Avoiding the 1 TB Storage Wall: Leveraging Ethereum's DHT to Reduce Peer Storage Needs

Jean-Philippe Eisenbarth
SnT, University of Luxembourg
Luxembourg, Luxembourg
philippe.eisenbarth@uni.lu

Thibault Cholez
University of Lorraine, CNRS, Inria, LORIA
Nancy, France
thibault.cholez@loria.fr

Olivier Perrin
University of Lorraine, CNRS, Inria, LORIA
Nancy, France
olivier.perrin@loria.fr

ABSTRACT

Blockchains face many challenges in time, among which the ever-growing storage needs for blockchains' data. In particular, Ethereum is quickly approaching the 1 TB storage limit for a node, what may significantly reduce the candidates able to run an Ethereum fullnode. In this paper, we propose a new synchronization and storage strategy for Ethereum that takes full advantage of the Distributed Hash Table implemented in all clients but left unused. By digging into the history of Ethereum's synchronization and storage strategies, we justify why such distribution of storage duties among peers makes much sense today. We implemented our solution in the official Ethereum client Geth and validated its smooth operation on a private Ethereum instance. Our solution can save around 60% of the storage of a node (360 GB) which represents a total of 12 PB of data at the network scale, while being fully backward compatible with current clients.

CCS CONCEPTS

• **Networks → Peer-to-peer networks**; • **Computer systems organization → Peer-to-peer architectures**.

KEYWORDS

blockchain, Ethereum, storage, scalability, Distributed Hash Table, synchronization, Geth

ACM Reference Format:
Jean-Philippe Eisenbarth, Thibault Cholez, and Olivier Perrin. 2023. Avoiding the 1 TB Storage Wall: Leveraging Ethereum's DHT to Reduce Peer Storage Needs. In *The 5th ACM International Symposium on Blockchain and Secure Critical Infrastructure (BSCI '23), July 10, 2023, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3594556.3594625

1 INTRODUCTION

It is widely acknowledged that blockchains are distributed systems with high resource consumption. In September 2022, Ethereum vastly reduced its computation cost by successfully moving from a proof-of-work to a new proof-of-stake consensus mechanism. However, computational resources are not the only ones consumed by an Ethereum node. Network and storage resources are also needed to exchange and store the blockchain data (blocks, transactions, etc.). The present paper is focused on the storage dimension in Ethereum.

Indeed, as an append-only data structure that is replicated on every peer, the amount of data to be stored to allow a node to run the blockchain is always increasing with its usage. In November 2022, an Ethereum node that joins the blockchain needs to store approximately 600 GB of data (representing both cold data – old blocks and receipts –, hot data and additional data structures). Thus, a new node needs between one and two days to synchronize with the head of the blockchain on a gigabit Internet connection. Our concern is that the 1 TB cap of requested storage capacity is quickly approaching and comes with the risk that many nodes that compose today the Ethereum system and ensure its resilience may leave rather than invest in new storage capacities. Beyond the short run 1 TB-wall issue, the storage of blockchain data remains a critical problem in the long run that can endanger their sustainability in time if not mitigated.

In this paper, we propose an elegant way to reduce the storage needs of an Ethereum node by leveraging Ethereum's Distributed Hash Table to distribute the storage of old blocks and receipts (also called cold data). Indeed, Ethereum's peer-to-peer (P2P) network is based on a well-known architecture named Kademlia [4] that can efficiently retrieve data at the P2P network level with a logarithmic complexity. However, while fully functional, Ethereum's DHT (Distributed Hash Table) is under-exploited so far and only used to list contacts (known nodes) but not to store any blockchain data. To understand why, we explain how blockchain synchronization is currently performed in the official client Go Ethereum (Geth). In particular, we remind the history of the synchronization strategies adopted in Ethereum and explain why the distribution of the storage makes much sense today, blocks being no more executed since genesis block by a new node. We then describe our solution, which is fully backward compatible with current clients and do not affect any core mechanism of the Ethereum blockchain. We provide an implementation of our new synchronization mechanism exploiting Ethereum's DHT in Geth. Our evaluation on a private Ethereum instance prove the validity of the solution. When applied to an Ethereum peer, our solution saves ~60% of the storage of a node (360 GB) which represents a total of 12 PB of data at the network scale.

Our contributions are thus manifold.

(1) We present a comprehensive analysis of Ethereum's past and current storage and synchronization strategies based on our investigation of the source code;

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 830927.

(2) We propose and evaluate a new synchronization strategy that fully takes advantage of Ethereum's DHT wasted potential;
(3) We provide the source code[1] of our implementation in Geth for an easier integration of our work by the community.

The rest of the paper is organized as follows. Section 2 surveys the previous work conducted on the problematic of data storage in blockchain systems. Section 3 presents our understanding of Ethereum's data storage and synchronization strategies. Then, Section 4 describes the new architecture we propose to synchronize a node and limit its storage needs by leveraging Ethereum's DHT. Section 5 presents an evaluation of the proposed strategy on a private Ethereum instance composed of modified Geth clients. Section 6 discusses the potential impact of our solution regarding the security of the Ethereum's blockchain. Finally, Section 7 concludes the paper and presents our future work.

2 RELATED WORK

Surprisingly, to the best of our knowledge, no study so far seems to address the problematic of the increasing storage needs of Ethereum nodes. In the literature, the problematic of storage in Ethereum is only seen from the user perspective rather than the system itself. In particular, papers are interested in how to predict [17] and optimize [7] the cost of data that are stored in Smart Contracts. On this specific orthogonal problematic, IPFS (InterPlanetary File System, which is also based on Kademlia) is often leveraged by applications to offload data from Smart Contracts on this external distributed storage service [11, 12], even if its reliability compared to the in-blockchain storage and hidden costs can be questioned.

On the more general problematic of blockchain's data storage at the system level, [18] propose to offload the storage of Bitcoin's transactions on IPFS. While this approach may seem close to ours, we think that it is a bad idea, in particular regarding Ethereum. In fact, while we know that the Ethereum P2P network is large enough, well distributed and stable [5] to be considered reliable, the IPFS instance that must be created to store blockchain's data must prove such properties. Moreover, IPFS being a general purpose distributed file system, many features proposed by IPFS are not needed to store blockchain's data. Ethereum already providing a fully functional Kademlia DHT does not have interest to rely on another one which integration will be more difficult and put backward compatibility at risk.

The authors of [1] evaluate scalability issues of Ethereum and precisely identify the data storage in the long run as a critical problem. They did not recommend any solution but think that the use of Sharding, as considered by Ethereum developers, is a possible solution to overcome this challenge. In a few words, Sharding is inherited from the world of distributed databases: the blockchain is split into shards, and subsets of validators are randomly assigned to each shard to validate transactions. So, Sharding goes beyond the simple distribution of storage and also encompass the parallel execution of transactions. According to [13] this introduces a new challenge to guarantee that critical data is published and stays available for as long as it is relevant to prevent unique fault attribution. The authors suggest two partial solutions to this issue and state that more research is still needed before a reliable large scale deployment of such architecture. The authors of [3] analyzed the performance of several prototypes of Ethereum 2.0 clients deployed on the Medalla Test Network that implement the Sharding feature. They witness different behaviors between clients and many stability issues. So there is a real need to improve the current Ethereum protocol while the next version is in heavy development. The solution we propose may also be complementary to Sharding and integrated in Ethereum 2.0.

[1]https://github.com/jpeisenbarth/go-ethereum/tree/DHT-storage

3 BACKGROUND ON SYNCHRONIZATION AND STORAGE IN GETH

In this section, we provide a comprehensive description of Ethereum's synchronization and storage strategies. Unfortunately, there is no proper documentation on these aspects, so we had to investigate the Geth client source code and some informal technical writings of the developers to understand these aspects.

3.1 Data structures in Geth

Abstractly, Ethereum is not only a simple distributed ledger but also a distributed state machine based on the execution (or application) of transactions. This state machine uses different types of data for its operation, as illustrated in Figure 2.

The current state of Ethereum is called *world state* and the update of this state is done by applying the set of transactions contained in the last accepted block that was created. The world state is composed of all the accounts created, knowing that an account is the association of a user's address or a smart contract with an account state. In the case of a user account, there is no associated code, so the hash of the code is the hash of the empty character string. For a smart contract account, it is the hash of the bytecode to be executed by the Ethereum Virtual Machine, i.e. the execution environment present on every node in the P2P network. When a new block is created, which transactions may involve crypto-currency exchanges as well as calls to smart contract functions, all the nodes in the network must generate the new world state and, when a block-level consensus is reached, all the nodes then share the same universal state. This is why Ethereum is sometimes called a distributed world computer.

In addition to the world state, we must also store the transactions received from the blocks of the blockchain. Major elements of a transaction include the price in gas to execute it, the address of the recipient (which can be a smart contract) and the value of the transaction. Then, an intermediate object is generated and manipulated by users, called a receipt. It is an object that represents the result of a transaction encapsulated in a block, including a digital fingerprint of the transaction, its block number, the quantity of gas used and, if a smart contract is deployed, its address.

Basically, the data structure used in Geth is a tree combining two other types of trees: the *Merkle tree* and the *PATRICIA tree*. This tree provides a key-value data structure based on common prefixes (thanks to the PATRICIA tree) and allows easy integrity checking by comparing the hash of the root (thanks to the Merkle tree). The Merkle tree is a classical data structure used in distributed systems. It is built as follows: the leaves of the tree are the hashes of each of the initial data, then these hashes are concatenated two by two (binary tree) to form the parent node and so on until the root of the tree. This root is then securely exchanged between the parts of the distributed system to effectively verify the integrity of the data set: a single modification of any data results in a different hash of the root. The PATRICIA tree is a type of prefix tree used to represent an associative array. It is a compact structure in which each node having only one child is merged with the latter. It allows search, insertion and deletion operations to be performed in $O(\log_2 N)$, N being the number of stored elements[2]. Figure 1 illustrates a simplified example of a Merkle-Patricia tree as implemented by Ethereum (the size of the key is reduced for the sake of readability, normal size is 32 bytes).

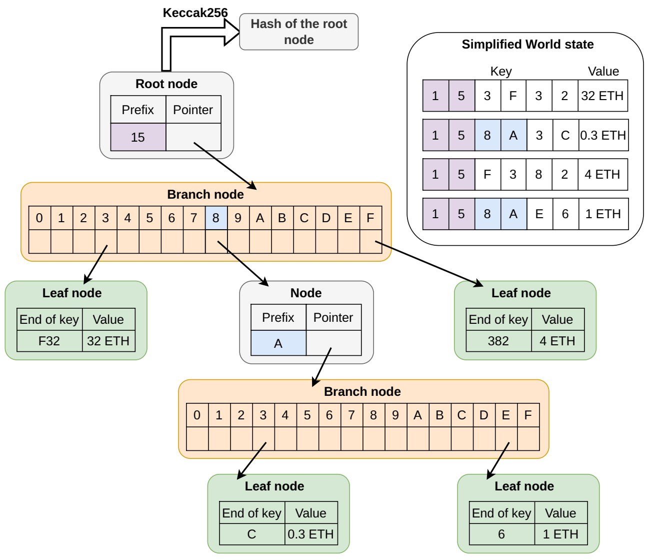

Figure 1: Simplified example of a Merkle-Patricia tree, inspired by the work of Lee Thomas [15].

In Figure 2, we show the links between these structures and the content of a block. We observe that a block is composed of three elements, namely its header, the list of transactions and the list of known uncles blocks[3]. The block header is composed of many elements, and we won't detail all of them here. MixHash and Nonce are related to the proof of work (to verify that the crypto-puzzle is well solved). The allowed gas limit and the total gas used in this block are also stored in the header. The hashes of the Merkle-PATRICIA tree roots of the accounts, transactions and receipts contained in the block are added in the header so that a peer of the network receiving this block can verify that when he applies the transactions on his own world state, he gets the same state.

This Merkle-PATRICIA tree, despite its optimized structure, shows its limits in terms of access performance. Indeed, the Ethereum blockchain is constantly growing, and despite the storage of databases on SSD, the numerous accesses to data structures required to apply the contents of a block are beginning to be problematic for the performance of the clients, especially during the synchronization phase that could take more than a week. This is why developers have introduced a new flat data structure allowing a $O(1)$ access complexity, called *Snapshot*. This is the complete view of the Ethereum state at a given block of the current epoch.

In the next subsection, we will see the different synchronization's mode of a Geth node and in particular its new default mode, snap, which takes advantage of this Snapshot data structure.

3.2 Synchronization modes of Geth

The Geth client offers two main synchronization modes: Snap (the default mode since 2021) and Full.

Snap. The synchronization of a new node in *snap* mode proceeds as follows :

- download and verification of block headers;
- download of all block bodies/receipts and in parallel download of raw data from the world state (leaves of the Merkle-PATRICIA tree) and build of the tree;
- heal the tree to prepare for newly arriving data (*healing phase*).

This mode also saves state checkpoints, but not all states. Snap mode takes advantage of the Snapshot structure to speed up the synchronization of a new node, because it retrieves states directly (without re-executing the entire history of transactions) from the other peers who quickly access these states via the Snapshot data structure. Before the introduction of the Snapshot structure, retrieving states from the other peers took much longer, because of the slower access time of the Merkle-PATRICIA tree. Once the synchronization is finished (i.e. it has caught up with the head of the blockchain) the node switches to the *Full* mode described hereafter.

To better understand the history of synchronization modes in Geth, the Snap synchronization mode now replaces the former *Fast* mode (default mode between 2017 v1.6.0 and 2021 v1.10.0). The idea was the same, i.e. to download directly the world state from other nodes without executing all the blocks since genesis, what took too much time at some point. The difference lies in the absence of the Snapshot data structure at the time that has for consequence a slower retrieval of state data to be sent to the joining peer and resulting in several days of synchronization time.

Full. The *Full* mode is the classic mode of synchronization for a blockchain and was the only mode available until 2017 (v1.6.0). It offers all the expected guarantees and consists in requesting all the blocks, checking their origin and the proof of work (or other proof if appropriate) and re-executing all the transactions they contain to update the world state accordingly. All the intermediate states since the genesis block are not kept, but the node can retain some state save points. Periodically the node cleans up the data pertaining to the previous states and, if needed, re-executes the transactions between the nearest state checkpoint and the target state. Historically, the Full synchronization was the only mode available, and it allowed the most rigorous verification of the blockchain data at the expense of a very long synchronization time (more than one week before Fast or Snap modes) that became unbearable with time and the continuous growth of the blockchain. This mode can however still be set manually for a joining node.

[2]As announced by the Geth developers, but in practice this should be put into perspective that the number of elements in the tree can be much larger than the key size, k, and the complexity of the string comparison $O(k)$
[3]As a reminder, these are the other blocks at the same level that share the same parent

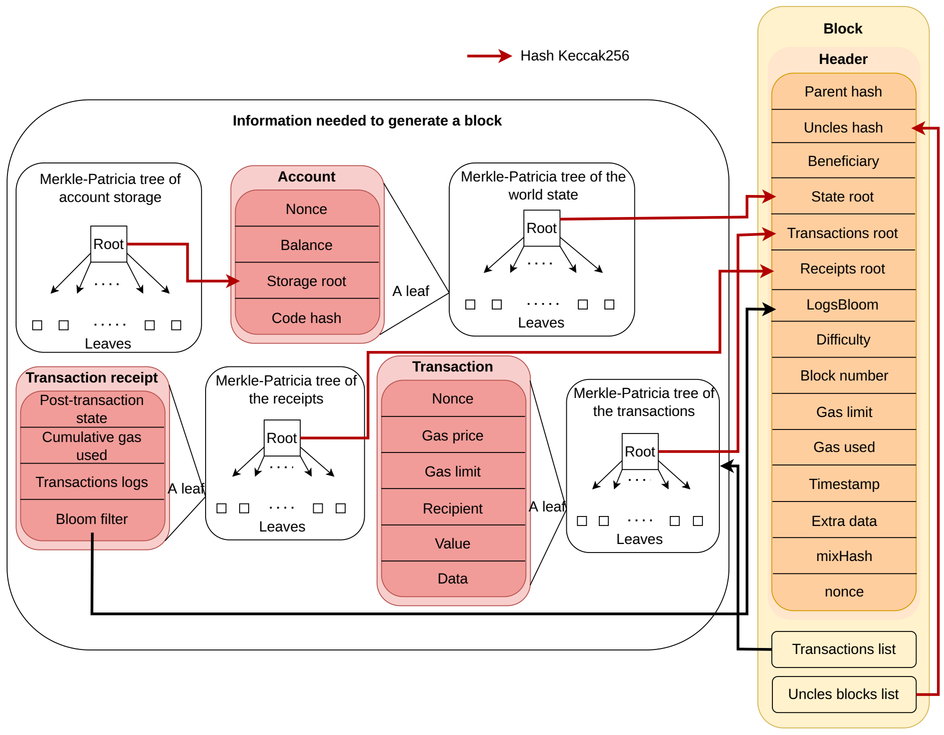

Figure 2: High-level picture of Ethereum data structures

3.3 Data storage in Geth

As said before, the Merkle-PATRICIA tree is the data structure used to store the world state, the transactions in a block, the receipts of the transactions and the storage data of an account. In Geth, this structure is implemented using two key-value databases: *LevelDB* and *FreezerDB*. LevelDB is used for storing items (blocks, headers, receipts, etc.) that are regularly read or changed. In practice, it is the storage of 3 epochs (an epoch is composed of 30,000 blocks, so the last 90,000 blocks for LevelDB). Beyond this threshold, the elements are transferred to the FreezerDB database for their long-term storage, in which access is very rare and modifications are impossible (append-only because the blockchain is considered ossified at this level). In Geth, LevelDB is also called *Key-Value Store* while FreezerDB is also called *Ancient Store*. Given the respective properties of these two databases, LevelDB data is described as hot data (storage intended on SSD) and FreezerDB data is described as cold data (storage intended on HDD).

In order to measure the size of a node's storage, we ran a new Geth node synchronization by default (Snap mode) in July 2022. This sync lasted about 26hr and the storage size reached almost 600 GB. A summary provided by Geth's `DB Inspect` tool is available in Table 1. We can observe that the storage of blocks bodies and receipt lists in FreezerDB accounts for about 60% (358.9 GB) of the total storage (597.14 GB). As we saw earlier, this type of data is not intended to be accessed regularly but only read when a new node asks for data, and written to store new data but older than 3 Epochs. This data can ultimately be used to replay all transactions and to generate a new state when a problem occurs. However, this storage is far less useful in the daily operation of the blockchain since the addition of Fast and Snap synchronization modes.

In addition, we also noticed that the size of our node's storage (shown in Table 1), once synchronized, increased faster than expected. The new blockchain data did not fully explain this growth. According to Etherscan[4], the trend of undue increase is due to data leakage in data structures which represent 50 GB per month. Only an offline cleaning (`geth snapshot prune-state`) can fix this.

To conclude this part, since the vast majority of nodes do not validate transactions since the genesis block anymore, there is no need for **all nodes** to store **all blocks** anymore. FreezerDB's characteristics make it an ideal candidate for distributing its data in the DHT to save on peer storage. This is described in the next section.

4 NEW STORAGE AND SYNCHRONIZATION STRATEGY

4.1 Motivation

Based on our observations in Section 3, we propose to efficiently distribute FreezerDB's storage, in particular the blocks bodies and receipt lists[5], on all the peers of the network by using the fully functional yet under exploited Ethereum's Kademlia DHT. This will drastically decrease the storage capacity needed to run an Ethereum node. Our solution does not concern the data stored in LevelDB, as

[4]https://etherscan.io/chartsync/chaindefault
[5]In the rest of this paper, we will omit the receipts lists storage and only mention block bodies for convenience, but the very same storage strategy can be applied to receipt lists

Table 1: Stored data of a newly synchronized Geth node, July 14th 2022 (sync time is 26hr).

Database	Category	Size (GB)	Share (%)
LevelDB	Blocks headers	0.05	0.01
LevelDB	Blocks bodies	7.41	1.24
LevelDB	Receipts lists	5.95	1.00
LevelDB	Smart Contracts codes	3.88	0.65
LevelDB	Trees' nodes	130.93	21.93
LevelDB	account Snapshot	8.66	1.45
LevelDB	Storage Snapshot	55.25	9.25
FreezerDB	Blocks headers	7.22	1.21
FreezerDB	Blocks bodies	242.17	40.55
FreezerDB	Receipts lists	116.73	19.55
Total		578.25	96.84
FreezerDB's block bodies and receipts lists		358.90	60.10
LevelDB Total		230.21	38.55
FreezerDB Total		366.93	61.45
Total (LevelDB + FreezerDB)		597.14	100.00

these are necessary for the generation of a new state and can be often modified, thus must remain available locally on every peer. As discussed earlier, a node that wishes to update its *world state* needs its previous *world state* and the last known consensus block that has not yet been integrated. Therefore, we consider that each peer in the network should keep all blocks that have not yet been transferred to FreezerDB.

Why the implementation of Kademlia was unused so far in Ethereum remains an open question. Kademlia being one of the most optimized P2P architecture, it may feel natural to implement it when aiming to build a P2P network. However, Ethereum does not even benefit from Kademlia's efficient routing properties as Ethereum's live data (blocks, transactions) are sent by the mean of another layer of unstructured P2P network using a gossip-like dissemination strategy. Our guess is that Kademlia [10] was initially thought to be used to store blockchain's data before realizing that every peer must store all the data in the early phase of a blockchain and Kademlia became useless under the initial **Full** synchronization mode that used to build everything from the genesis block. Another reason might be that DHTs are inherently vulnerable to localized Sybil attacks [4] that can potentially take the control over the stored information [8, 14], which may have frightened the developers. But several prevention mechanisms exist and have been proven efficient to avoid such attacks [2, 5].

4.2 New synchronization mode based on the DHT

The new synchronization mode we propose has two sides. The first one consists for a new node to only ask for ancient blocks that are in its *tolerance zone* which defines its responsibility regarding the new distributed storage of blockchain data. To compute the distance defining the *tolerance zone* of a node, we naturally use the XOR distance function, inherited from Kademlia, between the network ID of the peers (which as a reminder is the Keccak256 hash of the peer's ECDSA public key) and the Keccak256 hash of the block headers. Thus, we can define a distance threshold below which a peer becomes responsible for the storage of regarding blocks, guaranteeing a given replication factor, i.e. a number of peers responsible for the same block in order to ensure that all blocks are effectively stored by several peers, making the system more robust to churn and attacks. Once a peer is synchronized, it will use the very same metric throughout its life to know when data older than 3 epochs in LevelDB must be moved locally in FreezerDB or simply deleted to save storage space.

The other side of the new synchronization mode consists in requesting blockchain data to relevant peers only from the DHT perspective (i.e. close neighbors), as illustrated in Figure 3. This may seem natural, but the transition phase will inevitably mix both synchronization modes for a time and a node may choose to either get all the blocks it needs from a couple of classical full nodes or to exploit the DHT storage. The new synchronization mode is fully backward compatible with current clients. A full node can iterate over the DHT to retrieve all the blockchain data if necessary, as illustrated in Figure 4, while a node synchronizing through the DHT can ask a full node for the data it must store, as illustrated in Figure 3.

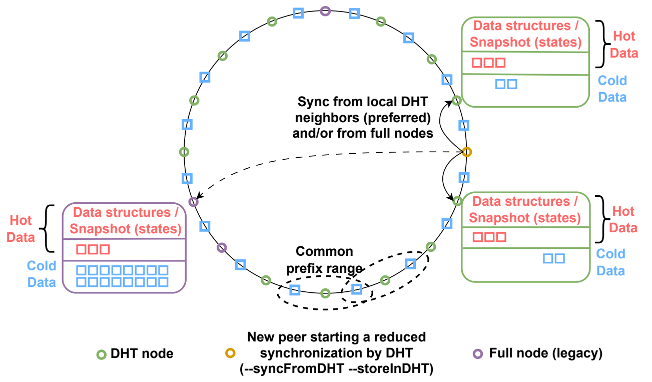
Data structures / Snapshot (states)
Hot Data
Cold Data
Sync from local DHT neighbors (preferred) and/or from full nodes
Data structures / Snapshot (states)
Hot Data
Cold Data
Hot Data
Data structures / Snapshot (states)
Cold Data
Common prefix range
DHT node
New peer starting a reduced synchronization by DHT (--syncFromDHT --storeInDHT)
Full node (legacy)

Figure 3: Illustration of a node doing a reduced synchronization over the DHT space (our proposal to spare storage space)

Another side-benefit of our solution is that it will better distribute the load established by a joining peer over the network. Figure 5 shows the number of blocks in percentage sent by the peers contacted during a default (Snap) synchronization. We can see that most of the packets are sent by only a few peers. The most prolific peer was responsible for approximately 12% of the blocks sent to the synchronizing peers. The top 10% of the most prolific peers (35

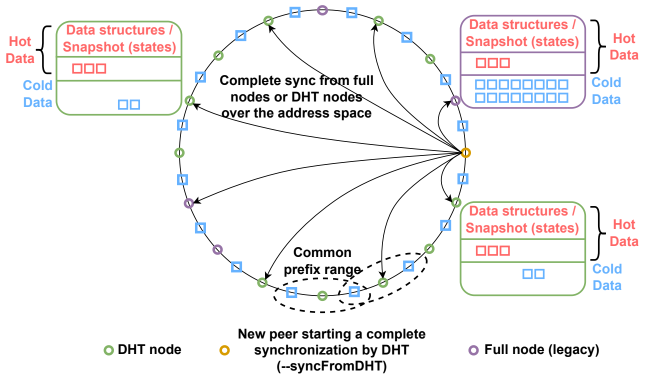
Hot Data
Cold Data
Data structures / Snapshot (states)
Complete sync from full nodes or DHT nodes over the address space
Common prefix range
New peer starting a complete synchronization by DHT (--syncFromDHT)
DHT node
Full node (legacy)

Figure 4: Illustration of a node doing a complete synchronization over the DHT space

peers) are responsible for 85.21% of the blocks sent. This tendency was confirmed by several experiments of default synchronization.

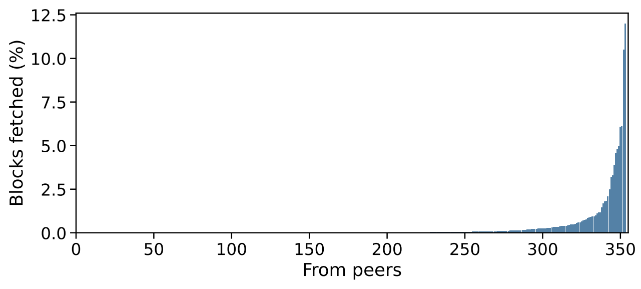

Figure 5: Percentage of total blocks sent by the contacted peers during a synchronization

4.3 Expected storage gain

The Ethereum P2P network can save a lot of storage space by adopting this new synchronization strategy. Given a storage size of block bodies and receipt lists in FreezerDB of 358.9 GB (see Table 1), Table 2 shows the expected storage gain for a single peer and for the entire P2P network. For the calculation of the gain on FreezerDB, we also took into account the storage of block headers (in FreezerDB) which storage is fixed on each peer. The first column in Table 2 defines the common prefix between a node and a data. The longer the prefix, the shorter the tolerance zone and the lower the replication factor. The red row represents the current state of the storage in the Ethereum's P2P network, while the blue row represents the state of the storage if our proposal is adopted. For a replication factor of 1125, i.e. 1125 nodes responsible for storing the same block, the expected long-term storage gain on a *fullnode* is about 95% (and about 58% gain on the total peer storage). Being given an Ethereum network composed of about 36000 nodes (according to the data from [5]), this represents a total gain of 12 PB of data at the network scale.

Table 2: Theoretical gain on a fullnode's storage.

Prefix (bits)	Replication factor	Storage of block bodies and receipts lists (single peer)	Gain on FreezerDB	Total gain (single peer)
0	36 000	358.90 GB	0.00%	0.00%
1	18 000	179.45 GB	48.91%	30.05%
2	9000	~89.73 GB	73.36%	45.08%
3	4500	~44.87 GB	85.58%	52.59%
4	2250	~22.44 GB	91.70%	56.35%
5	1125	~11.22 GB	94.75%	58.22%
6	563	~5.61 GB	96.28%	59.16%
7	282	~2.81 GB	97.05%	59.63%
8	141	~1.41 GB	97.43%	59.87%
9	71	~0.71 GB	97.62%	59.98%
10	36	~0.36 GB	97.71%	60.04%

5 EVALUATION

5.1 Implementation in Geth (proof of concept)

At the node discovery protocol level, Geth implements two versions, called "Discv4" and "Discv5". In 2022, the "Discv4" protocol is the only one used, "Discv5" being a planned evolution still waiting for a mass deployment. We will not detail exhaustively the differences between these two versions, but we had to use "Discv5" for our proof of concept, because in "Discv4" the peers are erroneously identified by their public key whereas in "Discv5", they are identified by their network ID (Keccak256 hash of the public key), as it should be.

As discussed in section 3, Geth uses two databases, namely LevelDB for all operations and FreezerDB for long-term storage. We have modified the storage requirements of the latter. For this proof of concept, we focused only on the storage of block bodies. The principle remains the same and is also applicable to receipts lists.

We have thus implemented two new options, summarized in Table 3, in the Geth client for which we provide the modified source code[6]. The first option retrieves the desired blocks only from peers that are responsible for them (see Figure 4), the second retrieves only the blocks a peer is responsible for (using the distance function). These two options can be combined to allow the client to work as we intended in our new solution, i.e. retrieve the blocks from peers responsible for the blocks our client needs to store due to its network ID (see Figure 3).

Specifically, there are two main functions we modified to implement our proposal. First, in file `core/rawdb/ chain_freezer.go`, the function `freezeRange` is modified to make the client store only the blocks it is responsible for. Second, in file `eth/downloader/queue.go`, the function `Schedule` is modified to

[6]https://github.com/jpeisenbarth/go-ethereum/tree/DHT-storage/README.md

Table 3: Summary of Geth's new options

Option	Description
`--syncFromDHT`	allows the client to contact only the peers responsible for a desired block
`--storeInDHT`	allows the client to request only the blocks for which it is responsible for storage

make the client send a block download request only to another node responsible for its storage (thanks to the XOR distance operation).

To verify that our proof-of-concept works as intended and to evaluate the execution time and storage gain, we conducted the following experiment. We deployed 16 Geth nodes (unmodified) in a private P2P network. Their NodeIDs were forced[7] to achieve a uniform distribution of peers over the DHT address space despite their small number. The common prefix is set to 4 bits, which results in a replication factor of 1. In this configuration, each of the 16 nodes is solely responsible for the blocks it stores. The arrangement of the 16 nodes is illustrated in Figure 6. We also deployed a *bootnode* so that peers discover and connect to each other among the private instance. In addition, we generated about 7,500 blocks to populate our test instance of Ethereum, representing 1.4 GB of data, which we preloaded onto the 16 clients. The experiment finally consists in inserting a test node, running our modified Geth client with different combinations of the new options, joining our test network and synchronizing, i.e. downloading the blocks it needs.

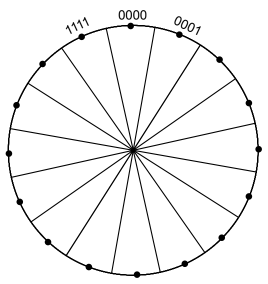

Figure 6: Forced placement of our 16 nodes in the DHT, partitioning uniformly the address space

5.2 Case of a complete synchronization by DHT (`--syncFromDHT` mode)

Our first evaluation concerns a classical synchronization when the synchronizing node will only request blocks from the peer responsible for the regarding blocks according to its location in the DHT. Thus, the node evaluates, according to the hash of the blocks' header, which peer(s) to contact to retrieve a particular block. Figure 7 shows the node retrieval of all the blocks, but in Figure 8, we observe that, unlike a classic synchronization (Figure 5), all peers in the small test network contribute uniformly to send these blocks. Of course in the real network all peers will not be contacted but, with a prefix length of 5 bits as we proposed, at least 32 nodes will be contacted to synchronize a new node willing to store all the blockchain's data. This experiment also proves the backward compatibility of our approach.

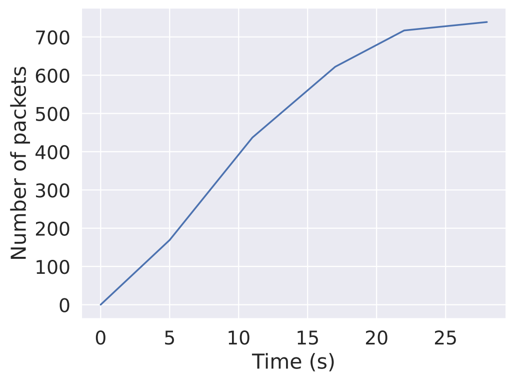

Figure 7: Cumulative block packets received by the joining node asking for all blockchain's data through the DHT.

5.3 Case of a reduced synchronization by DHT (`--syncFromDHT --storeInDHT` mode)

Our second experiment shows the validity of our new synchronization solution. The node that joins our private P2P network only requests the blocks for which it is responsible. As a reminder, this means that these are the blocks which header hash is close to its own NodeID, in the sense of DHT distance mentioned in section 4. Figure 9 shows that the node requires far fewer blocks to download and thus synchronizes faster[8], namely 7 seconds instead of 27 s. We also observe in Figure 10 that only one peer is requested, as expected, which is due to our test configuration with a replication factor equal to one. In this configuration, the node that synchronizes takes advantage of a massive storage gain compared to the previous synchronization modes.

Table 4 summarizes the characteristics we observed during our evaluation. We also calculated the Median Absolute Deviation (MAD) of the number of block packets downloaded by the synchronizing client, and it clearly shows that the use of the DHT allows a more homogeneous workload. Please note that in the case of the reduced synchronization, the standard deviation calculation does not really apply, because in our experiment there is only one node able to send packets to the synchronizing node.

6 DISCUSSION ON THE BLOCKCHAIN'S SECURITY

We want to discuss the potential impact of our solution on the blockchain's security to convince the reader of its safety. One may think that our solution may threaten the overall security of the Ethereum's blockchain by reducing the replication factor of old blocks to the whole network to a subpart of it. We must justify the safety of our solution against two main threats: the partition of the

[7] using this program we developed https://github.com/jpeisenbarth/go-ethereum/tree/DHT-storage

[8] in the context of a real blockchain synchronization, the node would also need to download the Snapshot

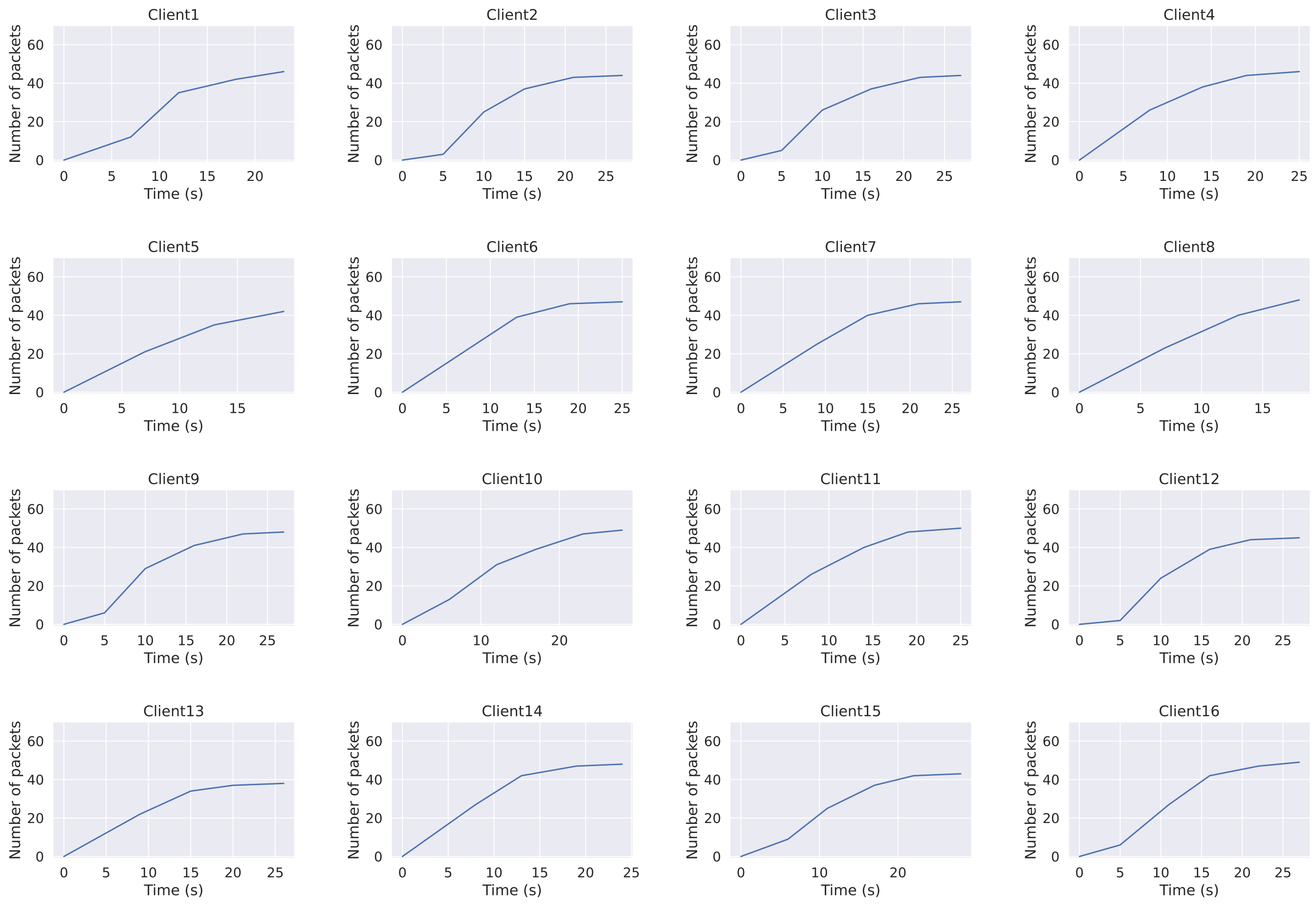

Figure 8: Cumulative block packets sent by the peers to the joining node asking for all blockchain's data through the DHT.

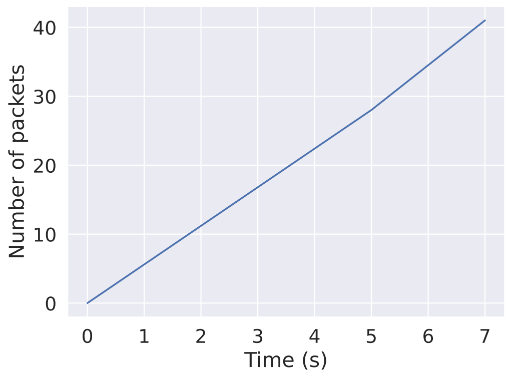

Figure 9: Cumulative block packets received by the joining node asking only for the blockchain's data it must store

Table 4: Summary of experiment results

Sync mode	Option	Sync time	MAD of block packets sent by peers
Classic	∅	27 s	8.5
Complete by DHT	`--syncFromDHT`	27 s	2.0
Reduced by DHT	`--syncFromDHT` `--storeInDHT`	7 s	0.0

P2P network and localized Sybil attacks that could take the control over a region of the DHT, thus allowing a denial of service attack preventing the retrieval of some targeted blocks (eclipse attack) and/or the propagation of corrupted blocks by malicious peers.

Previous studies [6, 9, 16] have shown the feasibility of an Eclipse attack to partition the network and expose it to double spending attacks, but we argue that our solution does not lower the security against such attack. Indeed, our proposal concerns only the ossified part of the blockchain (cold storage) : the 30 000 most recent blocks (hot data) remain available on all peers because they are the most critical ones allowing the change of the world state.

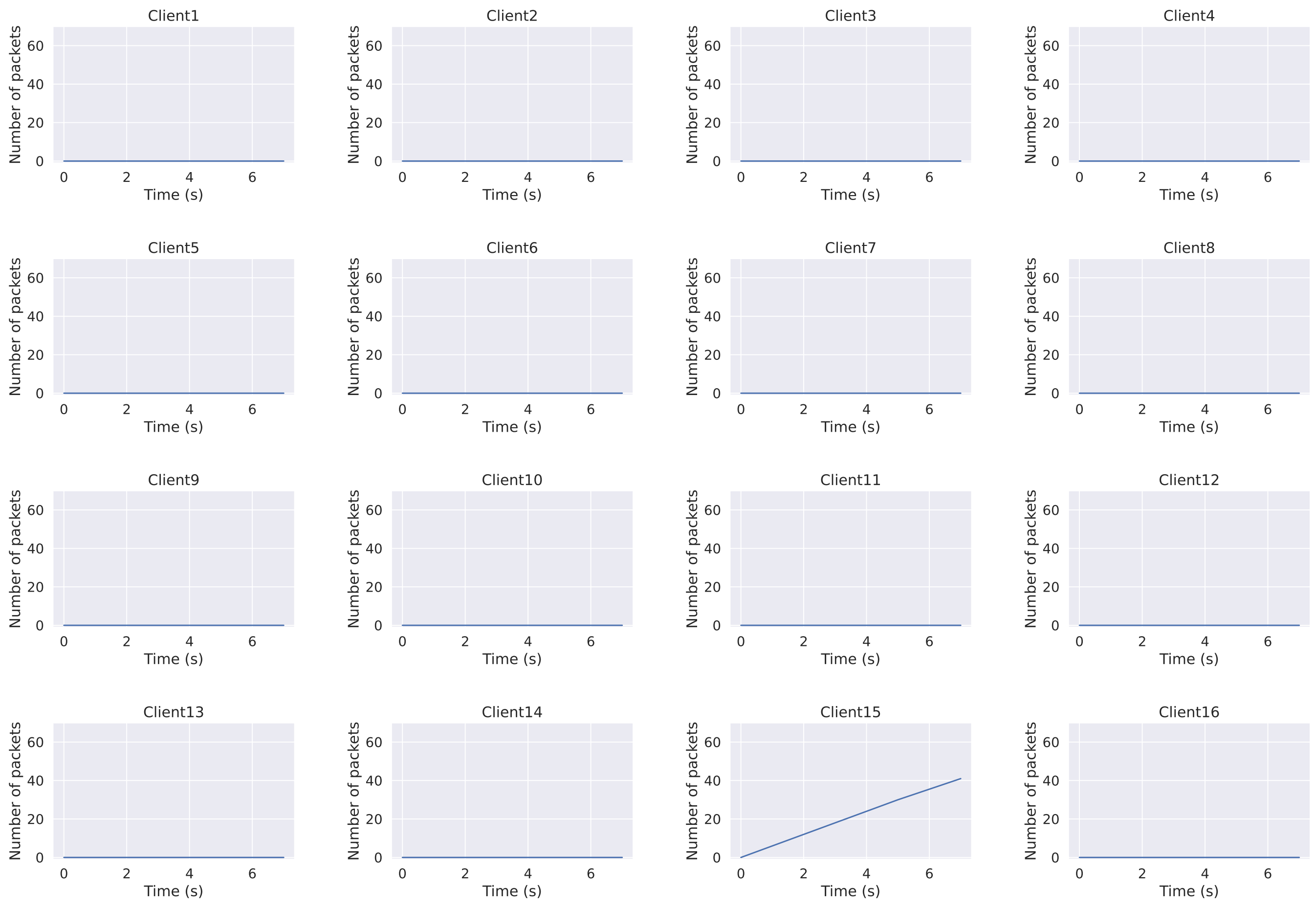

Figure 10: Cumulative block packets sent by the peers to the joining node asking only for the blockchain's data it must store

Against localized Sybil attacks (conducting to eclipse a specific resource), we oppose three arguments. First, we propose a relatively high replication factor (1125 as suggested in Section 4). Such a replication factor is a low assumption, ensuring a very high storage reliability. It would be very difficult to create enough Sybil nodes to overtake 1125 legitimate nodes and make a block disappear. In practice, the replication rate could probably be safely lowered, but the exponential decrease of space saving makes longer prefix less interesting, much of the gain being given by the first bits. Second, some Sybil attack prevention architecture have already been recommended for Ethereum [5] and a few simple preventive rules presented in the literature [2] are known to make such attacks even more costly and unlikely to happen. Third, because the worst case scenario must always be considered, some nodes with a high storage capacity can still decide to follow the current storage and synchronization policy and ultimately act as full backups of the blockchain in case of a successful attack against a region of the DHT. Concerning the corruption of old blocks, we can also rely on the inner security mechanisms of Ethereum that make it almost impossible to tamper with a block.

Finally, Ethereum's recent (end of year 2022) switch from proof-of-work to proof-of-stake does not impact our proposal : the storage of the blocks is not concerned by this modification and the storage capacity problem fully remains.

7 CONCLUSION

This paper addresses the critical problematic of blockchain's data storage. We provided a comprehensive description of Ethereum's synchronization and storage strategies through the investigation of Geth source code and technical writings of the developers. We explained why there is no point for all nodes to store all blocks anymore, since they do not execute the transactions contained in the blocks they download since the introduction of Fast and Snap synchronization modes. So we propose to distribute the ossified part of the blockchain currently stored in FreezerDB over the Kademlia DHT that is implemented in all Ethereum clients but not used so far. We implemented our solution in the official Geth client and evaluated our new synchronization and storage mode exploiting the DHT on a small private Ethereum instance. It proved the viability of our solution that is also fully backward compatible with current clients. For a replication factor of 1125 peers, this new distributed storage strategy can reduce the size of long-term storage for each peer, which represent 95% of the FreezerDB volume and 58% of the total volume. At the scale of the whole P2P network, this represents an overall gain of about 12 PB without reducing in practice the blockchain reliability.

In our future work, we plan to study the impact of the upcoming *Sharding* feature announced by Geth developers which might be complementary with our approach.

ACKNOWLEDGMENTS

We thank Thomas Martignon for his work on the storage of Ethereum data as part of his internship in our research group. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 830927.

REFERENCES

[1] Mirko Bez, Giacomo Fornari, and Tullio Vardanega. 2019. The Scalability Challenge of Ethereum: An Initial Quantitative Analysis. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. 167–176. https://doi.org/10.1109/SOSE.2019.00031

[2] Thibault Cholez, Isabelle Chrisment, Olivier Festor, and Guillaume Doyen. 2012. Detection and Mitigation of Localized Attacks in a Widely Deployed P2P Network. *Peer-to-Peer Networking and Applications* 6 (June 2012). https://doi.org/10.1007/s12083-012-0137-7

[3] Mikel Cortes-Goicoechea, Luca Franceschini, and Leonardo Bautista-Gomez. 2021. Resource Analysis of Ethereum 2.0 Clients. In *2021 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*. 1–8. https://doi.org/10.1109/BRAINS52497.2021.9569812

[4] John R. Douceur. 2002. The Sybil Attack. In *Peer-to-Peer Systems (Lecture Notes in Computer Science)*, Peter Druschel, Frans Kaashoek, and Antony Rowstron (Eds.). Springer, Berlin, Heidelberg, 251–260. https://doi.org/10.1007/3-540-45748-8_24

[5] Jean-Philippe Eisenbarth, Thibault Cholez, and Olivier Perrin. 2022. Ethereum's Peer-to-Peer Network Monitoring and Sybil Attack Prevention. *J Netw Syst Manage* 30, 4 (Oct. 2022), 65. https://doi.org/10.1007/s10922-022-09676-2

[6] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. 2015. Eclipse Attacks on Bitcoin's Peer-to-Peer Network. In *Proceedings of the 24th USENIX Conference on Security Symposium (SEC'15)*. USENIX Association, Berkeley, CA, USA, 129–144. https://doi.org/10.5555/2831143.2831152

[7] Periklis Kostamis, Andreas Sendros, and Pavlos Efraimidis. 2021. Exploring Ethereum's Data Stores: A Cost and Performance Comparison. In *2021 3rd Conference on Blockchain Research Applications for Innovative Networks and Services (BRAINS)*. 53–60. https://doi.org/10.1109/BRAINS52497.2021.9569804

[8] J. Liang, N. Naoumov, and K. W. Ross. 2006. The Index Poisoning Attack in P2P File Sharing Systems. In *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications (Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications)*. https://doi.org/10.1109/INFOCOM.2006.232

[9] Yuval Marcus, Ethan Heilman, and Sharon Goldberg. 2018. Low-Resource Eclipse Attacks on Ethereum's Peer-to-Peer Network. *IACR Cryptology ePrint Archive* 2018 (2018), 236.

[10] Petar Maymounkov and David Mazières. 2002. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *Peer-to-Peer Systems*, Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Peter Druschel, Frans Kaashoek, and Antony Rowstron (Eds.). Vol. 2429. Springer Berlin Heidelberg, Berlin, Heidelberg, 53–65. https://doi.org/10.1007/3-540-45748-8_5

[11] Robert Norvill, Beltran Borja Fiz Pontiveros, Radu State, and Andrea Cullen. 2018. IPFS for Reduction of Chain Size in Ethereum. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. 1121–1128. https://doi.org/10.1109/Cybermatics_2018.2018.00204

[12] P. Poornima Devi, Srinivasan Ananthanarayanan Bragadeesh, and Arumugam Umamakeswari. 2021. Secure Data Management Using IPFS and Ethereum. In *Proceedings of International Conference on Computational Intelligence, Data Science and Cloud Computing (Lecture Notes on Data Engineering and Communications Technologies)*, Valentina E. Balas, Aboul Ella Hassanien, Satyajit Chakrabarti, and Lopa Mandal (Eds.). Springer, Singapore, 565–578. https://doi.org/10.1007/978-981-33-4968-1_44

[13] Daniel Sel, Kaiwen Zhang, and Hans-Arno Jacobsen. 2018. Towards Solving the Data Availability Problem for Sharded Ethereum. In *Proceedings of the 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers (SERIAL'18)*. Association for Computing Machinery, New York, NY, USA, 25–30. https://doi.org/10.1145/3284764.3284769

[14] Moritz Steiner, Taoufik En-Najjary, and Ernst W. Biersack. 2007. Exploiting KAD: Possible Uses and Misuses. *SIGCOMM Comput. Commun. Rev.* 37, 5 (Oct. 2007), 65–70. https://doi.org/10.1145/1290168.1290176

[15] Lee Thomas. 2016. BlockchainIllustrations/Ethereum at Master · 4c656554/BlockchainIllustrations. https://github.com/4c656554/BlockchainIllustrations.

[16] Guangquan Xu, Bingjiang Guo, Chunhua Su, Xi Zheng, Kaitai Liang, Duncan S. Wong, and Hao Wang. 2020. Am I Eclipsed? A Smart Detector of Eclipse Attacks for Ethereum. *Computers & Security* 88 (Jan. 2020), 101604. https://doi.org/10.1016/j.cose.2019.101604

[17] Huijuan Zhang, Chengxin Jin, and Hejie Cui. 2018. A Method to Predict the Performance and Storage of Executing Contract for Ethereum Consortium-Blockchain. In *Blockchain – ICBC 2018 (Lecture Notes in Computer Science)*, Shiping Chen, Harry Wang, and Liang-Jie Zhang (Eds.). Springer International Publishing, Cham, 63–74. https://doi.org/10.1007/978-3-319-94478-4_5

[18] Qiuhong Zheng, Yi Li, Ping Chen, and Xinghua Dong. 2018. An Innovative IPFS-Based Storage Model for Blockchain. In *2018 IEEE/WIC/ACM International Conference on Web Intelligence (WI)*. 704–708. https://doi.org/10.1109/WI.2018.000-8