Traffic-aware Merkle Trees for Shortening Blockchain Transaction Proofs

Avi Mizrahi, Noam Koren, Ori Rottenstreich and Yuval Cassuto

***Abstract*—Merkle trees play a crucial role in blockchain networks in organizing network state. They allow proving a particular value of an entry in the state to a node that maintains only the root of the Merkle trees, a hash-based signature computed over the data in a hierarchical manner. Verification of particular state entries is crucial in reaching a consensus on the execution of a block where state information is required in the processing of its transactions. For instance, a payment transaction should be based on the balance of the two involved accounts. The proof length affects the network communication and is typically logarithmic in the state size. In this paper, we take advantage of typical transaction characteristics for better organizing Merkle trees to improve blockchain network performance. We focus on the common transaction processing where Merkle proofs are jointly provided for multiple accounts. We first provide lower bounds for the communication cost that are based on the distribution of accounts involved in the transactions. We then describe algorithms that consider traffic patterns for significantly reducing it. The algorithms are inspired by various coding methods such as Huffman coding, partition and weight balancing. We also generalize our approach towards the encoding of smart contract transactions that involve an arbitrary number of accounts. Likewise, we rely on real blockchain data to show the savings allowed by our approach. The experimental evaluation is based on transactions from the Ethereum network and demonstrates cost reduction for both payment transactions and smart contract transactions.**

***Index Terms*—Blockchain, Merkle Trees, Coding Theory.**

I. Introduction

THE blockchain technology received high public attention in recent years as a distributed secured ledger with no central authority behind it. It became popular as the technology behind cryptocurrencies such as the popular Bitcoin [1] and has been expanded to other applications including electronic voting, supply chain communications, and medical informatics [2], [3]. Following the first (genesis) block, each later block includes the output of a cryptographic hash function computed over the content of the previous block, making it impossible to alter a block without changing all subsequent blocks. Blocks are comprised of a list of transactions, which imply atomic state modifications. These can be simple payments (money transfers) or a complex state change of virtual machine throughout the execution of Turing-complete smart contracts.

The *Merkle tree* is a known tool in cryptography, first suggested by Merkle [4] which enables efficiently proving membership of a data element in a set, without revealing the entire set. In a Merkle tree, every node has a *Merkle label*. For the leaves, this label is the hash of a data item, and for every non-leaf node, this label is the hash of the concatenation of the labels of its children. In order to verify that some data is included in a Merkle tree, one needs to obtain a label R of the root of the tree, called the Merkle root, from a trusted source. A Merkle proof for the containment of some data x, which corresponds to a leaf in the tree, consists of the siblings path of the leaf, that includes the labels for the siblings of the nodes in a path from the leaf to the root. These values allow the verifier to compute the Merkle root, checking the validity of values including x based on matching with the known value R. Merkle trees make use of hash functions which are second preimage resistant [5]. Given a data set, it is difficult to find an alternative set such that the Merkle trees of the two sets have the same Merkle root label.

Merkle trees play a fundamental role in blockchain technology, allowing the secure verification of transactions. The network state can include for each account its current balance and additional associated information. Due to the large size of the state, nodes typically do not maintain their complete copy. Each block in the chain is associated with the Merkle root computed for the network state following the execution of the block transactions. The block approval process includes verifying the validity of transactions and their impact on the state. For instance, a payment requires a minimum value of the payer's balance, and when successful, it implies a change in the balance of the two involved accounts. When a node proposes a block, it associates with it the implied impact of its transactions on the state. These can be validated through proofs for the involved inputs to the transactions, such as the balance of the two accounts in a typical payment transaction.

Typically, accounts are organized in the Merkle tree arbitrarily, e.g. in an order determined by their associated addresses. Moreover, all accounts appear in the same tree height. On the contrary, we observe that letting the Merkle tree structure be traffic-aware with potential diversity in the heights of accounts allows savings in the amount of data provided for the membership proofs. In this paper, we *study an approach for traffic-aware construction of Merkle trees*. We aim to take advantage of a common property of Merkle proofs in blockchain networks where a proof jointly refers to several inputs such as multiple accounts in the same transaction.

The paper makes the following main contributions:

- We overview use-cases of Merkle trees in Blockchain systems such as in the Ethereum network (Section II).
- We suggest and formalize a traffic-aware encoding approach for shortening Merkle proof lengths (Section III).
- We present bounds on the communication cost and relate

Avi Mizrahi, Noam Koren, Ori Rottenstreich and Yuval Cassuto are with the Technion - Israel Institute of Technology, Israel (emails: avraham.m@cs.technion.ac.il, noam.koren@campus.technion.ac.il, or@technion.ac.il, ycassuto@ee.technion.ac.il).

the encoding problem to known problems (Section IV).
- We describe practical considerations and an implementation architecture in Section V.
- We develop algorithms for finding efficient traffic-aware Merkle trees (Section VI).
- We generalize the communication cost to smart contracts involving a large number of accounts (Section VII).
- We study characteristics of real Ethereum data and conduct experiments to demonstrate the effectiveness of the approach (Sections VIII-A-VIII-B).

A preliminary version of this paper appeared in the International Conference on Communication Systems and Networks (COMSNETS), January 2021 [6].

II. Merkle Trees in Blockchain and Related Work

In this section, we detail several use-cases of Merkle trees in blockchain systems and related applications.

In blockchain networks, a block is associated with a hash value computed over data of previous blocks making them immutable [7]. In addition, typically (such as in Ethereum [8]) a block also includes a Merkle root value computed over the state following the execution of the block. This is illustrated in red in the chain shown in Fig. 1. An agreement on the addition of a new block to the chain implies a specific Merkle root value so a node can demonstrate values in the state by providing Merkle proofs to other nodes that do not maintain the full state. A block is validated by verifying the changes its transactions imply on the state. A typical transaction involves two accounts whose data appears as part of the state. Accordingly, the validation examines proofs for the accounts involved in the transaction.

The Ethereum network [8] is often considered the second-largest blockchain network. Every block header in Ethereum contains three Merkle roots of three trees: a transaction tree that contains the transactions in the block; a receipt tree with the impact of these transactions; and a state tree that contains the state of the Ethereum virtual machine. The state Merkle tree is a key-value mapping where the account addresses are keys and values include the balance and potentially code or storage assigned for the account. The complete tree structure is stored only in a local database maintained by the network nodes, and only the root hash is stored on the blockchain. A transaction is verified by supplying the Merkle paths for the updated accounts (as described in details as part of the model description in Section III). Additionally, when the state is modified as a result of transaction execution, the paths from the changed accounts to the root are updated. If accounts that are frequently accessed together become closer in the state tree, fewer tree parts have to be read and updated as a result of their overlapping paths to the root.

Another application of Merkle trees in blockchain systems is the protection of data integrity in cloud systems. A solution suggested in [9] is to partition the data stored on the cloud server and keep a unique tag for each part in a T-Merkle tree, a combination of a Merkle tree and a T-tree, a balanced search tree designed for main memory databases. The Merkle tree is kept on a blockchain and is used to supply proofs for the integrity of the data stored by the cloud server. While their suggested verification process uses a random set of tags that results in random proof paths in the Merkle tree, user-based integrity proofs can be considered. When proofs are requested according to some user access pattern, the distribution for batches can be deduced allowing the applicability of one of the efficient Merkle tree algorithms we propose in this paper.

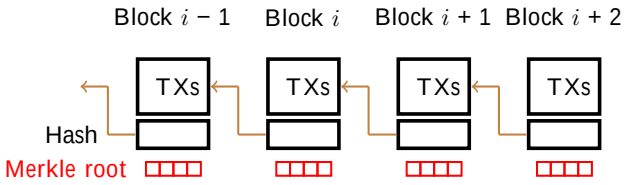


Fig. 1. Merkle roots as part of the block info. Besides the hash of a previous block, a block includes a Merkle root computed for the updated network state.

SmartSync [10], [11] is a recent work that aims to synchronize smart contracts between different blockchains. Their solution is applicable for account-based systems that use Merkle trees to hold their state, similarly to Ethereum. In the synchronization process, they used a multi-proof aggregation to save commonalities between the different proofs. Similarly, multi-proof is also a design choice in CBCS [12], a recent blockchain architecture. Both solutions can benefit from Merkle trees that produce smaller multi proofs on average. An energy efficient Merkle tree is designed in [13] to ease the burden of the energy consumption of blockchain systems.

Beyond blockchain, the Huffman Merkle Hash Tree (HuffMHT) [14] is used to make an efficient certificate revocation system. It minimizes the average length of the Merkle proof, given the probabilities of the elements, the same way as the Huffman tree constructed. In our domain, it can yield a shorter path to accounts that make transfers more frequently. However, it does not consider gains from overlapping proofs. The full details can be found later in Algorithm 2.

Compact Merkle multi-proofs [15] is a recent work that eliminate the overhead of the indices required to store non-leaf hashes in a multi-proof. This optimization is applicable to our proposed algorithms as well, and can further improve the efficiency of proof aggregation.

Additional applications of the Merkle Tree and its membership proof technique have been suggested [16], [17]. It is used to reduce the computational costs of using public-key digital signatures in securing routing protocols [18]. It can also be used in client-server protocol for web servers using SSL/TLS that minimizes the latency and improves resistance to denial-of-service attacks [19]. Help secure smart grid communication [20] and more. Such systems may improve communication costs if a proof batching is applicable.

III. Model and Problem Definition

A. Illustrative Example

A Merkle tree is illustrated in Fig. 2. It is computed for eight data items $x_0 \quad x_7$ (shown as leaves). Internal nodes including the Merkle root R are associated with hash values computed hierarchically. Based on R, to show the inclusion

TABLE I
SUMMARY OF MAIN NOTATIONS

Symbol	Meaning
a_i	account i
n	number of accounts
$Q = fq_1; : : : ; q_m g$	transaction distribution (appearance probabilities)
$( )$	prefix code
$j \ (a_i)j$	codeword length for account a_i
$c(tx; \ )$	multipath size for code with transaction tx
$C(Q; \ )$	encoding cost for code with transaction distribution Q
$OPT(Q)$	optimal encoding cost for transaction distribution Q
H_Q	accounts entropy for distribution Q
L_Q	Huffman code average length
$G = (V; E; w)$	transaction graph with weights describing transaction probabilities
	ratio of transactions serving as input to tree computation

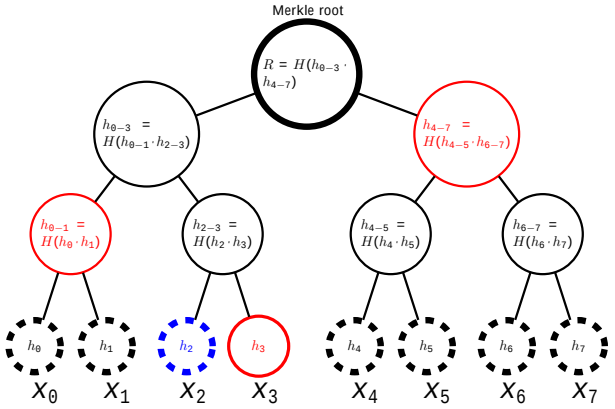


Fig. 2. Illustration of a Merkle tree of 8 items. Merkle proof of item x_2 (with a dashed blue leaf) includes nodes h_3, h_{0-1}, and h_{4-7} (in solid red).

among the values of a particular data items, x_2 for instance, $x_0 \ x_1$ and $x_3 \ x_7$ are useful. Together with x_2 one can check that they jointly imply R. For the same purpose, it is enough to provide only three values (shown in red in the figure): (i) h_3 (ii) $h_{0-1} = H(h_0 \ h_1)$ (iii) $h_{4-7} = H(H(h_4 \ h_5) \ H(h_6 \ h_7))$, where for $i \ 2 \ [0, 7]$, h_i denotes the values of $H(x_i)$. With these three values and x_2, the root value R can be computed as $R = H \ H \ h_{0-1} \ H \ H(x_2) \ h_3 \ h_{4-7}$.

For a tree computed for $m = 2^W$ items representing accounts, the inclusion proof includes W values as the length of the siblings path. To show the existence of multiple items, one can detail the siblings' path of each of them. Note that paths for different items can have nodes in common. In such a case, showing the inclusion of k items requires providing less than $k \ W$ values. For instance, in the example from Fig. 2 with $W = 3$ the proof path for x_1 is h_0, h_{2-3}, h_{4-7} and for x_2 is h_3, h_{0-1}, h_{4-7}. When both proofs are required, it is enough to provide only five values: $h_0, h_3, h_{0-1}, h_{2-3}, h_{4-7}$, as both share the node h_{4-7}. Moreover, neither h_{0-1} nor h_{2-3} are needed for the proof, as both can be derived by their leaves. Accordingly, jointly showing the inclusion of x_1, x_2 requires only 3 values, h_0, h_3, h_{4-7} such that $R = H \ H \ H \ h_0 \ H(x_1) \ H \ H(x_2) \ h_3 \ h_{4-7}$.

The potential savings are affected by the *location* of the items in the tree. For instance, upon showing the inclusion of x_2 and x_5, there are no savings since the nodes along their paths do not overlap. Moreover, as mentioned, the degree of freedom in the heights of items can also be used to shorten proofs for common accounts.

Table I summarizes the main notations of the paper.

B. Model

We present a model for the encoding of accounts with Merkle trees and formalize the problem of minimizing the communication cost in the processing of a transaction. We assume here that a transaction has two (distinct) accounts as in the common case of payment transactions.

Definition 1 (Transaction)**.** A *transaction* $tx \ , \ fa_1, a_2 g$ is characterized by a set of two accounts. Transactions are equivalent if they contain the same accounts.

In Section VII, we generalize the discussion to smart contracts transactions that can involve an arbitrary number of accounts.

Definition 2 (Transaction Distribution)**.** A *transaction distribution* $(TX, Q) \ , \ (ftx_1, \ldots, tx_m g \ , \ fq_1, \ldots, q_m g)$ is characterized by a set of transactions with their corresponding positive appearance probabilities. A transaction is drawn randomly according to the distribution Q, i.e., $P(tx = tx_i) = q_i$, with $q_i > 0$ and $\sum_{i=1}^{m} q_i = 1$.

Definition 3 (Account Encoding Function)**.** An *account encoding function* σ is a mapping $\sigma : A \ ! \ f0, 1g^{\leq n} \ n \ ;$, where A is the set of $jAj = n$ accounts and $f0, 1g^{\leq n}$ is the set of all binary vectors of length less than or equal to n.

Definition 4 (Prefix Code)**.** For a set of items S, code σ is called a *prefix code* if in its codeword set B, no codeword is a prefix (start) of any other codeword.

Kraft's inequality [21] formally expresses whether a prefix code of given codeword lengths exists: A code with n codewords of lengths $\ell_1, \ell_2, \ldots, \ell_n$ exists if and only if the inequality $\sum_{i=1}^{n} 2^{-\ell_i} \ 1$ holds.

The codeword lengths of σ imply a binary tree structure where each leaf is associated with a codeword obtained according to the path from the root with left and right edges corresponding to bits of 0 and 1, respectively. If leaves share the same height then all codewords have the same length. Consider a Merkle tree where n items appear in heights based on codewords lengths of a prefix code σ. To show the inclusion of an item x among the items in the tree, a proof includes $j\sigma(x)j$ items. Denote by $\sigma(x) = b = (b_1, \ldots, b_{| \ (x)|})$ the (binary) codeword for x. The proof includes values that correspond to nodes in the tree in locations $(b_1, \ldots, b_{i-1}, 1 \ b_i)$ for $i \ 2 \ [1, j\sigma(x)j]$. For instance, in Fig. 2 the proof for the inclusion of x_2 (with a location reached by a path corresponding to 010) was composed of h_3 (of location 011), h_{0-1} (of location 00) and h_{4-7} (of location 1).

The following definition refers to the communication cost in the processing of a transaction that involves two accounts.

Definition 5 (Multipath size)**.** For a transaction $tx = \{a_1, a_2\}$ and an account prefix encoding σ, the *multipath size* is defined as the size of the union of the paths of a_1 and a_2

$$c(tx, \sigma) = |\sigma(a_1)| + |\sigma(a_2)| - |\sigma(a_1, a_2)| \ ,$$

where $\sigma(a_1, a_2)$ is the common prefix of $\sigma(a_1)$ and $\sigma(a_2)$.

Definition 6 (Communication Cost)**.** For a transaction distribution (TX, Q) and an account prefix encoding σ, the *communication cost* (measured in units of bits) is defined as the average multipath size $C(Q, \sigma) = \sum_{tx \in TX} q_{tx} \cdot c(tx, \sigma)$, where q_{tx} is the probability of transaction tx.

The problem of constructing a Merkle tree that minimizes the communication cost can be formalized through finding a corresponding legal prefix code.

Problem 1 (Transaction Encoding Problem)**.** Given a transaction distribution (TX, Q) defined over a set of accounts A, find a prefix code for A that minimizes the communication cost. Namely,

$$\min \sum_{i=1}^{n} q_i \cdot |\sigma(a_{i,1})| + |\sigma(a_{i,2})| - |\sigma(a_{i,1}, a_{i,2})|$$

$$\text{s.t.} \sum_{a \in A} 2^{-|\sigma(a)|} \leq 1 \tag{1a}$$

$$|\sigma(a)| \geq 1 \quad \forall a \in A \tag{1b}$$

$$|\sigma(a)| \in \mathbb{N} \quad \forall a \in A \tag{1c}$$

For a transaction distribution Q we denote by $OPT(Q)$ the optimal encoding cost, i.e., the value $\min C(Q, \sigma)$ where σ is a prefix code.

Note that the communication cost can be reduced below the multipath size $c(tx, \sigma) = |\sigma(a_1)| + |\sigma(a_2)| - |\sigma(a_1, a_2)|$ for transactions of pairs. We can discard two additional items from the proof: the items located on the level that they are first different at, i.e. corresponds with the first-bit difference in their codewords. This is since we can calculate both by the other proof's subtree, as demonstrated in Section III-A. For convenience, we choose the simpler term of Problem 1 as the minimization target as they differ for simple transactions of two accounts by a constant (2 items), leading to equivalent optimization. In Section VII we generalize the discussion to smart-contract transactions that can involve more than two accounts. We then explain for this extension the potential impact on the communication cost.

IV. Basic Properties and Bounds

Towards developing solutions to the problem we study in this section its fundamental properties.

A. Accounts Order in a Transaction

A transaction is often described as an ordered pair of accounts to distinguish between the roles of a payer and payee. We simply observe that the encoding problem is insensitive to that order. This immediately follows the symmetry between the two accounts in Definition 1. This enables us to simplify the representation of a distribution while referring to transactions as unordered. We do so through the representation of a distribution as an unordered weighted graph as follows.

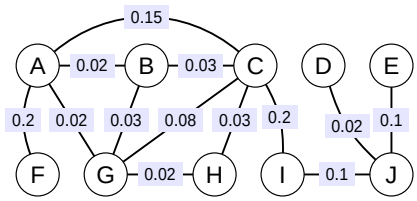


Fig. 3. Transaction graph example with $n = 10$ accounts. Every node represents an account, and a weighted edge describes the probability for a transaction between two accounts.

B. Graph Representation

Definition 7 (Transaction Graph)**.** A *transaction graph* $G = (V, E, w)$ is an undirected weighted graph. Nodes represent accounts.There is an edge $e = (v_1, v_2)$ if a transaction between accounts v_1 and v_2 can appear. The weight $w(e)$ is the probability of a transaction to involve the accounts of e.

Through the graph representation of the transaction distribution, the communication cost of a (prefix) code σ can be expressed as $C(G, \sigma) = \sum_{e \in E} w(e) \cdot |\sigma(e_1)| + |\sigma(e_2)| - |\sigma(e_1, e_2)|$ using the notation $e = (e_1, e_2)$. Fig. 3 shows an example for a transaction graph with ten accounts and thirteen potential transactions.

Assuming a transaction graph $G = (V, E, w)$ we denote by $OPT(G)$ the optimal encoding cost, i.e., the value $\min C(G, \sigma)$ where σ is a prefix code.

C. Bounds on the Communication Cost

We derive lower and upper bounds for the optimal communication cost $OPT(Q)$ for a given distribution Q. The distribution of the transactions Q implies a distribution of the appearance of accounts in the transactions. An account can appear as one of the two accounts in a transaction.

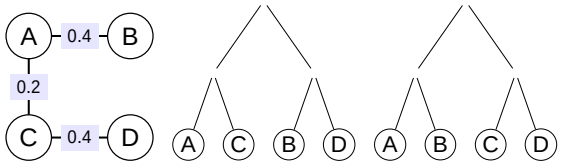
Definition 8 (Account Distribution)**.** An *account distribution* (A, P), $((a_1, \ldots, a_n), (p_1, \ldots, p_n))$ is characterized by a list of accounts with their corresponding positive appearance probabilities. For a transaction distribution (TX, Q) an account a_i has probability $p_i = \frac{1}{2} \sum_{tx = \{a_1, a_2\} \in TX} q_{tx} \cdot I(a_i \in tx)$.

A trivial upper bound on the communication cost derives from a code σ that allocates codewords of fixed length $\log_2(n)$ for the n accounts, namely $|\sigma(a_i)| = \log_2(n)$ for $a_i \in A$. The cost for a transaction is at most twice the fixed length bounding $C(Q, \sigma)$ and accordingly $OPT(Q)$.

Property 1. The optimal communication cost for a transaction distribution Q defined over n accounts satisfies $OPT(Q) \leq \log_2(n)$.

Note that some additional savings can always be achieved by assigning codewords with a non-empty common prefix to accounts in the most common transaction. Accordingly, the inequality in the above can be strong $OPT(Q) < \log_2(n)$.

Following the account distribution, we define two values. First, accounts entropy H_Q is defined as $H_Q = \sum_{i=1}^{n} p_i \log_2(p_i)$. Likewise, its Huffman code average



(a) Transaction distribution (b) Huffman (c) Optimal

Fig. 4. An example for a transaction distribution defined over four accounts (described by the graph in (a)) and corresponding Huffman-based code (in (b)) and an optimal code (in (c)).

length L_Q is defined as the weighted average codeword length for a Huffman code [22], computed over the account distribution (A, P). It is known that $H_Q \quad L_Q \quad H_Q + 1$ such that $L_Q = H_Q$ if for $i \ 2 \ [1, n]$ probabilities are of the form $p_i = 2^{-k_i}$ for some integer k_i.

The account distribution (A, P) computed for the transaction distribution Q helps us to derive bounds for $OPT(Q)$.

Theorem 2. Let (TX, Q) be a transaction distribution with a corresponding account distribution (A, P). The optimal communication cost satisfies $L_Q + 1 \quad OPT(Q) \quad 2 \ L_Q$.

Proof: Consider a prefix code σ for the transaction distribution. For a transaction $tx_i = (a_{i;1}, a_{i;2})$ w.p. q_i, the communication cost is $c_i = j\sigma \ a_{i;1} \ j + j\sigma \ a_{i;2} \ j \quad j\sigma \ a_{i;1}, a_{i;2} \ j$. It satisfies $c_i \quad j\sigma \ a_{i;1} \ j + 1$ and $c_i \quad j\sigma \ a_{i;2} \ j + 1$. First, for the lower bound, the communication cost $C(Q, \sigma)$ for any code σ is $\sum_{i=1}^{m} q_i \quad j\sigma \ a_{i;1} \ j + j\sigma \ a_{i;2} \ j \quad j\sigma \ a_{i;1}, a_{i;2} \ j$ $\sum_{i=1}^{m} q_i \ j\sigma \ a_{i;1} \ j + 1 \quad L_Q + 1$, where the last inequality follows properties of the Huffman code, achieving the optimal (minimal) average codeword length. To see the upper bound, let σ be the Huffman code itself for the account distribution (A, P) computed for Q. Then, its communication cost is $C(Q, \sigma) = \sum_{i=1}^{m} q_i \quad j\sigma \ a_{i;1} \ j + j\sigma \ a_{i;2} \ j \quad j\sigma \ a_{i;1}, a_{i;2} \ j$ $\sum_{i=1}^{m} q_i \quad j\sigma \ a_{i;1} \ j + j\sigma \ a_{i;2} \ j \quad = 2 \ L_Q$. This implies the upper bound on $OPT(Q)$. ■

Example 3. Fig. 4(a) illustrates a transaction distribution Q defined over four accounts A,B,C,D with three potential transactions. The probabilities for these transactions imply a distribution for the four accounts given as $(A, P) = (A, B, C, D), \ (0.4 + 0.2)/2, 0.4/2, (0.2 + 0.4)/2, 0.4/2 = ((A, B, C, D), (0.3, 0.2, 0.3, 0.2))$. The accounts entropy is $H_Q \quad 1.97$. An example for a Huffman code is illustrated in Fig. 4(b), with a fixed codeword length of 2 implying $L_Q = 2$. An optimal code is illustrated in Fig. 4(c). This implies cost of 2+2-1=3 for (A,B) (as A,B both appear in the left half of the tree), a cost of 2+2=4 for (A,C) (as A,C appear in two different halves of the tree) and a cost of 2+2-1=3 for (C,D) such that the average cost is $OPT(Q) = 0.4 \ 3 + 0.2 \ 4 + 0.4 \ 3 = 3.2$. We can see that the inequalities $L_Q + 1 = 3 \quad OPT(Q) = 3.2 \quad 2 \ L_Q = 4$ indeed hold.

By the communication cost of the Huffman code and the lower bound on the communication cost of any potential code from the last proof, we deduce that the Huffman code approximates the optimal code with a factor smaller than 2.

Property 4. For any transaction distribution (TX, Q) the Huffman code has a communication cost $C(Q, \sigma) \quad \frac{2 \cdot L_Q}{L_Q + 1} \ OPT(Q) < 2 \ OPT(Q)$.

Let σ be the above-mentioned Huffman code computed for the account distribution (A, P) implied by Q. Such a code neglects the probabilities of the various accounts to appear jointly in a transaction. Accordingly, while it allocates to accounts codewords of a particular length, it does not try to allocate to accounts that often appear jointly in a transaction codewords with a lengthy common prefix.

Considering the distribution of transactions (TX, Q) and not just the account distribution (A, P) allows us to further improve the upper bound $OPT(Q) \quad 2 \ \log_2(n)$ on the optimal communication cost from Property 1 but with more tedious expressions. The idea is to consider the transactions of highest weights and allocate codewords with a long common prefix to each pair of accounts from these transactions. Consider some $n/2$ transactions with n distinct accounts. Each such transaction $tx_i = \{a_1, a_2\}$ with probability q_i helps to reduce the upper bound of $2 \ \log_2(n)$ on the optimal communication cost by $q_i \ (\ \log_2(n) \quad 1)$ through allocating for the two accounts codewords that share all bits but the last. Note that we cannot always refer to the $n/2$ transactions of highest probability and potentially need to consider transactions of lower weights for the transactions to be of disjoint accounts. Note that the same property that allows the additional savings is not necessarily valid for the upper bound $2 \ L_Q$ from Theorem 2. The reason is that the particular codeword lengths in a Huffman code imply restrictions on the potential of two accounts to have a lengthy prefix in common.

D. Connection to Known Problems

We relate the transaction encoding problem (Problem 1) to the Quadratic Assignment Problem (QAP) from the field of facilities location problems [23].

Problem 2 (Quadratic Assignment Problem)**.** Given are N facilities and a demand matrix $D_{N \times N}$ where $D_{i;j}$ describes the amount of traffic sent from facility i to facility j. A graph $G = (V, E)$ with $|V| = N$ is given with a known travel cost between any pair of nodes. The goal is to map facilities to nodes such that the weighted average travel cost is minimized.

The transaction encoding problem allows flexibility in the codeword lengths (while satisfying the Kraft's inequality).

Property 5. While referring to trees of a known structure, e.g. the tree with leaves of a fixed height, the encoding problem of Problem 1 is an instance of the Quadratic Assignment Problem (QAP).

In our context, the distance between two nodes reached through paths of the form $\sigma(a_1), \sigma(a_2)$ is $\sigma(a_1) \ + \ \sigma(a_2) \quad 2 \ \sigma(a_1, a_2)$. While the QAP problem

is NP-hard in the general case, our hope is that particular graph properties might enable finding efficient solutions.

Next, we relate the transaction encoding problem (Problem 1) to another known problem. Interestingly, this connection allows better understanding of the computational hardness of Problem 1, again assuming a restriction that the tree is a complete tree with leaves of a fixed height.

Problem 3 (Data Arrangement in a Tree [24])**.** Given are an undirected weighted graph $G = (V, E, w)$ and an integer B 0. Determine whether there exists an injective mapping f from V to the leaves of a complete d-ary tree T of height $\lceil \log_d |V| \rceil$, such that

$$\sum_{e \in E} w(e) \; d_T \; f(e_1), f(e_2) \qquad B$$

where $d_T(u, v)$ is the distance between u, v in T.

The Data Arrangement problem is NP-hard even when $w(e) = 1$ for all e 2 E and for any d 2 [24]. In our terms, when considering fixed-length codewords, the problem is hard even without considering the transaction distribution. Although this problem uses the length between leaves, as opposed to the total length of the union of the paths from the root as in Problem 1, each can be solved by the other when adding the constraint of fixed length codewords to Problem 1. In a complete tree, our problem is equivalent to

$$\begin{aligned} C(G, \sigma) &= \min \sum_{i=1} \log_2 n + \log_2 n \quad \sigma \; a_{i;1}, a_{i;2} \\ &= m \log_2 n + \min \sum_{i=1} \log_2 n \quad \sigma \; a_{i;1}, a_{i;2} \\ &= m \log_2 n + \frac{1}{2} \min \sum_{e \in E} d_T \; f(e_1), f(e_2) \end{aligned}$$

since $\log_2 n \quad \sigma \; a_{i;1}, a_{i;2}$ is half the distance (d_T) between the leaves of $\sigma \; a_{i;1}$ and $\sigma \; a_{i;2}$. We can see that a solution to Problem 1 with fixed length codewords can be applied to Problem 3, and given a solution to Problem 3, we can search for the minimal value B and get $C(G, \sigma)$.

V. Practical Considerations of Traffic-Aware Merkle Trees

We explore practical tradeoffs of adopting Traffic-aware Merkle trees in blockchain systems. Specifically, we consider the implications of computational and storage overhead towards the expected savings in communication bandwidth. We start by presenting a system with Traffic-aware Merkle trees that shares the main design with the Ethereum system and point to the overhead and savings.

This system architecture would include a reconfiguration protocol to ensure that the Merkle tree remains optimized for recent traffic patterns. This protocol can be a simple block count for reconstruction of the tree, or it could be based on detecting significant changes in the traffic that warrant a reconfiguration. The reconfiguration process, as illustrated in Fig. 5, involves feeding a predefined algorithm for tree

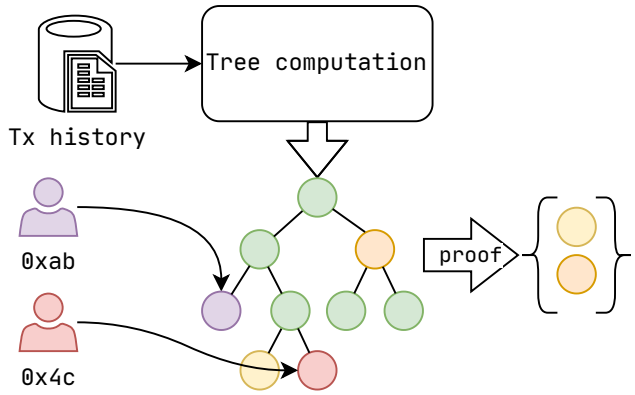


Fig. 5. Tree computation and usage. Given a transaction history (a sequence of blocks), a new Merkle tree is computed. Later, a proof is generated for a transaction between two accounts.

computation (such as among those from Section VI) with a transaction history from previous blocks and outputting an updated Merkle tree. Nodes adopt the updated tree structure until the next reconfiguration is triggered. It is noted that a balance should be implied between frequent updates, which could lead to higher operational costs, and infrequent updates, which may result in suboptimal Merkle trees that do not reflect the latest transaction patterns.

Nodes independently execute the protocol without storing state information on the blockchain, thus not adding additional storage requirements for the blockchain itself. This can be done thanks to the shared historical transaction data that serves as a common reference point for all nodes. This commonality ensures that every Merkle tree independently constructed by a node is consistent with trees of other nodes, obviating the need for an additional consensus protocol.

Fig. 6 demonstrates the evolution of the Merkle tree structure as transactions are added to the blockchain. When transactions are processed, the state of the tree is modified and the hashes are updated in the paths to the root, and the new root is kept in the block header. In these updates, using a traffic-aware tree saves node updates in the tree, as frequent accounts have a shorter path. When a reconfiguration occurs, a new tree structure is computed and its root is kept on chain as before.

Additional storage is required for mapping account addresses to codewords (nodes in the tree), but this is stored locally at each node and recalculated as necessary, thus it does not impact the blockchain's size. However, the size of the map depends not only on the number of accounts but also on the tree structure. For a complete binary tree, each codeword is of length $\log_2 n$, and in the extreme case of a degenerate tree the average codeword length is $O(n)$. In practice, one way to reduce the mapping cost in systems with a large number of accounts is to consider only the most frequent accounts in a traffic-aware tree, and to maintain another regular tree for the rest. This approach has the potential to balance the computational and storage trade-offs, ensuring that the blockchain remains scalable even as the number of accounts grows. This approach leverages the typical heavy-tailed distribution of transactions in blockchains, as explored

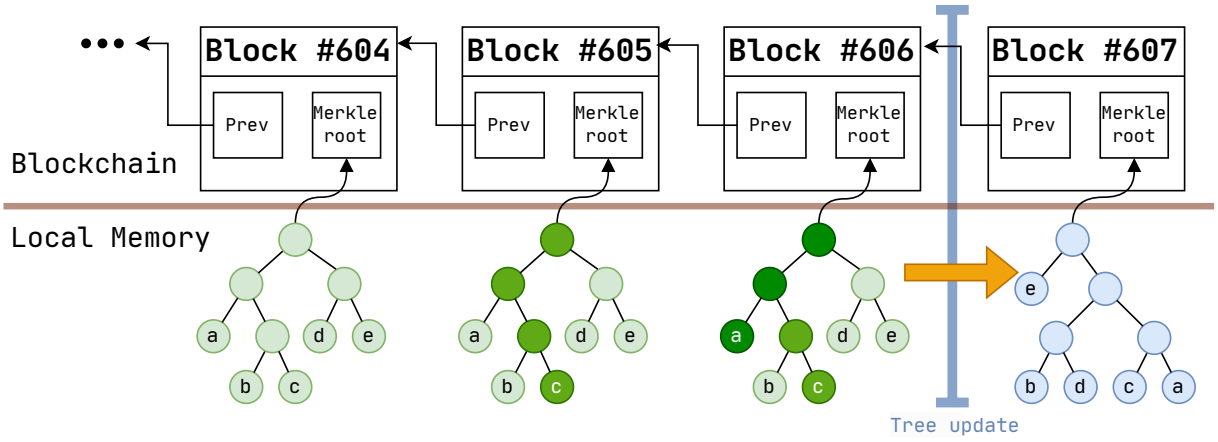


Fig. 6. Illustration of the dynamic nature of the Merkle tree as it evolves over time. In the first three blocks in the diagram, the tree keeps its structure, but hash values are changed along paths from accounts root for the individual transactions. By the fourth block, a reconfiguration of the tree occurs and a new structure is established. The updated tree structure is computed by one of the algorithms for tree computation based on recent transaction history.

in Section IV and in a similar work on traffic-aware algorithms in blockchain systems [25].

VI. Algorithms for Computing Traffic-aware Merkle Trees

Although the previous discussion suggests that it is likely hard to solve Problem 1 optimally for every instance, in this section we present efficient algorithms, and later in Section VIII-B demonstrate their performance improvements on realistic distributions. We describe various algorithms for finding a code σ for a given transaction distribution (TX, Q). The algorithms differ in several aspects, including (i) The codeword lengths they produce (either fixed or variable length) (ii) The information they rely on, such as the complete transaction distribution or only the account distribution.

Algorithm 1: Random (Fixed length codewords)

Input: Transaction distribution (TX, Q); Accounts $A = \{a_1, \ldots, a_n\}$
Output: Codewords per account $\sigma = \{\sigma_1, \ldots, \sigma_n\}$
return $\sigma(a_i) = bin(i)$/* $\log_2(n)$ -bit value */

The first simple algorithm allocates for all n accounts fixed length codewords of the minimal possible length of $\log_2(n)$ bits. Pseudo-code is shown in Algorithm 1. While the particular assignment of codewords affects the communication cost according to the transaction distribution, this simple algorithm does not use the transaction distribution nor the implied account distribution. The algorithm simply uses the standard binary representation to assign codewords to accounts.

The second algorithm (Algorithm 2) uses the account distribution implied by the input transaction distribution and is based on the Huffman Merkle Hash Tree (HuffMHT) [14]. The algorithm does not further make use of the informative transaction distribution. It assigns codewords of variable lengths to accounts as a Huffman code [22] based on each account probability. Accounts participating in more transactions are assigned shorter codewords such that the code minimizes the average codeword length based on the account distribution.

Algorithm 2: Huffman [14] (Variable length codewords)

Input: Transaction distribution (TX, Q); Accounts $A = \{a_1, \ldots, a_n\}$
Output: Codewords per account $\sigma = \{\sigma_1, \ldots, \sigma_n\}$
1 Compute account distribution (A, P) from transaction distribution Q
2 **return** *Huffman code for A based on P* /* [22] */

Algorithm 3: Partition (Fixed length codewords)

Input: Transaction distribution (TX, Q); Accounts $A = \{a_1, \ldots, a_n\}$
Output: Codewords per account $\sigma = \{\sigma_1, \ldots, \sigma_n\}$
1 Refer to all accounts as a single group with an empty string codeword
2 **while** *maximal group size* > 1 **do**
3 Select the group of the largest size
4 Compute a balanced partition of the group accounts with minimal weight of crossing edges, separating accounts into two subgroups /* [26] */
5 For each account a_i add 0 or 1 to its codeword σ_i, based on its subgroup
6 **end**
7 **return** *Codewords per account* $\{\sigma_1, \ldots, \sigma_n\}$

The Partition algorithm (Algorithm 3) assigns codewords of a roughly fixed length to all the accounts. The fixed length is achieved by recursively partitioning the accounts into two subsets so as to minimize the probability that a transaction involves accounts in different subsets. Accounts are arranged in a balanced hierarchy, with all accounts in the same partition

sharing the same codeword prefix. Thus, as the probability of two accounts making a transaction grows, their codes are expected to share more of their prefix. The algorithm, which is based on a top-down partitioning, is calling a balanced min-cut algorithm n 1 times. For finding the balanced min-cut we make use of a heuristic for the problem of a balanced partition with minimal crossing edge weights. The heuristic was suggested by Kernighan and Lin in 1970 [26]. The reason for using such a heuristic is that finding an optimally balanced partition (line 4 of Algorithm) is NP-complete [27].

Algorithm 4: Pairs-first Huffman (Variable length codewords)

Input: Transaction distribution (TX, Q); Accounts $A = \{a_1, \ldots, a_n\}$
Output: Codewords per account $\{\sigma_1, \ldots, \sigma_n\}$

1 Initialize all account codewords by the empty string
2 Compute the account distribution (A, P) from the transaction distribution Q
3 $A' \quad A$, $P' \quad P$
4 For each transaction $tx_k = \{a_i, a_j\} \in TX$, compute relative weight for its two accounts
$w(tx_k) = \frac{q_k}{p_{a_i}}, \frac{q_k}{p_{a_j}}$
5 **while** TX *is not empty* **do**
6 $tx \quad$ most frequent transaction in TX
7 **if** $\min w(tx) > (\alpha, \alpha)$ **then**
/* Merge accounts to subtree */
8 Append '0' to the codeword σ_i of a_i
9 Append '1' to the codeword σ_j of a_j
10 Remove all tx with either a_i, a_j from TX
11 $A' \quad A' \setminus \{a_i, a_j\}$
12 $A' \quad A' \cup \{a_i, a_j\}$
13 $p'_{i;j} \quad p_i + p_j$
14 **end**
15 **end**
16 **return** *Huffman code* σ *for* A' *based on account distribution* P' /* [22] */

Each of the previous two algorithms addresses one part of the multipath-size expression: Algorithm 2 minimizes $\sigma(a_i)$ and Algorithm 3 maximizes $\sigma(a_1, a_2)$. The following algorithms attempt a joint optimization of the two parts, optimizing both the individual representation sizes and the overlaps between pairs.

In the following, we make adaptations to the Huffman algorithm (as of Algorithm 2) to consider also the distribution of transactions rather than only of accounts. Intuitively, in addition to assigning short codewords to frequently appearing accounts, we try to allocate long common prefixes to pairs of accounts with many joint transactions. In Algorithm 4, the input accounts are first processed. An alternative account distribution P' is computed based on (A, P) and serves as the Huffman algorithm's input. We merge pairs of accounts with frequent joint transactions. In doing so, we would like the transaction to take for each of its two accounts, a high ratio among all transactions the accounts are involved in.

Algorithm 5: Weight Balance [28] (Variable length codewords, ordered accounts)

Input: Transaction distribution (TX, Q); Accounts tuple $A = (a_1, \ldots, a_n)$
Output: Codewords per account $\sigma = \{\sigma_1, \ldots, \sigma_n\}$

1 Compute account distribution (A, P) from transaction distribution Q
2 Let $\quad$ be a list of n empty codewords.
3 $I \quad (A, \quad)$
4 **while** I *is not empty* **do**
5 $A', \quad I.pop()$
6 $l \quad A'$
7 **if** $l = 1$ **then**
8 $\sigma \quad A'_1 \quad 1$
9 **else**
10 $k \quad \arg\min_{k \in [1;l]} \left| \sum_{i=1}^{k} p_i \quad \sum_{i=k+1}^{l} p_i \right|$
11 $L \quad A'_1, \ldots, A'_k \quad , (0 \quad 1, \ldots, 0 \quad k)$
12 $R \quad A'_{k+1}, \ldots, A'_l \quad , (1 \quad k+1, \ldots, 1 \quad l)$
13 $I \quad I \setminus \{L, R\}$
14 **end**
15 **end**
16 **return** *Codewords per account* $\{\sigma_1, \ldots, \sigma_n\}$

In Algorithms 2-4, the order of accounts in the Merkle tree leaves is determined by the algorithm. However, in some cases it is desired to keep the accounts' binary representations in their original order, for example when consecutive account numbers form some organizational grouping. For this case, we suggest using a known algorithm, called Weight Balance [28], that optimizes the representation lengths while preserving the order of the inputs. It is shown here as Algorithm 5.

The algorithm recursively partitions the account space from top to bottom into two subsets of consecutive accounts. At each step, the partition point is chosen as the one that minimizes the difference between the total weights of the two subsets. We refer to the weight of a subset as the sum of the account frequencies in it. In each partition, the accounts of the left subset are assigned the prefix 0 (in addition to previously bits assigned to them), and the accounts on the right the prefix 1. The algorithm then continues on each of the subsets recursively such that an account is associated with the codeword given as a concatenation of its assigned bits. With this algorithm the communication cost of its tree is upper bounded by $C \quad Q, \sigma_{Alg:5} \quad H_Q + 2 \quad (n + 2) \min_{i \in [1;n]} p_i$ [28], where H_Q and p_i are the account entropy and account probability, respectively (as earlier defined in Section IV-C).

Next, we introduce an algorithm that combines partition and weight balancing. Algorithm 6 initially partitions the accounts according to Algorithm 3 (thus potentially reordering them), and subsequently applies the weight-balance algorithm (Algorithm 5) on the re-ordered list of accounts. This approach combines the benefits of both methods: grouping together accounts with high traffic based on transaction frequency (partition step), and optimizing communication cost within a

Algorithm 6: Partition and weight balance (Variable length codewords)

Input: Transaction distribution (TX, Q); Accounts $A = \{a_1, \ldots, a_n\}$
Output: Codewords per account $= \{_1, \ldots, _n\}$

```
1 Partition (TX, Q, A)                    /* Alg. 3 */
2 A' ← A sorted lexicography by
3 WeightBalance(TX, Q, A')                /* Alg. 5 */
4 return
```

predetermined account order (weight-balance step).

Example 6 (Algorithm illustration). Consider the transaction distribution shown in Fig. 7a with $n = 8$ accounts $A, B, \ldots, H$. There are seven potential transactions (shown by the edges), including a very frequent transaction between accounts B, C. Fig. 7b-7g present the codes computed by Algorithms 1-6 with their corresponding communication costs. First, in Fig. 7b we can see the result of Algorithm 1 assigning an arbitrary (unique) codeword of length $\log_2(n = 8) = 3$ (described by the path from the root) to each of the $n = 8$ accounts. Here, the accounts in the common transaction $\{B, C\}$ have codewords 010 and 100, respectively, without a common prefix so that the communication cost of the transaction is $3 + 3 = 6$. The communication cost of the code is 5.52.

Fig. 7c illustrates the code for the Huffman-based solution from Algorithm 2. It assigns shorter codewords to active accounts C, B of 0 and 10 with lengths of 1 and 2 bits. This reduces for instance the cost for the transaction between B and C to only $1 + 2 = 3$ and the cost of the code to only 3.85. Fig. 7d shows the code derived by the partition algorithm from Algorithm 3 in which again all codewords have a length of $\log_2(n = 8) = 3$. The minimal balanced cut divides accounts into two subsets $\{A, B, C, D\}$ and $\{E, F, G, H\}$ (illustrated in red in Fig. 7a). It performs additional partitions for lower levels of the tree and results in a cost of 4.48. Note that the optimal first cut for fixed-length codewords is $\{B, C, D, E\}$ and $\{A, F, G, H\}$, and when continuing with recursive optimal cuts the cost is 4.37. Fig. 7e presents the code of the modified Huffman algorithm from Algorithm 4 that tries to merge pairs of accounts with a high amount of transactions before using the Huffman code on the merged accounts. In this case, the algorithm first merges accounts B and C as they have the highest transaction frequency, then D and E, and so on. Here, it results in an improved cost of 3.82.

A potential output of Algorithm 5 is presented in Fig. 7f. Assume for instance that accounts must preserve the alphabetical order. The algorithm aims to derive such a codeword assignment with low communication cost. Its first partition of the account space is done after account B, as the difference between the probability of $\{A, B\}$ to the probability of the other accounts $\{C, D, E, F, G, H\}$ is minimal, $|0.385 - 0.615|$. Similarly, the next steps partition the space further for each of the partitions for the next level of the tree, yielding a cost of 4.11 for this example. Finally, Fig. 7g shows the result of Algorithm 6. In this example, the initial partition is the same as the one obtained in Fig. 7d. Then, the algorithm performs weight balancing on the ordered list of accounts: (A, D, B, C, E, F, G, H). In this case the minimal difference in the first weight balance step is achieved when $\{A, D, B\}$ are separated to the left. The final cost for this method is 4.44.

(a) Transaction distribution graph with eight accounts

(b) Random (Algorithm 1), cost = 5.52

(c) Huffman (Algorithm 2), cost = 3.85

(d) Partition (Algorithm 3), cost = 4.48

(e) Pairs-first Huffman (Algorithm 4), cost = 3.82

(f) Weight balance (Algorithm 5), cost = 4.11

(g) Partition and weight balance (Algorithm 6), cost = 4.44

Fig. 7. Illustration of the code construction algorithms. (a) shows the input transaction distribution. (b)-(g) present as a tree structure the codes computed by Algorithms 1-6.

VII. Smart Contract Transactions

For smart contract transactions, an arbitrary number of accounts (potentially larger than two) can be accessed by a transaction. Such transactions can be represented as a hypergraph. An edge again stands for a transaction but can involve more than two nodes. Smart contracts are supported by blockchain networks such as Ethereum [8]. We explain how to generalize the communication cost function for smart contract transactions accessing an arbitrary number of accounts.

A. Generalized model

Let stx be a smart contract transaction. We refer to stx as the set of accounts accessed by the transaction. Let $$ be a prefix code allocating a codeword for each account in stx. We refer to accounts in stx in an order implied by $$ such that

Fig. 8. Illustration of the communication cost for a smart contract transaction with 4 accounts $\{x_1, x_2, x_4, x_5\}$ (shown in dashed blue) in a Merkle tree of 8 data elements. The Merkle proof includes values from the union of the proofs for the four accounts. These are shown as nodes in red with a total of 9 values (in solid red).

codewords for the items are ordered in an increasing binary value. We express the multipath size of the transaction stx.

Theorem 7. Let stx be a smart contract transaction. Using code γ, let $a_1, \ldots, a_{|stx|}$ be the order of the accounts in increasing codeword values. The multipath size of the transaction with the code γ is

$$c(stx; \gamma) = \ell(a_1) + \sum_{j=2}^{|stx|} \left(a_j - a_{j-1} \right).$$

Proof: The particular order of the codewords allows us to easily compute the number of additional values that are required to be included in the proof but were not part of the proof for earlier codewords. This is exactly the length of the current codeword minus the length of its common prefix with the prefix codeword. ■

It is easy to see that the above formula generalizes the communication cost for a simple (rather than a smart contract) transaction of two accounts (with $|stx| = 2$) from Definition 6. Based on the communication cost for a single smart contract transaction, the cost for the distribution of such transactions can be derived. The following example illustrates the computation of the communication cost for smart contracts.

Example 8. Let stx be a smart contract transaction with four accounts $\{a_1, a_2, a_3, a_4\} = \{x_1, x_2, x_4, x_5\}$. Assume the accounts are allocated the four codewords 001, 010, 100 and 101, respectively. These are illustrated with leaves in dashed blue in Fig. 8. The proof includes for x_1: h_0, $h_{2\ 3}$ and $h_{4\ 7}$, for x_2: h_3, $h_{0\ 1}$ and $h_{4\ 7}$, for x_4: h_5, $h_{6\ 7}$ and $h_{0\ 3}$ and for x_5: h_4, $h_{6\ 7}$ and $h_{0\ 3}$. There are exactly nine values in total (shown in solid red) with a number given by $c = |001| + (|010| - 0|) + (|100| - |10|) + (|101| - |10|) = 3+2+3+1 = 9$. To see that, recall that the first and second codewords share the first bit of 0, the second and the third do not share any prefix bits, and the third and fourth have the same two first bits. Also note that the leaf values $h_4 = H(x_4)$ and $h_5 = H(x_5)$ are common for the proof and the transaction.

Similar to the case of simple transactions from Section III, also for smart contract transactions the number of items in the proof can be lower than the size of the union of the paths for the various accounts in a transactions. We call this technique *multi-proof aggregation* and it allows additional savings and follows the property that some of them can be computed based on others. The number of items that can be saved is a function of the number of accounts in the transaction.

Theorem 9. Let stx be a smart contract transaction. The proof size of the transaction with the code γ is

$$c'(stx; \gamma) = c(stx; \gamma) - 2|stx| - 1.$$

Proof: By induction. Consider an arbitrary code For a transaction of size $|stx| = 1$ all items are required and $c'(stx; \gamma) = c(stx; \gamma)$. For transactions with more than one account, each additional leaf in the tree has a proof path that intersects with the multipath of the other accounts leaves. Thus, two items can be discarded from the proof: the items that are one level below the intersection point, as each can be computed recursively by the definition of the Merkle proof. ■

Consider again the transaction stx from Example 8 of size $|stx| = 4$. We refer again to the illustration of Fig. 8. Recall that accounts of the transaction appear as leaves in dashed blue and the union of the paths in red. The proof can be shortened to include only h_0, h_3 and $h_{6\ 7}$ such that the proof size can be reduced to $c(stx; \gamma) - 2|stx| - 1 = 9 - 2 \cdot (4 - 1) = 3$ items. For instance, two items that can be discarded from the proof are $h_{0\ 1}$ and $h_{2\ 3}$. $h_{0\ 1}$ can be computed based on h_0 that is supplied by the proof and $h_1 = h(x_1)$, and similarly, $h_{2\ 3}$ can be computed based on h_3 and $h_2 = h(x_2)$.

Also for smart contract transactions, this additional savings is fixed for all codes, even when the size of the smart contract transactions can vary. Thus an optimal code can be found as a code minimizing the simpler cost expressed in Theorem 7.

B. Algorithms

We describe an algorithm for the generalized problem of communication cost minimization for transactions of arbitrary number of accounts such as smart contract transactions.

The naive algorithm that assigns codewords randomly (Algorithm 1) is applicable of course also for the generalized problem and we also consider it as the baseline. Huffman algorithm (Algorithm 2) can be also used as it only requires the account frequencies, a property that can be easily derived from the transaction hypergraph.

We can enhance the Huffman algorithm to be sensitive also to the relations between the accounts and not only to their frequencies. This is a generalization of Algorithm 4 (previously suggested in Section VI) that we present here as Algorithm 7. The algorithm works in two phases. In the first phase, the most frequent smart contract transactions are visited from the most heavily weighted to the least. For each such transaction, a Huffman tree is built for its accounts (that have not appeared as part of earlier transactions) based on their frequencies. Each tree is then added to the pool of accounts, with a frequency of the sum of its leaves, the accounts in that transaction. These accounts are removed from the remaining

Algorithm 7: Heaviest-first Huffman (Variable length codewords)

Input: Smart contract transaction distribution (STX, Q); Accounts $A = \{a_1, \ldots, a_n\}$
Output: Codewords per account $\{t_1, \ldots, t_n\}$

```
Initialize all account codewords by the empty string
Compute the account distribution P(A, P) from the
  transaction distribution Q
A' ← A, P' ← P
while STX is not empty do
    stx ← most frequent transaction in STX
    if w(stx) > then
        /* Merge accounts to subtree */
        Build Huffman subtree for the accounts in stx
        for a ∈ stx do
            Append the resulting Huffman code to the
              codeword (a)
            Remove a from all txs in STX
        end
        A' ← A' \ stx
        A' ← A' ∪ {stx}
        P'(stx) ← Σ_{a∈stx} P'(a)
    end
end
return Huffman code for A', P' /* [22] */
```

transactions and the accounts pool. The second phase is to build a Huffman joint tree for all these subtrees and accounts left in the pool. Codewords for accounts not part of a subtree are simply their codeword implied by the joint tree. Codeword for accounts part of the subtree is a concatenation of their subtree codeword and their codeword within the joint tree.

VIII. Experimental Evaluation

A. Transaction Characteristics

In this section we examine real Ethereum data to study typical transaction distributions. We observe that a large portion of the traffic is covered by a small portion of transactions and that an account typically participates in transactions with a small number of other accounts.

We collected 2-month Ethereum data (January 1 - February 28, 2022) that end at block no. 14297758. We first examine the number of active accounts in different time periods. The curves in Fig. 9 refer to periods ranging from 5 minutes to 1 week. Clearly, the number of active accounts is highly affected by the period length. For instance, for a period of 1 hour, the number of active accounts ranges between 26855-45824 with an average of 33657. In longer periods of 1 week, there are 2.16M-2.55M accounts with a higher average of roughly 2.35M. Intuitively, Merkle roots computed for a larger number of accounts require tree structures with more levels and thus necessitates longer Merkle proofs. We explain how these numbers highly affect the length of the Merkle proofs.

Studies have shown that the distribution of accounts participating in transactions is not uniform and its bias can be expressed in two major properties. Consider a particular

Fig. 9. Ethereum real data: Number of active accounts in various periods across 2 months starting at January 1, 2022.

(a) Account degree

(b) Transaction centrality

Fig. 10. Transaction distribution in Ethereum: (a) The distribution of the degree of an address in a period. (b) The ratio the most frequent (unique) transactions (pairs of addresses) take among all transactions. Plots are computed for periods of 5 minutes, 1, 6 hours, 1 day, 1 week and 1 month.

account associated with an address. We can refer to its degree as the number of accounts it conducts transactions with. Often in such networks the degree follows a power law distribution [29], [30] such that the probability for an account degree d is $P(d) = d^{-}$ for some positive . This indicates that the majority of the accounts only transfer value to very few accounts, and on the other hand a minority of accounts are connected with many others. These highly connected nodes are commonly exchanges or mining pool accounts and are taking a major role in its shape, as observed for Ethereum [31]. Similarly, in a view of a transaction as its set of accessed addresses, a small number of unique transactions cover a large portion of transactions. A recent study [30] showed similar results for additional blockchain networks. Both distributions are affected by the length of the measured interval.

We also measured the account degrees and the weight of the most popular transactions in various periods that range from

TABLE II
RELATIVE SAVINGS (%) IN COMMUNICATION COST FOR THE VARIOUS ALGORITHMS IN DIFFERENT PERIOD LENGTHS (VS. BASELINE)

length	5m	6h	1d
active accounts (mean)	4139	156646	340692
Random	6.0%	4.6%	4.2%
Huffman [14]	20.6%	28.3%	28.9%
Partition	33.3%	23.2%	23.0%
Pairs- rst Huffman	30.0%	32.2%	32.5%
Weight Balance	19.4%	26.9%	27.5%
Partition-Balance	40.6%	42.4%	42.3%

(a) 6 hour period over a month

(b) 1 hour period over a day

Fig. 11. Jensen-Shannon divergence of transaction distribution between different periods illustrating the uctuations in Ethereum transaction patterns. Low divergence values indicate minor changes, while higher values suggest signi cant distribution shifts. We can observe a decent transition towards yellow away from the diagonal, suggesting an increase in divergence for periods that are far apart in time.

5 minutes to 1 month. As can be seen in Fig. 10, not only the account degree follows a heavy-tailed distribution, also the transaction frequency as few transactions repeated many times while most of the transactions appear only once. For instance, the top 1% most frequent transactions cover 19% and 42% of the traf c for period of 5 minutes and 1 week, respectively.

As our model is based on the transaction distribution in a given period, it is important to analyze how these distributions might change over time. A substantial change in the distribution upon which a tree was built would likely increase the average size of Merkle proofs. To assess the stability of these distributions, we apply the Jensen-Shannon divergence [32] as a method to quantify the similarity between different periods:

$$JSD(P;Q) = H(M) - \frac{1}{2}\left(H(P) + H(Q)\right)$$

where $P;Q$ are distributions and H is the entropy. This metric measures distance between distributions, where 0 indicates that the two distributions are equal, and a value close to 1 would mean that the distributions are very different.

We compared the transaction-frequency distribution of the highest-occurring account pairs across different time points across a month to see if signi cant shifts occur that might impact the Merkle proof sizes. For this, we computed the transaction distribution for different periods, applied the Jensen-Shannon divergence between all pairs and graphed these values in a matrix in Fig. 11. In Fig. 11a, there are 15 periods of 6 hours at evenly-spaced time points across January 2022, and a similar analysis for 24 periods of 1 hour throughout a single day is illustrated in Fig. 11b. The color-coded matrix shows the divergence values, with warmer colors representing higher divergence. We observed that periods closer to each other in time tend to have lower divergence values, indicating more similar transaction patterns. In contrast, periods that are temporally distant exhibit a higher divergence, highlighting a shift in the transaction patterns over time. We note, however, that this is only a trend, and in that comparison, the most different distributions are the last two periods at the end of the month. Our analysis shows that, while there are slight variations from time to time, the overall transaction distribution remains stable. In addition to distribution shifts, it is equally important to evaluate how such changes affect average proof costs. We cover this analysis in the following section.

B. Evaluation of the Algorithms for Computing Merkle Trees

We experimentally evaluated the quality of the code outputs of our algorithms (Section VI) on various period lengths of Ethereum transactions. We used Ethereum transaction data of approximately 1 month from block no. 13916166 (January 1, 2022) until block no. 14116760 (of January 31, 2022).

Each experiment has the following parameters:

Algorithm - The algorithm used to compute the code: Algorithms 1-6 and a baseline that has separate proofs for the two accounts having a xed codeword length of $\log_2(n)$ bits for n active accounts. Among these, we include a comparison with the Huffman (Algorithm 2) as of [14].

Time period length - The total period in which transactions are considered as input to the algorithm. We refer to lengths of 5 minutes, 6 hours and 1 day.

Start time - The time from which transactions are considered as input to the algorithm. There are a total of 120 start times, evenly spaced between the above blocks. For example, if the start time is 22/1/2022 15:00 and the length is 12 hours, the input transactions are all those that occurred between 22/1/2022 15:00 to 23/1/2022 03:00.

Each algorithm is executed on all the above transaction periods with different start and time period length combinations.

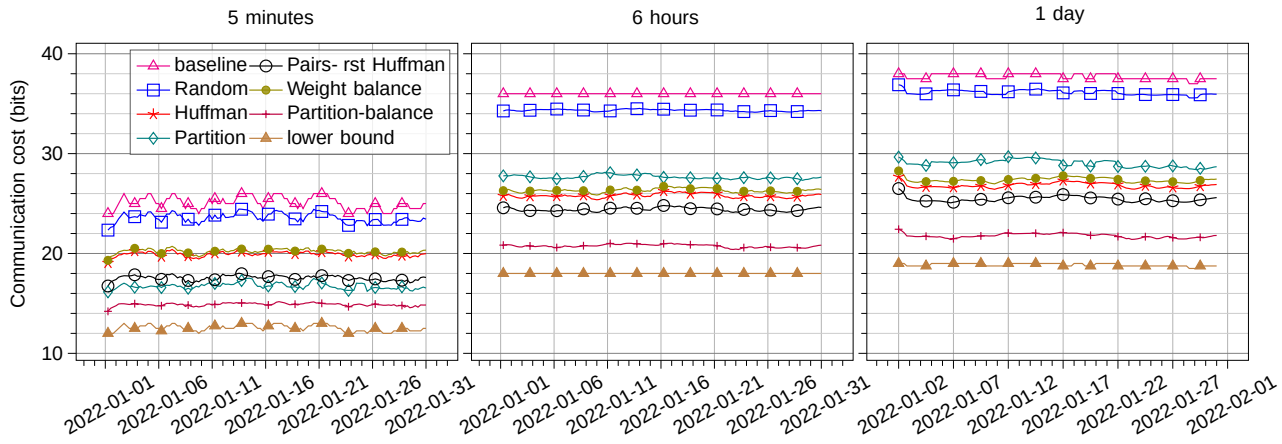


Fig. 12. The communication cost for the various algorithms for the Ethereum data in time periods of different lengths. The baseline refers to separate proofs for the two accounts having a fixed length of $\lceil \log_2(n) \rceil$ bits for n active accounts. Random, Huffman [14], Partition, Pairs-first Huffman, Weight Balance and Partition-balance refer to Algorithms 1-6.

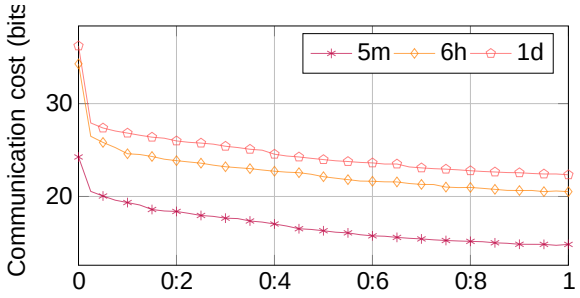


Fig. 13. The cost of unseen transactions. 0 1 is a parameter that indicates the fraction of the transaction period used to optimize the coding tree by Algorithm 6. The cost is an average of periods of 5 minutes, 6 hours and one day, respectively, in January 2022.

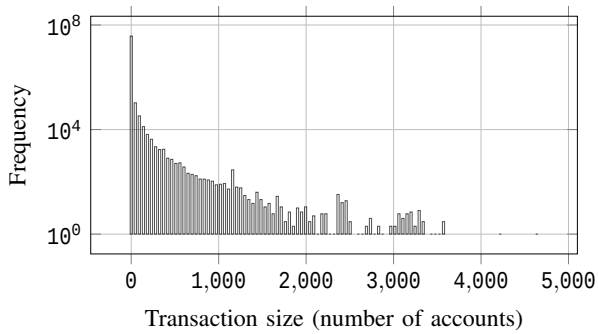


Fig. 14. Ethereum real data: Smart contract transaction size histogram for all transactions during January 2022.

The achieved communication costs, measured in bits, can be found in Fig. 12. The cost is influenced by the number of active accounts in the time lengths, with averages of 4.14K, 157K, and 341K. We observe relatively steady communication costs over the various start times. The average savings over the entire month vs. the baseline appear in Table II.

Partition-and-weight-balance gives the lowest communication costs, with an average improvement of 40.6% - 42.4% relative to the baseline for the tested time periods. Note the difference between the algorithm to each of its components: partition, and weight balance. The relative improvement for both Huffman and the Pairs-fist Huffman (relative to the baseline) slightly increases for longer periods. Partition performs also well, especially for the short time length of 5 min, improving 33.3% on average. For longer time lengths, its savings are slightly lower than both Huffman and Pairs-first Huffman but are still meaningful in the range of 27.4%-28.4%.

Following on the distribution-shift analysis in Section VIII-A, we want to better understand how the average proof size is affected by future transactions, as the distribution that the tree was optimized for evolves. To investigate this, we conducted an experiment where we constructed the coding tree using only the first fraction of the testing period, denoted as α, and then observed the communication cost as α varies. More formally, given a transaction distribution (TX, Q) and a parameter 0 α 1, we define TX to be the first $\alpha jTXj$ transactions in TX, and similarly Q is defined to be the probability distribution corresponding to TX . We first find σ by Algorithm 6 with $(TX\ , Q\ )$, and then calculate the cost of the code $C\,(Q, \alpha)$.

We began with the baseline α = 0, where no prior transactions are used to optimize the tree, and we incrementally increase α to include a greater fraction of the transaction period, thus refining the coding tree with more transaction data. As expected, the results show that as α grew, the communication cost decreased, and the coding tree became better tailored to the transactions within that period. The graph in Fig. 13 presents, unsurprisingly, a clear inverse relationship between the communication cost and the value of α, highlighting the importance of using a larger sample of transactions to construct a more efficient coding tree. However, the steepest part of the graph is when α became non-zero, and the improvements became marginal as α grows, suggesting a threshold in the benefit of using additional transaction data for optimization. This plateau indicates that an optimal balance can be found between computational resources and the effectiveness of the coding tree. It also implies that regularly

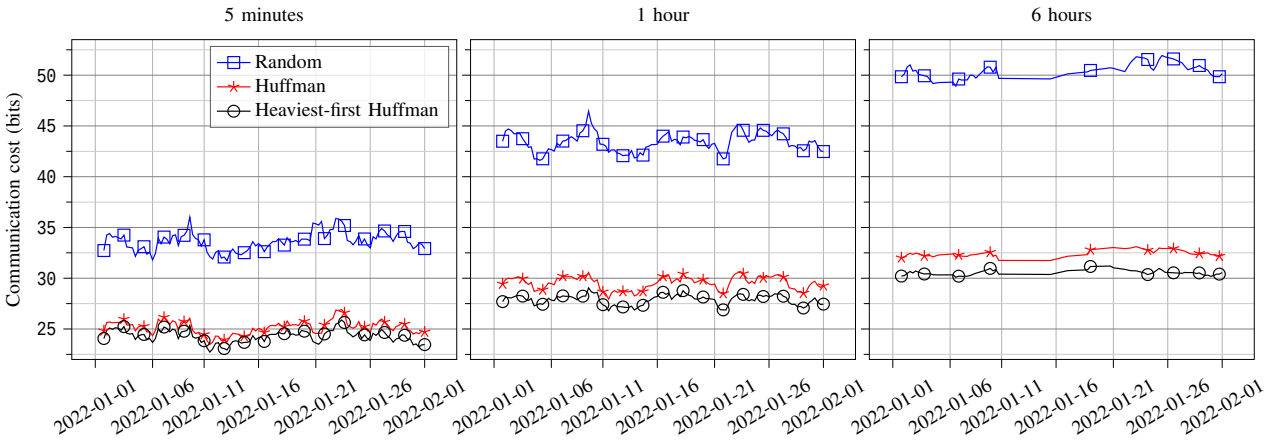


Fig. 15. Smart contract transactions: The communication cost for the applicable algorithms for the Ethereum data in time periods of different lengths. Random, Huffman and Heaviest-first Huffman refer to Algorithm 1, Algorithm 2, and Algorithm 7.

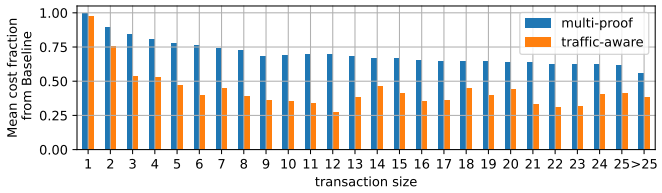


Fig. 16. The fraction of multi-proof aggregation with and without traffic-aware tree construction from the baseline of traditional Merkle tree, where the cost of multiple proofs is the sum of the individual proof sizes.

updating the tree with recent transaction data can ensure that encoding remains efficient without constant recomputation.

So far, we evaluated algorithms for regular transactions of two accounts. We now also evaluate the performance of the algorithm for smart contract transactions. To derive real data of such accounts accessed together, we use Ethereum smart contract transaction data of approximately 1 month from block no. 13916166 (January 1, 2022) until block no. 14116760 (January 31, 2022), over 38 million transactions in total. For each transaction, we recorded the set of accounts that were accessed during its execution. The average number of accounts in a transaction was 3.98 with a range of 1-4638. Frequencies of the transaction sizes for this time period appear in Fig. 14.

Fig. 15 shows the obtained communication cost for transactions collected in time periods of 5 minutes, 1 hour and 6 hours. As before, the cost is affected by the number of active accounts in the time lengths, with averages of roughly 5.6K, 41.3K, and 180K accounts for the three periods, respectively. An average improvement of 25.2%-35.6% in the communication cost is demonstrated with the Huffman algorithm in comparison with that of a random assignment. An additional 3.4%-5.9% is gained by using Algorithm 7 that also considers relations between accounts in the most frequent transactions.

Multi-proof aggregation, as explained in Section VII-A, utilizes the fact that some nodes in the tree are redundant when several proofs are combined together. It is used in several blockchain systems for their savings in communication costs [10], [11], [12]. We conducted an experiment to quantify the extra gains in proof size of traffic-aware Merkle trees over multi-proof aggregation. This experiment aimed to compare their gains against the baseline scenario, where the tree is not traffic-aware and all proofs are summed without node cancellations, namely, $c_{base}(stx, \sigma) = \sum_{j=1}^{|stx|} \sigma \; a_j$. We used 1 hour periods of smart contract transactions from the blocks above to serve as the transaction distribution input, and average the results over all periods. For each distribution, we constructed a traditional Merkle tree and a traffic-aware Merkle tree (using Algorithm 7), and tested the communication cost of (1) the baseline, a traditional tree when all proofs are summed, (2) the traditional tree with multi-proof aggregation and (3) the traffic-aware tree with multi-proof aggregation. The results are visualized in Fig. 16, with the x-axis represents transactions by size and the y-axis represents the fraction of the communication cost relative to the baseline. The data shows that the benefit of our traffic-aware with simple transactions of 2 accounts is subtle relative to smart contract transactions with more than 2 accounts, and for transactions of size 3 there is almost 50% reduction in communication costs relative to baseline. Multi-proof aggregation has always inferior communication cost without the traffic-aware tree.

IX. Conclusion

We studied the design of traffic-aware codes for data organized as Merkle trees in blockchain networks for reducing the communication cost in proofs for data membership. We presented the fundamental properties of the problem and analyzed bounds of the optimal communication cost. We developed various algorithms and demonstrated their efficiency based on real data from the Ethereum network. We presented a generalized problem for smart contract transactions, with a corresponding encoding algorithm and evaluated its performance. An open question sounds natural: Is there an optimal polynomial solution to the problem or the problem is hard to solve? Interestingly, we explained that codes based on Huffman (that can be found in linear time), as well as codes derived through balanced partitioning (which are difficult to be computed), can both be found efficient.