Overpass Channels: Horizontally Scalable, Privacy-Enhanced, with Independent Verification, Fluid Liquidity, and Robust Censorship Proof, Payments on TON

Brandon "Cryptskii" Ramsay

September 2024

Contents

Abstract

Overpass Channels presents a groundbreaking approach to blockchain scalability, offering a horizontally scalable, privacy-enhanced payment network with independent verification, fluid liquidity, and robust censorship resistance. This paper introduces a novel architecture that leverages zero-knowledge proofs, specifically zk-SNARKs, to ensure transaction validity and privacy while enabling unprecedented throughput and efficiency. By eliminating the need for traditional consensus mechanisms and miners, Overpass Channels achieves remarkable cost-effectiveness and energy efficiency. The system's design focuses on unilateral payment channels and off-chain transaction processing, allowing for high-speed, low-latency operations without compromising security or decentralization. This paper provides a comprehensive analysis of the Overpass Channels system, including its cryptographic foundations, scalability metrics, integration, and potential applications across various domains, from global payments to confidential voting systems and secure health record management.

1 Introduction

The advent of blockchain technology has revolutionized the concept of digital payments, offering unprecedented levels of security and decentralization. However, the widespread adoption of blockchain-based payment systems has been hindered by several critical issues, including high transaction fees, limited scalability, and privacy concerns. Overpass Channels addresses these challenges head-on by introducing a novel Layer 2 solution built on the blockchain.

Overpass Channels represents a paradigm shift in blockchain scalability and privacy. Unlike traditional blockchain networks that rely on miners or validators to achieve consensus, Overpass Channels eliminates these potential bottlenecks through its innovative use of unilateral payment channels and off-chain transaction processing. This approach not only ensures high throughput but also maintains a strong focus on user privacy through the implementation of zero-knowledge proofs, specifically zk-SNARKs.

The key innovations of Overpass Channels include:

1. **Horizontal Scalability:** By leveraging a unique channel structure and off-chain processing, Overpass Channels can scale horizontally, supporting an theoretically unlimited number of transactions without the need for additional on-chain resources.

2. **Privacy-Enhanced Transactions:** Through the use of zk-SNARKs, every transaction in the Overpass Channels network is cryptographically proven to be valid without revealing any sensitive information about the transaction details or the parties involved.

3. **Independent Verification:** Each transaction can be independently verified by the parties involved, eliminating the need for network-wide consensus and dramatically reducing latency.

4. **Fluid Liquidity:** The system's design allows for dynamic rebalancing of payment channels, ensuring optimal distribution of liquidity across the network.

5. **Robust Censorship Resistance:** By decentralizing the transaction processing and verification, Overpass Channels makes it extremely difficult for any single entity to censor or block transactions.

This paper provides a comprehensive exploration of the Overpass Channels system, including its cryptographic foundations, scalability analysis, integration with the TON blockchain, and potential applications across various domains. We begin with a detailed examination of the system's architecture, followed by an in-depth analysis of its security properties and scalability metrics. We then discuss the integration of Overpass Channels with the blockchain, highlighting the synergies between the two systems. Finally, we explore several use cases that demonstrate the versatility and potential impact of Overpass Channels in real-world scenarios.

As we delve into the technical details of Overpass Channels, we will provide rigorous mathematical proofs, algorithmic descriptions, and concrete examples to illustrate the system's functionality and benefits. Throughout this paper, we will use the hypothetical users Alice and Bob to demonstrate various scenarios and interactions within the Overpass Channels network.

Let us now embark on a journey through the intricate workings of Overpass Channels, a system poised to redefine the landscape of blockchain-based payments and beyond.

2 Key Innovations

Overpass Channels introduces several groundbreaking innovations that set it apart from existing blockchain and Layer 2 solutions. These innovations work in concert to create a system that is not only highly scalable and efficient but also privacy-preserving and resistant to censorship. Let's explore each of these key innovations in detail:

2.1 Horizontal Scalability without Validators

One of the most significant innovations of Overpass Channels is its ability to scale horizontally without the need for validators or miners. This is achieved through a combination of unilateral payment channels and off-chain transaction processing.

Definition 1 (Unilateral Payment Channel)**.** *A unilateral payment channel is a cryptographic construct that allows two parties to conduct multiple transactions off-chain, with only the opening and closing of the channel requiring on-chain operations.*

In Overpass Channels, each user can open multiple unilateral payment channels, forming a network of interconnected channels. Transactions within these channels are processed off-chain, dramatically reducing the load on the underlying blockchain.

Theorem 1 (Horizontal Scalability)**.** *The transaction throughput of the Overpass Channels network scales linearly with the number of active channels, independent of the underlying blockchain's capacity.*

Proof. Let n be the number of active channels in the network, and t be the average number of transactions per second (TPS) that can be processed within a single channel. The total network throughput T is given by:

$$T = n \times t$$

As n increases, T increases linearly, without being constrained by the underlying blockchain's capacity. This is because transactions within channels are processed off-chain and do not require immediate on-chain validation. ■

This horizontal scalability allows Overpass Channels to support an theoretically unlimited number of transactions, limited only by the number of active channels and the computational capacity of the network nodes.

2.2 Privacy-Enhanced Transactions with zk-SNARKs

Privacy is a cornerstone of Overpass Channels, achieved through the innovative use of zero-knowledge Succinct Non-interactive Arguments of Knowledge (zk-SNARKs).

Definition 2 (zk-SNARK)**.** *A zk-SNARK is a cryptographic proof construction where one can prove possession of certain information, e.g., a secret key, without revealing that information, and without any interaction between the prover and verifier.*

Algorithm 1 zk-SNARK Transaction Proof Generation

```
procedure GenerateTransactionProof(tx, sk, pk)
    inputs ← ExtractPublicInputs(tx)
    witness ← ConstructWitness(tx, sk)
    proof ← ProveZKSNARK(inputs, witness, pk)
    return proof
end procedure
```

In Overpass Channels, every transaction is accompanied by a zk-SNARK proof that demonstrates the validity of the transaction without revealing any sensitive details.

This approach ensures that while the network can verify the validity of transactions, it cannot access sensitive information such as transaction amounts, sender and receiver identities, or channel balances.

2.3 Independent Verification and Instant Finality

Overpass Channels achieves instant finality through a mechanism of independent verification, eliminating the need for network-wide consensus.

Theorem 2 (Instant Finality)**.** *In Overpass Channels, a transaction is considered final and irreversible as soon as it is verified by the recipient, without requiring confirmation from any other network participants.*

Proof. Let T be a transaction from Alice to Bob within a channel C. The proof proceeds as follows:

1) Alice generates a zk-SNARK proof P for T. 2) Alice sends T and P to Bob. 3) Bob verifies P using the public verification key vk. 4) If P is valid, Bob accepts T as final.

Since the verification of P depends only on vk, which is publicly known, and the inputs provided in T, Bob can independently verify the transaction without consulting any other network participants. Once Bob has verified P, he can be certain that T is valid and will be accepted by the network in any future on-chain settlement. ■

This independent verification mechanism allows for near-instantaneous transaction finality, a significant improvement over traditional blockchain systems that require multiple confirmations.

2.4 Fluid Liquidity through Dynamic Rebalancing

Overpass Channels incorporates a dynamic rebalancing mechanism that ensures optimal distribution of liquidity across the network.

Definition 3 (Channel Liquidity)**.** *The liquidity of a channel is defined as the maximum amount that can be sent through the channel in a single direction without requiring an on-chain transaction.*

The dynamic rebalancing algorithm continuously monitors channel liquidity and initiates rebalancing operations when necessary.

This fluid liquidity ensures that users can always find efficient paths for their transactions, even as usage patterns in the network change over time.

Algorithm 2 Dynamic Channel Rebalancing

```
procedure RebalanceChannel(channel, threshold)
    balance ← GetChannelBalance(channel)
    capacity ← GetChannelCapacity(channel)
    if balance < threshold × capacity then
        amount ← (capacity − balance)/2
        proof ← GenerateRebalanceProof(channel, amount)
        ExecuteRebalance(channel, amount, proof)
    end if
end procedure
```

2.5 Dynamic Rebalancing Analysis

In this section, we present a comprehensive analysis of the dynamic rebalancing mechanism within the Overpass Channels network. This mechanism is crucial for maintaining optimal liquidity distribution across the network, ensuring efficient transaction processing while preserving user privacy and system decentralization.

Formal Definition of the Dynamic Rebalancing Algorithm

The Overpass Channels network consists of a hierarchical structure of smart contracts and off-chain components, defined as follows:

Definition 4 (Overpass Channels Network Components)**.** *The Overpass Channels network comprises:*

- ***Channel Contracts (CC):*** *Off-chain unilateral payment channels between users.*
- ***Wallet Extension Contracts (WEC):*** *Smart contracts that manage multiple channels for a user, represented using an off-chain sparse Merkle tree.*
- ***Intermediate Contracts (IC):*** *Off-chain contracts that aggregate state updates from multiple wallet extensions.*
- ***Root Contract (RC):*** *On-chain contract that maintains the global state of the network on the TON blockchain.*
- ***Off-chain Storage Nodes (OSN):*** *Decentralized nodes that store redundant copies of off-chain states and proofs.*

The dynamic rebalancing mechanism operates primarily at two levels: within individual Wallet Extension Contracts (intra-WEC rebalancing) and between different Wallet Extension Contracts managed by the same Intermediate Contract (inter-WEC rebalancing).

Definition 5 (Dynamic Rebalancing Mechanism)**.** *The dynamic rebalancing mechanism is a process that redistributes liquidity:*

- *Within a Wallet Extension Contract, among its managed Channel Contracts.*
- *Between different Wallet Extension Contracts managed by the same Intermediate Contract.*

This process aims to optimize liquidity distribution, ensuring efficient transaction processing across the network while maintaining privacy and minimizing on-chain operations.

Let us now formally define the components and variables involved in the rebalancing process:

Definition 6 (Rebalancing Components and Variables)**.** *For a given Wallet Extension Contract W:*

- $\mathcal{C}_W = \{C_1, C_2, \ldots, C_n\}$ *is the set of Channel Contracts managed by* W*.*
- B_{C_i} *is the balance of Channel Contract* C_i*.*
- ΔB_{C_i} *is the adjustment to the balance of* C_i *through rebalancing.*
- θ_i *is the target liquidity ratio for* C_i*, representing the desired proportion of the total liquidity.*
- $L_W = \sum_{i=1}^{n} B_{C_i}$ *is the total liquidity managed by* W*.*

For an Intermediate Contract IC managing multiple Wallet Extension Contracts:

- $\mathcal{W} = \{W_1, W_2, \ldots, W_m\}$ *is the set of Wallet Extension Contracts managed by* IC*.*
- B_{W_j} *is the total balance of Wallet Extension Contract* W_j*.*
- ΔB_{W_j} *is the adjustment to the balance of* W_j *through inter-WEC rebalancing.*
- ϕ_j *is the target liquidity ratio for* W_j *within* IC*.*
- $L_{IC} = \sum_{j=1}^{m} B_{W_j}$ *is the total liquidity managed by* IC*.*

Intra-WEC Rebalancing Algorithm

The intra-WEC rebalancing process occurs within a single Wallet Extension Contract, redistributing liquidity among its managed Channel Contracts. This process is executed off-chain, with periodic updates to the off-chain storage nodes.

Algorithm 3 Intra-WEC Rebalancing

1: **procedure** IntraWECRebalance(W)
2: $\mathcal{C}_W \leftarrow$ GetChannelContracts(W)
3: $L_W \leftarrow \sum_{C_i \in \mathcal{C}_W} B_{C_i}$
4: $\{\Delta B_{C_i}\} \leftarrow$ SolveOptimizationProblem($\mathcal{C}_W, L_W, \{\theta_i\}$)
5: **for** $C_i \in \mathcal{C}_W$ **do**
6: $B_{C_i} \leftarrow B_{C_i} + \Delta B_{C_i}$
7: **end for**
8: $newRoot \leftarrow$ UpdateMerkleTree($W, \{B_{C_i}\}$)
9: $proof \leftarrow$ GenerateZKProof($W, newRoot, \{\Delta B_{C_i}\}$)
10: UpdateOffChainStorage($W, newRoot, proof$)
11: **end procedure**

The optimization problem solved in step 4 of Algorithm 3 is formulated as follows:

$$
\begin{aligned}
\min_{\{\Delta B_{C_i}\}} \quad & \sum_{i=1}^{n} w_i (B_{C_i} + \Delta B_{C_i} - \theta_i L_W)^2 \\
\text{subject to} \quad & \sum_{i=1}^{n} \Delta B_{C_i} = 0 \\
& -B_{C_i} \leq \Delta B_{C_i} \leq L_{C_i} - B_{C_i}, \quad \forall i \in \{1, \ldots, n\}
\end{aligned}
\tag{1}
$$

where $w_i > 0$ is a weighting factor for Channel Contract C_i, and L_{C_i} is the maximum allowed balance for C_i.

Inter-WEC Rebalancing Algorithm

The inter-WEC rebalancing process occurs between different Wallet Extension Contracts managed by the same Intermediate Contract. This process is more complex as it involves multiple parties and requires careful coordination to maintain privacy and security.

Algorithm 4 Inter-WEC Rebalancing

1: **procedure** InterWECRebalance(IC)
2: $\mathcal{W} \leftarrow$ GetWalletExtensionContracts(IC)
3: $L_{IC} \leftarrow \sum_{W_j \in \mathcal{W}} B_{W_j}$
4: $\{\Delta B_{W_j}\} \leftarrow$ SolveOptimizationProblem($\mathcal{W}, L_{IC}, \{\phi_j\}$)
5: **for** $W_j \in \mathcal{W}$ **do**
6: $proof_j \leftarrow$ GenerateZKProof($W_j, \Delta B_{W_j}$)
7: SendRebalanceRequest($W_j, \Delta B_{W_j}, proof_j$)
8: **end for**
9: WaitForConfirmations($\mathcal{W}$)
10: $newRoot_{IC} \leftarrow$ UpdateIntermediateMerkleTree($IC, \{B_{W_j}\}$)
11: $proof_{IC} \leftarrow$ GenerateZKProof($IC, newRoot_{IC}, \{\Delta B_{W_j}\}$)
12: UpdateOffChainStorage($IC, newRoot_{IC}, proof_{IC}$)
13: **end procedure**

The optimization problem solved in step 4 of Algorithm 4 is formulated as:

$$
\begin{aligned}
\min_{\{\Delta B_{W_j}\}} \quad & \sum_{j=1}^{m} v_j (B_{W_j} + \Delta B_{W_j} - \phi_j L_{IC})^2 \\
\text{subject to} \quad & \sum_{j=1}^{m} \Delta B_{W_j} = 0 \\
& -B_{W_j} \leq \Delta B_{W_j} \leq L_{W_j} - B_{W_j}, \quad \forall j \in \{1, \ldots, m\}
\end{aligned}
\tag{2}
$$

where $v_j > 0$ is a weighting factor for Wallet Extension Contract W_j, and L_{W_j} is the maximum allowed balance for W_j.

Analysis of Convergence and Stability

The convergence and stability of the dynamic rebalancing mechanism are crucial for the overall performance and reliability of the Overpass Channels network. We present a formal analysis of these properties for both intra-WEC and inter-WEC rebalancing processes.

Theorem 3 (Convergence of Intra-WEC Rebalancing)**.** *The intra-WEC rebalancing process, as defined by the optimization problem in Equation 1, converges to a unique global optimum.*

Proof. The objective function in Equation 1 is a sum of convex quadratic functions, and thus is itself convex. The constraints form a convex feasible region. Therefore, the optimization problem is convex.

For a convex optimization problem with linear constraints, the Karush-Kuhn-Tucker (KKT) conditions are necessary and sufficient for optimality. Let λ be the Lagrange multiplier for the equality constraint and μ_i^+, μ_i^- be the multipliers for the inequality constraints. The Lagrangian is:

$$\begin{aligned}\mathcal{L} = &\sum_{i=1}^{n} w_i (B_{C_i} + \Delta B_{C_i} - \theta_i L_W)^2 - \lambda \sum_{i=1}^{n} \Delta B_{C_i} \\ &+ \sum_{i=1}^{n} \mu_i^+ (\Delta B_{C_i} - (L_{C_i} - B_{C_i})) + \sum_{i=1}^{n} \mu_i^- (-\Delta B_{C_i} - B_{C_i})\end{aligned} \tag{3}$$

The KKT conditions are:

$$\frac{\partial \mathcal{L}}{\partial \Delta B_{C_i}} = 2w_i (B_{C_i} + \Delta B_{C_i} - \theta_i L_W) - \lambda + \mu_i^+ - \mu_i^- = 0, \quad \forall i \tag{4}$$

$$\sum_{i=1}^{n} \Delta B_{C_i} = 0 \tag{5}$$

$$-B_{C_i} \leq \Delta B_{C_i} \leq L_{C_i} - B_{C_i}, \quad \forall i \tag{6}$$

$$\mu_i^+, \mu_i^- \geq 0, \quad \forall i \tag{7}$$

$$\mu_i^+ (\Delta B_{C_i} - (L_{C_i} - B_{C_i})) = 0, \quad \forall i \tag{8}$$

$$\mu_i^- (-\Delta B_{C_i} - B_{C_i}) = 0, \quad \forall i \tag{9}$$

These conditions uniquely determine the optimal solution, proving the convergence to a global optimum. ■

A similar proof can be constructed for the inter-WEC rebalancing process:

Theorem 4 (Convergence of Inter-WEC Rebalancing)**.** *The inter-WEC rebalancing process, as defined by the optimization problem in Equation 2, converges to a unique global optimum.*

The proof follows the same structure as Theorem 3, with appropriate substitutions for the variables and constraints.

Privacy and Security Analysis

The privacy and security of the rebalancing process are paramount in the Overpass Channels network. We analyze these aspects for both intra-WEC and inter-WEC rebalancing.

Theorem 5 (Privacy Preservation in Intra-WEC Rebalancing)**.** *The intra-WEC rebalancing process preserves the privacy of individual channel balances from external observers.*

Proof. Let $\mathcal{A}$ be an adversary observing the intra-WEC rebalancing process. The information available to $\mathcal{A}$ consists of:

1. The Merkle root of the Wallet Extension Contract before rebalancing: $root_{pre}$ 2. The Merkle root of the Wallet Extension Contract after rebalancing: $root_{post}$ 3. The zk-SNARK proof π of the rebalancing operation

By the properties of cryptographic hash functions used in the Merkle tree, $\mathcal{A}$ cannot derive the individual channel balances from $root_{pre}$ or $root_{post}$. The zk-SNARK proof π only verifies that the rebalancing operation was performed correctly, without revealing any information about the channel balances or the rebalancing amounts.

Therefore, $\mathcal{A}$ gains no information about individual channel balances from observing the rebalancing process. ■

A similar theorem can be stated for inter-WEC rebalancing:

Theorem 6 (Privacy Preservation in Inter-WEC Rebalancing)**.** *The inter-WEC rebalancing process preserves the privacy of individual wallet extension balances from external observers and other participating wallet extensions.*

The proof follows a similar structure to Theorem 5, with the additional consideration that the zk-SNARK proofs exchanged between wallet extensions reveal no information about their individual balances.

Efficiency Analysis

The efficiency of the rebalancing process is crucial for the scalability of the Overpass Channels network. We analyze the computational and communication complexity of both intra-WEC and inter-WEC rebalancing.

Theorem 7 (Computational Complexity of Intra-WEC Rebalancing)**.** *The computational complexity of the intra-WEC rebalancing process is $O(n \log n)$, where n is the number of channel contracts managed by the wallet extension.*

Proof. The intra-WEC rebalancing process consists of the following steps:

1. Solving the optimization problem: $O(n \log n)$ using interior-point methods. 2. Updating the Merkle tree: $O(n \log n)$ for n updates. 3. Generating the zk-SNARK proof: $O(n \log n)$ for a circuit of size $O(n)$.

Therefore, the overall complexity is $O(n \log n)$. ■

Theorem 8 (Communication Complexity of Inter-WEC Rebalancing)**.** *The communication complexity of the inter-WEC rebalancing process is $O(m)$, where m is the number of wallet extension contracts managed by the intermediate contract.*

Proof. The inter-WEC rebalancing process involves the following communication steps:

1. Each wallet extension contract sends its current balance to the intermediate contract: $O(m)$ messages. 2. The intermediate contract sends rebalancing instructions to each wallet extension: $O(m)$ messages. 3. Each wallet extension sends a confirmation back to the intermediate contract: $O(m)$ messages. 4. The intermediate contract sends the final update to the off-chain storage: $O(1)$ message.

Therefore, the total communication complexity is $O(m)$. ■

Illustrative Example: Alice and Bob's Rebalancing Scenario

To provide a concrete understanding of the rebalancing process, let's consider a detailed example involving two users, Alice and Bob, whose wallet extensions are managed by the same intermediate contract.

Initial State

- Intermediate Contract IC_1 manages Wallet Extensions W_{Alice} and W_{Bob}.
- W_{Alice} has three channels: C_{A1}, C_{A2}, C_{A3}.
- W_{Bob} has two channels: C_{B1}, C_{B2}.
- Initial balances:
  - $B_{W_{Alice}} = 100$ units ($B_{C_{A1}} = 20, B_{C_{A2}} = 30, B_{C_{A3}} = 50$)
  - $B_{W_{Bob}} = 200$ units ($B_{C_{B1}} = 80, B_{C_{B2}} = 120$)
  - Total liquidity in IC_1: $L_{IC_1} = 300$ units
- Target ratios:
  - For IC_1: $\phi_{Alice} = 0.4, \phi_{Bob} = 0.6$
  - For W_{Alice}: $\theta_{A1} = 0.3, \theta_{A2} = 0.3, \theta_{A3} = 0.4$
  - For W_{Bob}: $\theta_{B1} = 0.4, \theta_{B2} = 0.6$

Rebalancing Process 1. Inter-WEC Rebalancing:

$$\Delta B_{W_{Alice}} = 0.4 \times 300 - 100 = +20 \text{ units} \tag{10}$$
$$\Delta B_{W_{Bob}} = 0.6 \times 300 - 200 = -20 \text{ units} \tag{11}$$

2. Intra-WEC Rebalancing for Alice:

$$\Delta B_{C_{A1}} = 0.3 \times 120 - 20 = +16 \text{ units} \tag{12}$$
$$\Delta B_{C_{A2}} = 0.3 \times 120 - 30 = +6 \text{ units} \tag{13}$$
$$\Delta B_{C_{A3}} = 0.4 \times 120 - 50 = -2 \text{ units} \tag{14}$$

3. Intra-WEC Rebalancing for Bob:

$$\Delta B_{C_{B1}} = 0.4 \times 180 - 80 = -8 \text{ units} \tag{15}$$
$$\Delta B_{C_{B2}} = 0.6 \times 180 - 120 = -12 \text{ units} \tag{16}$$

Final State After rebalancing, the new balances are:

- W_{Alice}: 120 units (C_{A1}: 36, C_{A2}: 36, C_{A3}: 48)
- W_{Bob}: 180 units (C_{B1}: 72, C_{B2}: 108)

Privacy-Preserving Proof Generation To ensure privacy, Alice and Bob generate zk-SNARK proofs for their respective rebalancing operations:

Algorithm 5 Generate Rebalancing Proof

```
procedure GenerateRebalancingProof(W, {B_{C_i}}, {ΔB_{C_i}}, root_pre, root_post)
    circuit ← LoadRebalancingCircuit()
    witness ← {B_{C_i}, ΔB_{C_i}, root_pre, root_post}
    proof ← GenerateZKProof(circuit, witness)
    return proof
end procedure
```

These proofs allow the intermediate contract to verify that the rebalancing was performed correctly without learning the actual balances or rebalancing amounts.

Conclusion and Future Work

The dynamic rebalancing mechanism in Overpass Channels provides an efficient, privacy-preserving, and decentralized solution for maintaining optimal liquidity distribution. Our analysis and simulations demonstrate its effectiveness in improving transaction success rates and network efficiency.

Future work could explore adaptive rebalancing strategies that dynamically adjust target ratios based on historical transaction patterns and network conditions. Additionally, investigating the impact of different network topologies on rebalancing performance could provide insights for optimizing the overall network structure.

Theorem 9 (Optimality of Dynamic Rebalancing)**.** *The dynamic rebalancing mechanism in Overpass Channels achieves an ϵ-optimal liquidity distribution in $O(\log(1/\epsilon))$ rebalancing rounds, where $\epsilon > 0$ is the desired precision.*

Proof. Let $\mathbf{x}_t$ be the vector of normalized channel balances after the t-th rebalancing round, and $\mathbf{x}^*$ be the optimal balance distribution. Define the Lyapunov function:

$$V(\mathbf{x}_t) = \|\mathbf{x}_t - \mathbf{x}^*\|_2^2 \tag{17}$$

We can show that each rebalancing round decreases $V(\mathbf{x}_t)$ by a factor of at least $(1 - \alpha)$ for some $\alpha > 0$:

$$V(\mathbf{x}_{t+1}) \leq (1 - \alpha)V(\mathbf{x}_t) \tag{18}$$

After T rounds:

$$V(\mathbf{x}_T) \leq (1 - \alpha)^T V(\mathbf{x}_0) \tag{19}$$

To achieve ϵ-optimality, we need:

$$(1-\alpha)^T V(\mathbf{x}_0) \leq \epsilon \tag{20}$$

Solving for T:

$$T \geq \frac{\log(\epsilon/V(\mathbf{x}_0))}{\log(1-\alpha)} = O(\log(1/\epsilon)) \tag{21}$$

Thus, ϵ-optimal liquidity distribution is achieved in $O(\log(1/\epsilon))$ rounds. ■

This theoretical result underscores the efficiency of the dynamic rebalancing mechanism in rapidly converging to an optimal liquidity distribution, further validating its practical utility in the Overpass Channels network.

2.6 Cross-Intermediate Contract Rebalancing and Global Liquidity Management

In this section, we present a comprehensive analysis of the cross-intermediate contract rebalancing mechanism and global liquidity management in the Overpass Channels network. This system enables efficient liquidity redistribution across different intermediate contracts, potentially spanning multiple shards in the underlying TON blockchain, while maintaining high transaction finality and minimizing on-chain operations.

System Architecture

The Overpass Channels network consists of a hierarchical structure of components:

Definition 7 (Overpass Channels Network Components)**.** *The network comprises:*

- ***Channel Contracts (CC):*** *Off-chain unilateral payment channels between users.*
- ***Wallet Extension Contracts (WEC):*** *Smart contracts managing multiple channels for a user, represented using an off-chain sparse Merkle tree.*
- ***Intermediate Contracts (IC):*** *Off-chain contracts aggregating state updates from multiple wallet extensions.*
- ***Root Contract (RC):*** *On-chain contract maintaining the global state of the network on the TON blockchain.*
- ***Off-chain Storage Nodes (OSN):*** *Decentralized nodes storing redundant copies of off-chain states and proofs.*

This hierarchical structure facilitates efficient off-chain operations while maintaining the security guarantees of the underlying blockchain.

Algorithm 6 Cross-Intermediate Rebalancing

procedure CROSSICREBALANCE($\mathcal{IC}, RC$)
  $L_{global} \leftarrow \sum_{IC_i \in \mathcal{IC}} B_{IC_i}$
  $\{\Delta B_{IC_i}\} \leftarrow$ SolveGlobalOptimizationProblem($\mathcal{IC}, L_{global}, \{\psi_i\}$)
  for $IC_i \in \mathcal{IC}$ **do**
    $proof_i \leftarrow$ GenerateZKProof($IC_i, \Delta B_{IC_i}$)
    SendRebalanceRequest($IC_i, \Delta B_{IC_i}, proof_i$)
  end for
  WaitForConfirmations($\mathcal{IC}$)
  $newRoot_{global} \leftarrow$ UpdateGlobalMerkleTree($RC, \{B_{IC_i}\}$)
  $proof_{global} \leftarrow$ GenerateZKProof($RC, newRoot_{global}, \{\Delta B_{IC_i}\}$)
  ScheduleForNextEpochSubmission($RC, newRoot_{global}, proof_{global}$)
end procedure

Cross-Intermediate Rebalancing Algorithm

The cross-intermediate rebalancing process operates at the global level, coordinated by the Root Contract. This process is crucial for maintaining optimal liquidity distribution across the entire network, including different shards.

The optimization problem solved in this algorithm is formulated as:

$$\begin{aligned} \min_{\{\Delta B_{IC_i}\}} \quad & \sum_{i=1}^{k} v_i (B_{IC_i} + \Delta B_{IC_i} - \psi_i L_{global})^2 \\ \text{subject to} \quad & \sum_{i=1}^{k} \Delta B_{IC_i} = 0 \\ & -B_{IC_i} \leq \Delta B_{IC_i} \leq L_{IC_i} - B_{IC_i}, \quad \forall i \in \{1, \ldots, k\} \end{aligned} \tag{22}$$

where $v_i > 0$ is a weighting factor for Intermediate Contract IC_i, and L_{IC_i} is the maximum allowed balance for IC_i.

This optimization problem ensures that the rebalancing process minimizes liquidity imbalances across all Intermediate Contracts while respecting balance constraints. The use of zk-SNARK proofs throughout the process maintains privacy and security, even during cross-shard operations.

Transaction Finality and Liquidity Accessibility

A key feature of Overpass Channels is the maintenance of strong transaction finality and continuous liquidity accessibility, even during global rebalancing operations. This is crucial for ensuring a seamless user experience and enabling efficient integration with payment systems.

Theorem 10 (Transaction Finality Preservation)**.** *Individual transactions in Overpass Channels achieve finality independently of cross-intermediate rebalancing operations and cross-shard settlements.*

Proof. Consider a transaction T between two users, Alice and Bob, whose wallets are managed by different Intermediate Contracts IC_A and IC_B, potentially in different shards. The proof proceeds as follows:

1. Transaction Execution: - Alice initiates transaction T to Bob. - A zk-SNARK proof π_T is generated for T.

2. Local Verification: - Bob verifies π_T locally. - If π_T is valid, Bob considers T final from his perspective.

3. Wallet Extension Update: - Alice's and Bob's Wallet Extensions (W_A and W_B) update their local states. - New zk-SNARK proofs π_{W_A} and π_{W_B} are generated for the updated wallet states.

4. Intermediate Contract Update: - IC_A and IC_B verify π_{W_A} and π_{W_B} respectively. - The intermediate contracts update their states and generate proofs π_{IC_A} and π_{IC_B}.

5. Cross-Intermediate Rebalancing: - If a cross-intermediate rebalancing occurs, it generates a separate proof π_{rebal}. - This rebalancing does not invalidate or depend on the proofs of individual transactions.

6. Global State Update: - The Root Contract verifies all proofs: π_{IC_A}, π_{IC_B}, and π_{rebal} (if applicable). - The global state is updated in the next epoch submission.

At each step, the proofs are independently verifiable. The validity of T (proven by π_T) is established at step 2 and is not affected by subsequent rebalancing operations. Therefore, T achieves finality from the users' perspective as soon as π_T is verified, regardless of later cross-intermediate or cross-shard operations. ■

This theorem ensures that users can rely on the finality of their transactions without waiting for global rebalancing or cross-shard settlement processes to complete.

Theorem 11 (Liquidity Accessibility)**.** *In the Overpass Channels system, a user's funds remain accessible for transactions and transfers regardless of the state of individual channel closures, provided the global network maintains sufficient liquidity.*

Proof. Consider a user Alice with funds in a channel C_A managed by Wallet Extension W_A under Intermediate Contract IC_A. Even if C_A is in the process of closing:

1. Alice can initiate a transaction to Bob, whose funds are in channel C_B under a different Intermediate Contract IC_B. 2. The transaction can be routed through: a) Intra-WEC rebalancing within W_A b) Inter-WEC rebalancing within IC_A c) Cross-Intermediate rebalancing between IC_A and IC_B d) Inter-WEC rebalancing within IC_B to reach W_B e) Intra-WEC rebalancing within W_B to reach C_B

3. Each step generates a zk-SNARK proof, ensuring the validity of the transfer without revealing the channel closure state.

Therefore, Alice's funds remain accessible and transferable throughout the network, independent of the state of C_A. ■

These theorems collectively demonstrate the robustness of the Overpass Channels system in maintaining both transaction finality and liquidity accessibility, which are crucial for seamless operation and user experience.

Efficient Channel Closure

Given the fluid nature of liquidity in the Overpass Channels system, channel closures can be handled efficiently without requiring immediate on-chain settlements. This approach minimizes on-chain operations while ensuring continuous fund accessibility for users.

Definition 8 (Lazy Channel Closure)**.** *A Lazy Channel Closure is a process where: 1. The channel is marked for closure in the off-chain state. 2. No new inbound transactions are accepted to the channel. 3. Existing funds in the channel remain accessible through network rebalancing. 4. The actual on-chain settlement is deferred until it's most efficient to do so.*

This lazy closure mechanism aligns with the overall off-chain focus of Overpass Channels, reducing the urgency of on-chain settlements and allowing for more efficient batching of closure operations.

Algorithm 7 Lazy Channel Closure

procedure LazyChannelClosure(C_i, W_j, IC_k)
    SetChannelState(C_i, "closing")
    $closure\_balance \leftarrow$ GetCurrentBalance(C_i)
    $closure\_proof \leftarrow$ GenerateClosureProof($C_i, closure\_balance$)
    AddToClosureQueue($C_i, closure\_balance, closure\_proof$)
    NotifyUserFundsAccessible(C_i)
end procedure
procedure BatchProcessClosures(RC)
    $closures \leftarrow$ GetQueuedClosures()
    $batch\_proof \leftarrow$ GenerateBatchClosureProof($closures$)
    SubmitBatchClosure($RC, closures, batch\_proof$)
end procedure

This approach offers several advantages: 1. Continuous Fund Accessibility: Users can still utilize their funds through network rebalancing, even if their original channel is marked for closure. 2. Reduced On-chain Overhead: Channel closures can be batched and processed when it's most gas-efficient to do so. 3. Simplified User Experience: Users don't need to wait for or monitor on-chain settlements; they can continue using the network seamlessly. 4. Optimized Cross-shard Operations: Closures involving cross-shard balances can be aggregated and settled efficiently without rushing individual transactions.

Cross-Shard Atomic Swaps

The cross-intermediate rebalancing process effectively enables cross-shard atomic swaps, leveraging TON's efficient cross-shard communication capabilities. This feature is crucial for maintaining liquidity across different shards and enabling seamless cross-shard transactions for users.

Theorem 12 (Cross-Shard Atomic Swap Correctness)**.** *The cross-intermediate rebalancing process guarantees atomicity and correctness of cross-shard swaps.*

Proof. Let $IC_a \in S_x$ and $IC_b \in S_y$ be two intermediate contracts in different shards participating in a cross-shard swap. The proof proceeds as follows:

1. Both IC_a and IC_b generate zk-SNARK proofs π_a and π_b of their respective balance adjustments ΔB_{IC_a} and ΔB_{IC_b}.

2. The Root Contract RC receives and verifies both proofs:

$$\text{Verify}(\pi_a) \wedge \text{Verify}(\pi_b) = \text{true} \tag{23}$$

3. *RC* ensures that the swap is balanced:

$$\Delta B_{IC_a} + \Delta B_{IC_b} = 0 \tag{24}$$

4. *RC* updates the global Merkle root to reflect the new balances of IC_a and IC_b.
5. The new global Merkle root is included in the next epoch submission to the TON blockchain.

If any step fails, the entire operation is reverted. Therefore, either both balance adjustments occur atomically, or neither does, ensuring the correctness of the cross-shard swap. ■

This theorem ensures that cross-shard operations in Overpass Channels maintain the same level of security and atomicity as single-shard operations, enabling seamless liquidity management across the entire network.

Integration with Payment Systems

The full potential of Overpass Channels is realized when payment systems are designed to integrate directly with the network. This integration allows for efficient, off-chain transactions that leverage the rebalancing capabilities of the system.

Definition 9 (Overpass-Integrated Payment System)**.** *An Overpass-Integrated Payment System is one that: 1. Recognizes Overpass Channel balances as valid payment sources. 2. Can initiate and receive payments through Overpass Channel rebalancing operations. 3. Defers to on-chain settlements only for large fund movements or when interacting with non-integrated systems.*

This integration is key to achieving high transaction throughput and minimizing on-chain operations in day-to-day transactions.

Theorem 13 (Off-chain Transaction Efficiency)**.** *In an ecosystem with Overpass-Integrated Payment Systems, the average transaction cost and time approach the efficiency of off-chain operations, regardless of the underlying on-chain settlement status of individual channels.*

Proof. Let T be a transaction in an Overpass-Integrated Payment System:

1. T is routed through Overpass Channels using rebalancing operations. 2. Each rebalancing step (intra-WEC, inter-WEC, cross-IC) occurs off-chain. 3. zk-SNARK proofs ensure the validity of each step without on-chain operations. 4. The transaction is considered complete once the recipient's balance is updated off-chain. 5. On-chain settlements occur in batches, amortizing the cost across many transactions.

Therefore, from the user's perspective, transaction cost and time are dominated by off-chain operations, approaching their efficiency limits as the number of transactions increases. ■

This theorem underscores the power of the Overpass Channels system when properly integrated with payment infrastructure. It demonstrates that users can enjoy the speed and cost-effectiveness of off-chain transactions while maintaining the security guarantees of the underlying blockchain.

Security and Privacy Analysis

The security and privacy of cross-intermediate rebalancing are ensured through the use of zk-SNARK proofs and the hierarchical structure of the system.

Theorem 14 (Privacy of Cross-Intermediate Rebalancing)**.** *The cross-intermediate rebalancing process preserves the privacy of individual intermediate contract balances from external observers and other intermediate contracts.*

Proof. Let $\mathcal{A}$ be an adversary observing the cross-intermediate rebalancing process. The information available to $\mathcal{A}$ consists of:

1. The global Merkle root before rebalancing: $root_{pre}$ 2. The global Merkle root after rebalancing: $root_{post}$ 3. The zk-SNARK proof π_{global} of the rebalancing operation

By the properties of cryptographic hash functions used in the Merkle tree, $\mathcal{A}$ cannot derive the individual IC balances from $root_{pre}$ or $root_{post}$. The zk-SNARK proof π_{global} only verifies that the rebalancing operation was performed correctly, without revealing any information about the IC balances or the rebalancing amounts.

Therefore, $\mathcal{A}$ gains no information about individual IC balances from observing the rebalancing process. ■

This theorem ensures that the privacy of individual users and intermediate contracts is maintained even during global rebalancing operations.

Efficiency Analysis

The efficiency of the system is crucial for its scalability across multiple shards. We analyze the computational and communication complexity of cross-intermediate rebalancing operations.

Theorem 15 (Complexity of Cross-Intermediate Rebalancing)**.** *The computational complexity of cross-intermediate rebalancing is $O(k \log k)$, and the communication complexity is $O(k + m)$, where k is the number of intermediate contracts and m is the number of shards.*

Proof. The proof follows from the analysis of Algorithm 6:

1. Solving the global optimization problem: $O(k \log k)$ using interior-point methods. 2. Generating and verifying zk-SNARK proofs for each IC: $O(k)$. 3. Updating the global Merkle tree: $O(k \log k)$. 4. Communication between Root Contract and ICs: $O(k)$. 5. Cross-shard communication for coordinating swaps: $O(m)$ in the worst case.

Therefore, the overall computational complexity is $O(k \log k)$, and the communication complexity is $O(k + m)$. ■

This theorem demonstrates that the cross-intermediate rebalancing process scales efficiently with the number of intermediate contracts and shards, making it suitable for large-scale deployments.

Integration with TON Blockchain

The Overpass Channels system leverages TON's sharding architecture and efficient cross-shard communication capabilities. The hypercube routing system of TON is particularly beneficial for cross-shard atomic swaps and global state updates.

Theorem 16 (Efficiency of Cross-Shard Operations)**.** *Cross-shard operations in Overpass Channels, when integrated with TON, have a communication complexity of $O(\log m)$, where m is the number of shards.*

Proof. TON's hypercube routing ensures that messages between any two shards can be delivered in $O(\log m)$ hops. In the worst case, our cross-shard operations might need to collect information from all shards. However, due to the hypercube topology:

1. Each shard can be reached in $O(\log m)$ hops. 2. Parallel message passing can be utilized for multiple shards. 3. The Root Contract can coordinate the process, reducing redundant communications.

Therefore, the overall communication complexity for cross-shard operations remains $O(\log m)$, regardless of the number of shards involved in the operation. ■

This integration ensures that Overpass Channels can maintain high efficiency even as the network scales across multiple shards.

Illustrative Example: Cross-Shard Rebalancing

To illustrate the cross-intermediate rebalancing process, consider the following example involving three intermediate contracts across two shards:

- IC_1 in Shard S_1 with balance $B_{IC_1} = 1000$ units
- IC_2 in Shard S_1 with balance $B_{IC_2} = 1500$ units
- IC_3 in Shard S_2 with balance $B_{IC_3} = 2500$ units
- Total network liquidity $L_{global} = 5000$ units
- Target ratios: $\psi_1 = 0.3, \psi_2 = 0.3, \psi_3 = 0.4$

The rebalancing process would proceed as follows:

1. Solve the optimization problem:

$$\Delta B_{IC_1} = 0.3 \times 5000 - 1000 = +500 \text{ units} \tag{25}$$
$$\Delta B_{IC_2} = 0.3 \times 5000 - 1500 = 0 \text{ units} \tag{26}$$
$$\Delta B_{IC_3} = 0.4 \times 5000 - 2500 = -500 \text{ units} \tag{27}$$

2. Generate zk-SNARK proofs for each adjustment.
3. Execute the cross-shard atomic swap between IC_1 and IC_3.
4. Update the global Merkle root to reflect the new balances.
5. Include the new global Merkle root in the next epoch submission to the TON blockchain.

This example demonstrates how the cross-intermediate rebalancing process can efficiently redistribute liquidity across shards while maintaining privacy and minimizing on-chain operations.

3 zk-SNARKs

The cornerstone of Overpass Channels' security and privacy features lies in its innovative use of zero-knowledge Succinct Non-interactive Arguments of Knowledge (zk-SNARKs). This section provides a detailed examination of how zk-SNARKs are integrated into the transaction validation process, ensuring both the validity of transactions and the privacy of users.

3.1 Overview

Before delving into the specifics of how Overpass Channels utilizes zk-SNARKs, it's important to understand the fundamental concepts behind this cryptographic technique.

Definition 10 (zk-SNARK)**.** *A zero-knowledge Succinct Non-interactive Argument of Knowledge (zk-SNARK) is a cryptographic protocol that allows one party (the prover) to prove to another party (the verifier) that a statement is true, without revealing any information beyond the validity of the statement itself.*

In the context of Overpass Channels, zk-SNARKs are used to prove the validity of transactions and state transitions without revealing the underlying transaction details.

3.2 zk-SNARK Circuit for Transaction Validation

The heart of the zk-SNARK integration in Overpass Channels is the circuit that defines the computation being proved. For transaction validation, this circuit encapsulates the logic of checking transaction validity, including balance updates and signature verification.

Algorithm 8 zk-SNARK Circuit for Transaction Validation

1: **procedure** TRANSACTIONVALIDATIONCIRCUIT($oldState, newState, tx, signature$)
2: AssertValidSignature($tx, signature$)
3: AssertSufficientBalance($oldState, tx.amount$)
4: AssertCorrectBalanceUpdate($oldState, newState, tx$)
5: AssertValidStateTransition($oldState, newState$)
6: AssertNonceIncrement($oldState.nonce, newState.nonce$)
7: **end procedure**

This circuit ensures that: 1. The transaction is properly signed by the sender. 2. The sender has sufficient balance to make the transaction. 3. The balances are correctly updated after the transaction. 4. The overall state transition is valid. 5. The nonce is correctly incremented.

3.3 Proof Generation and Verification

In Overpass Channels, transactions are signed based on the updated state at the conclusion of the previous transaction. This ensures the integrity and sequential consistency of the channel state. After each transaction, participants sign the new state, preventing any disputes about the validity or order of transactions.

When a user initiates a transaction, they generate a zk-SNARK proof attesting that the transaction is valid according to the circuit defined. However, before generating this proof, the transaction is signed according to the current state of the channel, which is the updated state following the previous transaction. This ensures that the state transition sequence is preserved and agreed upon by all parties.

The transaction signature generated in step 2, based on the private key and the updated state, ensures that the transaction reflects the most recent state of the channel. This mechanism helps enforce trust and prevents attempts to execute outdated or conflicting transactions.

Algorithm 9 Transaction Proof Generation and Verification

```
procedure GenerateTransactionProof(oldState, newState, tx, sk)
    signature ← Sign(tx, sk)                ▷ Transaction is signed based on the updated state
    witness ← (oldState, newState, tx, signature)
    proof ← ProveZKSNARK(TransactionValidationCircuit, witness)
    return proof
end procedure
procedure VerifyTransactionProof(proof, publicInputs)
    result ← VerifyZKSNARK(TransactionValidationCircuit, proof, publicInputs)
    return result
end procedure
```

3.4 Privacy Preservation

One of the key strengths of zk-SNARKs in Overpass Channels is the privacy it guarantees. The use of zk-SNARKs ensures that even though the transaction details are required to generate a proof, the actual transaction information (such as amounts, sender, and receiver) remains confidential, thanks to the zero-knowledge property. Only the updated state, which participants sign at the conclusion of each transaction, is reflected in the publicly shared information.

The proof guarantees that no one can infer additional details about the transaction beyond what is revealed by the public inputs, preserving confidentiality while maintaining verifiable state transitions.

3.5 Unilateral Channels: How They Work

In unilateral channels, transactions are initiated by one party, and the other party might not need to be involved actively in every transaction. However, both parties must still have agreed upon the updated state when a transaction occurs, ensuring that each party recognizes the new balances and the transition to the new state. Here's the process:

1. **Transaction Initiation**: One party (let's say Alice) initiates a transaction. For example, Alice wants to send Bob 10 tokens. When this transaction is initiated, Alice calculates the new state that will result from this payment (her balance reduced, Bob's balance increased).

2. **Signing on the New State**: Alice signs based on this updated state, which reflects the balances after the transaction is processed. Bob does not need to be online or sign off on this specific transaction at this moment because it's a unilateral channel—Alice can initiate and sign the transaction based on her agreement to the updated state.

3. **State Agreement Pre-Established**: While Bob doesn't sign at the moment of every transaction, both Alice and Bob would have agreed on how transactions work in this channel (including how balances are updated) when they set up the channel. This pre-establishment allows Alice to unilaterally process payments without Bob's real-time approval, as long as the rules of the channel (e.g., balance limits) are respected.

4. **Transaction Finality**: Once Alice signs the transaction reflecting the updated state, the transaction is executed. Bob can verify the transaction and updated state later, but he doesn't

need to be online for it to happen. The channel operates unilaterally, meaning Alice can push the updated state without needing Bob's immediate confirmation.

3.6 Instant and Asynchronous Nature

Instant Transactions: Transactions are effectively instant because Alice doesn't need to wait for Bob to be online or approve the transaction in real time. As soon as Alice signs and initiates the transaction, the new state is established, and the channel updates accordingly.

Asynchronous Execution: In this unilateral setup, transactions can happen even when one party (Bob) is offline. Alice can continue executing transactions as long as they follow the agreed rules for the channel, making the system highly asynchronous.

3.7 Online Requirements

For Alice: Alice needs to be online to initiate transactions and sign the updated state.

For Bob: Bob does not need to be online at the moment of every transaction. He only needs to periodically check or confirm the updated state when he comes online.

3.8 Practicality of zk-SNARKs

In this section, we present a comprehensive analysis of the computational costs associated with zk-SNARK proof generation and verification within the Overpass Channels system, focusing on practical implementation considerations. We examine the scalability of zk-SNARKs at scale, including potential optimizations to enhance efficiency. Specifically, we explore the utilization of PLONKY2, Poseidon hash functions, and the Goldilocks field, which are critical components in achieving practical performance for zero-knowledge proofs in our system.

Overview of PLONKY2, Poseidon, and Goldilocks Field

To understand the practicality of zk-SNARKs in Overpass Channels, it is essential to familiarize ourselves with the underlying cryptographic tools:

- **PLONKY2**: An efficient zk-SNARK proof system that builds upon the PLONK protocol. PLONKY2 is designed for high performance in both proof generation and verification, leveraging optimizations that reduce computational overhead.
- **Poseidon Hash Function**: A cryptographic hash function optimized for zero-knowledge proof systems. Poseidon is efficient within arithmetic circuits over finite fields, making it suitable for zk-SNARK implementations.
- **Goldilocks Field**: A prime field $\mathbb{F}_p$ where $p = 2^{64} - 2^{32} + 1$, chosen for its properties that enable efficient arithmetic operations on modern hardware architectures, particularly 64-bit processors.

Computational Costs of zk-SNARK Proof Generation

The computational cost of generating zk-SNARK proofs is a critical factor in the practicality of Overpass Channels. We analyze this cost by examining the steps involved in proof generation and their associated complexities.

Proof Generation Steps

For a prover (e.g., Alice) generating a zk-SNARK proof using PLONKY2, the following steps are involved:

1. **Circuit Definition**: Define an arithmetic circuit $\mathcal{C}$ that represents the computation to be proven (e.g., transaction validation logic).
2. **Constraint System Construction**: Translate $\mathcal{C}$ into a set of polynomial constraints over the Goldilocks field $\mathbb{F}_p$.
3. **Witness Computation**: Compute the witness $\mathbf{w}$, which includes secret inputs and any necessary intermediate values satisfying the constraints.
4. **Proof Generation Algorithm**: Execute the PLONKY2 proof generation algorithm to produce the proof π.

Complexity Analysis

The computational complexity of proof generation can be characterized as:

$$T_{\text{gen}} = O(n) \tag{28}$$

where n is the number of constraints in the circuit. The linear dependency indicates that as the circuit becomes more complex (i.e., as n increases), the time to generate the proof increases linearly.

Example: Alice's Transaction Proof Consider Alice initiating a transaction to send funds to Bob within an Overpass Channel. The transaction validation circuit $\mathcal{C}$ includes:

- Verification of Alice's digital signature.
- Ensuring Alice's balance is sufficient for the transaction amount.
- Correct updating of channel balances.
- Nonce incrementation to prevent replay attacks.

Suppose $\mathcal{C}$ comprises $n = 10^6$ constraints. The proof generation time T_{gen} for Alice is thus proportional to 10^6 constraint evaluations and associated cryptographic operations.

Optimizations in Proof Generation

To reduce T_{gen}, the following optimizations can be employed:

1. **Circuit Minimization**: Simplify $\mathcal{C}$ to reduce n by eliminating redundant computations and optimizing arithmetic operations.
2. **Efficient Cryptographic Primitives**: Use Poseidon hash functions within the circuit, as they are designed to be efficient in zk-SNARK contexts due to their low algebraic degree and efficient field operations.

3. **Parallel Processing**: Leverage multi-core processors or GPUs to perform parallel computations during proof generation.
4. **Recursive Proofs**: Utilize PLONKY2's support for recursion to aggregate multiple proofs into a single proof, reducing overall proof sizes and verification times.

3.9 Computational Costs of Proof Verification

Verification of zk-SNARK proofs must be efficient to ensure practicality, especially when verifying a large number of transactions.

Verification Steps

For a verifier (e.g., Bob) verifying a zk-SNARK proof π, the steps include:

1. **Public Input Preparation**: Gather the public inputs $\mathbf{x}$ required for verification (e.g., public keys, transaction details).
2. **Verification Algorithm Execution**: Use the PLONKY2 verification algorithm to check that π is a valid proof for the statement represented by $\mathcal{C}$ and $\mathbf{x}$.

Complexity Analysis

The verification time is given by:

$$T_{\text{ver}} = O(\log n) \tag{29}$$

This logarithmic dependency on the number of constraints n ensures that verification remains efficient even for large circuits.

Example: Bob Verifying Alice's Proof Bob receives the proof π from Alice and performs the verification:

1. Retrieves the necessary public inputs (e.g., Alice's public key, transaction amount).
2. Executes the PLONKY2 verification algorithm, which involves:
   - Checking polynomial commitments.
   - Verifying evaluation proofs.
   - Ensuring consistency with the public inputs.
3. If π is valid, Bob accepts the transaction and updates his local channel state.

3.10 Memory and Storage Considerations

Proof Size

The size of a PLONKY2 proof is relatively small, typically in the order of a few kilobytes, regardless of the size of the circuit. This succinctness is crucial for:

- Reducing network bandwidth usage during proof transmission.
- Minimizing storage requirements for archiving proofs.

Witness Size and Memory Usage

The prover must handle the witness $\mathbf{w}$, which can be large depending on n:

$$\text{Memory}_{\text{prover}} = O(n) \tag{30}$$

Optimizations to manage memory usage include:

- **Streaming Computations**: Process parts of the witness sequentially to avoid holding the entire witness in memory.
- **Memory-Efficient Data Structures**: Use data structures optimized for low memory overhead in representing field elements and constraints.

3.11 Bandwidth and Latency Analysis

Network Bandwidth

The transmission of proofs and associated data impacts network bandwidth:

- **Proof Transmission**: With proof sizes of a few kilobytes, the bandwidth required per transaction is minimal.
- **Batching Proofs**: Aggregating multiple proofs when transmitting to the same recipient can further reduce bandwidth overhead.

Latency Considerations

Proof generation time contributes to overall transaction latency:

- **User Experience**: For end-users like Alice and Bob, proof generation and verification should occur within timeframes that do not hinder usability (e.g., under a few seconds).
- **Optimizations**: Reducing T_{gen} and T_{ver} through optimizations directly improves latency.

3.12 Scalability at System Level

Aggregate Computational Load

For a system with M transactions per second (TPS), the total computational load is:

$$\text{Total } T_{\text{gen}} = M \times T_{\text{gen}} \tag{31}$$

$$\text{Total } T_{\text{ver}} = M \times T_{\text{ver}} \tag{32}$$

Parallelization Across Users

Since proof generation and verification are performed independently by users, the system naturally supports horizontal scaling:

- **No Central Bottlenecks**: There is no central entity performing all proof computations.
- **Distributed Computation**: Each user contributes their own computational resources.

3.13 Comparisons with Alternative Approaches

Alternative Zero-Knowledge Proof Systems

Other zk-SNARK protocols include:

- **Groth16**: Offers constant verification time and small proofs but requires a trusted setup.
- **Bulletproofs**: No trusted setup, but proof size and verification time are linear in n.

Rationale for Choosing PLONKY2

PLONKY2 is selected for Overpass Channels due to:

- **Efficient Proof Generation**: Suitable for devices with limited computational power.
- **Recursive Proofs Support**: Facilitates advanced features like proof aggregation.
- **Transparent Setup**: Avoids the need for a trusted setup, enhancing security.
- **Optimized for Goldilocks Field**: Takes advantage of efficient arithmetic operations.

3.14 Potential Bottlenecks and Mitigations

User Device Limitations

Challenge Users with older or less powerful devices may experience slower proof generation times.

Mitigation Strategies

- **Lightweight Clients**: Offer simplified clients that offload heavy computations to more capable devices or cloud services.
- **Hardware Acceleration**: Encourage the use of devices with hardware support for cryptographic operations.

Network Constraints

Challenge In regions with limited network infrastructure, bandwidth and latency can affect transaction processing.

Mitigation Strategies

- **Proof Compression**: Further compress proofs without sacrificing security.
- **Offline Transactions**: Allow users to queue transactions and proofs for later transmission when network conditions improve.

3.15 Security and Trust Considerations

Soundness and Zero-Knowledge

PLONKY2 ensures:

- **Soundness**: Invalid statements cannot produce valid proofs.
- **Zero-Knowledge**: No information about private inputs is revealed through the proof.

Cryptographic Assumptions

The security relies on:

- **Hardness of Discrete Logarithm Problem**: In the context of the elliptic curve used.
- **Collision Resistance of Hash Functions**: Poseidon must resist collision and pre-image attacks.

4 Balance Consistency

Maintaining consistent and accurate balances across all channels is crucial for the integrity and reliability of the Overpass Channels network. This section delves into the mathematical formalism and proofs that guarantee balance consistency throughout the system.

4.1 Formal Definition of Balance Consistency

Before we proceed with the theorem and proof, let's formally define what we mean by balance consistency in the context of Overpass Channels.

Definition 11 (Balance Consistency)**.** *A payment channel network exhibits balance consistency if and only if, for any valid sequence of transactions, the following conditions hold:*

1. *The sum of all balances across all channels remains constant (excluding external deposits and withdrawals).*
2. *For each channel, the sum of the balances of all participants in that channel remains equal to the channel's capacity.*
3. *No participant's balance in any channel ever becomes negative.*

4.2 Theorem of Balance Consistency

Now, we can state and prove the fundamental theorem that guarantees balance consistency in Overpass Channels.

Theorem 17 (Balance Consistency in Overpass Channels)**.** *In the Overpass Channels network, all valid transactions and state transitions preserve balance consistency as defined above.*

Proof. We will prove this theorem by induction on the number of transactions in the network.

Base case: At the network's initialization, all channels are created with a fixed capacity, and the initial balances sum to this capacity. Therefore, the balance consistency property holds initially.

Inductive step: Assume that the network is in a consistent state after n transactions. We need to prove that any valid $(n+1)$-th transaction will maintain balance consistency.

Let T be the $(n+1)$-th transaction, occurring in channel C between participants A and B. Without loss of generality, assume A is sending x tokens to B.

1. By the definition of a valid transaction in Overpass Channels, T must be accompanied by a valid zk-SNARK proof P.
2. The zk-SNARK circuit for transaction validation ensures:
   (a) A's balance in C is sufficient: $balance_A \geq x$
   (b) The new balances are correctly computed:
   $$newBalance_A = balance_A - x$$
   $$newBalance_B = balance_B + x$$
3. The zk-SNARK proof P is verified by B and, upon settlement, by the network.
4. After T is applied:
   (a) The sum of balances in C remains unchanged:
   $$(balance_A - x) + (balance_B + x) = balance_A + balance_B$$
   (b) No other channel's balances are affected.
   (c) A's new balance is non-negative (from 2a and 2b).
   (d) B's new balance is clearly non-negative as it only increases.
5. Therefore, all three conditions of balance consistency continue to hold after T:
   (a) The sum of all balances across all channels remains constant.
   (b) The sum of balances in C equals its capacity (from 4a).
   (c) No participant's balance becomes negative (from 4c and 4d).

By the principle of mathematical induction, balance consistency holds for any number of valid transactions in the network. ■

4.3 Implications and Practical Considerations

The Balance Consistency Theorem has several important implications for the Overpass Channels network:

1. **Security Against Double Spending**: The theorem guarantees that it's impossible for a participant to spend more tokens than they possess, effectively preventing double spending without requiring global consensus.

2. **Local Verification Sufficiency**: Because balance consistency is maintained for each valid transaction, participants only need to verify the zk-SNARK proof of the latest transaction to be assured of the channel's integrity.

3. **Simplified Conflict Resolution**: In case of disputes, the latest valid state (proven by zk-SNARKs) can be used to resolve conflicts without needing to replay the entire transaction history.

4. **Efficient State Updates**: The theorem allows for efficient updates of channel states without requiring updates to the global network state for every transaction.

5 Mitigating(MEV) in Overpass Channels

Miner Extractable Value (MEV) refers to the profit miners can extract from users by manipulating the ordering of transactions within a block. This has become a significant issue in public blockchain networks, particularly Ethereum, where it can lead to unfair advantages and financial losses for regular users. Overpass Channels offers several key benefits in mitigating MEV:

5.1 Privacy as a Shield Against MEV

The primary defense against MEV in Overpass Channels stems from its privacy-preserving architecture. By keeping transaction details confidential, the system dramatically reduces the attack surface for MEV exploitation.

Theorem 18 (MEV Resistance through Privacy)**.** *Let $T = \{t_1, t_2, ..., t_n\}$ be a set of transactions in Overpass Channels, and let $I(t_i)$ be the information available to a potential MEV extractor about transaction t_i. Then:*

$$\forall t_i \in T, I(t_i) = \{\textit{existence of } t_i\}$$

That is, an MEV extractor can only know of the existence of a transaction, but not its contents, sender, receiver, or value.

Proof. 1) All transactions in Overpass Channels are processed off-chain within individual payment channels.

2) The network nodes only receive:

- Merkle tree roots representing batches of transactions (epochs)
- zk-SNARK proofs verifying the validity of state transitions

3) zk-SNARK proofs, by definition, reveal no information about the transactions beyond their validity.

4) Therefore, no entity outside the direct participants of a transaction can access its details. ■

This theorem demonstrates that MEV extractors are effectively "blind" to the specifics of any transaction, severely limiting their ability to extract value.

Algorithm 10 Transaction Processing in Overpass Channels

```
procedure ProcessTransaction(sender, recipient, amount)
    oldState ← GetChannelState(sender, recipient)
    newState ← ComputeNewState(oldState, amount)
    proof ← GenerateZKProof(oldState, newState, amount)
    UpdateChannelState(sender, recipient, newState, proof)
end procedure
```

5.2 Front-Running Prevention

Front-running, a common form of MEV, becomes virtually impossible in Overpass Channels due to its privacy features and transaction processing mechanism.

In this process, transactions are settled immediately within the channel, leaving no opportunity for front-running. Even if an attacker could somehow detect a pending transaction, they would be unable to insert their own transaction ahead of it due to the channel's sequential and private processing.

5.3 Quantitative Analysis of MEV Reduction

To quantify the MEV reduction in Overpass Channels compared to a public blockchain like Ethereum, we can use a simplified model:

Let p be the probability of a transaction being exploited for MEV on a public blockchain, and v be the average value extracted per exploited transaction. The expected MEV $E(MEV)$ for n transactions is:

$$E(MEV)_{public} = n \cdot p \cdot v$$

In Overpass Channels, due to transaction privacy, $p \approx 0$. Therefore:

$$E(MEV)_{Overpass} \approx 0$$

This demonstrates a near-complete elimination of MEV in Overpass Channels.

5.4 Comparative Analysis: Overpass Channels vs. Other Solutions

Feature	**Overpass**	**Ethereum**	**Other L2s**
Transaction Privacy	Full	None	Partial
Front-running Prevention	Strong	Weak	Moderate
MEV Resistance	High	Low	Medium

Table 1: Comparison of MEV Resistance Across Different Systems

While other Layer 2 solutions offer some protection against MEV, Overpass Channels provides superior resistance due to its comprehensive privacy features and off-chain processing.

5.5 Example Scenario: Alice and Bob's DEX Interaction

Consider a decentralized exchange (DEX) scenario where Alice wants to swap 100 ETH for USDC:

1. **In a public blockchain:**
    - Alice submits a transaction to swap 100 ETH for USDC.
    - Bob, a MEV bot operator, sees Alice's pending transaction.
    - Bob front-runs Alice, causing her to receive less USDC than expected.
2. **In Overpass Channels:**
    - Alice initiates a swap in her payment channel with the DEX.
    - The swap details are kept private, known only to Alice and the DEX.
    - The transaction is processed immediately within the channel.
    - Bob has no opportunity to front-run or extract value from Alice's transaction.

This example illustrates how Overpass Channels protects users like Alice from MEV exploitation that is common in public blockchain networks.

5.6 Implementing Beneficial Arbitrage within Overpass Channels

While Overpass Channels' privacy features significantly reduce harmful MEV, the system can be designed to support beneficial arbitrage activities that enhance market efficiency without exposing sensitive transaction details.

- **Confidential Order Matching:** Implement zk-SNARK-based order matching protocols within payment channels. This allows arbitrageurs to execute trades based on aggregated market data without accessing individual order details. The zk-SNARK proofs can verify that trades are executed fairly and according to predefined market rules.
- **Aggregated Price Feeds:** Utilize oracles to provide aggregated and anonymized price feeds to payment channels. Arbitrageurs can use these feeds to identify and exploit price discrepancies across different channels or exchanges without gaining access to the underlying transaction data.
- **Incentive Structures for Arbitrageurs:** Design incentive mechanisms within Overpass Channels that reward arbitrageurs for maintaining market efficiency. These incentives can be encoded within the zk-SNARK proofs, ensuring that only legitimate arbitrage activities are rewarded without revealing the specifics of each arbitrage transaction.
- **Decoupled Execution and Settlement:** Separate the execution of arbitrage transactions from their settlement. Execution can occur within private channels using zk-SNARKs, while settlement aggregates the results on-chain in a privacy-preserving manner. This approach ensures that arbitrage activities contribute to market efficiency without introducing MEV vulnerabilities.

By carefully designing these mechanisms, Overpass Channels can harness the benefits of MEV in improving market efficiency while maintaining robust defenses against exploitative MEV practices.

6 Fraud Prevention Mechanisms in Overpass Channels

Ensuring the integrity of the system and preventing fraudulent activities are paramount in any financial network. Overpass Channels incorporates several sophisticated mechanisms to prevent fraud, leveraging its unique architecture and cryptographic foundations. This section provides a detailed examination of these fraud prevention mechanisms.

6.1 Key Safeguard: Pending Transaction Acceptance

To mitigate the potential security concern where Alice could exploit Bob's absence (as mentioned earlier), Overpass Channels employ a key safeguard: **pending transaction acceptance**.

1. **Pending Transaction Mechanism**: Before Alice can send a new transaction to Bob, Bob must first accept the pending one. This means that if Alice sends a token to Bob, the transaction remains pending until Bob verifies it. Only after Bob accepts the transaction does Alice's balance update, and the system allows her to initiate the next transaction.
2. **Mitigation of Fraud**: This mechanism mitigates the concern of Alice manipulating the channel when Bob is offline. Even if Bob remains offline for a while, no new transactions can proceed until Bob confirms the previous one, preventing Alice from spamming the channel with fraudulent or excessive transactions.
3. **Transaction Finality**: Once Bob accepts the pending transaction, the state is updated, and the next transaction can be executed. This ensures that each transaction is properly verified and final, preventing any ambiguity or risk of double-spending.
4. **Transaction Ordering**: In addition to pending transaction acceptance, Overpass Channels uses a combination of per-channel nonces and SEQNO to ensure strict ordering of transactions. This further prevents replay attacks and ensures state consistency.

6.2 Mitigation of Security Concerns for Extended Absence

To further mitigate the risk where Alice could try to manipulate the channel while Bob is offline for an extended period (as raised earlier in the downsides), dynamic rebalancing and pending transaction acceptance work in tandem to provide additional safeguards:

- **Pending Transactions**: Alice cannot proceed with new transactions unless Bob has accepted the previous ones. This prevents Alice from submitting multiple fraudulent or excessive transactions while Bob is offline.
- **Dynamic Rebalancing by Smart Contracts**: The wallet extension contract ensures that all channels are balanced and no channel exceeds its authorized limits. The 50% rule also caps the amount Alice can send in one transaction, limiting potential damage.

Together, these mechanisms ensure that Bob is protected from fraud or abuse while offline, allowing him to review and accept transactions upon returning online, while at the same time not leaving Alice in a position where her funds are unusable during the wait. The system's dynamic rebalancing enables Alice to continue transacting in other channels or reallocating resources as needed, ensuring fluid channel operations across the board.

6.3 Deterministic Conflict Resolution

In the event of a conflict, such as when parties disagree on the current state of a channel, Overpass Channels employs a deterministic conflict resolution mechanism.

Theorem 19 (Deterministic Conflict Resolution)**.** *Given any two conflicting channel states S_1 and S_2 submitted by different parties, the Overpass Channels protocol will deterministically select a single valid state.*

Proof. Let S_1 and S_2 be two conflicting states for channel C, submitted by parties A and B respectively. The proof proceeds as follows:

1. Both S_1 and S_2 must be accompanied by valid zk-SNARK proofs P_1 and P_2.
2. The on-chain contract will verify both P_1 and P_2.
3. If either proof fails verification, the corresponding state is rejected.
4. If both proofs are valid, the contract compares the nonces n_1 and n_2 of S_1 and S_2.
5. The state with the higher nonce is selected as the valid state.
6. In the unlikely event that $n_1 = n_2$, the contract applies a deterministic tie-breaking rule (e.g., selecting the state with the lexicographically smaller hash).

Therefore, given any two conflicting states, the protocol will always select a single valid state in a deterministic manner. ■

6.4 50% Spending Rule for Off-Chain Transactions

To prevent potential griefing attacks where a malicious party could repeatedly force channel closures, Overpass Channels implements a 50% spending rule for off-chain transactions.

Definition 12 (50% Spending Rule)**.** *In any single off-chain transaction within a channel, a party cannot spend more than 50% of their current channel balance.*

Theorem 20 (Griefing Prevention)**.** *The 50% spending rule in Overpass Channels prevents a malicious party from depleting their entire channel balance in a single transaction, ensuring that honest parties always have recourse to close the channel profitably.*

Proof. Let C be a channel between Alice and Bob, with Alice's balance B_A and Bob's balance B_B. The proof proceeds as follows:

1. Suppose Alice attempts to make a malicious transaction T to deplete her balance.
2. By the 50% spending rule, the maximum amount Alice can send in T is $\frac{B_A}{2}$.
3. After T, Alice's new balance B'_A is at least $\frac{B_A}{2}$.
4. If Bob detects malicious behavior, he can initiate a channel closure.
5. In the worst case (if Alice doesn't cooperate), Bob must close the channel on-chain.

6. The on-chain closing cost is C_{close}, which is less than $\frac{B_A}{2}$ by design.

7. Therefore, even after paying C_{close}, Bob is guaranteed to receive a positive balance from the channel closure.

Thus, the 50% spending rule ensures that honest parties always have a profitable recourse to close the channel, preventing griefing attacks. ■

This 50% spending rule, combined with the deterministic conflict resolution mechanism, provides strong guarantees against various forms of malicious behavior in Overpass Channels.

6.5 ZK-SNARK Proofs and State Updates

At the core of Overpass Channels' fraud prevention is the use of zk-SNARKs for validating state updates.

Definition 13 (Valid State Update)**.** *A state update in Overpass Channels is considered valid if and only if it is accompanied by a zk-SNARK proof that verifies:*

1. *The update transitions from a valid previous state to a valid new state.*
2. *The update follows all rules of the channel (e.g., balance changes, nonce increments).*
3. *The update is authorized by the appropriate parties.*

Proof of Validity

Let's formalize the proof of validity for state updates:

Theorem 21 (State Update Validity)**.** *Any state update in Overpass Channels that is accepted by the network is guaranteed to be valid according to the channel rules.*

Proof. Let S_0 be the initial state of a channel, and S_1 be the state after an update. The proof proceeds as follows:

1. For the update $S_0 \to S_1$ to be accepted, it must be accompanied by a zk-SNARK proof P.
2. P is generated using a circuit that encodes all channel rules, including:
   (a) Balance consistency (as proved in the previous section)
   (b) Proper nonce incrementing
   (c) Signature verification
3. The verification of P is performed by all relevant parties (the recipient in a transaction, and the network nodes during settlement).
4. By the soundness property of zk-SNARKs, if P verifies successfully, then with overwhelming probability, the prover must know a valid witness satisfying all constraints in the circuit.
5. Therefore, if P is accepted, S_1 must be a valid state reachable from S_0 according to all channel rules.

Thus, any accepted state update is guaranteed to be valid. ■

6.6 Cross-Shard Transaction Security

Overpass Channels leverages efficient cross-shard communication capabilities to ensure that cross-shard transactions maintain the same security guarantees as intra-shard transactions.

Theorem 22 (Cross-Shard Security)**.** *Cross-shard transactions in Overpass Channels maintain the same security guarantees as intra-shard transactions.*

Proof. Let $T_{A,B}$ be a transaction from channel A in shard S_A to channel B in shard S_B. The security of $T_{A,B}$ is ensured by:

1. A PLONKY2 proof P_A verifying the validity of the transaction in S_A
2. A PLONKY2 proof P_B verifying the validity of the state update in S_B
3. routing ensuring reliable message delivery between shards

The combination of these elements ensures that cross-shard transactions have the same security properties as intra-shard transactions. ■

This cross-shard security mechanism allows Overpass Channels to maintain consistent security guarantees across the entire network, regardless of the sharding structure of the underlying blockchain.

6.7 Prevention of Old State Submission

One potential attack vector in channel-based systems is the submission of old, outdated states. Overpass Channels prevents this through a combination of nonce usage, zk-SNARK proofs, and smart contract capabilities.

Definition 14 (Nonce and Seqno)**.** *In Overpass Channels:*

- *A nonce is a monotonically increasing integer associated with each channel state, incremented with each valid state update.*
- *Each channel contract stemming from the wallet contract is assigned a seqno, a capability provided by smart contract system.*

Theorem 23 (Old State Invalidation)**.** *In Overpass Channels, it is computationally infeasible to submit an old state as a valid current state.*

Proof. The proof proceeds by contradiction:

1. Assume an adversary can submit an old state S_{old} with nonce n_{old} and seqno seq_{old} as a valid current state.
2. Let the actual current state be $S_{current}$ with nonce $n_{current}$ and seqno $seq_{current}$, where $n_{current} > n_{old}$ and $seq_{current} > seq_{old}$.
3. For S_{old} to be accepted, the adversary must provide a valid zk-SNARK proof P_{old}.
4. The zk-SNARK circuit includes checks that:

(a) The nonce in the new state is greater than the nonce in the old state.
(b) The seqno in the new state matches the current seqno of the channel contract.

5. Therefore, for P_{old} to verify, the adversary must know a witness w such that:
   (a) w satisfies all constraints of the zk-SNARK circuit
   (b) w includes a valid signature for S_{old}
   (c) w demonstrates that $n_{old} > n_{current}$
   (d) w demonstrates that $seq_{old} = seq_{current}$
6. However, (5c) contradicts the fact that $n_{current} > n_{old}$, and (5d) contradicts $seq_{current} > seq_{old}$.
7. By the soundness property of zk-SNARKs and the monotonically increasing nature of the seqno, finding such a witness w is computationally infeasible.

Therefore, submitting an old state as a valid current state is computationally infeasible in Overpass Channels, reinforced by both the internal nonce mechanism and seqno capability. ■

It's worth noting that Overpass Channels' use of PLONKY2 for generating zk-SNARK proofs provides additional security against old state submission attempts, as PLONKY2 offers strong soundness guarantees without requiring a trusted setup.

6.8 Proof Consistency

Overpass Channels ensures that all proofs within the system are consistent with each other, preventing conflicting updates.

Theorem 24 (Proof Consistency)**.** *In Overpass Channels, it is computationally infeasible to generate two valid, conflicting proofs for the same channel.*

Proof. Let S_0 be an initial channel state. Suppose an adversary attempts to generate two conflicting proofs, P_1 and P_2, leading to different final states S_1 and S_2. The proof proceeds as follows:

1. For P_1 and P_2 to be valid, they must both prove transitions from S_0 to their respective final states.
2. The zk-SNARK circuit includes the hash of the initial state as a public input.
3. Therefore, P_1 and P_2 must use the same initial state hash.
4. The circuit also enforces that the nonce in the final state is exactly one more than the nonce in the initial state.
5. Thus, S_1 and S_2 must have the same nonce.
6. The circuit enforces that for a given initial state and nonce, there is only one valid final state (determined by the transaction details).
7. Therefore, for P_1 and P_2 to be simultaneously valid, S_1 and S_2 must be identical.
8. This contradicts the assumption that P_1 and P_2 lead to different final states.

Thus, it is computationally infeasible to generate two valid, conflicting proofs for the same channel. ■

6.9 On-Chain Verification

While most operations in Overpass Channels occur off-chain, the system includes on-chain verification as a final layer of fraud prevention.

Algorithm 11 On-Chain Verification of Channel Closure

```
procedure VerifyChannelClosure(channelID, finalState, proof)
    storedState ← GetStoredChannelState(channelID)
    isValid ← VerifyZKSNARK(proof, finalState, storedState)
    if isValid then
        UpdateChannelState(channelID, finalState)
        DistributeFunds(channelID, finalState) return TRUE
    else
        RejectClosure(channelID) return FALSE
    end if
end procedure
```

This on-chain verification serves as a crucial backstop against potential fraudulent activities, ensuring that even if off-chain mechanisms fail, the on-chain state remains secure.

6.10 Elimination of Need for Watchtowers

Traditional payment channel networks often rely on watchtowers to monitor for fraudulent channel closures. Overpass Channels have no need for such external monitoring through its use of zk-SNARKs and on-chain verification.

Theorem 25 (Watchtower Redundancy)**.** *In Overpass Channels, participants can securely close channels without relying on external watchtowers.*

Proof. The proof proceeds by showing that any attempt at fraudulent channel closure will be detected and prevented:

1. Let C be a channel between Alice and Bob, with current state $S_{current}$.
2. Suppose Alice attempts to fraudulently close C with an old state S_{old}.
3. To close the channel, Alice must submit to the on-chain contract:
   (a) The final state S_{old}
   (b) A zk-SNARK proof P that S_{old} is a valid state
4. The on-chain verification algorithm will:
   (a) Verify the zk-SNARK proof P
   (b) Check that the nonce in S_{old} is greater than the last known on-chain nonce
5. For the fraudulent closure to succeed, Alice must generate a valid proof P for S_{old} with a nonce higher than $S_{current}$.

6. However, by the proof of the Old State Invalidation theorem, generating such a proof is computationally infeasible.

7. Therefore, Alice's fraudulent closure attempt will be rejected by the on-chain contract.

8. Bob can close the channel with $S_{current}$ at any time, without needing to constantly monitor the blockchain.

Thus, participants can securely close channels without relying on external watchtowers. ■

6.11 Cryptographic Proofs and Tamper-Evident Records

Overpass Channels maintains a cryptographically secure, tamper-evident record of all wallet states, including their associated channel states and transactions. This is achieved through a combination of sparse Merkle trees and zk-SNARKs.

Definition 15 (Wallet State Sparse Merkle Tree)**.** *For each wallet contract W, a sparse Merkle tree T_W is maintained where:*

- *Each leaf represents a channel state.*
- *The leaf value is the hash of the state: $leaf_i = H(S_i)$.*
- *The leaf's position in the tree is determined by the channel's SEQNO.*
- *The root of T_W is included in each zk-SNARK proof.*

Theorem 26 (Tamper-Evident Wallet and Channel History)**.** *Any tampering with the history of wallet or channel states in Overpass Channels is detectable with overwhelming probability.*

Proof. Let $\{S_0, S_1, \ldots, S_n\}$ be the sequence of states for wallet W and its associated channels. The proof proceeds as follows:

1. Each state transition $S_i \to S_{i+1}$ (whether a wallet state change or a channel state change) is accompanied by a zk-SNARK proof P_i.

2. P_i includes:

   (a) The Merkle root R_i of T_W before the transition

   (b) The Merkle root R_{i+1} of T_W after the transition

   (c) A proof that S_{i+1} is a valid successor state to S_i

3. The zk-SNARK circuit verifies:

   (a) R_i is the correct Merkle root for T_W including all states up to S_i

   (b) R_{i+1} is the correct Merkle root for T_W after updating S_{i+1}

4. By the collision resistance property of the hash function H, finding two different states that produce the same leaf value is computationally infeasible.

5. By the security properties of sparse Merkle trees, modifying any state in the history would change the Merkle root.

6. Any change to a historical state S_j would invalidate all subsequent proofs P_k for $k \geq j$, as they would no longer have valid Merkle root transitions.

7. Generating new valid proofs for the modified history would require breaking the soundness of the zk-SNARK system, which is assumed to be computationally infeasible.

Therefore, any tampering with the wallet or channel history is detectable with overwhelming probability. ■

This tamper-evident property ensures that the entire history of a wallet and its associated channels can be cryptographically verified, providing strong guarantees against historical fraud or manipulation.

7 Transaction Processing and Conflict Resolution

Efficient transaction processing and robust conflict resolution mechanisms are crucial for the smooth operation of Overpass Channels. This section delves into the details of how transactions are processed and how potential conflicts are resolved in a deterministic and fair manner.

7.1 Transaction Processing

In Overpass Channels, transactions are processed off-chain within payment channels, with periodic settlements on the blockchain. Let's formalize the transaction processing mechanism:

Algorithm 12 Transaction Processing in Overpass Channels

```
procedure ProcessTransaction(sender, recipient, amount, channelID)
    channel ← GetChannelState(channelID)
    oldState ← channel.currentState
    newState ← ComputeNewState(oldState, sender, recipient, amount)
    proof ← GenerateZKProof(oldState, newState, sender, amount)
    signatureSender ← Sign(newState, sender.privateKey)
    signatureRecipient ← GetRecipientSignature(newState, recipient)
    if VerifyZKProof(proof) ∧ VerifySignatures(signatureSender, signatureRecipient) then
        channel.currentState ← newState
        UpdateOffChainState(channelID, newState)
        return SUCCESS
    else
        return FAILURE
    end if
end procedure
```

This algorithm ensures that each transaction is validated cryptographically before being applied to the channel state.

7.2 Assignment to Wallet Contracts

Each user in Overpass Channels is associated with a wallet contract on the blockchain. These wallet contracts play a crucial role in managing channels and resolving conflicts.

Definition 16 (Wallet Contract)**.** *A wallet contract in Overpass Channels is a smart contract on the blockchain that:*

- *Holds the user's on-chain balance*
- *Manages the user's participation in payment channels*
- *Handles channel opening, closing, and dispute resolution*

8 Conditional Payments

8.1 Conditional Transactions using zk-SNARKs

Conditional transactions allow participants in Overpass Channels to transfer funds only when predefined conditions are met. In this section, we present an approach to implement conditional transactions using zk-SNARKs, enabling privacy-preserving, off-chain transaction processing.

Overview of zk-SNARK Conditional Transactions

In Overpass Channels, zk-SNARKs provide an efficient and privacy-preserving mechanism for conditional payments. Conditional transactions can be triggered by various events, such as a task completion, multi-signature approval, or external oracle data. These conditions are encoded in a zk-SNARK proof, ensuring that conditions are met without revealing sensitive information about the transaction.

Conditional Transaction Workflow

To illustrate the workflow of a conditional transaction using zk-SNARKs, consider the following steps:

1. **Condition Setup**: Alice wants to send Bob 50 tokens, but only if Bob completes a task (e.g., proving ownership of a document or a digital asset). The condition is encoded as a cryptographic task that Bob must prove using zk-SNARKs.
2. **Proof Generation**: Bob generates a zk-SNARK proof, denoted as P, which proves that the condition has been satisfied (e.g., task completion) without revealing any sensitive details.
3. **Verification by Alice**: Alice receives the zk-SNARK proof P from Bob. She verifies P using the public verification key vk associated with the zk-SNARK circuit. If the proof is valid, Alice releases the funds.
4. **Off-Chain Execution**: Since the transaction is off-chain, the state of the channel is updated to reflect the new balances once the proof is verified. The zk-SNARK ensures that no sensitive information, such as the task details or the exact condition, is revealed.

The zk-SNARK proof ensures that the conditional logic is satisfied without requiring interaction with the blockchain until settlement.

zk-SNARK Circuit for Conditional Transactions

The zk-SNARK circuit for conditional transactions can be described as follows:

```
Algorithm 1: zk-SNARK Conditional Transaction Circuit
1. procedure ConditionalTxCircuit(cond, tx, sig, balance)
2.     AssertConditionMet(cond)   // Verify that the condition is met
3.     AssertValidSignature(tx, sig)   // Verify transaction signature
4.     AssertSufficientBalance(balance, tx.amount)   // Check balance
5.     AssertCorrectStateUpdate(balance)   // Update state correctly
6. end procedure
```

This circuit allows us to verify that the condition (e.g., task completion or oracle verification) is satisfied before releasing funds, without revealing the actual details of the condition.

Use Cases for zk-SNARK Conditional Transactions

- **Task-Driven Payments**: Alice can set up a payment to Bob that is contingent upon Bob completing a certain task or milestone. Bob generates a zk-SNARK proof demonstrating that the task has been completed, allowing Alice to release the funds.
- **Event-Based Transactions**: Using oracles, a conditional payment can be triggered by an external event (e.g., price movement of an asset, delivery confirmation) with a zk-SNARK proof generated from oracle data.
- **Multi-Signature Agreements**: Multiple parties can approve a transaction by providing zk-SNARK proofs that their conditions for approval have been met.

8.2 Hash Time-Locked Contracts (HTLC) in Overpass Channels

In addition to zk-SNARKs, Overpass Channels can implement Hash Time-Locked Contracts (HTLCs) to facilitate simpler conditional payments. HTLCs allow payments to be contingent on both a cryptographic condition (hash preimage) and a time constraint. This approach is particularly useful for cross-channel payments and time-sensitive transactions.

Overview of HTLC

HTLCs are widely used in off-chain systems to enable conditional payments based on the revelation of a secret (hash preimage). In Overpass Channels, HTLCs can be implemented to provide conditional payments with automatic refund mechanisms based on timeouts.

HTLC Workflow

The process of creating an HTLC-based transaction is as follows:

1. **Alice Initiates HTLC**: Alice wants to send Bob 50 tokens if Bob can provide the preimage X of a hash $H(X)$. Alice creates an HTLC in the payment channel and locks the funds under the condition that Bob must provide X within a certain time T.

2. **Bob Reveals Preimage**: If Bob can reveal the correct preimage X within the time T, he provides it to Alice. Alice verifies that $H(X)$ matches the hash condition.

3. **Payment Finalization**: Upon verifying the preimage X, Alice releases the 50 tokens to Bob, completing the conditional transaction.

4. **Timeout Mechanism**: If Bob fails to provide the preimage X before time T expires, the funds are returned to Alice automatically.

Algorithm for HTLC in Overpass Channels

The algorithm for setting up an HTLC in Overpass Channels is as follows:

```
Algorithm 2: HTLC Setup and Execution
1. procedure HTLCSetup(sender, receiver, amount, H(X), timeout)
2.     Lock(amount) under condition: receiver must provide preimage X
3.     Wait for receiver to reveal X
4.     if H(X) is valid and X is correct:
5.         Transfer funds to receiver
6.     else if timeout has expired:
7.         Refund funds to sender
8.     end if
end procedure
```

Use Cases for HTLC in Overpass Channels

- **Cross-Channel Payments**: HTLCs allow Alice and Bob to securely make payments across channels without trusting each other. If the condition is met, Bob can claim the payment.

- **Refundable Payments**: In scenarios where time-sensitive payments are required (e.g., online escrow), HTLC ensures that funds are returned to the sender if conditions are not met within a defined period.

- **Atomic Swaps**: HTLCs can facilitate atomic swaps between different blockchains or assets, where each party provides a hash preimage to claim the assets.

Comparison of HTLC and zk-SNARK Conditional Transactions

HTLCs are a simple and effective mechanism for conditional payments where the conditions are based on hash preimage and time. However, for more complex conditional logic that requires privacy-preserving proofs, zk-SNARKs provide a more flexible solution.

9 Hierarchical Ordering and System-Level Efficiency

Overpass Channels employs a hierarchical structure to achieve system-level efficiency and scalability. This section explores the different levels of this hierarchy and how they contribute to the overall performance of the network.

Feature	zk-SNARKs	HTLCs
Privacy	High, condition details are hidden	Low, condition details are exposed
Flexibility	Supports complex conditions	Limited to hash and time-based conditions
Complexity	High, requires cryptographic proof generation	Low, simple cryptographic check
Use Cases	Multi-sig, oracle-based, event-driven conditions	Time-sensitive or hash preimage transactions

Table 2: Comparison between zk-SNARKs and HTLCs for Conditional Payments

9.1 Hierarchical Structure

The Overpass Channels network is organized into four main levels:

1. Payment Channel Contracts
2. Wallet Contracts
3. Intermediate Contracts
4. Root Contract

Let's examine each level in detail:

Payment Channel Contracts

At the lowest level, payment channel contracts operate as child contracts of wallet contracts. Each payment channel is unilateral, facilitating transactions in a single direction. If bidirectional payments are required between two parties, two separate unilateral channels can be established. These channels are identified by unique SEQNO numbers, allowing for efficient management and reference within the system.

Definition 17 (Payment Channel Contract)**.** *A payment channel contract in Overpass Channels is a unilateral agreement facilitating off-chain transactions in a single direction, with periodic on-chain settlements. It is identified by a unique SEQNO number within its parent wallet contract.*

Theorem 27 (Channel Efficiency)**.** *The computational complexity of processing a transaction within a channel is O(1) with respect to the total number of transactions in the network.*

Proof. Let T be a transaction within channel C between Alice and Bob. The proof proceeds as follows:

1. Processing T involves:
   (a) Generating a zk-SNARK proof P
   (b) Verifying P
   (c) Updating the local channel state
2. The zk-SNARK proof generation and verification depend only on the circuit complexity, which is constant for all transactions.

3. Updating the local channel state involves modifying a fixed number of variables (balances, nonce, etc.).

4. None of these operations depend on the total number of transactions in the network.

Therefore, the computational complexity of processing T is O(1) with respect to the total number of transactions in the network. ■

Wallet Contracts

Wallet contracts serve as the primary interface for users, managing their funds and initiating payment channels. Each wallet contract can create and oversee multiple payment channel contracts.

Definition 18 (Wallet Contract)**.** *A wallet contract in Overpass Channels is a smart contract that:*

- *Manages a user's funds*
- *Initiates and oversees multiple payment channel contracts*
- *Interacts with intermediate contracts for state aggregation*

Intermediate Contracts

Intermediate contracts form the next layer in the hierarchy. Each intermediate contract aggregates state updates from multiple wallet contracts. This aggregation includes information about all payment channels associated with each wallet. By compiling data from numerous wallets, intermediate contracts significantly reduce the computational load on the root contract.

Definition 19 (Intermediate Contract)**.** *An intermediate contract in Overpass Channels is a smart contract that:*

- *Manages a subset of wallet contracts*
- *Aggregates state updates from these wallet contracts, including all associated payment channels*
- *Periodically submits aggregated updates to the root contract*
- *Is identified by a unique SEQNO number*

Theorem 28 (Intermediate Contract Efficiency)**.** *The computational complexity of processing updates in an intermediate contract is O(log w), where w is the number of wallet contracts managed by the intermediate contract.*

Proof. Let IC be an intermediate contract managing w wallet contracts. The proof proceeds as follows:

1. IC maintains a Merkle tree T of wallet contract states.
2. For each wallet contract update:
   (a) Verify the zk-SNARK proof (O(1))
   (b) Update the corresponding leaf in T (O(log w))

(c) Recompute the Merkle root (O(log w))

3. Periodically, IC submits the new Merkle root to the root contract (O(1))

The dominant operation is updating the Merkle tree, which has a complexity of O(log w). Therefore, the overall complexity of processing updates in IC is O(log w). ■

Off-chain Intermediate Contract Verification and Security

In the Overpass Channels system, Intermediate Contracts (ICs) are implemented as off-chain components written in Rust, utilizing WASM bindgen for interoperability. This architecture, combined with SNARK circuits for state transitions and epistemically redundant off-chain storage, presents a unique security model:

1. **Off-chain Execution**: ICs execute off-chain, allowing for efficient and complex computations without blockchain congestion.

2. **SNARK Circuit Verification**: Each state transition is encapsulated in a SNARK circuit, providing cryptographic proof of correctness:

$$\pi_{\text{transition}} = \text{GenerateProof}(\text{circuit}, \text{old\_state}, \text{new\_state}, \text{transition\_params})$$

3. **Redundant Storage**: State data is stored across multiple off-chain nodes with epistemic redundancy, ensuring availability and integrity:

$$\text{Reliability} \propto 1 - (1 - p)^n$$

where p is the probability of a single node being available, and n is the number of redundant nodes.

4. **Verification Process**: When a state transition occurs:

   - The transition is computed off-chain.
   - A SNARK proof $\pi_{\text{transition}}$ is generated.
   - The proof and new state are distributed to redundant storage nodes.
   - Other participants can verify the proof to confirm the transition's validity.

Security Analysis This off-chain model with SNARK-based verification offers several security advantages:

- **Cryptographic Verifiability**: Every state transition is provably correct.
- **Enhanced Privacy**: Off-chain execution and SNARKs protect sensitive transaction details.
- **Scalability**: Off-chain processing allows for higher transaction throughput.
- **Flexibility**: Rust implementation enables complex logic beyond typical smart contract capabilities.

However, this model also presents unique challenges:

- **Data Availability**: Ensuring all verification data is consistently available.
- **Synchronization**: Maintaining consistency across all off-chain nodes.
- **Finality Determination**: Establishing clear rules for transaction finality in the absence of global blockchain consensus.

Despite these challenges, the combination of SNARK-based verification, redundant off-chain storage, and the flexibility of Rust implementation provides a robust security model. This approach offers comparable, and in some aspects superior, security to traditional on-chain smart contracts, particularly in terms of privacy and scalability.

Root Contract

At the apex of the Overpass Channels hierarchy is the root contract. This contract manages all intermediate contracts, each identified by its SEQNO number. The root contract is responsible for maintaining the global state of the entire network, processing the aggregated data from intermediate contracts.

Definition 20 (Root Contract)**.** *A root contract in Overpass Channels is a smart contract that:*

- *Manages multiple intermediate contracts, each identified by its SEQNO number*
- *Maintains the global state of the network*
- *Handles final dispute resolution*

Theorem 29 (Root Contract Scalability)**.** *The computational complexity of updating the global state in the root contract is O(log m), where m is the number of intermediate contracts.*

Proof. Let RC be the root contract managing m intermediate contracts. The proof proceeds as follows:

1. RC maintains a Merkle tree T_{global} of intermediate contract states.
2. For each update from an intermediate contract:
   (a) Verify the submitted Merkle root (O(1))
   (b) Update the corresponding leaf in T_{global} (O(log m))
   (c) Recompute the global Merkle root (O(log m))

The dominant operation is updating T_{global}, which has a complexity of O(log m).
Therefore, the overall complexity of updating the global state in RC is O(log m). ■

9.2 System-Level Efficiency

The hierarchical structure of Overpass Channels contributes to its overall efficiency and scalability.

Theorem 30 (System-Level Efficiency)**.** *The Overpass Channels network can process N transactions with a total computational complexity of*

$$O(N + \log m \log w)$$

, where N is the total number of transactions, m is the number of intermediate contracts, and w is the average number of wallet contracts per intermediate contract.

Proof. Consider a network with N transactions, m intermediate contracts, and an average of w wallet contracts per intermediate contract. The proof proceeds as follows:

1. **Transaction Processing** $O(N)$: Each of the N transactions occurs within a specific channel, and each channel is managed by a wallet contract. The complexity of processing each individual transaction is $O(1)$, as it involves verifying the sender's balance and updating the channel state. Therefore, the total complexity for processing all N transactions is $O(N)$.

2. **Wallet State Updates** $O(N \log c)$: Each transaction updates the state of a channel within a wallet contract. Wallet contracts use a Merkle tree T_{wallet} to store the state of all channels they manage. Updating the state of a specific channel requires:

   - Updating the corresponding leaf node in T_{wallet} with the new state of the channel.
   - Recomputing the Merkle root for T_{wallet}.

   If a wallet contract manages c channels on average, updating and recomputing the Merkle root has a complexity of $O(\log c)$. As each of the N transactions requires updating T_{wallet}, the total complexity for updating wallet states is $O(N \log c)$.

3. **Intermediate Contract Updates** $O(N \log w)$: After a wallet contract processes transactions and updates its local state, it submits the updated Merkle root to its parent intermediate contract. Each intermediate contract maintains a Merkle tree $T_{\text{intermediate}}$ to track the states of the w wallet contracts it manages. Updating $T_{\text{intermediate}}$ requires:

   - Verifying the updated Merkle root of the wallet contract (which takes $O(1)$).
   - Updating the corresponding leaf in $T_{\text{intermediate}}$ (which takes $O(\log w)$).
   - Recomputing the Merkle root of $T_{\text{intermediate}}$ (which also takes $O(\log w)$).

   The amortized cost of updating the intermediate contract state per transaction is $O(\log w)$. Therefore, the total complexity of intermediate contract updates is $O(N \log w)$.

4. **Global State Updates** $O(N \log m)$: After an intermediate contract processes updates from its wallet contracts, it submits its updated Merkle root to the root contract. The root contract maintains a global Merkle tree T_{global} to track the states of the m intermediate contracts. Updating T_{global} requires:

   - Verifying the updated Merkle root of the intermediate contract (which takes $O(1)$).
   - Updating the corresponding leaf in T_{global} (which takes $O(\log m)$).
   - Recomputing the global Merkle root (which also takes $O(\log m)$).

The amortized cost of updating the global state per transaction is $O(\log m)$. Therefore, the total complexity of global state updates is $O(N \log m)$.

Total Computational Complexity: Combining these results, the total computational complexity for processing N transactions and maintaining the global state is:

$$O(N) + O(N \log c) + O(N \log w) + O(N \log m) = O(N(\log c + \log w + \log m)).$$

Since c (the average number of channels per wallet) is typically much smaller than w (the number of wallets per intermediate contract), we can simplify this to:

$$O(N(\log w + \log m)) = O(N \log(mw)).$$

Thus, the total computational complexity is $O(N \log(mw))$, where N is the number of transactions, w is the average number of wallet contracts per intermediate contract, and m is the number of intermediate contracts. ■

10 Storage Nodes and Data Management

10.1 Individual User Devices and Wallet Trees

In Overpass Channels, each user's device is responsible for maintaining a single sparse Merkle tree for the wallet contract. This tree tracks the states of all channel contracts associated with the wallet, ensuring efficient and secure updates. These operations are supported by *off-chain storage nodes* that form part of the Overpass system.

1. **Sparse Merkle tree for the wallet contract:** The wallet contract contains a sparse Merkle tree where each leaf corresponds to the state of an associated channel contract, indexed by its SEQNO.

2. **Keys stored on user devices:** The cryptographic keys required to manage and update the wallet and its channels are securely stored on the user's device.

3. **Tree synchronization:** When a channel contract's state is updated, the corresponding leaf in the wallet's sparse Merkle tree is modified, and a new root is generated. This root and its cryptographic proof are then submitted to the off-chain storage nodes for redundancy and verification.

10.2 Periodic Channel Updates and Off-Chain Storage Nodes

The Overpass system relies on *off-chain storage nodes* for maintaining the periodic updates of wallet contract states. These storage nodes store data redundantly and securely, ensuring efficient retrieval and availability.

1. **Storage nodes in the Overpass system:** The storage nodes are responsible for storing wallet contract roots and their associated proofs. They do not operate on the blockchain but are instead a decentralized component of the Overpass off-chain system.

Algorithm 13 Wallet Update

```
procedure UpdateWalletTree(walletID, channelID, channelState)
    tree ← LoadWalletTree(walletID)
    leafIndex ← SEQNO(channelID)
    leafValue ← Hash(channelState)
    tree.update(leafIndex, leafValue)
    newRoot ← tree.getRoot()
    proof ← tree.generateProof(leafIndex)
    return (newRoot, proof)
end procedure
```

2. **Epidemic overlapping of shard data:** The storage nodes follow a design of *epidemic overlapping*, where each node stores data for a set number of intermediate contracts, and data overlaps with other storage nodes. This redundancy ensures that the system is decentralized and highly available.

3. **Scalability and efficiency:** The epidemic overlap mechanism allows the storage nodes to efficiently store data without requiring all nodes to store everything. As the network grows, the number of overlaps remains constant, preventing excessive storage demands and maintaining decentralized redundancy.

Algorithm 14 Submit Wallet Update to Off-Chain Storage Nodes

```
procedure SubmitPeriodicUpdate(walletID, walletRoot)
    proof ← GenerateProof(walletRoot)
    storageNodes ← GetAssociatedStorageNodes(walletID)
    for all node ∈ storageNodes do
        node.StoreWalletUpdate(walletID, walletRoot, proof)
    end for
end procedure
```

10.3 Storage Nodes and Intermediate Contracts

Storage nodes also handle data related to *intermediate contracts*, which aggregate the states of multiple wallet contracts. The *epidemic overlapping* mechanism ensures that intermediate contract data is stored redundantly across the storage nodes, ensuring availability and resilience.

1. **Storage of wallet roots:** Off-chain storage nodes store the roots and proofs of wallet contracts. These roots are retrieved by intermediate contracts for aggregation.

2. **Intermediate contract aggregation:** Intermediate contracts aggregate the states of several wallet contracts. This aggregation involves computing a Merkle root over the stored wallet contract roots, which is then redundantly stored across the storage nodes.

Algorithm 15 Intermediate Contract Data Aggregation

procedure AGGREGATEINTERMEDIATEDATA($intermediateContractID$)
    $walletContracts \leftarrow$ GetAssociatedWalletContracts($intermediateContractID$)
    $walletRoots \leftarrow []$
    for all $wallet \in walletContracts$ **do**
        $(root, proof) \leftarrow$ RetrieveLatestWalletUpdate($wallet$)
        $walletRoots$.append($(wallet, root, proof)$)
    end for
    $intermediateRoot \leftarrow$ ComputeMerkleRoot($walletRoots$)
    $intermediateProof \leftarrow$ GenerateProof($intermediateRoot, walletRoots$)
    StoreIntermediateUpdate($intermediateContractID, intermediateRoot, intermediateProof$)
end procedure

10.4 Root Contract and Global State

While the storage nodes operate off-chain, the *root contract* functions on the *blockchain* and plays a key role in generating the global state of the network. The root contract aggregates data from all intermediate contracts and periodically submits the global state on-chain in sharded blockchains. This process ensures decentralized verification and availability of the network's state.

The root contract also interacts with the storage nodes, retrieving the processed Merkle roots and proofs for all intermediate contracts, which have been stored after aggregating the individual wallet roots.

1. **Global aggregation:** The root contract retrieves the processed Merkle roots and proofs from the storage nodes, where the intermediate contracts' data is stored after the aggregation of wallet roots. These intermediate contract roots are then aggregated to compute the global Merkle root.

2. **On-chain submission:** The global root and its cryptographic proof are submitted on-chain at the end of each epoch. This ensures that the entire network's state is publicly verifiable and tamper-proof, leveraging sharded blockchain for decentralized availability.

10.5 Why the Redundant Copy Attack is Pointless

Even if a malicious actor owned all the redundant copies of certain data, they wouldn't be able to exploit this because:

1. **Challenging Mechanism**:

    - Each redundant copy of data can be challenged by other nodes or clients that still have valid copies. If someone tries to modify the data or shut down a portion of the system, the rest of the nodes can detect this inconsistency through the cryptographic proofs (Merkle trees and zk-SNARKs).
    - As you mentioned, the **client still holds the original data** and can provide the correct proofs to invalidate any tampered version.

Algorithm 16 Global State Generation and Storage

```
procedure GenerateGlobalState
    intermediateContracts ← GetAllIntermediateContracts()
    intermediateRoots ← []
    for all contract ∈ intermediateContracts do
        (root, proof) ← RetrieveIntermediateUpdateFromStorageNodes(contract)
        intermediateRoots.append((contract, root, proof))
    end for
    globalRoot ← ComputeGlobalMerkleRoot(intermediateRoots)
    globalProof ← GenerateGlobalProof(globalRoot, intermediateRoots)
    StoreGlobalState(globalRoot, globalProof)
    SubmitGlobalStateOnChain(globalRoot, globalProof)
    PruneOldGlobalStates()
end procedure
```

2. **Hierarchical Structure**:
    - The leaves at one level of the hierarchy serve as the roots for the next level. This makes it **self-reinforcing**, as the integrity of one level supports the integrity of the next.
    - If a malicious actor were to try to corrupt a particular node, it would be immediately caught by the mismatch in the hash values of the Merkle trees.
3. **Automatic Slashing/Disconnection**:
    - In the case that a node tries to game the system (e.g., by withholding or tampering with redundant data), the system has **automatic penalties** such as **slashing or disconnection**. This makes it economically unviable to attempt to manipulate the network, as they would simply lose their stake and get removed from the system.

10.6 Gaining Control Over All Redundancies is Impractical

Even if someone controls all the redundant copies of a specific subset of data, this is impractical for a few reasons:

- **Scalability of the Network**: The larger the network, the more difficult it is to control all redundancies for any significant portion of data. The system is designed to distribute redundancy across a wide array of nodes, meaning an attacker would need to control a massive number of nodes to affect even a small subset of the network.
- **Decentralized Nature of Redundancy**: Data is stored redundantly across many nodes. In order to "own" all redundant copies, an attacker would have to compromise multiple nodes at once, which is both **logistically difficult** and **economically expensive**. Moreover, controlling a few nodes doesn't provide any strategic advantage because the rest of the network would quickly detect tampered data.

10.7 Hierarchical Security: Redundancy in the Network Itself

The hierarchical structure of Overpass Channels adds an extra layer of **in-built security**:

1. **Each level validates the previous one**: The cryptographic proofs at each level (leaf, root, etc.) ensure that **tampering at any level** is detectable at the next. If a malicious actor were to modify data at any level, the discrepancy would immediately be caught when other nodes try to verify it.
2. **No Single Point of Failure**: Since data is distributed and redundantly stored, there is no single point of failure. If one node fails or acts maliciously, others can step in to provide the correct data, ensuring network reliability and integrity.

10.8 Client and On-Chain Challenge Mechanism

Finally, **clients** (or users) are always in control of their own data, and **on-chain records** serve as a reliable source of truth. If a node attempts to tamper with off-chain data:

1. **Clients can challenge** the data directly using zk-SNARKs and Merkle proofs to demonstrate the correct state.
2. **On-chain records** (such as root hashes) ensure that any off-chain modifications that don't match the on-chain state can be easily detected and corrected.

10.9 No Incentive for Attackers

Any potential attacker would face several hurdles:

- **Cryptographic proofs** make tampering virtually impossible without being detected.
- **Immediate penalties** such as slashing or disconnection.
- **No strategic gain** from controlling redundant data, as it's easily challengeable by clients or the network.

The inherent structure of Overpass Channels—leveraging cryptographic guarantees, decentralization, and hierarchical Merkle trees—provides all the necessary protection. There is no real need to add complexity, as redundancy is already **secure, decentralized, and self-reinforcing**.

In short, **even if an attacker controls all redundant copies, the network's design ensures that any malicious actions will be swiftly detected and neutralized**. The combination of client-side control, automatic slashing, and robust proof mechanisms makes the system resilient against manipulation.

10.10 Incentivizing and Maintaining via Staking and Battery Charging

Efficient and reliable storage is paramount for the Overpass Channels network to maintain data redundancy, availability, and integrity. To ensure that storage nodes remain active, synchronized, and trustworthy, Overpass Channels introduces a staking mechanism combined with a novel "battery charging" system. This section elaborates on how staking enables participation as a storage node and how the battery charging system incentivizes optimal node performance.

10.11 Staking to Become a Storage Node

Staking serves as both a commitment mechanism and an incentive structure, ensuring that storage nodes have a vested interest in maintaining the network's reliability and security.

Definition 21 (Storage Node Staking)**.** *Staking in Overpass Channels involves locking a certain amount of tokens as collateral to become an authorized storage node. This stake acts as a security deposit that can be forfeited in cases of malicious behavior or failure to maintain synchronization and availability.*

Theorem 31 (Staking as a Security Mechanism)**.** *Let S be the amount of tokens staked by a storage node, and let B represent the node's battery charge. The probability of a storage node behaving maliciously is inversely proportional to S and directly proportional to B.*

$$P(\textit{malicious behavior}) \propto \frac{B}{S}$$

Proof. 1. A higher stake S increases the financial loss incurred by malicious behavior, thereby providing a stronger deterrent.

2. A higher battery charge B indicates better synchronization and availability, reducing the likelihood of operational failures that could be exploited.

3. Therefore, the probability of malicious behavior increases with B (since a well-synced node is more attractive for exploitation) and decreases with S (due to higher stakes discouraging misconduct).

■

10.12 Battery Charging Mechanism for Storage Nodes

The "battery charging" system is designed to dynamically adjust the operational status and incentives of storage nodes based on their synchronization and overlap with other nodes. This mechanism ensures that only reliable and well-synchronized nodes remain active, while underperforming nodes are temporarily suspended and replaced.

Battery Charging Dynamics

The battery charging mechanism operates as follows:

- **Charging Up**: Storage nodes charge their batteries by maintaining precise synchronization and overlapping with other storage nodes. The more nodes that overlap and synchronize, the higher the charge rate.
- **Discharging**: If a storage node falls out of synchronization or fails to overlap accurately with other nodes, its battery begins to discharge.
- **Thresholds**:
  - **Optimal Charge**: 98% - 100% battery charge. Nodes within this range receive maximum rewards.
  - **High Charge**: 80% - 98% battery charge. Nodes in this range receive proportional rewards based on their charge level.

  - **Low Charge**: Below 80% battery charge. Nodes start to incur penalties and may eventually be suspended if the charge depletes to 0%.

- **Suspension and Replacement**: If a storage node's battery depletes to 0%, it is disconnected from the network and placed in a suspension status for a predefined duration. During suspension, the node cannot participate until its battery is recharged above a certain threshold. After suspension, the node can be replaced by another storage node through the staking mechanism.

Algorithm: Battery Charging and Maintenance

Algorithm 17 Battery Charging and Maintenance for Storage Nodes

```
procedure UpdateBattery(nodeID, synchronizationStatus)
    node ← GetNode(nodeID)
    if synchronizationStatus = Synchronized then
        node.battery ← min(node.battery + ΔC, 100)                ▷ Charge up
    else
        node.battery ← max(node.battery − ΔD, 0)                  ▷ Discharge
    end if
    if node.battery = 0 then
        SuspendNode(nodeID, duration)
        ReplaceNode(nodeID)
    else if node.battery ≥ 98 then
        node.rewardMultiplier ← 1.0                               ▷ Max reward
    else if node.battery ≥ 80 then
        node.rewardMultiplier ← node.battery / 100                ▷ Proportional reward
    else
        node.rewardMultiplier ← 0.0                               ▷ No reward, potential penalty
    end if
    SaveNodeState(node)
end procedure
```

Charging Mechanics and Synchronization

Theorem 32 (Battery Charging Efficiency)**.** *The rate of battery charging is directly proportional to the number of synchronized and overlapping storage nodes.*

$$\Delta C \propto \sum_{i=1}^{n} Overlap(node, node_i)$$

Proof. 1. Each synchronized and overlapping node contributes to the charge rate of a storage node.

2. The cumulative effect of multiple overlapping nodes increases the overall charge rate proportionally.

3. Therefore, the battery charging increment ΔC scales with the number of overlapping and synchronized nodes.

■

Reward Distribution Based on Battery Charge

Algorithm 18 Reward Distribution to Storage Nodes

```
procedure DistributeRewards(transactionFees)
    for all node ∈ ActiveNodes do
        if node.battery ≥ 80 then
            if node.battery ≥ 98 then
                reward ← transactionFees × 0.10                          ▷ Max reward: 10%
            else
                reward ← transactionFees × (0.10 × node.battery / 100)   ▷ Proportional reward
            end if
            Transfer(node.address, reward)
        end if
    end for
end procedure
```

Theorem 33 (Optimal Reward Incentivization)**.** *Storage nodes with battery charges between 98% and 100% receive the maximum reward, incentivizing them to maintain high synchronization and overlap.*

$$Reward(node) = \begin{cases} Max\ Reward & if\ 98 \leq battery \leq 100 \\ Proportional\ Reward & if\ 80 \leq battery < 98 \\ 0 & if\ battery < 80 \end{cases}$$

Proof. 1. Nodes with battery charges $\geq 98\%$ are optimally synchronized and overlapping, warranting maximum rewards to incentivize continued high performance.

2. Nodes with battery charges between 80% and 98% are sufficiently synchronized but may have minor lapses, receiving proportional rewards to encourage maintaining higher synchronization.

3. Nodes below 80% battery charge are considered underperforming, receiving no rewards and being subject to potential penalties or suspension to maintain overall network integrity.

4. This tiered reward structure aligns incentives with desired node behaviors, promoting optimal synchronization and overlap across the network.

■

10.13 Battery Charging Interaction with Intermediate Contracts

The battery charging mechanism interacts seamlessly with intermediate contracts to ensure that storage nodes maintain high levels of synchronization and reliability.

- **State Aggregation**: Intermediate contracts continuously aggregate state updates from managed wallet contracts. The synchronization status of these updates directly influences the battery charging rate of associated storage nodes.
- **Redundancy and Overlap**: As intermediate contracts manage multiple wallet contracts, storage nodes responsible for these contracts benefit from redundancy. High overlap between

storage nodes' responsibilities increases synchronization precision, thereby enhancing battery charges.

- **Failure Mitigation**: If synchronization lapses occur, the discharging mechanism ensures that underperforming nodes are identified and temporarily removed from active participation, maintaining the network's overall health.

10.14 Security and Robustness of the Battery Charging System

The battery charging system introduces additional layers of security and robustness to the Overpass Channels network.

- **Economic Incentives**: By tying rewards to battery charge levels, storage nodes are economically motivated to maintain high synchronization and overlap, reducing the likelihood of intentional or accidental misbehavior.
- **Redundancy through Overlapping Nodes**: The overlapping storage nodes create a redundant verification layer. Even if some nodes attempt to behave maliciously, the high battery charges required to earn rewards ensure that only genuinely synchronized nodes remain incentivized.
- **Automated Maintenance**: The suspension and replacement mechanisms automatically handle underperforming nodes, ensuring continuous network reliability without manual intervention.
- **Dynamic Adjustments**: The system can dynamically adjust the reward multipliers and suspension durations based on network conditions, maintaining an optimal balance between incentives and penalties.

10.15 Implementation Considerations

Implementing the staking and battery charging system requires careful consideration of various technical aspects to ensure seamless integration with Overpass Channels' existing architecture.

1. **Smart Contract Design**:
    - **Staking Contracts**: Manage the staking process, including lock-up periods, stake withdrawals, and slashing conditions.
    - **Battery Management Contracts**: Handle battery charge updates, reward distributions, and node suspensions.
2. **Synchronization Monitoring**: Develop robust mechanisms to monitor and assess the synchronization status of storage nodes in real-time, feeding accurate data into the battery charging system.
3. **Redundant Storage Coordination**: Ensure that storage nodes correctly implement epistemic redundancy, maintaining synchronized copies across multiple nodes to facilitate accurate battery charging and discharging.

4. **Reward Distribution Logic**: Implement efficient and secure reward distribution algorithms that calculate and transfer rewards based on battery charge levels without introducing bottlenecks or vulnerabilities.

5. **Suspension and Replacement Protocols**: Define clear protocols for suspending underperforming nodes and replacing them with new storage nodes, ensuring minimal disruption to the network.

6. **Scalability and Performance Optimization**: Optimize the staking and battery charging mechanisms to handle large numbers of storage nodes without degrading network performance.

10.16 Potential Vulnerabilities and Mitigations

While the staking and battery charging system enhances network reliability and security, it introduces potential vulnerabilities that must be addressed:

- **Sybil Attacks**: Malicious actors could attempt to create multiple storage nodes to gain disproportionate rewards.
  - **Mitigation**: Implement identity verification and staking requirements that make Sybil attacks economically unviable.
- **Battery Manipulation**: Storage nodes might attempt to manipulate synchronization data to artificially inflate battery charges.
  - **Mitigation**: Employ cryptographic proofs (e.g., zk-SNARKs) to verify synchronization without revealing sensitive data, ensuring that battery charges accurately reflect true synchronization status.
- **Network Partitioning**: Temporary network splits could cause storage nodes to lose synchronization, depleting their batteries and leading to unnecessary suspensions.
  - **Mitigation**: Incorporate grace periods and allow for automatic battery recharge upon re-synchronization, preventing long-term penalties from transient network issues.
- **Economic Exploits**: Nodes could collude to game the reward system, ensuring that their batteries remain charged without providing genuine service.
  - **Mitigation**: Design the reward mechanism to require independent verification of synchronization and overlap, preventing collusion from bypassing actual service provision.

10.17 Integrating Beneficial MEV: Enabling Arbitrage without Compromising Privacy

While Overpass Channels' primary goal is to mitigate harmful MEV, integrating beneficial MEV activities like arbitrage can enhance market efficiency. Below are mechanisms to support beneficial arbitrage within Overpass Channels:

- **Confidential Order Matching**: Implement zk-SNARK-based order matching protocols within payment channels. Arbitrageurs can execute trades based on aggregated market data without accessing individual order details. zk-SNARK proofs verify that trades comply with market rules without revealing transaction specifics.
- **Aggregated Price Feeds**: Utilize decentralized oracles to provide aggregated and anonymized price feeds to payment channels. Arbitrageurs can identify and exploit price discrepancies across different channels or exchanges based on these feeds without accessing underlying transaction data.
- **Incentive Structures for Arbitrageurs**: Design incentive mechanisms that reward arbitrageurs for maintaining market efficiency. Rewards can be encoded within zk-SNARK proofs, ensuring that only legitimate arbitrage activities are compensated without disclosing transaction specifics.
- **Decoupled Execution and Settlement**: Separate the execution of arbitrage transactions from their settlement. Execution occurs within private channels using zk-SNARKs, while settlement aggregates results on-chain in a privacy-preserving manner. This ensures that arbitrage activities enhance market efficiency without introducing MEV vulnerabilities.

11 Incentivizing Storage Nodes

Efficient and reliable storage is paramount for the Overpass Channels network to maintain data redundancy, availability, and integrity. To ensure that storage nodes remain active, synchronized, and trustworthy, Overpass Channels introduces a staking mechanism combined with a novel "battery charging" system. This section elaborates on how staking enables participation as a storage node and how the battery charging system incentivizes optimal node performance.

11.1 Staking to Become a Storage Node

Staking serves as both a commitment mechanism and an incentive structure, ensuring that storage nodes have a vested interest in maintaining the network's reliability and security.

Definition 22 (Storage Node Staking)**.** *Staking in Overpass Channels involves locking a certain amount of tokens as collateral to become an authorized storage node. This stake acts as a security deposit that can be forfeited in cases of malicious behavior or failure to maintain synchronization and availability.*

Theorem 34 (Staking as a Security Mechanism)**.** *Let S be the amount of tokens staked by a storage node, and let B represent the node's battery charge. The probability of a storage node behaving maliciously is inversely proportional to S and directly proportional to B.*

$$P(\textit{malicious behavior}) \propto \frac{B}{S}$$

Proof. 1. A higher stake S increases the financial loss incurred by malicious behavior, thereby providing a stronger deterrent.

2. A higher battery charge B indicates better synchronization and availability, reducing the likelihood of operational failures that could be exploited.

3. Therefore, the probability of malicious behavior increases with B (since a well-synced node is more attractive for exploitation) and decreases with S (due to higher stakes discouraging misconduct).

■

11.2 Battery Charging Mechanism for Storage Nodes

The "battery charging" system is designed to dynamically adjust the operational status and incentives of storage nodes based on their synchronization and overlap with other nodes. This mechanism ensures that only reliable and well-synchronized nodes remain active, while underperforming nodes are temporarily suspended and replaced.

Battery Charging Dynamics

The battery charging mechanism operates as follows:

- **Charging Up**: Storage nodes charge their batteries by maintaining precise synchronization and overlapping with other storage nodes. The more nodes that overlap and synchronize, the higher the charge rate.
- **Discharging**: If a storage node falls out of synchronization or fails to overlap accurately with other nodes, its battery begins to discharge.
- **Thresholds**:
  - **Optimal Charge**: 98% - 100% battery charge. Nodes within this range receive maximum rewards.
  - **High Charge**: 80% - 98% battery charge. Nodes in this range receive proportional rewards based on their charge level.
  - **Low Charge**: Below 80% battery charge. Nodes start to incur penalties and may eventually be suspended if the charge depletes to 0%.
- **Suspension and Replacement**: If a storage node's battery depletes to 0%, it is disconnected from the network and placed in a suspension status for a predefined duration. During suspension, the node cannot participate until its battery is recharged above a certain threshold. After suspension, the node can be replaced by another storage node through the staking mechanism.

Algorithm: Battery Charging and Maintenance

Charging Mechanics and Synchronization

Theorem 35 (Battery Charging Efficiency)**.** *The rate of battery charging is directly proportional to the number of synchronized and overlapping storage nodes.*

$$\Delta C \propto \sum_{i=1}^{n} Overlap(node, node_i)$$

Algorithm 19 Battery Charging and Maintenance for Storage Nodes

procedure UPDATEBATTERY($nodeID$, $synchronizationStatus$)
  $node \leftarrow$ GetNode($nodeID$)
  if $synchronizationStatus =$ Synchronized **then**
    $node$.battery $\leftarrow \min(node.\text{battery} + \Delta C, 100)$ ▷ Charge up
  else
    $node$.battery $\leftarrow \max(node.\text{battery} - \Delta D, 0)$ ▷ Discharge
  end if
  if $node$.battery $= 0$ **then**
    SuspendNode($nodeID$, duration)
    ReplaceNode($nodeID$)
  else if $node$.battery ≥ 98 **then**
    $node$.rewardMultiplier $\leftarrow 1.0$ ▷ Max reward
  else if $node$.battery ≥ 80 **then**
    $node$.rewardMultiplier $\leftarrow \frac{node.\text{battery}}{100}$ ▷ Proportional reward
  else
    $node$.rewardMultiplier $\leftarrow 0.0$ ▷ No reward, potential penalty
  end if
  SaveNodeState(node)
end procedure

Proof. 1. Each synchronized and overlapping node contributes to the charge rate of a storage node.

2. The cumulative effect of multiple overlapping nodes increases the overall charge rate proportionally.

3. Therefore, the battery charging increment ΔC scales with the number of overlapping and synchronized nodes.

■

Reward Distribution Based on Battery Charge

Algorithm 20 Reward Distribution to Storage Nodes

procedure DISTRIBUTEREWARDS($transactionFees$)
  for all $node \in$ ActiveNodes **do**
    if $node$.battery ≥ 80 **then**
      if $node$.battery ≥ 98 **then**
        $reward \leftarrow transactionFees \times 0.10$ ▷ Max reward: 10%
      else
        $reward \leftarrow transactionFees \times \left(0.10 \times \frac{node.\text{battery}}{100}\right)$ ▷ Proportional reward
      end if
      Transfer($node$.address, $reward$)
    end if
  end for
end procedure

Theorem 36 (Optimal Reward Incentivization)**.** *Storage nodes with battery charges between 98% and 100% receive the maximum reward, incentivizing them to maintain high synchronization and overlap.*

$$Reward(node) = \begin{cases} Max\ Reward & if\ 98 \leq battery \leq 100 \\ Proportional\ Reward & if\ 80 \leq battery < 98 \\ 0 & if\ battery < 80 \end{cases}$$

Proof. 1. Nodes with battery charges $\geq 98\%$ are optimally synchronized and overlapping, warranting maximum rewards to incentivize continued high performance.

2. Nodes with battery charges between 80% and 98% are sufficiently synchronized but may have minor lapses, receiving proportional rewards to encourage maintaining higher synchronization.

3. Nodes below 80% battery charge are considered underperforming, receiving no rewards and being subject to potential penalties or suspension to maintain overall network integrity.

4. This tiered reward structure aligns incentives with desired node behaviors, promoting optimal synchronization and overlap across the network.

■

11.3 Battery Charging Interaction with Intermediate Contracts

The battery charging mechanism interacts seamlessly with intermediate contracts to ensure that storage nodes maintain high levels of synchronization and reliability.

- **State Aggregation**: Intermediate contracts continuously aggregate state updates from managed wallet contracts. The synchronization status of these updates directly influences the battery charging rate of associated storage nodes.
- **Redundancy and Overlap**: As intermediate contracts manage multiple wallet contracts, storage nodes responsible for these contracts benefit from redundancy. High overlap between storage nodes' responsibilities increases synchronization precision, thereby enhancing battery charges.
- **Failure Mitigation**: If synchronization lapses occur, the discharging mechanism ensures that underperforming nodes are identified and temporarily removed from active participation, maintaining the network's overall health.

11.4 Security and Robustness of the Battery Charging System

The battery charging system introduces additional layers of security and robustness to the Overpass Channels network.

- **Economic Incentives**: By tying rewards to battery charge levels, storage nodes are economically motivated to maintain high synchronization and overlap, reducing the likelihood of intentional or accidental misbehavior.
- **Redundancy through Overlapping Nodes**: The overlapping storage nodes create a redundant verification layer. Even if some nodes attempt to behave maliciously, the high battery charges required to earn rewards ensure that only genuinely synchronized nodes remain incentivized.

- **Automated Maintenance**: The suspension and replacement mechanisms automatically handle underperforming nodes, ensuring continuous network reliability without manual intervention.
- **Dynamic Adjustments**: The system can dynamically adjust the reward multipliers and suspension durations based on network conditions, maintaining an optimal balance between incentives and penalties.

11.5 Implementation Considerations

Implementing the staking and battery charging system requires careful consideration of various technical aspects to ensure seamless integration with Overpass Channels' existing architecture.

1. **Smart Contract Design**:
    - **Staking Contracts**: Manage the staking process, including lock-up periods, stake withdrawals, and slashing conditions.
    - **Battery Management Contracts**: Handle battery charge updates, reward distributions, and node suspensions.
2. **Synchronization Monitoring**: Develop robust mechanisms to monitor and assess the synchronization status of storage nodes in real-time, feeding accurate data into the battery charging system.
3. **Redundant Storage Coordination**: Ensure that storage nodes correctly implement epistemic redundancy, maintaining synchronized copies across multiple nodes to facilitate accurate battery charging and discharging.
4. **Reward Distribution Logic**: Implement efficient and secure reward distribution algorithms that calculate and transfer rewards based on battery charge levels without introducing bottlenecks or vulnerabilities.
5. **Suspension and Replacement Protocols**: Define clear protocols for suspending underperforming nodes and replacing them with new storage nodes, ensuring minimal disruption to the network.
6. **Scalability and Performance Optimization**: Optimize the staking and battery charging mechanisms to handle large numbers of storage nodes without degrading network performance.

11.6 Integrating Beneficial MEV: Enabling Arbitrage without Compromising Privacy

Integrating beneficial MEV activities like arbitrage can enhance market efficiency. Below are mechanisms to support beneficial arbitrage within Overpass Channels:

- **Confidential Order Matching**: Implement zk-SNARK-based order matching protocols within payment channels. Arbitrageurs can execute trades based on aggregated market data without accessing individual order details. zk-SNARK proofs verify that trades comply with market rules without revealing transaction specifics.

- **Aggregated Price Feeds**: Utilize decentralized oracles to provide aggregated and anonymized price feeds to payment channels. Arbitrageurs can identify and exploit price discrepancies across different channels or exchanges based on these feeds without accessing underlying transaction data.
- **Incentive Structures for Arbitrageurs**: Design incentive mechanisms that reward arbitrageurs for maintaining market efficiency. Rewards can be encoded within zk-SNARK proofs, ensuring that only legitimate arbitrage activities are compensated without disclosing transaction specifics.
- **Decoupled Execution and Settlement**: Separate the execution of arbitrage transactions from their settlement. Execution occurs within private channels using zk-SNARKs, while settlement aggregates results on-chain in a privacy-preserving manner. This ensures that arbitrage activities enhance market efficiency without introducing MEV vulnerabilities.

12 Hierarchical Sparse Merkle Trees in Overpass Channels

In Overpass Channels, *sparse Merkle trees* are employed at multiple levels of the system to manage state updates efficiently, securely, and with minimal data overhead. These trees are particularly well-suited for scenarios where most potential data entries (leaves) are empty, ensuring that even in sparse configurations, proofs remain compact and efficient.

Sparse Merkle trees enable the Overpass system to achieve a *high level of scalability and efficiency*, especially when combined with off-chain storage nodes and the decentralized structure of the blockchain for global state verification.

12.1 Benefits of Sparse Merkle Trees in Overpass Channels

The use of sparse Merkle trees in Overpass Channels provides several distinct advantages:

1. **Compact Proofs and Constant Size:** Sparse Merkle trees allow for *constant-sized proofs* regardless of the number of leaves (or channels) in the tree. This is critical in Overpass Channels, as it ensures that the submission of proofs remains efficient even as the network grows. Each proof, whether for a wallet contract or the global state, involves only recalculating the path from a specific leaf to the root.
2. **Efficient State Updates:** Sparse Merkle trees enable O(log n) complexity for updates, meaning that only the path from the affected leaf to the root needs to be recalculated when a state changes. In the context of Overpass Channels, this allows for *incremental updates* when a channel contract state changes, ensuring that the system remains scalable and responsive.
3. **Proof of Integrity:** Any alteration in a channel's state is immediately reflected in the corresponding wallet's Merkle tree root. This ensures *data integrity* at every level of the hierarchy, as any modification

   to a wallet or channel contract would trigger a corresponding update in the global state. The ability to trace any change back to its root provides robust security guarantees.

4. **Sparse Data Handling:** Sparse Merkle trees are designed to handle cases where most leaves are unoccupied. This feature is particularly useful for Overpass Channels, where the number of possible channel contracts is vast, but only a subset may be active at any given time. The system is able to efficiently manage and verify these active states without the need to store or process large amounts of empty data.

5. **Redundant and Decentralized Storage:** In conjunction with *off-chain storage nodes*, the Merkle tree structure enables redundant and decentralized storage. Intermediate and wallet contract roots are stored across multiple nodes with *epidemic overlap*, allowing for redundancy and resilience in the system without each node needing to store all data. The efficient and decentralized nature of this storage ensures that the network remains robust as it scales.

12.2 Application of Sparse Merkle Trees in the Overpass Hierarchy

Sparse Merkle trees play a critical role in the *multi-tiered architecture* of Overpass Channels:

- At the *wallet contract* level, the tree tracks the state of associated channels.
- At the *intermediate contract* level, the Merkle roots of several wallet contracts are aggregated.
- Finally, at the *root contract* level, the global Merkle tree is computed and submitted on-chain, ensuring that the entire system's state remains verifiable and transparent.

The *constant-size nature of the global root submission* ensures that even as the network grows in size, the cost of verifying and submitting the global state remains predictable and efficient.

13 Implementation Considerations

Implementing Overpass Channels requires careful consideration of several technical aspects. This section outlines key implementation details and provides pseudocode for critical components.

13.1 Merkle Tree Structure

Overpass Channels uses a Merkle tree to efficiently represent the state of all channels. Here's a high-level implementation of the Merkle tree structure:

This Merkle tree structure allows for efficient updates and verifications of channel states.

13.2 zk-SNARK Circuit

The zk-SNARK circuit for Overpass Channels must encode the logic for valid state transitions. Here's a simplified representation of the circuit:

This circuit ensures that all state transitions in Overpass Channels are valid and authorized.

13.3 Minimal Cross-Shard Data

To minimize cross-shard communication overhead, Overpass Channels uses a compact representation for cross-shard transactions that includes both ‘shardID‘ and ‘workchainID‘:

Algorithm 21 Merkle Tree for Overpass Channels

procedure UPDATEMERKLETREE($channelID, newState$)
    $leaf \leftarrow$ Hash($channelID \parallel newState$)
    $path \leftarrow$ GetMerklePath($channelID$)
    $newRoot \leftarrow$ ComputeNewRoot($leaf, path$)
    UpdateRoot($newRoot$)
end procedure
function VERIFYCHANNELSTATE($channelID, state, proof$)
    $leaf \leftarrow$ Hash($channelID \parallel state$)
    $root \leftarrow$ GetCurrentRoot()
    return VerifyMerkleProof($leaf, root, proof$)
end function

Algorithm 22 zk-SNARK Circuit for Overpass Channels

procedure CHANNELTRANSITIONCIRCUIT($seqno, oldState, newState, tx, signature, oldNonce, newNonce$)
    AssertChannelSeqnoMatch($seqno, oldState.seqno, newState.seqno$)
    AssertValidSignature($tx, signature$)
    AssertValidBalanceTransition($oldState, newState, tx$)
    AssertValidNonceIncrement($oldNonce, newNonce$)
    AssertValidStateTransition($oldState, newState$)
end procedure

Algorithm 23 Cross-Shard Transaction Encoding

function ENCODECROSSSHARDTX($tx, proof, workchainID, shardID$)
    $encodedTx \leftarrow$ Hash(tx)
    $encodedProof \leftarrow$ CompressZKProof($proof$)
    return $encodedTx \parallel encodedProof \parallel workchainID \parallel shardID$
end function
function DECODECROSSSHARDTX($encodedData$)
    $encodedTx, encodedProof, workchainID, shardID \leftarrow$ Split($encodedData$)
    $tx \leftarrow$ LookupTx($encodedTx$)
    $proof \leftarrow$ DecompressZKProof($encodedProof$)
    return $tx, proof, workchainID, shardID$
end function

13.4 Integration

Integrating Overpass Channels involves creating smart contracts that can interact with the Overpass Channels system:

Algorithm 24 Smart Contract for Overpass Channels

```
procedure ProcessOverpassTx(encodedTx, proof)
    tx, decodedProof ← DecodeCrossShardTx(encodedTx)
    isValid ← VerifyZKProof(decodedProof, tx)
    if isValid then
        UpdateChannelState(tx)
        EmitEvent("ChannelUpdated", tx)
    else
        RevertTransaction()
    end if
end procedure
```

This smart contract serves as the interface between and Overpass Channels, ensuring that all channel operations are properly verified and recorded on the blockchain.

By carefully implementing these components, Overpass Channels can be seamlessly integrated with, providing a powerful, scalable, and privacy-preserving payment solution.

14 Tokenomics

The tokenomics of Overpass Channels are designed to ensure network stability, incentivize participation, and maintain a balanced ecosystem. This section outlines the key aspects of the token distribution and utility.

14.1 Fixed Supply and Initial Distribution

Overpass Channels employs a fixed supply model, with all tokens minted at genesis. This approach aligns with the constant balance theorem, facilitating accurate tracking of token movement within the system.

- Total Supply: 100 billion tokens
- Initial Distribution:
  - 70% (70 billion tokens) to be airdropped
  - 20% (20 billion tokens) allocated to Treasury/Governance
  - 10% (10 billion tokens) distributed between the team, investors, and advisors

The airdrop distribution strategy aims to ensure wide token distribution and foster community engagement. Specific details of the airdrop mechanism will be announced at a later date.

14.2 Governance and Treasury

The 20% allocation (20 billion tokens) to governance and treasury functions enables decentralized decision-making and sustainable ecosystem development.

- Governance and Treasury Allocation: 20 billion tokens (20% of total supply)

The treasury, operated by the governance mechanism, plays a crucial role in managing the ecosystem's resources, funding future developments, and ensuring long-term sustainability.

14.3 Token Utility and Fee Structure

Overpass Channels implements a balanced fee structure that supports various components of the network:

Reserve Status

Tokens that are designated as "burned" are not permanently removed from circulation. Instead, they enter a reserve status within the treasury. These reserved tokens serve multiple purposes:

- Provide liquidity for system operations
- Fund future developments and improvements
- Support the ecosystem's long-term sustainability

Off-Chain Storage Node Compensation

A portion of transaction fees is allocated to compensate off-chain storage nodes, which are essential for the Overpass system's efficient operation.

TON Node Compensation

Supporting TON nodes that accept and store the global root also receive a share of the transaction fees.

14.4 Fee Distribution

Transaction fees are divided into three main categories:

1. Recycling to Reserve: A portion returns to the treasury's reserve status for future ecosystem funding and development.
2. Off-Chain Node Payments: Compensates the off-chain storage nodes in the Overpass system.
3. On-Chain Node Payments: A small portion goes to on-chain TON nodes for global root storage and processing.

This balanced approach ensures the sustainability of the network, incentivizes crucial infrastructure providers, and maintains a flexible token supply that can be utilized for the ecosystem's ongoing needs and future growth.

15 Wallet-Managed Channel Grouping

The wallet application (e.g., Tonkeeper) will handle the grouping of channels on the user side, while still using the underlying payment channels for transactions and zk-SNARK proofs on-chain. This approach simplifies things for both the user and the system.

15.1 Key Components of the Wallet Interface

Group-Based UI (Checking, Savings, Custom Accounts)

- The grouping logic is only a UI/UX feature in the wallet.
- Users can assign channels to different groups like Checking, Savings, or Custom at the interface level. However, this grouping will not affect how the channels are structured on-chain.
- The wallet will simply aggregate and display balances from these groupings as a single total balance (or split by groups), without involving smart contracts in the process.

Interaction with Channels

- When a user performs a transaction (sending funds or opening a channel), the wallet will route the transaction through the selected group but keep all channel interactions on-chain as individual channels.
- The underlying contract will remain the same (i.e., a standard payment channel), but the user will be able to choose the group from the wallet interface, making it feel like they are interacting with different "accounts."

15.2 How the Wallet Should Work

- **UI Level Grouping**: Users assign channels to different groups (Checking, Savings, Custom) within the wallet interface. The groups are simply tags or filters applied to the channels, with no on-chain changes required.
- **Total Balance Calculation**: The wallet calculates a total balance from all the payment channels associated with a user. It presents this balance as unified for the user but keeps the actual channel details hidden unless the user wants to see a breakdown.
- **Transaction Process**: When the user initiates a transaction, they select the group (Checking, Savings, or Custom) from which they want to send funds. The wallet then pulls the appropriate funds from the corresponding channels but groups the logic only within the UI.

15.3 Updated Example Code for Wallet-Level Grouping

UI Grouping Module

This module defines how channels are grouped in the wallet without affecting the underlying smart contracts.

```
// groupUIManager.js
class GroupUIManager {
    constructor() {
        this.groups = {
            checking: [],
            savings: [],
            custom: []
        };
    }

    // Assign a channel to a UI group (Checking, Savings, Custom)
    assignChannelToGroup(channelId, groupType) {
        switch (groupType) {
            case 'Checking':
                this.groups.checking.push(channelId);
                break;
            case 'Savings':
                this.groups.savings.push(channelId);
                break;
            case 'Custom':
                this.groups.custom.push(channelId);
                break;
            default:
                throw new Error(`Unknown group type: ${groupType}`);
        }
    }

    // Retrieve group balances for the UI
    async getGroupBalances() {
        const checkingBalance = await this.getGroupBalance('Checking');
        const savingsBalance = await this.getGroupBalance('Savings');
        const customBalance = await this.getGroupBalance('Custom');
        return {
            breakdown: {
                checking: checkingBalance,
                savings: savingsBalance,
                custom: customBalance
            },
            totalBalance: checkingBalance + savingsBalance + customBalance,
        };
    }

    // Example to get the balance of all channels in a group
    async getGroupBalance(groupType) {
        const group = this.groups[groupType.toLowerCase()];
        let total = 0;
        for (const channelId of group) {
            total += await blockchain.getChannelBalance(channelId);
        }
        return total;
    }
}

export default new GroupUIManager();
```

Transaction Flow Module

This module handles the interaction with the blockchain when the user initiates a transaction. It routes the transaction based on the selected group.

```
// transactionManager.js
import groupUIManager from './groupUIManager';

class TransactionManager {
    async initiateTransaction(amount, groupType) {
        const group = await groupUIManager.getGroupBalance(groupType);
        if (group < amount) {
            throw new Error('Insufficient funds in the selected group');
        }
        // Select the appropriate channels and send the transaction via blockchain
        const channels = groupUIManager.groups[groupType.toLowerCase()];
        const transactionResult = await blockchain.submitTransaction(channels,
            ↪ amount);
        return transactionResult;
    }
}

export default new TransactionManager();
```

15.4 Optimized UX/UI

- **Balance Display**: The wallet interface shows the user their total balance across all groups (Checking, Savings, Custom) but also allows them to see detailed group breakdowns if they choose.
- **Transaction Process**: When users initiate transactions, they select a group in the UI, and the wallet handles the details of which channels to use and how much liquidity is available in each.

By keeping the grouping logic in the wallet interface and not on-chain, the complexity is reduced. The wallet simply filters and displays channels as groups, but the underlying wallet contracts handle all the zk-SNARK proofs and rebalancing as needed. The grouping is purely for the user's ease of interaction, and the contract logic remains simple, efficient, and secure.

16 Analysis

16.1 Censorship Resistance in Overpass Channels

The censorship resistance in Overpass Channels is fundamentally different from other systems due to its unique architecture. Let's clarify this:

Theorem 37 (Overpass Channels Censorship Impossibility)**.** *In the Overpass Channels system, it is computationally infeasible for any entity or group of entities to censor individual transactions.*

Proof. The proof proceeds as follows:
1) Transactions in Overpass Channels are processed entirely off-chain within individual channels.
2) The network nodes do not see or process individual transactions. Instead, they only receive:

- Merkle tree roots representing batches of transactions (epochs)
- zk-SNARK proofs verifying the validity of state transitions

3) For a transaction T to be censored, one of the following must occur:

- Prevention of T's inclusion in the Merkle tree.
- Rejection of a valid Merkle root containing T.
- Rejection of a valid zk-SNARK proof for an epoch including T.

4) (3a) is impossible because Merkle tree construction is done locally by channel participants.
5) (3b) is computationally infeasible because:

- Nodes cannot distinguish which transactions are included in a Merkle root.
- Rejecting a valid Merkle root would require rejecting all transactions in an epoch.

6) (3c) is computationally infeasible because:

- zk-SNARK proofs reveal no information about individual transactions.
- Rejecting a valid proof would require rejecting an entire epoch of transactions.

7) Any attempt to censor by rejecting epochs would result in rejecting all transactions, including legitimate ones, which is not in the nodes' interest.

Therefore, censorship of individual transactions in Overpass Channels is computationally infeasible. ■

This proof demonstrates that the Overpass Channels system achieves censorship resistance through its architecture, which makes it impossible for nodes to target individual transactions for censorship. The privacy-preserving nature of the system, combined with the batching of transactions into epochs, ensures that no entity can selectively censor transactions without disrupting the entire network.

Comparison with Other Systems

To further illustrate the unique censorship resistance of Overpass Channels, let's compare it with other systems:

1. **Bitcoin and other PoW blockchains:**
   - Miners can potentially censor transactions by excluding them from blocks.
   - Requires a majority of miners to collude for effective censorship.
2. **Layer 2 solutions (e.g., Rollups):**
   - Sequencers or operators can potentially censor by excluding transactions.
   - Often rely on centralized components that are vulnerable to censorship pressure.
3. **Overpass Channels:**

- Nodes cannot see or process individual transactions.
- Censorship attempts would require rejecting entire epochs of transactions.
- No centralized components that could be targeted for censorship.

Theorem 38 (Overpass Channels Superior Censorship Resistance)**.** *Overpass Channels provides stronger censorship resistance than existing blockchain and Layer 2 solutions.*

Proof. Let $p_c(S)$ be the probability of successful censorship in system S. The proof proceeds as follows:

1) For Bitcoin: $p_c(\text{BTC}) > 0$, as a majority of miners could potentially censor transactions.

2) For Layer 2 solutions: $p_c(\text{L2}) > 0$, due to centralized components vulnerable to censorship.

3) For Overpass Channels:

$$p_c(\text{OC}) \approx 0$$

as censorship would require rejecting entire epochs, which is against the economic interests of nodes.

4) Therefore:

$$p_c(\text{OC}) < p_c(\text{BTC}) \quad \text{and} \quad p_c(\text{OC}) < p_c(\text{L2})$$

Thus, Overpass Channels provides superior censorship resistance compared to existing solutions. ■

16.2 Scalability and Performance

Given the unique architecture of Overpass Channels, it's important to revisit our scalability and performance analysis to ensure accuracy.

Theorem 39 (Overpass Channels Scalability)**.** *The transaction throughput of Overpass Channels scales linearly with the number of channels, with each channel capable of processing transactions independently.*

Proof. Let:

- n be the number of channels in the network
- T_c be the throughput of a single channel
- T_{total} be the total network throughput

The proof proceeds as follows:

1. Each channel operates independently, processing transactions off-chain.
2. The throughput of the network is the sum of the throughputs of all channels:

$$T_{total} = \sum_{i=1}^{n} T_{c_i}$$

3. Assuming an average throughput per channel, this simplifies to:

$$T_{total} = n \cdot T_c$$

4. As n increases, T_{total} increases linearly.
5. The only limitation on n is the capacity of the underlying blockchain to process epoch commitments.

Therefore, Overpass Channels exhibits linear scalability with respect to the number of channels. ■

16.3 Potential Transaction Throughput

Given this architecture, the system's transaction throughput potential is significant:

1. **Channel transactions**: Each unilateral channel can process transactions near-instantaneously, limited only by the user's device capabilities and network latency.
2. **Aggregation at intermediate level**: Intermediate contracts can manage a large number of wallet roots, potentially thousands or tens of thousands.
3. **Root contract aggregation**: The root contract can combine multiple intermediate roots, further increasing the scalability.

16.4 Factors Affecting TPS

Several factors influence the overall transactions per second (TPS) of the system:

1. **ZKP generation time**: The speed of generating and verifying ZKPs on user devices will impact individual channel throughput.
2. **Off-chain storage capacity**: The ability of decentralized storage nodes to handle concurrent updates and queries.
3. **Intermediate contract capacity**: The number of wallet roots each intermediate contract can efficiently manage.
4. **Root contract updates**: The frequency at which the root contract can update the on-chain state.
5. **Blockchain limitations**: The underlying blockchain's capacity to process root contract updates.

16.5 Estimated Throughput

While it's challenging to provide an exact TPS without specific implementation details, we can make some educated estimates:

1. **Per-channel TPS**: Potentially thousands per second, limited by ZKP generation and verification speed.
2. **System-wide TPS**: Millions to billions per second off-chain, depending on the number of active channels and intermediate contracts.
3. **On-chain TPS**: Limited by the underlying blockchain, but each on-chain transaction represents a large number of off-chain transactions.

16.6 System Architecture

The Overpass Channels system architecture consists of several key components:

1. **User-controlled channels**: Each user manages their own unilateral channels via their client-side device.
2. **Sparse Merkle trees**: Used to generate Zero-Knowledge Proofs (ZKPs) for each transaction in the channels.
3. **Smart contract hierarchy**:
   - Wallet contract (parent)
   - Unilateral channel contracts (children)
4. **Off-chain storage**: Decentralized nodes store proofs and Merkle roots.
5. **Intermediate contracts**: Aggregate wallet roots into a single Merkle tree.
6. **Root contract**: Combines all intermediate roots and submits the final root to the blockchain.

16.7 Optimizing the System

To maximize throughput, several optimization strategies can be employed:

1. Optimize ZKP generation and verification algorithms.
2. Implement efficient data structures for Merkle tree management.
3. Design a robust off-chain storage system with high concurrency.
4. Carefully balance the number of channels per intermediate contract.
5. Implement batching and aggregation techniques at each level.

This system has the potential for extremely high off-chain TPS, limited primarily by the computational resources of user devices and the capacity of the off-chain storage network. The on-chain bottleneck is significantly reduced, as only aggregate roots need to be recorded on the blockchain periodically.

17 Efficiency and Cost Analysis

The efficiency of Overpass Channels in terms of cost per transaction and energy consumption is a key advantage over existing systems, while also providing incentives for network participants.

Theorem 40 (Overpass Channels Efficiency)**.** *Overpass Channels achieves a lower cost per transaction and energy consumption compared to traditional blockchain systems and most Layer 2 solutions, while maintaining a balanced fee structure for network sustainability.*

Proof. Let:

- C_{OC} be the cost per transaction in Overpass Channels

- E_{OC} be the energy consumption per transaction in Overpass Channels
- N be the number of transactions in an epoch
- $C_{on-chain}$ be the cost of an on-chain epoch commitment
- $E_{off-chain}$ be the energy consumption of an off-chain transaction
- $F_{storage}$ be the fee for off-chain storage nodes
- F_{TON} be the fee for participating TON nodes
- F_{burn} be the burn fee allocated to the treasury

The proof proceeds as follows:
1) Cost per transaction in Overpass Channels:

$$C_{OC} = \frac{C_{on-chain}}{N} + F_{storage} + F_{TON} + F_{burn} + \epsilon$$

where ϵ is the negligible cost of off-chain processing.
2) Energy consumption per transaction:

$$E_{OC} = E_{off-chain} + \frac{E_{on-chain}}{N}$$

where $E_{on-chain}$ is the energy cost of an on-chain epoch commitment.
3) As N increases:

$$\lim_{N \to \infty} C_{OC} = F_{storage} + F_{TON} + F_{burn} + \epsilon$$

$$\lim_{N \to \infty} E_{OC} = E_{off-chain}$$

4) In comparison:

- Traditional blockchains incur costs and energy consumption for every transaction.
- Layer 2 solutions with centralized components have overhead for operator maintenance and profit.

5) Therefore, for sufficiently large N:

$$C_{OC} < C_{traditional} \text{ and } C_{OC} < C_{Layer2}$$

$$E_{OC} < E_{traditional} \text{ and } E_{OC} < E_{Layer2}$$

Thus, Overpass Channels achieves higher efficiency in terms of cost and energy consumption per transaction, while also providing incentives for storage nodes, TON nodes, and maintaining a treasury through the burn fee. ■

17.1 Fee Distribution

The fee structure in Overpass Channels is designed to incentivize network participants and ensure long-term sustainability:

- **Storage Node Fee ($F_{storage}$)**: Compensates off-chain storage nodes for maintaining transaction data and proofs.
- **TON Node Fee (F_{TON})**: Rewards TON nodes that participate in submitting the global Merkle root.
- **Burn Fee (F_{burn})**: Allocated to the treasury for future ecosystem development and sustainability.

This balanced fee structure ensures that all critical components of the network are incentivized, while maintaining overall efficiency. The exact proportions of these fees can be adjusted based on network requirements and governance decisions.

18 Privacy Analysis

The privacy guarantees of Overpass Channels are fundamental to its design and operation.

Theorem 41 (Overpass Channels Transaction Privacy)**.** *In Overpass Channels, no information about individual transactions is revealed to the network beyond what is explicitly included in zk-SNARK proofs.*

Proof. The proof proceeds as follows:

1. Transactions are processed entirely off-chain within channels.
2. The only information shared with the network is:
   (a) Merkle roots of transaction batches (epochs)
   (b) zk-SNARK proofs of state transitions
3. Merkle roots reveal no information about individual transactions due to the pre-image resistance of cryptographic hash functions.
4. zk-SNARK proofs, by definition, reveal no information beyond the validity of the statement being proved.
5. The statements proved by zk-SNARKs in Overpass Channels are of the form: "There exists a valid set of transactions that transition the state from S_1 to S_2."
6. This statement does not reveal any information about individual transactions, their amounts, or the parties involved.

Therefore, Overpass Channels provides strong transaction privacy, revealing no information about individual transactions to the network. ■

19 Remarks

Overpass Channels represents a significant advancement in blockchain scaling solutions, offering:

1. Superior censorship resistance through its unique architecture that makes individual transaction censorship computationally infeasible.
2. Exceptional scalability, with the throughput reaching millions or even billions in TPS.
3. High efficiency in terms of cost per transaction and energy consumption, approaching zero marginal cost for off-chain transactions.
4. Strong privacy guarantees, ensuring that no information about individual transactions is revealed to the network.

20 Integration with TON Blockchain

The integration of Overpass Channels with the TON (The Open Network) blockchain further enhances its capabilities and provides a robust foundation for large-scale deployment. This section examines the synergies between Overpass Channels and TON, and how this integration addresses key challenges in blockchain scalability and usability.

20.1 TON's Sharding Architecture

TON's multi-threaded, sharded architecture complements the scalability features of Overpass Channels.

Theorem 42 (Overpass-TON Scalability Synergy)**.** *The integration of Overpass Channels with TON's sharding architecture results in multiplicative scalability improvements.*

Proof. Let:

- n be the number of Overpass Channels
- m be the number of shards in TON
- T_{OC} be the throughput of a single Overpass Channel
- T_{TON} be the throughput of a single TON shard

The proof proceeds as follows:

1. The total throughput of Overpass Channels: $T_{total_{OC}} = n \cdot T_{OC}$
2. The total throughput of TON: $T_{total_{TON}} = m \cdot T_{TON}$
3. With integration, each TON shard can support multiple Overpass Channels:

$$T_{integrated} = m \cdot (k \cdot T_{OC})$$

where k is the number of Overpass Channels per shard.

4. Assuming even distribution of channels across shards:

$$k = \frac{n}{m}$$

5. Therefore:

$$T_{integrated} = m \cdot \left(\frac{n}{m} \cdot T_{OC} \right) = n \cdot T_{OC}$$

6. This means that the integrated system can fully utilize both the scalability of Overpass Channels and TON's sharding.

Thus, the integration results in multiplicative scalability improvements, allowing the system to scale with both the number of channels and the number of shards. ■

20.2 Smart Contract Integration

TON's flexible smart contract system allows for efficient implementation of Overpass Channels' core components.

Algorithm 25 Overpass Channel Smart Contract on TON

procedure ChannelContract
  State Variables:
  $channelId \leftarrow$ UniqueIdentifier
  $participants \leftarrow \{\text{ParticipantA}, \text{ParticipantB}\}$
  $currentEpoch \leftarrow 0$
  $latestMerkleRoot \leftarrow$ InitialMerkleRoot
  $stateHash \leftarrow$ InitialStateHash
  procedure SubmitEpoch($newMerkleRoot, zkProof, epochNumber$)
Require: $msg.sender \in participants$
Require: $epochNumber = currentEpoch + 1$
    assert VerifyZKProof($zkProof, latestMerkleRoot, newMerkleRoot, stateHash$)
    $latestMerkleRoot \leftarrow newMerkleRoot$
    $currentEpoch \leftarrow epochNumber$
    emit EpochSubmitted($channelId, currentEpoch, newMerkleRoot$)
  end procedure
  procedure CloseChannel($finalStateProof$)
Require: $msg.sender \in participants$
    assert VerifyFinalStateProof($finalStateProof, stateHash$)
    $finalState \leftarrow ExtractFinalState(finalStateProof)$
    DistributeFunds($finalState$)
    emit ChannelClosed($channelId, finalState$)
  end procedure
end procedure

This smart contract structure allows for efficient on-chain management of Overpass Channels, leveraging TON's capabilities for fast execution and low-cost state updates.

20.3 Cross-Shard Operations

TON's ability to handle cross-shard communications efficiently enhances Overpass Channels' capability to process inter-channel transactions.

Theorem 43 (Efficient Cross-Shard Overpass Transactions)**.** *Cross-shard transactions in the integrated Overpass-TON system can be processed with $O(\log m)$ communication complexity, where m is the number of shards.*

Proof. Let:

- m be the number of shards
- $T_{A,B}$ be a transaction from channel A in shard S_A to channel B in shard S_B

The proof proceeds as follows:

1. The cross-shard transaction process involves:
   (a) Generating a zk-SNARK proof in shard S_A
   (b) Transmitting the proof to shard S_B
   (c) Verifying the proof in shard S_B
2. TON's hypercube routing for cross-shard messages ensures that the number of hops for a message to travel between any two shards is $O(\log m)$.
3. The size of the zk-SNARK proof is constant, regardless of the transaction details.
4. Therefore, the communication complexity for transmitting the proof is $O(\log m)$.
5. Proof generation and verification times are constant and independent of the number of shards.

Thus, the overall complexity of cross-shard Overpass transactions in TON is $O(\log m)$, ensuring efficient scaling even with a large number of shards. ■

20.4 TON DNS Integration

Integration with TON DNS allows for human-readable addresses in Overpass Channels, enhancing usability.

This integration simplifies the user experience by allowing easy-to-remember addresses for Overpass Channels, similar to domain names on the internet.

21 Use Cases for Privacy

The enhanced privacy features of Overpass Channels, leveraging zk-SNARKs, enable a wide range of applications that require confidentiality while maintaining transparency and verifiability. This section explores four key use cases that demonstrate the versatility and potential impact of the system.

Algorithm 26 Overpass Channel Address Resolution via TON DNS

```
procedure ResolveChannelAddress(humanReadableAddress)
    dnsRecord ← TON.DNS.Resolve(humanReadableAddress)
    if dnsRecord contains OverpassChannelID then
        channelId ← dnsRecord.OverpassChannelID
        channelContract ← GetChannelContract(channelId)
        return channelContract.address
    else
        return null
    end if
end procedure
```

21.1 Confidential Voting Systems

Confidential voting systems are a critical application for enhanced privacy in decentralized networks. Overpass Channels can provide a solution that guarantees the confidentiality of votes while allowing for public verification of election results.

Implementation on TON

1. **Voter Registration**: A smart contract on TON manages voter registration, issuing unique identifiers to eligible voters.
2. **Vote Casting**: Voters use their Overpass Channel to cast votes, generating a zk-SNARK proof that:
   - They are an eligible voter
   - They have not voted before
   - Their vote is for a valid candidate

   The proof is submitted to a voting contract on TON.
3. **Vote Tallying**: The voting contract aggregates votes without revealing individual choices. A final zk-SNARK proof is generated to prove the correctness of the tally.
4. **Result Verification**: Anyone can verify the final tally by checking the zk-SNARK proof on the TON blockchain.

21.2 Private Asset Transfers

Private asset transfers are another significant use case for enhanced privacy in a decentralized network. Overpass Channels enable confidential transfers while maintaining the integrity and verifiability of transactions.

Implementation on TON

1. **Asset Tokenization**: Assets are represented as tokens on the TON blockchain.
2. **Private Transfers**: Users transfer tokens through Overpass Channels, generating zk-SNARK proofs that:
   - The sender owns the tokens
   - The transfer amount is valid (non-negative and within the sender's balance)
   - The receiver is a valid recipient
3. **On-chain Settlement**: Periodically, the aggregate state of transfers is settled on-chain through a smart contract, updating token balances without revealing individual transaction details.
4. **Audit Capability**: While individual transactions are private, users can generate proofs of their transaction history for auditing purposes when required.

21.3 Secure Health Records Management

Secure health records management is an essential use case where sensitive health information must be kept confidential while ensuring that authorized parties can verify the records.

Implementation on TON

1. **Record Storage**: Health records are stored off-chain, with hashes of the records stored in Overpass Channels.
2. **Access Control**: Smart contracts on TON manage access rights to health records.
3. **Data Sharing**: When sharing data, a zk-SNARK proof is generated to prove:
   - The data belongs to the claimed patient
   - The recipient has the right to access the data
   - The data has not been tampered with
4. **Verification**: Authorized parties can verify the authenticity and integrity of the records on the TON blockchain without accessing the actual content.

21.4 Global Payment System

A global payment system is perhaps the most scalable and impactful use case for Overpass Channels, providing sufficient privacy to protect user transactions while ensuring transparency and scalability to facilitate mass adoption.

Implementation on TON

1. **Multi-Currency Support**: The system supports multiple currencies, with exchange rates managed by oracle contracts on TON.
2. **Cross-Border Transfers**: Users can make cross-border payments through Overpass Channels, with zk-SNARKs proving:
   - The sender has sufficient funds
   - The currency conversion is accurate
   - Compliance with relevant regulations (e.g., AML checks)
3. **Liquidity Pools**: Smart contracts on TON manage liquidity pools to facilitate instant currency conversions.
4. **Regulatory Compliance**: The system includes built-in compliance features, such as generating confidential reports for regulators without compromising user privacy.
5. **Integration with Traditional Finance**: Bridge contracts on TON facilitate interaction with traditional financial systems, allowing for deposits and withdrawals while maintaining privacy for on-network transactions.

These use cases illustrate how Overpass Channels can achieve a balance between privacy and transparency, facilitating mass adoption while maintaining the necessary confidentiality. By leveraging zk-SNARKs, the system provides enhanced privacy and scalability, making it suitable for a wide range of applications in various sectors.

22 Future Directions and Challenges

While Overpass Channels on TON presents a powerful solution for scalable, private, and censorship-resistant payments, several areas warrant further research and development:

22.1 Post-Quantum Security

As quantum computing advances, ensuring the long-term security of Overpass Channels becomes crucial.

Theorem 44 (Post-Quantum Vulnerability)**.** *The current zk-SNARK implementations used in Overpass Channels are vulnerable to quantum attacks.*

Proof. 1. Current zk-SNARK constructions rely on discrete logarithm or elliptic curve discrete logarithm problems.

2. Shor's algorithm, when implemented on a sufficiently powerful quantum computer, can solve these problems in polynomial time.

3. Therefore, a quantum computer could potentially break the security of current zk-SNARK implementations.

■

Future research should focus on developing and implementing post-quantum secure zero-knowledge proof systems that maintain the efficiency and privacy guarantees of current zk-SNARKs.

22.2 Privacy-Preserving Analytics

While Overpass Channels provide strong transaction privacy, this can make it challenging to gather network-wide analytics, which can be crucial for optimizing performance and detecting potential issues.

Theorem 45 (Privacy-Analytics Tradeoff)**.** *There exists a fundamental tradeoff between transaction privacy and the ability to perform network-wide analytics in Overpass Channels.*

Research into privacy-preserving data analysis techniques, such as secure multi-party computation or differential privacy, could help address this challenge without compromising the core privacy guarantees of the system.

23 Conclusion

Overpass Channels, integrated with the TON blockchain, represents a significant advancement in blockchain scalability, privacy, and censorship resistance. By leveraging zk-SNARKs, off-chain processing, and TON's sharding architecture, it offers a solution that can potentially scale to global payment network levels while maintaining strong security and privacy guarantees.

The system's ability to process millions, if not billions, of transactions per second, with linear scalability as nodes are added, positions it as a highly competitive solution in the blockchain space. Its unique architecture makes censorship of individual transactions computationally infeasible, providing robust protection against potential attacks or regulatory pressures.

Furthermore, the integration with TON's flexible smart contract system, efficient cross-shard communication, and user-friendly features like DNS integration creates a comprehensive ecosystem for next-generation decentralized applications.

As the blockchain industry continues to evolve, solutions like Overpass Channels on TON pave the way for a future where decentralized, private, and efficient global payment systems become a reality, potentially revolutionizing financial interactions on a global scale.

Bibliography

[1] Poon, J., & Dryja, T. (2016). The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments. Lightning Network Whitepaper. https://lightning.network/lightning-network-paper.pdf

[2] Buterin, V., & Poon, J. (2017). Plasma: Scalable Autonomous Smart Contracts. Plasma Whitepaper. https://plasma.io/plasma.pdf

[3] Raiden Network Team. (2017). Raiden Network: Fast, Cheap, Scalable Token Transfers for Ethereum. Raiden Network

[4] Celer Network. (2019). Celer Network: Bring Internet Scale to Every Blockchain. Celer Network Whitepaper. https://www.celer.network/doc/CelerNetwork-Whitepaper.pdf

[5] PLONK Documentation. (n.d.). ZK-SNARKs: PLONK. Retrieved from https://docs.plonk.cafe/

[6] Ben-Sasson, E., Chiesa, A., Tromer, E., & Virza, M. (2014). Scalable Zero-Knowledge via Cycles of Elliptic Curves. In International Cryptology Conference (pp. 276-294). Springer, Berlin, Heidelberg.

[7] Groth, J. (2016). On the Size of Pairing-based Non-interactive Arguments. In Annual International Conference on the Theory and Applications of Cryptographic Techniques (pp. 305-326). Springer, Berlin, Heidelberg.

[8] Zhang, F., Cecchetti, E., Croman, K., Juels, A., & Shi, E. (2016). Town Crier: An Authenticated Data Feed for Smart Contracts. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (pp. 270-282).

[9] Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E., & Virza, M. (2015). SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge. In Annual Cryptology Conference (pp. 90-108). Springer, Berlin, Heidelberg.

[10] The Open Network, based on the work of Dr. Nikolai Durov. (2021). The Open Network Whitepaper. Retrieved from https://ton.org/whitepaper.pdf

[11] Hioki, L., Dompeldorius, A., & Hashimoto, Y. (n.d.). Plasma Next: Plasma without Online Requirements. Ethereum Research. Retrieved from https://ethresear.ch/t/plasma-next-plasma-without-online-requirements/18786

24 Appendix

1 Circuits

The Overpass Channels system uses hierarchical circuits to manage transactions, state updates, and global consistency. These circuits are organized into individual channels, intermediate contracts, and root contracts. Below is an explanation of how individual channels work, along with the related algorithms.

Channel 1: From Party A to Party B

In this section, we will describe the setup of a unilateral channel from Party A to Party B, followed by the process of executing a transaction and verifying the result.

Unilateral Channel Setup The setup for a unilateral payment channel involves locking the funds and verifying the signatures from both parties. The following algorithm explains how the setup works.

```
Algorithm 1: Unilateral Channel Setup
1. Inputs: Funding Party (A), Receiving Party (B), Initial Balance (B_AB1),
           Channel ID (id_AB1), Signatures (\sigma_A, \sigma_B)
2. Initialize channel with balances:
    B_A = B_AB1, B_B = 0, nonce = 0
3. PoseidonHash the channel ID and initial balance:
    h_channel1 = PoseidonHash(id_AB1, B_AB1)
4. Verify signatures from both parties:
    SignatureVerify(\sigma_A, A)
    SignatureVerify(\sigma_B, B)
5. Generate recursive zk-SNARK proof \pi_setup1
6. Output: zk-SNARK result C_setup1
```

In this process:

1. Initializes the payment channel with the balance from Party A.
2. Hashes the channel's initial data using the Poseidon hashing algorithm.
3. Verifies that both parties have signed and acknowledged the setup.

Transaction Execution Once the channel is set up, Party A can send funds to Party B through a transaction. The following algorithm describes the transaction process.

```
Algorithm 2: Transaction Execution from A to B
1. Inputs: Channel ID (id_AB1), Current Balances (B_A, B_B),
           Transaction Amount (\Delta_1), Nonce (n_1), Signature (\sigma_A)
2. PoseidonHash the current state:
    h_state1 = PoseidonHash(id_AB1, B_A, B_B, n_1)
3. Update balances:
```

```
    B'_A = B_A - \Delta_1
    B'_B = B_B + \Delta_1
4. Verify balance conservation:
    B'_A + B'_B = B_A + B_B
5. Ensure 50\% spending rule:
    \Delta_1 \leq \frac{B_A}{2}
6. Increment nonce:
    n'_1 = n_1 + 1
7. Verify signature from A:
    SignatureVerify(\sigma_A, A)
8. Generate zk-SNARK proof \pi_transaction1
9. Output: zk-SNARK result C_transaction1
```

This algorithm handles the transaction between Party A and Party B, ensuring that:

1. The current state is hashed to preserve consistency of the channel data.
2. The balance conservation rule and the 50% spending rule are both followed.
3. The sender's signature is verified before generating the zk-SNARK proof.

Transaction Acceptance Once Party A has executed the transaction, Party B must accept it. The following algorithm explains the acceptance procedure.

```
Algorithm 3: Transaction Acceptance by Party B
1. Inputs: New State (id_AB1, B'_A, B'_B, n'_1),
           Transaction Proof (\pi_transaction1), Signature (\sigma_B)
2. Verify zk-SNARK proof \pi_transaction1
3. Verify signature from B on the new state:
    SignatureVerify(\sigma_B, B)
4. Update the channel contract with the new state
5. Remove pending status from the transaction
6. Output: Channel state is updated and transaction is finalized
```

During transaction acceptance:

1. The zk-SNARK proof generated by Party A is verified.
2. Party B's signature on the new state is validated.
3. The channel contract is updated, and the transaction is finalized.

1.1 Channel State Verification for Both Channels

Both channels independently verify their states using Merkle proofs to ensure their integrity and consistency. Each channel maintains a Merkle root that reflects the current state of the balances and transactions.

Closure for Channel 1 ($A \to B$)

```
Algorithm: Closure for Channel 1
1. Inputs:
   - Channel Identifier (id_AB1)
   - Final Balances (B_A, B_B)
   - Final Signatures (\sigma_A, \sigma_B)
   - Merkle Proof for Channel State (\pi_Merkle1)
2. PoseidonHash the final state:
   h_final1 = PoseidonHash(id_AB1, B_A, B_B)
3. Verify that the final balances match the expected total:
   B_A + B_B must equal the total recorded balance.
4. Verify signatures from both parties:
   SignatureVerify(\sigma_A, A)
   SignatureVerify(\sigma_B, B)
5. Verify the Merkle proof \pi_Merkle1 to ensure the correct state closure.
6. Generate zk-SNARK proof \pi_closure1 for closing the channel.
7. Output: zk-SNARK result C_closure1
```

In this process:

1. The final state of the channel is hashed using the Poseidon algorithm, ensuring that the balances and channel identifier are encoded securely.
2. The balance verification ensures that the final balances match the expected total, preventing any discrepancy before closure.
3. Both parties' signatures are verified, ensuring mutual agreement on the channel closure.
4. The Merkle proof confirms that the state being closed is the correct and most up-to-date state.
5. Finally, a zk-SNARK proof is generated, which will confirm the validity of the closure.

1.2 Intermediate Contract Circuits

Rebalancing Circuit

```
Algorithm: Rebalancing Circuit
1. Inputs:
   - Channel Identifier (id)
   - Current Balance Distribution (B_A, B_B)
   - Desired Balance Distribution (B'_A, B'_B)
   - Rebalancing Amount (\Delta_rebalance)
   - Signatures (\sigma_A, \sigma_B)
   - Current Shard Merkle Root (R_M_shard)
2. PoseidonHash the balances before and after rebalancing:
   h_before = PoseidonHash(id, B_A, B_B)
   h_after  = PoseidonHash(id, B'_A, B'_B)
```

```
3. Verify balance conservation:
   B_A + B_B = B'_A + B'_B
4. Calculate rebalancing amount:
   \Delta_rebalance = |B'_A - B_A| = |B'_B - B_B|
5. Update the shard's Merkle tree with the new state:
   R_M_shard_new = UpdateMerkleRoot(R_M_shard, h_after)
6. Generate zk-SNARK proof \pi_rebalance showing rebalancing and agreement.
7. Output: zk-SNARK proof \pi_rebalance and updated Merkle root R_M_shard_new.
```

In this process:

1. The balances are hashed using Poseidon before and after rebalancing to ensure integrity.
2. Balance conservation is checked to guarantee that the total funds are preserved during rebalancing.
3. The rebalancing amount is calculated to determine how much to adjust the balances.
4. The shard's Merkle tree is updated to reflect the new state after rebalancing.
5. A zk-SNARK proof is generated, confirming that the rebalancing occurred as expected.

Cross-Channel Communication Circuit

Both channels independently verify their states using Merkle proofs to ensure their integrity and consistency. Each channel maintains a Merkle root that reflects the current state of the balances and transactions.

```
Algorithm: Cross-Channel Communication Circuit
1. Inputs:
   - Source Channel State (S_source)
   - Destination Channel State (S_destination)
   - Transaction Amount (\Delta)
   - Merkle Proofs for Both Channels (\pi_Merkle_source, \pi_Merkle_destination)
   - Current Shard Merkle Root (R_M_shard)
2. Verify Merkle proofs of both source and destination states:
   VerifyMerkleProof(R_M_shard, \pi_Merkle_source, S_source)
   VerifyMerkleProof(R_M_shard, \pi_Merkle_destination, S_destination)
3. Ensure sufficient funds in source channel:
   B_source \geq \Delta
4. Update balances:
   B_source_new = B_source - \Delta
   B_destination_new = B_destination + \Delta
5. Generate zk-SNARK proof \pi_cross_channel, verifying transaction validity.
6. Compute new shard Merkle root:
   R_M_shard_new = UpdateMerkleRoot(R_M_shard, S_source_new, S_destination_new)
7. Output: zk-SNARK proof \pi_cross_channel and updated Merkle root R_M_shard_new
```

In this process:

1. The Merkle proofs for the source and destination channels are verified to ensure inclusion in the current shard state.
2. The system checks if the source channel has enough balance to execute the transaction.
3. Balances in both the source and destination channels are updated accordingly.
4. A zk-SNARK proof is generated to confirm the transaction's validity.
5. Finally, the Merkle root is updated to reflect the new state of both channels.

Shard State Submission Circuit

This circuit handles the submission of the shard's updated state to the root contract, ensuring all updates and transactions are reflected in the global state.

```
Algorithm: Shard State Submission Circuit
1. Inputs:
   - Current Shard Merkle Root (R_M_shard)
   - New Shard Merkle Root (R_M_shard_new)
   - Set of Aggregated Update Proofs (\pi_aggregation1, \pi_aggregation2, ...)
   - Cross-Channel Transaction Proofs (\pi_cross_channel1, \pi_cross_channel2, ...)
2. Verify all aggregated update proofs:
   For each \pi_aggregation_i:
       VerifyProof(\pi_aggregation_i)
3. Verify all cross-channel transaction proofs:
   For each \pi_cross_channel_j:
       VerifyProof(\pi_cross_channel_j)
4. Recalculate the shard's Merkle root based on all updates:
   R_M_shard_final = RecalculateMerkleRoot(R_M_shard, all updates)
5. Generate zk-SNARK proof \pi_shard_update, verifying the correct shard state update.
6. Output: zk-SNARK proof \pi_shard_update and final shard Merkle root R_M_shard_final
```

In this process:

1. Aggregated update proofs are verified to ensure all individual state transitions are valid.
2. Cross-channel transaction proofs are also verified for integrity.
3. The shard's Merkle root is recalculated based on all the updates and transactions within the shard.
4. A zk-SNARK proof is generated to confirm the correct update of the shard's state.
5. Finally, the final Merkle root and proof are submitted to the root contract.

1.3 Summary of Intermediate Contract Circuit Responsibilities

1. **Rebalancing Circuit**: Manages liquidity by rebalancing channel balances and recalculating Merkle roots.
2. **State Aggregation Circuit**: Aggregates multiple channel updates in a shard into one efficient zk-SNARK proof.
3. **Cross-Channel Communication Circuit**: Facilitates transactions between channels within the same shard.
4. **Shard State Submission Circuit**: Consolidates all shard updates and prepares the final state for submission to the root contract.

Global State Update Circuit

This circuit ensures that all shard updates are aggregated and the global state is updated consistently across the entire network.

```
Algorithm: Global State Update Circuit
1. Inputs:
   - Current Global Merkle Root (R_M_global)
   - Set of Shard Merkle Roots (R_M_shard1, R_M_shard2, ...)
   - Set of Shard Update Proofs (\pi_shard_update1, \pi_shard_update2, ...)
2. Merkle Tree Verification:
   For each shard i:
       h_i = PoseidonHash(R_M_shardi)
       VerifyMerkleProof(R_M_global, h_i, proof_i)
3. Balance Consistency:
   Verify total balance consistency:
       sum(B_shardi) = B_total
   Ensure no overflows:
       forall i, B_shardi < 2^64
4. Recursive zk-SNARK Proof Generation:
   \pi_global_update = Plonky2Generate(\pi_shard_update1, \pi_shard_update2, ...)
5. Global Merkle Root Update:
   R_M_global_new = UpdateGlobalRoot(R_M_shard1, R_M_shard2, ...)
6. Output: zk-SNARK proof \pi_global_update and updated global Merkle root R_M_global_new
```

In this process:

1. Each shard's Merkle root is verified and included in the global Merkle tree.
2. The total balance across all shards is checked to ensure consistency, and overflows are prevented.
3. A zk-SNARK proof is generated to validate the state updates across all shards.
4. The global Merkle root is updated to reflect the new state of the network.

Cross-Shard Transaction Verification Circuit

This circuit processes transactions between shards, ensuring balances are updated in both the source and destination shards.

```
Algorithm: Cross-Shard Transaction Verification Circuit
1. Inputs:
   - Current Global Merkle Root (R_M_global)
   - Source Shard Merkle Root (R_M_source)
   - Destination Shard Merkle Root (R_M_destination)
   - Cross-Shard Transaction Proof (\pi_cross_shard)
   - Transaction Details (sender, receiver, amount)
2. Merkle Tree Verification:
   Hash source and destination shards:
       h_source = PoseidonHash(R_M_source)
       h_dest = PoseidonHash(R_M_destination)
   Verify Merkle proofs:
       VerifyMerkleProof(R_M_global, h_source, proof_source)
       VerifyMerkleProof(R_M_global, h_dest, proof_dest)
3. Balance Validation:
   Check sender balance:
       B_sender >= amount
   Update balances:
       B_sender_new = B_sender - amount
       B_receiver_new = B_receiver + amount
4. Recursive zk-SNARK Proof Generation:
   \pi_cross_shard = Plonky2Generate(transaction_details, R_M_source, R_M_destination)
5. Global Merkle Root Update:
   R_M_global_new = UpdateGlobalRoot(R_M_global, R_M_source_new, R_M_destination_new)
6. Output: zk-SNARK proof \pi_cross_shard and updated global Merkle root R_M_global_new
```

In this process:

1. The source and destination shard states are verified through their respective Merkle proofs.
2. The sender's balance is validated, ensuring they have sufficient funds for the transaction.
3. The balances for both the sender and receiver are updated.
4. A zk-SNARK proof is generated to confirm the validity of the cross-shard transaction.
5. The global Merkle root is updated to reflect the changes in both shards.

Global State Consistency Check Circuit

This circuit checks the consistency of the global state across all shards, ensuring that no shard has an inconsistent state and preventing double-spending.

```
Algorithm: Global State Consistency Check Circuit
1. Inputs:
   - Current Global Merkle Root (R_M_global)
   - Set of Channel States (S1, S2, ..., SN)
   - Merkle Proofs for Each Channel State (\pi_Merkle_channel1, \pi_Merkle_channel2, ...)
2. Merkle Tree Verification:
   For each channel state Si:
       h_i = PoseidonHash(S_i)
       VerifyMerkleProof(R_M_global, h_i, \pi_Merkle_channeli)
3. State Validation:
   For each channel i:
       B_i >= 0
       B_i < 2^64
       SequenceNumber_i >= PreviousSequenceNumber_i
4. Recursive Proof Generation:
   \pi_global_consistency = Plonky2Generate(S1, S2, ..., SN, R_M_global)
5. Output: zk-SNARK proof \pi_global_consistency and Boolean result (IsConsistent)
```

In this process:

1. Each channel state is verified using its Merkle proof and checked for inclusion in the global Merkle tree.
2. Balances and sequence numbers are validated to ensure no channel is in an inconsistent state.
3. A zk-SNARK proof is generated to verify the overall global state consistency.
4. A Boolean output indicates whether the global state is consistent.

Epoch Transition Circuit

This circuit handles periodic state transitions by processing all transactions within an epoch and updating the global state.

```
Algorithm: Epoch Transition Circuit
1. Inputs:
   - Current Epoch Number (E)
   - Current Global Merkle Root (R_M_global)
   - Set of Transactions for Current Epoch (T1, T2, ..., TM)
   - Set of Transaction Proofs (\pi1, \pi2, ..., \piM)
2. Merkle Tree Verification:
   For each transaction Ti:
       h_i = PoseidonHash(T_i)
       VerifyMerkleProof(R_M_global, h_i, proof_i)
3. Transaction Validation:
   For each transaction Ti:
       B_sender >= amount_i
       B_sender_new = B_sender - amount_i
```

```
       B_receiver_new = B_receiver + amount_i
4. zk-SNARK Proof Generation:
   \pi_epoch = Plonky2Generate(T1, T2, ..., TM, \pi1, \pi2, ..., \piM)
5. Global State Update:
   R_M_global_new = UpdateGlobalRoot(R_M_global, T1, T2, ..., TM)
   Increment epoch: E_new = E + 1
6. Output: zk-SNARK proof \pi_epoch, updated global Merkle root R_M_global_new, new epoch number E_n
```

In this process:

1. Each transaction in the current epoch is verified and included in the global state.
2. The sender's and receiver's balances are updated according to each transaction.
3. A zk-SNARK proof is generated to validate the epoch's transactions.
4. The global Merkle root is updated, and the epoch number is incremented.

Conditional Payments Circuit

Objective: Facilitate conditional payments between parties, where the payment is released only when certain conditions, such as the revelation of a preimage or the completion of a task, are met. This leverages zk-SNARKs to verify the conditions off-chain while maintaining on-chain trustlessness.

```
Algorithm: Conditional Payments Circuit
1. Inputs:
   - Sender (S)
   - Receiver (R)
   - Payment Amount (P)
   - Condition (C), e.g., a preimage X such that H(X) = H_target
   - zk-SNARK Proof (\pi_condition) for condition fulfillment
2. Hash Condition Verification:
   Verify that the condition holds:
      h_condition = PoseidonHash(H_target, X)
      Assert h_condition == H_target
3. Balance Update (Goldilocks):
   Check if the sender has enough funds:
      B_S >= P
   Update balances:
      B_S_new = B_S - P
      B_R_new = B_R + P
4. Recursive zk-SNARK Proof Generation:
   \pi_conditional_payment = Plonky2Generate(C, \pi_condition, B_S, B_R)
5. Output: zk-SNARK proof \pi_conditional_payment and updated balances (B_S_new, B_R_new)
```

In this process:

1. The condition for the payment, such as revealing a correct preimage, is verified using the Poseidon hashing algorithm.

2. The sender's balance is checked to ensure sufficient funds for the payment.

3. If the condition is met, the balances for both the sender and receiver are updated.

4. A zk-SNARK proof is generated to verify the correct execution of the conditional payment.

HTLC (Hashed Time-Lock Contract) Circuit

Objective: Implement a time-locked conditional payment system where the receiver can claim funds only by revealing the correct preimage within a specified time period. If the receiver fails to do so, the funds are returned to the sender.

```
Algorithm: HTLC Circuit
1. Inputs:
   - Sender (S)
   - Receiver (R)
   - Payment Amount (P)
   - Hash Condition H_target
   - Timeout (T)
   - zk-SNARK Proof (\pi_htlc)
2. Hash Condition Verification:
   Verify the preimage:
      Assert PoseidonHash(X) == H_target
3. Timeout Check:
   If timeout T has expired, revert the payment:
      If current_time > T, refund P to S
4. Balance Update (Goldilocks):
   If condition is met and within time limit:
      B_S_new = B_S - P
      B_R_new = B_R + P
   If timeout:
      B_S_new = B_S
      B_R_new = B_R
5. Recursive zk-SNARK Proof Generation:
   \pi_htlc = Plonky2Generate(H_target, X, T, current_time)
6. Output: zk-SNARK proof \pi_htlc and updated balances
```

In this process:

1. The receiver must reveal the correct preimage that hashes to the target hash using Poseidon hashing.

2. The circuit checks whether the timeout has been reached before processing the payment.

3. If the condition is met within the time limit, the balances are updated; otherwise, the funds are returned to the sender.

4. A zk-SNARK proof is generated to confirm the correct execution of the HTLC.

2 Decentralized Exchange (DEX) on Overpass

The Overpass Channels-based decentralized exchange (DEX) employs a hybrid architecture that integrates both on-chain and off-chain components to ensure scalability, efficiency, and security. By leveraging zk-SNARKs for every individual action and Sparse Merkle Trees (SMTs) for state tracking, the system allows for private and secure off-chain trading while maintaining cryptographic proofs for all operations. This architecture does not rely on traditional consensus mechanisms or rollups; instead, it uses overpass zk-SNARKs for each state transition, eliminating the need for trust between participants and allowing instant validation of actions.

2.1 Overview of the System Components

The system consists of three primary components:

- **On-Chain Hub Contract**: Manages on-chain operations like liquidity deposits, channel opening/closing, and final settlement.
- **Off-Chain Router**: Handles off-chain operations, including order book management, order matching, and state tracking using zk-SNARKs. The router is also responsible for managing its own Sparse Merkle Tree (SMT) to store off-chain state updates.
- **zk-SNARK Circuits**: Used for every state transition in the system, ensuring that all actions (e.g., placing orders, executing trades, updating balances) are cryptographically secure.

This section explores how each component interacts, how zk-SNARKs and SMTs are used for security, and how the off-chain router functions as an intermediary for state management.

2.2 On-Chain Hub Contract

The on-chain hub contract serves as the core on-chain component, ensuring that liquidity providers (LPs) can deposit assets, traders can open and close payment channels, and all state changes are cryptographically verified before being settled on-chain. The hub contract interacts with the off-chain router to finalize state transitions after they have been validated using zk-SNARKs.

Key Functions of the On-Chain Hub Contract:

- **Liquidity Management**: LPs can deposit assets directly into the hub contract to provide liquidity to the DEX. These assets are stored securely on-chain and can be withdrawn when an LP chooses to exit the pool.
- **Payment Channel Opening and Closing**: Traders open payment channels with the hub contract, allowing them to engage in off-chain trading. When a trader decides to settle their channel, the hub contract ensures that the final off-chain state is reflected on-chain.
- **Final Settlement**: The hub contract verifies zk-SNARK proofs submitted by the router, ensuring that the final off-chain balances and trades are correctly reflected on-chain when channels are closed.

The hub contract operates primarily as a verifier of zk-SNARK proofs, minimizing the need for direct on-chain execution of logic, and enabling low-cost, fast transactions.

2.3 Off-Chain Router

The off-chain router is the central component for managing all off-chain operations within the DEX. It handles the interaction between users, the order book, and liquidity providers. The router manages the majority of the logic for the DEX, including:

- **Order Management and Matching**: All buy and sell orders are submitted to the off-chain router. The router maintains the order book, matches orders, and ensures that all orders meet the necessary conditions before being added to the book.
- **State Management with Sparse Merkle Trees**: The router maintains its own Sparse Merkle Tree (SMT), which tracks every state transition, including order placements, order matches, and user balance updates. This SMT allows the router to efficiently manage and verify the validity of all state transitions.
- **Interaction with zk-SNARK Circuits**: Every action processed by the router is accompanied by a zk-SNARK proof, which verifies the correctness of the action (e.g., verifying that a buyer has sufficient funds to place an order). These proofs ensure the correctness of each state transition.

The router is responsible for tracking and validating the entire state of the off-chain order book, balances, and trades, ensuring that only valid and cryptographically proven actions are processed.

Router's Role in State Transitions and Conditional Proofs

In the Overpass Channels system, the router is responsible for processing state transitions using **zk-SNARK proofs** to ensure all conditions are met before an action is accepted. These actions include:

- **Placing Orders**: When a user places a buy or sell order, the router triggers a zk-SNARK operation to validate the user's available balance (for buy orders) or assets (for sell orders). This validation ensures that the order is valid before it is added to the order book.
- **Order Matching**: Once orders are matched by the router, it verifies that both the buyer and seller meet the required conditions (e.g., the buyer has sufficient funds, the seller has the assets to sell) before executing the trade off-chain.
- **Trade Execution**: After a match is confirmed, the router updates the off-chain balances of both parties and records the state transition in its dedicated SMT. This ensures that the correct balances are maintained off-chain and that the entire process is verifiable via zk-SNARK proofs.

Pseudocode for Order Placement and Execution:
The following pseudocode outlines how the router processes an order and verifies its validity using zk-SNARKs:

```
Algorithm: Router's Order Placement with zk-SNARK Validation

1. procedure SubmitOrder(User, OrderType, Amount, Price)
```

```
2.     if OrderType == 'Buy' then
3.         zkProof = zkSNARK_Validate_BuyOrder(User, Amount, Price)
4.         if zkProof is valid then
5.             AddOrderToOrderBook(User, 'Buy', Amount, Price)
6.             UpdateRouterSMT(User, 'Buy', Amount, Price)
7.         else
8.             RejectOrder(User)  # Insufficient funds or invalid conditions
9.         end if
10.    else if OrderType == 'Sell' then
11.        zkProof = zkSNARK_Validate_SellOrder(User, Amount, Price)
12.        if zkProof is valid then
13.            AddOrderToOrderBook(User, 'Sell', Amount, Price)
14.            UpdateRouterSMT(User, 'Sell', Amount, Price)
15.        else
16.            RejectOrder(User)  # Insufficient assets or invalid conditions
17.        end if
18.    end if
19. end procedure
```

In this process: - The router validates all order conditions via zk-SNARK proofs before an order is added to the order book. - Once validated, the router updates its Sparse Merkle Tree (SMT) with the new state, ensuring that each state transition is cryptographically secure and stored for later verification.

2.4 Sparse Merkle Trees for State Tracking

Sparse Merkle Trees (SMTs) provide an efficient way to store and verify state transitions within the system. The off-chain router maintains its own SMT dedicated to tracking all state updates, including order placements, matches, and balance adjustments. Each state change is recorded in the SMT, allowing the system to generate **Merkle proofs** to validate the current state when required.

Structure of the Router's Sparse Merkle Tree

The router's SMT stores state updates in a tree-like structure where each leaf node represents a specific state change (e.g., an order placed, a trade executed). The Merkle root represents the cumulative state of all off-chain actions up to a specific point in time.

Each leaf in the tree contains a hashed value representing:

- **Order Information**: Details about the buy or sell order, including user ID, token type, amount, and price.
- **Balance Updates**: Changes to a user's balance after an order is placed or executed.
- **Trade Completion**: Final state of a trade, including the new balances of both the buyer and seller.

Sparse Merkle Tree Example:

$$\text{Merkle Root} \leftarrow \text{H}(\text{H}(\text{Order 1}, \text{Order 2}), \text{H}(\text{Order 3}, \text{Order 4}))$$

The root of the tree is updated every time a new state transition occurs, ensuring that the entire history of state transitions is securely stored and verifiable.

State Transition Verification via Merkle Proofs

When the router needs to verify a state transition (e.g., during settlement or upon request), it provides a **Merkle proof** that proves the current state of the system is valid based on prior updates. This proof includes the following components:

- The **Merkle root** representing the latest state of the SMT.
- The **Merkle path** leading to the specific state (e.g., an order or balance) that needs to be verified.
- The **leaf value** corresponding to the state transition being verified.

The **on-chain hub contract** can verify this Merkle proof to ensure that the state transition is valid before updating on-chain balances. This reduces the need for large-scale on-chain storage and ensures that the system remains efficient.

Pseudocode for Merkle Proof Verification:

```
Algorithm: Verifying a State Transition with Merkle Proof

1. procedure VerifyMerkleProof(MerkleRoot, MerklePath, LeafValue)
2.     currentHash = LeafValue
3.     for each node in MerklePath do
4.         currentHash = Hash(currentHash, node)
5.     end for
6.     if currentHash == MerkleRoot then
7.         return True  # State transition is valid
8.     else
9.         return False  # Invalid state transition
10.    end if
11. end procedure
```

2.5 zk-SNARK Integration for Validating Every Action

The Off-Chain Router ensures that every action taken by users is valid by generating zk-SNARK proofs for each operation. Whether a user is placing a buy or sell order, or performing a trade, these operations are validated off-chain with zk-SNARKs before being reflected in the Sparse Merkle Tree (SMT). This validation process ensures that each action meets the necessary conditions (such as sufficient funds or valid asset ownership), making the DEX system robust, secure, and resistant to fraudulent activity.

zk-SNARK Generation for State Transitions

For every state transition, the router generates a zk-SNARK proof to validate the following:

- **For Buy Orders**: The buyer has sufficient funds locked in their payment channel to cover the order. The proof verifies that the balance exists off-chain and that it has been properly locked.
- **For Sell Orders**: The seller has the corresponding assets available to fulfill the sell order. This proof validates the presence of the assets in the seller's off-chain balance.
- **For Order Matching**: The router ensures that the matched buy and sell orders meet all predefined conditions, such as price compatibility and quantity availability.

These proofs are generated using the zk-SNARK framework, ensuring that every action performed off-chain is verifiable without requiring consensus or extensive on-chain verification. Once the zk-SNARK proof is generated, the router updates its Sparse Merkle Tree (SMT) with the new state, cryptographically ensuring that the state change is valid and secure.

Pseudocode for zk-SNARK Proof Generation in a Trade:

```
Algorithm: zk-SNARK Proof for State Transition in a Trade

1. procedure Generate_ZKProof(OrderType, User, Amount, Token)
2.     if OrderType == 'Buy' then
3.         Check user has sufficient funds locked
4.         Create proof zkProof = Prove(User.Balance >= Amount)
5.     else if OrderType == 'Sell' then
6.         Check user has sufficient assets to sell
7.         Create proof zkProof = Prove(User.Asset >= Amount)
8.     end if
9.     return zkProof
10. end procedure
```

Once the zk-SNARK proof is successfully generated, it guarantees that the user's action is valid, allowing the system to proceed with order placement or execution.

2.6 Trade Execution Process

When the router matches a buy and sell order, it executes the trade off-chain by updating both parties' balances. This process is done in conjunction with zk-SNARK proofs to ensure that the conditions for both parties (buyer and seller) are met. The steps for executing a trade are as follows:

Off-Chain Trade Execution

Once the orders are matched, the router performs the following actions:

1. **Generate zk-SNARK Proofs for Buyer and Seller**: - The router generates zk-SNARK proofs to verify that the buyer has sufficient funds and the seller has the required assets to fulfill the trade. If both proofs are valid, the trade proceeds.

2. **Update Router's Sparse Merkle Tree**: - After the trade is validated by zk-SNARK proofs, the router updates the SMT to reflect the new state: - The buyer's balance is reduced by the amount they paid (e.g., USDT). - The seller's asset balance is reduced by the quantity sold (e.g., ETH). - Both new balances are reflected in the SMT, and the Merkle root is updated accordingly.

Pseudocode for Off-Chain Trade Execution:

```
Algorithm: Off-Chain Trade Execution

1. procedure ExecuteTrade(BuyOrder, SellOrder)
2.     # Validate both orders using zk-SNARKs
3.     buyerProof = Generate_ZKProof('Buy', BuyOrder.User, BuyOrder.Amount, Token)
4.     sellerProof = Generate_ZKProof('Sell', SellOrder.User, SellOrder.Amount, Token)
5.     if buyerProof is valid and sellerProof is valid then
6.         # Update balances in the router's Sparse Merkle Tree (SMT)
7.         UpdateRouterSMT(BuyOrder.User, 'Subtract', BuyOrder.Amount)
8.         UpdateRouterSMT(SellOrder.User, 'Subtract', SellOrder.Amount)
9.         UpdateRouterSMT(BuyOrder.User, 'Add', SellOrder.Amount)  # Buyer gets tokens
10.        UpdateRouterSMT(SellOrder.User, 'Add', BuyOrder.Amount)  # Seller gets funds
11.        return Success
12.    else
13.        return Failure  # Invalid trade conditions
14.    end if
15. end procedure
```

3. **Distribute Liquidity Provider Rewards**: - Liquidity providers (LPs) are rewarded based on the trade volume. The router uses the SMT to keep track of liquidity utilization and calculates rewards based on the LP's contribution. LP rewards are recorded as state changes in the SMT and distributed accordingly.

4. **Final State in the Router's SMT**: - The final post-trade state is stored in the router's Sparse Merkle Tree. This state includes the updated balances of both the buyer and seller, as well as the LP rewards. The Merkle root is updated to reflect the latest state.

2.7 Sparse Merkle Trees for Order and Balance Tracking

The router leverages Sparse Merkle Trees (SMTs) to efficiently track all off-chain actions, including orders, trades, and balance updates. Each state transition is stored as a node in the SMT, and every action (whether it be placing an order or fulfilling a trade) results in an update to the tree.

Router's SMT Structure

The router's SMT tracks all state transitions, ensuring that each state change is cryptographically secure and can be verified using Merkle proofs. The structure of the SMT can be described as follows:

- **Leaves**: Each leaf node in the SMT represents a specific state transition, such as a balance update, an order placed in the order book, or a completed trade.

- **Branches**: The branches of the SMT represent the intermediary states of multiple leaves. The hashes of these branches allow for efficient verification of the entire state.
- **Root**: The Merkle root summarizes the state of all active orders, user balances, and liquidity data. The root is updated after each state transition and can be used for final verification when settling on-chain.

Example of Router's SMT for a Trade:

$$\text{Merkle Root} \leftarrow \text{H}(\text{H}(\text{Buy Order}, \text{Sell Order}), \text{H}(\text{Balance A}, \text{Balance B}))$$

The Merkle root represents the final, cumulative state of all orders and balances in the DEX system. Each new trade or balance update results in the Merkle root being recalculated to reflect the latest state.

Merkle Proofs for Verifying Trades

When a trade is executed or a balance is updated, the router provides a Merkle proof to verify the correctness of the state transition. The Merkle proof consists of:

- **Merkle root**: The latest root of the SMT, which summarizes the entire state of the system.
- **Merkle path**: The set of hashes from the leaf node to the root, proving that a specific state change (e.g., a trade or balance update) is valid.
- **Leaf node**: The specific node representing the state change (e.g., a user's balance after a trade).

The router uses this proof to confirm that a state transition is valid without needing to reveal the entire state tree. The on-chain hub contract can verify the Merkle proof to ensure that the off-chain state is valid before updating on-chain balances.

2.8 On-Chain Final Settlement

When users want to settle their balances on-chain or close their payment channels, the router submits the final off-chain state to the **on-chain hub contract**. The router provides both a **Merkle proof** (to validate the state transition in the SMT) and a **zk-SNARK proof** (to validate that all actions leading to the final state were correct). This two-pronged approach ensures that the state transitions were valid without requiring extensive on-chain computations.

Closing a Payment Channel

When a user decides to close their payment channel with the DEX, the following process occurs:

- **Provide zk-SNARK and Merkle Proofs**: The router submits both the zk-SNARK proof (proving that the state transitions were valid) and the Merkle proof (proving that the current state reflects the final balance) to the on-chain hub contract.

- **Verify Proofs On-Chain**: The hub contract verifies the zk-SNARK proof to ensure that all off-chain state transitions (e.g., order placements, trade executions) were valid. It also verifies the Merkle proof to confirm that the current state is correctly reflected in the router's SMT.
- **Finalize Settlement**: Once both proofs are verified, the hub contract updates the user's on-chain balance based on the final state of their off-chain payment channel.

This process ensures that final settlement is secure, efficient, and cryptographically proven.
Pseudocode for On-Chain Settlement:

```
Algorithm: On-Chain Final Settlement

1. procedure FinalizeSettlement(User, zkProof, MerkleProof)
2.     # Verify the zk-SNARK proof for all state transitions
3.     if VerifyZKProof(zkProof) == True then
4.         # Verify the Merkle proof to confirm the final state
5.         if VerifyMerkleProof(MerkleProof) == True then
6.             # Update on-chain balances based on final off-chain state
7.             UpdateOnChainBalance(User, FinalState)
8.             return Success
9.         else
10.            return Failure  # Invalid Merkle proof
11.        end if
12.    else
13.        return Failure  # Invalid zk-SNARK proof
14.    end if
15. end procedure
```

In this process, the system ensures that every off-chain action leading to the final on-chain settlement is valid and secure, eliminating the risk of fraud or incorrect state transitions.

2.9 Liquidity Provision and LP Rewards Distribution

Liquidity provision is a key component of the decentralized exchange (DEX) on Overpass Channels. Liquidity providers (LPs) deposit assets into the system, enabling traders to execute orders by leveraging the available liquidity in the pools. The rewards for providing liquidity are distributed based on the trading activity and the LP's contribution to the liquidity pool.

Depositing Liquidity to the Hub Contract

Liquidity providers deposit assets directly into the **on-chain hub contract**, which manages the liquidity pools for different trading pairs (e.g., ETH/USDT, BTC/ETH). Each LP is issued **LP tokens** representing their share of the pool, which entitles them to a proportional share of the trading fees and rewards.

The process of depositing liquidity works as follows:

1. **LP Deposits Assets On-Chain**: An LP deposits a certain amount of assets (e.g., 100 ETH) into the hub contract.

2. **LP Tokens Minted**: In return, the LP receives LP tokens that represent their proportional share in the liquidity pool. These tokens can be used to track the LP's stake in the pool and earn rewards based on trading activity.

3. **Hub Contract Updates Liquidity Pools**: The hub contract updates the corresponding liquidity pool for the specific trading pair (e.g., ETH/USDT) to reflect the newly added liquidity.

Pseudocode for Liquidity Deposit:

```
Algorithm: Liquidity Deposit

1. procedure DepositLiquidity(LP, Token, Amount)
2.     # LP deposits assets into the on-chain hub contract
3.     MintLPTokens(LP, Token, Amount)  # Issue LP tokens to the liquidity provider
4.     UpdateLiquidityPool(Token, Amount)  # Update the liquidity pool state
5.     return Success
6. end procedure
```

Once the liquidity is deposited, the LP is eligible to earn rewards based on the trading volume in the pool.

Off-Chain Liquidity Utilization

Once liquidity is deposited into the hub contract, the router manages its utilization in off-chain trades. The router keeps track of which trades are consuming liquidity from specific pools and ensures that LPs are compensated for providing liquidity.

- **Liquidity Tracking in SMT**: Each time liquidity is used to fulfill a trade, the router updates its Sparse Merkle Tree (SMT) to reflect the amount of liquidity consumed. This ensures that the off-chain state accurately tracks liquidity usage.
- **LP Rewards Calculation**: As trades are executed off-chain, the router calculates the trading fees and rewards for each LP based on their contribution to the liquidity pool.
- **Final State Recorded in SMT**: The final state of liquidity utilization is recorded in the router's SMT, ensuring that the rewards distribution can be verified using Merkle proofs.

LP Rewards Distribution Based on Usage

LPs are rewarded based on the trading activity that occurs within the liquidity pool. The more trades that are executed using the liquidity provided by the LP, the higher the rewards they earn. The process works as follows:

1. **Trade Execution**: When a trade is executed off-chain, the router calculates the trading fees based on the volume of the trade.
2. **Proportional Reward Allocation**: The fees generated from the trade are distributed to LPs based on their contribution to the liquidity pool. LPs that have provided more liquidity receive a larger share of the fees.

3. **Update in the SMT**: The router updates the SMT to record the rewards distribution, ensuring that the rewards are properly reflected in the final state.

Pseudocode for LP Rewards Distribution:

```
Algorithm: LP Rewards Distribution

1. procedure DistributeLPRewards(TradeVolume, Pool, FeePercentage)
2.     # Calculate total fees for the trade
3.     TotalFees = TradeVolume * FeePercentage
4.     for each LP in Pool do
5.         # Calculate LP's share based on their contribution
6.         LpShare = LP.Contribution / Pool.TotalLiquidity
7.         # Distribute fees to the LP
8.         LpReward = TotalFees * LpShare
9.         UpdateRouterSMT(LP, 'Add', LpReward)  # Update SMT with the LP reward
10.    end for
11.    return Success
12. end procedure
```

Once the rewards are calculated and distributed, the LPs can claim their rewards either off-chain by keeping the LP tokens or on-chain by closing their channel with the hub contract.

2.10 Order Lifecycle with zk-SNARK Validations and SMT Updates

The lifecycle of an order on the DEX follows a highly secure and efficient process, with every action verified through zk-SNARK proofs and reflected in the router's Sparse Merkle Tree. This ensures that only valid orders are processed, trades are executed correctly, and state transitions are cryptographically secured.

Order Placement and Validation

When a user places an order (either buy or sell), the router validates the order using a zk-SNARK proof to ensure that all conditions are met. This includes verifying that the user has the required balance or assets to fulfill the order.

Steps in Order Placement:

1. **User Submits Order**: The user submits a buy or sell order to the off-chain router, specifying the asset, price, and quantity.

2. **zk-SNARK Validation**: The router triggers a zk-SNARK circuit to validate the order. For buy orders, it verifies that the user has sufficient funds locked in their payment channel. For sell orders, it checks that the user has enough of the asset to sell.

3. **Order Added to Order Book**: Once validated, the router adds the order to the off-chain order book and updates its Sparse Merkle Tree (SMT) to reflect the new state.

Pseudocode for Order Placement with zk-SNARK Validation:

```
Algorithm: Order Placement

1. procedure SubmitOrder(User, OrderType, Amount, Price)
2.     if OrderType == 'Buy' then
3.         zkProof = zkSNARK_Validate_BuyOrder(User, Amount, Price)
4.         if zkProof is valid then
5.             AddOrderToOrderBook(User, 'Buy', Amount, Price)
6.             UpdateRouterSMT(User, 'Buy', Amount, Price)
7.         else
8.             RejectOrder(User)  # Insufficient funds or invalid conditions
9.         end if
10.    else if OrderType == 'Sell' then
11.        zkProof = zkSNARK_Validate_SellOrder(User, Amount, Price)
12.        if zkProof is valid then
13.            AddOrderToOrderBook(User, 'Sell', Amount, Price)
14.            UpdateRouterSMT(User, 'Sell', Amount, Price)
15.        else
16.            RejectOrder(User)  # Insufficient assets or invalid conditions
17.        end if
18.    end if
19. end procedure
```

This process ensures that only valid orders are placed and prevents spam or invalid orders from entering the system.

Order Matching and Trade Execution

Once an order is placed and added to the order book, the router constantly checks for matching orders. When a buy order and sell order meet the necessary conditions (price, quantity, etc.), the router executes the trade off-chain, updating the balances of both parties.

Steps in Order Matching and Trade Execution:

1. **Order Matching**: The router scans the order book for matching orders. A buy order is matched with a sell order if the price and quantity are compatible.
2. **Trade Validation with zk-SNARKs**: Before executing the trade, the router generates zk-SNARK proofs for both the buyer and seller to ensure that both parties can fulfill the trade (e.g., buyer has enough funds, seller has enough assets).
3. **Trade Execution**: Once validated, the router updates the off-chain balances of both parties and records the trade in its SMT.

Pseudocode for Order Matching and Trade Execution:

```
Algorithm: Order Matching and Trade Execution

1. procedure MatchOrders()
2.     for each BuyOrder in BuyOrderBook do
```

```
3.         for each SellOrder in SellOrderBook do
4.             if BuyOrder.Price >= SellOrder.Price and
5.                BuyOrder.Amount >= SellOrder.Amount then
6.                 # Validate both orders using zk-SNARKs
7.                 buyerProof = zkSNARK_Validate_BuyOrder(BuyOrder.User, BuyOrder.Amount)
8.                 sellerProof = zkSNARK_Validate_SellOrder(SellOrder.User, SellOrder.Amount)
9.                 if buyerProof is valid and sellerProof is valid then
10.                    ExecuteTrade(BuyOrder, SellOrder)
11.                    UpdateRouterSMT(BuyOrder.User, 'Subtract', BuyOrder.Amount)
12.                    UpdateRouterSMT(SellOrder.User, 'Subtract', SellOrder.Amount)
13.                    UpdateRouterSMT(BuyOrder.User, 'Add', SellOrder.Amount)
14.                    UpdateRouterSMT(SellOrder.User, 'Add', BuyOrder.Amount)
15.                    RemoveMatchedOrders(BuyOrder, SellOrder)
16.                end if
17.            end if
18.        end for
19.    end for
20. end procedure
```

This process ensures that trades are executed only when both orders are valid and all conditions are met. The off-chain router takes care of updating the balances and recording the trade in its Sparse Merkle Tree.

Trade Finalization and Settlement

After the trade is executed and recorded in the router's SMT, the users may choose to settle on-chain. The final state is submitted to the on-chain hub contract, along with zk-SNARK and Merkle proofs, to finalize the trade and update the users' on-chain balances.

Steps in Trade Finalization:

1. **Final State Submission**: When a user closes their payment channel, the router provides the final state of their off-chain balance, along with zk-SNARK and Merkle proofs.
2. **On-Chain Verification**: The on-chain hub contract verifies both the zk-SNARK proof (ensuring that the off-chain actions were valid) and the Merkle proof (ensuring that the final balance is accurate).
3. **Final Settlement**: Once verified, the hub contract updates the user's on-chain balance based on the final off-chain state.

Pseudocode for Final Settlement:

```
Algorithm: Final Settlement and Balance Update

1. procedure SettleOnChain(User, zkProof, MerkleProof)
2.     if VerifyZKProof(zkProof) == True then
3.         if VerifyMerkleProof(MerkleProof) == True then
4.             UpdateOnChainBalance(User, FinalState)
```

```
5.             return Success
6.         else
7.             return Failure  # Invalid Merkle proof
8.         end if
9.     else
10.        return Failure  # Invalid zk-SNARK proof
11.    end if
12. end procedure
```

This ensures that the final on-chain balance reflects all valid off-chain actions, creating a secure and efficient settlement process.

3 Security and Efficiency through zk-SNARKs and SMTs

The use of zk-SNARKs and Sparse Merkle Trees ensures the security, privacy, and efficiency of the Overpass Channels DEX. Every action is cryptographically proven, ensuring that only valid trades are executed, liquidity providers are compensated accurately, and users' final balances are securely settled on-chain.

3.1 Scalability with Off-Chain Processing

By performing most operations off-chain, the DEX minimizes the load on the underlying blockchain. This allows the system to scale to handle a high volume of trades without being constrained by on-chain throughput. The use of zk-SNARK proofs ensures that all off-chain actions are secure and verifiable, while the Sparse Merkle Tree structure enables efficient tracking and verification of state transitions.

3.2 Privacy through zk-SNARKs

zk-SNARKs enable private transactions by allowing the router to validate actions without revealing sensitive information, such as the exact amount of a user's balance or the details of a trade. This ensures that users' financial privacy is preserved, even as their actions are verified cryptographically.

3.3 Security through Merkle Proofs

The use of Sparse Merkle Trees allows for efficient verification of the system's state. Every state transition (whether it be an order placement, trade execution, or liquidity update) is recorded in the SMT, ensuring that the entire history of actions can be cryptographically proven. This reduces the risk of fraud or manipulation and ensures that final on-chain settlements are based on a valid, verifiable off-chain state.

4 Advanced Order Types and Centralized Exchange-like Experience

One of the goals of the Overpass Channels DEX architecture is to provide a user experience that rivals that of centralized exchanges (CEXs) while maintaining the security, privacy, and trustless

nature of a decentralized system. To achieve this, the DEX supports a variety of advanced order types, fast execution, and an order book that feels as responsive as its centralized counterparts.

4.1 Order Types in the DEX

The Overpass Channels DEX supports multiple order types, enabling users to execute complex trading strategies. These order types are implemented off-chain via the router, which handles the order matching and execution while ensuring that every action is validated via zk-SNARKs.

Market Orders

A market order allows a trader to buy or sell an asset immediately at the best available price. This order type does not specify a price; instead, it is executed against the current available orders in the order book.

Steps in Market Order Execution:

1. **Order Submission**: A trader submits a market order (buy or sell) to the router.
2. **Order Matching**: The router immediately matches the market order with the best available limit orders in the order book.
3. **zk-SNARK Validation**: The router verifies the trader's balance via a zk-SNARK proof before executing the trade.
4. **Execution**: The trade is executed at the best available price, and the corresponding balances are updated in the Sparse Merkle Tree (SMT).

Pseudocode for Market Order Execution:

```
Algorithm: Market Order Execution

1. procedure SubmitMarketOrder(User, OrderType, Amount)
2.     # Fetch best available order from the order book
3.     BestOrder = FetchBestAvailableOrder(OrderType)
4.     if BestOrder is found then
5.         # Validate market order using zk-SNARK
6.         zkProof = zkSNARK_ValidateOrder(User, Amount)
7.         if zkProof is valid then
8.             # Execute market order at the best price
9.             ExecuteTrade(User, BestOrder, Amount)
10.            UpdateRouterSMT(User, 'Subtract', Amount)
11.            UpdateRouterSMT(BestOrder.User, 'Add', Amount)
12.            return Success
13.        else
14.            return Failure  # Invalid balance or conditions
15.        end if
16.    else
17.        return Failure  # No matching orders available
18.    end if
19. end procedure
```

Limit Orders

A limit order allows traders to specify the price at which they want to buy or sell an asset. The trade will only be executed if the market price reaches the specified limit price or better.

Steps in Limit Order Placement and Execution:

1. **Order Submission**: A trader submits a limit order specifying the price and amount.
2. **zk-SNARK Validation**: The router generates a zk-SNARK proof to validate the trader's balance or assets required for the order.
3. **Order Added to Order Book**: If validated, the limit order is added to the order book.
4. **Order Matching**: The router continuously monitors the order book to check if there is a match with an incoming buy or sell order that meets the limit price.
5. **Execution**: When the conditions are met (e.g., the market price matches or exceeds the limit price), the router executes the trade, and the balances are updated in the SMT.

Pseudocode for Limit Order Execution:

```
Algorithm: Limit Order Execution

1. procedure SubmitLimitOrder(User, OrderType, Amount, LimitPrice)
2.     # Validate the limit order using zk-SNARK
3.     zkProof = zkSNARK_ValidateLimitOrder(User, Amount, LimitPrice)
4.     if zkProof is valid then
5.         # Add limit order to the order book
6.         AddOrderToOrderBook(User, OrderType, Amount, LimitPrice)
7.         UpdateRouterSMT(User, 'Lock', Amount)  # Lock funds for the order
8.         return Success
9.     else
10.        return Failure  # Invalid balance or conditions
11.    end if
12. end procedure
```

Stop-Loss Orders

A stop-loss order allows traders to set a predefined price at which an asset should be sold to limit losses. The order is triggered when the asset's market price falls to the stop price.

Steps in Stop-Loss Order Execution:

1. **Order Submission**: The trader submits a stop-loss order with a specified stop price and amount.
2. **zk-SNARK Validation**: The router verifies the stop-loss order using a zk-SNARK proof to ensure that the user has sufficient assets to sell.
3. **Order Triggering**: Once the asset's market price reaches the stop price, the router triggers the stop-loss order and converts it into a market order.

4. **Execution**: The market order is executed, and the trader's assets are sold at the best available price.

Pseudocode for Stop-Loss Order Execution:

```
Algorithm: Stop-Loss Order Execution

1. procedure SubmitStopLossOrder(User, Amount, StopPrice)
2.     # Validate the stop-loss order using zk-SNARK
3.     zkProof = zkSNARK_ValidateStopLoss(User, Amount, StopPrice)
4.     if zkProof is valid then
5.         AddStopLossOrder(User, Amount, StopPrice)
6.         UpdateRouterSMT(User, 'Lock', Amount)  # Lock assets for stop-loss
7.         return Success
8.     else
9.         return Failure  # Invalid balance or conditions
10.    end if
11. end procedure

12. procedure TriggerStopLossOrder(User, Amount, StopPrice)
13.    if MarketPrice <= StopPrice then
14.        ConvertStopLossToMarketOrder(User, Amount)
15.        SubmitMarketOrder(User, 'Sell', Amount)
16.    end if
17. end procedure
```

Stop-Limit Orders

A stop-limit order combines the functionality of a stop-loss order and a limit order. Once the stop price is reached, the order becomes a limit order, and the trade will only be executed at the limit price or better.

Steps in Stop-Limit Order Execution:

1. **Order Submission**: The trader submits a stop-limit order specifying the stop price, limit price, and amount.
2. **zk-SNARK Validation**: The router validates the order and the trader's balance using a zk-SNARK proof.
3. **Order Triggering**: When the market price reaches the stop price, the order is converted into a limit order.
4. **Execution**: The limit order is executed once the market price meets or exceeds the limit price.

Pseudocode for Stop-Limit Order Execution:

```
Algorithm: Stop-Limit Order Execution
```

```
1. procedure SubmitStopLimitOrder(User, Amount, StopPrice, LimitPrice)
2.     # Validate the stop-limit order using zk-SNARK
3.     zkProof = zkSNARK_ValidateStopLimit(User, Amount, StopPrice, LimitPrice)
4.     if zkProof is valid then
5.         AddStopLimitOrder(User, Amount, StopPrice, LimitPrice)
6.         UpdateRouterSMT(User, 'Lock', Amount)  # Lock assets for stop-limit
7.         return Success
8.     else
9.         return Failure  # Invalid balance or conditions
10.    end if
11. end procedure

12. procedure TriggerStopLimitOrder(User, Amount, StopPrice, LimitPrice)
13.    if MarketPrice <= StopPrice then
14.        ConvertStopLimitToLimitOrder(User, Amount, LimitPrice)
15.        SubmitLimitOrder(User, 'Sell', Amount, LimitPrice)
16.    end if
17. end procedure
```

4.2 Order Book Management and User Experience

The order book in the Overpass Channels DEX is designed to provide a fast, responsive experience similar to that of centralized exchanges. The order book is managed off-chain by the router and continuously updated as new orders are placed, matched, or cancelled.

Centralized Exchange-like Speed

The DEX's off-chain architecture allows for near-instantaneous order placement and matching, providing a user experience that feels as fast as using a centralized exchange. The key to achieving this speed is by performing the order matching and validation off-chain, using zk-SNARKs for verification while keeping on-chain interactions minimal.

Features of CEX-like Speed and Experience:

- **Instant Order Book Updates**: Since the order book is managed off-chain, new orders appear in the order book instantly once validated.
- **Fast Order Matching**: The off-chain router constantly scans for matching orders, ensuring that trades are executed as quickly as possible once the conditions are met.
- **Secure Execution with zk-SNARKs**: Every order and trade is validated securely using zk-SNARK proofs, ensuring that the system remains decentralized and trustless, even while delivering a centralized exchange-like experience.

4.3 Liquidity, Depth, and Advanced Trading Strategies

To further replicate the feel of a centralized exchange, the DEX supports deep liquidity pools and allows for advanced trading strategies. With LPs providing liquidity across multiple trading pairs,

traders can execute large orders without causing significant slippage, similar to what they would expect on a traditional exchange.

Features for Advanced Trading Strategies:

- **High Liquidity**: LPs deposit assets into the on-chain hub contract, creating deep liquidity pools for major trading pairs.
- **Low Slippage**: The deep liquidity pools reduce slippage, enabling large trades to be executed without significantly affecting the market price.
- **Advanced Order Types**: The DEX supports various order types (market, limit, stop-loss, stop-limit), allowing traders to implement sophisticated trading strategies.

Index