Grease: A Private Payment Channel Protocol for Monero

Grease Team

July 21, 2025

Contents

1. Introduction

Monero is, alongside cash, the world's most private[1–3], and, arguably the best currency in circulation, but the user *experience* remains less than ideal. This comment is not necessarily aimed at user *interfaces* – for example, there are Monero wallets that are very attractive and easy to use – but the fundamental design of Monero means that:

- many, especially new, users find they can make only one payment roughly every 20 minutes when their wallet holds a single spendable output (the change from the first payment typically requires 10 confirmations before it can be spent again),
- due to the lack of scripting capabilities, use-cases that capture the public imagination, like DeFi, are not possible in vanilla Monero.

Therefore, the *experience* of using Monero tends to be one of waiting, and limited functionality.

For Monero to achieve mass adoption, it will need to find ways to:

- provide an order of magnitude *better UX* (again, not necessarily UI). Locking UTXOs after spending and block

confirmation times add significant friction to Monero and is a turn-off for new users who are already unsure about how cryptocurrency works.
- provide *instant confirmations* when purchasing with Monero.
- enable *seamless point-of-sale transactions* so that using Monero for purchases feels no different to using a credit card or Venmo.
- enable DeFi for Monero. DeFi is the future of finance. The lack of permissionless access to bank-like services (loans, insurance, and investments) is a key barrier to truly democratic money.
- provide for Monero-backed and/or privacy-maximizing stable coins.

A payment-channel solution for Monero is one of the foundational requirements for achieving these goals in Monero. The other is smart contracting functionality, but that is out of scope for this project.

1.1. Payment channels in Monero

Monero's primary function is private, fungible money. This goal very likely excludes any kind of meaningful on-chain state management for Monero, since state implies heterogeneity. And heterogeneity immediately breaks fungibility. That's not to say that some hitherto undiscovered insight won't allow this in future, but for the short and medium-term at least, any kind of state management for Monero transactions or UTXOs would have to be stored off-chain.

It makes the most sense to store this off-chain state on another decentralized, private protocol. Zero-knowledge Rollup blockchains (ZKR) fit the bill nicely.

It's the goal of this project to marry Monero (for private money) with a ZK-rollup chain (for private state management) to create a proof-of-concept Monero payment channel for Monero.

1.1.1. Enter Grease

The Grease protocol is a new bi-directional payment channel design with unlimited lifetime for Monero. It is fully compatible with the current Monero implementation and is also fully compatible with the upcoming FCMP++ update.

Using the Grease protocol, two peers may trustlessly cooperate to share, divide and reclaim a common locked amount of Monero XMR while minimizing the online transaction costs and with minimal use of outside trusted third parties.

The Grease protocol maintains all of Monero's security. No identifiable information about the peers' privately owned Monero wallets are shared between the peers. This means that there is no way that privacy can be compromised. Each channel lifecycle requires two Monero transactions, with effectively unlimited near-instant updates to the channel balance in between these two transactions. This dramatically improves the scalability of Monero.

The Grease protocol is based on the original AuxChannel[4] paper and MoNet[5] protocol. These papers introduced new cryptography primitives that are useful for trustlessly proving conformity by untrusted peers. These primitives are useful abstractly, but the means of implementation were based on innovative and non-standard cryptographic methods that have not gained the general acceptance of the cryptographic community. This may change in time, while the Grease protocol bypasses this limitation by the use of generally accepted methods for the primitives' implementation.

Every update and the final closure of the channel require an online interaction over the Grease network. In order to prevent the accidental or intentional violation of the protocol by a peer not interacting and thus jamming the channel closure, Grease introduces an external Key Escrow Service (KES). The KES needs to run on a stateful, logic- and time-aware platform. A decentralized zero-knowledge smart contract platform satisfies this requirement while also providing the privacy-focused ethos familiar to the Monero community.

1.1.2. Why does another chain have to be involved?

Offline payment channels necessarily *require* a trustless state management mechanism. Typically, the scripting features for a given blockchain allow for this state to be managed directly. However, Monero's primary design goals are privacy and fungibility. Attaching state to UTXOs would create a heterogeneity that threatens these goals. (Fungibility is more important than specialty for maintaining privacy.)

The state does not have to be managed on the same chain though. Any place where the state is:

- available,
- reliable and verifiable,
- trustless,

will suffice.

The initial implementation uses the any Noir-compatible execution environment that supports the Barretenberg Plonky proving system, the Aztec blockchain being one candidate.

The KES acts as a third-party judge in disputes. At initialization, each peer splits a secret using a 2-of-2 scheme and encrypts one share for the counterparty and one for the KES. Any single share is useless on its own. If a dispute arises, the KES identifies the violating peer and releases its share of that peer's secret to the wronged peer. Combined with the counterparty-held share already in their possession, the wronged peer can reconstruct the secret and simulate the missing online interaction to close the channel with the latest agreed balance. Only valid channel states can be unilaterally closed; fabricated updates cannot be simulated.

2. Design principles

Grease is a bidirectional two-party payment channel. This means that funds can flow in both directions, but in the vast majority of cases, funds will flow from one party (the client, or private peer) to the other (the merchant, or public peer).

Grease embraces this use case and optimizes the design and UX based on the following assumptions:

- The public peer is responsible for recording the channel state on the ZK chain.
- The public peer pays for gas fees on the ZK chain and will need to have some amount of ZK chain tokens to pay for these fees.
- The public peer will be able to recover gas fees from the client peer.
- The client peer does not *have* to have any ZK chain tokens, but will need to hold Monero to open the channel.
- The client peer will need ZK chain tokens if they want to dispute a channel closure. In the vast majority of cases, this won't be necessary, since funds almost always flow in one direction from the client to the merchant. However, in instances where this is not the case, the client is able to dispute the channel closure by watching the ZK chain and proving that the channel was closed with outdated state.
- In the vast majority of cases, the client opens a channel with m XMR and the public peer starts with a zero XMR balance (since the public peer is providing assets or services and not monetary value).
- Usually, both parties mutually close the channel. Either party *may* force close the channel, and are able to claim their funds after a predetermined time-out. In this case, the forcing party is usually the merchant since they have the greater incentive to do so in the case where a channel has been abandoned by the client.

2.1. Anti-principles

The following design goals are explicitly *excluded* from the Grease design:

- Multi-hop channels. Multi-hop channels are probably *possible* in Grease, but they are not a design goal.

Taking the Lightning Network as the case study, CJ argues[6] that the vast majority of the utility of lightning is captured by bilateral channels, with a tiny fraction of the complexity.

3. The Grease Channel Lifecycle

3.1. Overall design description

Grease largely follows the MoNet[5] design, which is a payment channel protocol that uses a key escrow service (KES) to manage the funds in the channel.

A grease payment channel is a 2-party bidirectional channel. The most common use case is in a multi-payment arrangement between a customer and a merchant, and so we will label the parties as such.

To set up a new channel, the customer and merchant agree on the funds to be locked in the channel. It's usually all paid by the customer, but it doesn't need to be. These funds are sent to a new 2-of-2 multisig wallet, which is created on the Monero blockchain for the sole purpose of serving the channel.

The idea is that the transaction (called the *commitment transaction* for reasons that will be made clear later) that spends the funds out of the multisig wallet back to the customer and merchant can be trustlessly, securely and rapidly updated many thousands of times by the customer and merchant without having to go on-chain.

Every time the channel is updated, the customer and merchant provide signatures that can't be used to spend the funds out of the multisig wallet, but prove that, minus a small piece of data, will be able to spend the funds if that data were provided. When the channel is closed, the customer gives the merchant that little piece of data and the funds are spent out of the multisig wallet to the customer and merchant, closing the channel.

If the merchant cheats and tries to close the channel with an outdated state, or decides not to broadcast the commitment transaction, the customer can dispute the closure of the channel with the key escrow service (KES).

The KES is a smart contract on the ZK chain that is responsible for arbitrating disputes. It won't be called upon for the vast majority of channel instances, but its presence it mandatory to disincentivize cheating.

> **Note**
>
> In fact, you could run Grease without a KES, if there is a high-trust relationship between the customer and merchant.

When the 2-of-2 multisig wallet is created, both the customer and merchant split their spending keys into two encrypted pieces. They give one piece to their counterparty, and the other piece they give to the KES. The splitting is done in a way that is

- verifiable - even though the secrets are encrypted, we can prove that the pieces form the original spending key.
- secure - the pieces are encrypted to the public keys of the KES and counterparty respectively, so they can be publicly communicated without worrying about the spending key being reconstructed by a malicious party.

If, say, the merchant tries to force-close a channel using an outdated state (which is itself enacting the dispute process), or refuses to publish anything at all (in which case the customer will enact the force-close process), the customer has a certain window in which it can prove to the KES that it has a valid, more recent channel state signature.

In a successful dispute, either by waiting for the challenge period to end, or the KES accepts the challenge, the KES will hand over the merchant's secret share and the customer will be able to reconstruct the spending key and spend the funds out of the multisig wallet. In this case, the customer can then spend any state from the entire history of the channel, including any states that favour the customer. This is a form of punishment that should motivate parties to behave honestly.

> ⚠ **Warning**
>
> One a channel is closed, neither party should use the 2-of-2 multisig wallet again, since there exists another party that can immediately spend out of that wallet.

3.2. High-level state machine

On a high level, the payment channel lifecycle goes through 6 phases:

- `New` - The channel has just been created and is entering the establishment negotiation phase. Basic info is swapped in this phase, including the public keys of the peers, the nominated KES, and the initial balance. The channel ID is derived from the public keys, the initial balance, plus some salt. An `AckNewChannel` message is sent between the peers to acknowledge the channel creation. A rejection message, `RejectNewChannel`, can also be sent, for example, if the counterparty does not agree on the amount, or the validity of the KES public key (which must come from a trusted shared whitelist). Once acknowledged, an `OnNewChannelInfo` event is emitted, and the channel will move to the `Establishing` state. Otherwise, the state will time out and move to the `Closed` state via an `onTimeout` or `onRejectNewChannel` event.
- `Establishing` - The channel is being established. This phase includes the KES establishment and funding transaction. Once the KES is established and both parties have verified the funding transaction, the parties will share an `AckChannelEstablished` message. Once acknowledged, an `OnChannelEstablished` event is emitted, and the channel will move to the `Open` state.
- `Open` - The channel is open and ready for use. Any number of channel update events can occur in this phase and the channel can remain in this state indefinitely. The channel remains in this state until the channel is closed via the amicable `Closing` state or the `Disputing` state. The peers share an `AckWantToClose` message to signal a desire to close the channel. This triggers an `OnStartClose` event, and the channel will move to the `Closing` state. If the counterparty party initiates a force-close on the channel via the KES, an `onForceClose` event is emitted, and the channel moves to the `Disputing` state. If the counterparty stops responding to updates or for whatever other reason, you can trigger a force close (an `onTriggerForceClose` event), and the channel will move to the `Disputing` state.
- `Closing` - The channel is being closed. This phase includes the KES closing, sharing of adapter sharing secrets and signing of the final commitment transaction. The merchant is responsible for closing down the KES. Once both parties have signed the final commitment transaction, any party will be able to broadcast it, but by convention it will be the

merchant that does so. If all communications have resolved amicably, the peers will share an `AckChannelClosed` message. This triggers an `OnSuccessfulClose` event, and the channel will move to the `Closed` state. Otherwise, an `onInvalidClose(reason)` event is emitted, and the channel will move to the `Disputing` state.

- `Closed` - The channel is closed and the funds are being settled. This phase includes the KES closing and broadcast of the commitment transaction (if necessary).
- `Disputing` - The channel is being disputed because someone initiated a force-close. If the local party initiated the force close, this phase includes invoking the force-close on the KES, and waiting for the challenge window to expire so that the counterparty's secret share can be recovered in order to reconstruct the spending key. If the other party initiated the force-close, we can invoke the KES to challenge the closing state, or do nothing and accept the state given by the counterparty. The final state transition is always to the `Closed` state, only the reason can vary. If the counterparty successfully disputes the closure (because you tried a force-close with an outdated state), an `onDisputeResolved(reason)` event is emitted. Otherwise, an `onForceCloseResolved` event is emitted.

Each state contains an internal state machine that describes the events that can occur in that state, including handling of all p2p and blockchain network communications.

3.3. New Channel

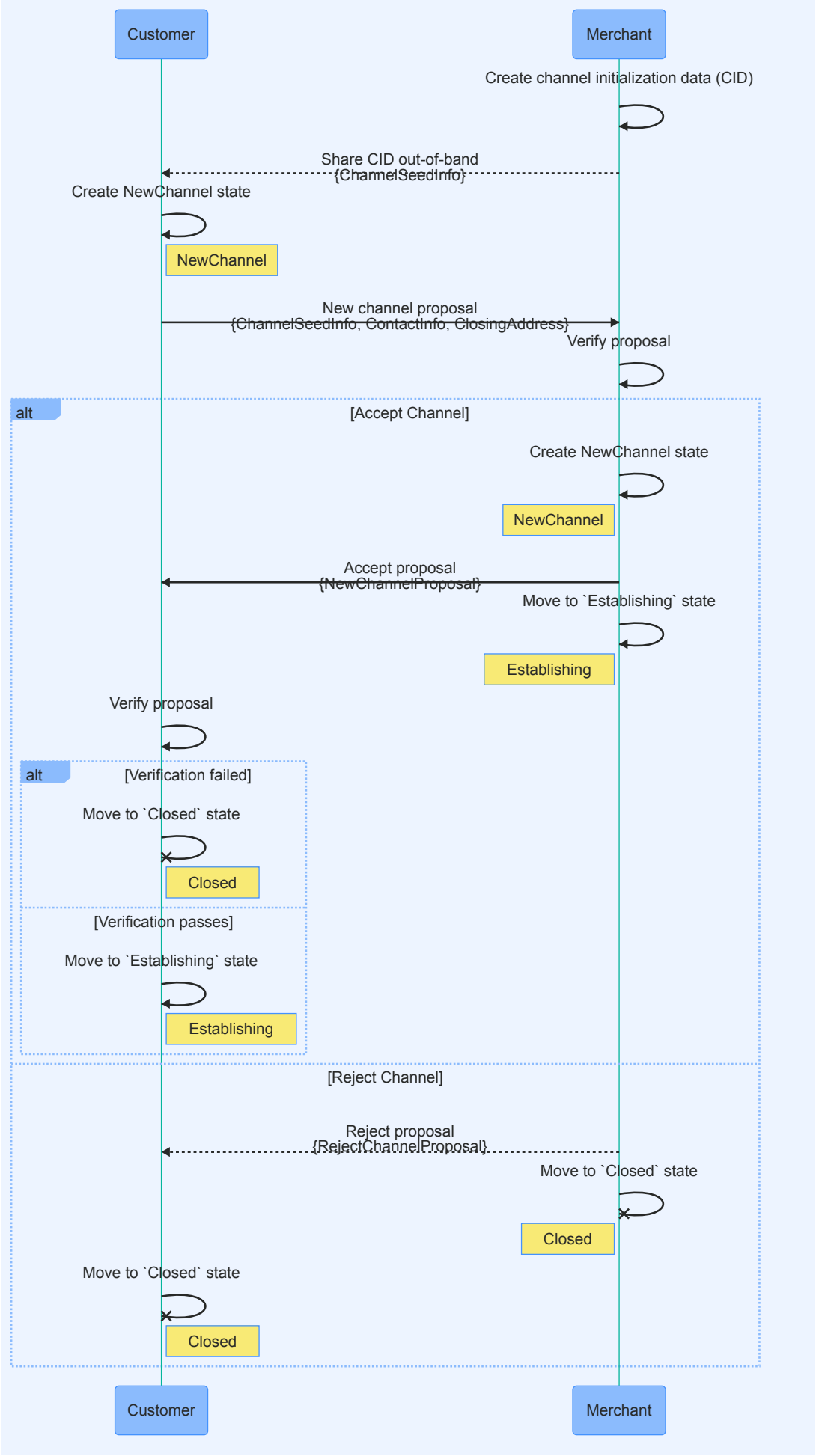
A new channel is established when a Merchant shares some initialization data with a Customer out-of-band.

The customer takes this data, combines it with their own information, and sends a channel proposal to the Merchant.

There are **three** half-rounds of communication in this phase[1]:

1. Out-of-band channel initialization data (CID) sharing from Merchant to Customer:
   - Contact information for the merchant
   - Channel seed metadata. This includes metadata so that both merchant and customer can uniquely identity the channel throughout the channel's lifetime. This includes:
     - an id for the channel
     - The merchant's closing address
     - The requested initial balances
     - The merchant's id
   - Protocol-specific intialization data. This might include commitments for parameters that will be shared later, the KES public keys that can be accepted, etc.

[1] See `server.rs:customer_establish_new_channel`


Customer
Merchant
Create channel initialization data (CID)
Share CID out-of-band
{ChannelSeedInfo}
Create NewChannel state
NewChannel
New channel proposal
{ChannelSeedInfo, ContactInfo, ClosingAddress}
Verify proposal
alt
[Accept Channel]
Create NewChannel state
NewChannel
Accept proposal
{NewChannelProposal}
Move to `Establishing` state
Establishing
Verify proposal
alt
[Verification failed]
Move to `Closed` state
Closed
[Verification passes]
Move to `Establishing` state
Establishing
[Reject Channel]
Reject proposal
{RejectChannelProposal}
Move to `Closed` state
Closed
Move to `Closed` state
Closed
Customer
Merchant

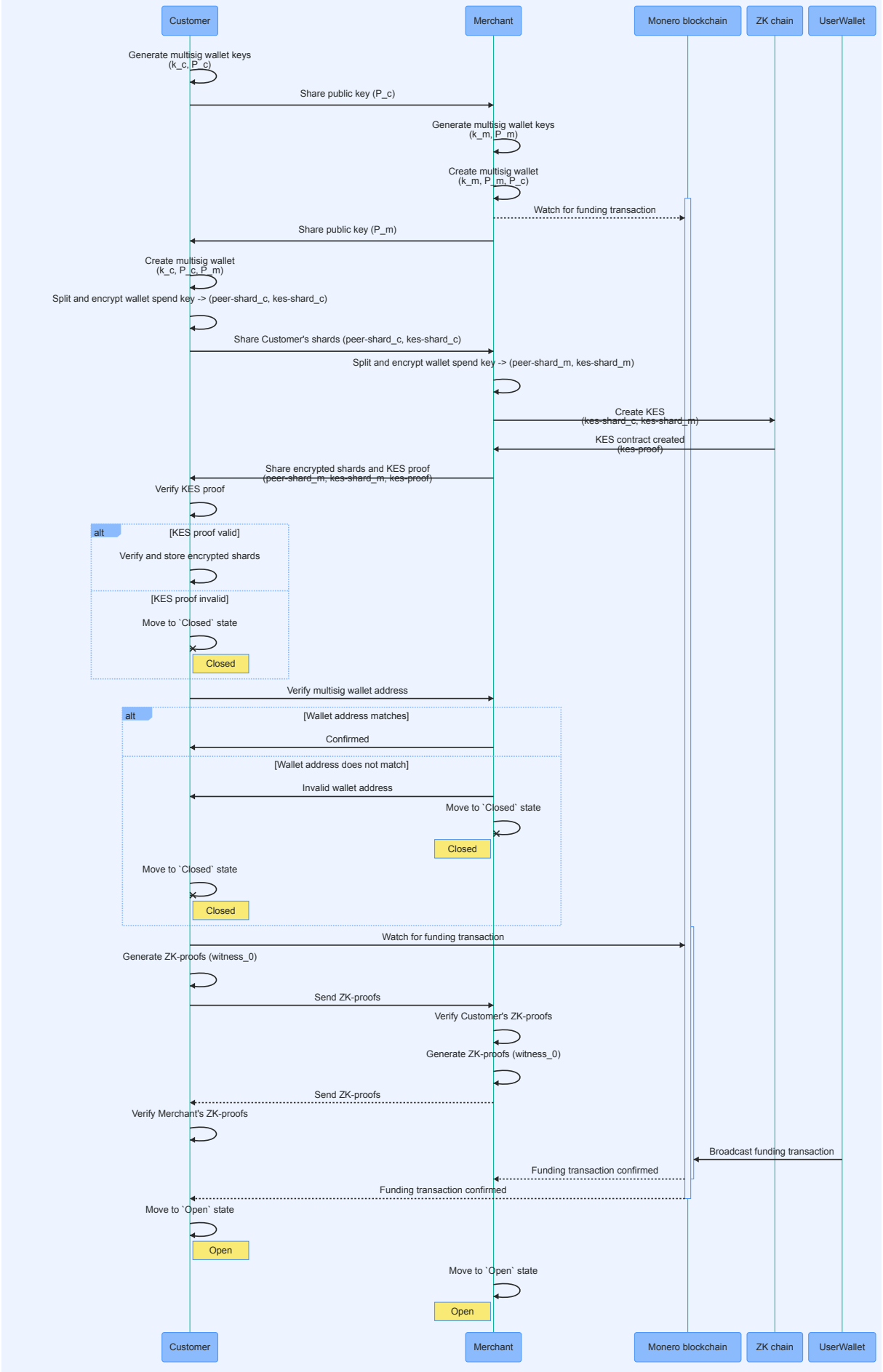
3.4. Establishing the Channel

Establishing a channel to accept payments requires the following preparatory steps:

1. Both parties collaboratively create a new shared multisig wallet to hold the channel funds.
2. Each party watches the Monero blockchain for the funding transaction to confirm it has been included in a block.
3. Each party splits their spend key into two parts and encrypts them. One share is encrypted for the other party, and

the other share is encrypted for the KES.

4. The merchant creates a new KES commitment on the ZK-chain smart contract and commits the encrypted shares to it.
5. The merchant provides a proof of the KES commitment to the customer who can verify that the KES was set up correctly.
6. Each party creates the initial ZK proof for their initial secret (witness) and shares it with the counterparty.
7. The customer funds the multisig wallet with the agreed initial balance.
8. Once the funding transaction is confirmed, the channel is open and ready to use.


Customer
Merchant
Monero blockchain
ZK chain
UserWallet
Generate multisig wallet keys (k_c, P_c)
Share public key (P_c)
Generate multisig wallet keys (k_m, P_m)
Create multisig wallet (k_m, P_m, P_c)
Watch for funding transaction
Share public key (P_m)
Create multisig wallet (k_c, P_c, P_m)
Split and encrypt wallet spend key -> (peer-shard_c, kes-shard_c)
Share Customer's shards (peer-shard_c, kes-shard_c)
Split and encrypt wallet spend key -> (peer-shard_m, kes-shard_m)
Create KES (kes-shard_c, kes-shard_m)
KES contract created (kes-proof)
Share encrypted shards and KES proof (peer-shard_m, kes-shard_m, kes-proof)
Verify KES proof
alt
[KES proof valid]
Verify and store encrypted shards
[KES proof invalid]
Move to `Closed` state
Closed
Verify multisig wallet address
alt
[Wallet address matches]
Confirmed
[Wallet address does not match]
Invalid wallet address
Move to `Closed` state
Closed
Move to `Closed` state
Closed
Watch for funding transaction
Generate ZK-proofs (witness_0)
Send ZK-proofs
Verify Customer's ZK-proofs
Generate ZK-proofs (witness_0)
Send ZK-proofs
Verify Merchant's ZK-proofs
Broadcast funding transaction
Funding transaction confirmed
Funding transaction confirmed
Move to `Open` state
Open
Move to `Open` state
Open
Customer
Merchant
Monero blockchain
ZK chain
UserWallet

4. The Zero-Knowledge Contracts

4.1. Grease payment channel lifetime

4.1.1. Channel Initialization

At **initialization**, two peers will:

1. communicate out-of-band, share connection information and agree to a fixed balance amount in XMR,
2. connect over a dedicated private communication channel,
3. create a new temporary Monero 2-of-2 multisignature wallet where each peer has full view access and 1-of-2 spend access,
4. create a KES subscription,
5. create proofs of randomizing a new root secret,
6. create proofs of using that root secret for an adaptor signature,
7. create proofs of sharing that root secret with the KES,
8. verify those proofs from the peer,
9. create a shared closing transaction where both peers receive a v_{out} output to their private Monero wallet with

the exact amount of their starting balance using the adaptor signature, so that each peer has 3-of-4 pieces of information needed to broadcast the transaction,

1. verify the correctness of the closing transaction using the shared view key, the unadapted signatures and the adaptor statements,
2. create a shared funding transaction where both peers provide a v_{in} input from their private Monero wallet with the exact amount of their balance,
3. verify the correctness of the funding transaction using the shared view key,
4. activate the KES with the root secret shares,
5. and finally broadcast the funding transaction to Monero.

4.1.2. Channel Update

When updating the channel balance, the two peers will:

1. update the balance out-of-band,
2. create proofs of deterministically updating the previous secret,
3. create proofs of using the updated secret for a new adaptor signature,
4. verify those proofs from the peer,
5. update the shared closing transaction where both peers receive an updated v_{out} output to their private Monero wallet with the new amount of their balance using the new adaptor signature, so that each peer has the new 3-of-4 pieces of information needed to broadcast the transaction,
6. and finally verify the correctness of the updated closing transaction using the shared view key, the new unadapted signatures and the new adaptor statements.
7. Repeat as often as desired.

4.1.3. Channel Closure

When closing the channel, the two peers will:

1. share their most recent secret,

2. adapt the unadapted signature of the closing transaction to gain the 4-of-4 pieces of information needed to broadcast the transaction,
3. and finally broadcast the closing transaction to Monero.

4.1.4. Channel Dispute

In case of a **dispute**, a plaintiff will:
1. provide the unadapted signatures of the closing transaction to the KES,
2. monitor the KES public state for the dispute status.

After verifying the dispute provided by the plaintiff the KES will:
1. signal the dispute on its public state,
2. monitor Monero for the closing transaction,
3. start a timer for the dispute-response window (Δ_{dispute}) and wait for it to expire.

The defendant will monitor the KES public state for a dispute. If the defendant detects the dispute, the defendant will:
1. accept the dispute and send the adapted signature of the closing transaction to the KES,
2. accept the dispute and broadcast the closing transaction to Monero,
3. protest the dispute and send the unadapted signatures of a newer closing transaction to the KES,
4. ignore the dispute and allow the KES to rule in favor of the plaintiff.

The KES reacts to the defendant by:
1. verifying the adapted signature:
    1. if verified the KES will close the dispute and update the public state with the adapted signature,
    2. if not verified the KES will rule in favor of the plaintiff and proceed with the unlock process.
2. observing the closing transaction on Monero and closing the dispute,
3. processing the protest by verifying the unadapted signatures of the future transaction:
    1. if verified the KES will rule in favor of the defendant and proceed with the unlock process,
    2. if not verified the KES will rule in favor of the plaintiff and proceed with the unlock process.
4. recognizing that the dispute response window has expired, finding that the plaintiff is wronged and proceeding with the unlock process.

The KES may start the unlock process for the wronged peer against the violating peer:
1. the KES will close the dispute and update the public state with the violating peer's saved root-secret share, encrypted to the wronged peer.

The wronged peer will monitor the KES public state for the dispute closure. If the wronged peer detects the unlock process, the wronged peer will:
1. reconstruct the violating peer's root secret,
2. deterministically advance the secret until a valid closing transaction can be formed[2],

[2]Under CLSAG, older transactions may quickly become stale and be rejected by the network. This time window will be relaxed post-FCMP++.

3. adapt the unadapted signature of the closing transaction using the violating peer's most recent secret to gain the 4-of-4 pieces of information needed to broadcast the transaction,
4. broadcast the closing transaction to Monero.

4.2. Grease Protocol

The Grease protocol operates in four stages: initialization, update, closure and dispute. The ZKPs are used only in the initialization and update stage, as the closure and dispute do not need further verification to complete.

4.2.1. Initialization

4.2.1.1. Motivation

The two peers will decide to lock their declared XMR value and create a Grease payment channel so that they can begin transacting in the channel and not on the Monero network.

4.2.1.2. Preliminary

For the initialization stage to begin, the peers must agree upon a small amount of information:

Resource	
Channel ID	The identifier of the private communications channel. This will include the public key identifier of the peers and information about the means of communications between them.
Locked Amount	The two values in XMR (with either but not both allowed as zero) that the peers will lock into the channel during its lifetime.

At the start of the initialization stage the peers provide each other with the following resources and information:

Before Initialization		
Resource	**Visibility**	
Π_{peer}	Public	The public key/curve point on Baby Jubjub for the peer
Π_{KES}	Public	The public key/curve point on Baby Jubjub for the KES
ν_{peer}	Public	Random 251 bit value, provided by the peer (`nonce_peer`)

The peers will also agree on a third party agent to host the Key Escrow Service (KES). When the peers agree on the particular KES, the public key to this service is shared as Π_{KES}.

Each participant will create a new one-time key pair to use for communication with the KES in the case of a dispute. The peers share the public keys with each other, referring to the other's as Π_{peer}.

During the interactive setup, the peers send each other a nonce, ν_{peer}, that guarantees that critically important data must be new and unique for this channel. This prevents the reuse of old data held by the peers.

The ZKP protocols prove that the real private keys are used correctly and that if a dispute is necessary, it will succeed.

4.2.1.3. Initialization protocol

The Grease protocol requires the generation and sharing of the ZKPs. The public data and the small proofs are shared between peers, then are validated as a means to ensure protocol conformity before **MoNet** protocol stage 3 begins.

Each peer generates a set of secret random values to ensure security of communications, the **private** variables listed in Table 1. These are not shared with the peer.

Input	Visibility	
ν_{ω_0}	Private	Random 251 bit value (`blinding`)
a_1	Private	Random 251 bit value
ν_1	Private	Random 251 bit value (`r_1`)
ν_2	Private	Random 251 bit value (`r_2`)
ν_{DLEQ}	Private	Random 251 bit value (`blinding_DLEQ`)
ν_{peer}	Public	Random 251 bit value, provided by the peer (`nonce_peer`)
Π_{peer}	Public	The public key/curve point on Baby Jubjub for the peer
Π_{KES}	Public	The public key/curve point on Baby Jubjub for the KES

Table 1: Inputs to ZKPs for the Grease Initialization Protocol

The ZKP operations produce the set of output values listed in Table 2. The publicly visible values must be shared with the peer in addition to the generated proofs while the privately visible values must be stored for later use.

Output	Visibility	
T_0	Public	The public key/curve point on Baby Jubjub for ω_0
ω_0	Private	The root private key protecting access to the user's locked value (`witness_0`)
c_1	Public	`Feldman commitment 1` (used in tandem with `Feldman commitment 0` $= T_0$), which is a public key/curve point on Baby Jubjub
σ_1	Private	The split of ω_0 shared with the peer (`share_1`)
Φ_1	Public	The ephemeral public key/curve point on Baby Jubjub for message transportation to the peer (`fi_1`)
χ_1	Public	The encrypted value of σ_1 (`enc_1`)
σ_2	Private	The split of ω_0 shared with the KES (`share_2`)
Φ_2	Public	The ephemeral public key/curve point on Baby Jubjub for message transportation to the KES (`fi_2`)

Output	Visibility	
χ_2	Public	The encrypted value of σ_2 (`enc_2`)
S_0	Public	The public key/curve point on Ed25519 for ω_0
C	Public	The Fiat–Shamir heuristic challenge (`challenge_bytes`)
Δ_{BJJ}	Private	Optimization parameter (`response_div_BabyJubjub`)
ρ_{BJJ}	Public	The Fiat–Shamir heuristic challenge response on the Baby Jubjub curve (`response_BabyJubJub`)
Δ_{Ed}	Private	Optimization parameter (`response_div_ed25519`)
ρ_{Ed}	Public	The Fiat–Shamir heuristic challenge response on the Ed25519 curve (`response_div_ed25519`)

Table 2: Outputs of ZKPs for the Grease Initialization Protocol

During the initialization stage, the following operations are performed:

- **VerifyWitness0**
- **VerifyWitnessSharing**
- **VerifyEquivalentModulo**
- **VerifyDLEQ**

Particular details about these operations can be found in Section 5.

4.2.1.4. Post-initialization

After receiving the publicly visible values and ZK proofs from the peer, the Grease protocol requires the ZKP verification operations to ensure protocol conformity.

Once verified, the variables listed in Table 3 must be stored. With these outputs the initialization stage is complete and the channel is open. The peers can now transact and update the channel state or close the channel and receive the locked XMR value in the **Monero Refund Wallet**.

Resource	
Φ_1	The ephemeral public key/curve point on Baby Jubjub for message transportation from the peer (`fi_1`)
χ_1	The encrypted value of σ_1 (`enc_1`) for the peer's ω_0
ω_0	The root private key protecting access to the user's locked value (`witness_0`)
S_0	The public key/curve point on Ed25519 for the peer's ω_0

Table 3: Resources and Information after Grease Initialization

4.2.2. Channel Update

4.2.2.1. Motivation

Once a channel is open the peers may decide to transact and update the XMR balance between the peers. The only requirement is that the peers agree on the change in ownership of the **Locked Amount**.

Note that with an open channel there is no internal reason to perform an update outside of a peer-initiated change. However, the current Monero protocol requires that a newly broadcast transaction be created within a reasonable timeframe. As such, existing open channel should create a "zero delta" update at reasonable timeframes to ensure the channel may be closed arbitrarily. The specifics on this are outside of current scope.

Note that post-FCMP++, the signing mechanism for Monero transactions will be such that decoy selection can be deferred until channel closing[7]. This will simplify channel updates in two important ways:

- Updates will not need to query the Monero blockchain to select decoys at update time, which present a significant performance improvement.
- Channels can stay open indefinitely, without risk of the closing transaction becoming stale.

4.2.2.2. Preliminary

For the update stage to begin, the peers must agree upon a small amount of information:

Resource	
Δ	The change in the two values in XMR (positive or negative) from the previous stage. This is a single number since the **Locked Amount** must stay the same.

4.2.2.3. Update protocol

Grease replaces the MoNet update protocol completely with the generation and sharing of update ZKPs. The public data and the small proofs are shared between peers, then are validated as a means to ensure protocol conformity.

The ZKP operations require the previous ω_i (now ω_{i-1}) and a random value to ensure security of communications, as described in Table 4. These are not shared with the peer.

Input	Visibility	
ω_{i-1}	Private	The current private key protecting access to close the payment channel (`witness_im1`)
ν_{DLEQ}	Private	Random 251 bit value (`blinding_DLEQ`)

Table 4: Inputs to ZKPs for the Grease Update Protocol

The ZKP operations produce the set of output values listed in Table 5. The public values must be shared with the peer in addition to the generated proofs while the private values are stored for later use.

Output	Visibility	
T_{i-1}	Public	The public key/curve point on Baby Jubjub for ω_{i-1}
T_i	Public	The public key/curve point on Baby Jubjub for ω_i
ω_i	Private	The next private key protecting access to close the payment channel (`witness_i`)
S_i	Public	The public key/curve point on Ed25519 for ω_i
Δ_{BJJ}	Private	Optimization parameter (`response_div_BabyJubjub`)
ρ_{BJJ}	Public	The Fiat–Shamir heuristic challenge response on the Baby Jubjub curve (`response_BabyJubJub`)
Δ_{Ed}	Private	Optimization parameter (`response_div_ed25519`)
ρ_{Ed}	Public	The Fiat–Shamir heuristic challenge response on the Ed25519 curve (`response_div_ed25519`)
C	Public	The Fiat–Shamir heuristic challenge (`challenge_bytes`)
R_{BJJ}	Public	DLEQ commitment 1, which is a public key/curve point on Baby Jubjub (`R_1`)
R_{Ed}	Public	DLEQ commitment 2, which is a public key/curve point on Ed25519 (`R_2`)

Table 5: Outputs of ZKPs for the Grease Update Protocol

During the update stage, the following operations are performed:

- **VerifyCOF**
- **VerifyEquivalentModulo**
- **VerifyDLEQ**

Particular details about these operations can be found in Section 5.

4.2.2.4. Post-update

After receiving the publicly visible values and ZK proofs from the peer, the Grease protocol requires the ZKP verification operations to ensure protocol conformity.

Once verified, the variables listed in Table 6 must be stored:

Resource/Variable	
ω_i	The current private key protecting access to close the payment channel (`witness_i`)
S_i	The public key/curve point on Ed25519 for the peer's ω_i

Table 6: Variables to be stored after every channel update

With these outputs, the update stage is complete and the channel remains open. The peers can now transact further updates or close the channel and receive the **Channel Balance** (locked XMR) in the **Monero Refund Wallet**.

5. Grease ZKP Operations

The Grease protocol requires the creation and sharing of a series of Zero Knowledge proofs (ZKPs) as part of the lifetime of a payment channel. Most are Non-Interactive Zero Knowledge (NIZK) proofs in the form of Turing-complete circuits created using newly-established Plonky-based proving protocols. The others are classical interactive protocols with verification.

5.1. VerifyWitness0

5.1.1. Inputs

Input	Visibility	
ν_{peer}	Public	Random 251 bit value, provided by the peer (`nonce_peer`)
ν_{ω_0}	Private	Random 251 bit value (`blinding`)

5.1.2. Outputs

Output	Visibility	
T_0	Public	The public key/curve point on Baby Jubjub for ω_0
ω_0	Private	The root private key protecting access to the user's locked value (`witness_0`)

5.1.3. Summary

The **VerifyWitness0** operation is a Noir ZK circuit using the UltraHonk prover/verifier. It receives the provided random entropy inputs and produces the deterministic outputs. The circuit is ZK across the inputs, so no information is gained about the private inputs even with knowledge of the private output. The T_0 output is used for the further **VerifyEquivalentModulo** and **VerifyDLEQ** operations.

The operation uses the **blake2s** hashing function for its one-way random oracle simulation.

The scalar order of the Baby Jubjub curve is represented here by L_{BJJ}.

5.1.4. Methods

$$\begin{aligned} C &= H_{\text{blake2s}}\left(\text{HEADER} \parallel \nu_{\text{peer}} \parallel \nu_{\omega_0}\right) \\ \omega_0 &= C \bmod L_{\text{BJJ}} \\ T_0 &= \omega_0 \cdot G_{\text{BJJ}} \end{aligned} \tag{1}$$

5.2. FeldmanSecretShare_2_of_2

5.2.1. Inputs

Input	Visibility	

ω_0	Private	The root private key protecting access to the user's locked value (`secret`)
a_1	Private	Random 251 bit value

5.2.2. Outputs

Output	Visibility	
T_0	Public	Feldman commitment 0, which is the public key/curve point on Baby Jubjub for ω_0
c_1	Public	Feldman commitment 1, which is a public key/curve point on Baby Jubjub
σ_1	Private	The split of ω_0 shared with the peer (`share_1`)
σ_2	Private	The split of ω_0 shared with the KES (`share_2`)

5.2.3. Summary

The **FeldmanSecretShare_2_of_2** operation is a Noir ZK circuit using the UltraHonk prover/verifier. It receives the provided secret data and random entropy inputs. The outputs are the two perfectly binding Feldman commitments and the two encoded split shares to send to the destinations. The circuit is not ZK across the inputs so that full knowledge of the private outputs can reconstruct the private inputs.

The outputs are used for the further **VerifyEncryptMessage**, **VerifyFeldmanSecretShare_peer**, **VerifyFeldmanSecretShare_KES**, and **ReconstructFeldmanSecretShare_2_of_2** operations.

The scalar order of the Baby Jubjub curve is represented here by L_{BJJ}.

Note that for the peer to verify the validity of the secret sharing protocol, the calculation is:

$$\mathrm{VerifyFeldmanSecretShare\_peer}(T_0, c_1, \sigma_1) \tag{2}$$

Note that for the KES to verify the validity of the secret sharing protocol, the calculation is:

$$\mathrm{VerifyFeldmanSecretShare\_KES}(T_0, c_1, \sigma_2) \tag{3}$$

Note that for reconstructing the secret input ω_0, the calculation is:

$$\omega_0 = \mathrm{ReconstructFeldmanSecretShare\_2\_of\_2}(\sigma_1, \sigma_2) \tag{4}$$

The negative σ_1 is non-standard compared to typical Shamir/Feldman schemes. We use this to make the mathematics clearer and to avoid subtraction operations in certain legacy circuit implementations.

5.2.4. Methods

$$
\begin{aligned}
T_0 &= \omega_0 \cdot G_{\text{BJJ}} \\
c_1 &= a_1 \cdot G_{\text{BJJ}} \\
\sigma_1 &= -(\omega_0 + a_1) \bmod L_{\text{BJJ}} \\
\sigma_2 &= 2 * \omega_0 + a_1 \bmod L_{\text{BJJ}}
\end{aligned}
\tag{5}
$$

5.3. ReconstructFeldmanSecretShare_2_of_2

5.3.1. Inputs

Input	Visibility	
σ_1	Private	The split of ω_0 shared with the peer (`share_1`)
σ_2	Private	The split of ω_0 shared with the KES (`share_2`)

5.3.2. Outputs

Output	Visibility	
ω_0	Private	The root private key protecting access to the user's locked value (`secret`)

5.3.3. Summary

The **ReconstructFeldmanSecretShare_2_of_2** operation is a reconstruction protocol and is language independent. It receives the two split shares as inputs and outputs the original ω_0 secret.

The scalar order of the Baby Jubjub curve is represented here by L_{BJJ}.

5.3.4. Methods

$$
\omega_0 = \sigma_1 + \sigma_2 \bmod L_{\text{BJJ}} \tag{6}
$$

5.4. VerifyFeldmanSecretShare_peer

5.4.1. Inputs

Input	Visibility	
T_0	Public	Feldman commitment 0, which is the public key/curve point on Baby Jubjub for ω_0
c_1	Public	Feldman commitment 1, which is a public key/curve point on Baby Jubjub
σ_1	Private	The split of ω_0 shared with the peer (`share_1`)

5.4.2. Outputs

There are no outputs.

5.4.3. Summary

The **VerifyFeldmanSecretShare_peer** operation is a verification protocol and is language independent. This operation is redundant, in that the successful verification of the previous **FeldmanSecretShare_2_of_2** operation with the same publicly visible parameters implies that this operation will succeed.

5.4.4. Methods

$$\text{assert}(\sigma_1 \cdot G_{\text{BJJ}} == -(T_0 + c_1)) \tag{7}$$

5.5. VerifyFeldmanSecretShare_KES

5.5.1. Inputs

Input	Visibility	
T_0	Public	Feldman commitment 0, which is the public key/curve point on Baby Jubjub for ω_0
c_1	Public	Feldman commitment 1, which is a public key/curve point on Baby Jubjub
σ_2	Private	The split of ω_0 shared with the KES (`share_2`)

5.5.2. Outputs

There are no outputs.

5.5.3. Summary

The **VerifyFeldmanSecretShare_KES** operation is a verification protocol and is language independent. This operation is not redundant, in that the successful verification of the previous **FeldmanSecretShare_2_of_2** operation with the same publicly visible parameters implies that this operation will succeed, but the conditions are different.

This operation will be implemented by the KES in its own native implementation language where the successful verification of the previous **FeldmanSecretShare_2_of_2** operation cannot be assumed. As such, this operation will exist and can called independently of any other operations.

5.5.4. Methods

$$\text{assert}(\sigma_2 \cdot G_{\text{BJJ}} == 2 \cdot T_0 + c_1) \tag{8}$$

5.6. VerifyEncryptMessage

5.6.1. Inputs

Input	Visibility	
σ	Private	The secret 251 bit message (`message`)
ν	Private	Random 251 bit value (`r`)

Π	Public	The public key/curve point on Baby Jubjub for the destination

5.6.2. Outputs

Output	Visibility	
Φ	Public	The ephemeral public key/curve point on Baby Jubjub for message transportation (`fi`)
χ	Public	The encrypted value of σ (`enc`)

5.6.3. Summary

The **VerifyEncryptMessage** operation is a Noir ZK circuit using the UltraHonk prover/verifier. It receives the provided secret data and random entropy inputs. The outputs are the perfectly binding public key commitment and the perfectly hiding encrypted scalar value to send to the destinations. The circuit is ZK across the inputs since the outputs are publicly visible.

The method of encryption is the ECDH (Elliptic-curve Diffie–Hellman) key agreement protocol. The operation uses the **blake2s** hashing function for its shared secret commitment simulation. Note that the unpacked form of the ephemeral key is used for hashing, instead of the standard $\mathrm{PACKED}()$ function.

Note that for reconstructing the secret input σ given the private key κ where $\Pi = \kappa \cdot G_{\mathrm{BJJ}}$, the calculation is:

$$(\Pi, \sigma) = \mathrm{DecryptMessage}(\kappa, \Phi, \chi) \tag{9}$$

Note that this operation does not call for the use of HMAC or other message verification protocol due to the simplicity of the interactive steps and their resistance to message tampering. A more complicated or distributed protocol would requires this attack prevention.

The scalar order of the Baby Jubjub curve is represented here by L_{BJJ}.

5.6.4. Methods

$$\begin{aligned} \Phi &= \nu \cdot G_{\mathrm{BJJ}} \\ \nu_{\Pi} &= \nu \cdot \Pi \\ C &= H_{\mathrm{blake2s}}(\nu_{\Pi}.\mathrm{x} \parallel \nu_{\Pi}.\mathrm{y}) \\ s &= C \bmod L_{\mathrm{BJJ}} \\ \chi &= \sigma + s \bmod L_{\mathrm{BJJ}} \end{aligned} \tag{10}$$

5.7. DecryptMessage

5.7.1. Inputs

Input	Visibility	
κ	Private	The private key for the public key Π

Φ	Public	The ephemeral public key/curve point on Baby Jubjub for message transportation (`fi`)
χ	Public	The encrypted value of σ (`enc`)

5.7.2. Outputs

Output	Visibility	
Π	Public	The public key/curve point on Baby Jubjub for the destination
σ	Private	The secret 251 bit message (`message`)

5.7.3. Summary

The **DecryptMessage** operation is a verification protocol and is language independent.

The method of decryption is the ECDH (Elliptic-curve Diffie–Hellman) key agreement protocol. The operation uses the **blake2s** hashing function for its shared secret commitment simulation. Note that the unpacked form of the ephemeral key is used for hashing, instead of the standard $\mathrm{PACKED}()$ function.

The scalar order of the Baby Jubjub curve is represented here by L_{BJJ}.

5.7.4. Methods

$$\begin{aligned} \Pi &= \kappa \cdot G_{\mathrm{BJJ}} \\ \kappa_{\Phi} &= \kappa \cdot \Phi \\ C &= H_{\mathrm{blake2s}}(\kappa_{\Phi}.\mathrm{x} \parallel \kappa_{\Phi}.\mathrm{y}) \\ s &= C \bmod L_{\mathrm{BJJ}} \\ \sigma &= \chi - s \bmod L_{\mathrm{BJJ}} \end{aligned} \tag{11}$$

5.8. VerifyWitnessSharing

5.8.1. Inputs

Input	Visibility	
ω_0	Private	The root private key protecting access to the user's locked value (`witness_0`)
a_1	Private	Random 251 bit value
ν_1	Private	Random 251 bit value (`r_1`)
Π_{peer}	Public	The public key/curve point on Baby Jubjub for the peer
ν_2	Private	Random 251 bit value (`r_2`)
Π_{KES}	Public	The public key/curve point on Baby Jubjub for the KES

5.8.2. Outputs

Output	Visibility	
c_1	Public	`Feldman commitment 1` (used in tandem with `Feldman commitment 0` $= T_0$), which is a public key/curve point on Baby Jubjub
σ_1	Private	The split of ω_0 shared with the peer (`share_1`)
Φ_1	Public	The ephemeral public key/curve point on Baby Jubjub for message transportation to the peer (`fi_1`)
χ_1	Public	The encrypted value of σ_1 (`enc_1`)
σ_2	Private	The split of ω_0 shared with the KES (`share_2`)
Φ_2	Public	The ephemeral public key/curve point on Baby Jubjub for message transportation to the KES (`fi_2`)
χ_2	Public	The encrypted value of σ_2 (`enc_2`)

5.8.3. Summary

The **VerifyWitnessSharing** operation is a Noir ZK circuit using the UltraHonk prover/verifier. It passes through the provided inputs and calls the **FeldmanSecretShare_2_of_2** and **VerifyEncryptMessage** operations.

5.8.4. Methods

$$
\begin{aligned}
(c_1, \sigma_1, \sigma_2) &= \text{FeldmanSecretShare\_2\_of\_2}(\omega_0, a_1) \\
(\Phi_1, \chi_1) &= \text{VerifyEncryptMessage}\big(\sigma_1, \nu_1, \Pi_{\text{peer}}\big) \\
(\Phi_2, \chi_2) &= \text{VerifyEncryptMessage}(\sigma_2, \nu_2, \Pi_{\text{KES}})
\end{aligned}
\tag{12}
$$

5.9. VerifyCOF

5.9.1. Inputs

Input	Visibility	
ω_{i-1}	Private	The current private key protecting access to close the payment channel (`witness_im1`)

5.9.2. Outputs

Output	Visibility	
T_{i-1}	Public	The public key/curve point on Baby Jubjub for ω_{i-1}
T_i	Public	The public key/curve point on Baby Jubjub for ω_i
ω_i	Private	The next private key protecting access to close the payment channel (`witness_i`)

5.9.3. Summary

The **VerifyCOF** operation is a Noir ZK circuit using the UltraHonk prover/verifier. It receives the provided deterministic input and produces the deterministic outputs. The circuit is ZK across the inputs, so no information is gained about the private input even with knowledge of the private output. The T_i output is used for the further **VerifyEquivalentModulo** and **VerifyDLEQ** operations.

The operation uses the **blake2s** hashing function for its one-way random oracle simulation.

The scalar order of the Baby Jubjub curve is represented here by L_{BJJ}.

5.9.4. Methods

$$\begin{aligned} T_{i-1} &= \omega_{i-1} \cdot G_{\mathrm{BJJ}} \\ C &= H_{\mathrm{blake2s}}(\mathrm{HEADER} \parallel \omega_{i-1}) \\ \omega_i &= C \bmod L_{\mathrm{BJJ}} \\ T_i &= \omega_i \cdot G_{\mathrm{BJJ}} \end{aligned} \tag{13}$$

5.10. VerifyEquivalentModulo

5.10.1. Inputs

Input	Visibility	
ω_i	Private	The current private key protecting access to close the payment channel (`witness_i`)
ν_{DLEQ}	Private	Random 251 bit value (`blinding_DLEQ`)

5.10.2. Outputs

Output	Visibility	
T_i	Public	The public key/curve point on Baby Jubjub for ω_i
S_i	Public	The public key/curve point on Ed25519 for ω_i
C	Public	The Fiat–Shamir heuristic challenge (`challenge_bytes`)
Δ_{BJJ}	Private	Optimization parameter (`response_div_BabyJubjub`)
ρ_{BJJ}	Public	The Fiat–Shamir heuristic challenge response on the Baby Jubjub curve (`response_BabyJubJub`)
Δ_{Ed}	Private	Optimization parameter (`response_div_ed25519`)
ρ_{Ed}	Public	The Fiat–Shamir heuristic challenge response on the Ed25519 curve (`response_div_ed25519`)

5.10.3. Summary

The **VerifyEquivalentModulo** operation is a Noir ZK circuit using the UltraHonk prover/verifier. It receives the provided deterministic and random entropy inputs and produces the

random outputs. The circuit is not ZK across the inputs since part of the private outputs can be used to reveal information about the private input. The T_i, S_i, ρ_{BJJ}, and ρ_{Ed} outputs are used for the further **VerifyDLEQ** operation.

This operation proves that the two separate ephemeral ρ outputs are both modulo equivalent values determined from the same root value. This ensures that there is no need to compress the embedded size of secret data values transported across the different group orders of the Baby Jubjub and Ed25519 curves, and also avoids the need for the random abort process as specified here: https://eprint.iacr.org/2022/1593.pdf°

Note that the Δ_{BJJ} and Δ_{Ed} outputs are used only for optimization of the Noir ZK circuit and may be removed as part of information leakage prevention.

The operation uses the **blake2s** hashing function for its Fiat–Shamir heuristic random oracle model simulation.

The scalar order of the Baby Jubjub curve is represented here by L_{BJJ}. The scalar order of the Ed25519 curve is represented here by L_{Ed}.

5.10.4. Methods

$$
\begin{aligned}
T_i &= \omega_i \cdot G_{\mathrm{BJJ}} \\
S_i &= \omega_i \cdot G_{\mathrm{Ed}} \\
C &= H_{\mathrm{blake2s}}(\mathrm{HEADER} \parallel \mathrm{PACKED}(T_i) \parallel \mathrm{PACKED}(S_i)) \\
\rho &= \omega_i * C - \nu_{\mathrm{DLEQ}} \\
\rho_{\mathrm{BJJ}} &= \rho \bmod L_{\mathrm{BJJ}} \\
\Delta_{\mathrm{BJJ}} &= \frac{\rho - \rho_{\mathrm{BJJ}}}{L_{\mathrm{BJJ}}} \\
\rho_{\mathrm{Ed}} &= \rho \bmod L_{\mathrm{Ed}} \\
\Delta_{\mathrm{Ed}} &= \frac{\rho - \rho_{\mathrm{Ed}}}{L_{\mathrm{Ed}}}
\end{aligned}
\tag{14}
$$

5.11. VerifyDLEQ

5.11.1. Inputs

Input	Visibility	
T_i	Public	The public key/curve point on Baby Jubjub for ω_i
ρ_{BJJ}	Public	The Fiat–Shamir heuristic challenge response on the Baby Jubjub curve (`response_BabyJubJub`)
S_i	Public	The public key/curve point on Ed25519 for ω_i
ρ_{Ed}	Public	The Fiat–Shamir heuristic challenge response on the Ed25519 curve (`response_div_ed25519`)

5.11.2. Outputs

Output	Visibility	
C	Public	The Fiat–Shamir heuristic challenge (`challenge_bytes`)
R_{BJJ}	Public	DLEQ commitment 1, which is a public key/curve point on Baby Jubjub (`R_1`)
R_{Ed}	Public	DLEQ commitment 2, which is a public key/curve point on Ed25519 (`R_2`)

5.11.3. Summary

The **VerifyDLEQ** operation is a verification protocol and is language independent. This operation is not redundant, in that the successful verification of the previous **VerifyEquivalentModulo** operation with the same publicly visible parameters implies that this operation will succeed, but the conditions are different.

This operation will be implemented by the peers outside of a ZK circuit in its own native implementation language where the successful verification of the previous **VerifyEquivalentModulo** operation cannot be assumed complete. As such, this operation will exist and can called independently of any other operations.

This operation proves that the T_i and S_i public key/curve points were generated by the same secret key ω_i . Given that the two separate ephemeral ρ output values are both modulo equivalent values determined from the same root value, the reconstruction of the two separate R commitments proves this statement. The use of two separate ephemeral ρ output values ensures that there is no need to compress the embedded size of the secret data ω_i transported across the different group orders of the Baby Jubjub and Ed25519 curves, and also avoids the need for the random abort process as specified here: https://eprint.iacr.org/2022/1593.pdf

The operation uses the **blake2s** hashing function for its Fiat–Shamir heuristic random oracle model simulation.

5.11.4. Methods

$$
\begin{aligned}
C &= H_{\text{blake2s}}(\text{HEADER} \parallel \text{PACKED}(T_i) \parallel \text{PACKED}(S_i)) \\
\text{P}_{\text{BJJ}} &= \rho_{\text{BJJ}} \cdot G_{\text{BJJ}} \\
C_{T_i} &= C \cdot G_{\text{BJJ}} \\
R_{\text{BJJ}} &= C_{T_i} - \text{P}_{\text{BJJ}} \\
\text{P}_{\text{Ed}} &= \rho_{\text{Ed}} \cdot G_{\text{Ed}} \\
C_{S_i} &= C \cdot G_{\text{Ed}} \\
R_{\text{Ed}} &= C_{S_i} - \text{P}_{\text{Ed}}
\end{aligned}
\tag{15}
$$

6. Future extensions and known limitations

The Grease protocol represents the first attempt to extend the high-security features of Monero while also using the problem-solving flexibility of the latest Turing-complete ZKP tools. Given the rate at which the ZKP technology is advancing there may be many more opportunities to extend Monero's security to new features and markets, connecting the future of Monero with the larger blockchain community and bringing greater attention and interest to the security that Monero has consistently proven.

KES funding specifics are not assumed. If the KES runs on a ZKP-compatible smart contract blockchain then both peers will require a funded temporary key pair for the blockchain. With account abstraction this would be trivial. Without account abstraction this can be implemented by the peer that funds the KES to transfer gas to the anonymous peer to accommodate a possible dispute, with the anonymous peer refunding the gas after channel closure (or simply revealing the temporary private key).

7. Nomenclature

7.1. Symbols

Symbol	Description
G_{BJJ}	Generator point for curve Baby JubJub
G_{Ed}	Generator point for curve Ed25519
L_{BJJ}	The prime order of curve Baby JubJub
L_{Ed}	The prime order for curve Ed25519
ω_i	The witness value for party i
T_i	The public point corresponding to ω_i on curve Baby JubJub
S_i	The public point corresponding to ω_i on curve Ed25519

7.2. Subscripts

Subscript	Referent
BJJ	Curve Baby JubJub
Ed	Curve Ed25519
merchant	The peer playing the role of merchant
customer	The peer playing the role of customer
Initiator	The peer playing the role of initiator
Responder	The peer playing the role of responder

References

1. What is Monero?. https://www.getmonero.org/get-started/what-is-monero/°
2. Chainalysis. All About Monero. https://www.chainalysis.com/blog/all-about-monero/°
3. Alexander Culafi. Monero and the complicated world of privacy coins. January 24, 2022. https://www.techtarget.com/searchsecurity/news/252512394/Monero-and-the-complicated-world-of-privacy-coins°
4. Zhimei Sui, Joseph K. Liu, Jiangshan Yu, Man Ho Au, Jia Liu. *Auxchannel: Enabling Efficient Bi-Directional Channel for Scriptless Blockchains.*; 2022. https://eprint.iacr.org/2022/117.pdf°
5. Sui Z, Liu JK, Yu J, Qin X. *Monet: A Fast Payment Channel Network for Scriptless Cryptocurrency Monero.*; 2022. https://eprint.iacr.org/2022/744.pdf°
6. CJ, Sean Coughlin. Grease: A minimal Monero Payment Channel implementation for Monero. https://cfp.twed.org/mk5/talk/QYDGPM/°
7. jeffro256. Personal communication: Signature changes post-FCMP. Published online June 22, 2025.

Index of Tables