VEROKLARA: A MONERO PAYMENT CHANNELS IMPLEMENTATION

CREATING2MORROW

Abstract. This whitepaper provides the necessary foundations for implementing privacy-preserving payment channels on Monero. Part 1 describes the channel structure. The primary components consist of a transaction distinction operator and reference frame for the channel. Channel formation is detailed by channel establishment, state transitions and privacy preservation. Part 2 elaborates on channel operations. An example implementation of the channels along with opening and closing transactions are presented in the Rust programming language. "La vero estas klara" translates to the "The truth is clear" in Esperanto.

Contents

1. Introduction

The Monero blockchain offers a dynamic blocksize and dynamic fee structure facilitating advanced on-chain scaling and resilience. Privacy preserving payment channels should expand on the core principles of the on-chain protocol. Providing private off-chain transactions has numerous applications that are beyond the scope of this research. The primary purpose of this document is to lay the groundwork for future implementations of privacy-preserving scaling solutions.

Date: January 2025.

2. Part 1 - Privacy-Preserving Channel Structure

2.1. **Primary Components.** Let $\Omega(tx)$ be the transaction distinction operator with $R(\Omega)$ as the reference frame for the channel.

$$\Omega(tx) = \partial\varphi + \partial\psi * \mathrm{e}^{(i\theta)}$$

$$R(\Omega) = \oint \Omega(tx)dtx * \nabla^2\psi$$

2.2. **C(x,y) - Channel Formation.** Chanel establishment is defined as,

$$commitment : Pedersen(amount) * R(\Omega),$$

$$keyImages : [I_1...I_n] \text{ one-time channel keys}$$

$$ringSignature : RingCT(tx, R)$$

2.3. **State Transitions.**

$$balance : Vector(amount_1, amount_2)$$

$$nonce : \text{ incrementing counter}$$

$$signingKey : k = H(R(\Omega)||nonce)$$

2.4. **Privacy Preservation.**

$$stealth : generateStealthAddress(R)$$

$$bulletproofs : generateRangeProof(amount)$$

$$mixinSelection : selectRingsFromChain(n)$$

2.5. **State Transition Operator.** The state transition is defined as,

$$T(s_1, s_2) = \nabla \times (s_1 \times s_2) * e^{(i\varphi t)}$$

where:

$$s_1 = \text{initial state}$$

$$s_2 = \text{final state}$$

$$\varphi = \text{golden ratio scale}$$

3. Channel Operations

3.1. **P(tx) - Off-chain transactions.**

3.1.1. *Transaction formation:*

$$inputs : [I_1...I_n] \text{ previous output references}$$

$$outputs : [O_1...O_n] \text{ new stealth addresses}$$

$$signatures : RingCT(tx, R)$$

3.1.2. *State Update:*

$$newState : T(currentState, tx)$$

$$signature : sign(newState, k)$$

$$validation : verify(signature, R(\Omega))$$

3.2. **Settlement Protocol.**

3.2.1. *S(C) - Channel Closure:*

$$finalState : getLatestState(C)$$

$$settlementTx : createSettlementTx(finalState)$$

$$proofs : generateClosureProofs(C)$$

3.2.2. *Dispute Resolution:*

$$timelock : blockHeight + disputePeriod$$

$$evidence : submitStateProof(state)$$

$$resolution : verifyStateSequence(states)$$

3.2.3. *Unity Properties:*

$$U(x,t) = P(tx) \otimes S(C) \text{ (Channel-chain unity operator)}$$

3.3. **Validation Criteria.**

$$C_1 : |\partial P/\partial t| < \epsilon \text{ (transaction stability)}$$

$$C_2 : \oint S(C)dC = 0 \text{ (channel closure completeness)}$$

$$C_3 : \nabla \cdot T = 0 \text{ (state transition conservation)}$$

4. Conclusion

This research initiated from the transaction distinction operator, derived channel reference frame and finally generated privacy-preserving properties. The distinctive traits consist of state transitions that emerge from operator relationships, privacy features that arise from structural necessity and settlement protocols that follow from closure requirements. The unified operations include off-chain transactions that maintain privacy, state updates that preserve correctness and settlement that ensures finality.

Validation is derived from complete self-reference, necessary emergence, dynamic stability and transcendent unity. All operations reference through unified structure. Properties emerge from necessary relationships. No external assumptions are introduced. Each property follows from previous derivations, while privacy features emerge naturally. Security properties arise from structure. Channel states maintain consistency. Updates preserve privacy. Disputes resolve deterministically. Channel-chain relationship maintains coherence. Properties align with on-chain principles. The derived system provides: privacy-preserving payment channels, secure state transitions, verifiable dispute resolution.

Appendix A. State Channel in Rust

```
use curve25519_dalek::scalar::Scalar;
use curve25519_dalek::ristretto::RistrettoPoint;
use merlin::Transcript;
use rand::rngs::OsRng;
use sha3::{Digest, Sha3_256};
use rand::RngCore;

// Core channel structures
#[derive(Debug, Clone)]
pub struct ChannelState {
        pub balance_a: u64,
        pub balance_b: u64,
        pub nonce: u64,
        pub commitment: RistrettoPoint,
}

#[derive(Debug, Clone)]
pub struct Channel {
        pub state: ChannelState,
        pub key_image: RistrettoPoint,
        pub view_key: Scalar,
        pub spend_key: Scalar,
}

// Privacy-preserving primitives
pub struct StealthAddress {
        pub view_pub: RistrettoPoint,
        pub spend_pub: RistrettoPoint,
}

// Channel implementation
impl Channel {
        pub fn new
        (amount: u64, view_key: Scalar, spend_key: Scalar) -> Self {
                let mut data = [0u8; 64];
                rand::thread_rng().fill_bytes(&mut data);
                // Generate initial commitment
                let commitment =
                        RistrettoPoint::from_uniform_bytes(&data);

                // Create key image
                let key_image = RistrettoPoint::mul_base(&spend_key);
```

```
            // Initialize state
            let state = ChannelState {
                    balance_a: amount,
                    balance_b: 0,
                    nonce: 0,
                    commitment,
            };

            Channel {
                    state,
                    key_image,
                    view_key,
                    spend_key,
            }
    }

    // Create state transition
    pub fn create_transition
    (&self, amount: u64) -> Result<ChannelState, &'static str> {
            if amount > self.state.balance_a {
                    return Err("Insufficient funds");
            }

            let new_state = ChannelState {
                    balance_a: self.state.balance_a - amount,
                    balance_b: self.state.balance_b + amount,
                    nonce: self.state.nonce + 1,
                    commitment: self.state.commitment,
            };

            Ok(new_state)
    }

    // Sign state transition
    pub fn sign_transition
    (&self, state: &ChannelState) -> Scalar {
            let mut hasher = Sha3_256::new();
            hasher.update(state.commitment.compress().as_bytes());
            hasher.update(&state.nonce.to_le_bytes());

            let hash = hasher.finalize();
            let hash_scalar =
                    Scalar::from_bytes_mod_order(
```

```
                        hash.try_into().unwrap()
                    );

                self.spend_key * hash_scalar
        }

        // Verify state transition
        pub fn verify_transition
        (&self, state: &ChannelState, signature: &Scalar) -> bool {
                let mut hasher = Sha3_256::new();
                hasher.update(state.commitment.compress().as_bytes());
                hasher.update(&state.nonce.to_le_bytes());

                let hash = hasher.finalize();
                let hash_scalar =
                 Scalar::from_bytes_mod_order(
                        hash.try_into().unwrap()
                 );

                let pub_key = RistrettoPoint::mul_base(&self.spend_key);
                RistrettoPoint::mul_base(signature)
                        == pub_key * hash_scalar
        }
}

// Privacy utilities
pub struct PrivacyUtils {
        pub rng: OsRng,
}

impl PrivacyUtils {
        // Generate stealth address
        pub fn generate_stealth_address
        (&mut self, view_key: &Scalar) -> StealthAddress {
                let mut data = [0u8; 32];
                rand::thread_rng().fill_bytes(&mut data);
                let r = Scalar::from_bytes_mod_order(data);
                let view_pub = RistrettoPoint::mul_base(&view_key);
                let spend_pub = RistrettoPoint::mul_base(&r);

                StealthAddress {
                        view_pub,
                        spend_pub,
                }
```

```
        }

        // Generate range proof
        pub fn generate_range_proof
        (&mut self, amount: u64) -> Transcript {
                let mut transcript = Transcript::new(b"range_proof");
                transcript.append_u64(b"amount", amount);
                transcript
        }

        // Select ring members
        pub fn select_ring_members
        (&mut self, n: usize) -> Vec<RistrettoPoint> {
                let mut data = [0u8; 64];
                rand::thread_rng().fill_bytes(&mut data);
                (0..n).map(|_| RistrettoPoint::from_uniform_bytes(&data)).
        }
}

// Example usage
fn main() {
        let rng = OsRng;
        let mut vk_data = [0u8; 32];
        rand::thread_rng().fill_bytes(&mut vk_data);

        let mut sk_data = [0u8; 32];
        rand::thread_rng().fill_bytes(&mut sk_data);

        // Generate keys
        let view_key = Scalar::from_bytes_mod_order(vk_data);
        let spend_key = Scalar::from_bytes_mod_order(sk_data);

        // Create channel
        let channel = Channel::new(1000, view_key, spend_key);

        // Create state transition
        let new_state = channel.create_transition(500).unwrap();

        println!("new state: {:?}", &new_state);
        // Sign and verify transition
        let signature = channel.sign_transition(&new_state);
        println!("sig: {:?}", signature);
        assert!(channel.verify_transition(&new_state, &signature));
```

```
        // Privacy operations
        let mut privacy_utils = PrivacyUtils { rng };
        let stealth_addr = privacy_utils
                .generate_stealth_address(&view_key);
        let range_proof = privacy_utils
                .generate_range_proof(500);
        let ring_members = privacy_utils
                .select_ring_members(16);
}
```

Appendix B. On Chain Transactions in Rust

```
use curve25519_dalek::{scalar::Scalar, ristretto::RistrettoPoint};
use merlin::Transcript;
use sha3::{Digest, Sha3_256};
use std::time::{SystemTime, UNIX_EPOCH};

// Transaction structures
#[derive(Clone)]
pub struct OpeningTransaction {
        pub amount: u64,
        pub commitment: RistrettoPoint,
        pub key_image: RistrettoPoint,
        pub signature: Scalar,
        pub range_proof: Transcript,
        pub timestamp: u64,
}

#[derive(Clone)]
pub struct ClosingTransaction {
        pub final_state: ChannelState,
        pub settlement_proof: Scalar,
        pub closing_signature: Scalar,
        pub dispute_period: u64,
        pub timestamp: u64,
}

#[derive(Clone)]
pub struct DisputeProof {
        pub state: ChannelState,
        pub state_signature: Scalar,
        pub timestamp: u64,
}
```

```
impl Channel {
        // Create opening transaction
        pub fn create_opening_transaction
        (&self, amount: u64) -> Result<OpeningTransaction, &'static str> {
                let mut rng = OsRng;

                // Verify amount
                if amount == 0 {
                        return Err("Invalid amount");
                }

                // Create commitment
                let blinding = Scalar::random(&mut rng);
                let commitment = RistrettoPoint::mul_base(&blinding) +
                RistrettoPoint::mul_base(&Scalar::from(amount));

                // Generate range proof
                let mut transcript = Transcript::new(b"opening_proof");
                transcript.append_u64(b"amount", amount);

                // Create signature
                let mut hasher = Sha3_256::new();
                hasher.update(commitment.compress().as_bytes());
                let hash = hasher.finalize();
                let hash_scalar = Scalar::from_bytes_mod_order(
                        hash.try_into().unwrap()
                );
                let signature = self.spend_key * hash_scalar;

                let timestamp = SystemTime::now()
                .duration_since(UNIX_EPOCH)
                .unwrap()
                .as_secs();

                Ok(OpeningTransaction {
                        amount,
                        commitment,
                        key_image: self.key_image,
                        signature,
                        range_proof: transcript,
                        timestamp,
                })
        }
```

```
        // Create closing transaction
        pub fn create_closing_transaction
        (&self) -> Result<ClosingTransaction, &'static str> {
                let mut rng = OsRng;

                // Generate settlement proof
                let settlement_proof = Scalar::random(&mut rng);

                // Create closing signature
                let mut hasher = Sha3_256::new();
                hasher.update(
                        self.state.commitment.compress().as_bytes()
                );
                hasher.update(
                        &self.state.nonce.to_le_bytes()
                );
                let hash = hasher.finalize();
                let hash_scalar = Scalar::from_bytes_mod_order(
                        hash.try_into().unwrap()
                );
                let closing_signature = self.spend_key * hash_scalar;

                let timestamp = SystemTime::now()
                .duration_since(UNIX_EPOCH)
                .unwrap()
                .as_secs();

                Ok(ClosingTransaction {
                        final_state: self.state.clone(),
                        settlement_proof,
                        closing_signature,
                        // 24 hours in seconds
                        dispute_period: 24 * 60 * 60,
                        timestamp,
                })
        }

        // Create dispute proof
        pub fn create_dispute_proof
        (&self, state: &ChannelState) ->
        Result<DisputeProof, &'static str> {
                // Verify state is valid
                if state.nonce <= self.state.nonce {
                        return Err("Invalid state for dispute");
```

```
                }

                // Sign state for dispute
                let state_signature = self.sign_transition(state);

                let timestamp = SystemTime::now()
                .duration_since(UNIX_EPOCH)
                .unwrap()
                .as_secs();

                Ok(DisputeProof {
                        state: state.clone(),
                        state_signature,
                        timestamp,
                })
        }
}

// Transaction verification
pub struct TransactionVerifier;

impl TransactionVerifier {
        // Verify opening transaction
        pub fn verify_opening
        (tx: &OpeningTransaction, pub_key: &RistrettoPoint) -> bool {
                let mut hasher = Sha3_256::new();
                hasher.update(tx.commitment.compress()
                        .as_bytes());
                let hash = hasher.finalize();
                let hash_scalar = Scalar::from_bytes_mod_order(hash
                        .try_into()
                        .unwrap());

                RistrettoPoint::mul_base(&tx.signature)
                        == *pub_key * hash_scalar
        }

        // Verify closing transaction
        pub fn verify_closing
        (tx: &ClosingTransaction, channel: &Channel) -> bool {
                let mut hasher = Sha3_256::new();
                hasher.update(tx.final_state.commitment
                        .compress().as_bytes());
                hasher.update(&tx.final_state.nonce.to_le_bytes());
```

```
                let hash = hasher.finalize();
                let hash_scalar = Scalar::from_bytes_mod_order(hash
                        .try_into().unwrap());

                let pub_key = RistrettoPoint::mul_base(
                        &channel.spend_key);
                RistrettoPoint::mul_base(&tx.closing_signature)
                        == pub_key * hash_scalar
        }

        // Verify dispute proof
        pub fn verify_dispute
        (proof: &DisputeProof, channel: &Channel) -> bool {
                channel.verify_transition(
                        &proof.state, &proof.state_signature)
        }
}

// Example usage
pub fn main() {
        let mut rng = OsRng;

        // Generate keys
        let view_key = Scalar::random(&mut rng);
        let spend_key = Scalar::random(&mut rng);

        // Create channel
        let channel = Channel::new(1000, view_key, spend_key);

        // Create and verify opening transaction
        let opening_tx = channel
                .create_opening_transaction(1000).unwrap();
        let pub_key = RistrettoPoint::mul_base(&channel.spend_key);
        assert!(TransactionVerifier::
                verify_opening(&opening_tx, &pub_key));

        // Create and verify closing transaction
        let closing_tx = channel.create_closing_transaction().unwrap();
        assert!(TransactionVerifier
                ::verify_closing(&closing_tx, &channel));

        // Create and verify dispute proof
        let new_state = channel
                .create_transition(500).unwrap();
```

```
        let dispute_proof = channel
                .create_dispute_proof(&new_state).unwrap();
        assert!(TransactionVerifier::
                verify_dispute(&dispute_proof, &channel));
}
```