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 about 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 is 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 cryptographic 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, whereas 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,
- trustless,

will suffice.

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

The KES acts as a third-party judge in disputes. At initialization, each peer encrypts a secret (their ω_0 witness) for the KES. If a dispute arises, the KES identifies the violating peer and releases that peer's secret to the wronged peer. The wronged peer can use the secret to 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 timeout. 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, they 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 encrypt their adapter signature offset to the KES.

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 first adapter offset. The customer will

then be able to sign any transaction in the history of the channel by reconstructing the appropriate adapter signature offset, 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 adaptor 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 can be recovered in order to synthesize the closing transaction. 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

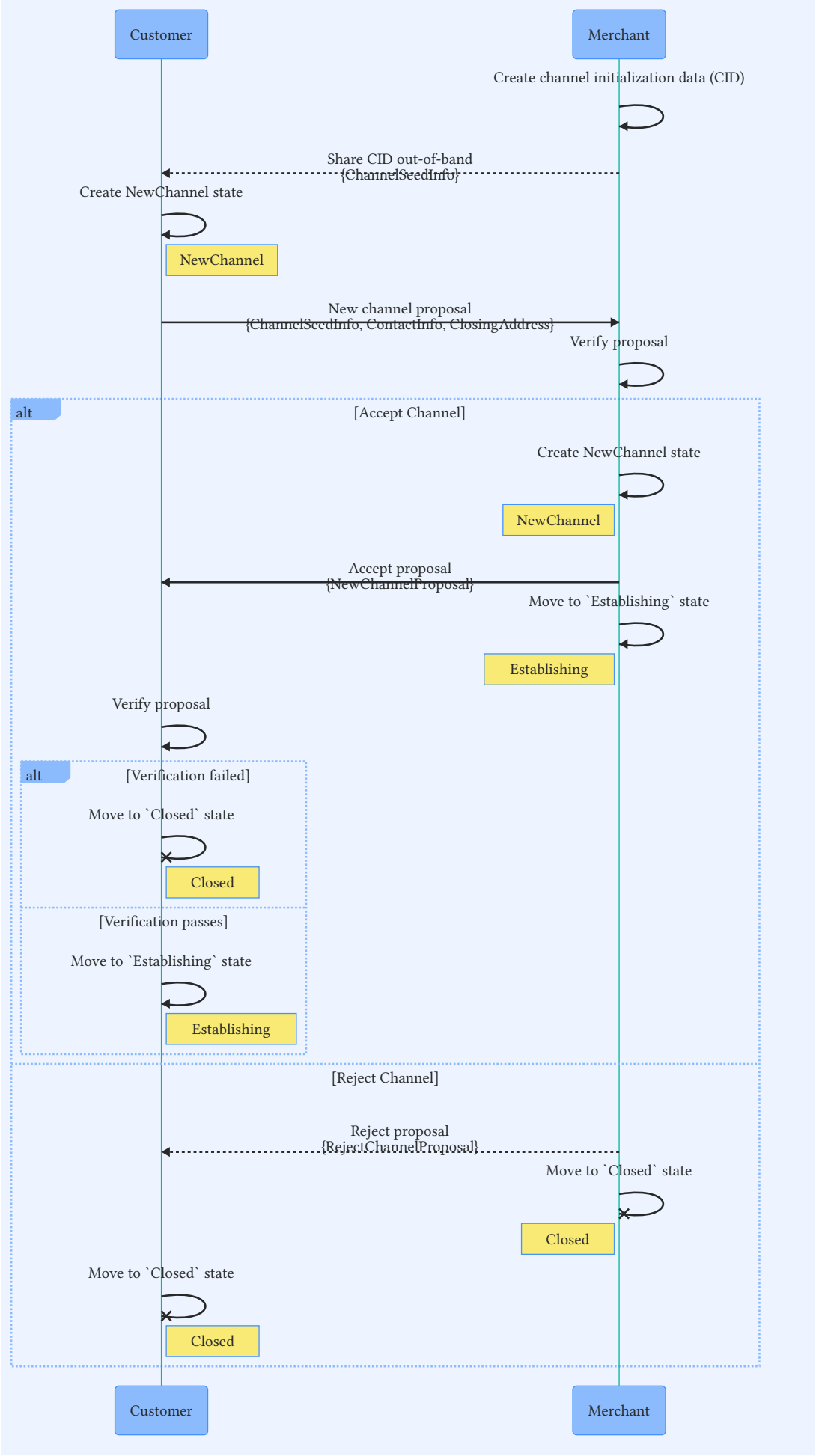
A new channel is established when a Merchant shares some initialization data with a Customer out-of-band.

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 identify the channel throughout the channel's lifetime. This includes:
     - a local name for the channel
     - The merchant's closing address
     - The requested initial balances
     - The merchant's id
   - Protocol-specific initialization data. This might include commitments for parameters that will be shared later, the KES public keys that can be accepted, etc.
2. New channel proposal from Customer to Merchant
3. Accepting the proposal from Merchant to Customer

[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

3.3.1. Channel Id

The channel id is a 64 character hexadecimal string that uniquely identifies the channel.

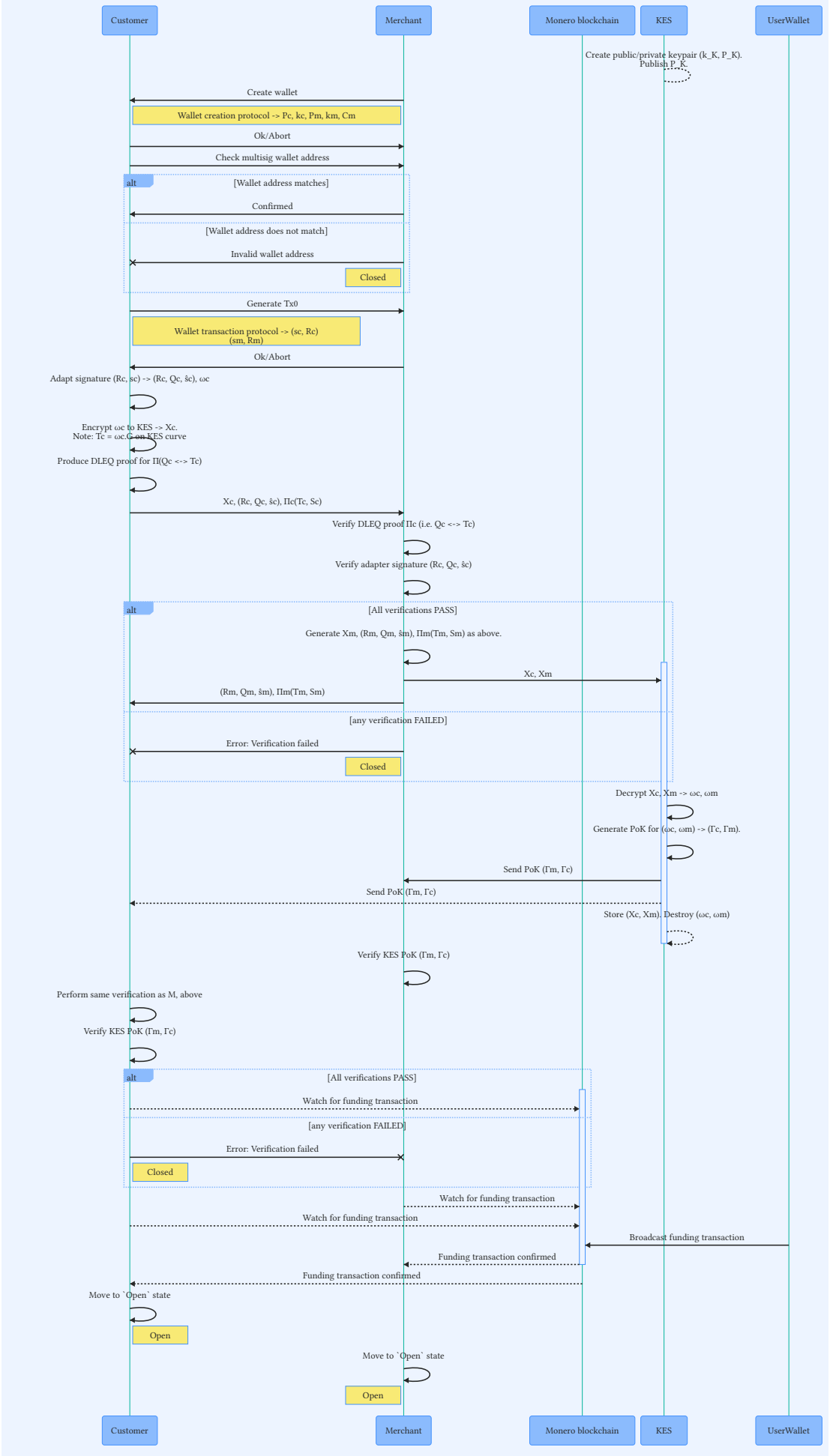
It is derived from as the first **32 bytes** of the **Blake2b-512** hash of the following transcript represented in hexadecimal format:

- The merchant public key, P_B, 32 bytes in little-endian byte order,
- The customer public key, P_A, 32 bytes in little-endian byte order,
- The merchant initial balance in piconero, as a 64-bit unsigned integer in little-endian byte order,
- The customer initial balance in piconero, as a 64-bit unsigned integer in little-endian byte order,
- A channel nonce, a 64-bit little-endian unsigned integer as the sum of
  - a 32-bit unsigned integer in little-endian byte order, randomly chosen by the customer, and
  - a 32-bit unsigned integer in little-endian byte order, randomly chosen by the 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 encrypts their initial adapter signature offset to 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
KES
UserWallet
Create public/private keypair (k_K, P_K).
Publish P_K.
Create wallet
Wallet creation protocol -> Pc, kc, Pm, km, Cm
Ok/Abort
Check multisig wallet address
alt
[Wallet address matches]
Confirmed
[Wallet address does not match]
Invalid wallet address
Closed
Generate Tx0
Wallet transaction protocol -> (sc, Rc)
(sm, Rm)
Ok/Abort
Adapt signature (Rc, sc) -> (Rc, Qc, ŝc), ωc
Encrypt ωc to KES -> Xc.
Note: Tc = ωc.G on KES curve
Produce DLEQ proof for Π(Qc <-> Tc)
Xc, (Rc, Qc, ŝc), Πc(Tc, Sc)
Verify DLEQ proof Πc (i.e. Qc <-> Tc)
Verify adapter signature (Rc, Qc, ŝc)
alt
[All verifications PASS]
Generate Xm, (Rm, Qm, ŝm), Πm(Tm, Sm) as above.
Xc, Xm
(Rm, Qm, ŝm), Πm(Tm, Sm)
[any verification FAILED]
Error: Verification failed
Closed
Decrypt Xc, Xm -> ωc, ωm
Generate PoK for (ωc, ωm) -> (Γc, Γm).
Send PoK (Γm, Γc)
Send PoK (Γm, Γc)
Store (Xc, Xm). Destroy (ωc, ωm)
Verify KES PoK (Γm, Γc)
Perform same verification as M, above
Verify KES PoK (Γm, Γc)
alt
[All verifications PASS]
Watch for funding transaction
[any verification FAILED]
Error: Verification failed
Closed
Watch for funding transaction
Watch for funding transaction
Broadcast funding transaction
Funding transaction confirmed
Funding transaction confirmed
Move to `Open` state
Open
Move to `Open` state
Open
Customer
Merchant
Monero blockchain
KES
UserWallet

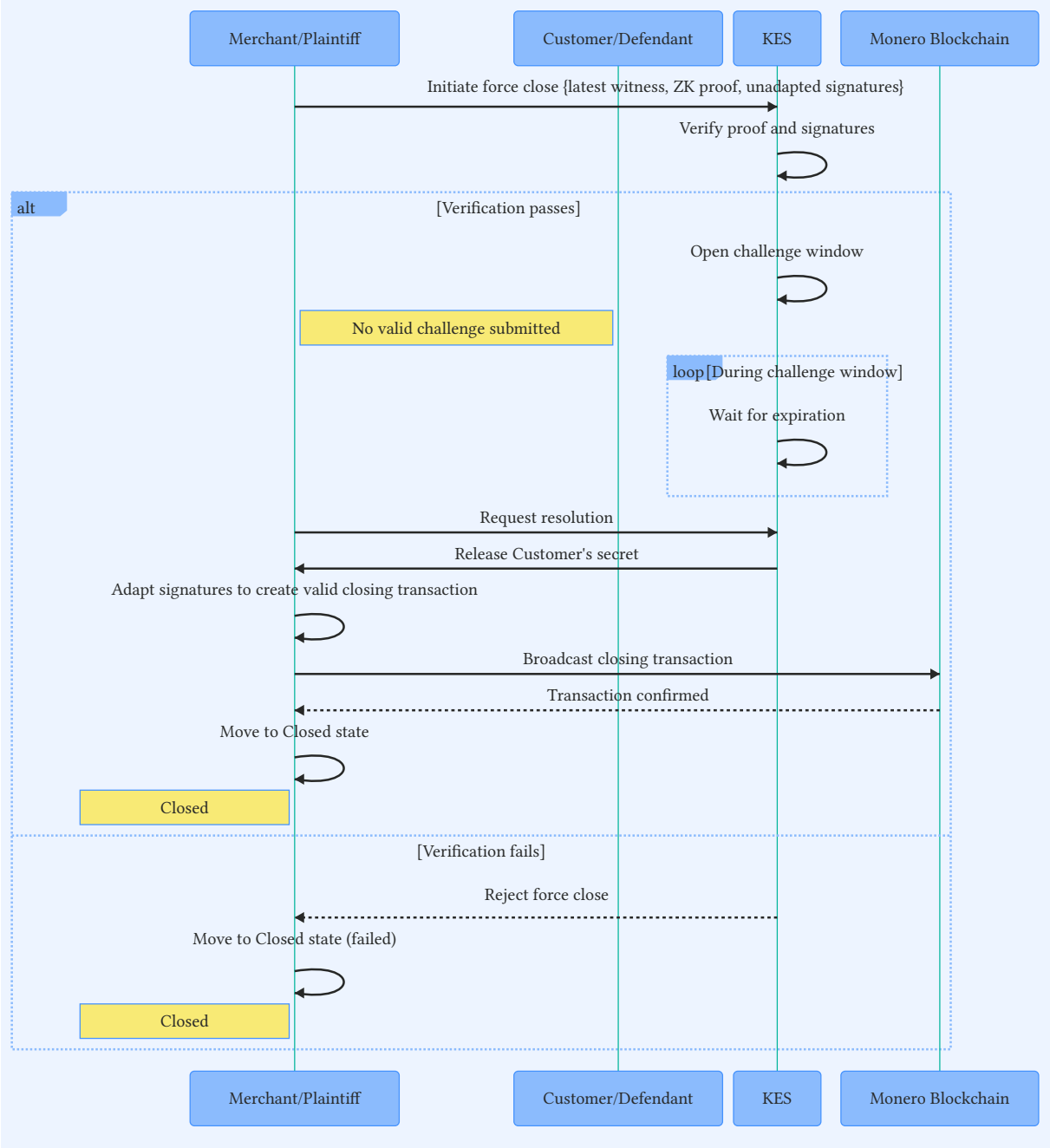
3.5. Co-operative Close

3.6. Channel Dispute

In the event of a dispute, such as when one peer becomes unresponsive or attempts to close the channel with an outdated state, the aggrieved peer initiates a force-close procedure. This process leverages the KES to ensure fair resolution and allows the wronged peer to reclaim funds according to the latest agreed channel state.

Disputing a channel requires the following steps:

1. The plaintiff (e.g., the Merchant) submits the latest witness, zero-knowledge proof, and unadapted signatures to the KES to initiate the force close.
2. The KES verifies the submitted proof and signatures.
3. If verification succeeds, the KES opens a challenge window to allow the defendant (e.g., the Customer) an opportunity to respond.
4. During the challenge window, the KES monitors for any valid challenge from the defendant, such as evidence of a newer channel state.
5. If no valid challenge is submitted and the window expires, the plaintiff requests resolution from the KES.
6. The KES releases the defendant's encrypted secret to the plaintiff.
7. The plaintiff adapts the signatures to form a valid closing transaction.
8. The plaintiff broadcasts the closing transaction to the Monero blockchain.
9. Once the transaction is confirmed on the Monero blockchain, the channel transitions to the Closed state.


Merchant/Plaintiff
Customer/Defendant
KES
Monero Blockchain
Initiate force close {latest witness, ZK proof, unadapted signatures}
Verify proof and signatures
alt
[Verification passes]
Open challenge window
No valid challenge submitted
loop [During challenge window]
Wait for expiration
Request resolution
Release Customer's secret
Adapt signatures to create valid closing transaction
Broadcast closing transaction
Transaction confirmed
Move to Closed state
Closed
[Verification fails]
Reject force close
Move to Closed state (failed)
Closed
Merchant/Plaintiff
Customer/Defendant
KES
Monero Blockchain

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,
10. verify the correctness of the closing transaction using the shared view key, the unadapted signatures and the adaptor statements,
11. 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,
12. verify the correctness of the funding transaction using the shared view key,
13. activate the KES with the root secrets,
14. 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 and enforce a dispute-response window of length Δ_{dispute}; accept and process defendant responses during this window, and if it expires without resolution, proceed with the unlock process.

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, ruling in favor of the plaintiff and proceeding with the unlock process.

The KES will 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 ω_0, 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],

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}.

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

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)
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)
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)

Output	Visibility	
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**†
- **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 channels 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 presents 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

Output	Visibility	
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. VerifyEquivalentModulo

5.2.1. 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)

5.2.2. 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_BabyJubJub)
ρ_{Ed}	Public	The Fiat–Shamir heuristic challenge response on the Ed25519 curve (response_div_ed25519)

5.2.3. 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}.

5.2.4. Methods

$$
\begin{aligned}
T_i &= \omega_i \cdot G_{\text{BJJ}} \\
S_i &= \omega_i \cdot G_{\text{Ed}} \\
C &= H_{\text{blake2s}}(\text{HEADER} \parallel \text{PACKED}(T_i) \parallel \text{PACKED}(S_i)) \\
\rho &= \omega_i * C - \nu_{\text{DLEQ}} \\
\rho_{\text{BJJ}} &= \rho \bmod L_{\text{BJJ}} \\
\Delta_{\text{BJJ}} &= \frac{\rho - \rho_{\text{BJJ}}}{L_{\text{BJJ}}} \\
\rho_{\text{Ed}} &= \rho \bmod L_{\text{Ed}} \\
\Delta_{\text{Ed}} &= \frac{\rho - \rho_{\text{Ed}}}{L_{\text{Ed}}}
\end{aligned}
\tag{2}
$$

5.3. VerifyDLEQ

5.3.1. 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)

5.3.2. 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)

5.3.3. 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 **VerifyEquiva-**

lentModulo 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.

5.3.4. Methods

$$
\begin{aligned}
C &= H_{\text{blake2s}}(\text{HEADER} \parallel \text{PACKED}(T_i) \parallel \text{PACKED}(S_i)) \\
\text{P}_{\text{BJJ}} &= \rho_{\text{BJJ}} \cdot G_{\text{BJJ}} \\
C_{T_i} &= C \cdot G_{\text{BJJ}} \\
R_{\text{BJJ}} &= C_{T_i} - \text{P}_{\text{BJJ}} \\
\text{P}_{\text{Ed}} &= \rho_{\text{Ed}} \cdot G_{\text{Ed}} \\
C_{S_i} &= C \cdot G_{\text{Ed}} \\
R_{\text{Ed}} &= C_{S_i} - \text{P}_{\text{Ed}}
\end{aligned}
\tag{3}
$$

5.3.5. Summary

The outputs are used for the further **VerifyEncryptMessage** operation.

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

5.3.6. Methods

$$
\begin{aligned}
T_0 &= \omega_0 \cdot G_{\text{BJJ}} \\
c_1 &= a_1 \cdot G_{\text{BJJ}} \\
\sigma_1 &= -(\omega_0 + a_1) \bmod L_{\text{BJJ}} \\
\sigma_2 &= 2 * \omega_0 + a_1 \bmod L_{\text{BJJ}}
\end{aligned}
\tag{4}
$$

5.4. VerifyEncryptMessage

5.4.1. 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

5.4.2. 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)

5.4.3. 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:

$$(\Pi, \sigma) = \mathrm{DecryptMessage}(\kappa, \Phi, \chi) \tag{5}$$

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}.

5.4.4. Methods

$$\begin{aligned} \Phi &= \nu \cdot G_{\mathrm{BJJ}} \\ \nu_{\Pi} &= \nu \cdot \Pi \\ C &= H_{\mathrm{blake2s}}(\nu_{\Pi}.\mathrm{x} \parallel \nu_{\Pi}.\mathrm{y}) \\ s &= C \bmod L_{\mathrm{BJJ}} \\ \chi &= \sigma + s \bmod L_{\mathrm{BJJ}} \end{aligned} \tag{6}$$

5.5. DecryptMessage

5.5.1. 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)

5.5.2. Outputs

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

5.5.3. 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}.

5.5.4. Methods

$$\begin{aligned} \Pi &= \kappa \cdot G_{\mathrm{BJJ}} \\ \kappa_{\Phi} &= \kappa \cdot \Phi \\ C &= H_{\mathrm{blake2s}}(\kappa_{\Phi}.\mathrm{x} \parallel \kappa_{\Phi}.\mathrm{y}) \\ s &= C \bmod L_{\mathrm{BJJ}} \\ \sigma &= \chi - s \bmod L_{\mathrm{BJJ}} \end{aligned} \tag{7}$$

5.6. VerifyCOF

5.6.1. Inputs

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

5.6.2. 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)

5.6.3. 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}.

5.6.4. Methods

$$\begin{aligned} T_{i-1} &= \omega_{i-1} \cdot G_{\mathrm{BJJ}} \\ C &= H_{\mathrm{blake2s}}(\mathrm{HEADER} \parallel \omega_{i-1}) \\ \omega_i &= C \bmod L_{\mathrm{BJJ}} \\ T_i &= \omega_i \cdot G_{\mathrm{BJJ}} \end{aligned} \tag{8}$$

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. The Key Escrow Service (KES) Design

7.1. Introduction

In short, the KES will hold an encrypted secret, ω_0, from each of 2 parties. Either party can request their counterpart's secret if certain conditions are met.

Therefore the KES needs a keypair for each channel (k_g, P_g) under which both parties can encrypt their secrets and send it to the KES for safe-keeping using `EncryptMessage` (Algorithm 1). If the conditions are satisfied, the KES can decrypt ω_0 and send it to the claimant.

The KES only participates in the channel during the initialization, force-close and, dispute stages. It is not active in the update and co-operative close phases. The KES also has an optional cleanup step after a channel has closed. Table 7 summarizes the KES' role throughout the channel lifecycle.

Table 7: The role of the KES during the channel lifecycle

Stage	KES required	KES role
Initialization	Yes	Securely store ω_0 for each party.
Update	No	
Co-operative close	No	
Closed	Recommended	Discard ω_0 and other cleanup.
Force Close	Yes	Deliver counterparty ω_0 after dispute window closes.
Dispute	Yes	Deliver ω_0 to counterparty if dispute conditions are satisfied.

7.2. Preliminaries

Let $\mathbb{G}_1$ and $\mathbb{G}_2$ be two additive groups of points on elliptic curves $\mathbb{E}_1$ and $\mathbb{E}_2$, respectively.

$\mathbb{E}_1$ is defined over a prime finite field $\mathrm{GF}_1(p_1)$, with p_1 a large prime. Let N_1 be the size of the group generated by the Generator point, G_1 for group $\mathbb{G}_1$, such that $N.G \equiv \mathcal{O}$.

$\mathbb{E}_2$ is a SNARK-friendly curve and is similarly defined.

$H_F(...)$ is a one-way function under the random oracle assumption that hashes the binary representation of its input to a scalar element in the group order. Domain separation and other hygiene considerations are assumed to be properly taken care of.

Similarly, $H_P(...)$ is a one-way function under the random oracle assumption that hashes the binary representation of its input to a group element in $\mathbb{G}$.

The KES can utilize $\mathbb{E}_1$, $\mathbb{E}_2$, or both, depending on where it's deployed and which dispute resolution option (Section 7.8) is selected. The VCOF utilizes a ZK-SNARK, and will therefore utilize $\mathbb{E}_2$.

`EncryptMessage`. $P = k \cdot G$ is the recipient's public key. m is a scalar to be encrypted

$(R, \chi) = \text{EncryptMessage}(m, P, \mathbb{G}, \mathbb{F})$
Select random scalar element, r where $0 < r < N$.
Calculate:
  $R = r \cdot G$
  $P_s = r \cdot P (\equiv rk \cdot G)$
  $s = H_F(P_s)$
  $\chi = (m + s) \bmod N$
Return (R, χ).

Algorithm 1: The `EncryptMessage` algorithm.

`DecryptMessage`. $P = k \cdot G$ is the recipient's public key. m is the encrypted scalar

$m = \text{DecryptMessage}(R, \chi, \mathbb{G}, \mathbb{F})$
Calculate:
  $P_s = k \cdot R (\equiv rk \cdot G)$
  $s = H_F(P_s)$
  $m = (\chi - s) \bmod N$
Return m.

Algorithm 2: The `DecryptMessage` algorithm.

In Grease, $\mathbb{E}_1$ is always Curve25519[8], the curve behind Monero. $\mathbb{E}_2$ can be any suitable curve. For the reference implementation of Grease, $\mathbb{E}_2$ is the BabyJubJub curve.

7.3. Global keys

We denote the curve that the KES operates on (whether $\mathbb{E}_1$ or $\mathbb{E}_2$) as $\mathbb{E}_K$.

On creation, the KES creates a new, master keypair (k_K, P_K) on $\mathbb{E}_K$. P_K is made available publicly.

7.4. Channel Encryption keys

When a merchant-customer pair create a new channel, they create an ephemeral channel id secret on $\mathbb{E}_K$, κ, using Algorithm 3.

1. The merchant (or customer, it doesn't matter) then encrypts κ to the KES using `EncryptMessage` (see Algorithm 1) on $\mathbb{E}_K$.
2. The KES calculates the unique channel keypair for $\mathbb{E}_K$:

$$\begin{aligned} k_g &= \kappa \cdot k_K \\ P_g &= k_g \cdot G \end{aligned} \tag{9}$$

3. The KES shares P_g with both parties.
4. **Critical**: The KES discards κ.

ECDH-MC

1 $k_s = \hat{k}_a \hat{P}_b \left(\equiv \hat{k}_b \hat{P}_a \right)$ on $\mathbb{E}_K$
2 $\kappa \leftarrow H_F(\text{ks, channel\_id})$ on $\mathbb{E}_K$
3 $P_\kappa \leftarrow \kappa \cdot G$ on $\mathbb{E}_K$

Algorithm 3: Diffie-Hellman shared secret derivation for new channels

> **Important**
>
> $\hat{k}_a$ **must** be chosen at random for *every* new channel a party creates. Failure to do so may result in loss of funds.

Even though $\hat{k}_a$) must be unique for every channel opening, clients do not have to store a large database of keys (although the can). One could use a **master key**, k_a and derive unique channel keys using a simple deterministic scheme,

$$\hat{k}_a = H_F(\text{k\_a, channel\_id}) \tag{10}$$

Since the channel id is already a function of the counterparty public keys, the initial balances, and a nonce, it will be unique for each channel.

7.4.1. Security properties

The channel keys have the following properties:

- If k_g becomes compromised, the channel is compromised, but other current and future channels are still safe, since the attacker does not know κ for the other channels.
- If k_K becomes compromised, channels are *still* safe, as long as the κ for any given channel is unknown. This is why is it critical to discard κ after deriving the keys. While the damage would be contained, a compromised k_K is incredibly serious and would require the KES to start afresh with a new master keypair and inform all parties to close open channels immediately.

7.5. Communication with the KES

Either, or both parties can carry out the communication duties with the KES, since none of the communication protocols with the KES require trust between the counterparties. Every protocol provides sufficient cryptographic proof that the steps were completed correctly.

A working assumption is that the Merchant typically has more reliable internet access and possesses a more powerful machine (typically a server) than the Customer (typically a mobile phone).

Thus, for the sake of brevity, it is assumed that the Merchant acts as proxy for all KES communications. But it is certainly possible for the Customer, or both parties to perform the same tasks, with the same, idempotent result.

7.6. Channel Initialization

The Merchant (M) and Customer (C) partially sign a 2-of-2 multisig transaction with an adaptor signature. Either party needs to know their counterpart's adapter signature offset, ω_0 to complete the signature and spend the funds out of the multisig wallet, thus closing the channel.

The primary role of the KES during channel initialization is to store the encrypted ω_0 values for each party. The full procedure for calculating ω_0 is described in Section 4.2.1.3.

It is important to note that the adapter signature offsets, ω_0^C and ω_0^M and their corresponding points, Q_0^C and Q_0^M are elements of $\mathbb{E}_1$, the Ed25519 curve. For the KES to work with them, they need to be translated onto $\mathbb{E}_K$.

The offsets are chosen such that they are valid scalars on both curves, that is

$$0 < \omega_0 < \min(N_1, N_2) \tag{11}$$

The group element on $\mathbb{E}_1$ corresponding to ω_0 is Q_0, while the corresponding point on $\mathbb{E}_K$ is denoted T_0. A summary of the points and their equivalents is given in Table 8.

> **Note**
>
> It is possible that $\mathbb{E}_K$ is also Curve25519. This does not change the requirements or procedures. However, many things are much simpler. The DLEQ proofs are trivial, for instance.

7.6.1. What the parties send to the KES

Each party computes a discrete-log equivalence proof (DLEQ), Π_0 that proves that Q_0 and T_0 are generated from the same scalar, ω_0.

The Customer encrypts ${\omega_0}_C$ to the KES using `EncryptMessage` (Algorithm 1) and sends the following package to the Merchant:

- the encrypted offset, χ_C
- The channel id, `id` (Section 3.3.1),
- The Customer's public key, P_A,
- The length of the dispute window, `dw` in seconds. The default is 86,400 (24hrs),
- A payload signature signing `id` , χ_C, `dw` , and T_0^C with $\hat{k}_a$
- The DLEQ proof, Π_0^C.

The merchant validates Π_0^C and forwards the rest of the package to the KES.

The Merchant performs the analogous tasks and forwards both his and the customer's data to the KES. He also sends his DLEQ proof, Π_0^M the the Customer, who validates it.

7.6.2. What the KES send back to parties

Once the KES has received both packages, it performs the validation procedure outlines in Algorithm 4

Open Channel validation

1 Validate the payload signatures.
2 Decrypt $\chi_C \rightarrow \omega_0^C$ and $\chi_M \rightarrow \omega_0^M$.
3 Are dispute windows, `dw`, values equal?
4 | No? **Return** Fail(`InvalidConfiguration`)
5 Store an `OpenChannel` record in private/encrypted storage.
6 Calculate proof-of-knowledge proofs, Γ_C and Γ_M, for ω_0^C and ω_0^M respectively.
7 Send the PoK proofs to the Merchant and/or Customer.
8 Discard ω_0^C and ω_0^M.

Algorithm 4: The KES validates each new channel request, returning proofs of knowledge if successful.

7.6.3. `OpenChannel` record

`OpenChannel` records the state the KES needs to carry out its duties for every channel:

- The channel id, `id`
- The dispute window, in seconds (default: 84,600)
- The proofs-of-knowledge, Γ_C and Γ_M
- The merchant public key, P_B and encrypted initial offset χ^M,
- The customer public key, P_A and encrypted initial offset χ^C,

> **Caution**
>
> Don´t store any other metadata, initial balances or anything else that would reduce privacy and increase attack surface.

> **Important**
>
> Notwithstanding other checks that are required, a party **must not** allow channel opening to proceed without verifying both PoK proofs. For the most part, the Customer can proxy communication with the KES through the merchant, but it is **recommended** that the Client at least queries the KES directly to obtain these proofs.

7.6.4. Failure scenarios

The Customer trusts the Merchant to relay messages to the KES faithfully. Let's examine scenarios where this breaks down:

- The Merchant never relays any messages: The Customer never receives a PoK for ω_0 and refuses to sign the channel opening.
- The Merchant uses a different KES that he controls: The KES cannot decrypt χ_C, since it is encrypted to the initially agreed-upon KES.
- The KES colludes with the merchant and gives him ω_0^C: **Failure**: This allows the Merchant to spend all funds out of the multisig wallet. This is a known failure mode in both Grease and AuxChannel. if the KES operates as a trusted third party, then the risk of collusion

is governed by that degree of trust. It is preferable therefore to have the KES deployed as a publicly auditable contract on a Zero-Knowledge blockchain. The keys can be stored according to the procedure described above without revealing their values and collusion then requires subverting the entire blockchain.

- The Merchant follows the procedure as described, except that it withholds the PoKs from the Customer and immediately triggers a force-close: After the dispute window ends, the channel closes, allowing the return of just the original balances to each party. The Merchant is unable to construct any valid Monero transaction besides reversing the original funding transaction.
- The Merchant follows the procedure as described, except that it withholds the PoKs and goes offline indefinitely: The Customer, after some timeout period, can query the KES directly, retrieve the PoKs, and trigger a force-close herself. She can get her funds back after the dispute window closes.

Table 8: Equivalent curve points in grease

SCALAR	$\mathbb{E}_1$	$\mathbb{E}_2$	LABEL
ω_0	Q_0	T_0	Initial adapter signature offset

7.7. Channel Force Close

7.7.1. The original Monet/AuxChannel proposal

Suppose Bob (P_B) wants to close the channel at state i, but Alice (P_A) is unresponsive.

1. Bob posts a "trigger transaction" to the KES smart contract on L2. This transaction includes his decryption key k_{di}^B, the state index i, and the L2 signatures $\sigma_i'^A, \sigma_i'^B$. This starts a timer.
2. Alice has a time window to react. She has two options:
   - **If state i is the latest, correct state:** She cooperates by posting a "release transaction", revealing her decryption key k_{di}^A. Bob can then use this key to recover Alice's signature and close the channel on L1. This option has the same outcome as the co-operative close but with the added overhead of involving the KES and paying L2 gas fees. It is significantly cheaper for Alice and Bob to co-operate.
   - **If state i is outdated (e.g., Bob is trying to use an old state):** Alice posts an "update transaction" on L2, where $j > i$ is the actual latest state index. This transaction proves that Bob's claim is stale. To verify the proof, the KES is required to execute the VCOF $j - i$ times to verify Alice's claim. Depending on the nature of the L2 this may or may not be an expensive operation.
3. **Timeout:** If Alice does not respond within the time window, the KES contract automatically releases her **initial** decryption key k_{d0}^A on L2.
4. **Resolution:**
   - If Alice released k_{di}^A, Bob can close the channel at state i.
   - If the KES released k_{d0}^A, Bob can use the `KeyUpdate` function to sequentially derive all subsequent keys $\left(k_{d1}^A, \ldots, k_{di}^A\right)$ up to the state he claimed. He can then recover Alice's signature for state i and close the channel. This mechanism also acts as a punishment: if Alice was the malicious party trying to force an old state, Bob can now use her released

initial key to derive the decryption key for any **later** state $j > i$ that is favorable to him, and close the channel at that state, stealing her funds.

Grease deviates from this original design in several ways:

- No transactions are shared with the KES. These are replaced with simple digitally-signed state messages.
- The protocol is generalized with the introduction of the channel id. This permits all parties to handle multiple concurrent channels.
- The dispute process uses a simple update count to significantly reduce the computational load on the KES.

7.8. Force-closing in Grease

Either party in the channel, the *claimant*, can trigger a force-close at any time. In short, the claimant informs the KES about the intent to force close, along with the current update count of the channel, n.

If the claim is valid, the KES starts the dispute window. If the counterparty, the *defendant*, has not presented a dispute proof before the window closes, the KES either

- encrypts and sends the defendant's ω_0 to the claimant (Option I).
- calculates the signature offset, ω_n and sends it to the claimant (Option II).

The data the claimant receives is a configuration option and is specific to the KES. In general, if the VCOF is very fast and cheap for the KES, then consider Option II. Otherwise, default to Option I. The biggest benefit of Option I combined with the dispute resolution improvement of the Grease protocol over AuxChannel is that the KES can be completely ignorant of the VCOF.

> **Important**
>
> The Grease design allows the KES to be completely independent and ignorant of the VCOF.

This is extremely useful if generating ZK-SNARKS on the KES platform is impossible, or very expensive. It also allows us to upgrade the VCOF on channels without having to change KES executables, which on blockchains is a non-trivial exercise.

A more detailed description of the process is as follows.

If a claimant wishes to initiate a force close they send a `ForceCloseRequest` message to the KES which triggers an `onForceCloseRequestEvent` event. The KES responds to the request event with the `HandleForceCloseRequest` procedure, outlined in Algorithm 5.

The claimant can follow up with a `ClaimChannelRequest` message after the dispute window has closed. When the KES receives a `ClaimChannelRequest`, it triggers an `OnClaimChannelEvent` event.

The KES responds to this event with the `HandleClaimChannelRequest` algorithm, Algorithm 6.

Finally when the KES receives an `AckChannelClaimedEvent(channel_id)` signed by P_c it can permanently delete the corresponding `PendingChannelClose` record. It can also perform any other

`onForceCloseRequestEvent(req: ForceCloseRequest)` :

```
1  Validates the signature in req.
2  if valid:
3      Fetch the OpenChannel (Section 7.6.3) record with id == req.id => channel
       (Only allow channel parties to initiate a force-close)
4      if exists(channel) && req.Pc in [channel.Pa, channel.Pb] :
5          Create a new PendingChannelClose entry with status = Pending .
6          try to send a copy of the PendingChannelClose to the counterparty.
7          return Ok( ForceCloseResponse )
8      else
9          return Fail( FailReason )
10 else
11     return Fail( InvalidSignature )
```

Algorithm 5: The KES `HandleForceCloseRequest` algorithm.

cleanup relating to the channel. At this point, the channel is closed and no trace of it remains on the KES.

`OnClaimChannelEvent(req: ClaimChannelRequest)` :

```
1  Validates the signature in req.
2  if valid:
3      Fetch PendingChannelClose where id == req.id => pending
4      if !exists(pending) return Fail(No such channel)
5      if req.claimant != id.req.claimant return Fail(Unauthorized)
6      if current time ≥ pending.expiry
7          pending.req.defendant ⇒ P_d
8          pending.req.claimant ⇒ P_c
9          using Algorithm 2, decrypt χ for pending.offsets where public key is P_c and ⇒ ω_0.
10         Case claim option:
11             Option I: ω_0 ⇒ ω
12             Option II: using Section 4.2.2.3 for n = pending.req.update_count_claimed , calculate ω_n ⇒
               ω
13         encrypt ω for P_c ⇒ χ_c using Algorithm 1
14         update pending , set status to ForceClosed , add (Pc, Xc, n) to chi
15         try send χ_c to claimant
16     else return Fail("Dispute window still open")
17 else return Fail( InvalidSignature )
```

Algorithm 6: The KES `HandleClaimChannelRequest` algorithm.

7.8.1. `ForceCloseRequest` message

A party sends a `ForceCloseRequest` message to initiate the force-close process.

```
pub struct ForceCloseRequest<Curve> {
  /// The globally unique channel id
  channel_id: ChannelId,
  /// The public key of the party initiating the force close, $P_c$.
  claimant: PublicKey,
  /// The public key of the counterparty, $P_d$,
  defendant: PublicKey
  /// The most recent claimed state update count
  update_count_claimed: u64,
  /// A signature under $P_a$ for `id | update_count_claimed | Pc | Pd"
  signature: SchnorrSignature,
}
```

7.8.2. `PendingChannelClose` message

The KES must track all channel that are under dispute.

```
pub struct PendingChannelClose {
  /// The globally unique channel id
  channel_id: ChannelId,
  /// A timestamp for when the dispute window closes
  expiry: u64,
  /// Initial offsets (encrypted)
  chi0: Vec<(PublicKey, EncryptedOffset)>
  /// A copy of the close request that intitiated the dispute
  req: ForceCloseRequest
  /// The status of the force-close channel
  status: PendingCloseStatus
  /// Encrypted offsets available for parties to close the channel
  chi: Vec<(PublicKey, EncryptedOffset, UpdateCount)>
}

pub enum PendingCloseStatus {
  /// The channel is pending force-closure and the dispute window is open.
  Pending,
  /// The dispute window is closed, and the channel is claimable by the claimant
  Claimable,
  /// The channel is abandoned. It is now claimable by either party
  Abandoned,
  /// The channel was closed via consensus and X value(s) are available for download.
  ConsensusClosed,
  /// The channel was claimed via force close,
  ForceClosed,
  /// The channel was claimed after being abandoned
  AbandonedClaimed,
  /// The channel was closed after a successful dispute
  DisputeSuccessful,
}
```

7.8.3. `ClaimChannelRequest` message

Once the dispute window has closed a claimant can claim their counterpart's ω_0 with a `ClaimChannelRequest` message:

```
pub struct ClaimChannelRequest {
  /// The globally unique channel id
  channel_id: ChannelId,
```

```
    /// The public key of the claimant, `P_c`
    claimant: PublicKey,
    /// A signature under $P_c$ for `id | Pc`"
    signature: SchnorrSignature
}
```

The KES responds to a `ClaimChannelRequest` message by triggering an `onClaimChannelRequestEvent` event. It is handled by `handleClaimChannelRequest` as described in Algorithm 6.

A `ClaimAbandonedChannelRequest` message is identical to `ClaimChannelRequest`. However, its semantics are different:

- Either claimant **or** defendant may sign a `ClaimAbandonedChannelRequest` message.
- The KES will only allow the claim if if `current_time >= pending.expiry + req.dw`, i.e. typically an additional 24 hours after the dispute window closes.
- After a successful claim, `pending.status` is set to `AbandonedClaimed`.

This message handles the case where a merchant has initiated a force-close but then has stopped responding. Without the ability to claim an abandoned channel, Alice has no way to recover her funds.

In summary, the various windows during the dispute phase are:

Time	Valid signers	Valid Messages
$t_{\text{close}} \longrightarrow$	No-one. Dispute window	
$t_{\text{close}} + \text{dw} \longrightarrow$	Claimant	`ClaimChannelRequest`
$t_{\text{close}} + 2\text{dw} \longrightarrow t_D$	Either	`ClaimChannelRequest` (Claimant) `ClaimAbandonedChannelRequest` (Either)
$t_D \longrightarrow \infty$	No-one. Record deleted	

> **Note**
>
> In the "abandoned" window, that is, between the closing of the dispute window and the record deletion (Section 7.10), the claimant can claim the channel using either type of message, but it is recommended to use `ClaimChannelRequest`.

7.9. Channel Dispute

Once a party (the defendant) becomes aware that there is a force-close in progress on one of her channels, she can:

1. Dispute the force close, and submit the data that would have closed the channel cooperatively,
2. Dispute the force close, proving a more recent channel state than the force-close state,
3. Do nothing and let the dispute window close.

7.9.1. Force close with consensus state

If Alice cannot prove a more recent state, under Option II, she can choose to do nothing at no cost to herself.

Under Option I, if there exists a prior channel state that is more disadvantageous to her than the claimed closing state, Alice is incentivized to publish a `ConsensusCloseRequest` (Section 7.9.1.1) which will close the channel in the state submitted by Bob in the `ForceCloseRequest`.

> **Note**
>
> It is strictly more expensive and time-consuming for both parties to close a channel under consensus rather than a co-operative close (Section 3.5), with exactly the same result.

When the KES receives a `ConsensusCloseRequest` message, it triggers an `onConsensusCloseRequestEvent` which leads to the `handleConsensusCloseRequest` methods carrying out Algorithm 7.

7.9.1.1. `ConsensusCloseRequest` message

The defendant submits a signed χ_n to allow the claimant to complete the refund transaction under the `update_count_claimed` in their `ForceCloseRequest`.

```
pub struct ConsensusCloseRequest {
  /// The globally unique channel id
  channel_id: ChannelId,
  /// The public key of the party initiating the force close, `P_c`
  claimant: PublicKey,
  /// The public key of the counterparty, $P_d$,
  defendant: PublicKey
  /// The most recent claimed state update count
  update_count_claimed: u64,
  /// The encrypted offset for state n, chi_n
  encrypted_offset: Scalar,
  /// A signature under $P_d$ for `id | update_count_claimed | Pa | Pb"
  signature: SchnorrSignature,
}
```

7.9.2. Dispute with recent state proof

The `ForceCloseRequest` message† contains a signature from both parties signing the update count. This is the same information that parties exchange on *every* channel update. Therefore, it is now very straightforward for the KES to verify a dispute.

`handleConsensusCloseRequest(req: ConsensusCloseRequest)`

1 if `!req.signature` valid, **return** Fail(`Unauthorized`)
2 if `exists(req.channel_id)` in `PendingChannelClose` records as `p` :
3 | if `p.defendant != req.defendant` or `p.claimant != req.claimant` , **return** Fail(`Invalid`)
4 | if `p.update_count_claimed != req.update_count_claimed` , **return** Fail(`Invalid`)
5 | Decrypt `req.encrypted_offset` $\rightarrow \omega_n$ (Algorithm 2), else **return** Fail(`Invalid`)
6 | Encrypt $\omega_n \rightarrow \chi_n^D$ under P_D (Algorithm 1)
7 | Update `p` : set `p.chi_D =` χ_n^D, `p.status = ConsensusClosed`
8 | Try send χ_n^D to defendant.
9 else **return** Fail(`NotFound`)

Algorithm 7: Resolving a channel force-close via consensus

1. Alice provides a `DisputeChannelState` message with an `UpdateRecord` that has been signed by Bob:

```
pub struct DisputeChannelState {
  /// The globally unique channel id
  channel_id: ChannelId,
  /// The public key of the claimant, $P_c$
  claimant: PublicKey,
  /// The public key of the defendant, $P_d$,
  defendant: PublicKey
  /// The channel update count
  update_count: u64,
  /// Other data not relevant to the KES, but included in the signed message
  ...
  /// A signature under $P_b$ for `id | update_count | Pa | Pb | ..."
  update_record: SchnorrSignature,
  /// This record signed by the defendant
  signature: SchnorrSignature,
}
```

The KES makes the following checks:

1. `update_count` > `n_claimed`, i.e. that Alice's proof is for a more recent state than Bob's force-close request.
2. P_a and P_b are identical in both the force-close request and dispute channel state messages.
3. The `signature` in the dispute channel state message is valid and signed by the defendant.
4. The `update_record` signature is valid and signed by the claimant.

If all three checks pass, the KES decrypts ω_0^B with Algorithm 2, encrypts it to P_A using Algorithm 1 and sends the encrypted ω_0^B to Alice.

7.9.3. Do Nothing

Alice can do nothing, in which case the dispute window will eventually close and Bob can claim the missing signature offset via Algorithm 6.

7.10. Channel cleanup

7.10.1. After co-operative close

- The merchant will deliver proof that the channel has closed. This proof consists of a `ChannelClose` message corresponding to the channel with id `id` signed by each of the Merchant and Customer.
- If, and only if, both signatures are valid, the KES **must**
  - destroy all data associated with channel `id` insofar as it is possible.
  - return any resources (e.g. deposits or collateral) belonging to either party that were being held as part of the KES, net of any fees associated with maintaining the KES.

7.10.2. Hygiene

To maximize privacy, the KES will delete old channels after a reasonable storage period. t_D is configurable by the KES, but should typically be one month after the claim period starts, i.e.

$$t_D = \texttt{expiry} + 2\,\texttt{dw} + \texttt{one\_month} \tag{12}$$

Periodically, the KES will scan all records in `PendingChannelClose` and

- delete all records where `current_time` > t_D
- where `current_time > expiry + 2dw`, set `status = Abandoned`
- where `current_time > expiry + dw`, set `status = Claimable`

7.11. Utility functions

The KES must also expose the following utility functions:

- `getChannelDefinition(channel_id, auth)`
- `getPendingClose(channel_id, auth)`

This allows the customer to query the channel lifecycle independently of the merchant. In both cases authorization consists of a Schnorr signature signing the channel id and requester's public key.

- If the signature is valid, but the channel record does not exist, the KES returns "Not found".
- If the signature is invalid, the KES returns "Unauthorized".
- If the channel record exists, **and** the signer's public key is recorded in the record, return the record. Otherwise return "Not found".

Table of Algorithms

8. Nomenclature

8.1. 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

8.2. 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.
8. Daniel J. Bernstein. Curve25519: new Diffie-Hellman speed records. February 9, 2006. https://cr.yp.to/ecdh/curve25519-20060209.pdf°

Index of Tables