# Kuroshio: Pools and Payment Channels for Monero

floppy

Draft, November 2026

**Abstract**

Monero has no scripting language. An output is controlled by a single key and payments cannot be made conditional. Under the FCMP++ rules no transaction can wait for a deadline either. These are the primitives that pools, payment channels and multi-hop payments are built from on Bitcoin. This paper shows that the FCMP++ upgrade, together with a proposed relative lock, provides enough structure to build all three. Kuroshio is a protocol in which $n$ parties hold funds in one output under an $n$-of-$n$ FROST key and sign every exit transaction before any of them pays in. Because an FCMP++ input refers to the output it spends only through a membership proof, a pool state can be made identical whichever exits produced it, which reduces the number of exit signatures from factorial to $O(n\,2^n)$ for unrestricted exit orders. Adaptor signatures for the FCMP++ spend authorization proof give punishable two-party payment channels following a construction by tevador. Placed inside pools, these channels form a channel factory. Point time-locked contracts route payments between channels, with points decorrelated at every hop and refunds measured in whole multiples of the single lock length. The paper specifies the off-chain update protocol, on-chain dispute handling and a watchtower that cannot read what it stores before a breach. A Rust implementation runs every path on regtest networks, including one built from the proposed consensus change.

## 1 Introduction

Off-chain payment systems on Bitcoin are assembled from a small set of script features. Multi-signature outputs let parties share funds. Hash locks make a payment conditional on a secret. Relative and absolute time locks give an honest party time to react and refund a sender when a payment does not complete. The Lightning Network [1] combines these into channels and multi-hop payments. Payment pools and channel factories use the same pieces to let many users hold funds in one output.

Monero has none of these features. Every output has a one-time key and is spent by a proof of knowledge of that key. Monero multisignature wallets exist, but a ring signature commits to a ring that contains the spent output, so no transaction can be signed before the output it spends is on chain. The only time field, the unlock time, locks the outputs a transaction creates and the FCMP++ rules require it to be zero. Earlier work on Monero channels therefore relied on a new signature scheme that requires a hard fork [14], on verifiable timed linkable ring signatures [15] or on a key escrow service [22], [23].

The FCMP++ upgrade [2] replaces ring signatures with a membership proof over every spendable output on chain and a separate spend authorization and linkability (SAL) proof about a re-randomized output. Three properties of this design matter here. First, outputs have two independent secret components and one of them can be split among signers with a standard threshold Schnorr protocol. Second, the data a SAL proof signs does not contain the membership proof, so a transaction can be fully authorized before the output it spends has been mined. Third, an input identifies its output only through the membership proof, so two different transactions that create an output with the same key and commitment are interchangeable for any later spend. A proposed consensus change adds a relative lock [7], [8].

**Contributions :**

- A pool construction in which $n$ parties pay into one output and sign every exit before paying in (Section 4). Pool states are path independent, which bounds the number of signed exits by $n$ times the number of reachable member subsets instead of the number of exit orders.

- Adaptor signatures for the FCMP++ SAL proof under *n*-of-*n* FROST (Section 5).
- Punishable two-party channels based on a construction by tevador [7], with an update protocol whose message order lets an honest party enforce the latest state it committed to at every step (Section 6).
- A channel factory in which pool payouts are channel funding outputs, so that each channel state is signed once, with settlement, refresh into new channel pairs and unilateral exit (Section 7).
- Multi-hop payments using point time-locked contracts with per-hop decorrelated points, onion routing and refunds in multiples of a single lock length (Section 8).
- On-chain enforcement of stuck payments and a blind watchtower (Section 9), a security and privacy analysis (Sections 10 and 11) and an implementation with measurements (Section 12).

## 2 Background

### 2.1 Notation

All group operations are on the Ed25519 curve, written additively, with prime subgroup order $\ell$. $G$ is the base point, $H$ the amount generator and $T$, $U$, $V$ are the additional FCMP++ generators. $H_p$ is a hash to the curve and $H_s$ a hash to $\mathbb{Z}_\ell$. keccak is Keccak-256 and $\|$ denotes concatenation. A Pedersen commitment to amount $a$ with mask $m$ is $C = mG + aH$. Labels in quotation marks are byte strings used for domain separation.

### 2.2 FCMP++ outputs and spends

An FCMP++ output is a pair $(O, C)$ with one-time key $O = xG + yT$. Its key image is $J = x\, H_p(O)$. To spend it, the owner re-randomizes the output as

$$\tilde{O} = O + r_o\, T, \qquad \tilde{I} = H_p(O) + r_i\, U, \qquad R = r_i\, V + r_{r_i}\, T, \qquad \tilde{C} = C + r_c\, G$$

and attaches two proofs. The membership proof shows that $(\tilde{O}, \tilde{I}, \tilde{C})$ re-randomizes some output in the tree of all outputs. The SAL proof shows knowledge of $x$ and $y$ for $\tilde{O}$ and that $J$ is computed from the same $x$. The transaction balances if the sum of the $\tilde{C}$ equals the sum of the output commitments plus the fee times $H$.

The message signed by every SAL proof of a transaction is

$$\text{signable} = \text{keccak}(\ \text{keccak(prefix)} \,\|\, \text{keccak(base)}\ )$$

where the prefix contains the unlock time, the key images, the output keys and the extra field. The base contains the fee, the encrypted amounts and the output commitments. The membership proof, the range proof, the reference block and the pseudo-outputs $\tilde{C}$ are outside this hash. The $\tilde{C}$ are still bound, because each SAL proof commits to its input tuple. The weight of a transaction, from which its fee is computed, depends only on the number of inputs and outputs and the size of the extra field. It does not depend on the depth of the tree at broadcast time. Consequently a transaction can be signed, with its final fee, before the outputs it spends exist.

### 2.3 Carrot

Carrot [3] is the addressing protocol introduced with FCMP++. A payment to address $(K_s, K_v)$ creates an output with key $O = K_s + k_g^{\,o}\, G + k_t^{\,o}\, T$, where $k_g^{\,o}$ and $k_t^{\,o}$ are derived from a shared secret between sender and receiver. Its ephemeral key is derived from a random anchor and the input context of the transaction, which for a transaction that is not a coinbase is "R" $\|$ $J_1$ for the first key image $J_1$.

### 2.4 Threshold signatures

FROST [4] is a two-round threshold Schnorr signature protocol. In the first round each signer publishes two nonce commitments. In the second round each signer releases a response share and the shares sum to an ordinary Schnorr response. Key generation in Kuroshio uses MuSig key aggregation [5] over the participants' keys on $T$, so the aggregate secret never exists in one place and no interactive key generation is needed. The implementation uses the modular-frost library [21].

### 2.5 Adaptor signatures

An adaptor signature, or pre-signature, against a point $A = tT$ is an incomplete signature that becomes valid when combined with $t$, such that anyone holding both the pre-signature and the completed signature learns $t$ [6]. It replaces the hash lock. Revealing a secret becomes a side effect of spending.

### 2.6 Relative locks

tevador proposed a relative lock for FCMP++ [7], implemented in [8]. A transaction with unlock_time = 1 is valid only if its membership proof is against a tree state at least $L = 2160$ blocks old, about three days. Every input of such a transaction is therefore older than $L$ blocks. An output enters the tree about ten blocks after it is mined, so a lock lasts slightly longer than $L$ blocks. The unlock time is part of the signed prefix, so a pre-signed transaction cannot drop its lock. There is a single lock length. The change is under review and not part of Monero consensus. All locked transactions in this paper depend on it.

## 3 Model and goals

A pool has $n$ members. Each member runs its own process with its own wallet and reaches the others through a message relay. Members may deviate arbitrarily from the protocol, collude with each other and go offline at any point. The analysis assumes that the discrete logarithm problem is hard in the Ed25519 prime subgroup, that the hash functions behave as random oracles and that the FROST and SAL proofs are unforgeable. It also assumes that an honest party that needs to react on chain can get a transaction mined within $L$ blocks of the event it reacts to. For routed payments it assumes that the transactions a router broadcasts for one payment are delayed by fewer than $L$ blocks in total. The relay sees ciphertexts, sizes and timing. It may drop or delay messages. A chain observer sees every transaction.

The protocol targets four properties. The first three hold for a single honest member against all other parties. Privacy among members needs at least two honest members, because $n - 1$ colluding members find the remaining payout address by elimination.

1. **Balance security :** An honest member never ends with less than its balance in the latest state it committed to, minus fees fixed at signing time.
2. **Unilateral exit :** An honest member can obtain that balance on chain without the cooperation of any other party.
3. **No trusted party :** No operator, coordinator or escrow is involved. The relay can stop progress but cannot affect balances.
4. **Privacy :** A chain observer cannot link a member's entry to its payout. Members do not learn which payout address belongs to which other member. Routers of a payment do not learn its payer or receiver.

# 4 Pools

## 4.1 Keys and the pool seed

Member $i$ picks a secret $s_i$ and publishes $S_i = s_i\,T$. Members are numbered by the encoding of their keys. The pool identifier binds the configuration $c$ (member count, denomination, exit structure and fee rate) to the keys,

$$\text{id} = \text{keccak}(\text{"kuroshio v0"} \,\|\, c \,\|\, S_1 \,\|\, \cdots \,\|\, S_n).$$

The $S_i$ are aggregated into $K_{ps} = k_{ps}\,T$ with MuSig key aggregation in context id and each member obtains an $n$-of-$n$ FROST share of $k_{ps}$. Each member then sends a random 32-byte contribution $\rho_i$ to every other member, encrypted pairwise. The pool seed is

$$\text{seed} = \text{keccak}(\text{"kuroshio seed"} \,\|\, \text{id} \,\|\, \rho_1 \,\|\, \cdots \,\|\, \rho_n).$$

Only the members know the seed. Each member publishes keccak("kuroshio seed check" ‖ seed) and stops unless all values agree. A member that sent different contributions to different members is detected here, before any payout address is sealed under a key derived from the seed. All other values of the pool are derived from it with domain labels. The entry address is the Carrot address $(K_s,\ K_v) = (k_{gi}\,G + K_{ps},\ k_v\,G)$ with $k_{gi} = H_s(\text{seed}, \text{"k_gi"})$ and $k_v = H_s(\text{seed}, \text{"k_v"})$. Every wallet can pay it. An entry output paid to it has key $O = K_s + k_g^{\,o}\,G + k_t^{\,o}\,T$, whose spend secrets are

$$x = k_{gi} + k_g^{\,o}, \qquad y = k_{ps} + k_t^{\,o}.$$

The $G$ component $x$ is known to every member. The $T$ component $y$ is shared through $k_{ps}$.

## 4.2 Path-independent states

A *state* is the set $\Sigma$ of members still in the pool. For every state that the exit structure can reach, the pool fixes one output,

$$O_\Sigma = x_\Sigma\,G + K_{ps} + t_\Sigma\,T, \qquad C_\Sigma = m_\Sigma\,G + a_\Sigma\,H,$$

with $x_\Sigma$, $t_\Sigma$ and $m_\Sigma$ derived from the seed and the label of $\Sigma$. The amount $a_\Sigma$ is defined in Section 4.4. Every exit that produces state $\Sigma$ creates exactly $(O_\Sigma, C_\Sigma)$.

On Bitcoin an input names its outpoint, which contains the identifier of the transaction that created the output. The same set of remaining members reached in two different orders is then two different outpoints and every later transaction has to be signed once for each order. An FCMP++ input names no outpoint. Its SAL proof signs a re-randomization of $(O, C)$ and the membership proof shows that $(O, C)$ is in the tree. A set of exits signed once from $\Sigma$ therefore spends $(O_\Sigma, C_\Sigma)$ whichever exit created it. This is the property the rest of the design relies on. A Bitcoin channel factory signs every channel state once per way its funding output can be created. Here each state is signed once.

## 4.3 Exit structures

The exit structure determines which exits are signed from each state. In every structure a state of two members has one exit, which pays both.

- **Any order :** From a state of three or more members, one exit per member, paying it and creating the state without it.
- **Tree :** From a state of three or more members, one exit that splits it into at most $w$ groups of near equal size in member order. Groups of one are paid and larger groups become states.
- **Chain :** From a state of three or more members, one exit paying its first member.

An any-order pool signs one exit per member of every state of size at least three and one exit per pair, about $n\,2^{n-1}$ exits (Figure 1). The analogous Bitcoin construction, joinstr v2 [10], signs the exits of a state again for every order in which the departed members left. Table 1 compares the two. A tree of arity $w$ has one exit per internal node and places every member within about $\log_w n$ exits of the root.

**Table 1 :** Signed exits of an any-order pool.

| Members | States | Exits | joinstr v2 exits |
|---|---|---|---|
| 3 | 4 | 6 | 6 |
| 4 | 11 | 22 | 28 |
| 5 | 26 | 65 | 145 |
| 6 | 57 | 171 | 876 |
| 7 | 120 | 420 | 6 139 |
| 8 | 247 | 988 | 49 120 |
| 10 | 1 013 | 5 065 | 4 420 900 |

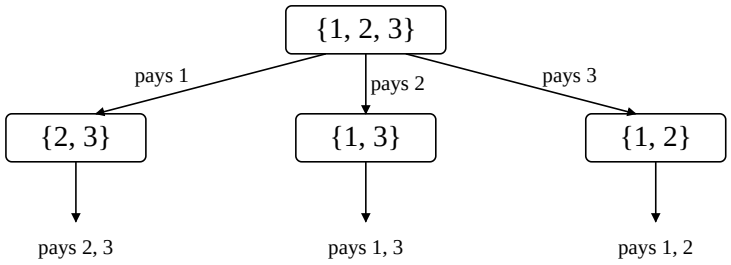


**Figure 1 :** The six exits of a three-member any-order pool. Each box is one output whose key and commitment are fixed by the seed.

### 4.4 Amounts

Let $u$ be the denomination and $f(i, o)$ the fee of a transaction with $i$ inputs and $o$ outputs at the announced rate. The amount of a state is

$$a_\Sigma = u \cdot |P_\Sigma| + \sum_{\Sigma' \in D_\Sigma} a_{\Sigma'} + f(1, o_\Sigma),$$

where $P_\Sigma$ are the payouts, $D_\Sigma$ the child states and $o_\Sigma$ the output count of an exit from $\Sigma$. In every structure all exits from $\Sigma$ have the same output count and children of the same size, so $a_\Sigma$ depends only on $\Sigma$ and not on the exit that created it. Each member pays $\lceil (a_{root} + f(n, 2)) / n \rceil$ and every payout is exactly $u$.

### 4.5 Transactions

The plan of a pool consists of the join, which spends the $n$ entries into the root state, one refund per entry and every exit. A transaction needs at least two outputs, so a transaction with a single output gets an additional zero-amount output. Under Carrot, a transaction with two outputs has a single ephemeral key, which cannot serve payments to two different wallets, so an exit paying two members has three outputs. Payments are Carrot payments to the members' wallets, found by ordinary scanning. State outputs and zero-amount outputs carry random view tags and encrypted amounts and are indistinguishable from payments.

All randomness in the plan, including Carrot anchors, ephemeral keys, dummy outputs and re-randomizations, comes from a ChaCha20 stream seeded with keccak(seed ‖ label) for the label of each transaction. Every member therefore builds byte-identical transactions. Before signing, members exchange a digest over all signable hashes and abort on a mismatch.

**Balancing onto fixed commitments :** Every output mask in the plan is fixed in advance, Carrot payment masks by their derivation and state masks by the seed. Balance requires

$$\sum_{\text{outputs}} m = \sum_{\text{inputs}} (m_{in} + r_c).$$

The last input's $r_c$ is set to satisfy this equation and the others are random. For a single input this reveals nothing, because balance already determines its $\tilde{C}$.

### 4.6 Funding

Members cannot spend their wallet outputs directly in the join, because one FCMP++ membership proof covers all inputs of a transaction and its prover learns which outputs are spent. Each member first pays an entry output to the pool and the join spends the entries. Funding therefore takes three steps.

1. Each member's wallet builds its entry payment to the entry address without broadcasting it and the member sends the transaction and its hash to the others. Each member checks that the entry pays the expected amount to the entry address, that its hash matches and that no output is claimed by two members.
2. The members derive the plan, check its digest and sign every input of every transaction in it. Each input is one FROST session and all sessions run in one batch per round. Each member saves the signed plan.
3. Only then do the wallets broadcast the entries. Once all are mined and spendable, any member broadcasts the join.

A member whose entry has been confirmed for 30 blocks while another is still missing broadcasts its own refund. The join and the refund spend the same entry, so whichever is mined first decides. The presence of the root output on chain shows which one was mined.

### 4.7 Unilateral exit

To exit, a member walks the plan from the root. If the current state's output is not yet spendable it waits. If it is unspent it broadcasts the exit from that state that pays the member, or in a tree or chain the exit leading toward the member. If it is spent it finds the child state that contains the member and continues there. Every step uses only pre-signed transactions. The broadcaster attaches a fresh membership proof against the current tree.

### 4.8 Payout addresses

Exits pay payout positions, not members. The positions are the payout addresses in sorted order. Each member registers a fresh subaddress as its payout, encrypted under a key derived from the seed, signed by a one-time key and sent over a fresh Tor circuit after a random delay. Refunds go to a separate address registered under the member's session key. The members see $n$ payout addresses from $n$ unrelated keys. A member that registers two payouts makes the count exceed $n$ and the pool stops before anything is signed.

## 5 Adaptor signatures for SAL proofs

The SAL proof for $\tilde{O} = O + r_o\, T$ with $O = xG + yT$ contains several equations in which $x$ appears in products with blinding factors, so $x$ cannot be shared linearly between signers. The FROST algorithm for SAL proofs in monero-oxide [2], built on modular-frost [21], leaves $x$ known to all signers and shares only $y$. The only response that involves $y$ is

$$s_y = r_y + e\,(y + r_o), \qquad R_y = r_y\, T,$$

an ordinary Schnorr response with its nonce commitment on $T$. $R_y$ enters the challenge $e$ through the commitments $R_O = \alpha\, G + R_y$ and $R_P = r_z\, U - R_y + r_{r_p}\, T$, where $\alpha$, $r_z$, $r_p$ and $r_{r_p}$ are nonces of the proof that every signer derives from the transcript. A second response, $s_{r_p} = -s_y + r_{r_p} + e\,(r_p - r_{r_i})$, depends on $s_y$.

**Construction 1** (Adaptor SAL) : Let $A = tT$. The signers run FROST for the SAL proof with $R_y$ replaced by $R_y + A$ in every commitment and in the challenge and with $A$ added to the transcript. The aggregated responses are

$$s'_y = s_y - t, \qquad s'_{r_p} = s_{r_p} + t.$$

The pre-signature verifies against $A$ by checking the four verification equations of the SAL proof with $s'_y\, T + A$ in place of $s_y\, T$ and $s'_{r_p}\, T - A$ in place of $s_{r_p}\, T$. Whoever knows $t$ completes it as $s_y = s'_y + t$ and $s_{r_p} = s'_{r_p} - t$, which yields an ordinary SAL proof that the unmodified verifier and the network accept. Anyone holding both the pre-signature and the completed proof extracts $t = s_y - s'_y$. The statements of the proof that do not involve $y$ are unchanged. With $A$ the identity point the construction is ordinary signing.

## 6 Payment channels

A channel between Alice and Bob is funded by an output $F$ under a two-party FROST key $K_c = k_c\, T$, aggregated from the partners' keys in the context ctx = keccak(id ‖ "channel" ‖ $j$) for channel index $j$. The channel seed $\sigma$ = keccak("kuroshio channel seed" ‖ ctx ‖ $\rho_A$ ‖ $\rho_B$) combines a random contribution from each partner, so no third party can compute the channel's outputs or key images. Every output the channel can create has its key and commitment fixed by $\sigma$ and a label, exactly as pool states are. The construction follows tevador [7].

### 6.1 Transactions

State $N$ assigns balances ($a_N$, $b_N$) and a list of pending payments. For each state the partners pick fresh secrets $t_A(N)$ and $t_B(N)$ and reveal only the points $P_A(N) = t_A(N)\, T$ and $P_B(N) = t_B(N)\, T$. Let $K_N$ be the close output of state $N$. Its key and mask are fixed by $\sigma$, $N$ and a hash of the balances and pending payments of the state, so transactions signed for one attempt at state $N$ spend nothing of another attempt. The outputs of a pending payment are bound to its point, direction, amount and stage count in the same way.

**Table 2 :** Transactions of a channel in state $N$.

| Transaction | Spends | Pays | Signature |
|---|---|---|---|
| Close($N$) | $F$ | $K_N$ | pre-signed twice, against $P_A(N)$ and against $P_B(N)$ |
| Recover($N$) | $K_N$ | $a_N$, $b_N$ and one output per pending payment | signed, relatively locked |
| PunishA($N$) | $K_{N-1}$ | everything to Bob | pre-signed against $P_A(N-1)$ |
| PunishB($N$) | $K_{N-1}$ | everything to Alice | pre-signed against $P_B(N-1)$ |
| Withdraw | $F$ | final balances | signed only at a cooperative close |

Close($N$) is one transaction with two pre-signatures (Table 2, Figure 2). Alice holds her own secret, so only she can complete her version and completing it publishes $t_A(N)$ in its SAL proof. Both versions create the identical output $K_N$, so a single Recover($N$) follows either. Revoking a state sends no secret. After the update to state $N$, Bob holds PunishA($N$). If Alice later broadcasts Close($N-1$), Bob extracts $t_A(N-1)$ from the chain, completes PunishA($N$) and takes the whole channel, while Recover($N-1$) is still locked for $L$ blocks.

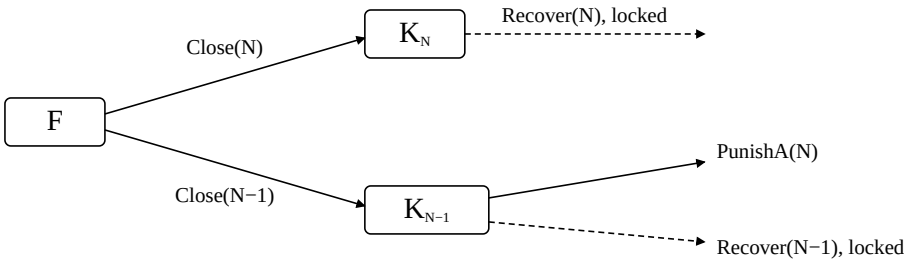


**Figure 2 :** Transaction graph of a channel. Dashed edges are relatively locked. A close in the old state $N-1$ can be punished before Recover($N-1$) unlocks.

### 6.2 Updates

Every update after the opening is an operation that makes exactly one side worse off. That side proposes it and the other side accepts or refuses. Table 3 lists the operations. The update to state $N$ signs Close($N$) twice, Recover($N$), PunishA($N$) and PunishB($N$), plus a claim and a refund for every stage of every pending payment (Section 8). The partners exchange FROST nonce commitments for all of these at once and then release response shares in a fixed order (Table 4). In two-party FROST, whoever receives the last share of a transaction can complete it while withholding its own, so the order determines what each side can hold at any moment.

**Table 3 :** Channel operations and who proposes them.

| Operation | Proposed by |
|---|---|
| open with Alice's share $a_0$ | Alice, accepted only if $a_0$ is the agreed opening |
| pay | the payer |
| add a pending payment | its sender |
| settle a pending payment | its sender, once the receiver has revealed the secret |
| cancel a pending payment | its receiver |

**Table 4 :** Message order of an update from state $N-1$ to state $N$. I is the initiator, R the responder.

| # | From | Content | Lets the receiver complete |
|---|---|---|---|
| 1 | I | operation, $P_I(N)$ | |
| 2 | R | $P_R(N)$, digest of state $N$, commitments | |
| 3 | I | commitments, digest, shares of claims, refunds and Recover($N$) | claims, refunds, Recover($N$) |
| 4 | R | shares of claims, refunds, Recover($N$) and Close($N$) | claims, refunds, its version of Close($N$), Recover($N$) |
| 5 | I | shares of Close($N$) and of the punishment of its own old close | its version of Close($N$), Punish$_I$($N$) |
| 6 | R | share of the punishment of its own old close | Punish$_R$($N$) |

The initiator commits to state $N$ after message 4 and the responder after message 5. Section 10 checks every stopping point. From message 3 on the responder holds Recover($N$) and the claims and refunds of state $N$. If the update stops after message 4, the responder keeps them, watches for $K_N$ and follows a close in state $N$ like a close in its latest state. If the update fails after message 4, the responder signs nothing else for the channel, which can then only be closed. A message that does not belong to the running update is refused and changes nothing. If message 6 is lost, the initiator waits for it and proposes nothing. The responder repeats its shares in its next proposal and the initiator refuses any proposal without them. An update proposed by the responder

makes the responder worse off, so state $N-1$ can become better for it than the latest state and only message 6 punishes its close in $N-1$. If both sides propose at once, Alice's proposal proceeds and Bob proposes again afterwards.

### 6.3 Closing

Both partners can close cooperatively with Withdraw, which pays the final balances plus an equal split of the fees saved against the force-close path. In a force close, one partner broadcasts its version of Close($N$) and, after $L$ blocks, Recover($N$). A close in an old state is answered with the matching punishment.

## 7 Channel factories

A factory is a pool whose payouts are channel funding outputs instead of wallet addresses [9]. A payout is either Wallet($K_s$, $K_v$) or Output($O$, $m$) for an output with a fixed key and mask. For each channel the two partners derive $F$ from the channel seed, both announce it to all members and the members require the two announcements to agree before the pool pays into it. Each member's entry is its share of every channel it belongs to plus an equal share of the pool's fees, so every channel is funded by both partners. The opening state of every channel is signed before any entry is broadcast. Because pool states are path independent, a funding output has the same key and commitment whichever exits create it and each channel state is signed once.

Payments do not touch the pool. In the normal case no channel funding output ever appears on chain.

- **Settle :** At the end of a round each member waits until its channels have no update running and no pending payment, reports their latest balances and accepts no further update. Partners must report the same state. If they do not, the members proceed as with an absent member. Every member builds the same settle transaction, which spends the current pool state and pays each member its total over its channels. The members sign it in one FROST session. Fees saved by not exiting are split evenly among the members involved.
- **Refresh :** Instead of settling, the members can move into a new pool with new channel pairs. Each member's total, plus an equal share of what the pool state holds beyond the channels after the refresh fee and minus an equal share of the new pool's exit fees, is split evenly over its new channels. All members compute the same split. The partners of every new channel sign its opening state, the members sign every exit of the new pool and only then sign the refresh transaction that moves the old state into the new root. Messages carry a round number so the rounds do not interfere.
- **Unilateral exit of a channel :** Either partner broadcasts the pool exits leading to the channel's funding output, after which the partners close the channel cooperatively or by force close. The remaining members settle the state that remains. With the any-order structure such a state exists for every set of two or more channels. When only one channel would remain, the exits pay it out as well and its partners close it cooperatively.
- **Absent member :** Settle and refresh require every member. If the end of a round waits longer than a configured period, every member exits unilaterally with its own channels using the pre-signed exits and closes each with its partner. If a partner does not answer within two minutes, the member force-closes in the latest state, so an absent member still receives its balance through Recover.

## 8 Multi-hop payments

### 8.1 Pending payments and lock multiples

A pending payment of amount $v$ from $X$ to $Y$ against point $Q$ is an output of Recover with a refund chain of $k$ stages. Stage 0 is created by Recover($N$). For $0 \le s < k - 1$ the refund of stage $s$ creates stage $s + 1$ and the refund of the last stage pays $X$. Every refund is relatively locked. At every stage a claim, pre-signed against $Q$, pays $v$ to $Y$. Each stage holds $v$ plus the fees of the remaining refunds. The refund of a payment with $k$ stages therefore reaches $X$ no earlier than $(k + 1)\ L$ blocks after the channel's close, counting the lock of Recover, while the claim is available from the moment stage 0 exists.

Lightning gives each router time to claim upstream by making each HTLC on a route expire later than the next one. The relative lock has one fixed length, so Kuroshio counts deadlines in whole multiples of it. For a route of $h$ channels from the payer, channel $j$ for $0 \le j < h$ carries a pending payment with

$$k_j = h - j + \varepsilon$$

stages, where $\varepsilon$ is drawn uniformly from $\{0, \ldots, \min(3, 8 - h)\}$ by the payer and 8 is the maximum number of stages. Each router keeps a flat fee $\varphi$, so channel $j$ carries $v + (h - 1 - j)\varphi$. A router accepts an incoming payment only if it carries exactly one stage more than the outgoing one it is asked to add and exactly its fee more. The extra $\varepsilon$ hides from a router how many channels remain, up to three (Figure 3). An outgoing payment with a single stage is always the last channel, which occurs when $\varepsilon = 0$.

### 8.2 Decorrelated points

Using one payment point along a route would let two routers recognize the same payment. Kuroshio uses per-hop tweaks in the manner of anonymous multi-hop locks [11]. The receiver picks $t$ and gives the payer $P = tT$. The payer picks random $d_0, \ldots, d_{h-1}$ and sets

$$Q_{h-1} = P + d_{h-1}\,T, \qquad Q_j = Q_{j+1} + d_j\,T \qquad (0 \le j < h - 1).$$

Channel $j$ is locked to $Q_j$. Router $j+1$ receives $Q_j$ and $d_j$ and forwards on $Q_{j+1} = Q_j - d_j\,T$. The receiver claims $Q_{h-1}$ with $t + d_{h-1}$. A router that learns the secret $q$ of its outgoing point obtains its incoming secret as $q + d_j$. The payer ends with $q_0$ and recovers $t = q_0 - \sum_j d_j$, which matches $P$ and serves as proof of payment. Without the tweaks between them, the points of two channels are independent uniform group elements.

### 8.3 Onion routing

Forwarding instructions travel in a Sphinx packet [12], laid out as in BOLT 4 [13] but over Ristretto. Each member publishes an onion key. A packet consists of a 32-byte ephemeral point, eight slots of 96 bytes and a 32-byte MAC, 832 bytes in total. A slot holds the next member, the amount to forward, the outgoing stage count, the point tweak $d_j$ and a MAC for the next layer. Each hop derives a shared secret from the ephemeral point, checks the MAC, decrypts the routing information with a ChaCha20 keystream, shifts it by one slot, fills the end with keystream bytes and blinds the ephemeral point for the next hop. The MAC of each layer covers the point $Q_j$ of the pending payment it travels with, so a router cannot attach a layer to a different payment. Every packet has the same size at every position. Each hop records a replay identifier derived from its shared secret and rejects repeats.

The payer obtains $P$ by sending an invoice request under a one-time key over a fresh circuit, so the receiver does not learn who pays. If a router refuses to forward, a channel lacks balance or the receiver has no such invoice, the pending payment is cancelled hop by hop off chain and every sender recovers its funds. A router

withdraws a forwarded payment that its next partner has not accepted within 30 seconds and cancels the incoming one, because a partner that stalls could otherwise accept the payment after the incoming one was refunded.

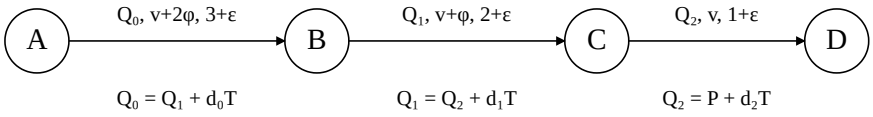


**Figure 3 :** A payment from A to D over three channels. Each channel is locked to its own point and carries one stage more than the next.

## 9 On-chain enforcement

### 9.1 Stuck payments

A member exits a channel unilaterally and force-closes it when a pending payment does not progress. A receiver that has revealed the secret and is not paid within a fixed period claims on chain. A sender whose payment is neither settled nor cancelled within a period proportional to its stage count plus one refunds on chain. Routers upstream hold more stages, so they wait longer than the routers after them. The member announces the exit to the other members, broadcasts the pool exits for the channel, broadcasts its version of the latest close and, after the lock, Recover. Each pending payment is then claimed with its secret or refunded stage by stage. A sender that observes a claim reads the secret from its SAL proof. A payer obtains its proof of payment this way and a router settles its incoming payment off chain. A router whose outgoing payment is refunded cancels its incoming one.

Independently of payments, a running member watches the chain for a spend of the funding output of every channel it is in. A close is the transaction that spends the funding output and creates the close output of some state. An output that only carries the key of a close output is ignored. Any close, announced or not, is handled the same way. A close in an old state is punished and a close in the latest state is recovered.

### 9.2 Watchtowers

A member that may be offline for longer than $L$ blocks hands its punishments to a watchtower. For each old state $N$ of the partner the member holds Punish($N+1$) with its pre-signature, the partner's pre-signature of Close($N$) and the partner's point. Completing the punishment requires the partner's secret, which only a close in state $N$ reveals, so the tower cannot spend anything on its own.

Before a breach the tower should also learn nothing about the channel. The close output $K_N$ of state $N$ is known to the member in advance and appears on chain only if state $N$ is closed. The member files the punishment under

$$\text{hint} = \text{keccak}(\text{"kuroshio tower hint"} \,\|\, K_N)_{[0,\,16)}, \qquad \kappa = \text{keccak}(\text{"kuroshio tower key"} \,\|\, K_N),$$

where $K_N$ stands for the key of the close output. The blob is encrypted under κ with ChaCha20-Poly1305. The tower computes the hint of every new output on chain. When one matches, it decrypts the blob with the key derived from the output and tries to extract the closer's secret from the transaction. If that succeeds it broadcasts the punishment. Otherwise the output was not created by the partner's close and the tower keeps watching. This follows the practice of Lightning watchtowers that encrypt justice transactions under the breach transaction identifier. Monero's transaction identifiers are not known in advance, because the membership proof is created at broadcast time, so Kuroshio keys the blob on the close output instead.

## 10 Security

This section argues the properties of Section 3 informally, assuming unforgeability of SAL proofs, the adaptor properties of Construction 1 and the liveness assumption that an honest party can get a transaction mined within $L$ blocks.

**Proposition 1** (Pool exit) : *After the join is mined, every member can obtain its payout using only transactions signed before it paid in.*

*Argument :* Every exit in the plan is fully signed. Its inputs are state outputs whose keys and commitments are fixed, so the SAL proofs remain valid whichever exit created them. The membership proof is not signed and is computed by whoever broadcasts, against the current tree. The fee was fixed from the weight, which does not depend on the tree depth. In every structure each state containing the member has an exit that either pays the member or leads to a smaller state containing it, so the walk of Section 4.7 terminates with the member's payout. □

**Proposition 2** (Pool funds) : *No set of fewer than n members can spend a pool output in any way other than the transactions of the plan.*

*Argument :* Spending requires a SAL proof for $y = k_{ps} + t_{\Sigma}$. The share of $k_{ps}$ held by an honest member never leaves it and it releases a response share only for an input of the plan whose digest it checked. Every transaction of the plan pays fixed amounts to fixed outputs. □

**Proposition 3** (Funding) : *Either the join is mined or every member that paid in can recover its entry minus one fee.*

*Argument :* A member broadcasts its entry only after its refund is signed. The refund spends only that entry, so it stays valid as long as the join is not mined. □

**Proposition 4** (Update safety) : *If a party stops an update from state $N-1$ to state $N$ at any message, its partner can still obtain at least the balance of whichever of the two states is worse for the partner and each party can obtain at least its balance in the latest state it committed to.*

*Argument :* By the proposer rule state $N$ is no worse than state $N-1$ for the responder and no better for the initiator. Messages 1 to 3 transfer no completable close, so both remain in state $N-1$. Claims, refunds and Recover($N$) released in message 3 spend outputs that exist only after Close($N$), which nobody can yet complete. After message 4 the initiator can complete its version of Close($N$) and Recover($N$). If it closes in state $N$, the responder broadcasts Recover($N$), which it has held since message 3. This pays its balance of state $N$. The responder still holds $\text{Punish}_I(N-1)$ against any older close of the initiator and can close in $N-1$, which is worse for the responder only. It signs no other contents for state $N$. The close output of state $N$ is also bound to its contents, so nothing signed for this attempt can be combined with a later one. After message 5 the responder holds its version of Close($N$) and $\text{Punish}_I(N)$, which punishes a close by the initiator in state $N-1$. The initiator lacks $\text{Punish}_R(N)$ until message 6, but a close by the responder in $N-1$ only benefits the initiator. Refunds of pending payments are completable by their senders before the other side can complete Close($N$). □

**Proposition 5** (Punishment) : *Let N be the latest state both parties committed to. If a party broadcasts* Close*(M) for $M < N$, its partner can take the whole channel minus the fees of the close and the punishment. The one exception is $M = N - 1$ with the closer being the responder of the update to state N and message 6 of that update not delivered. The partner then receives its balance of state $N - 1$, which is at least its balance of state N.*

*Argument :* If the closer initiated the update to state $M + 1$, the partner received Punish($M+1$) in message 5, before it committed to that state. If the closer was its responder, the partner received Punish($M+1$) in message 6 or in the closer's next proposal, because an initiator that misses message 6 proposes nothing and accepts no proposal without it. Only the update to state $N$ can lack it. The broadcast close reveals the closer's secret of state $M$ by extraction. The closer's only alternative spend of $K_M$ is Recover($M$), which is locked for $L$ blocks, during which the punishment can be mined. □

**Proposition 6** (Routing) : *Let $\delta$ be the total number of blocks by which the transactions a router broadcasts for one payment are mined after they become valid. If $\delta < L$, a router whose outgoing pending payment is claimed can claim its incoming one before its refund completes. A payer whose payment is claimed obtains $t$.*

*Argument :* A claim of the outgoing payment on $Q_{j+1}$ reveals $q$ with $qT = Q_{j+1}$ by extraction and $q + d_j$ is the secret of $Q_j$. Off chain, the router learns $q$ before it settles its outgoing payment. On chain, the outgoing claim must precede the final refund of the outgoing payment, which the router gets mined at most $k_{j+1} + 1$ locks and $\delta$ blocks after the outgoing channel closes. The incoming refund completes no earlier than $k_j + 1 = k_{j+1} + 2$ locks after the incoming channel closes. If the outgoing channel closes first, the router therefore has at least $L - \delta$ blocks after the latest possible outgoing claim. If the incoming channel closes first, the router closes the outgoing channel at once and the blocks until that close is mined count toward $\delta$. A claim at the payer's first hop reveals $q_0$ and hence $t = q_0 - \sum_j d_j$. □

**Proposition 7** (Watchtower) : *A watchtower can broadcast only punishments. Until an output with the key of an old close output appears on chain it learns only the number of stored entries and when they were added.*

*Argument :* Each punishment completes only with a closer's secret for an old state. Each blob is encrypted under a key derived from a close output key that is unknown until the close is mined and indexed by an independent hash of the same key. Only the two partners know that key. □

## 11 Privacy

**Chain observer :** Entries are ordinary wallet transactions. The join is an $n$-input transaction, each exit and settlement a one-input transaction with two or more outputs. All outputs, including state, close and pending-payment outputs, are Carrot-shaped with encrypted amounts. With FCMP++ the output spent by each input is hidden among all outputs, so exits cannot be linked to each other or to the join. Relatively locked transactions are recognizable through their unlock field and appear only when a channel is force closed.

**Members :** Members see the entry transaction of each member but not which outputs it spends. They learn the set of payout addresses but not their owners. They also learn the pool's outputs and key images. Channel balances and close outputs are known only to the two partners until a settlement or refresh, in which each member reports the state number and the balances of its channels to all members.

**Relay :** All messages are NIP-04 encrypted to the pool key [20]. From the seed check on they are also sealed under keys derived from the seed. Messages between two members are sealed under keccak("kuroshio pair" ‖ ECDH ‖ seed) and are indistinguishable from pool messages. Every connection to a relay or daemon not on the local host goes through Tor, with one circuit per session key and one fresh circuit per one-time key. A pool can use several relays at once and survives the loss of all but one.

**Routers :** A router learns its predecessor, its successor, the amount it forwards and the stage count of its incoming payment. It does not learn the payer. The extra stages hide its distance to the receiver by up to three channels, except that an outgoing payment with a single stage identifies its successor as the receiver. Amounts and timing can still correlate the hops of one payment for colluding routers.

## 12 Implementation and evaluation

The implementation is written in Rust, about 10 600 lines, on monero-oxide [2] and modular-frost [21], with 147 unit and integration tests. Members are separate processes that communicate over nostr relays through Tor. The tests check transaction serialization, the signable hash and the weight function against transactions built by the reference wallet. Table 5 lists transaction weights and Table 6 the scenarios, which ran on regtest networks of the FCMP++ beta (v0.19.0.0-beta.3.1) and of a daemon built from the relative lock change [8].

**Table 5 :** Transaction weights. The fee of a transaction is its weight times the fee rate per byte, rounded up to a multiple of the fee quantization. The quantization was 10 000 piconero in all runs. The rate was 18 000 piconero on the FCMP++ beta daemon and between 4.2 and 4.6 million piconero on the daemon with the relative lock, which estimates higher fees on regtest.

| Transaction | Inputs | Outputs | Weight |
|---|---|---|---|
| exit paying one member, close, punish, claim, refund | 1 | 2 | 7 650 |
| exit paying two members, withdraw, recover with at most one pending payment | 1 | 3 | 7 872 |
| settle of four members, recover with two pending payments | 1 | 4 | 7 998 |
| settle of sixteen members | 1 | 16 | 9 626 |
| join of three members | 3 | 2 | 9 768 |
| join of four members | 4 | 2 | 11 851 |
| join of eight members | 8 | 2 | 17 239 |

**Signing cost :** Each signed input is one FROST session in which each signer sends 64 bytes of nonce commitments and a 32-byte share. An any-order pool of ten members signs 5 085 inputs, so each member sends about 490 KB before encoding. A channel update with no pending payments produces five signatures over four transactions, the two pre-signatures of Close($N$), the signature of Recover($N$) and the pre-signatures of the two punishments. Each pending payment with $k$ stages adds $k$ pre-signed claims and $k$ signed refunds.

**Table 6 :** Scenarios run between separate member processes on regtest, in a factory of four members with channels Alice-Bob, Bob-Carol and Carol-Dave of 2 XMR each.

| Scenario | Outcome |
|---|---|
| direct and routed payments, settle | one settle transaction pays every member its exact total; routed payments over two and three channels return a valid proof of payment |
| refresh into channels Alice-Carol and Bob-Dave | new pool signed before the move, payment in a new channel, settle |
| cooperative close of one channel | one pool exit releases the channel, the partners close it cooperatively and the remaining state settles |
| close in an old state | partner extracts the secret and receives 1.9997246 XMR of the 2 XMR channel |
| force close | Recover accepted after the lock, both partners receive their latest balances |
| sender withholds settlement | receiver claims 0.25 XMR on chain, payer obtains the proof of payment from the claim |
| receiver withholds the secret | sender refunds 0.25 XMR on chain after the stage locks |
| watchtower | tower decrypts the punishment when the old close appears and broadcasts it while the member is offline |
| member absent at settlement | the other members exit unilaterally with their channels and close them; the absent member receives its balance through Recover |
| pool member crashes after the join | exits unilaterally with its saved plan |

## 13 Limitations

- The relative lock is not part of Monero consensus. Without it, force closes and refunds of pending payments cannot be enforced, although cooperative closes, settlement and punishment work.
- Fees are fixed when transactions are signed. A Monero output cannot be spent before it has 10 confirmations, so there is no child-pays-for-parent and a pre-signed transaction cannot be fee-bumped.
- Settlement and refresh need every member online. If one is absent, the others exit unilaterally with their channels one by one, at a cost of roughly one exit and one close per channel instead of one transaction.
- Exits in a tree or chain structure also pay out other members and any-order pools are limited to about ten members by signing cost.
- A member that registers a second payout stops the pool before anything is signed and cannot be identified. Linkable ring signatures on payout registration, which joinstr and joinstr v2 use to close denial of service vectors, would reject the second registration without identifying the member.
- A stuck payment triggers a force close after a fixed period of time rather than a number of blocks.
- An update that fails after the responder released its close share leaves the channel usable only for closing.
- The pool key is given to whoever asks for it, so an outsider can register and stall the setup of a pool. No funds are at risk before the plan is signed.
- A factory member keeps channel state in memory. After a crash its partners close its channels in the latest state and its watchtower handles old closes, but it cannot rejoin the factory.
- Payments to Bitcoin or Lightning require a proof that one secret underlies points on both curves, as in atomic swaps between Bitcoin and Monero [19], or a bridge from hash locks to point locks.

## 14 Related work

The Lightning Network [1] introduced revocable channels with hash time-locked contracts. Channel factories [9] place many channels in one funding output. CoinPool [17] is a payment pool design for Bitcoin and joinstr v2 [10] implements payment pools with pre-signed exits. Kuroshio's exit structures follow joinstr v2 and path independence reduces their signing cost. Anonymous multi-hop locks [11] replace hash locks with decorrelated point locks. Sphinx [12], as used by BOLT 4 [13], hides routes. Pisa [18] studies accountable watchtowers and Sleepy Channels [16] avoid watchtowers by confining disputes to a short predetermined window.

For Monero, DLSAG [14] proposes a new linkable ring signature that enables refund transactions, which requires a hard fork. PayMo [15] builds channels from verifiable timed linkable ring signatures, with a bounded channel lifetime. AuxChannel [22] commits to each payment with verifiably encrypted signatures and resolves disputes through a key escrow service. MoNet [23] builds a payment channel network for Monero on the same approach. These designs predate FCMP++. Kuroshio instead relies on FCMP++ output re-randomization and the proposed relative lock [7]. It needs no third party, places no bound on channel lifetime and adds pools and channel factories.

## 15 Conclusion

FCMP++ separates what an input proves from which output it spends. This makes pools with pre-signed, path-independent exits possible on Monero. With a relative lock the same structure supports punishable channels, channel factories and routed payments without trusted parties. The construction is implemented and runs end to end on regtest networks. Pools need only FCMP++, while channels also need the relative lock to become part of consensus.

## References


[1] J. Poon and T. Dryja. The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments. 2016.

[2] L. Parker. Full-Chain Membership Proofs++ (FCMP++). 2024. Implementation in monero-oxide, https://github.com/monero-oxide/monero-oxide.

[3] jeffro256. Carrot: Cryptonote Address For Rerandomizable-RingCT-Output Transactions. 2024.

[4] C. Komlo and I. Goldberg. FROST: Flexible Round-Optimized Schnorr Threshold Signatures. In *Selected Areas in Cryptography*, 2020.

[5] G. Maxwell, A. Poelstra, Y. Seurin and P. Wuille. Simple Schnorr Multi-Signatures with Applications to Bitcoin. *Designs, Codes and Cryptography*, 87(9), 2019.

[6] A. Poelstra. Scriptless Scripts. Presentation, 2017.

[7] tevador. Relative locks with FCMP++. Monero research-lab, issue 161. https://github.com/monero-project/research-lab/issues/161.

[8] Relative locks. Pull request 507, https://github.com/seraphis-migration/monero/pull/507.

[9] C. Burchert, C. Decker and R. Wattenhofer. Scalable Funding of Bitcoin Micropayment Channel Networks. *Royal Society Open Science*, 5(8), 2018.

[10] floppy. joinstr v2: MuSig2 payment pools with pre-signed exits. https://gitlab.com/1440000bytes/joinstr-v2.

[11] G. Malavolta, P. Moreno-Sanchez, C. Schneidewind, A. Kate and M. Maffei. Anonymous Multi-Hop Locks for Blockchain Scalability and Interoperability. In *NDSS*, 2019.

[12] G. Danezis and I. Goldberg. Sphinx: A Compact and Provably Secure Mix Format. In *IEEE Symposium on Security and Privacy*, 2009.

[13] BOLT 4: Onion Routing Protocol. Lightning Network specifications. https://github.com/lightning/bolts.

[14] P. Moreno-Sanchez, A. Blue, D. V. Le, S. Noether, B. Goodell and A. Kate. DLSAG: Non-Interactive Refund Transactions for Interoperable Payment Channels in Monero. In *Financial Cryptography and Data Security*, 2020.

[15] S. A. K. Thyagarajan, G. Malavolta, F. Schmidt and D. Schröder. PayMo: Payment Channels For Monero. Cryptology ePrint Archive, 2020/1441.

[16] L. Aumayr, S. A. K. Thyagarajan, G. Malavolta, P. Moreno-Sanchez and M. Maffei. Sleepy Channels: Bi-directional Payment Channels without Watchtowers. In *ACM CCS*, 2022.

[17] G. Naumenko and A. Riard. CoinPool: Efficient Off-Chain Payment Pools for Bitcoin. 2020.

[18] P. McCorry, S. Bakshi, I. Bentov, S. Meiklejohn and A. Miller. Pisa: Arbitration Outsourcing for State Channels. In *ACM Advances in Financial Technologies*, 2019.

[19] J. Gugger. Bitcoin-Monero Cross-chain Atomic Swap. Cryptology ePrint Archive, 2020/1126.

[20] Nostr implementation possibilities, NIP-01, NIP-04 and NIP-42. https://github.com/nostr-protocol/nips.

[21] modular-frost. Serai. https://github.com/serai-dex/serai.

[22] Z. Sui, J. K. Liu, J. Yu, M. H. Au and J. Liu. AuxChannel: Enabling Efficient Bi-directional Channel for Scriptless Blockchains. In *ACM Asia Conference on Computer and Communications Security*, 2022.

[23] Z. Sui, J. K. Liu, J. Yu and X. Qin. MoNet: A Fast Payment Channel Network for Scriptless Cryptocurrency Monero. In *IEEE International Conference on Distributed Computing Systems*, 2022.