be done until a specified block height T with an absolute timelock $\texttt{absTimelock}(T)$, or after a specified number of blocks t after the transaction is included onchain, using a relative timelock $\texttt{relTimelock}(t)$.
- **Other conditions.** We can denote by `True` and `False` conditions that are always and never fulfilled, respectively. The `False` condition can be used to indicate that coins are *burned*, i.e., rendered unspendable.

Remark 3.1. *Signature locks can be unlocked by providing a signature of the spending transaction with the appropriate secret key. With the use of* `SIGHASH` flags*, one can specify which parts of the transaction should be signed. For example, a* `SIGHASH_ALL` *flag signs all inputs and all outputs, meaning that modifying the transaction in any way would render the signature invalid. However, using a* `SIGHASH_NONE` *flag only signs the transaction inputs, so one can modify the transaction outputs without changing the signature. As we discuss in Section 4.3, the* `SIGHASH_ALL` *flag enables us to create connectors, effectively conditioning the validity of one transaction on another transaction being included onchain.*

We express more complex scripts in terms of these building blocks and logical operators, as well as other expressions available in Bitcoin Script. Finally, since the Taproot protocol upgrade, it is also possible to have a locking script made up solely of a *tweaked* public key, that can either be unlocked by providing a *Schnorr signature* by the appropriate public key, or by satisfying one of the locking scripts that has been committed to in that tweaked public key. This allows an output to have multiple spending conditions, while only revealing one of these spending conditions when the output is spent. This is achieved by bringing together the different spending conditions into one Merkle tree. The Merkle root of this tree is then used together with an internal public key pk_I to create the actual locking script of the output, which will be the *tweaked public key*. The output can then be spent either (i) via the key path (if enabled), providing just a signature corresponding to the tweaked public key, or (ii) via a script path. In the latter case, the output is spent by satisfying just one of the spending conditions in the tree, as well as providing a Merkle inclusion proof of that script in the Merkle tree. We denote a Taproot locking script as $\texttt{Taproot}(pk_I\texttt{;scriptPath}_1\texttt{,...,scriptPath}_n\texttt{)}$, where pk_I is the internal public key. If the key path is unspendable, we replace pk_I by `False`.

As mentioned earlier, Taproot makes use of Schnorr signatures. These have several advantages over Bitcoin's ECDSA signatures. For our purposes, Schnorr signatures allow for the efficient multi-signature scheme *Musig2* [7]. This scheme enables multiple parties, each with their own individual private key, to collaboratively produce signatures corresponding to an aggregate public key. These signatures can only be created if all parties cooperate, effectively yielding a n-of-n multi-signature locking script. For n parties with public keys $pk_1, \ldots, pk_n$, we denote the aggregate public key by $\bigoplus_{i=1}^{n} pk_i$ and a corresponding signature as $\sigma_{\bigoplus_{i=1}^{n} pk_i}$. Furthermore, for a Taproot locking script which checks whether a valid signature under $\bigoplus_{i=1}^{n} pk_i$ is provided, we simply write $\texttt{checkSig}_{\bigoplus_{i=1}^{n} pk_i}$.

3.2 Covenants

Covenants are special locking scripts that allow an output to be spent only in a specific way. A simple example of a covenant would be an n-of-n multi-signature output script, in which the n signers agree to only sign transactions that spend the output in a prearranged way. Note that this covenant relies on at least 1 out of the n signers to stick to the arrangement. New opcodes in Bitcoin Script, such as `OP_CTV` or `OP_CAT`, would allow for stronger covenants that do not require this 1-of-n honesty assumption. However, introducing such new opcodes would require a Bitcoin soft fork. As of the time of writing this text, it does not seem likely any of these proposals will make its way into Bitcoin Script any time soon. Hence, in this paper, we focus on defining the Ark protocol purely with the means that are currently available in Bitcoin Script. If a stronger covenant would eventually make its way into Bitcoin, our formalisation of the Ark protocol can easily accommodate this covenant instead of the n-of-n multi-signature output script. Such a covenant would improve the protocol in terms of non-interactivity and efficiency, but we stress that it is in no way necessary for the correct functioning of Ark.

4 Ark Construction

In this section, we outline the Ark protocol. We consider one Ark operator O with secret-public key pair (sk_O, pk_O), as well as Ark users such as Alice and Bob, with respective secret-public key pairs (sk_A, pk_A) and (sk_B, pk_B).

At the core of the Ark protocol is the use of virtual UTXOs, or VTXOs for short. These are unspent transaction outputs that can be spent without an onchain footprint. We will see in Section 4.2 how this can be done using Ark transactions. To ensure the proper functionality, a VTXO is a transaction output with a very specific locking script. This locking script is a Taproot script, with at least one *collaborative* and one *unilateral* script path. As we will see, this allows the owner of a VTXO to either spend the VTXO in collaboration with the Ark operator or spend it without the Ark operator.

Definition 4.1 (Virtual UTXO / VTXO)**.** *A* virtual UTXO*, or* VTXO*, held by the VTXO holder with an Ark operator O, is an unspent transaction output* $\texttt{vtxo} := (\texttt{value}, \texttt{vtxoLockScript})$ *where the locking script* `vtxoLockScript` *is a Taproot locking script such that*

- *the key path is unspendable,*
- *there is at least one* collaborative *script path, i.e., a script path that*
  - *requires the signatures of both the VTXO holder and O, and*
  - *can be delayed by an absolute timelock,*
- *there is at least one* unilateral *script path, i.e., a script path that*
  - *does not require the signature of O,*
  - *must be delayed by a relative timelock t_v which is at least as long as a minimum delay determined by O.*

We say that a VTXO is spent collaboratively*, if it is spent by satisfying a collaborative script path (giving a corresponding witness w_{collab}). Similarly, a VTXO is spent* unilaterally*, if it is spent by satisfying a unilateral script path (giving a corresponding witness w_{unilat}).*

The simplest example of a VTXO would be the equivalent of a *single-signature* output, i.e., an amount that the VTXO holder Alice can spend by providing a signature. The locking script of this VTXO with operator O would then be:

$$\texttt{vtxoLockScript} = \texttt{Taproot}\big(\texttt{False}; \texttt{checkSig}_{pk_O \oplus pk_A}, \texttt{checkSig}_{pk_A} \wedge \texttt{relTimelock}(t_v)\big).$$

In this case, the collaborative witness would be $w_{collab} = \sigma_{O \oplus A}$, and the unilateral witness would be $w_{unilat} = \sigma_A$.

We can now proceed with describing Ark's core functionality: transaction batching, allowing us to reduce many VTXOs to one single onchain output, called a *batch*. Next, we explain how Ark users can transact offchain with these VTXOs, spending existing VTXOs and creating new VTXOs in the process. Through *batch swapping*, these newly created VTXOs can be swapped atomically for fresh VTXOs that are part of a new batch. This batch swapping will motivate the structure of so-called commitment transactions, which will be the only onchain footprint. We then add to that structure to allow users to board and exit the Ark. Finally, we formalise the protocol's operations step-by-step.

4.1 Transaction Batching

As indicated by the name, VTXOs are intended to be virtual. Although their construction allows them to be turned into a UTXO at any point in time via the unilateral exit path, this will optimistically never happen. If a VTXO holder wishes to claim the funds in the VTXO onchain, the preferred way to do so will be by collaborating with the operator, as we describe in Section 4.5. Based on this insight, we can devise a structure that encapsulates multiple VTXOs with a minimal onchain footprint, while still allowing—in the worst case—every VTXO holder to exit unilaterally. This structure is called a *batch*. It is simply a transaction output created and funded by the operator, that can only be spent in two ways, either through a *sweep* path that allows the operator to claim back the entire output, or through an *unroll* path that splits the single output into the constituent VTXOs according to a so-called *virtual transaction tree (VTXT)*. A virtual transaction is just a regular Bitcoin transaction that will optimistically never go onchain.

Definition 4.2 (Virtual transaction tree)**.** *A* virtual transaction tree (VTXT) *is a directed rooted tree, given by the ordered pair $G = (V, A)$, where V is a set of virtual transactions (the* nodes*), and A is a set of ordered pairs of virtual transactions (the* edges*), such that, for every $v \in V$ and every $i \in v.\texttt{inputs}$, there exists an edge $(u, v) \in A$ such that $u = i.\texttt{prevTx}$. There is moreover exactly one virtual transaction $r \in V$ such that there are no edges $a \in A$ of the form (u, r), where $u \in V$. This r is called the* root*. Any $\ell \in V$ for which there are no edges of the form (ℓ, u) in A (for $u \in V$) are called* leaves*.*

Definition 4.3 (Batch)**.** *A* batch *is a transaction output which is locked by a taproot script* `batchScript` *with an unspendable key path and exactly two script paths:*

- *a* sweep *path that allows the Ark operator to claim the entire output after a time T_e, which we call the* batch expiry*, and*
- *an* unroll *path that specifies spending according to a VTXT with root spending the full batch, where each leaf of the VTXT has a VTXO as its only output, and where the remaining nodes of the VTXT are virtual transactions that have batches as their only outputs.*

Remark 4.4. *A batch essentially allows multiple VTXO holders to specify how to distribute the value of the batch amongst them.*

Remark 4.5. *The structure of the VTXT is enforced by a covenant, ensuring that the batch can* only *be spent according to the VTXT. When we emulate the covenant by using an n-of-n multi-signature lock (using for example Musig2), the operator needs to coordinate a signing session with all involved VTXO holders, producing for each node in the VTXT the appropriate signature. Multiple design choices could be made here. One option could be having all involved VTXO holders sign every virtual transaction. Alternatively, we could have each VTXO holder sign only those transactions on a path to their respective VTXOs, as shown in Figure 2. This approach decreases the interactivity requirement, as fewer signers are needed, without sacrificing safety when assuming rational signers. Indeed, for a virtual transaction $\texttt{vtx}$ spending an output with value $\sum_{i=1}^{n} v_i$, the sub-VTXT starting from $\texttt{vtx}$ will specify how to divide the output into VTXOs with values $(v_i)_{i=1}^{n}$, held by their respective holders $\{1, \dots, n\}$ (who we assume to be distinct for simplicity). Any alternative distribution of funds $(v'_i)_{i=1}^{n}$ implied by another sub-VTXT will necessarily have $v'_j < v_j$ for some $j \in \{1, \dots, n\}$. This (rational) signer j will not agree to sign the transactions in this alternative sub-VTXT. Hence, besides the operator, we do not require any signer outside of $\{1, \dots, n\}$, i.e., any other VTXO holder that does not have $\texttt{vtx}$ on its path from the batch to its VTXO.*

The purpose of the batch expiry will become clear in Section 4.4. For now, it is important to note that every VTXO holder who knows the path of virtual transactions in the VTXT that spends from the original batch to eventually create the holder's VTXO can broadcast this path onchain. In the—hopefully rare—event of a unilateral exit, the burden of exiting falls on the VTXO holder who wants to exit, as this user needs to broadcast the appropriate virtual transactions onchain. An obvious example of a VTXT would be a binary tree, as the one that can be found in Figure 2. In that setting, VTXO holder U_1 would have to broadcast the transactions with output values $(v_1 + v_2, v_3 + v_4)$, (v_1, v_2) and v_1 to exit unilaterally. Notice in particular that the user must pay the corresponding onchain transaction fees. This *exit cost* may prove to be a significant portion of the VTXO value.

Remark 4.6. *Realise that the VTXT structure allows a VTXO holder to exit unilaterally,* without *having any other VTXOs ending up onchain. This will be of key importance to batch swap VTXOs.*

Remark 4.7. *The VTXT is not limited to the aforementioned binary tree. Many structures are possible, all with their own advantages and disadvantages regarding exit cost, storage requirements and interactivity.*

4.2 Ark transactions

As discussed in the previous section, VTXOs are intended to remain offchain to allow for the offchain execution of transactions. This is done via *Ark transactions*, which spend one or multiple VTXOs encapsulated in a batch through the collaborative spending path, creating one or multiple new VTXOs.

Let us describe how the Ark transaction mechanism works through a basic example. Consider an Ark with operator O, in which an Ark user Alice wants to send an amount p over to Bob. Alice holds a VTXO $\texttt{vtxo}_A$ with O in a batch. This VTXO has a value $a > p$. Alice will construct an Ark transaction. In Transaction 1, we show a general Ark transaction.

Transaction 1: A general Ark transaction.

Inputs	Witnesses	Outputs
$\texttt{vtxo}_{i,1}$	w^1_{collab}	$\texttt{vtxo}_{o,1}$
$\texttt{vtxo}_{i,2}$	w^2_{collab}	$\texttt{vtxo}_{o,2}$
$\vdots$	$\vdots$	$\vdots$
$\texttt{vtxo}_{i,n}$	w^n_{collab}	$\texttt{vtxo}_{o,m}$

For our simple example, the Ark transaction would be Transaction 1 with $n = 1$, $\texttt{vtxo}_{1,1} = \texttt{vtxo}_A$, $w^1_{collab} = \sigma_{O \oplus A}$, and possibly three output VTXOs, a VTXO $\texttt{vtxo}_B$ with amount p spendable by Bob, a change VTXO $\texttt{vtxo}_{A'}$ returning the change a' back to Alice, as well as a third VTXO paying a fee f to the operator.

Alice unlocks the input in Transaction 1 using the collaborative path, signing the Ark transaction and requesting O to sign as well. Once O signed too, the transaction is completed and sent to Bob.

Informally, we could say that Bob now holds a VTXO of his own, $\texttt{vtxo}_B$. Observe that Bob is not required to hold any prior funds in the Ark or on the Bitcoin blockchain to receive this VTXO. However, one may argue in two ways that Bob does not really hold a VTXO $\texttt{vtxo}_B$ yet.

First, Bob cannot turn $\texttt{vtxo}_B$ into a UTXO via the VTXO's unilateral exit path whenever he wants. The only thing he could do is to broadcast the Ark transaction he received from Alice onchain, as soon as the input of Transaction 1 would appear onchain. This would be the case if Alice were to exit unilaterally with $\texttt{vtxo}_A$. Either way, Bob must constantly monitor the blockchain to prevent Alice from claiming the funds from $\texttt{vtxo}_A$. We propose a solution to this. Recall that Alice being able to exit unilaterally amounts to her holding several virtual transactions in the VTXT of the batch $\texttt{vtxo}_A$ is a part of. If Alice were to send all these transactions over to Bob, Bob would now also be able to exit unilaterally by posting all these transactions to put $\texttt{vtxo}_A$ onchain, as well as the subsequent Ark transaction to obtain $\texttt{vtxo}_B$.

However, a second, perhaps bigger problem remains even with this patch. Bob has to trust Alice and the operator not to collude and sign another Ark transaction double-spending $\texttt{vtxo}_A$. Since the Ark transaction is virtual, Alice and a rational operator could theoretically spend the same VTXO arbitrarily many times. If Alice were to take $\texttt{vtxo}_A$ onchain, only one of those Ark transactions could be included onchain. In the next section, we will describe how both problems can be solved by batch swapping $\texttt{vtxo}_B$ for a fresh VTXO that will be part of a batch confirmed onchain.

4.3 Batch Swaps

To finalise the Ark transaction from Alice to Bob, Bob can swap $\texttt{vtxo}_B$ *atomically* for a new VTXO. We call this *batch swapping* a VTXO, as a user is swapping a VTXO part of one batch for a new VTXO in a future batch. In our previous setting, batch swapping $\texttt{vtxo}_B$ would mean that Bob does not need to wait for Alice to unilaterally exit with $\texttt{vtxo}_A$ to post the Ark transaction he received and turn $\texttt{vtxo}_B$ into a UTXO with value b, and that he does not have to trust the operator not to collude with Alice. To enable this, Bob's new VTXO, which we will refer to as $\texttt{vtxo}_{B'}$, should also be part of a *batch*. This batch will serve as an onchain confirmation that Bob indeed owns a virtual output with value b.

As mentioned earlier, batch swapping a VTXO is essentially an atomic swap. Bob will give $\texttt{vtxo}_B$ to the operator, and in return, the operator will give $\texttt{vtxo}_{B'}$ to Bob. This happens in such a way that

either both transfers succeed or both transfers fail.

A batch swap begins with Bob sending a request to the operator to batch swap $\texttt{vtxo}_B$. The operator will construct a transaction like the one in Transaction 2, which, as we will see in Section 4.4, is a very simple commitment transaction. This transaction spends some of the operator's own onchain funds, and has two outputs. One with value b', requiring a signature from both Bob and the operator, and one *anchor output* that can be spent by the operator. This output has a dust value ε, and is crucial to ensure the atomicity of the batch swap.

Transaction 2: A very simple commitment transaction.

Inputs	Witnesses	Outputs
$\texttt{out}_O$	σ_O^C	$(b', \texttt{checkMultiSig}_{2,2;pk_O,pk_B})$
		$(\varepsilon, \texttt{checkSig}_{pk_O})$

The operator also creates a virtual transaction that spends from the first output in Transaction 2 and has as output $\texttt{vtxo}_{B'}$:

Transaction 3: Virtual transaction spending from the commitment transaction in 2.

Inputs	Witnesses	Outputs
$(b', \texttt{checkMultiSig}_{2,2;pk_O,pk_B})$	$[\sigma'_B, \sigma'_O]$	$\texttt{vtxo}_{B'}$

The operator signs this virtual transaction and shares it with Bob, together with the anchor output. Bob can now construct a *forfeit transaction*:

Transaction 4: Forfeit transaction.

Inputs	Witnesses	Outputs
$\texttt{vtxo}_B$	w_{collab}^B	$(b, \texttt{checkSig}_{pk_O})$
$(\varepsilon, \texttt{checkSig}_{pk_O})$	σ_O^F	

This is essentially an Ark transaction that spends $\texttt{vtxo}_B$ and gives b to the server, conditional on Transaction 2 being included onchain, i.e., $\texttt{vtxo}_{B'}$ being turned into a UTXO. Bob signs Transaction 4 and sends it to the operator. The corresponding `SIGHASH` flag is `SIGHASH_ALL`. Bob's signature is only valid for this specific forfeit transaction, containing the anchor output. In other words, the forfeit transaction is only valid if Transaction 2 is included onchain.

The operator can now safely broadcast Transaction 2, as the anchor output enables atomicity. Indeed, if Bob somehow is able to turn $\texttt{vtxo}_B$ into a UTXO, the operator can broadcast the forfeit transaction and claim b. On the other hand, the operator is unable to claim $\texttt{vtxo}_B$ using the forfeit transaction as long as Transaction 2 is not included onchain.

Bob has successfully swapped $\texttt{vtxo}_B$ for $\texttt{vtxo}_{B'}$, which he can now unilaterally exit with by posting Transaction 3. Bob no longer needs to monitor the blockchain in case Alice decides to turn $\texttt{vtxo}_A$ into a UTXO. If this happens, it is now the operator's responsibility to broadcast the Ark transaction creating $\texttt{vtxo}_B$, as well as the forfeit transaction to claim back what are now his funds. Bob also no longer runs the risk of Alice double-spending $\texttt{vtxo}_A$. Again, this is because the operator will now no longer agree to double-spend $\texttt{vtxo}_A$, since it is the operator who eventually loses out because of that.

This construction understandably seems cumbersome. However, it is not limited to dealing with just one VTXO. Indeed, the first output of Transaction 2 is just a batch containing one VTXO (where the VTXT consists only of Transaction 3). We can increase the size of this batch, including other VTXOs that may also have been created through batch swaps.

As a final side note, the current construction also locks the funds the operator puts in, as a signature from Bob is also required. We will see in the next section how these funds will eventually be unlocked to be claimed by the operator again.

4.4 Commitment transactions

Transaction 2 offers a natural way to contain batches as its outputs. Moreover, the second output of Transaction 2 enables to swap VTXOs atomically. This second output takes on the role of a so-called *connector*.

Definition 4.8 (Connector)**.** *A* connector *is a transaction output which is locked by a taproot script* `connectorScript` *with an unspendable key path and a script path that specifies spending according to a VTXT where the root spends the full connector, and where each leaf of the VTXT has an anchor output as its only output. The remaining nodes of the VTXT are virtual transactions that have connectors as their only outputs.*

In the Ark protocol, a connector thus encapsulates all the anchor outputs serving as inputs to forfeit transactions that can only be included onchain if the commitment transaction containing that connector is included onchain. Unlike a batch, the virtual transactions in a connector are only signed by the operator. We can now formally introduce the *commitment transaction*.

Definition 4.9 (Commitment transaction)**.** *A* commitment transaction *is a transaction broadcast by the Ark operator with at least one batch and one connector as outputs.*

Transaction 5: Commitment transaction.

Inputs	Witnesses	Outputs
$\texttt{out}_O$	σ_O^C	$(b, \texttt{batchScript})$
		$(\varepsilon', \texttt{connectorScript})$

A commitment transaction may contain multiple batches and connectors. Additionally, the commitment transaction may contain inputs and outputs related to users joining, or *boarding*, and leaving the Ark. We will specify these procedures and how they may alter the commitment transaction in the next section.

4.5 Boarding and leaving the Ark

Boarding. Consider a user Alice with some onchain funds locked in a UTXO $\texttt{out}_A$. Alice can join, or *board* the Ark via a two-step process. First of all, Alice will construct and broadcast the *boarding transaction*

Transaction 6: Boarding transaction.

Inputs	Witnesses	Outputs
$\texttt{out}_A$	σ_A	$\texttt{out}'_A$

where the locking script of the output $\texttt{out}'_A$ is given by

$$\texttt{Taproot}\big(\texttt{False}; \texttt{checkSig}_{pk_O \oplus pk_A}, \texttt{checkSig}_{pk_A} \wedge \texttt{relTimelock}(t_b)\big)$$

That is, a Taproot script with an unspendable key path and two script paths. The first script path is an *exit* path, which allows Alice to unlock her funds after a timeout period t_b, and the second is a *cooperative* path, where Alice and the operator can spend the funds together.

Alice can now send a boarding request to the operator, who will first verify that $\texttt{out}'_A$ cannot be spent by Alice alone, and then create a VTXO $\texttt{vtxo}_A$ for Alice in a batch of the next commitment transaction. In return, the UTXO $\texttt{out}'_A$ is added as an input to this commitment transaction via the cooperative path of $\texttt{out}'_A$.

In case Alice would not want to board the Ark after all, she can either spend $\texttt{out}'_A$ via the exit path after the timeout, or cooperate with the operator to spend $\texttt{out}'_A$ for a new UTXO that can be spent by Alice only.

Leaving. Alice now holds a VTXO $\texttt{vtxo}_A$, which is part of a confirmed commitment transaction batch. Therefore, she is able to leave the Ark in two ways: *unilaterally* and *collaboratively*. The former does not require any interaction with the Ark operator, and boils down to Alice broadcasting each virtual transaction in the VTXT that lies on a path from the batch root to the batch leaf with output $\texttt{vtxo}_A$.

Alternatively, Alice can leave by collaborating with the operator. To this end, Alice will batch swap $\texttt{vtxo}_A$, but instead of receiving another VTXO in the next commitment transaction, the server will add an extra output to the commitment transaction that can be spent directly by Alice. Apart from that, the batch swap will proceed just as in Section 4.3, with the operator receiving a signed forfeit transaction, in case Alice tries to exit with $\texttt{vtxo}_A$ unilaterally.

4.6 Protocol operations

We can formally summarise the operations performed as part of the Ark protocol in Sections 4.2 to 4.5 as Operations 1 and 2. The former describes how a user Alice can make an Ark transaction to Bob, and the latter bundles all the steps needed to construct a valid commitment transaction. Multiple users will be involved in Operation 2, each making one or more requests. There are three types of requests:

- Boarding requests: A user U owns a UTXO and shares with the operator a confirmed boarding transaction `boarding_tx`, with the aim to swap the UTXO for a VTXO.
- Leaving requests: A user U holds a VTXO and specifies to the operator that he wants to exit that particular VTXO, effectively swapping it for a UTXO.
- Batch swap requests: A user U holds a VTXO and specifies to the operator that he wants to batch swap that particular VTXO for a new VTXO.

We denote by B, L, S the sets of boarding, leaving, and batch swap requests, respectively. An arbitrary request is denoted by r, and we denote by $u(r)$ the user that submitted request r.

Given the requests B, L, S the operator will construct a commitment transaction `commitment_tx`, which has as inputs the outputs of the boarding transactions $r \in B$, as well as an additional UTXO $\texttt{utxo}_O$ coming from the operator to cover the rest of the required funds. The outputs of `commitment_tx` will be a batch and a connector, which we abstract by $\beta := \beta(B \cup S)$ and $\gamma := \gamma(L \cup S)$, respectively. The batch β will implicitly define a VTXT (V, A). We moreover specify a function $\varsigma : V \to \mathcal{P}(u(B) \cup u(S) \cup \{O\})$[4], which tells us for each virtual transaction in the VTXT which subset of $u(B) \cup u(S) \cup \{O\}$ should sign that virtual transaction. This function is uniquely determined by the design choice discussed in Remark 4.5. We will consider a signing session as a black box, indicating which virtual transactions and which parties are involved, and assuming that at the end of the signing session, every involved party has a valid signature that allows that party to broadcast that transaction onchain. Finally, the connector γ will also implicitly define a VTXT, which will have $|L| + |S|$ leaves, defining an anchor output ε_r for each $r \in L \cup S$.

Running Operation 2 will lead to a commitment transaction being confirmed onchain, if everyone involved is responsive[5]. To summarise, we show all the onchain and virtual transactions and their dependencies together in Figure 2, for a very simple scenario similar to the one described in Figure 1. U_1 boards the Ark via a boarding transaction. The operator then uses these funds, together with some of its own funds, to fund the next commitment transaction. U_4 made a request to swap a VTXO he received as output from an Ark transaction, and U_5 made a request to leave the Ark. Hence, the commitment transaction will contain a batch, which includes amongst others a VTXO for U_1 and a fresh VTXO for U_4, an output that can be spent by U_5 immediately, and a connector, enabling amongst others the operator to claim the old VTXOs of U_4 and U_5, would these ever go onchain. Keep in mind that an actual commitment transaction will most likely contain (multiple) batches with more VTXOs each.

5 Security and Scalability Considerations

Having described the protocol in detail, we can now informally argue under which conditions the Ark protocol satisfies the security and scalability properties introduced in Section 2.3.

(NS) **Ark Onramp Safety**: A rational operator would never include a VTXO in a commitment transaction which does not spend the output from a corresponding boarding transaction, as this would amount to the operator funding this VTXO himself and effectively giving away the corresponding funds to the VTXO holder. A malicious operator could do this and would be indifferent about giving away the funds to the corresponding user, but the user could never be at a loss.

(NL) **Ark Onramp Liveness**: This follows immediately from the construction of the boarding transaction, regardless of whether the operator is rational or malicious.

(AS) **Ark Safety**: Recall that Ark transactions are only considered final once the commitment transaction containing the outputs of that Ark transaction is confirmed onchain. This commitment

[4] We denote by $u(B)$ the set $\{u(r) : r \in B\}$ of all users who made a boarding request (similar notation for L and S).

[5] In practice, we would assume that everyone making a request is online and will respond. If this is not the case, Operation 2 will fail. It can be repeated, now excluding the request(s) from the user who did not respond.

transaction is funded by the operator. Hence, a double-spend would imply the operator funding two times the transaction amount, while only receiving one time the transaction amount through expiry of the VTXO that has been spent. A rational operator would thus never agree to double-spend, as this would come at a loss for him. A malicious operator might agree, but would once again be the only one at a loss.

(AL) **Ark Liveness**: A rational operator needs to be incentivised to run an Ark, which can be done through Ark transaction fees, and to run it honestly, i.e., without censoring transactions. For now, the only argument we can give is that if an operator were to censor transactions, users would stop using that Ark and go elsewhere, meaning the operator would lose out on future transaction fees. It is a topic for future research to formalise and quantify these incentives.

(FS) **Ark Offramp Safety**: In the case of a unilateral exit, the user must publish the corresponding chain of virtual transactions (ending in the VTXO) onchain. For a collaborative exit, it would not be rational for the operator to add a UTXO for the exiting user without making sure the old VTXO is forfeited, as the exiting user could now also claim the old VTXO, entailing a loss for the operator. Once again, a malicious operator would not care about this loss, but the users are never at risk of losing funds.

(FL) **Ark Offramp Liveness**: Each VTXO is part of a batch in a commitment transaction. The user can exit unilaterally and be sure that this VTXO cannot be double-spent, as the user signed off on the commitment transaction. In the case of a collaborative exit, because the user correctly forfeited the old VTXO for a specific commitment transaction, the operator has no way to somehow claim both the old VTXO and the new UTXO. If an operator broadcasts this specific commitment transaction, the user will claim the funds onchain. If not, the user will retain the original VTXO until it expires. This holds true for both a rational and malicious operator. Of course, it is preferable that an operator does actually broadcast the commitment transaction, as the VTXO will eventually expire, forcing the VTXO holder to either exit unilaterally (which can be done regardless of the operator's actions) or to lose the funds. In practice, a similar argument as for (AL) could be made to show that a rational operator will, in fact, be incentivised to broadcast the specific commitment transaction. Rational operators do not want to lose out on transaction fees by users leaving because of commitment transactions not being broadcast. Once again, this argument requires a more formal treatment, outside of the scope of this litepaper.

The balance security follows from the above properties, as long as $T_o < T_e$. Recall that T_o was the period with which we assumed the users to come online. From the perspective of the user, as long as a VTXO is confirmed in a batch, the user will come online before the batch expires, and is then able to exit with that VTXO, either unilaterally or collaboratively. From the operator's perspective, the above properties, together with the feature that all batches will expire after T_e, ensure that an operator will eventually regain full custody of his funds.

As for the scalability properties, we note the following.

- **Constant execution cost**: Offchain transactions are finalised with the appropriate batch swaps. The only on-chain footprint is consequently a commitment transaction containing the corresponding batch.
- **Constant exit cost (optimistic)**: If the operator is online, the exiting user can perform a collaborative exit to obtain a UTXO in the next commitment transaction.
- **Logarithmic exit per-VTXO cost (pessimistic)**: In the worst case, an Ark user will exit a VTXO unilaterally, which means this user needs to broadcast virtual transactions onchain, spending the batch output of the corresponding commitment transaction, to eventually claim the VTXO onchain. Assuming a binary tree-like structure for the VTXT specified by the batch output, containing t VTXOs, there will be $\mathcal{O}(\log t)$ virtual transactions that need to be broadcast.

6 Applications

We briefly touch on some potential applications of Ark's transaction batching. The most straightforward application would be to perform offchain payments with no prior setup requirements for the receiver. In Section 4.2, we already described how to transact within an Ark by considering a simple payment

from Alice to Bob. In order to receive funds, Bob only needed to receive the Ark transaction from Alice and batch swap the newly created VTXO with the operator. This is in contrast to, for example, the Lightning Network, where to receive funds, one must have set up a payment channel beforehand, incurring an onchain cost. Bob's wallet only needs to be Ark-aware to receive VTXOs and subsequently batch swap them. From this point onward, he is capable of everything any other VTXO holder is capable of.

In general, applications of Ark leverage the fact that virtual transactions are essentially offchain Bitcoin transactions, and thus also employ Bitcoin Script to formulate spending conditions on VTXOs. With some extra caution, this enables more complex constructions, such as escrows, discreet log contracts, or unidirectional payment channels [4], to be built on Ark. This caution concerns primarily the requirement to properly time any timelocks present in the construction with regards to the batch expiry time.

7 Discussion and Limitations

Centralisation of Ark Operator. The Ark protocol fundamentally relies on a single operator to coordinate offchain transactions, manage liquidity, and produce commitment transactions. This introduces a centralisation vector that may be undesirable in the context of Bitcoin's decentralised ethos. While the protocol ensures that users can unilaterally exit the system and retrieve their funds, the reliance on a central party to enable efficient offchain transfers means that the operator remains a potential single point of failure. If the operator becomes unavailable or behaves maliciously (e.g., by censoring users), users may suffer from degraded performance or increased costs associated with unilateral exits. Future work could explore designs involving multiple operators, either in a federated setting or through cryptographic coordination, to mitigate this centralisation risk. It could also investigate to what extent the responsibilities of the operator—liquidity providing, signing, and batching—could be split across different entities.

Preconfirmation. If an Ark user were to trust the operator not to collude with previous VTXO owners to double-spend, that Ark user would theoretically not need to batch swap the VTXO he received from an Ark transaction. Instead, he could spend this VTXO instantly in a new Ark transaction. Repeating this would lead to a directed acyclic graph (DAG) of Ark transactions, with batch swaps occurring only at the leaves. We sketch in more detail how this would look in Appendix A. Although rational assumptions and settlement mechanisms reduce the scope for abuse, this "instant payment"-model inherently trades off security for performance in the preconfirmation variant. Future research might explore cryptographic means (e.g., succinct fraud proofs or threshold attestations) to reduce the required trust in this preconfirmation variant of the Ark protocol. Ideally, this future research will enable the preconfirmation variant to satisfy the security properties of Section 2.3 while assuming a rational operator.

Liquidity Requirements. The operator must provide a substantial amount of upfront capital to maintain liquidity within the Ark system. This requirement may pose a barrier to entry and raises questions regarding capital efficiency and long-term sustainability. Furthermore, the operator bears the financial risk of fronting liquidity for users' benefit, potentially without immediate compensation. The protocol partially addresses this via transaction fees and the ability to sweep expired VTXOs. Nonetheless, more work is needed to quantify liquidity requirements under realistic usage patterns and to design incentive mechanisms (e.g., fee markets, operator bonding) that attract sufficient capital while discouraging misbehaviour.

Onchain cost of unilateral exit. In case of a unilateral exit, a user is responsible for broadcasting the appropriate virtual transactions to claim his VTXO onchain. The user will consequently have to pay onchain transaction fees, which may add up to a significant portion of the VTXO value the user is trying to claim onchain. As a result, holders of smaller VTXOs may be *priced out* of exiting unilaterally. Future work could study different VTXT designs that may minimise these exit costs for certain users. For example, one might try—if possible—to put smaller VTXOs not too deep in the VTXT, or having a small VTXO share most of its path with a larger VTXO such that in case both exit the holder of the large VTXO can cover most of the exit cost.

Bank Run Scenario. Related to the previous point, a worst-case limitation arises in the case of a bank run, where many users attempt to exit the Ark simultaneously. In such a scenario, the protocol mandates that users publish their corresponding virtual transactions onchain, resulting in a surge of transactions proportional to the number VTXOs in the considered batch. Given Bitcoin's limited throughput, such a spike may cause significant congestion and delay exits, undermining user confidence. Although the protocol ensures eventual exits (i.e., liveness), the associated costs and delays may be substantial. Techniques such as rate-limited exits, time-staggered batching, or offchain exit markets (where users can sell exit rights) might alleviate this issue and constitute promising directions for future work.

8 Conclusion

Ark is a transaction batching protocol that enables executing multiple Bitcoin transactions offchain and batching them together in a single onchain transaction output. This is achieved by introducing virtual UTXOs (VTXOs), a novel abstraction that allows users to transact offchain while retaining the ability to unilaterally settle onchain, and connectors, ensuring atomicity while swapping said VTXOs. As a result, Ark enables onchain constructions, such as payment channels and escrows, to be lifted offchain, marking a new chapter in Bitcoin scaling solutions.

References

[1] Lukas Aumayr, Zeta Avarikioti, Robin Linus, Matteo Maffei, Andrea Pelosi, Christos Stefo, and Alexei Zamyatin. BitVM: Quasi-turing complete computation on bitcoin. Cryptology ePrint Archive, Paper 2024/1995, 2024.

[2] Christian Decker and Roger Wattenhofer. A fast and scalable payment network with bitcoin duplex micropayment channels. In *Stabilization, Safety, and Security of Distributed Systems: 17th International Symposium, SSS 2015, Edmonton, AB, Canada, August 18-21, 2015, Proceedings 17*, pages 3–18. Springer, 2015.

[3] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. *Journal of the ACM*, 71(4):1–49, 2024.

[4] Mike Hearn and Jeremy Spillman. Contract. `https://en.bitcoin.it/wiki/Contract`, 2023. Accessed: April 2025.

[5] Gleb Naumenko and Antoine Riard. Coinpool: efficient off-chain payment pools for bitcoin.

[6] Jonas Nick, Andrew Poelstra, and Gregory Sanders. Liquid: A bitcoin sidechain. *Liquid white paper. URL https://blockstream. com/assets/downloads/pdf/liquid-whitepaper. pdf*, 2020.

[7] Jonas Nick, Tim Ruffing, and Yannick Seurin. MuSig2: Simple two-round schnorr multi-signatures. Cryptology ePrint Archive, Paper 2020/1261, 2020.

[8] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments, 2016.

[9] Ruben Somsen. Statechains: Off-chain transfer of utxo ownership, 2018.

A Preconfirmation

As mentioned in Section 7, one could immediately after receiving a VTXO from an Ark transaction, spend this VTXO again in another transaction. The VTXO is consequently not being confirmed through a batch swap, but remains in what one could informally call a *preconfirmation* state. This could create a sequence of Ark transactions spending preconfirmed VTXOs, until someone decides to batch swap the received VTXO.

Recall that spending a VTXO through an Ark transaction is a collaborative spend of that VTXO and thus requires a signature from the operator. Hence, a malicious user Alice would not be able to double-spend a preconfirmed VTXO already spent in a transaction to Bob, as long as the operator is

honest. Alice could still exit this preconfirmed VTXO unilaterally, but because the honest operator and Bob both know the Ark transaction spending that VTXO, Bob can claim the funds. Of course, the operator and/or Bob must monitor the mainchain for this.

Assuming an honest operator is a significant trust assumption. As discussed in Section 7, future work could investigate whether such a preconfirmation mechanism could also be achieved securely assuming a rational operator. This should make use of the fact that a double-spend as described above would be *detectable*. If it could be proven that the operator misbehaved, the only problem—which is by no means trivial—would be to devise a punishment mechanism that would disincentivise the rational operator from colluding in the first place.

We claim that such a deviation from the protocol by the operator is indeed detectable. We argue this informally for two cases:

- Consider a malicious user Alice colluding with a rational operator, and two honest users Bob and Charlie. Assume that Alice spends her preconfirmed VTXO $\texttt{vtxo}_A$ first in a transaction to Bob, and then double-spends it, colluding with the operator, in a transaction to Charlie. Now, Bob and Charlie could expose the operator by both coming forward and showing that the operator signed off on two Ark transactions spending $\texttt{vtxo}_A$.

- Alternatively, consider a malicious user Alice colluding with a rational operator, and one honest user Bob. Assume that Alice spends her preconfirmed VTXO $\texttt{vtxo}_A$ in a transaction to Bob, who receives $\texttt{vtxo}_B$, and then double-spends $\texttt{vtxo}_A$ by sending it to herself (for example by batch swapping it). Assuming that the rational operator allowed Alice to batch swap, it cannot allow Bob to batch swap $\texttt{vtxo}_B$. Indeed, this would come at a loss for the operator. If we assume moreover that Alice gave all the virtual transactions spending from the batch output to obtain $\texttt{vtxo}_A$, Bob is able to exit with $\texttt{vtxo}_B$ unilaterally. Since Bob is not getting $\texttt{vtxo}_B$ batch swapped, he will exit unilaterally. This would, however, again come at a loss for the operator, as the operator also funded Alice's fresh VTXO that came out of batch swapping $\texttt{vtxo}_A$. The operator would be forced to either take the loss, or to post the forfeit transaction for $\texttt{vtxo}_A$ which has now come onchain. This would again prove that the operator signed two transactions spending $\texttt{vtxo}_A$ (the Ark transaction and the forfeit transaction).

However, assuming that a certain class of users are willing to make this trust assumption, there is still one problem left. With the preconfirmation mechanism, a forfeit transaction is only signed for a VTXO that is batch swapped. Hence, a malicious user Mallory holding a VTXO $\texttt{vtxo}_M$ could, in theory, make a chain of Ark transactions to himself, each time spending the new VTXO in the next transaction, and batch swap the final VTXO $\texttt{vtxo}_{M'}$ or leave the Ark entirely. He could then exit the VTXO $\texttt{vtxo}_M$ unilaterally. The operator now has to post all Ark transactions onchain, incurring onchain fees to finally broadcast the forfeit transaction that spends $\texttt{vtxo}_{M'}$. As the potential Ark transaction fees will be significantly lower than the corresponding onchain fees, his griefing attack could allow Mallory to illegitimately claim both $\texttt{vtxo}_M$ and $\texttt{vtxo}_{M'}$, essentially stealing $\texttt{vtxo}_{M'}.\texttt{value}$ from the operator. To prevent this from happening, we can introduce checkpoints. These additional virtual transactions will allow the operator to claim a VTXO, unless the VTXO holder posts the Ark transaction that will be spent from this checkpoint transaction.

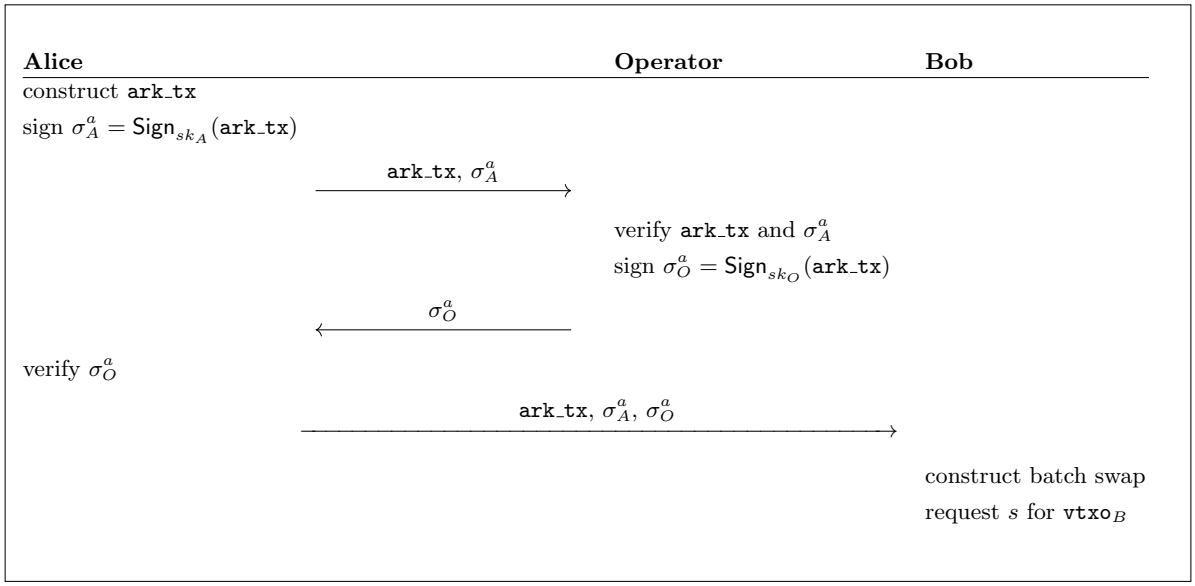
Once again, suppose that Alice wants to send the amount p to Bob. Apart from the Ark transaction defined earlier, she will also construct the checkpoint transaction:

Transaction 7: Checkpoint transaction.

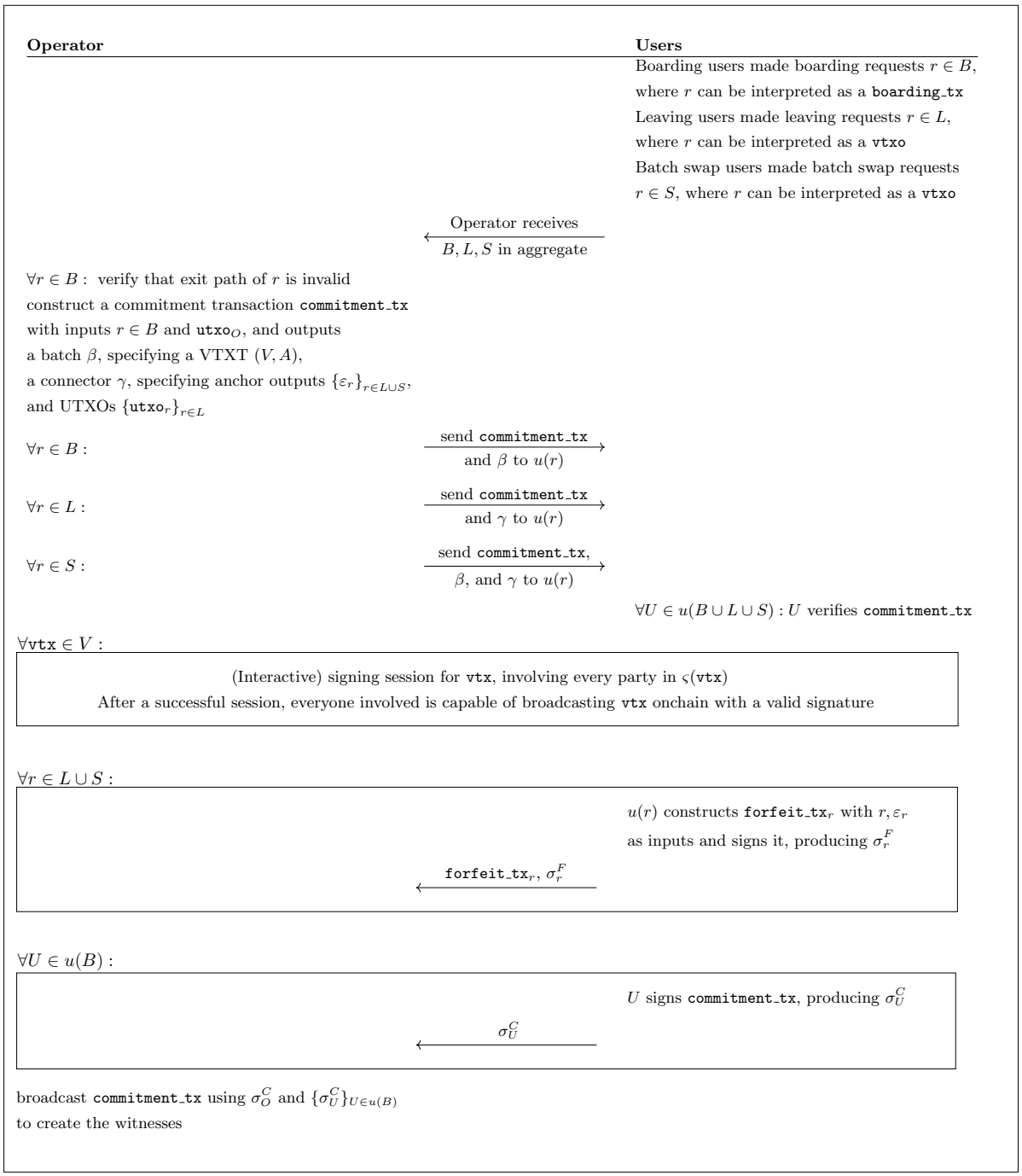
Inputs	Witnesses	Outputs
$\texttt{vtxo}_A$	w_{collab}	$(p, \texttt{Taproot}(\texttt{False}; \texttt{checkSig}_{pk_O \oplus pk_A}, \texttt{checkSig}_{pk_O} \wedge \texttt{absTimelock}(T_e)))$

Note how the output can either be spent by Alice and the operator collaboratively or swept by the operator as soon as the batch expires. Alice now signs only the Ark transaction and passes both `ark_tx` and `checkpoint_tx` to the operator. The operator verifies the scripts, signs both, and sends the signed transactions back to Alice. Alice can now safely sign the checkpoint transaction and pass it to the operator. In the earlier setting we sketched, where Mallory exited unilaterally with $\texttt{vtxo}_M$, the operator will now simply broadcast the appropriate checkpoint transaction, forcing Mallory to post the subsequent Ark transaction. The operator will always force Mallory to post the next Ark transaction, until eventually the operator can post the forfeit transaction. Mallory having to post all the Ark transactions to try to claim $\texttt{vtxo}_M$ should incentivise Mallory not to try to claim both $\texttt{vtxo}_M$ and $\texttt{vtxo}_{M'}$.

B Protocol Operations and Transaction Dependencies



Operation 1: Alice performs an Ark transaction, sending funds to Bob. Upon receiving `ark_tx` as described underneath Transaction 1 and the appropriate signatures, Bob will request a batch swap as specified in Operation 2 and only consider the transaction finalised once the corresponding commitment transaction is confirmed onchain.



Operation 2: Construction of a commitment transaction from boarding, leaving, and batch swap requests B, L, S respectively.

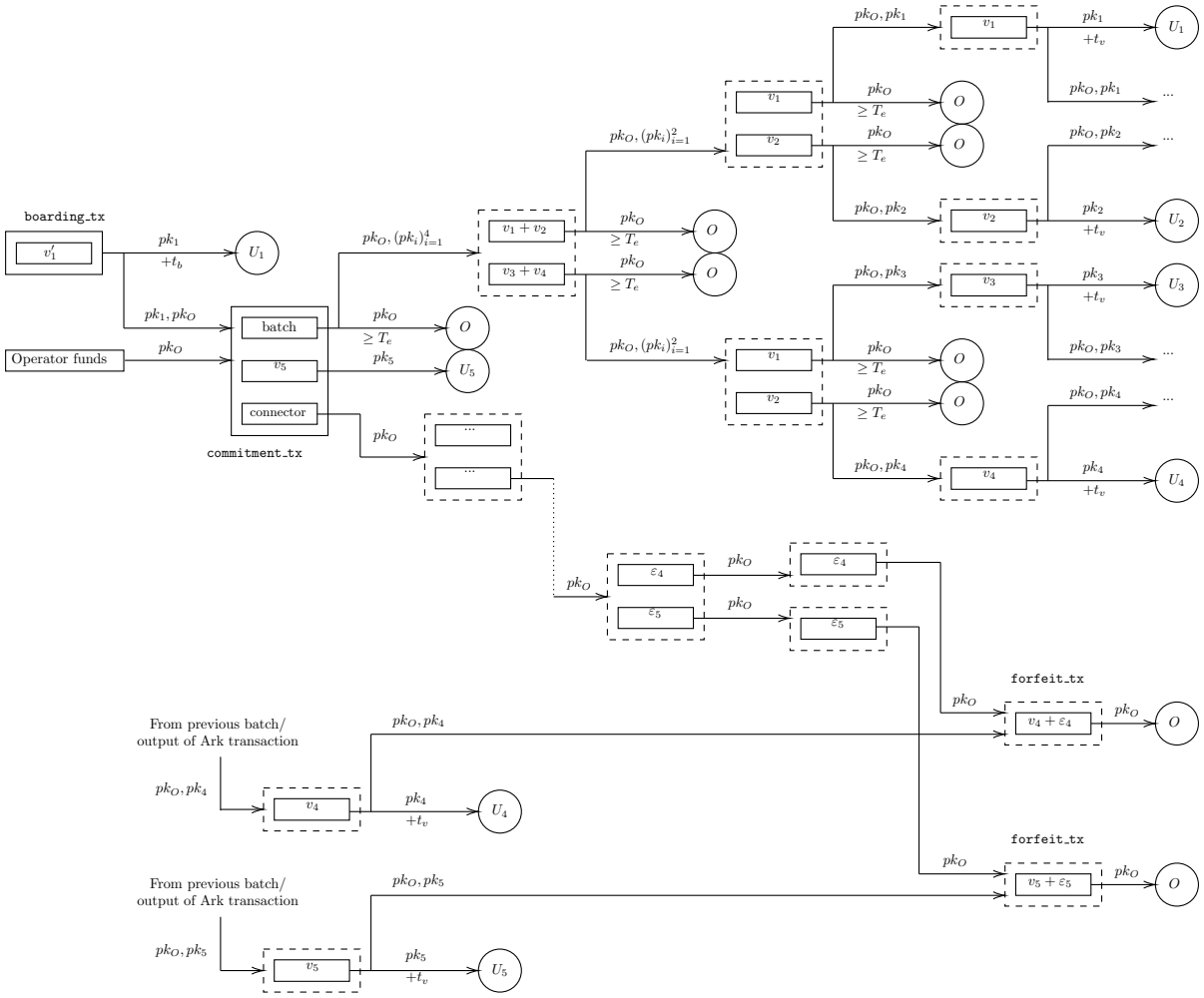


Figure 2: Transaction dependencies within the Ark protocol. Dashed transactions are virtual transactions and optimistically never appear onchain. Recall that T_e is the batch expiry, t_v the minimum delay for a unilateral VTXO exit, and t_b the boarding transaction timeout period. The batch expiry is represented as an absolute timelock, and this figure should be read assuming `commitment_tx` is included in the Bitcoin mainchain in a block we denote with time $t = 0$.