CoinPool: efficient off-chain payment pools for Bitcoin

Gleb Naumenko
gleb@thelab31.xyz

Antoine Riard
antoine@thelab31.xyz

ABSTRACT

CoinPool is a new multi-party construction to improve Bitcoin onboarding and transactional scaling by orders of magnitude.

CoinPool allows many users to share a UTXO and make instant *off-chain* transfers inside the UTXO while allowing withdrawals at any time without permission from other users.

In-pool accounts can be used for advanced protocols (e.g., payment channels). Connecting them to other CoinPool instances, or even to the Lightning Network, makes in-pool funds highly liquid.

CoinPool construction relies on $SIGHASH\_GROUP$, $SIGHASH\_ANYPREVOUT$ and $OP\_MERKLESUB$ changes to Bitcoin. It also assumes a high degree of interactivity between pool participants.

1 INTRODUCTION

While Bitcoin has established itself as a store-of-value, it is yet to be explored whether it can enable high transaction throughput without sacrificing security. Among many proposals [7, 9, 19] to achieve this goal, most attention was drawn to the Lightning Network [17].

Once LN matures and expands, its scalability and usability may face the following challenges:

- while technically two-party payment channels can exist off-chain forever, in practice they suffer from liquidity depletion bounding their lifetime
- two-party payment channels occupy significant block space *per user* when they get opened/closed, setting a bound on the number of LN users concurrently
- multi-hop trust-minimized routing may cause even more channel closings (in-flight HTLCs leading to on-chain claims)

Based on the lessons learned from the design and operation of the LN and other protocols, we propose CoinPool: a new scalability construction based on sharing UTXO ownership. It provides the following features:

- users can transfer their funds off-chain inside the pool
- every user can unilaterally exit a pool at any time without permission from other users
- in-pool accounts could be used for advanced contracts (e.g., payment channels)
- in-pool accounts could be used to transact with users of the LN or another CoinPool instance

2 BACKGROUND

Bitcoin is a peer-to-peer electronic cash system, in which all transactions are verified by users running node software. To let everyone participate in the most secure way and with minimized trust, Bitcoin limits transaction throughput to cap system requirements.

2.1 Bitcoin Transactions

Bitcoin addresses and the corresponding amounts define the state of the funds. Every address refers to a challenge to be satisfied to use a corresponding amount. Satisfying the most basic challenge requires a digital signature corresponding to a given public key.

Bitcoin Script is a language for expressing challenges that lock the coins. Bitcoin Script was designed to ensure low verification cost and enable running a Bitcoin node on low-tier hardware.

The main building block in Bitcoin Script is an opcode. For example, OP_CHECKSIG is used for signature verification. Other opcodes allow locking funds for a certain time, or until a preimage is revealed for a given hash.

Every transaction consists of the following components:

- inputs referring to the coins being spent
- scripts containing the rules locking the coins (scriptPubKey)
- signatures satisfying the scripts (scriptSig)
- outputs referring to the new rules locking those coins

A script and a corresponding solution (e.g., a digital signature) together are called *a witness*, as they allow to check transaction validity but not to determine transaction effects.

When Alice sends her coins to Bob, she crafts a transaction proving her ownership of a certain Unspent Transaction Output (UTXO). Then Alice broadcasts this transaction in the Bitcoin network to get it recorded in the Bitcoin blockchain.

2.1.1 Sighashes, SIGHASH_GROUP and SIGHASH_ANYPREVOUT.

A sighash flag is part of a signature indicating which part of the transaction body is signed. For example, a signature could cover all inputs and outputs ($SIGHASH\_ALL$) or all input and one output ($SIGHASH\_ONE$). These different modes of operation become useful in multi-key settings.

The following sighash flags were proposed to extend Bitcoin validation rules:

- $SIGHASH\_GROUP$ to enable bundling inputs with any number of outputs
- $SIGHASH\_ANYPREVOUT$ to avoid committing to outputs being spent

2.1.2 Covenants.

Bitcoin Script opcodes constraining spending transactions in terms of scriptPubKeys of their outputs are called covenants [10]. For example, Alice can send funds to Bob, which Bob can only spend to a specific address. Alice enforces this by asking Bob to construct his scriptPubKey in a certain way; and sends to that address.

Covenants are useful for protocols in which several users share a single UTXO. With this feature, users can be sure that they will receive a fraction of coins when the UTXO is spent.

2.1.3 *Taproot.*

Taproot is a UTXO format, which could be spent by satisfying one of the alternative conditions, represented as a Merkle tree. Spending a Taproot UTXO involves revealing a scriptPubKey, a corresponding Merkle branch, and a scriptSig.

In a multi-party setting, protocol participants may choose to tweak the Merkle root of a tree with an aggregation of their public keys, to avoid revealing the tree if they agree on payment details at the spending time. This aggregation is called an internal pubkey.

2.2 Payment channels

Off-chain solutions increase Bitcoin throughput at the cost of additional assumptions.

Payment channel protocols allow making multiple payments within a collateral capacity without recording a transaction in the blockchain (except for channel opening and closing transactions). Instead, channel counterparties pre-sign Bitcoin transactions and keep them locally. These transactions appear in the blockchain to enforce the latest channel state (balances) only when channels get closed.

This optimization comes with a new threat model. Users have to monitor the chain (and act in case of misbehavior) so that their counterparty cannot quietly settle an outdated state. This requirement contrasts with regular Bitcoin payments, in which users monitor the chain only at payment receiving time. This new threat model comes with an extended set of assumptions [1, 8, 18].

2.2.1 *Routing and the Lightning Network.*

Payment channels can be connected in a graph to enable payments between users not connected via a direct channel. The process of forwarding payments is called *routing*.

Lightning Network is an instance of a Payment Channel Network based on two-party channels. It is the most widely used off-chain protocol, with thousands of bitcoins locked in the network.

2.3 Off-chain update mechanisms

An update mechanism is what invalidates a state of a payment channel (or similar construction) while transitioning to a new state without touching the blockchain.

Currently, payment channels rely on LN-Penalty [17]: a payer discloses a secret revoking a previous set of pre-signed transactions. If a revoked transaction is published, the payee can use the disclosed secret to confiscate the payer's balance.

LN-Penalty does not work for N-channel users because 1) it requires storing all historical pre-signed transactions and revocation secrets, which does not scale well with the number of users; 2) distributing confiscated balance across N users remains an open game-theoretic question.

Eltoo is an alternative update mechanism [3] that overcomes these issues by introducing state ordering at the Bitcoin Script level, once $SIGHASH\_ANYPREVOUT$ is deployed.

3 OFF-CHAIN SCALING

When it comes to scaling Bitcoin with off-chain protocols, two factors matter: onboarding and transactional scaling. Let's now analyze the LN in light of these two metrics.

3.1 Onboarding scaling

This metric defines how many users can co-exist off-chain, considering throughput limits at the Bitcoin base layer.

Let's assume the following properties of the LN:

- channel lifetime is $Tdays$
- every time a user closes a channel, they open a new one in the same transaction
- every block can contain at most 3000 transactions

Then, $Tdays * 144blocks/day * 3000txs/block$ is the upper bound for a number of co-existing channels. If an average channel lifetime is 30 days, at most 13,000,000 channels could exist simultaneously in a secure way.

3.2 Transactional scaling

This metric defines how many transfers can be performed off-chain **per on-chain transaction**, considering the properties of the off-chain system.

The off-chain/on-chain ratio in the LN is limited because of the liquidity issues:

- Routing limits the capabilities to pay and receive payments in the network, forcing users to go on-chain.
- Channel owners have different goals while maintaining a payment channel [1].

4 COINPOOL: SCALABILITY/SECURITY MODEL

CoinPool is a trust-minimized multi-user ($N > 2$) off-chain protocol that provides high onboarding and transactional scaling at the cost of high interactivity.

Balances in the pool are organized in a Taproot tree, where leaves ensure every user can spend their balance immediately without permission from other users. A user cannot lose balance or be forced to leave the pool.

To make sure a withdrawing user preserves the state of the remaining CoinPool, unilateral withdrawals must be restricted, so that a user cannot withdraw more than their balance, and the rest of the tree should remain unmodified (see Fig. 1).

Off-chain transfers work by spending a CoinPool output into new CoinPool output of arbitrary state via replacing

[1]For example, if one channel counterparty raises routing fees over the channel, it may become prohibitively expensive for another counterparty to rebalance the channel.

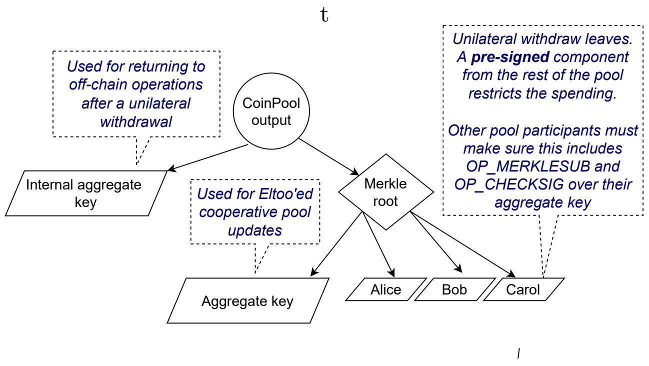
t
Used for returning to off-chain operations after a unilateral withdrawal
CoinPool output
Unilateral withdraw leaves. A **pre-signed** component from the rest of the pool restricts the spending.
Other pool participants must make sure this includes OP_MERKLESUB and OP_CHECKSIG over their aggregate key
Internal aggregate key
Used for Eltoo'ed cooperative pool updates
Merkle root
Aggregate key
Alice
Bob
Carol

Figure 1: CoinPool is organized as a tree, in which leaves correspond to user balances and enable unilateral withdrawals. Two extra spending paths are used for cooperative pool operations.

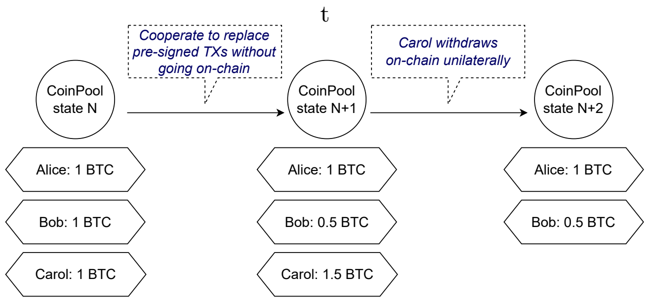

Figure 2: CoinPool: users can transfer funds within a single UTXO and withdraw at any time.

pre-signed transactions. This transition must be authorized by all participants' aggregate public key and with the use of Eltoo.

CoinPool balances can be locked with arbitrary scripts, which allows them to be used as payment channels. This allows 1) connecting CoinPool instances to each other and to the LN; 2) not to ask permission for balance-level state changes from the rest of the pool, which improves scalability [2] and privacy. Other pool-level actions (for example, transfers between balances) do not affect the operation of these in-pool payment channels.

4.1 Enabling any-order exits in a scalable way with OP_MERKLESUB, SIGHASH_ANYPREVOUT and SIGHASH_GROUP

The ability to exit a pool at any time without permission is a fundamental CoinPool feature.

Two primitives are used to restrict a withdrawing participant from violating rules: $SIGHASH\_GROUP$ to restrict the balance, and $OP\_MERKLESUB$ to maintain the tree, which we present shortly (but also uses $SIGHASH\_GROUP$ and $SIGHASH\_ANYPREVOUT$) [3].

Now, we present $OP\_MERKLESUB$.

4.1.1 Enforcing tree transitions with OP_MERKLESUB. Every pool withdrawal results in two new UTXOs: one for the exiting user, and one for the rest of the pool. $OP\_MERKLESUB$ is a new opcode enforcing that the new CoinPool UTXO is equal to the UTXO this transaction is spending, but:

- user's leaf is subtracted from the Merkle tree
- user's pubkey is subtracted from the internal pubkey

At the transaction verification level, these rules require Merkle tree subtraction and elliptic curve point addition.

This scheme also relies on:

- $SIGHASH\_ANYPREVOUT$ allowing signatures to not commit to a particular input
- $SIGHASH\_GROUP$ allowing signatures to not commit to a particular output

All these features combined ensure any-order pool exits without authorizing every possible sequence of exits via a factorial number of signatures.

4.2 Enabling leaf-level payment channels with SIGHASH_ANYPREVOUT

Using CoinPool leaves as two-party payment channels or similar protocols largely increases the scalability of CoinPool.

This feature requires $SIGHASH\_ANYPREVOUT$, because otherwise transactions enforcing leaf-level payment channels would have to commit to a particular CoinPool state hash, which means every state change of the pool would invalidate these payment channels.

5 COINPOOL OPERATIONS

CoinPool instance state is defined by a set of public keys and the associated balances, which are debited and credited during pool operation.

This state is maintained by an on-chain UTXO and a set of pre-signed transactions. Pre-signed transactions are the only objects aware of user balances: they restrict users from exceeding balances while withdrawing.

Off-chain transfers between pool accounts require replacing pre-signed transactions with new ones. All pool participants have to approve these transitions (see Fig. 2). If an update reduces the balance of a user unexpectedly, this user should refuse to authorize the update.

If an outdated transaction is published on-chain, participants should challenge it to make sure their funds remain safe. This action requires all pool participants to store the pool state locally.

[2]In the next section, we will show that by default, such actions require authorization from all pool participants

[3]These restrictions could be also achieved by pre-signing all possible sequences of state transitions (producing, storing and exchanging all these signatures), which scales poorly (factorial) with the number of participants. For balances specifically, another potential solution is to enforce all balances to be equal, which would significantly reduce CoinPool usability.

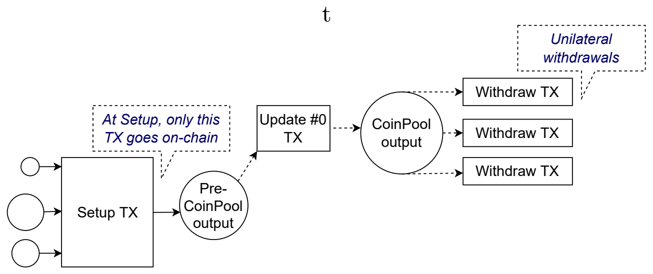

Figure 3: Pool Setup: Update #0 TX and Withdraw TXs are signed, then Setup TX is signed and submitted on-chain.

The CoinPool UTXO usually gets re-spent **on-chain** when one of the participants leaves the pool. This action does not require permission from the rest of the pool.

The distribution of the balances, and even the existence of the pool, is hidden from external observers.

5.1 Pool Setup

Bitcoin users initiate a CoinPool instance by crafting a **Setup transaction**, an **Update transaction** with an initial state; and one **Withdraw transaction** spending an Update output for every user (see Fig. 3).

To make sure funds don't get frozen, transactions are signed in reverse topological order.

Inputs of a Setup transaction correspond to users' contributions, and the only output funds the CoinPool state. This output can be spent by all participants together unconditionally. This output is called *pre-CoinPool*: it does not allow unilateral withdrawals yet (an already signed Update transaction must be published first).

We will now explain the construction of Update and Withdraw transactions.

5.2 Pool Update (off-chain)

Initiating a transfer to another pool balance or changes to the lock conditions (key rotation, adding a timelock) requires proposing an **Update transaction** spending pre-CoinPool UTXO into new off-chain output. If another update already existed on top of the pre-CoinPool UTXO, it should be replaced via Eltoo.

After verifying the correctness of the transition, other participants authorize this update by 1) generating/signing a new set of Withdraw transactions; 2) signing the proposed Update transaction (see Fig. 4).

The output of the Update transaction is a CoinPool Taproot output with the following spending paths:

- Key path remains the same as in the UTXO being spent and requires authorization from all pool participants cooperatively
- Script path represents a tree ensuring: 1) every user has a leaf allowing them to withdraw at any time without permission, although in a restricted way (see next Section); 2) one Eltoo-enabled leaf based on the aggregate public key enables off-chain. transitions [4]

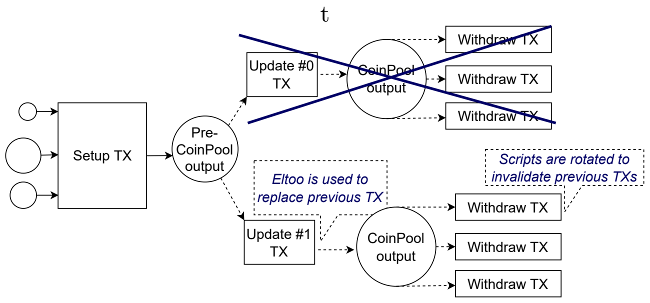

Figure 4: CoinPool Update: Users cooperate to produce a new Update transaction enforcing pool state transition. Withdraw transactions should be invalidated by rotating scripts.

Pool participants must force each other to rotate their withdraw scripts, to invalidate existing Withdraw pre-signatures, which otherwise may be used to double-spend honest users.

Once a new Update transaction and a new set of Withdraw transactions are generated and signed, this update is finalized. No recording in the blockchain is required.

5.3 Pool Withdrawal

Every pool participant can withdraw their balance from the pool at any time. Spending a **Withdraw transaction** always involves using a private component by the leaf owner and a pre-signed component signed by other participants.

First, a withdrawing participant submits the latest Update transaction.

Then, the contestation delay takes place: a withdrawing user should wait for a certain number of blocks before submitting a Withdraw transaction. This delay lets other users react if the submitted Update transaction was not the latest. This delay is enforced in the pre-signed component of Withdraw transactions.

Once the delay is over, a Withdraw transaction spends the CoinPool UTXO into two outputs: one for the withdrawal (see Fig. 5), and one for the remaining pool. In addition to satisfying the user's scriptPubKey content of the leaf, the transaction must satisfy the following:

(1) Second output is a Taproot output with the same tree from the UTXO being spent, but the withdrawing user is excluded from both key and script paths.
(2) A point and a leaf removed from the tree should correspond to the withdrawing user's point and leaf.
(3) First output amount should correspond to the balance assigned to the leaf.
(4) Second output amount should equal the previous pool amount without the second output amount.

[4]This can't be done via key path. Key path in the CoinPool output from an Update transaction is not supposed to be spent at all, it is only used for *OP_MERKLESUB*.

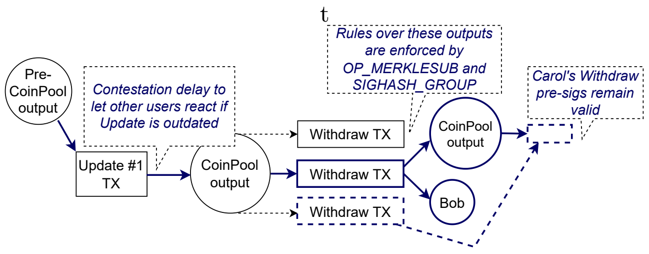
t
Pre-CoinPool output
Contestation delay to let other users react if Update is outdated
Update #1 TX
CoinPool output
Rules over these outputs are enforced by OP_MERKLESUB and SIGHASH_GROUP
Withdraw TX
Withdraw TX
Withdraw TX
CoinPool output
Bob
Carol's Withdraw pre-sigs remain valid

Figure 5: CoinPool Withdraw: Bob leaves the pool by using his private key and a pre-signed Withdraw component from other participants. Then, Carol considers leaving with an existing pre-signed transaction.

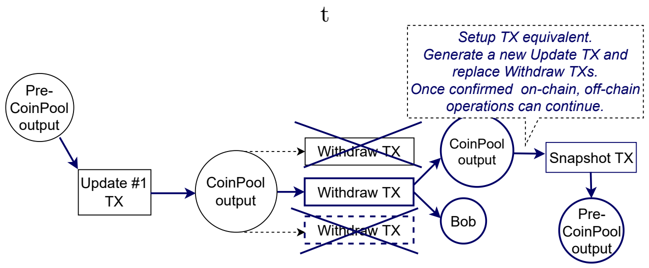

Figure 6: Coinpool Snapshot: once Bob withdrawn, a Snapshot transaction should be produced and confirmed on-chain in the same way as Setup. Then, off-chain operations can proceed.

(1) and (2) are enforced via $OP\_MERKLESUB$, which must be satisfied to spend the UTXO. Pool participants force each other to always include $OP\_MERKLESUB$ in their leaves.

(3) and (4) are enforced via $SIGHASH\_GROUP$ flag applied to the pre-signed component of the Withdraw transaction. Pool participants force each other to always include $OP\_CHECKSIG$ over the aggregate key in their leaves and pre-sign these authorizations only with $SIGHASH_GROUP$.

Users can withdraw from the pool one after each other. In that case, no Update transaction will be recorded in the blockchain, but the second user would still have to wait during the contestation delay after the first user [5].

5.4 Pool Snapshot

Once a user withdraws from the pool, the remaining participants can either continue withdrawing non-cooperatively or return to off-chain pool operations. For the latter, a **Snapshot transaction** must be confirmed on-chain. It ensures that outdated Withdraw pre-signed components are invalidated.

A Snapshot transaction takes one input from the key path of the UTXO produced by the last Withdraw transaction and spends it into one output, identical to the Setup output.

Similarly to Setup, participants should produce all required Withdraw and Update transactions spending a Snapshot output, before authorizing the Snapshot transaction.

5.5 Expanding the pool

To add a new user to the CoinPool instance, current participants spend the pool UTXO into new UTXO which includes the new user (both in key path and script path), via an on-chain transaction [6]. Similarly to Setup, all required Withdraw and Update transactions should be generated.

[5]It doesn't provide additional security against the second withdrawal, but it significantly simplifies the protocol design.
[6]This transaction must be recorded on-chain, because otherwise new user can't be secure.

This action could use a cooperative Update transaction or a Snapshot transaction. A transaction could be expanded to include a contribution from the new user.

5.6 Optimizations

Going on-chain usually happens at unilateral Withdraw, or while onboarding new users. These actions can be aggregated with each other or with other updates. One transaction can include Alice leaving the pool, adding Bob to the pool, and transferring funds from Carol to Dave.

Cooperative on-chain action is always cheaper in fees than unilateral Withdraw, because transaction witness does not contain Merkle path, and thus, is smaller and cheaper. For off-chain, it just avoids producing a separate signature for a transfer.

6 UPLIFTING OFF-CHAIN PROTOCOLS

CoinPool leaves could be used for advanced protocols (payment channels, Discreet Log Contracts [6], CoinSwaps [12]). This is one of the key scalability improvements CoinPool provides, which we call *uplifting*. Those protocols can operate inside the pool without affecting the entire pool state (see Fig. 7).

An uplifted protocol instance is secure for all participants only if all of them are involved in authorizing pool-wide actions: signing cooperative updates and pre-signing unilateral withdrawals. Otherwise, a leaf can be modified without the user's consent.

Activity within an uplifted protocol instance (at a leaf-level) remains unknown to non-involved pool participants.

An uplifted protocol instance adheres to the same threat model, as the non-uplifted version. A user has to:

- monitor the blockchain to ensure no outdated state is submitted [1]
- make sure their reaction transactions could be timely included in the blockchain [8, 18]

Additionally, pool timelocks (in Withdraw transactions) should be accounted for while picking timelocks for uplifted protocol instances.

t

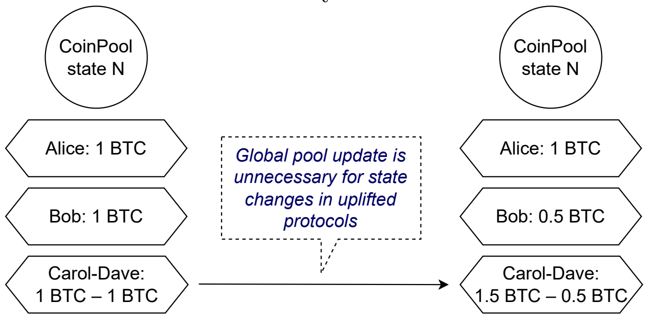
CoinPool state N
Alice: 1 BTC
Bob: 1 BTC
Carol-Dave: 1 BTC – 1 BTC
Global pool update is unnecessary for state changes in uplifted protocols
CoinPool state N
Alice: 1 BTC
Bob: 0.5 BTC
Carol-Dave: 1.5 BTC – 0.5 BTC

Figure 7: Uplifting: a leaf-level payment channel can be updated without affecting the rest of the pool.

Leaf-level payment channels may be used to route payments to the Lightning Network, or another CoinPool instance. To achieve this, one of the payment channel participants must have another payment channel in the second system. This user would take the risks corresponding to cross-system transfers, similarly to how routing nodes in the LN take the risk for forwarding payments through their channels.

These payments have to be authorized only by leaf participants and remain unknown to the rest of the pool.

7 DISCUSSION

7.1 Economic soundness of multi-party Eltoo

Eltoo assumes publishing a reaction transaction in case a counterparty confirmed an outdated transaction on-chain.

In a two-party setting, there is always an attacker and a victim. In N-party protocols, there could be $M < N$ victims. It is unclear how victims coordinate the reaction, especially in terms of reaction transaction fees.

A protocol instance (e.g., CoinPool) may have an extra leaf (or a standalone UTXO) with contributions from all pool participants for this reason. Additionally, it can encourage honest pool participants to be first at reacting by offering a reward paid from this additional collateral.

7.2 Off-chain Cancellation of Withdraw Transactions

Once a unilateral withdrawal was confirmed on-chain, continuing pool off-chain operations requires confirming a Snapshot transaction, which results in extra on-chain footprint and coordination.

A Snapshot transaction invalidates Withdraw transactions, because otherwise, if Alice withdraws right after Bob, Bob remains in the Eltoo leaf of Alice's remaining pool output. The remaining pool participants must not rely on Bob contesting further updates.

Alternatively, this can be achieved by extending Bitcoin Script rules to function across outputs. More specifically, pool participants should be able to force (via a pre-signed component) Withdraw transactions to use an internal pubkey from their first output (the remaining pool) as a condition locking the second output (timelocked UTXO of a withdrawing participant), so that withdrawing users are excluded from further contesting.

7.3 Interactivity requirement

CoinPool updates require the participation of all pool participants. This sets an upper bound on the number of pool users. The main bottlenecks are:

- online: downtime of a single participant halts pool updates
- DoS-resistance: malicious participants proposing an excessive number of updates; or denying to authorize any updates
- resources: latency and bandwidth/space required to coordinate updates

Interactivity requirement also makes *cold storage* security approach impossible for the keys authorizing updates.

To overcome these issues, we suggest several solutions:

- applying reputation systems to select pool participants for DoS-resistance
- replicated key hosting for better availability
- kicking-out at the pool level: forcing a user to withdraw after inactivity
- in-pool fragmentation: updates in one branch without authorization from another branches

7.4 Withdraw witness size

Unilateral Withdraw transactions have large witness sizes because they require revealing a Merkle branch containing the withdrawing user's leaf. Witness size in this case grows logarithmically with the pool size.

This results in the inefficient use of Bitcoin throughput, and high fees for the withdrawing user.

We suggest the following optimizations:

(1) Sharing leaves across several users via leaf-level conditions enabled by Bitcoin Script
(2) A more efficient accumulator for inclusion proving [5].

7.5 Dynamic Membership

CoinPool does not allow onboarding new users' without an on-chain transaction. This results in reduced scalability and usability of CoinPool.

Onboarding new users off-chain securely is hard, because existing pool participants can cooperate to trick a new user into believing new user's funds are safe, and then double-spending via free-to-produce state *fork*.

We leave exploring this problem for future research.

7.6 Confidentiality enhancements

Even though CoinPool hides most of the activity from the outsider, an in-pool observer can track all in-pool activities unless they happen at the leaf level.

Anonymity can be increased by setting a lower bound on in-pool balance and a timelock to prevent a spy from leaving the pool immediately after entering, or via periodic in-pool mixing.

8 COINPOOL APPLICATIONS

8.1 Trading Pools

CoinPool can facilitate trading or liquidity management (for example, between a broker firm and its clients) more efficiently than the LN or other systems, due to better confidentiality and funds velocity.

Pool policies could be optimized for trading, and include market makers, trade ordering rules, or alternative threshold conditions. For example, a broker could authorize a state transition if 50% of participants confirm.

8.2 Commodity Contracts

Commodity trade is a widespread business activity with longer timespans and across multiple jurisdictions. For these deals, it is often useful to take many different events into consideration and act accordingly.

CoinPool can be used to facilitate these advanced contracts if every possible event corresponds to a pre-signed transaction enforcing pool state transition.

8.3 Smart Corporation

A CoinPool instance can represent a corporation, with stakeholders having different balances and voting rights to implement corporate governance and distribute dividends. A special balance could be used as a corporate account and controlled by the majority of the participants.

This idea gets more powerful once combined with other advanced protocols (e.g., Chaumian banks [2] or vaults [14, 23]).

9 RELATED WORK

First multi-party ($N > 2$) protocols for Bitcoin were *mixers* enhancing privacy by obfuscating coin ownership across many users [11, 12]. Similarly to CoinPool, they provide confidentiality against an outsider, but not against an in-protocol spy. Although confidentiality boost is a secondary feature for CoinPool.

Sidechains [7, 22] scale Bitcoin by locking Bitcoins controlled by an external consensus (e.g., a federation) to authorize state transitions. CoinPool is more trust-minimized than sidechains, but, unlike sidechains, it is limited by Bitcoin Script for off-chain activities. Sidechains could use CoinPools for locking and governing locked Bitcoins.

The Lightning Network [17] aims to increase transaction throughput by moving most of the activity off the chain via two-party payment channels. CoinPool builds on similar scalability ideas while providing better funds velocity at a cost of higher interactivity.

Channel factories [4] enhance the LN by batching openings and closings of many channels into one transaction. CoinPool also builds on the idea of UTXO sharing, but, unlike Channel factories, CoinPool remains operational after one participant withdraws from the UTXO.

Joinpool [13] is a more generic UTXO sharing protocol, which, unlike Channel factories, remains operational after a user withdraws. CoinPool extends this by offering off-chain operations and protocol uplifting.

Both Channel factories and Joinpool protocols offer mixing coins against an external observer, so does CoinPool.

The pooling idea was further advanced by representing accounts in a Radix tree [21] for witness size reduction. Compared to this construction, CoinPool requires generation more transactions and interactive setup, while offering a smaller on-chain footprint and allowing unilateral withdrawals.

This Radix tree was prototyped on pre-signed transactions without modifications to Bitcoin Script, while OP_CTV [20] was proposed to reduce construction complexity and risks associated with pre-signed transactions.

$OP\_TAPROOT\_LEAF\_UPDATE\_VERIFY$ [24] is a Bitcoin Script extension allowing accumulator-based operations, similarly to $OP\_MERKLESUB$. OP_CAT [15] and OP_CSFS [16] are two other general-purpose covenant proposals to Bitcoin Script. These proposals may be used to design pool protocols, although this direction was not thoroughly explored.

10 CONCLUSIONS

Lightning Network scales Bitcoin payments via two-party payment channels and routing, although it may be bounded in terms of scaling and usability by its design.

We propose CoinPool, a covenant-based payment pool construction, which takes the idea of sharing UTXO ownership to the next level. In CoinPool, Bitcoin users lock funds in many accounts within a single UTXO to instantly transact across the pool without much on-chain footprint; or use their accounts inside the pool for advanced protocols (e.g., payment channels), possibly even connected to other CoinPools or the LN. CoinPool users can withdraw their funds from the pool at any time.

CoinPool introduces an alternative set of trade-offs: users get increased funds velocity and use Bitcoin throughput more efficiently, at the cost of high interactivity required by pool participants. CoinPool requires modifications to the Bitcoin protocol.

REFERENCES

[1] Gleb Naumenko Antoine Riard. Time-dilation attacks on the lightning network, https://arxiv.org/pdf/2006.01418.pdf, 2020.

[2] David Chaum. Blind signatures for untraceable payments. In David Chaum, Ronald L. Rivest, and Alan T. Sherman, editors, *Advances in Cryptology*, pages 199–203, Boston, MA, 1983. Springer US.

[3] Rusty Russell Christian Decker and Olaoluwa Osuntokun. eltoo: A simple layer2 protocol for bitcoin, https://blockstream.com/eltoo.pdf, 2018.

[4] Christian Decker Conrad Burchert and Roger Wattenhofer. Scalable funding of bitcoin micropayment channel networks, https://nakamotoinstitute.org/static/docs/scalable-funding-of-bitcoin-micropayment-channel-networks.pdf, 2018.

[5] Ben Fisch Dan Boneh, Benedikt Bunz. Batching techniques for accumulators with applications to iops and stateless blockchains, https://eprint.iacr.org/2018/1188.pdf, 2018.

[6] Thaddeus Dryja. Discreet log contracts, https://adiabat.github.io/dlc.pdf, 2017.

[7] Adam Back et al. Enabling blockchain innovations with pegged sidechains, https://www.blockstream.com/sidechains.pdf, 2014.

[8] Aviv Zohar Jona Harris. Flood & loot: A systemic attack on the lightning network, https://arxiv.org/pdf/2006.08513.pdf, 2020.

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

[10] Gregory Maxwell. Coincovenants using scip signatures, an amusingly bad idea, https://bitcointalk.org/index.php?topic=278122.0, 2013.

[11] Gregory Maxwell. Coinjoin: Bitcoin privacy for the real world, https://bitcointalk.org/?topic=279249, 2013.

[12] Gregory Maxwell. Coinswap: Transaction graph disjoint trustless trading, https://bitcointalk.org/index.php?topic=321228.0, 2013.

[13] Gregory Maxwell. Rolling coinjoin(”joinpool”), https://gist.github.com/harding/a30864d0315a0cebd7de3732f5bd88f0, 2020.

[14] Malte Moser, Ittay Eyal, and Emin Gün Sirer. How to implement secure bitcoin vaults, Feb 2016.

[15] Satoshi Nakamoto. misc changes, https://github.com/bitcoin/bitcoin/commit/4bd188c4383d6e614e18f79dc337fbabe8464c82, 2010.

[16] Russell O’Connor. Cat and schnorr tricks i, https://www.wpsoftware.net/andrew/blog/cat-and-schnorr-tricks-i.html, 2021.

[17] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments, https://lightning.network/lightning-network-paper.pdf, 2016.

[18] Antoine Riard. Pinning: The good, the bad, the ugly, https://lists.linuxfoundation.org/pipermail/lightning-dev/2020-june/002758.html, 2020.

[19] Casey Rodarmor. Lightning mints, https://rodarmor.com/blog/lightning-mints/, 2021.

[20] Jeremy Rubin. Checktemplateverify, https://github.com/bitcoin/bips/blob/master/bip-0119.mediawiki, 2020.

[21] Jeremy Rubin. Coinpool, exploring generic payment pools for fun and privacy, https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2020-june/017968.html, 2020.

[22] Ruben Somsen. Softchains: Sidechains as a soft fork via proof-of-work fraud proofs, https://gist.github.com/rubensomsen/7ecf7f13dc2496aa7eed8815a02f13d1, 2020.

[23] Jacob Swambo, Spencer Hommel, Bob McElrath, and Bryan Bishop. Custody protocols using bitcoin vaults, 2020.

[24] Anthony Towns. Tapleaf_update_verify covenant opcode, https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2021-september/019419.html, 2021.