Grease: A Private Payment Channel Protocol for Monero

Grease Team

January 21, 2026

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 absence 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 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. Monet provides a sketch of a payment channel design but many details were omitted or elided. Grease is a full-fledged implementation with full consideration given to state management, multiple concurrent channels, peer-to-peer communication and encryption and practical performance.

Every update and the final closure of the channel require an online interaction between channel parties. 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 features when it comes to 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 offset) 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 generate a valid Monero transaction and recover funds from the channel. Only valid channel states can be unilaterally closed; fabricated updates cannot be simulated. Section 4 has full details on the KES design and implementation.

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 "customer") to the other (the "merchant"). This designation is somewhat arbitrary, either party in a channel may assume either role. Most channels will have participants that are easily identifiable in either the customer or merchant role and we use that terminology throughout this document.

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

- The merchant is responsible for recording the channel state on the ZK chain.
- The merchant 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 customer will need a small quantity of 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 customer to the merchant. However, in instances where this is not the case, the customer 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 customer opens a channel with m XMR and the merchant starts with a zero XMR balance (since the merchant is providing assets or services in exchange for Monero).
- 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 of an abandoned channel.

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. They're usually all provided 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 a *commitment transaction*, so-called for reasons that will be made clear later, 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 they will be able to spend the funds if a small piece of missing data is provided[1]. 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 responsible for arbitrating disputes. It is ideally a permissionless, decentralized private smart contract, but it can be a centralized 3rd-party service as well. 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.

[1] These signatures are called adapter signatures.

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**
>
> Once 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 information 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 data describing the channel, including, the initial balance, public keys and closing addresses (See Section 3.3.1). If both parties are satisfied with the proposed channel parameters, the channel moves to the `Establishing` state.
- `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. At each update, both parties cryptographically *commit* to the new state of the channel and collaborate to produce a new, partially-signed commitment transaction.
- `Closing` - The channel is being closed. This phase includes sharing of adapter secrets and signing of the final commitment transaction. 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.
- `Closed` - The channel is closed. The merchant **should** inform the KES of the closure so that it can clean up any state associated with the channel. A channel can reach the `Closed` state after a co-operative close, following a resolved dispute, or after several error conditions arising during the `New` and `Establishing` phases.
- `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 dispute 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, submit the correct signature offset, ω_n, or do nothing and allow the counterparty to use your ω_0 to recover any previous

channel state. The final state transition is always to the `Closed` state, only the reason can vary. See Section 4 for details.

3.3. New Channel

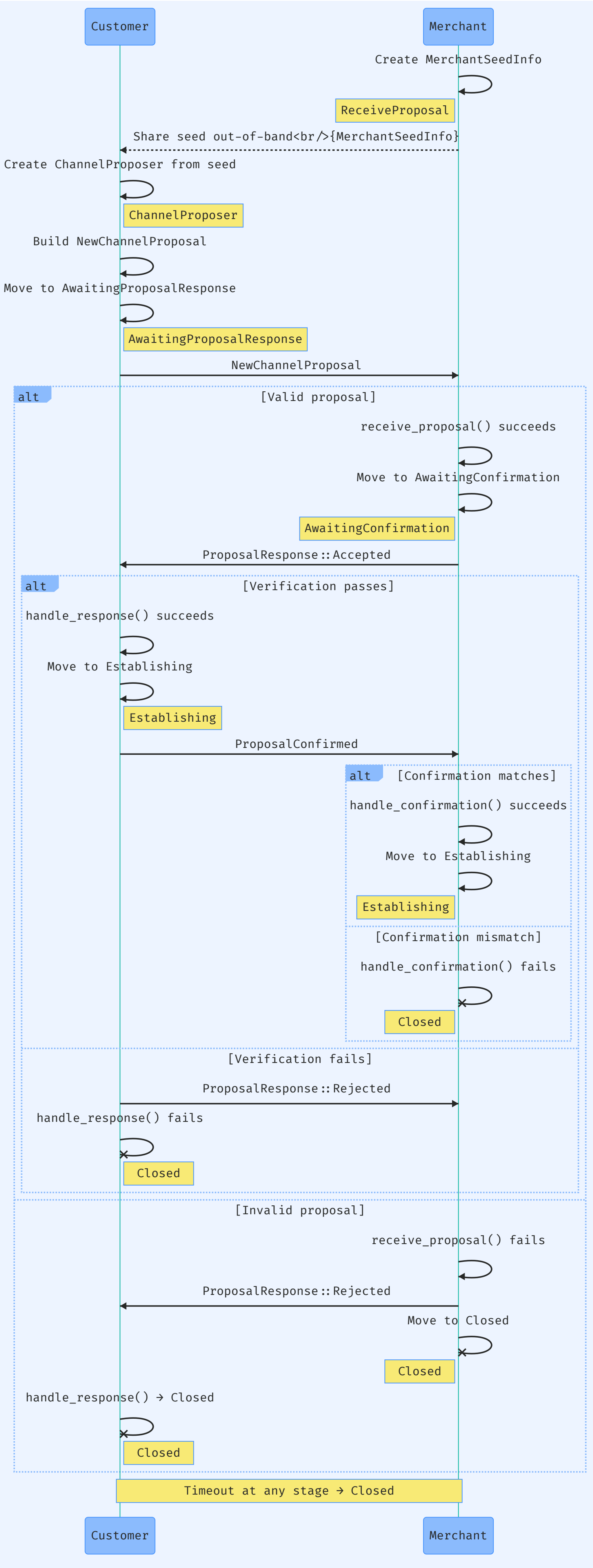
A new channel is established when a Merchant shares some initialization data with a Customer out-of-band.

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[2]:

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 random nonce id,
     - the merchant's closing address,
     - the requested initial balances,
     - the merchant's public key,
     - the dispute window duration
   - 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

[2] See `server.rs:customer_establish_new_channel`



Listing 1: The New Channel proposal sequence

3.3.1. Channel Id

The channel id is a 65 character hexadecimal string that uniquely identifies the channel. It is defined as the prefix "XGC", followed by the first 31 bytes of the **Blake2b-512** hash of the channel metadata in hexadecimal format.

The hash is calculated from the following transcript:

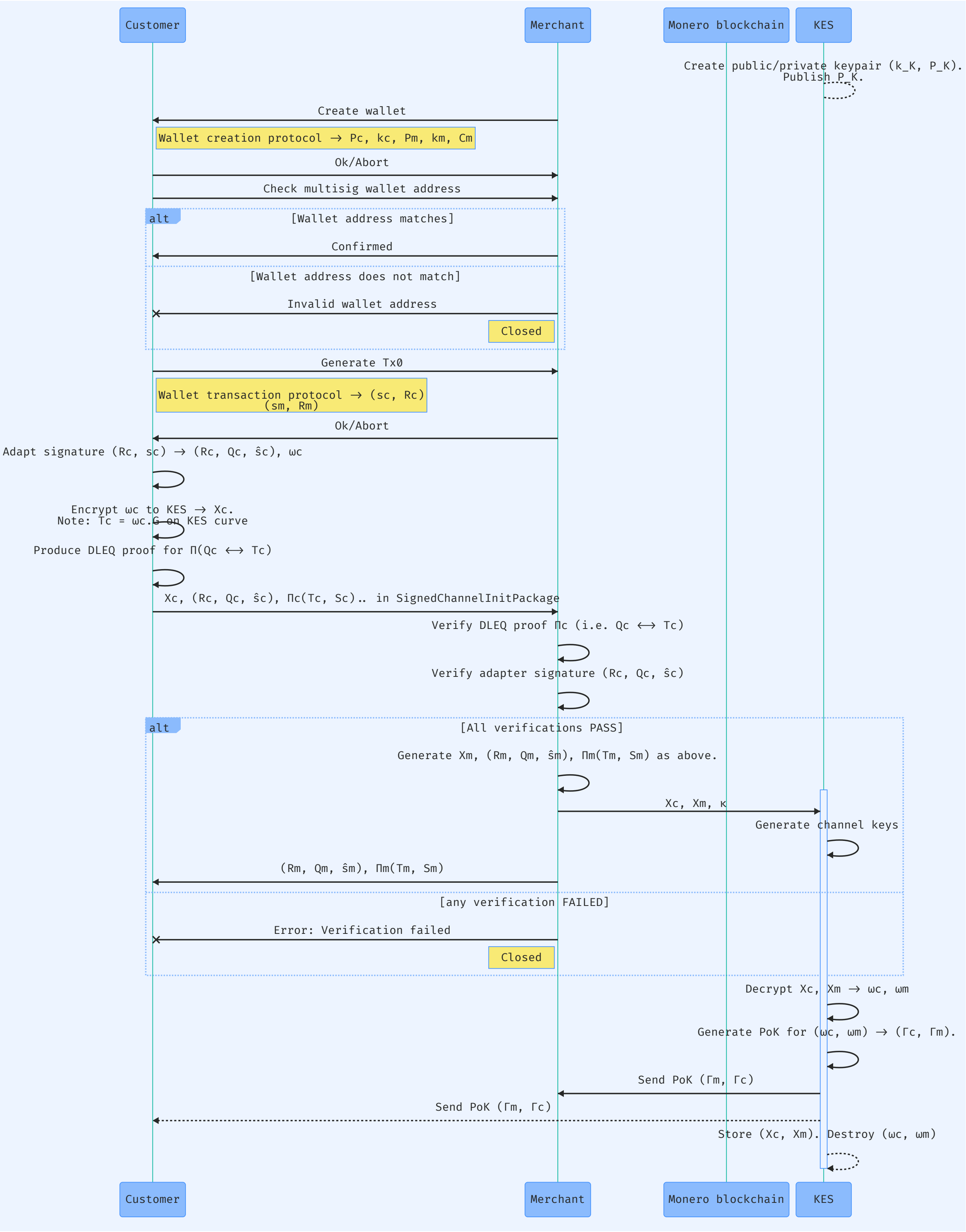
- The merchant public key, P_B, 32 bytes in little-endian byte order,
- The customer public key, P_A, 32 bytes in little-endian byte order,
- The merchant initial balance in piconero, as a 64-bit unsigned integer in little-endian byte order,
- The customer initial balance in piconero, as a 64-bit unsigned integer in little-endian byte order,
- The nominated closing address of the merchant, as a Base58 string,
- The nominated closing address of the customer, as a Base58 string,
- A merchant nonce, a 64-bit little-endian unsigned integer, randomly chosen by the merchant, and
- A customer nonce, a 64-bit little-endian unsigned integer, randomly chosen by the customer

3.4. Establishing the Channel

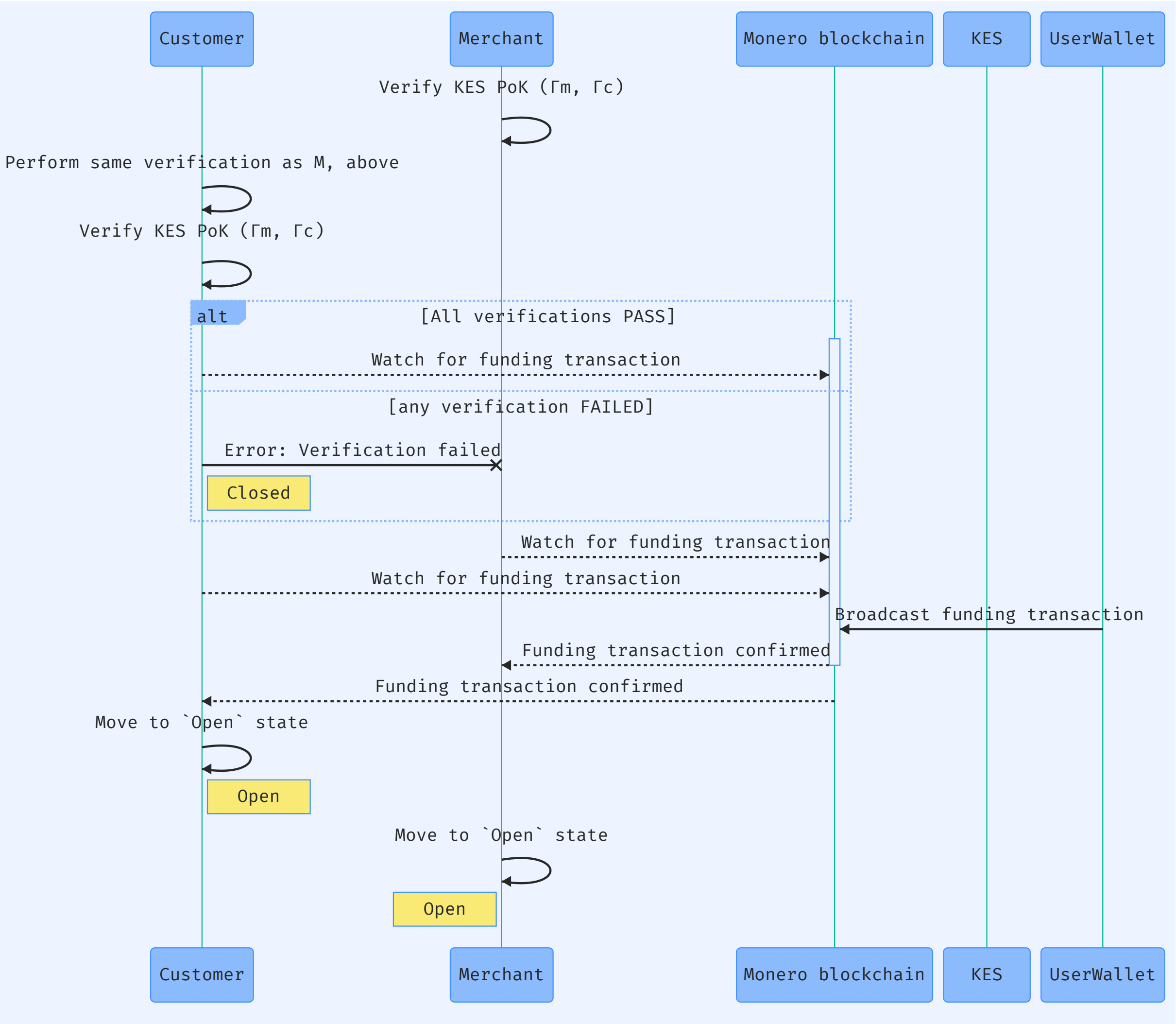
Establishing a channel to accept payments requires the following preparatory steps:

1. Both parties collaboratively create a new shared multisig wallet to hold the channel funds.
2. Each party calculates a unique, shared secret for the channel, κ (see Section 4.4).
3. Each party watches the Monero blockchain for the funding transaction to confirm it has been included in a block.
4. Each party encrypts their initial adapter signature offset to the KES.
5. The merchant creates a new KES commitment on the ZK-chain smart contract and commits the encrypted shares to it.
6. The merchant provides a proof of the KES commitment to the customer who can verify that the KES was set up correctly.
7. Each party creates the initial proof for their initial secret (witness) and shares it with the counterparty.
8. The customer funds the multisig wallet with the agreed initial balance.
9. Once the funding transaction is confirmed, the channel is open and ready to use.

> ⓘ **Note**
>
> No SNARKs are required for channel establishment. However, if the KES is deployed on a ZK-enabled chain, the KES proof of knowledge proofs should be calculated in a zero-knowledge manner.


Customer
Merchant
Monero blockchain
KES
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).. in SignedChannelInitPackage
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, κ
Generate channel keys
(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)
Customer
Merchant
Monero blockchain
KES

Listing 2: Establishing a new Channel


Customer
Merchant
Monero blockchain
KES
UserWallet
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

Listing 3: Establishing a new Channel, continued

3.5. Wallet transaction protocol

3.5.1. Overview

Grease uses 2-of-2 multisig transactions generated using FROST (Flexible Round-Optimized Schnorr Threshold signatures) combined with adapter signatures. The roles of customer and merchant are interchangeable, but for simplicity we will refer to the customer as the party initiating the protocol.

3.5.1.1. Phase 1: Customer Preprocessing

The Customer initiates the protocol by generating preprocessing data based on transaction details. This preprocessing step is fundamental to FROST and involves:

- Creating commitments for the signing process
- Generating nonce values that will be used during signature creation
- Producing data (denoted as C^{C}) that encapsulates these commitments.

The Customer then transmits both the preprocessing data and the transaction details to the Merchant.

3.5.1.2. Phase 2: Merchant Preprocessing and Partial Signing

Upon receiving the Customer's data, the Merchant carries out:

1. **Generates its own preprocessing data** (C^{M}) using the same transaction details
2. **Creates a partial signature** on the transaction using:
   - The transaction details
   - Its own preprocessing commitments
   - This produces a partial signature, $\left(R_0^{\mathrm{M}}, s_0^{\mathrm{M}}\right)$.
3. **Adapts the signature** by converting the partial signature into an adapter signature format:
   - Produces an adapted signature tuple $\left(R_0^{\mathrm{M}}, Q_0^{\mathrm{M}}, \hat{s}_0^{\mathrm{M}}\right)$
   - During channel initialization, it generates a random witness value (ω_0^{M}) that serves as a secret offset
   - During updates, this witness will be updated using the VCOF mechanism instead of being generated randomly.

The adapter signature is a cryptographic construct that allows the Merchant to create a valid-looking signature that is "locked" by a secret witness value. This signature appears complete but cannot be verified as a standard signature without knowledge of the witness.

The Merchant transmits its preprocessing data and the adapted signature back to the Customer, but notably **withholds** the witness value ω_0^{M}.

3.5.1.3. Phase 3: Customer Verification and Signing

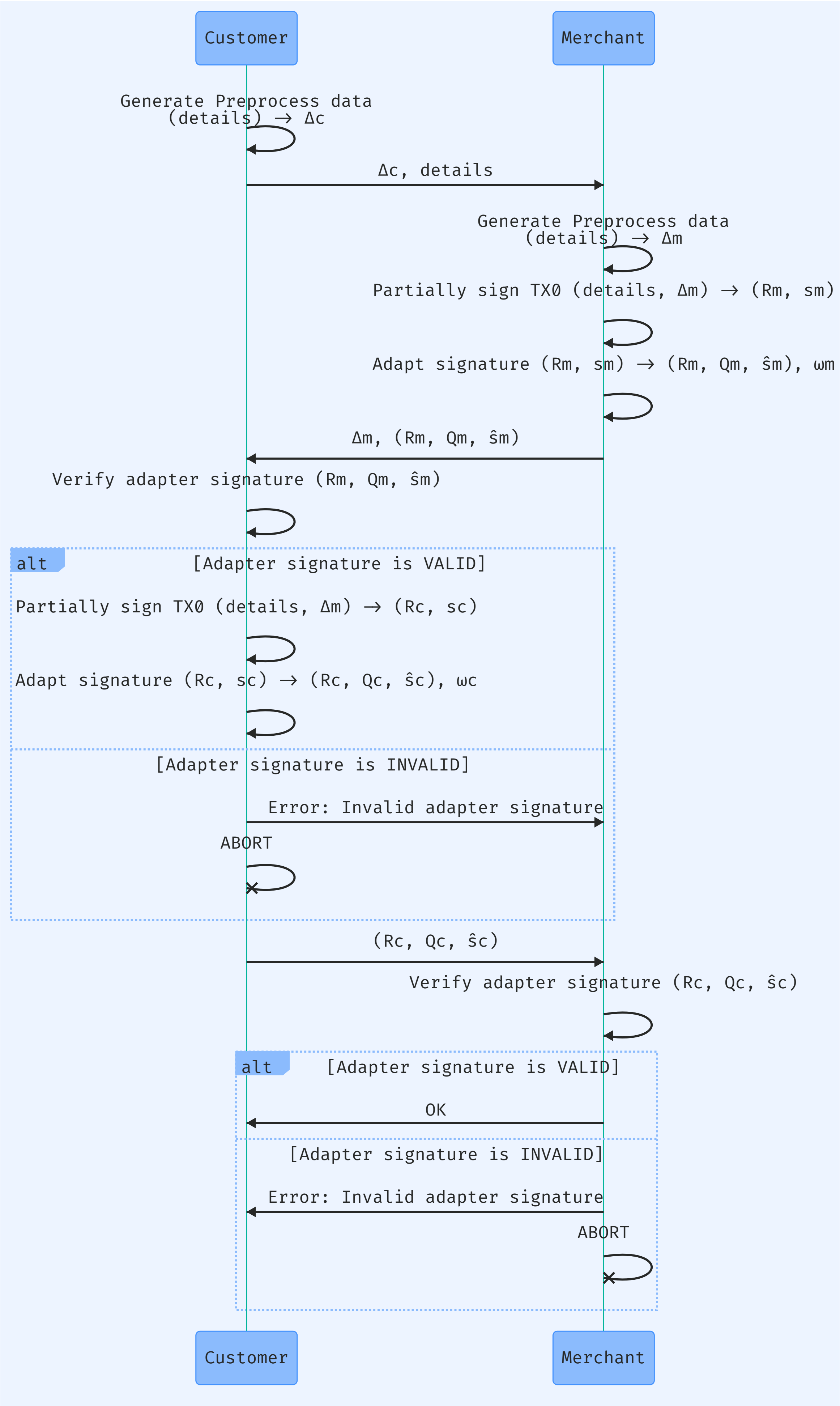
The Customer now performs verification and, if successful, creates its own adapted signature:

- **Verifies the adapter signature** $\left(R_0^{\mathrm{M}}, Q_0^{\mathrm{M}}, \hat{s}_0^{\mathrm{M}}\right)$ using the Merchant's public key
- If verification succeeds:
  - Creates its own partial signature $\left(R_0^{\mathrm{C}}, s_0^{\mathrm{C}}\right)$ on the transaction
  - Adapts this signature to produce $\left(R_0^{\mathrm{C}}, Q_0^{\mathrm{C}}, \hat{s}_0^{\mathrm{C}}\right)$ and witness ω_0^{C}
  - Transmits the adapted signature to the Merchant
- If verification fails:
  - Sends an error message to the Merchant
  - Aborts the protocol

3.5.1.4. Phase 4: Final Verification

The Merchant receives the Customer's adapted signature and performs the final verification:

- **Verifies the adapter signature** $\left(R_0^{\mathrm{C}}, Q_0^{\mathrm{C}}, \hat{s}_0^{\mathrm{C}}\right)$ using the Customer's public key
- If verification succeeds:
  - Sends confirmation to the Customer
  - Both parties now hold valid adapted signatures
- If verification fails:
  - Sends an error message to the Customer
  - Aborts the protocol


Customer
Merchant
Generate Preprocess data (details) → Δc
Δc, details
Generate Preprocess data (details) → Δm
Partially sign TX0 (details, Δm) → (Rm, sm)
Adapt signature (Rm, sm) → (Rm, Qm, ŝm), ωm
Δm, (Rm, Qm, ŝm)
Verify adapter signature (Rm, Qm, ŝm)
alt
[Adapter signature is VALID]
Partially sign TX0 (details, Δm) → (Rc, sc)
Adapt signature (Rc, sc) → (Rc, Qc, ŝc), ωc
[Adapter signature is INVALID]
Error: Invalid adapter signature
ABORT
(Rc, Qc, ŝc)
Verify adapter signature (Rc, Qc, ŝc)
alt
[Adapter signature is VALID]
OK
[Adapter signature is INVALID]
Error: Invalid adapter signature
ABORT
Customer
Merchant

Listing 4: Creating a new multisig wallet

3.6. Channel Updates

Once a channel is open the parties may transact and update the XMR balance between themselves. This is done entirely off-chain, and happens near-instantaneously.

> ⓘ **Note**
>
> Under CLSAG, unbroadcast Monero transactions become stale after some time because of how decoy selection is mandated and thus channels need to be kept “fresh” by providing periodic zero-delta updates. This is not an issue with FCMP++.
>
> 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.

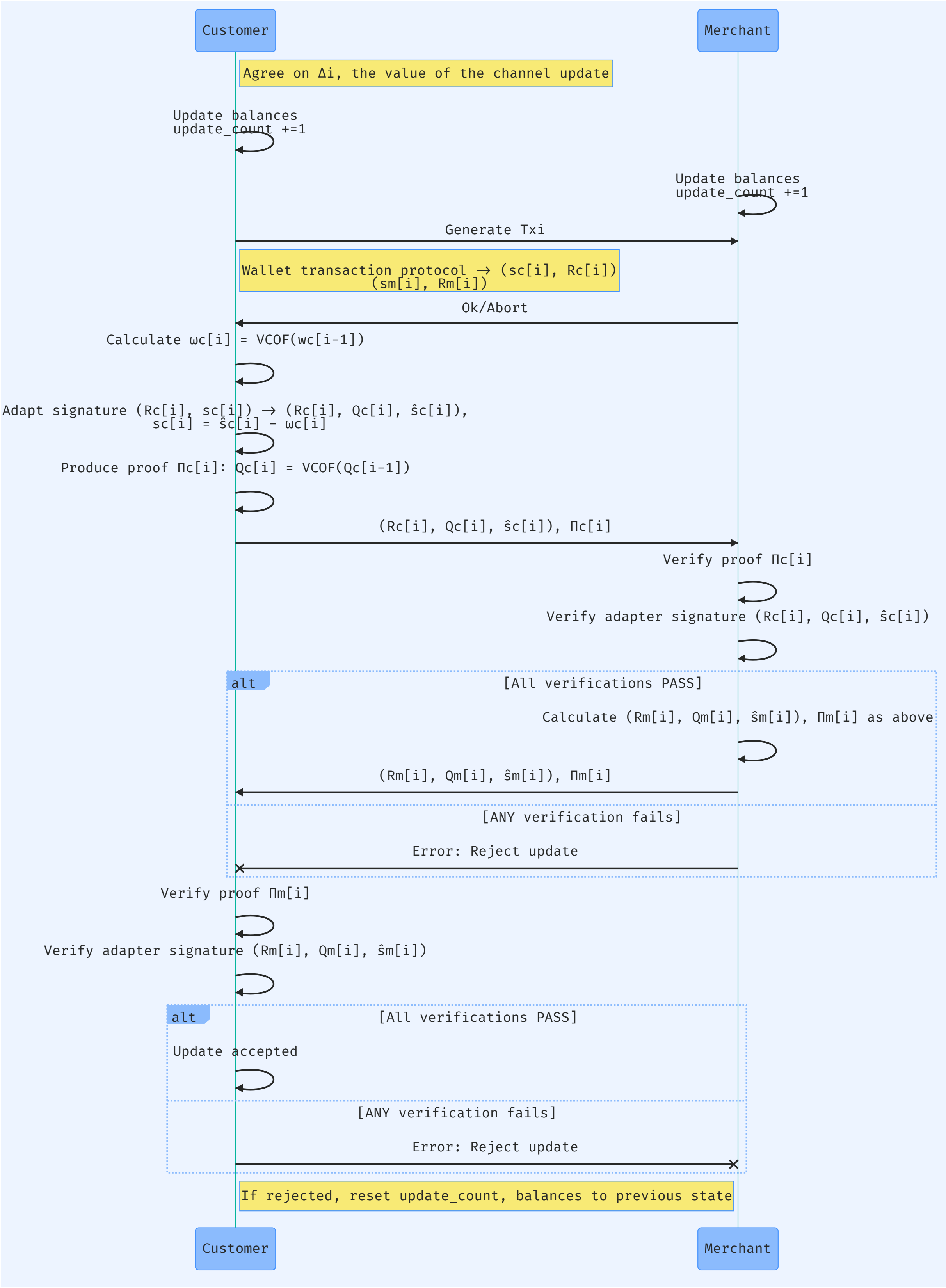
Either party can initiate a channel update. Since it is usually the merchant, we designate them as the initiator of the update, but the labels can be switched without loss of generality.

The merchant decides on the delta value, which can be positive (customer pays merchant) or negative (merchant refunds customer). He then carries out phase 1 on the transaction protocol to create a new Monero transaction reflecting the updated balances.

1. Privately, compute ω_i using Algorithm 1 from the previous ω_{i-1}.
2. Generate a DLEQ proof for T_i and Q_i.
3. Generate a ZK-SNARK proof Π_i^V using Algorithm 2.
4. Send the update package to the peer.

Verification consists of:

1. Verify the DLEQ proof for T_i and Q_i.
2. Verify the ZK-SNARK proof Π_i^V using Algorithm 3.


Customer
Merchant
Agree on Δi, the value of the channel update
Update balances
update_count +=1
Update balances
update_count +=1
Generate Txi
Wallet transaction protocol → (sc[i], Rc[i])
(sm[i], Rm[i])
Ok/Abort
Calculate ωc[i] = VCOF(ωc[i-1])
Adapt signature (Rc[i], sc[i]) → (Rc[i], Qc[i], ŝc[i]),
sc[i] = ŝc[i] - ωc[i]
Produce proof Πc[i]: Qc[i] = VCOF(Qc[i-1])
(Rc[i], Qc[i], ŝc[i]), Πc[i]
Verify proof Πc[i]
Verify adapter signature (Rc[i], Qc[i], ŝc[i])
alt
[All verifications PASS]
Calculate (Rm[i], Qm[i], ŝm[i]), Πm[i] as above
(Rm[i], Qm[i], ŝm[i]), Πm[i]
[ANY verification fails]
Error: Reject update
Verify proof Πm[i]
Verify adapter signature (Rm[i], Qm[i], ŝm[i])
alt
[All verifications PASS]
Update accepted
[ANY verification fails]
Error: Reject update
If rejected, reset update_count, balances to previous state
Customer
Merchant

Listing 5: Updating a Channel

ⓘ **Note**

> The actual implementation may streamline communication rounds by batching messages together. In particular, the VCOF messages and transaction protocol messages can be sent together to reduce the total number of round trips.

3.6.1. The Verifiable Consecutive One-way Function (VCOF)

The core idea behind AuxChannel is the Verifiable Consecutive One-way Function (VCOF). The VCOF is used to generate new adapter signature offsets for each channel update in that only the party who generated the new offset can compute it, but both parties can verify that it is valid and consecutive to the previous offset.

As such, the VCOF has the following security requirements:

- *validity*: It requires that for any message, the encrypted signature can be verified.
- *unforgeability*: It is hard to forge a valid encrypted signature.
- *recoverability*: Informally speaking, recoverability requires that it is easy to recover a decryption key by knowing the encrypted signature and its original signature.

These requirements are standard for any encrypted signature scheme. There are three additional requirements specific to the VCOF:

- *Consecutiveness*: requires that the decryption-encryption key-pair used in the i^{th} update of the VCOF is derived from the decryption key of the $(i-1)^{\text{th}}$ update.
- *Consecutive verifiability*: Given two VCOF outputs and the corresponding proof, anyone can be convinced that the outputs are consecutive, meaning that the new secret is generated from applying the VCOF to the previous secret.
- *One-wayness*: given ω_i, no one can derive any ω_j, where $0 \leq j < i$. This is important because, without one-wayness, when a channel is closed co-operatively, a malicious counterparty could broadcast *any* previous channel state to the blockchain by computing a previous state.

3.6.2. Grease VCOF

The Grease VCOF makes use of a ZK-SNARK to produce the following update algorithm (`KeyUpdate` using the AuxChannel parlance):

Informal security arguments for the Grease VCOF:

- One-wayness: If `H2F` is selected to be a suitable one-way hash function, then the Grease `KeyUpdate` function, Algorithm 1, is also one-way.
- Consecutiveness: If Algorithm 1 is used to generate ω_{i+1} from ω_i, then the proof generated by Algorithm 2 will verify under Algorithm 3. This is because the prover knows ω_i and can compute all the necessary values to satisfy the constraints in Algorithm 2.

```
VcofUpdate($i, \omega_i$)
$\omega_{i+1}$ = H2F($\omega_i$, $i$)
return $(\omega_{i+1}, i+1)$
```

Algorithm 1: Grease VCOF Update function

```
VcofProve($i, \omega_i, \omega_{i+1}, T_i, T_{i+1}$)
$\omega$ = H2F($\omega_i$, $i$)
assert $\omega == \omega_{i+1}$
$P_1 = \omega_{i+1} \cdot \mathbb{G}_2$
assert $T_{i+1} == P_1$
$P_2 = \omega_{i+1} \cdot \mathbb{G}_2$
assert $T_{i+1} == P_2$
return $\Pi^V_{i+1}$
```

Algorithm 2: The ZK-SNARK Grease `VCOFProve` function

```
VcofVerify($i, T_i, T_{i-1}, \Pi^V_i$)
verify ZK-SNARK proof $\Pi^V_i, T_i, T_{i-1}$
verify DLEQ proof for $T_i, Q_i$
```

Algorithm 3: Grease VCOF Verify function

- Consecutive verifiability: The ZK-SNARK proof produced by Algorithm 2 convinces any verifier running Algorithm 3 that ω_{i+1} was correctly derived from ω_i using Algorithm 1 without revealing either secret value.

3.7. Co-operative Close

Either party can initiate a co-operative close of the channel at any time by sending their latest adapter offset to the counterparty in a `RequestChannelClose` message. As an illustration, we will assume that the customer initiates the close.

When the merchant receives the offset, it verifies it, by trying to complete the closing transaction signature for the expected update count. It then broadcasts the transaction to the network. The channel is then marked as `Closed`. If the closure was successful, the merchant responds with an `ChannelCloseSuccess` message. If any errors occur, the merchant responds with a `RequestCloseFailed` message, and the channel remains `Open`. If the merchant did not provide the transaction id in its response, the customer may use the provided offset to reconstruct and broadcast the closing transaction herself.

If a party becomes unresponsive during the co-operative close process, one may initiate a force-close via the KES, as described in Section 4.

3.7.1. Channel Close messages

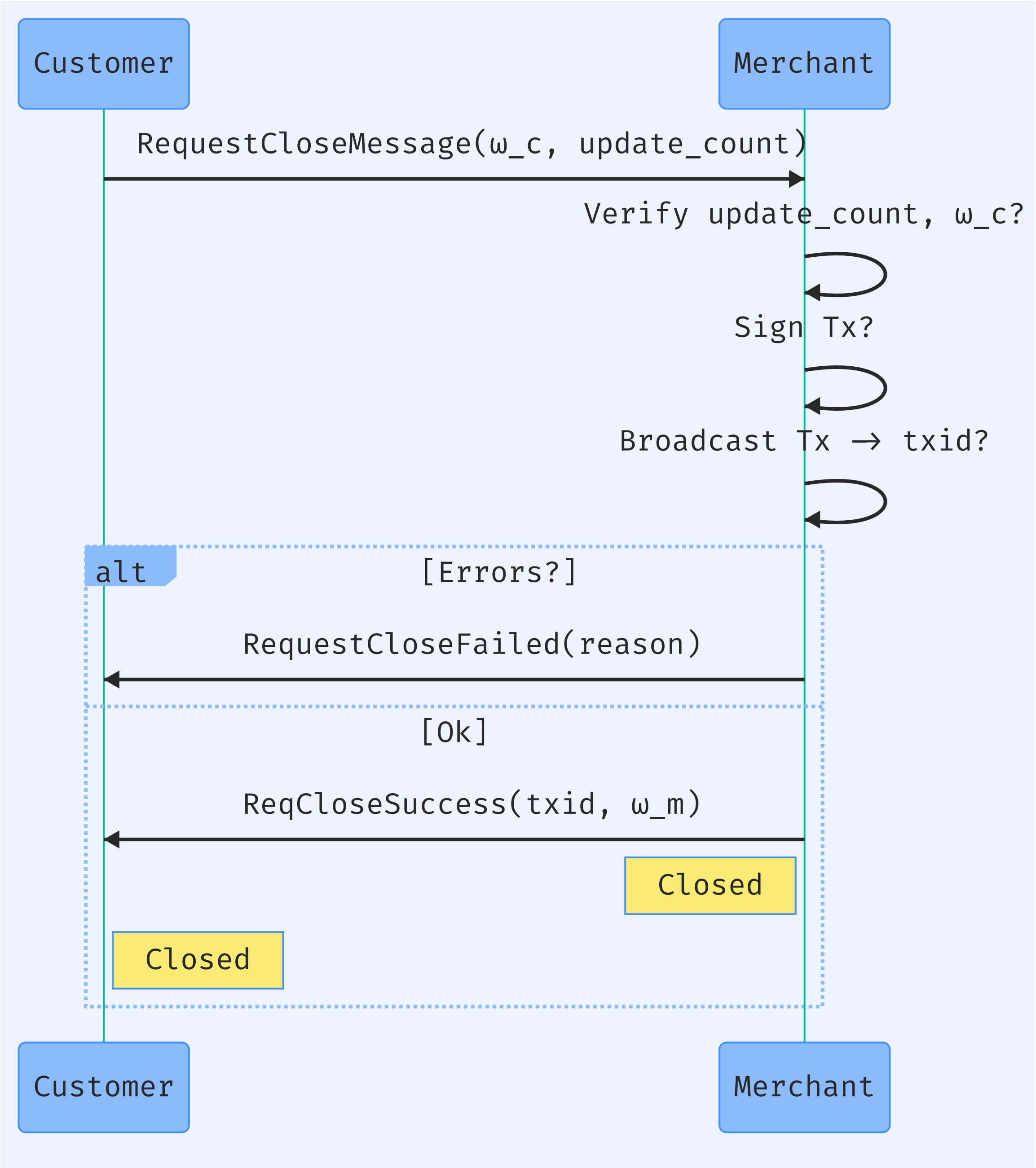
A party sends a `RequestChannelClose` message to initiate a co-operative channel closure.

```
pub struct RequestChannelClose {
  /// The globally unique channel id
  channel_id: ChannelId,
  /// The latest adapter offset for the initiating party
  offset: Scalar,
  /// The update count corresponding to the latest adapter offset
  update_count: u64,
}
```

The counterparty responds with either a `RequestCloseFailed` or `ChannelCloseSuccess` message.

```
pub struct ChannelCloseSuccess {
  /// The globally unique channel id
  channel_id: ChannelId,
  /// The latest adapter offset for the responding party
  offset: Scalar,
  /// The transaction id of the closing transaction. Optional.
  txid: Option<TxId>,
}
```

```
pub struct RequestCloseFailed {
  /// The globally unique channel id
  channel_id: ChannelId,
  /// An error code indicating the reason for the failure
  reason: CloseFailureReason,
}
```


Customer
Merchant
RequestCloseMessage(ω_c, update_count)
Verify update_count, ω_c?
Sign Tx?
Broadcast Tx → txid?
alt
[Errors?]
RequestCloseFailed(reason)
[Ok]
ReqCloseSuccess(txid, ω_m)
Closed
Closed
Customer
Merchant

Listing 6: The Channel Close sequence

3.8. 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.

The dispute process is described in detail in Section 4.8.

4. The Key Escrow Service (KES) Design

4.1. 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 4). If the unlock 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 1 summarizes the KES' role throughout 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.

Table 1: The role of the KES during the channel lifecycle

4.2. 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 $N_1.G \equiv \mathcal{O}$.

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

$H_F(\ldots)$ 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, $H_P(\ldots)$ 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 4.8) 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 $(R, \chi)$.
```

Algorithm 4: 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 5: 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.

4.3. 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.

4.4. 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 6.

1. The merchant (or customer, it doesn't matter) then encrypts κ to the KES using `EncryptMessage` (see Algorithm 4) on $\mathbb{E}_K$.
2. The KES calculates the unique channel keypair for $\mathbb{E}_K$:

$$\begin{aligned} k_g &= \kappa \cdot k_K \\ P_g &= k_g \cdot G \end{aligned} \tag{1}$$

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

```
ECDH-MC
$k_s = \hat{k}_a \hat{P}_b \left(\equiv \hat{k}_b \hat{P}_a\right)$ on $\mathbb{E}_K$
$\kappa \leftarrow H_F(k_s, \text{channel\_id})$ on $\mathbb{E}_K$
$P_\kappa \leftarrow \kappa \cdot G$ on $\mathbb{E}_K$
```

Algorithm 6: 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 they can). One could use a **master key**, k_a and derive unique channel keys using a simple deterministic scheme,

$$\hat{k}_a = H_F(\text{k\_a}, \text{channel\_id}) \tag{2}$$

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.

4.4.1. 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.

4.5. 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.

4.6. 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 3.4.

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

$$0 < \omega_0 < \min(N_1, N_2) \qquad (3)$$

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 2.

> ⓘ **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.

4.6.1. 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 ω_{0_C} to the KES using `EncryptMessage` (Algorithm 4) 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 to the Customer, who validates it.

4.6.2. What the KES send back to parties

Once the KES has received both packages, it performs the validation procedure outlines in Algorithm 7

Open Channel validation

```
Validate the payload signatures.
Decrypt $\chi_C \to \omega_0^C$ and $\chi_M \to \omega_0^M$.
Are dispute windows, dw, values equal?
    No? Return Fail(InvalidConfiguration)
Store an OpenChannel record in private/encrypted storage.
Calculate proof-of-knowledge proofs, $\Gamma_C$ and $\Gamma_M$, for $\omega_0^C$ and $\omega_0^M$ respectively, using the per-channel key $k_g$ (not the global key $k_K$).
Send the PoK proofs to the Merchant and/or Customer.
Discard $\omega_0^C$ and $\omega_0^M$.
```

Algorithm 7: The KES validates each new channel request, returning proofs of knowledge if successful.

4.6.3. `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: 86,400)
- 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.
>
> Note that to verify the PoKs, parties only need the KES global public key P_K, since the per-channel public key is simply $P_g = \kappa \cdot P_K$.

4.6.4. 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.

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

Table 2: Equivalent curve points in grease

4.7. Channel Force Close

4.7.1. 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 $\left(k_{d1}^A, ..., k_{di}^A\right)$ 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.

4.8. Force-closing in Grease

4.8.1. Summary

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, `wn` $_n$ 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 is given in the following sections.

```
onForceCloseRequestEvent(req: ForceCloseRequest):

Validates the signature in req.
if valid:
    Fetch the OpenChannel (Section 4.6.3) record with id == req.id => channel
    (Only allow channel parties to initiate a force-close)
    if exists(channel) && req.Pc in [channel.Pa, channel.Pb]:
        Create a new PendingChannelClose entry with status = Pending.
        try to send a copy of the PendingChannelClose to the counterparty.
        return Ok( ForceCloseResponse )
    else
        return Fail( FailReason )
else
    return Fail( InvalidSignature )
```

Algorithm 8: The KES `HandleForceCloseRequest` algorithm.

4.8.2. Force-close procedure

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 8.

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 9.

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

4.8.3. `ForceCloseRequest` message

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

```rust
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,
}
```

```
OnClaimChannelEvent(req: ClaimChannelRequest):

Validates the signature in req.
if valid:
    Fetch PendingChannelClose where id == req.id => pending
    if !exists(pending) return Fail(No such channel)
    if req.claimant ≠ pending.req.claimant return Fail(Unauthorized)
    if current time $\geq$ pending.expiry
        pending.req.defendant $\Rightarrow P_d$
        pending.req.claimant $\Rightarrow P_c$
        using Algorithm 5, decrypt $\chi$ for pending.offsets where public key is $P_c$ and $\Rightarrow \omega_0$.
        Case claim option:
            Option I: $\omega_0 \Rightarrow$ wn
            Option II: using Section 3.6.1 for n = pending.req.update_count_claimed, calculate
            wn$_n$ $\Rightarrow$ wn
        encrypt wn for $P_c \Rightarrow \chi_c$ using Algorithm 4
        update pending, set status to ForceClosed, add (Pc, χc, n) to chi
        try send $\chi_c$ to claimant
    else return Fail("Dispute window still open")
else return Fail(InvalidSignature)
```

Algorithm 9: The KES `HandleClaimChannelRequest` algorithm.

4.8.4. `PendingChannelClose` message

The KES must track all channel that are under dispute.

```rust
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)>
}
```

```rust
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,
}
```

4.8.5. `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 9.

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

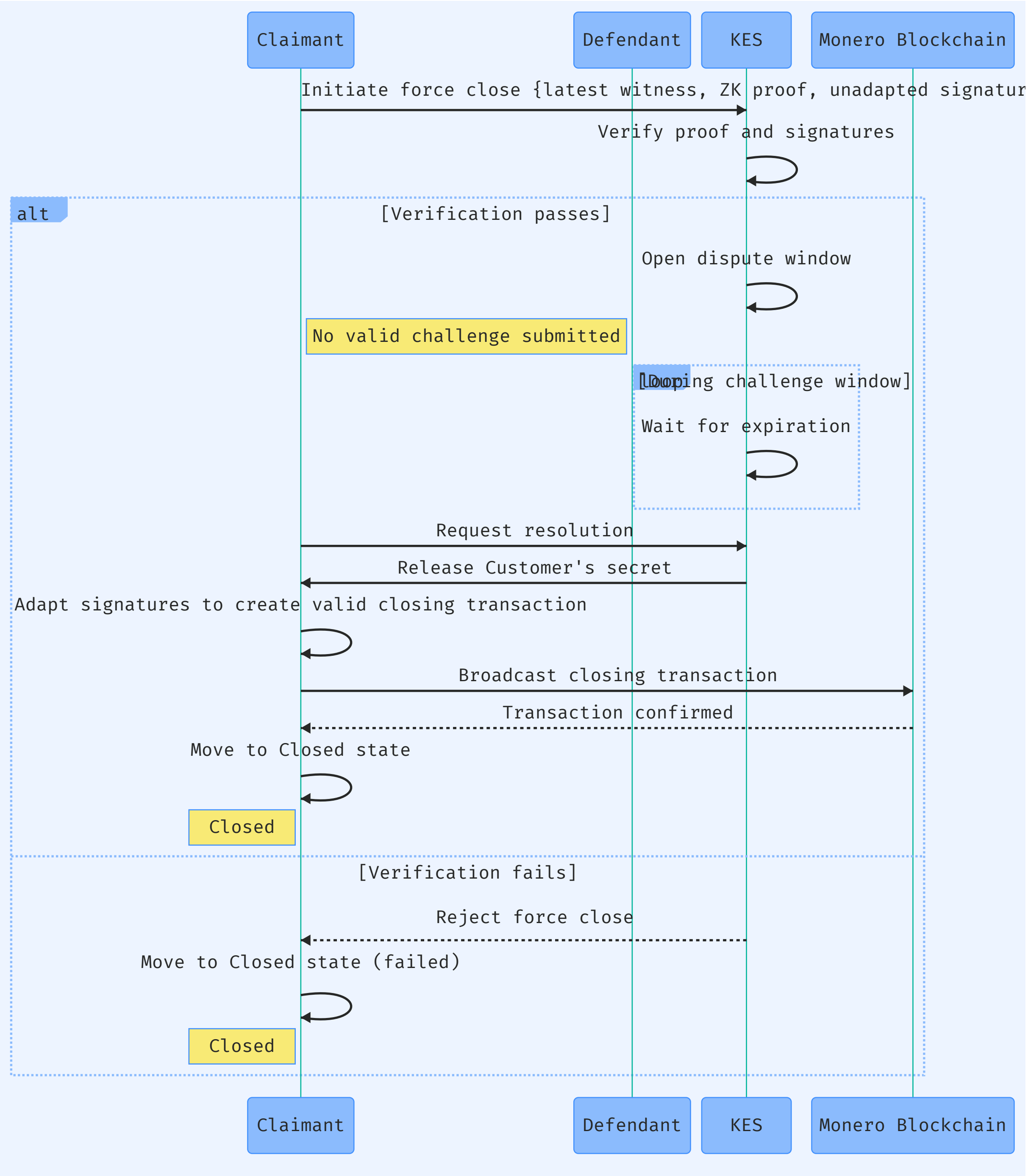
- Either claimant **or** defendant may sign a `ClaimAbandonedChannelRequest` message.
- The KES will only allow the claim if if `current_time ⩾ pending.expiry + req.dw`, i.e. typically an additional 24 hours after the dispute window closes.
- After a successful claim, `pending.status` is set to `AbandonedClaimed`.

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 4.10), the claimant can claim the channel using either type of message, but it is recommended to use `ClaimChannelRequest`.


Claimant
Defendant
KES
Monero Blockchain
Initiate force close {latest witness, ZK proof, unadapted signatur
Verify proof and signatures
alt
[Verification passes]
Open dispute 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
Claimant
Defendant
KES
Monero Blockchain

Listing 7: The channel dispute sequence

4.9. 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 co-operatively,
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.

4.9.1. 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 4.9.1.1) 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 3.7), 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 10.

4.9.1.1. `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,
}
```

```
handleConsensusCloseRequest(req: ConsensusCloseRequest)

if !req.signature valid, return Fail( Unauthorized )
if exists(req.channel_id) in PendingChannelClose records as p :
    if p.defendant $\neq$ req.defendant or p.claimant $\neq$ req.claimant , return Fail( Invalid )
    if p.update_count_claimed $\neq$ req.update_count_claimed , return Fail( Invalid )
    Decrypt req.encrypted_offset $\rightarrow$ wn $_n$ (Algorithm 5), else return Fail( Invalid )
    Encrypt wn $_n$ $\rightarrow \chi_n^D$ under $P_D$ (Algorithm 4)
    Update p : set p.chi_D = $\chi_n^D$, p.status = ConsensusClosed
    Try send $\chi_n^D$ to defendant.
else return Fail( NotFound )
```

Algorithm 10: Resolving a channel force-close via consensus

4.9.2. 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.

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

```rust
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 checks pass, the KES decrypts ω_0^B with Algorithm 5, encrypts it to P_A using Algorithm 4 and sends the encrypted ω_0^B to Alice.

4.9.3. 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 9.

4.10. Channel cleanup

4.10.1. 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.

4.10.2. 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.

$$t_D = \texttt{expiry} + 2\,\texttt{dw} + \texttt{one\_month} \tag{4}$$

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`

4.11. 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".

5. 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.

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).

Table of Algorithms

6. Nomenclature

6.1. Symbols

Symbol	Description
$\mathbb{G}_2$	Generator point for curve Baby JubJub
G_1	Generator point for curve Ed25519
L_2	The prime order of curve Baby JubJub
L_1	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

6.2. Subscripts

Subscript	Referent
BJJ	Curve Baby JubJub
Ed	Curve Ed25519
M	The peer playing the role of merchant
C	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

Index of Listings