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 that they can only send a single send every 20 minutes (since their wallets contain a single UTXO),
- 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

Grease is a proof-of-concept Monero payment channel that uses a ZK-rollup chain for off-chain state management.

It aims to tackle the use cases that are exemplified by the following scenarios:

1.1.1.1. Rapid point-of-sale

Alice is a customer of Bob's bar. Alice will be making multiple purchases throughout an evening. She opens a channel at the beginning of the evening with a certain amount of Monero, and can make instant purchases against it until she and Bob mutually close the channel at the end of the evening and the final settlement is recorded on the Monero chain.

1.1.1.2. Micro-transactions

Bob owns a Monero-enabled arcade. Dave can open a channel and play dozens, or hundreds of games until his balance runs out. Each payment is instant and secure, does not bloat the Monero blockchain, and is completely private. Some games might offer rebate prizes which can be pushed straight back into the channel.

1.1.1.3. Private and anonymous content consumption

Erica's online newspaper utilizes a pay-per-view model. Instead of a monthly subscription fee, users open a channel with their maximum "reading budget" and instantly and seamlessly pay for each article they read. No accounts, no KYC and no email addresses are required. At the end of the month, users can close the channel to settle their bills, or default opt to continue their balance to the next month without closing the channel (and hence performing an on-chain swap with the associated XMR fees). That is to say that if Fred has read 100 articles at 0.0005 XMR each, and has sent 0.05 XMR down the channel, he can pay 0.05 XMR on-chain, and Erica pushes that amount back up the channel, effectively resetting the state for the new month.

This provides the ability to have a combination of the use-or-lose minimum fee plus À la carte options which is standard in legacy subscription models.

1.1.2. Why does another chain have to be involved?

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

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

- available,
- reliable and verifiable,
- trustless,

will suffice.

The AuxChannel[4] and Monet[5] papers, summarized in Sui's PhD thesis[6] provide a sketch of how this might be achieved, using Ethereum as the state management chain. However, by using Ethereum, the channel metadata, including the peer's public keys and the channel state (open, disputed) is scrutable by the public.

Grease aims to improve on this by making the payment channel metadata private as well. Zero knowledge proofs provide a way to do this.

1.2. Design principles

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

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

- The public peer is responsible for recording the channel state on the ZK chain.
- The public peer pays for gas fees on the ZK chain and will need to have some amount of ZK chain tokens to pay for these fees.
- The public peer will be able to recover gas fees from the client peer.
- The client peer does not *have* to have any ZK chain tokens, but will need to hold Monero to open the channel.
- The client peer will need ZK chain tokens if they want to dispute a channel closure. In the vast majority of cases, this won't be necessary, since funds almost always flow in one direction from the client to the merchant. However, in instances where this is not the case, the client is able to

dispute the channel closure by watching the ZK chain and proving that the channel was closed with outdated state.
- In the vast majority of cases, the client opens a channel with *m* XMR and the public peer starts with a zero XMR balance (since the public peer is providing assets or services and not monetary value).
- Usually, both parties mutually close the channel. Either party *may* force close the channel, and are able to claim their funds after a predetermined time-out. In this case, the forcing party is usually the merchant since they have the greater incentive to do so in the case where a channel has been abandoned by the client.

1.2.1. 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° that the vast majority of the utility of lightning is captured by bilateral channels, with a tiny fraction of the complexity.

2. 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 satisfices the security trilemma (Security, Decentralization, Scalability) by maintaining complete security and improving scalability at a minimal cost of network centralization. Since no identifiable information about the peers' privately owned Monero wallets are shared between the peers there is no means by which privacy can be compromised. And since only the one initialization transaction and one closure transaction impact the Monero blockchain, scalability is improved by the unlimited number of updates to the channel in between these two transactions.

Software implementations of the Grease protocol could create new markets opportunities for interactions that require a proof of potential payment or known and locked value for an arbitrary amount of time. Simple examples are down payments, subscriptions, creating a tab at a restaurant or reserving access to a future event.

With the upcoming FCMP++ update Monero will lose the ability to provably lock a known amount of XMR for a fixed length of time. This was used to create a simple proof of known and locked value outside of trusted third parties. The Grease protocol automatically includes this proof of known and locked value since the channel balance is exactly this proof, with the benefit that the value can be locked for any length of time and any value is easily transferable or refundable.

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

Every update and the final closure of the channel require an online interaction over the Grease network. In order to prevent the accidental or intentional violation of the protocol by a peer not interacting and thus jamming the channel closure, Grease protocol requires the use of an external Key

Escrow Service (KES). While a software service and not tied to any particular technology, we can still consider the KES to be an L2 since it will likely run on a ZKP-compatible smart contract blockchain. The example implementation uses the Aztec blockchain and its support of the Barretenberg Plonky proving system, but there are many other blockchains and proving systems that are fully compatible. Most blockchains have limitations on security due to their design, and if the funding for the KES requires publicly viewable transactions then the peer that funds the KES may confront security loss while the other peer does not and can be completely anonymous.

The KES will act as third-party judge of disputes. At initialization, each peer provably encrypts a 2-of-2 secret shared to both the other peer and to the KES. Any one share does not have enough information to interfere with the channel's operations or violate security. In the case of a dispute the KES will decide which peer is in violation and then release its share of the violating peer's secret to the wronged peer. The wronged peer can then reconstruct the original secret. This secret will allow the wronged peer to simulate the missing online interaction of the violating peer, allowing the channel to close with the existing balance. Only willfully valid channel balances can be closed as there is no way to simulate false updates.

3. Grease payment channel lifetime

A quick walk-through of a Grease channel lifetime:

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 **vout** 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 **vin** 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 secret shares,
14. and finally broadcast the funding transaction to Monero.

At **update**, 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 **vout** 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.

At **closure**, 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.

In case of a **dispute** a plaintiff will:
1. provide the unadapted signatures of the closing transaction to the KES.

If a dispute is detected the other peer will:
1. respond with the adapted signature of the closing transaction to the KES,
2. or simply broadcast the closing transaction to Monero.

If a dispute is lodged and the other peer does not respond:
1. the KES will provide the saved root secret share of the violating peer to the wronged peer,
2. the wronged peer will reconstruct the violating peer's root secret,
3. the wronged peer will deterministically update the the secret until finding the most recent secret,
4. the wronged peer will adapt the unadapted signature of the closing transaction using the most recent secret to gain the 4-of-4 pieces of information needed to broadcast the transaction,
5. and finally the wronged will broadcast the closing transaction to Monero.

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.

4. Future extensions

There are a number of possible extensions that Grease can support but are not implemented in this example implementation.

- The MoNet protocol allows for multi-hop payment for multiplying the scalability of the the Monero L1. We do not implement this but the MoNet design can be leveraged quickly and easily.
- The KES is assumed to be a single web service with one public key and is susceptible to Sybil attacks. We can add security with the use of TEEs, multiple servers using MPCs (such as co-SNARKs), or creating a more complex shared secret splitting mechanism among a larger number of KES nodes.
- 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).

... TODO

5. 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.

5.1. Part 1: Protocol Stages

5.1.1. Initialization

5.1.1.1. 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.

5.1.1.2. Preliminary

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

Before Initialization	
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.

5.1.1.3. MoNet: Before Initialization

The **MoNet** protocol specifies that the peers must agree upon the following using the defined classical interactive protocols with verification:

Before Initialization	
Resource	
Monero Funding Wallet	Each peer must have a source wallet with its private spend key. This wallet will need to have at least the **Locked Amount** available.
Monero Refund Wallet	Each peer must have a destination wallet

5.1.1.4. Grease: Before Initialization

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 on the L2, knows as the Key Escrow Service (KES). When the peers agree on the particular KES, the publicly known public key to this service is shared as Π_{KES}.

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

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

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

5.1.1.4.1. MoNet: During Initialization

The **MoNet** protocol is very specific on its stages and operations. The Grease protocol maintains the main stages of the MoNet protocol in general but replaces the *2P-CLRAS.JGen()*, *2P-CLRAS.SWGen()*, *2P-CLRAS.NewSW()*, *2P-CLRAS.CVrfy()* and *2P-CLRAS.PSign()* functions. These functions were base on innovative and non-standard cryptographic methods that have not gained the general acceptance of the cryptographic community. This may change in time, while the MoNet protocol bypasses this by the use of generally accepted methods.

Note that the original MoNet protocol was not completely compatible with the existing Monero protocol due to the lack of Transaction Chains features. Note that this compatibility flaw will change with the FCMP++ update.

The 10 stages of the MoNet protocol for initialization are:

MoNet		
Stage	**Name**	
1	*Call 2P-CLRAS.JGen* *and Obtain* $(\text{vk}_{\text{AB}}, \tilde{\text{sk}}_X)$	This shared DKG temporary Monero wallet function is replaced entirely during MoNet. ... TODO The peers will use the well-established Monero wallet implementation of the DKG MPC to generate a 2-of-2 Monero wallet. Once complete, both peers will have the wallet view key vk_{AB} and each peer will have 1 of the 2 private spend keys $\tilde{k}_i$ / $\tilde{\text{sk}}_X$. This wallet will require both σ_{vk_A} and σ_{vk_B} signatures to complete any transaction. Since the only outgoing transaction on this wallet is the T_{x_c} commitment transaction, there will only be the 2 signatures, but these signatures will change during every channel update.
2	*Call 2P-CLRAS.SWGen()* *Obtain* $(S^0, (S^0_X, \omega^0_X), P^0_X)$	This statement/witness generation function is replaced entirely by the Grease operations during initialization. Note that this is the stage at which all of the Grease operations for proving and verification will occur.
3.0	*Generate* T_{x_f}	The peers will collaborate to create the funding transaction T_{x_f} that will require each peer to each create 1 or 2 **vin** inputs and 1 or 2 **vout**

		outputs, so T_{x_f} will have 2 to 4 **vin** inputs and 2 to 4 **vout** outputs. The **vout** data will include their Pedersen commitments (*out_pk*). Since the peers will share the new wallet's view key vk_{AB} the peers can verify that 2 of the **vout** outputs will target the encrypted *one-time address* of the temporary Monero wallet.
3.5	*Generate* $T_{x_c^0}$	The peers will also collaborate to create the commitment transaction $T_{x_c^0}$ so that the 2 **vin** inputs for $T_{x_c^0}$ will include the T_{x_f} transacstion's 2 **vout** outputs, and the 2 **vout** outputs for $T_{x_c^0}$ will each target the encrypted *one-time address* of each peer's **Monero Refund Wallet** destination. Note that before the FCMP++ update there are no transaction chains features available, and the translation of the 2 **vout** outputs in T_{x_f} to the 2 **vin** inputs in $T_{x_c^0}$ requires that the funding transaction T_{x_f} is mined on the Monero blockchain so that the **vout index numbers** will exist. This is possible only at step 10. This limitation can be bypassed by creating an unusable $T_{x_c^0}$ with invalid **vout index numbers** that may be verified by the peers but cannot be broadcast to the Monero network.

4	*Call 2P-CLRAS.PSign()* *and Obtain* $\hat{\sigma}^0_{\tilde{\mathrm{sk}}_A,\tilde{\mathrm{sk}}_B}$	The peers collaborate to create the unadapted signature $\hat{\sigma}^0_{\tilde{\mathrm{sk}}_A,\tilde{\mathrm{sk}}_B}$ for $T_{x_c^0}$ by using $\tilde{\mathrm{sk}}_X$ and verifying with S^0_X.
5, 6, 7, 8	*Call* $\mathrm{LRS.Sign}_{\mathrm{sk}_X}\left(T_{x_f}\right)$ *and Obtain* σ_{vk_X}	The peers will collaborate atomically to complete the signature of T_{x_f} one step at a time. First the peers will calculate the amount commitments (*ecdh_info*) and Bulletproofs. Then they will use their **Monero Funding Wallet** private spend keys to create the funding signatures σ_{vk_X}.
9	*Broadcast signed* T_{x_f} *to Monero*	With the preliminaries complete, one of the peers broadcasts the complete T_{x_f} to the Monero network. Both peers verify that T_{x_f} is pending mining.
10	*Channel Established*	Once T_{x_f} is mined on the Monero blockchain, the channel is established. Note that before the FCMP++ update the peers can decide on whether to recompute step 3.5 to ensure that $T_{x_c^0}$ is usable.

5.1.1.5. Grease: During Initialization

As a substitute for **MoNet** protocol stage 2, 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.

The ZKP operations require random values to ensure security of communications. These are not shared with the peer:

5.1.1.5.1. Inputs:

During Initialization		
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)

The ZKP operations produce output values, 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 usage:

5.1.1.5.2. Outputs:

During Initialization		
Output	**Visibility**	
T_0	Public	The public key/curve point on Baby Jubjub for ω_0
ω_0	Private	The root private key protecting access to the user's locked value (witness_0)
c_1	Public	Feldman commitment 1 (used in tandem with Feldman commitment 0 $= T_0$), which is a public key/curve point on Baby Jubjub
σ_1	Private	The split of ω_0 shared with the peer (share_1)
Φ_1	Public	The ephemeral public key/curve point on Baby Jubjub for message transportation to the peer (fi_1)
χ_1	Public	The encrypted value of σ_1 (enc_1)
σ_2	Private	The split of ω_0 shared with the KES (share_2)
Φ_2	Public	The ephemeral public key/curve point on Baby Jubjub for message transportation to the KES (fi_2)
χ_2	Public	The encrypted value of σ_2 (enc_2)
S_0	Public	The public key/curve point on Ed25519 for ω_0

C	Public	The Fiat–Shamir heuristic challenge (challenge_bytes)
$\Delta_{\text{BabyJubjub}}$	Private	Optimization parameter (response_div_BabyJubjub)
$\rho_{\text{BabyJubjub}}$	Public	The Fiat–Shamir heuristic challenge response on the Baby Jubjub curve (response_BabyJubJub)
Δ_{Ed25519}	Private	Optimization parameter (response_div_BabyJubJub)
ρ_{Ed25519}	Public	The Fiat–Shamir heuristic challenge response on the Ed25519 curve (response_div_ed25519)

During the initialization stage, the following operations are performed:

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

Particular details about these operations can be found in **Part 2: Grease ZKP Operations**.

5.1.1.6. Grease: After 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 following resources and information are available and must be stored:

After Initialization	
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

5.1.1.7. MoNet: After Initialization

When complete the following resources and information are available and must be stored:

After Initialization	
Resource	
Monero Temporary Wallet	The temporary 2-of-2 multisignature Monero wallet with the locked XMR value of the channel
$\mathrm{vk}_{\mathrm{AB}}$	The shared **Monero Temporary Wallet** view key
$\tilde{\mathrm{sk}}_X$	The **Monero Temporary Wallet** spend key
Channel Balance	The two values in XMR (with either but not both allowed as zero) for the peers, equal to **Locked Amount**
$T_{x_c^0}$	The commitment transaction that closes the channel, with the **vout** XMR values equal to **Channel Balance**

With these outputs the 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**.

5.1.2. Update

5.1.2.1. 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. The FCMP++ update does not change the need for this, but does alter the timeframe. As such, existing open channel should create a "zero delta" update at reasonable timeframes to ensure the channel may be closed arbitrarily. The specifics on this are outside of current scope.

5.1.2.2. Preliminary

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

Before Update	
Resource	
Delta	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.

5.1.2.2.1. MoNet: During Update

The **MoNet** protocol is very specific on its stages and operations. The Grease protocol maintains the stages of the MoNet protocol in general but replaces all of the *2P-CLRAS.NewSW()*, *2P-CLRAS.CVrfy()* and *2P-CLRAS.PSign()* functions. These functions were base on innovative and non-standard cryptographic methods that have not gained the general acceptance of the cryptographic community. This may change in time, while the MoNet protocol bypasses this by the use of generally accepted methods.

The 3 stages of the MoNet protocol for update are:

MoNet		
Stage	**Name**	
11	*Call 2P-CLRAS.NewSW()* *Obtain* $(S^i, (S^i_X, \omega^i_X), P^i_X)$	This statement/witness generation function is replaced entirely by the Grease **Verify-COF** operation, and the verification function is replaced entirely by the Grease **Verify-DLEQ** operation. Note that this is the stage at which all of the Grease operations for proving and verification will occur.
12	*If* *2P-CLRAS.CVrfy* $((S^{i-1}_X, S^i_X), P^i_X)) = 0 : \perp$	
13	*Elif Call 2P-CLRAS.PSign()* *and Obtain* $\hat{\sigma}^i_{\tilde{\mathrm{sk}}_A, \tilde{\mathrm{sk}}_B}$	In MoNet, the peers collaborate to create the unadapted signature $\hat{\sigma}^i_{\tilde{\mathrm{sk}}_A, \tilde{\mathrm{sk}}_B}$ for new $T_{x^i_c}$ by using $\tilde{\mathrm{sk}}_X$ and verifying with S^i_X.

		This shared DKG Monero wallet function is replaced entirely during MoNet. ... TODO

5.1.2.3. Grease: During Update

As a substitute for **MoNet** protocol stages 11 and 12, 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 13 begins.

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

5.1.2.3.1. Inputs:

During Update		
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)

The ZKP operations produce output values, 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 usage:

5.1.2.3.2. Outputs:

During Update		
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 private key protecting access to close the payment channel (witness_i)
S_i	Public	The public key/curve point on Ed25519 for ω_i

C	Public	The Fiat–Shamir heuristic challenge (challenge_bytes)
$\Delta_{\text{BabyJubjub}}$	Private	Optimization parameter (response_div_BabyJubjub)
$\rho_{\text{BabyJubjub}}$	Public	The Fiat–Shamir heuristic challenge response on the Baby Jubjub curve (response_BabyJubJub)
Δ_{Ed25519}	Private	Optimization parameter (response_div_BabyJubJub)
ρ_{Ed25519}	Public	The Fiat–Shamir heuristic challenge response on the Ed25519 curve (response_div_ed25519)
C	Public	The Fiat–Shamir heuristic challenge (challenge_bytes)
$R_{\text{BabyJubjub}}$	Public	DLEQ commitment 1, which is a public key/ curve point on Baby Jubjub (R_1)
R_{Ed25519}	Public	DLEQ commitment 2, which is a public key/ curve point on Ed25519 (R_2)

During the update stage, the following operations are performed:

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

Particular details about these operations can be found in **Part 2: Grease ZKP Operations**.

5.1.2.4. Grease: After 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 following resources and information are available and must be stored:

After Update	
Resource	
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

5.1.2.5. MoNet: After Update

When complete the following resources and information are available and must be stored:

After Initialization	
Resource	
Channel Balance	The two values in XMR (with either but not both allowed as zero) for the peers
$T_{x_c^i}$	The commitment transaction that closes the channel, with the **vout** XMR values equal to **Channel Balance**

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

5.2. Part 2: 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.

5.2.1. VerifyWitness0

5.2.1.1. 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`)

5.2.1.2. 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`)

5.2.1.3. 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 $L_{\text{BabyJubjub}}$.

5.2.1.4. Methods

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

5.2.2. FeldmanSecretShare_2_of_2

5.2.2.1. Inputs:

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

5.2.2.2. Outputs:

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

5.2.2.3. Summary

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

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

The scalar order of the Baby Jubjub curve is represented here by $L_{\text{BabyJubjub}}$.

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

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

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

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

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

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

5.2.2.4. Methods

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

5.2.3. ReconstructFeldmanSecretShare_2_of_2

5.2.3.1. Inputs:

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

5.2.3.2. Outputs:

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

5.2.3.3. Summary

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

The scalar order of the Baby Jubjub curve is represented here by $L_{\text{BabyJubjub}}$.

5.2.3.4. Methods

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

5.2.4. VerifyFeldmanSecretShare_peer

5.2.4.1. Inputs:

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

5.2.4.2. Outputs:

There are no outputs.

5.2.4.3. Summary

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

5.2.4.4. Methods

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

5.2.5. VerifyFeldmanSecretShare_KES

5.2.5.1. Inputs:

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

5.2.5.2. Outputs:

There are no outputs.

5.2.5.3. Summary

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

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

5.2.5.4. Methods

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

5.2.6. VerifyEncryptMessage

5.2.6.1. Inputs:

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

5.2.6.2. Outputs:

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

5.2.6.3. Summary

The **VerifyEncryptMessage** operation is a Noir ZK circuit using the UltraHonk prover/verifier. It receives the provided secret data and random entropy inputs. The output are the perfectly binding public key commitment and the perfectly hiding encrypted scaler 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 $\Pi = \kappa \cdot G_{\mathrm{BabyJubjub}}$, the calculation is:

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

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

The scalar order of the Baby Jubjub curve is represented here by $L_{\mathrm{BabyJubjub}}$.

5.2.6.4. Methods

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

5.2.7. DecryptMessage

5.2.7.1. Inputs:

Input	Visibility	
κ	Private	The private key for the public key Π
Φ	Public	The ephemeral public key/curve point on Baby Jubjub for message transportation (`fi`)
χ	Public	The encrypted value of σ (`enc`)

5.2.7.2. Outputs:

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

5.2.7.3. Summary

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

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

The scalar order of the Baby Jubjub curve is represented here by $L_{\mathrm{BabyJubjub}}$.

5.2.7.4. Methods

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

5.2.8. VerifyWitnessSharing

5.2.8.1. Inputs:

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

5.2.8.2. Outputs:

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

5.2.8.3. Summary

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

5.2.8.4. Methods

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

5.2.9. VerifyCOF

5.2.9.1. Inputs:

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

5.2.9.2. Outputs:

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

5.2.9.3. Summary

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

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

The scalar order of the Baby Jubjub curve is represented here by $L_{\text{BabyJubjub}}$.

5.2.9.4. Methods

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

5.2.10. VerifyEquivalentModulo

5.2.10.1. Inputs:

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

5.2.10.2. Outputs:

Output	Visibility	
T_i	Public	The public key/curve point on Baby Jubjub for ω_i
S_i	Public	The public key/curve point on Ed25519 for ω_i
C	Public	The Fiat–Shamir heuristic challenge (challenge_bytes)
$\Delta_{\text{BabyJubjub}}$	Private	Optimization parameter (response_div_BabyJubjub)
$\rho_{\text{BabyJubjub}}$	Public	The Fiat–Shamir heuristic challenge response on the Baby Jubjub curve (response_BabyJubJub)
Δ_{Ed25519}	Private	Optimization parameter (response_div_BabyJubJub)
ρ_{Ed25519}	Public	The Fiat–Shamir heuristic challenge response on the Ed25519 curve (response_div_ed25519)

5.2.10.3. Summary

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

entropy inputs and produces the random outputs. The circuit is not ZK across the inputs since part of the private outputs can be used to reveal information about the private input. The T_i, S_i, $\rho_{\text{BabyJubjub}}$, and ρ_{Ed25519} 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 $\Delta_{\text{BabyJubjub}}$ and Δ_{Ed25519} 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 $L_{\text{BabyJubjub}}$. The scalar order of the Ed25519 curve is represented here by L_{Ed25519}.

5.2.10.4. Methods

$$
\begin{aligned}
T_i &= \omega_i \cdot G_{\text{BabyJubjub}} \\
S_i &= \omega_i \cdot G_{\text{Ed25519}} \\
C &= H_{\text{blake2s}}(\text{HEADER} \parallel \text{PACKED}(T_i) \parallel \text{PACKED}(S_i)) \\
\rho &= \omega_i * C - \nu_{\text{DLEQ}} \\
\rho_{\text{BabyJubjub}} &= \rho \bmod L_{\text{BabyJubjub}} \\
\Delta_{\text{BabyJubjub}} &= \frac{\rho - \rho_{\text{BabyJubjub}}}{L_{\text{BabyJubjub}}} \\
\rho_{\text{Ed25519}} &= \rho \bmod L_{\text{Ed25519}} \\
\Delta_{\text{Ed25519}} &= \frac{\rho - \rho_{\text{Ed25519}}}{L_{\text{Ed25519}}}
\end{aligned}
\tag{14}
$$

5.2.11. VerifyDLEQ

5.2.11.1. Inputs:

Input	Visibility	
T_i	Public	The public key/curve point on Baby Jubjub for ω_i

$\rho_{\text{BabyJubjub}}$	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_{Ed25519}	Public	The Fiat–Shamir heuristic challenge response on the Ed25519 curve (response_div_ed25519)

5.2.11.2. Outputs:

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

5.2.11.3. Summary

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

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

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

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

5.2.11.4. Methods

$$
\begin{aligned}
C &= H_{\text{blake2s}}(\text{HEADER} \parallel \text{PACKED}(T_i) \parallel \text{PACKED}(S_i)) \\
\mathrm{P}_{\text{BabyJubjub}} &= \rho_{\text{BabyJubjub}} \cdot G_{\text{BabyJubjub}} \\
C_{T_i} &= C \cdot G_{\text{BabyJubjub}} \\
R_{\text{BabyJubjub}} &= C_{T_i} - \mathrm{P}_{\text{BabyJubjub}} \\
\mathrm{P}_{\text{Ed25519}} &= \rho_{\text{Ed25519}} \cdot G_{\text{Ed25519}} \\
C_{S_i} &= C \cdot G_{\text{Ed25519}} \\
R_{\text{Ed25519}} &= C_{S_i} - \mathrm{P}_{\text{Ed25519}}
\end{aligned}
\tag{15}
$$

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. Zhimei Sui. *Payment Channel Network for Scriptless Blockchains.* 2023. https://bridges.monash.edu/articles/thesis/Payment_Channel_Network_for_Scriptless_Blockchains/23909907°