Overpass Channels :BOCs SMTs OPs

Brandon "Cryptskii" Ramsay

October 2024

1 Core Components: BOCs, OP Codes, and SMTs

The core architecture of Overpass Channels relies on several critical components that enable scalable and verifiable off-chain operations. This section provides an in-depth analysis of the fundamental building blocks: Bag of Cells (BOCs), OP codes, and Sparse Merkle Trees (SMTs).

1.1 Bag of Cells (BOCs)

Bag of Cells (BOCs) provide a versatile data serialization model used extensively in Overpass Channels to represent contract logic, state data, and transaction information. The BOC structure is based on a directed acyclic graph (DAG), which enables efficient and compact data representation.

Formal Definition of BOCs

Definition 1 (Bag of Cells (BOC))**.** *A* ***Bag of Cells (BOC)*** *is defined as a tuple* $B = (V, E, C)$*, where:*

- V *is a finite set of vertices, each representing a cell, i.e., a distinct piece of data or logic.*
- $E \subset V \times V$ *is a set of directed edges, forming a directed acyclic graph (DAG).*
- $C : V \to D$ *is a mapping from vertices to data or code objects, where* D *represents the domain of data, including contract state, transaction instructions, or other relevant information.*

The DAG structure ensures that there are no cyclic dependencies among cells, allowing for a hierarchical representation of execution and state.

Key Features of BOCs

Compact Data Representation BOCs serialize data into a directed acyclic graph (DAG), facilitating efficient storage and transmission. This is crucial for scalability in Overpass Channels, where minimizing on-chain storage requirements is a priority. The DAG structure enables redundancy reduction through shared subgraphs, providing a compact data representation.

Versatility BOCs represent various entities in Overpass Channels, such as smart contract code, state data, and transaction instructions. This versatility makes them integral to both execution logic and state management. Formally, let $B_i \in B$ be an individual BOC representing state S_i at time t_i, which can be serialized and deserialized for both on-chain and off-chain operations.

Efficiency in Off-Chain Processing BOCs are instrumental in executing and managing the state of off-chain contracts, reducing the computational burden on the blockchain. Given that each cell within a BOC represents an atomic operation or piece of state, the BOC provides an efficient way to handle dependencies during the off-chain contract execution process.

Serialization of BOCs

An essential aspect of BOCs is their serialization mechanism, which defines how the data contained within the cells is encoded into a binary format suitable for storage and transmission. The serialization process ensures that the BOC structure can be reconstructed accurately from the serialized data, preserving the integrity and relationships between the cells.

Serialization Format The serialization format of BOCs involves converting the DAG of cells into a linear sequence of bytes. The serialization process follows specific rules to ensure that the structure and content of the BOC can be fully reconstructed. The main components of the serialization format include:

- **Cell Data**: Each cell's data, including its type, content, and references to other cells.
- **Reference Indices**: Indices that specify how cells reference each other within the BOC.
- **BOC Header**: Metadata about the BOC, such as the total number of cells, size of the cells, and flags indicating serialization options.

BOC Header Structure The BOC header contains essential metadata required for deserialization. The header typically includes the following fields:

- **Magic Number**: A predefined constant used to identify the data as a BOC.
- **Flags and Size**: Flags indicating serialization options and the size of the BOC.
- **Number of Cells**: The total number of cells included in the BOC.
- **Root Cells**: The indices of the root cells in the BOC.

Cell Serialization Each cell in the BOC is serialized individually, including its content and references to other cells. The serialization of a cell includes:

- **Cell Descriptor**: Information about the cell, such as the number of references it contains and the size of its data.
- **Cell Data**: The actual content of the cell, typically in binary form.
- **Reference List**: A list of indices pointing to other cells that this cell references.

Serialization Algorithm The serialization process can be formalized as follows:

1. **Assign Indices to Cells**: Traverse the DAG and assign a unique index to each cell.
2. **Serialize Cells**: For each cell, create a serialized representation including its descriptor, data, and reference list.
3. **Construct BOC Header**: Create the BOC header containing metadata about the BOC.
4. **Assemble Serialized BOC**: Concatenate the BOC header and the serialized cells into a single binary sequence.

Mathematically, let $B = (V, E, C)$ be a BOC with n cells. The serialization function Serialize : $B \to \{0,1\}^*$ maps the BOC to a binary sequence. The deserialization function Deserialize : $\{0,1\}^* \to B$ reconstructs the BOC from the binary sequence.

Example of BOC Serialization Consider a simple BOC with three cells:

- **Cell 1**: Contains data d_1 and references Cell 2.
- **Cell 2**: Contains data d_2 and references Cell 3.

- **Cell 3**: Contains data d_3 and has no references.

The serialization process would proceed as follows:

1. Assign indices:
    - Cell 1: Index 0
    - Cell 2: Index 1
    - Cell 3: Index 2
2. Serialize cells:
    - Cell 1 serialization includes:
        - Descriptor indicating 1 reference and size of d_1.
        - Data d_1.
        - Reference list containing index 1.
    - Cell 2 serialization includes:
        - Descriptor indicating 1 reference and size of d_2.
        - Data d_2.
        - Reference list containing index 2.
    - Cell 3 serialization includes:
        - Descriptor indicating 0 references and size of d_3.
        - Data d_3.
        - Empty reference list.
3. Construct BOC header with magic number, flags, number of cells (3), and root cell index (0).
4. Assemble the serialized BOC by concatenating the header and serialized cells.

Deserialization The deserialization process involves parsing the binary sequence to reconstruct the BOC:

1. **Read and Parse the BOC Header**: Extract metadata such as the number of cells and root indices.
2. **Deserialize Cells**: Read each serialized cell, reconstructing its descriptor, data, and reference list.
3. **Rebuild the DAG Structure**: Use the reference lists to link cells according to their indices, forming the original DAG.

Serialization Format Standards In practice, BOCs may utilize existing serialization standards or custom formats optimized for Overpass Channels. Common serialization formats include:

- **TLV (Type-Length-Value) Encoding**: Encodes data by specifying the type of data, the length of the data, and the value itself. This allows for flexible and extensible data representation.
- **Binary Encoding**: Efficiently represents data in binary form, minimizing the size of the serialized BOC.
- **Self-Describing Formats**: Include metadata within the serialized data to facilitate parsing and validation.

Choosing an appropriate serialization format involves balancing efficiency, ease of implementation, and compatibility with cryptographic operations.

Implications and Practical Considerations Including serialization details in the definition of BOCs has several important implications:

- **Interoperability**: A well-defined serialization format ensures that different participants and systems can accurately interpret and process BOCs.
- **Security**: Proper serialization and deserialization prevent issues such as data corruption and injection attacks.
- **Efficiency**: Optimizing the serialization format can reduce the size of BOCs, leading to lower storage and transmission costs.
- **Determinism**: A consistent serialization process ensures that the same BOC structure always results in the same binary representation, which is important for hashing and cryptographic operations.

Mathematical Representation of BOCs in Off-Chain Contracts

BOCs play a crucial role in managing the state and execution of off-chain contracts. They act as containers for encapsulating state transitions and contract logic, with the DAG structure allowing for clear and immutable execution paths. Unlike typical DAGs, which establish links between nodes, BOCs are managed within Sparse Merkle Trees (SMTs), enabling greater flexibility in representing state transitions.

State Serialization

Definition 2 (State Serialization)**.** *Each state S of a contract can be represented as a sequence of cells $S = \{c_1, c_2, \ldots, c_n\}$, where each cell $c_i \in V$ represents a distinct piece of state information. A state transition from S_{i-1} to S_i is serialized into a BOC and represented as a leaf in a Sparse Merkle Tree (SMT).*

Execution Flow Representation The execution flow of a smart contract is encapsulated within a BOC, with each cell in the BOC representing a step in the sequence of contract operations. The directed acyclic nature of the BOC DAG ensures that the execution flow is deterministic and prevents cyclic dependencies, which could otherwise lead to inconsistencies in state transitions.

Theorem 1 (Deterministic Execution Flow)**.** *Let $B = (V, E, C)$ be a BOC representing the execution flow of a contract. If B is a directed acyclic graph, then there exists a topological ordering of the vertices V such that each vertex v_i is processed only after all its predecessors have been processed. Thus, the execution flow is deterministic.*

Proof. Since B is a DAG, a topological ordering π of V can be determined in $O(|V| + |E|)$ time. In this ordering, each vertex v_i appears only after all vertices with edges directed towards v_i. This ensures that each operation (cell) in the BOC is processed after its dependencies, leading to a deterministic execution flow. ■

Implications and Practical Considerations

Understanding the serialization mechanism of BOCs is crucial for several reasons:

- **Consistency in State Representation**: Accurate serialization ensures that the state of contracts and transactions is represented consistently across different nodes in the network.
- **Efficient Data Transmission**: Optimized serialization reduces the size of the data transmitted between parties, enhancing the efficiency of the network.
- **Secure Hashing and Verification**: Since BOCs are hashed and included in SMTs, consistent serialization is essential for generating correct hashes used in cryptographic proofs.

- **Compatibility with Cryptographic Operations**: The serialization format must be compatible with cryptographic functions used in the system, such as hashing algorithms and digital signatures.

Comparison with Other Serialization Methods

Compared to traditional serialization methods used in blockchain systems, such as encoding transactions or blocks in a linear sequence, the BOC serialization approach offers:

- **Hierarchical Structuring**: The DAG-based structure allows for more complex relationships between data elements, which is beneficial for representing intricate contract logic.
- **Reusability of Subgraphs**: Shared subgraphs within the DAG can be serialized once and referenced multiple times, reducing redundancy.
- **Flexibility**: The BOC format can accommodate various types of data and logic, making it adaptable to different contract requirements.

Best Practices for Implementing BOC Serialization

When implementing BOC serialization in Overpass Channels, consider the following best practices:

- **Define Clear Serialization Rules**: Establish unambiguous rules for how cells and their relationships are serialized to ensure consistency.
- **Validate Serialized Data**: Implement validation checks during serialization and deserialization to detect and prevent errors.

- **Optimize for Performance**: Use efficient data structures and algorithms to minimize the computational overhead of serialization.
- **Ensure Compatibility**: Maintain compatibility with existing protocols and standards where possible to facilitate integration.

By carefully designing and implementing the serialization of BOCs, Overpass Channels can achieve efficient, secure, and reliable off-chain contract management.

Integration with Sparse Merkle Trees (SMTs)

The separation of state transitions and execution logic into BOCs and SMTs respectively allows Overpass Channels to perform complex computations without requiring an on-chain virtual machine. This makes the system more efficient and scalable, particularly for handling large decentralized applications.

Each BOC representing a state transition is stored as a leaf node in a Sparse Merkle Tree. Let M be an SMT with root r and leaves $\{l_1, l_2, \ldots, l_n\}$, where each leaf l_i corresponds to a serialized BOC representing a state transition. The Merkle root r provides a cryptographic commitment to all state transitions, ensuring the integrity and consistency of the off-chain contract states.

Definition 3 (Merkle Root Commitment)**.** *The* ***Merkle root*** r *of an SMT* M *with leaves* $\{l_1, l_2, \ldots, l_n\}$ *is defined as:*

$$r = H(H(l_1, l_2), H(l_3, l_4), \ldots)$$

where H *is a cryptographic hash function. The Merkle root* r *serves as a commitment to the entire set of leaves, allowing for efficient verification of individual state transitions.*

Lemma 2 (Efficient Verification of State Transitions)**.** *Given an SMT with root r and a leaf l_i, the proof of inclusion for l_i can be verified in $O(\log n)$ time, where n is the number of leaves. This enables efficient verification of state transitions represented by BOCs.*

Proof. The proof of inclusion consists of the hash values along the path from leaf l_i to the root r. Since the SMT is a balanced binary tree, the height of the tree is $O(\log n)$. Thus, the number of hash operations required to verify the proof is also $O(\log n)$. ■

2 Example: Alice and Bob's Bidirectional Payment Channel

In this section, we present a detailed example that demonstrates the lifecycle of transactions in the Overpass Channels system, utilizing two unilateral channels between Alice and Bob. This example illustrates the use of BOCs, OP codes, and SMTs throughout the process, from off-chain operations to on-chain settlement.

2.1 Setup

We consider two participants, Alice (A) and Bob (B), who set up two unilateral channels:

- Channel $C_{A\to B}$: Alice to Bob
- Channel $C_{B\to A}$: Bob to Alice

Initially, both Alice and Bob have 100 tokens each. The initial state S_0 for both channels is:

$$S_0(C_{A\to B}) = S_0(C_{B\to A}) = \{B_A = 100, B_B = 100, n = 0\}$$

where B_A and B_B represent the balances of Alice and Bob respectively, and n is the nonce.

2.2 Transaction Rounds

We will walk through three rounds of transactions, demonstrating the full lifecycle at each level of the Overpass Channels system.

Round 1: Alice Sends 20 Tokens to Bob

Off-Chain Operation Alice initiates a transaction to send 20 tokens to Bob.

1. **Create Transaction BOC**:

$$BOC_{tx} = \{\text{OP\_CREATE\_TX}, \{\text{sender} : A, \text{receiver} : B, \text{amount} : 20\}\}$$

2. **Generate zk-SNARK Proof**:

$$\pi_1 = \text{SNARK.Prove}(BOC_{tx}, S_0(C_{A\to B}))$$

3. **Update Channel State**:

$$S_1(C_{A\to B}) = \{B_A = 80, B_B = 120, n = 1\}$$

4. **Create State BOC**:

$$BOC_{state} = \{\text{OP\_UPDATE\_STATE}, S_1(C_{A\to B})\}$$

5. **Update Channel's SMT**:

$$SMT_{C_{A\to B}} = \text{SMT.Update}(SMT_{C_{A\to B}}, H(BOC_{state}))$$

Intermediate Contract Level Aggregate state updates from multiple channels.

1. **Create Aggregated State BOC**:

$$BOC_{agg} = \{\text{OP\_AGGREGATE\_STATES}, \{C_{A\to B} : S_1(C_{A\to B}), C_{B\to A} : S_0(C_{B\to A}$$

2. **Update Intermediate Contract's SMT**:

$$SMT_{IC} = \text{SMT.Update}(SMT_{IC}, H(BOC_{agg}))$$

Root Contract Level No action is taken at this stage; the system waits for more transactions to accumulate.

Round 2: Bob Sends 30 Tokens to Alice

Bob sends 30 tokens to Alice via Channel $C_{B\to A}$.

Off-Chain Operation

1. **Create Transaction BOC**:

$$BOC_{tx} = \{\text{OP\_CREATE\_TX}, \{\text{sender} : B, \text{receiver} : A, \text{amount} : 30\}\}$$

2. **Generate zk-SNARK Proof**:

$$\pi_2 = \text{SNARK.Prove}(BOC_{tx}, S_0(C_{B\to A}))$$

3. **Update Channel State**:

$$S_1(C_{B\to A}) = \{B_B = 70, B_A = 130, n = 1\}$$

4. **Update SMT** as in Round 1 for $C_{B\to A}$.

Intermediate Contract Level Aggregate latest states:

$$BOC_{agg} = \{\text{OP\_AGGREGATE\_STATES}, \{C_{A\to B} : S_1(C_{A\to B}), C_{B\to A} : S_1(C_{B\to A})\}\}$$

Round 3: Alice Sends 15 Tokens to Bob

Alice sends another 15 tokens to Bob.

Off-Chain Operation

1. **Create Transaction BOC**:

   $$BOC_{tx} = \{\text{OP\_CREATE\_TX}, \{\text{sender} : A, \text{receiver} : B, \text{amount} : 15\}\}$$

2. **Generate zk-SNARK Proof**:

   $$\pi_3 = \text{SNARK.Prove}(BOC_{tx}, S_1(C_{A\to B}))$$

3. **Update Channel State**:

   $$S_2(C_{A\to B}) = \{B_A = 65, B_B = 135, n = 2\}$$

4. **Update SMT** for $C_{A\to B}$.

Intermediate Contract Level Aggregate latest states:

$$BOC_{agg} = \{\text{OP\_AGGREGATE\_STATES}, \{C_{A\to B} : S_2(C_{A\to B}), C_{B\to A} : S_1(C_{B\to A})\}\}$$

Root Contract Level Submit the global state update.

1. **Aggregate Updates** from intermediate contracts.
2. **Create Global State BOC**:

   $BOC_{global} = \{\text{OP\_GLOBAL\_UPDATE}, \{IC_1 : SMT_{IC}.root, \ldots\}\}$
3. **Submit Global Merkle Root**:

   $TX_{onchain} = \{\text{OP\_SUBMIT\_GLOBAL\_ROOT}, SMT_{global}.root\}$

2.3 DAG Connections

The DAG structure of contract BOCs allows for efficient representation of the contract's history.

- Channel $C_{A\to B}$: $BOC_{contract} \to BOC_{state(S_1)} \to BOC_{state(S_2)}$
- Channel $C_{B\to A}$: $BOC_{contract} \to BOC_{state(S_1)}$

The SMT structure is:

$$SMT_{global}.root \to \{SMT_{IC_1}.root, SMT_{IC_2}.root, \ldots\} \to \{SMT_{C_{A\to B}}.root, SMT_{C_{B\to A}}.roo$$

2.4 Channel Closing

Alice decides to close Channel $C_{A\to B}$.

Off-Chain Operation

1. **Create Channel Closure BOC**:

   $BOC_{close} = \{\text{OP\_CLOSE\_CHANNEL}, \{\text{channel} : C_{A\to B}, \text{final\_state} : S_2(C_{A\to B})\}$
2. **Generate zk-SNARK Proof**:

   $$\pi_{close} = \text{SNARK.Prove}(BOC_{close}, \{S_0, S_1, S_2\})$$

Intermediate Contract Level Update SMT to reflect channel closure.

$$SMT_{IC} = \text{SMT.Update}(SMT_{IC}, H(BOC_{close}))$$

Root Contract Level Submit channel closure information.

1. **Create Global State BOC** with updated SMT roots.
2. **Submit Channel Closure Transaction**:

   $TX_{closure} = \{\text{OP\_PROCESS\_CHANNEL\_CLOSURE}, \{\text{channel} : C_{A\to B}, \text{final\_sta}$

On-Chain Operation

1. **Verify zk-SNARK Proof**:

   $$\text{SNARK.Verify}(\pi_{close}, BOC_{close}, SMT_{global}.root) = \text{true}$$

2. **Update On-Chain Balances**:

   $$B_A^{onchain} = B_A^{onchain} + 65$$
   $$B_B^{onchain} = B_B^{onchain} + 135$$

3. **Mark Channel as Closed**

2.5 Conclusion

This comprehensive example demonstrates the intricate workings of the Overpass Channels system, illustrating how BOCs, OP codes, and SMTs interact to facilitate efficient off-chain transactions with on-chain security guarantees.

Key aspects and benefits include:

- **Encapsulation of Logic and State**: BOCs encapsulate transaction logic and state information, allowing efficient off-chain processing.
- **Hierarchical State Management**: SMTs at multiple levels enable efficient state updates and verification.
- **Privacy and Verifiability**: zk-SNARK proofs ensure transaction validity without revealing details.
- **Scalability**: Off-chain processing with periodic on-chain updates improves scalability.
- **Efficient Channel Closure**: The system efficiently settles final balances on-chain using off-chain state history and zk-SNARK proofs.

2.6 OP Codes

OP codes are used to control the lifecycle of off-chain contracts in Overpass Channels. These codes are embedded within BOCs and dictate how contracts are created, executed, and terminated. By leveraging OP codes, Overpass Channels streamline the execution of contract logic without needing to rely on complex virtual machine environments.

Formal Definition of OP Codes

Definition 4 (OP Code)**.** *An **OP Code** is an atomic instruction that defines an action to be performed on an off-chain contract. Formally, an OP code $OP \in \mathbb{O}$, where $\mathbb{O}$ is the set of all possible OP codes in the Overpass Channels system. Each OP code OP can be viewed as a function:*

$$OP : (S, P) \to (S', R)$$

where:

- *S is the current state of the contract.*
- *P represents the set of parameters for the operation.*
- *S' is the updated state after executing the operation.*
- *R is the result of the operation, which may include return values or errors.*

Role in Contract Lifecycle

OP codes are essential for managing the lifecycle of off-chain contracts, including creation, execution, state updates, and termination.

Contract Creation

Definition 5 (Contract Creation)**.** *Let $OP_{create} \in \mathbb{O}$ be the OP code that triggers contract creation. The operation is defined as:*

$$OP_{create} : (\emptyset, P_{init}) \rightarrow (S_1, R_1)$$

where P_{init} represents the parameters for the initial state.

Contract Execution After creation, the contract execution is controlled by a sequence of OP codes that dictate the flow of contract logic. Each OP code represents a discrete operation, such as transferring tokens, invoking a function, or verifying a condition.

Theorem 3 (Deterministic Contract Execution)**.** *Let C be an off-chain contract defined by a sequence of states $S_0, S_1, \ldots, S_n$. Let $OP_1, OP_2, \ldots, OP_n \in \mathbb{O}$ be the sequence of OP codes executed on the contract. Then the sequence of state transitions is deterministic if each OP code OP_i is deterministic.*

Proof. Each OP code OP_i is a function $(S_{i-1}, P_i) \to (S_i, R_i)$. Since OP_i is deterministic, the output state S_i and result R_i are uniquely determined by the input state S_{i-1} and parameters P_i. Therefore, the entire sequence of state transitions $S_0 \to S_1 \to \cdots \to S_n$ is deterministic. ■

State Transitions and Termination The state of a contract evolves through a series of transitions governed by OP codes. The termination of a contract is determined by a special OP code $\mathrm{OP}_{\mathrm{terminate}}$, which sets the contract state to a terminal value.

Definition 6 (State Transition)**.** *A* ***State Transition*** T_i *is defined as:*

$$T_i : (S_{i-1}, OP_i) \to S_i$$

Definition 7 (Contract Termination)**.** *Let* $OP_{terminate} \in \mathbb{O}$ *be the OP code that triggers contract termination. The operation is defined as:*

$$OP_{terminate} : (S_n, P_{term}) \to (S_f, R_f)$$

where S_f *is a terminal state that cannot be modified further.*

Algorithm for Processing OP Codes

Algorithm 1 Processing OP Codes in BOCs

```
1: procedure ProcessBOC(boc)
2:     opcode ← boc.op_code
3:     if opcode = OP_CREATE_CONTRACT then
4:         CreateContract(boc)
5:         Update State S_0 → S_1
6:     else if opcode = OP_EXECUTE_CONTRACT then
7:         ExecuteContract(boc)
8:         Update State S_i → S_{i+1}
9:     else if opcode = OP_UPDATE_STATE then
10:        UpdateState(boc)
11:        Update State S_i → S_{i+1}
12:    else if opcode = OP_TERMINATE_CONTRACT then
13:        TerminateContract(boc)
14:        Update State S_n → S_f
15:    else
16:        HandleUnknownOpCode(boc)
17:    end if
18: end procedure
```

Complexity Analysis

The complexity of processing a BOC depends on the number of cells n it contains and the types of OP codes executed. Let T_{op} denote the time complexity of executing an individual OP code. The total complexity for processing a BOC with n cells is:

$$T_{\text{total}} = \sum_{i=1}^{n} T_{\text{op}_i}$$

In practice, OP codes in Overpass Channels are designed to be efficient to ensure that off-chain contract processing remains scalable.

Comparison with Traditional Smart Contract Execution

In traditional blockchain systems, smart contract execution involves a tightly coupled relationship between state management and execution logic, typically managed through an on-chain virtual machine, such as the Ethereum Virtual Machine (EVM). This approach has significant limitations:

- **On-Chain Bottlenecks**: Each state transition must be executed and verified on-chain, leading to scalability limitations due to high computational and storage costs.
- **Complexity of Virtual Machines**: Managing state transitions in a general-purpose virtual machine introduces additional complexity and overhead.

In contrast, Overpass Channels use OP codes embedded in BOCs to perform state transitions off-chain. This separation allows for:

- **Efficient State Management**: State transitions are managed using lightweight OP codes, which are significantly more efficient than using a general-purpose virtual machine.
- **Scalability through Off-Chain Execution**: By executing contracts off-chain and only submitting the resulting state commitments (i.e., SMT roots) on-chain, Overpass Channels significantly reduce the computational load on the blockchain.

2.7 Sparse Merkle Trees (SMTs)

Sparse Merkle Trees (SMTs) are a fundamental component in Overpass Channels, used to manage and verify off-chain state transitions.

SMTs provide a scalable and efficient mechanism for state representation and verification, particularly when dealing with large datasets. This section formalizes the structure and properties of SMTs, explains their integration into the Overpass Channels architecture, and provides detailed mathematical analysis and examples.

Formal Definition of Sparse Merkle Trees

Definition 8 (Sparse Merkle Tree (SMT))**.** *A* ***Sparse Merkle Tree (SMT)*** *is a Merkle tree with a fixed height h, where each leaf represents a possible state of the system, indexed by a unique key $k \in \{0,1\}^h$. An SMT is defined as a tuple (T, H, L), where:*

- *T is a full binary tree of height h, representing all possible states.*
- *$H : D \times D \to D$ is a cryptographic hash function.*
- *$L \subseteq \{0,1\}^h \times D$ is a set of key-value pairs representing non-default (active) leaves of the tree.*

Properties of SMTs

- **Fixed Height**: An SMT has a fixed height h, determining the number of possible leaves 2^h.
- **Cryptographic Integrity**: Each node is associated with a cryptographic hash, and the root provides a commitment to the entire state.
- **Sparse Representation**: Efficient handling of empty nodes allows for scalable state representation without storing all nodes explicitly.

Mathematical Representation of SMT Updates

Suppose we update a specific key k in an SMT with root r_{i-1} to a new value v_i. The process is:

1. **Update the Leaf Node**:

$$l_k = H(k \parallel v_i)$$

2. **Update the Path to the Root**:

For each node p_j along the path, $p_j = H(p_{j,\text{left}}, p_{j,\text{right}})$

3. **Update the Root**:

$r_i =$ Updated root hash after recomputing hashes along the path

Verification of State Transitions Using SMTs

To verify a state transition, a proof of inclusion is provided, consisting of sibling hashes along the path from the leaf to the root.

Definition 9 (Proof of Inclusion)**.** *A* ***Proof of Inclusion*** *for key k is a sequence $(h_1, h_2, \ldots, h_h)$ of sibling hashes used to reconstruct the root r.*

Verification involves recomputing the root from the provided leaf value and proof and checking if it matches the known root r.

Complexity Analysis of SMT Operations

Time Complexity Both updating a value in an SMT and verifying a proof of inclusion have time complexity $O(h) = O(\log N)$, where $N = 2^h$.

Space Complexity The space complexity is $O(n \log N)$, where n is the number of non-default leaves.

Integration of SMTs with BOCs

In Overpass Channels, BOCs are stored as leaves in an SMT, representing state transitions. The Merkle root of the SMT provides a cryptographic commitment to all off-chain state transitions.

State Data as BOCs Each state transition S_i is serialized into a BOC B_i, hashed, and stored in the SMT:

$$l_i = H(B_i)$$

Merkle Root for State Verification The Merkle root r is periodically submitted to the blockchain, serving as a verifiable commitment to the off-chain state transitions.

Example: Payment Channel Update with SMTs

Consider a payment channel between Alice and Bob with an initial SMT root r_0. Alice sends 20 tokens to Bob, resulting in a new state S_1.

1. **State Update**: Serialize S_1 into BOC B_1.
2. **Leaf Update**: Compute $l_1 = H(B_1)$.
3. **Path Update**: Recompute hashes along the path to update the root r_1.
4. **Commitment to Blockchain**: Submit r_1 on-chain for verification.

Security Theorem for SMTs

Theorem 4 (Security of State Transitions in SMTs)**.** *The probability of an adversary successfully tampering with an SMT without detection is negligible, given the collision resistance of the hash function H.*

Proof. An adversary would need to find a collision in H to alter the SMT without changing the root r. Given H is collision-resistant, the probability is negligible. ■

Computational Complexity Analysis

Theorem 5 (Computational Complexity of Off-Chain Transactions)**.** *The computational complexity of processing an off-chain transaction in Overpass Channels is $O(\log n)$, where n is the number of transactions in the channel.*

Proof. Processing involves creating BOCs, generating zk-SNARK proofs, and updating SMTs, each with $O(\log n)$ complexity due to the height of the SMT. ■

Theorem 6 (On-Chain Efficiency)**.** *The on-chain computational cost for Overpass Channels is $O(1)$ per global state update, regardless of the number of off-chain transactions processed.*

Proof. On-chain operations involve verifying submitted Merkle roots and updating on-chain state, both of which are $O(1)$ operations. ■

These theorems highlight the scalability advantages of Overpass Channels compared to traditional on-chain transaction processing