Ethanos: Efficient Bootstrapping for Full Nodes on Account-based Blockchain

Jae-Yun Kim, Junmo Lee, Yeonjae Koo, Sanghyeon Park, and Soo-Mook Moon*
Seoul National University
{jaeyun.kim,junmo.lee,ricegod,lukepark,smoon}@snu.ac.kr

Abstract

Ethereum is a popular account-based blockchain whose number of accounts and transactions has skyrocketed, causing its data explosion. As a result, ordinary clients using PCs or smartphones cannot easily bootstrap as a *full node*, but rely on other full nodes to verify transactions, thus being exposed to security risks. The most serious overhead is caused by synchronizing the state of all accounts in the block's *state trie*, which takes several tens of gigabytes. Observing that more than 95% of the accounts are dormant, we propose a novel state optimization technique, named *Ethanos*. Ethanos downsizes the state trie by periodically *emptying* it, and then re-build it only with the *active* accounts used in the period's transactions. Ethanos runs transactions using the accounts available in the current period's state trie as well as those available at the end of the previous period's state trie. For an account in neither of the tries, the account first restores itself by transmitting a *restore* transaction. One important result of this state management is that a node can now bootstrap only with the latest period's state trie, yet can fully verify all transactions thereafter. We evaluated Ethanos with real Ethereum transactions for 300,000 blocks from the 7.0 million block, with a one-week period of emptying the state trie. Our result shows that Ethanos can sharply reduce the state trie, with only a tiny fraction of the restore transactions. More importantly, unlike the Ethereum state trie which continues to grow as time goes on, the Ethanos state trie size at the end of each period is bounded by a few hundred MB, when there are more than one million, one-week-active accounts.

CCS Concepts: • **Software and its engineering**;

Keywords: Blockchain, Synchronization, State trie

ACM Reference Format:
Jae-Yun Kim, Junmo Lee, Yeonjae Koo, Sanghyeon Park, and Soo-Mook Moon. 2021. Ethanos: Efficient Bootstrapping for Full Nodes on Account-based Blockchain. In *Sixteenth European Conference on Computer Systems (EuroSys '21), April 26–29, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3447786.3456231

1 Introduction

Ethereum is an account-based blockchain system [1]. Unlike *Bitcoin*, a UTXO (unspent transaction output)-based system that implicitly represents the global state of the blockchain using unspent transactions [2], Ethereum explicitly records the global state of accounts/balances separately from the transactions. This state-based model is simpler and more intuitive for developing *smart contract*, a tiny Turing-complete program which performs specific tasks using account states when invoked by transactions, and whose integrity is verified by every node in the blockchain. This is the reason that account-based model has become a mainstream, so many of modern blockchain systems such as Cardano [3], Polkadot [4], Solana [5], or Algorand [6] follow the model.

The popularity of Ethereum has exposed one problem of account-based blockchain, though. The soaring number of accounts and transactions made the Ethereum data more than 3.6TB as of Jan 2020 [7]. It is more serious than Bitcoin, since the Ethereum block should save the account states as well as the transactions. As a result, ordinary people using personal devices with limited storage and network bandwidth cannot easily bootstrap as a *full node* [8] that can completely verify transactions. If they cannot fully validate transactions themselves, they, as a *light node* [8], should rely on other full nodes such as miners or central service providers like Infura [9], to delegate transmission and validation of transactions. However, this might make the users vulnerable to financial fraud or transaction censorship. More seriously, if the number of full nodes and miners is small, the blockchain network itself becomes vulnerable to manipulation, compromising the security of the blockchain.

To solve this problem, we need to reduce the essential data to become a full node. The primary bottleneck is the *state trie* that stores all account states using a data structure called *Merkle Patricia Trie (MPT)* [10]. Normally, a node becomes a full node with *full sync*, by running all transactions from the genesis block to the current block to reproduce the current state trie and all of its update histories. This takes a long time and requires a huge space. *Go-ethereum (Geth)* client, one of the most popular Ethereum clients [11], employs a more efficient synchronization mode called *fast sync*. Fast sync downloads only the snapshot of the state trie of the *pivot block*, 64 blocks behind the current block, and replays only the transactions from the pivot block to the current block to reproduce the current state trie. Fast sync obviates generating and saving the update histories of the old state tries, substantially reducing the storage size to about 235GB in Jan 2020 [7], only 6.7% of the total Ethereum data. However, 235GB is still too heavy for an ordinary client to synchronize.

*Corresponding author.

More seriously, the MPT-based state trie, estimated more than 70GB, requires hash-based random database access, which renders traversing, transferring, and rebuilding the state trie heavily suffer from high disk I/O (only SSD, not HDD, is usable [12]), making fast sync still take a long time.

Through in-depth investigation of the Ethereum data, however, we found that more than 95% of Ethereum accounts are dormant, that is, not active for a week or for a month. Based on this observation, we propose a new state optimization technique named *Ethanos* that periodically sweeps dormant accounts to reduce the size of the state trie. Since it is not easy to isolate and remove dormant accounts from the state trie, Ethanos simply builds an empty trie at the start of each period, and then re-build only with the *active* accounts used in the period's transactions. For this, we cache the state trie at the end of the previous period, so that if an account used by a transaction is not available in the current trie (initially so since we start with an empty trie), we search for it from the cached trie, copy it to the current trie, and then update it as a result of the transaction. If the account is in neither of the tries when the account wants to be a *sender* of a transaction, the account first restores itself by transmitting a *restore* transaction; if the account is a *receiver*, we do not have to restore it but create a "*crumb*" account as if we are creating a new account, then merge later with the original account when it becomes a sender. We implement restore transaction efficiently with Merkle proof, Void proof, and bloom filters.

One important consequence of the proposed state management is that a node now can bootstrap only with the latest period's state trie, yet can still perform full verification of all transactions. We experimented with the real Ethereum data from the 7.0 million to 7.3 million blocks, when Ethereum blocks are fully packed with many transactions and active accounts. We found that Ethanos can reduce the state trie size of each one-week period to around 220MB, which is tiny compared to the full state trie size of 70GB. More importantly, the size of Ethanos trie would remain the same while the size of Ethereum trie increases as time goes on, since Ethanos trie will include only the active accounts in each period. If we exploit additional optimizations to obviate old transactions [13] and less-important block headers [14], Ethanos can reduce the space/time cost sharply, reasonable enough for ordinary users to synchronize as a full node. This not only increases the number of full nodes/miners, thus improving blockchain security, but allows occasional on-line users to be a full node whenever transmitting a transaction, verifying it for themselves, thus improving user security. This paper made the following contributions:

- We proposed a technique to sharply reduce the state trie of account-based blockchain system by emptying it periodically to remove dormant accounts.
- We proposed a restore transaction which can reactivate a dormant account, possibly after merging with its crumb accounts scattered across different periods.
- We designed a full-node protocol that can quickly and efficiently bootstrap with the latest period's state trie, and fully verify all transactions thereafter.
- We evaluated Ethanos with real Ethereum transactions on a real Geth client, to show that it can sharply reduce the synchronization cost and work efficiently.

The rest of this paper is composed of as follows. Section 2 reviews Ethereum and its state trie. Section 3 observes the Ethereum account activities. Section 4 describes how Ethanos works. Section 5 shows the experimental results. Related work is in Section 7. Summary is in Section 8.

2 Review of Ethereum and State Trie

To make the paper self-contained, we review how Ethereum works, especially its state trie. Ethereum is a representative account-based blockchain that maintains a singleton chain of blocks, synchronized among the participating nodes in a peer-to-peer network. Its state, σ, is a set of address-account pairs that are updated by the transactions [10]. The updated state and the transactions are recorded in a block and propagated across the entire Ethereum network by *Gossip protocol* [15]. A block is a unit of state synchronization generated in every round, and a node that creates a block is called a *miner*. Other nodes in the network receive the block and replay the transactions to reproduce and verify the current state. When most participating nodes synchronize to a block with the current state σ_c, a new round starts to generate the next state σ_{c+1}. A miner is decided pseudo-randomly by sortition based on an algorithm named *Ethash* [16], similar to *proof-of-work* in Bitcoin [17]. It uses the *longest chain rule* to solve the race condition when more than one block are generated in the same round, finding the canonical chain and rejects orphan blocks. These *consensus algorithms* guarantee that all participants eventually synchronize to a single state with identical blocks. Figure 1 shows an example of Ethereum block chain and the account state in each block, which will be explained shortly.

2.1 Account

An account in Ethereum is accessed by an *address* composed of 40 hexadecimal characters. An account includes four types of data: *nonce*, *balance*, *storageRoot*, and *codeHash*. *nonce* records the number of transactions the account has ever made, to prevent a single transaction from being executed multiple times. *balance* shows the amount of *Ether (ETH)*, the cryptocurrency of Ethereum, that the account has. If an account is an *externally-owned account (EOA)*-i.e., user's account, its *storageRoot* and *codeHash* fields are *null*. Otherwise, the account is a *contract account (CA)*, and the *storageRoot* has the key of the storage where the value is the state of the smart contract, while the *codeHash* shows the hash value of the code used to retrieve the contract code from the database.

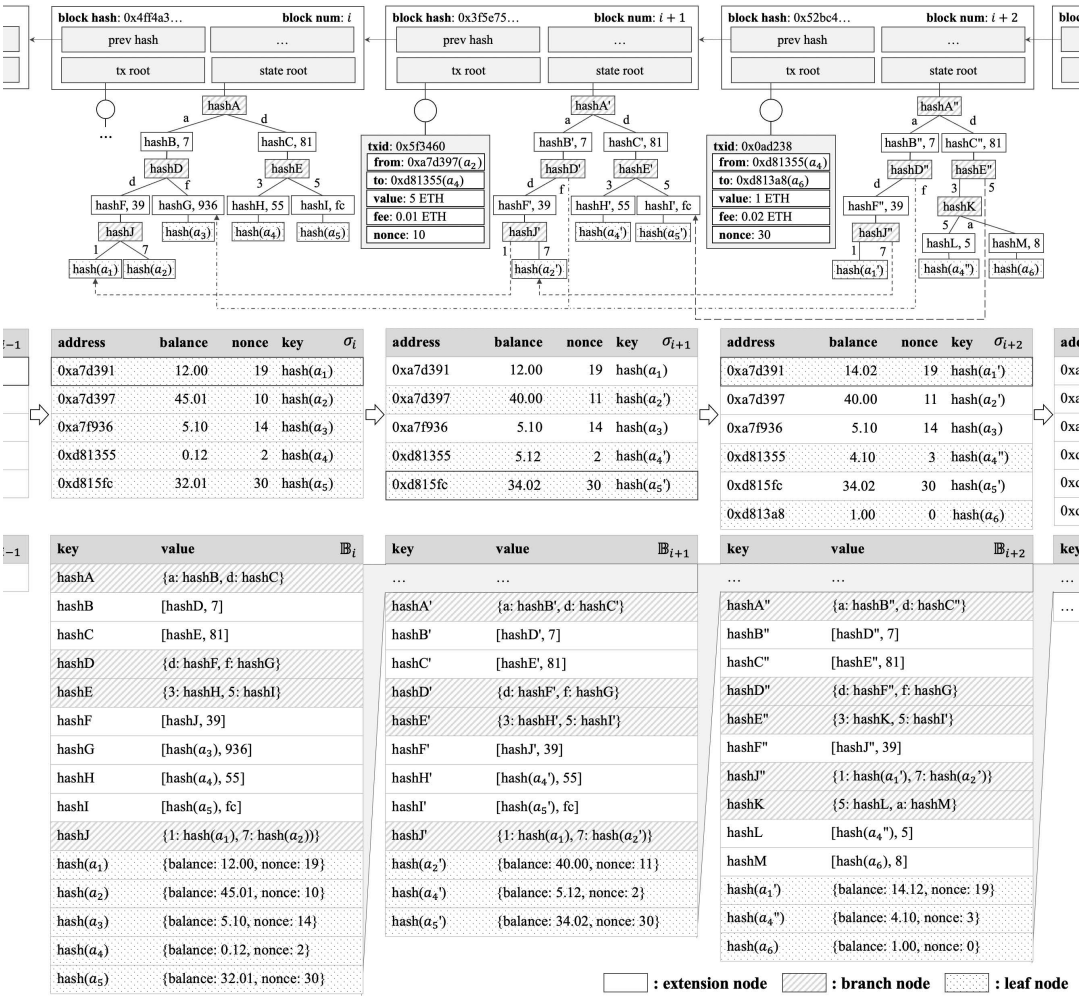


Figure 1. State trie, state (σ), and database ($\mathbb{B}$) structure of Ethereum.

2.2 Transaction

Transaction transfers a value from a sender (*from*) account to a receiver (*to*) account, thereby changing the *balance* of both accounts, and increments *nonce* of the sender account. To prove the validity of a transaction, one needs to check if the sender has a sufficient *balance*, and the transaction *nonce* is equal to the sender's *nonce*. A transaction includes a transaction fee to be credited to the miner, who will also get a block reward for successful mining, so the *balance* of the miner account is also updated. Each transaction contains a signature of the sender account using private key to prove the ownership of the digital asset that the transaction transfers.

2.3 State Transition

Let Tx_i be a transaction list of the block i and Tx_i^n be the $n+1^{th}$ transaction in the Tx_i list [10, 18]. Formally: $Tx_i = [Tx_i^0, Tx_i^1, \cdots, Tx_i^{j-1}]$ when the list size is j. Then, block-level state transition function Π is defined as: $\sigma_i = \Pi(\sigma_{i-1}, Tx_i)$ where σ_{i-1} is the state of the block i-1, and Π applies each transaction in Tx_i to σ_{i-1} in a serial order, generating σ_i, the state of the block i. Equivalently, we can obtain σ_i by sequentially applying transaction lists to the state σ_0 of the genesis block: $\sigma_i = \Pi(\cdots\Pi(\Pi(\sigma_0, Tx_1), Tx_2), \cdots, Tx_i)$.

For example, Figure 1 (left) shows that σ_i of the block i consists of five accounts whose addresses are 0xa7d391 (a_1), 0xa7d397 (a_2), 0xa7f936 (a_3), 0xd81355 (a_4), 0xd815fc (a_5) (we represent an address with six characters for simplicity, and assume that all accounts are EOA). If the transaction list Tx_{i+1} of block $i+1$ contains only one transaction Tx_{i+1}^0 that transfers 5 ETH from the account a_2 to the account a_4 with a 0.01 ETH fee, the *balance* of a_2 decreases from 45.01 to 40.00 ETH and the *nonce* increases by one. Also, the *balance* of a_4 increases from 0.12 to 5.12 ETH with no change of its *nonce*. If a_5 is the miner of the $i+1^{th}$ round, the *balance* of a_5 is incremented by 2.01 ETH (0.01 ETH is the transaction fee and 2 ETH is the block reward). This constitutes σ_{i+1} as depicted in the figure (middle). A transaction Tx_{i+2}^0 of block $i+2$ makes the account a_4 sends 1 ETH to a new account a_6 whose address is 0xd813a8, which creates a_6, and updates a_4 and a_1 (miner), leading to σ_{i+2} in the figure (right).

2.4 State Trie

Ethereum stores the account states in each block using a *Merkle Patricia Trie* (MPT) [10], a data structure combining *Patricia trie* [19] and Merkle tree [20]. *Patricia trie* is a search tree saving (*address*, *data*) of accounts. The character sequence of an address becomes a deterministic path to its data located at the leaf of the trie, and those accounts with common prefixes in their address share the same sub-paths in the trie. This makes address search efficient. *Merkle* tree labels every leaf node with the hash value of their data, and labels every intermediate node with the hash of the labels of its child nodes recursively, until labeling the root node. This allows efficient and secure verification of a large tree since any single change of data leads to a different root hash.

As a merged data structure, MPT is implemented as follows. MPT has two types of intermediate nodes: a *branch node* and an *extension node*. In Figure 1, for example, the root node of the block i's state trie (labeled by *hashA*) is a branch node that distinguishes five addresses into two groups: those starting with a and those starting with d. The child of a branch node is an extension node to add prefixes common to its descendants before branching; for example, the node after branching thru d (labeled by *hashC*) adds the common prefix 81 for its descendant accounts, a_4 and a_5. The child of an extension node is another branch node, so this alternation of the two types of nodes repeats until the address of each account can be distinguished, and the account is added as a leaf node. Now, MPT labels each leaf node and intermediate node with a *hash* (key), computed using its *node data* (value). This allows implementing the MPT using a key-value database such as *LevelDB* [21]. More specifically, an extension node is a two-item node of the form [*key*, *path*] where *path* is the common prefix and *key* is the hash of its only child node, used for looking up the child in the database. A branch node is a 17-item node of the form $[v_0, v_1, \cdots, v_f, value]$ where the position of v_i means the one-character prefix i (from 0 to f) and v_i is the *key*, the hash for the corresponding child node (v_i is null if there is no child of the prefix i). Figure 1 (left) shows the database $\mathbb{B}_i$ corresponding to σ_i for the block i, which depicts the key-value for each node.

Now we see how each transaction updates the state trie. When an account is updated as a result of a transaction, its hash should also be updated. Instead of just changing the hash for the corresponding leaf node, Ethereum *creates* a new trie node with a new key (hash) and value (data), then *appends* it to the database. This will make the parent node also be updated, appending a new trie node for it. This continues until a new root node is added. Figure 1 shows that the transaction Tx_{i+1} adds new nodes at the end of $\mathbb{B}_i$, leading to $\mathbb{B}_{i+1}$, so the state root of the block $i+1$ now has the hash for the new root (hashA'). Similarly, Tx_{i+2} adds new nodes including a new account node for a_6, leading to $\mathbb{B}_{i+2}$. By adding new trie nodes, Ethereum can keep track of the history of all state tries (e.g., the block i can still access the old root and its descendants, thus its snapshot of state trie, even after the block $i+1$ is added), which strengthens the security of the blockchain, yet with a high space overhead.

2.5 Proof of Membership

Keeping the update history for the state trie allows a simple membership check for an account, only using the *block header* even without the state trie. Figure 1 shows what the block header stores: *prev hash* (the hash of the previous block), *tx root* (the root hash of the transaction trie in the *block body*), and *state root* (the root hash of the state trie). Membership of an account in a particular block can be proven only with *state root* in the block header, if a *Merkle proof* is given. The Merkle proof of an account for a state trie is composed of every node data from the root node to that account node. For example in Figure 1, the Merkle proof of an account a_5 at the block i is a list of node data, expressed using $\mathbb{B}$ (=$\mathbb{B}_i$) as follows: $P_M[i]_{a_5}$=[$\mathbb{B}(hashA)$, $\mathbb{B}(hashC)$, $\mathbb{B}(hashE)$, $\mathbb{B}(hashI)$, $\mathbb{B}(hash(a_5))$] , where $\mathbb{B}(hashA) = \{a : hashB, d : hashC\}$, $\mathbb{B}(hashC) = [hashE, 81]$, $\mathbb{B}(hashE) = \{3 : hashH, 5 : hashI\}$, $\mathbb{B}(hashI) = [hash(a_5), fc]$, $\mathbb{B}(hash(a_5)) = \{balance : 32.01, nonce : 30\}$. Then, if a verifier is a given a Merkle proof $P_M[i]_{a_5}$, it can check if a_5 really existed in a block i and if a_5 had the same data as in $P_M[i]_{a_5}$, only with the *state root* value of the block header (the state trie itself or the block body are unnecessary). That is, the verifier reproduces the hash keys by applying the hash function to each node data in $P_M[i]_{a_5}$ recursively from the leaf to the root, as follows: $hash(a_5) = hash(\{balance : 32.01, nonce : 30\})$, $hashI = hash([hash(a5), fc])$, $hashE = hash(\{3 : hashH, 5 : hashI\})$, $hashC = hash([hashE, 81])$, $hashA = hash(\{a : hashB, d : hashC\})$. Finally, we can check the membership of a_5 and its account data simply by comparing $hash(hashA)$ with the *state root*. This makes sense because it is almost impossible to manipulate some data (e.g., *balance*, *nonce*, fc, *hashH*, 81, *hashB*) to produce the same root hash in the Merkle proof.

Similarly, one can prove non-membership of an account using a *Void proof*. For example, we can generate a Void proof to show that some account 0xa7ec6b (a_n) does not exist in state σ_i as follows: $P_V[i]_{a_n} = [\mathbb{B}(hashA), \mathbb{B}(hashB), \mathbb{B}(hashD)]$. If a verifier is given the Void proof, it can reproduce *hashA*, *hashB*, and *hashD* by applying hash function for each item in $P_V[i]_{a_n}$, to confirm the validity of the proof. Then it can know that $\mathbb{B}(hashD) = \{d : hashF, f : hashG\}$ does not contain the key e, meaning that σ_i does not have an account whose address starts with 0xa7e. Actually, our proposed Ethanos exploits the Merkle proofs and the Void proofs effectively for restore transactions (see Section 4).

2.6 Synchronization

When a new node joins the Ethereum as a full node, it must synchronize to the current state, which involves downloading all blocks, replaying the transactions from the genesis

block to the current block to regenerate the historical data including all state tries, and verifying all block headers based on the data [22]. This full node is also called an *archive node*. However, because of the enormous data, full sync requires massive storage and excessively long running time (a few weeks). So, the Geth client provides another synchronization mode to be a full node, *fast sync* [23], which downloads all blocks as in full sync, but also downloads a snapshot of the state trie for a block called the *pivot block* (which is 64 blocks behind the current block in the latest version of Geth), then replays only the transactions from the pivot block to the current block to generate the current state trie. The idea is that older blocks are harder to counterfeit, thus lesser need for verification. Since fast sync bypasses the replay process of the transactions that occurred before the pivot block, it can reduce the space overhead and the synchronization time. Unfortunately, fast sync still requires a huge space of about 235GB in Jan 2020 and takes a day. Among 235GB, the state trie is estimated to be over 70GB [24], while the rest is dominated by the transaction data. As to the synchronization time, it is mainly composed of *state trie download phase* and *transaction download phase*, which are parallelizable. Between the two, the state trie download phase is known to be more critical because it requires traversing, transferring, and rebuilding a huge MPT, which suffer from intensive disk I/O due to MPT's hash-based database accesses [12], so it is a bottleneck for synchronization (although transactions are much larger, we can batch-download them relatively fast using the transaction index in the block).

3 Observation of Account Activity

To reduce the bottleneck of synchronization, we first of all need to reduce the size of the state trie to download. For this, we observed the activity of the Ethereum accounts. We found that more and more accounts are getting dormant as time goes on and the proportion of the dormant accounts is increasing, when we trace the number of active accounts from the genesis block to the 8 million block, as depicted in Figure 2; for each weekly-sampled block we count the number of accounts that send, receive, or mine during the last one week, and depict it as one-week active accounts for that block. Similarly, we measure one-month active accounts. We can see that only 3% and 5% accounts are active for one-week and for one-month periods, respectively. This trend continues as of Jan 2020, where the account number is more than 85M, growing at a rate of about 50K–150K per day [7].

Another observation is that active accounts are likely to be updated soon after they are updated. For each account we measure the average time distance between two consecutive updates (as a sender or a receiver of a transaction, or as a miner of a block), and classify the accounts by the average time distance, as shown in Table 1. We can roughly interpret that an account is updated in a week ($\approx 40,320$ blocks) with

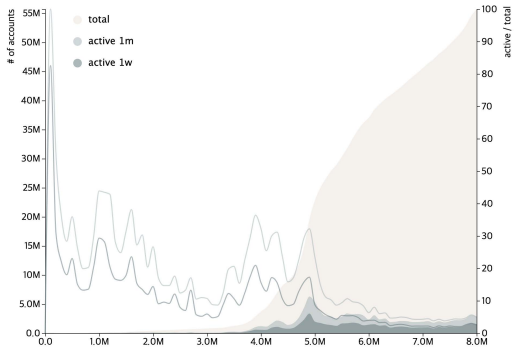


Figure 2. Number of active accounts (left axis, line of shaded area) and its ratio (right axis, line) at each block of Ethereum.

distance (≤)	1 day	1 week	2 weeks	1 month	6 months
ratio (%)	62.3	76.0	82.4	89.0	98.4

Table 1. Average update distance of the accounts.

a probability of 76%, and in a month ($\approx 172,800$ blocks) with a probability of 89%, after it is updated.

If we combine both observations, we can say that only a fraction of accounts are active at any point of time, but once they are activated, they are likely to be accessed again soon. This motivates the idea of *Ethanos*. That is, sweeping dormant accounts once a week or once a month would sharply reduce the state trie size, allowing a faster sync. Also, we can handle most transactions of this week or this month with the (sharply-reduced) state trie of the previous week or the previous month, respectively, since many accounts active in the previous trie will be active again in the current trie. If an account required for processing a transaction is not available in the previous trie, we need to restore the account before invoking a regular transaction, but the frequency will be low. Finally, the size of a reduced state trie would not grow as time goes on, unlikely the Ethereum state trie, since the number of transactions mined with a week or a month is bounded, so is the number of active accounts. The problem is how to sweep dormant accounts, restore them if needed, and synchronize efficiently, which will be discussed below.

4 Ethanos

Ethanos is a novel state optimization technique, proposed for account-based blockchain such as Ethereum. Ethanos periodically sweeps and rebuilds the state trie, and uses the reduced trie for bootstrapping. This requires a new way of governing transactions and synchronizing to a state.

4.1 Sweep

Ethanos defines an *epoch*, λ, the number of blocks after which Ethanos sweeps the dormant accounts periodically (λ is around 40,320 blocks for one-week period in Ethereum). We call the last block of an epoch a *checkpoint*. Since it is extremely expensive to traverse a state trie to find and remove

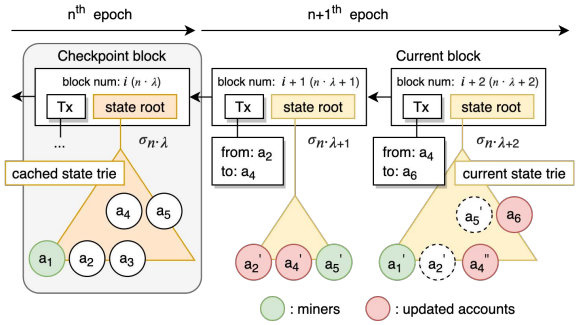


Figure 3. State trie of Ethanos (simplified from Figure 1).

dormant accounts, Ethanos simply creates a new empty trie every time a new epoch starts, and fill it with the active accounts used by the transactions in the new epoch. Miners or verifiers will execute a transaction as usual, using the new trie to find and update the accounts involved with the transaction. Since the trie is initially empty, however, no account is available, so Ethanos *caches* the state trie of the previous checkpoint and use it for finding an account state as well. That is, if an account used by a transaction is not available in the new trie, we search for it from the cached trie, and if it is there, copy it to the new trie, and update it as a result of the transaction. If the account is not available in neither of the tries when the account wants to be a sender of a transaction, it should first restore itself. At the end of the current epoch, Ethanos caches the current state trie, discarding the old cached trie, then starts a new epoch and the same process repeats.

The reason that Ethanos caches only one previous state trie (instead of caching multiple tries for better chance of finding) is that this allows a full node to bootstrap with a single trie, thus reducing the download size. Also, it will make every full node behave the same (instead of using multiple cached tries by some nodes while using one cached trie by others), as in the original blockchain. Finally, the state tries of consecutive checkpoints will include many redundant, active accounts, so it is more efficient to use only one.

Figure 3 illustrates how Ethanos blockchain works using the example in Figure 1. We assume that the n^{th} epoch ends at the block i (=$n \cdot \lambda$), which thus becomes a checkpoint, with a cached trie $\sigma_{n \cdot \lambda}$. The n+1^{th} epoch starts with an empty trie, σ_ϕ. A miner a_5 executes a transaction transferring from a_2 to a_4 to generate the block $(n \cdot \lambda) + 1$. None of a_2, a_4, or a_5 are in the current empty trie, so we find them from the cached trie $\sigma_{n \cdot \lambda}$, copy to the current trie $\sigma_{(n \cdot \lambda)+1}$, and update. For the next block $(n \cdot \lambda) + 2$, the sender of a transaction, a_4, can be found in the current trie $\sigma_{(n \cdot \lambda)+1}$, the miner a_1 can be found in the cached trie $\sigma_{n \cdot \lambda}$, but the receiver a_6 is neither of the two tires, so a_6 is created as a new account, leading to a new current trie $\sigma_{(n \cdot \lambda)+2}$. In this way, we can obtain a state trie consisting only of the accounts active during the $n + 1^{th}$ epoch, which will be cached at the next checkpoint $\sigma_{n \cdot (\lambda+1)}$.

4.2 Restoration of Dormant Account

In Figure 3, if an account neither in the current trie nor in the cached trie, say a_7, is a *dormant* account, it should first be reactivated before if it is a sender or a receiver of a transaction. There are two cases. If the account wants to be a sender, it should first transmit a *restore transaction*, which will create the account in the current trie in its last account state. If the account is a receiver of a transaction, there are also two cases. One is that the account is indeed a dormant account. The other is that the account is a new account because, in Ethereum, when someone sends some value to an arbitrary address that does not exist, a new account for that address is created in the state trie with a *nonce* of zero, regardless of the existence of the owner, as a_6 in Figure 3. In Ethanos, we simply create a new account even for the former case, which we call a *crumb* account. A crumb account can be a sender of another transaction if it has enough balance, but if not, it should also transmit a restore transaction, which will merge the crumb accounts (there can be multiple instances in multiple epochs) with the original account. In this way, Ethanos can reduce the frequency of the restore transactions.

4.2.1 Restore Transaction.

A restore transaction is initiated by the owner of a dormant account, but the transaction itself is created and transmitted by an archive node on behalf of the owner (Ethanos archive node works in the same way as Ethereum archive node, except that it deals with reduced tries; see Section 4.3). Then, miners and verifiers will run it as other regular transactions. So there are three roles involved.

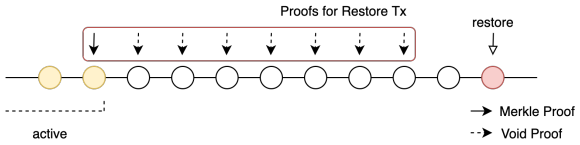
The owner should first determine if its account is dormant by checking if two epoch periods have passed since the last occurrence of its transaction or by inquiring if its account is accessible in the current epoch. Since the owner can easily bootstrap as a full node with Ethanos, it can determine the dormancy for itself. If the account is used once in a while, the owner may use a long-running smart contract to send a small amount of Ether to the account regularly.

An archive node who gets a request from the owner creates a restore transaction by accessing the state tries of related checkpoints and collecting the essential data, including:

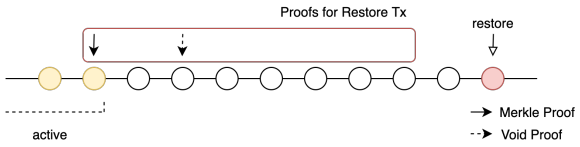
- A Merkle proof of the account at *last active checkpoint*
- Void proofs of the account at each checkpoint from the last active checkpoint up to the *latest checkpoint*-1

That is, a restore transaction should have a proof that the account was active at the last active checkpoint and was inactive thereafter. The Void proof for the latest checkpoint is unnecessary as we can check it thru a cached state trie.

Miners or verifiers will execute the restore transaction when it is in the transaction pool. They verify if the Merkle proof and the Void proofs are correct using only the state root in the checkpoint block headers, as explained in Section 2.5, and create the account in the current block's state trie.



(a) Restore transaction has a Merkle proof for last active checkpoint and Void proofs for checkpoints thereafter.



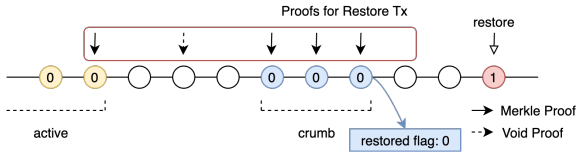
(b) Bloom filters can reduce the number of Void proofs.

Figure 4. Proofs and bloom filters for restore transaction.

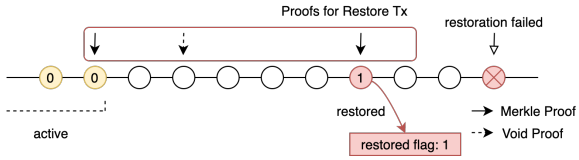
Suppose that an account *addr* wants to restore itself when its last active checkpoint number is k and the latest checkpoint number is n where $k < n$ (checkpoint number starts from 1). So, the state trie of the checkpoint k has this account ($\sigma_{k\cdot\lambda}[addr] \neq \phi$), whereas the state tries after the checkpoint k do not have the account ($\sigma_{>k\cdot\lambda}[addr] = \phi$). So the restore transaction includes a Merkle proof $P_M[k \cdot \lambda]_{addr}$ to prove that the account was valid in the block $k \cdot \lambda$, and Void proofs $P_V[(> k) \cdot \lambda]_{addr}$ to assure that the checkpoint k was the last active one. Figure 4(a) shows an example. These proofs will keep a malicious restore transaction from being verified during its execution. For example, if there is an attempt to restore an account state earlier than its last active checkpoint k (e.g., the earlier state has a higher balance), the verification will fail at k since the Void proof for k cannot pass.

One issue is that as $n - k$ increases, the number of Void proofs increases, making a large restore transaction. So we save a *bloom filter* in the block header of each checkpoint, a space-efficient probabilistic data structure to confirm the non-membership of an account [25]. A bloom filter always answers positive for a member, but it can also answer positive for a non-member (false-positive) with a low probability. On the other hand, if the bloom filter answers negative, the non-membership can be assured (i.e., there is no false-negative). Therefore, when the bloom filter of a checkpoint i answers negative (non-membership) for an account address *addr*, the restore transaction does not have to include the Void proof $P_V[i \cdot \lambda]_{addr}$, because the bloom filter guarantees the non-membership of the address. This can significantly reduce the number of Void proofs (e.g., Figure 4(b)). The verifier should confirm the non-membership using the bloom filter when executing the restore transaction, though, to prevent malicious restoration. Considering the space overhead of bloom filters, we currently set the probability of false-positives around 10%, thus obviating 90% of the Void proofs.

The reason that restore transaction is made by archive node instead of by Ethanos full node (e.g., the client node who has the account) is that Ethanos full node bootstrapped with a state trie of some checkpoint might not have enough data to make a restore transaction, if the the last active checkpoint of the account is older than the bootstrap checkpoint. If not, the Ethanos full node makes the restore transaction for itself.



(a) Restore transaction includes additional Merkle proofs for those checkpoints that have crumb accounts.



(b) An attempt to restore an already-restored account (flag=1) as a crumb account fails.

Figure 5. Restore crumb accounts with restored flag.

4.2.2 Restore Transaction with Crumb Accounts.

A crumb account is created in the current state trie as if a new account is created in Ethereum. One concern is that there is no way for a regular Ethanos full node to tell if an account is crumb or new since it does not have the state tries of all checkpoints. So, if we simply treat a crumb account as a new account and set the *nonce* as 0, an attacker might transmit a transaction already executed in the past, namely a *replay attack*, to receive a value again. To eliminate the possibility of such an attack, Ethanos sets the initial *nonce* of any account as *current block number (k)* times *average transactions per account per block*, a value which will always exceed any *nonce* value that has ever appeared previously [26].

If we need to restore an account that has crumb accounts, Ethanos should restore the account by merging the account in the last active checkpoint with its crumb accounts, possibly scattered across multiple checkpoints. So, the restore transaction should include a Merkle proof for each checkpoint that has a crumb account as well as for the last active check point. If multiple, *consecutive* checkpoints have the crumb accounts, the restore transaction should merge the balance of only the last checkpoint since it will have the final balance that integrates the account activity of those checkpoints. For example, Figure 5 (a) shows that there are three consecutive checkpoints, all of which have a crumb account, but only the balance of the last one will be merged with the original account of the last active checkpoint.

There is one issue for verification. For a restore transaction with a single Merkle proof, a malicious transaction that attempts to restore the account state in an old checkpoint

earlier than the last active checkpoint k cannot pass the Void proof for k, as discussed previously. A restore transaction with crumb accounts may allow two Merkle proofs P_{M1} and P_{M2} in a transaction, which would be interpreted as P_{M1} being the last active account and P_{M2} being as a crumb account. However, if P_{M1} is an old active account, while P_{M2} is a restored account of P_{M1}, thus being a new last active account, this interpretation is incorrect since it is attempting to add the balance of P_{M1} again. We solve this problem by adding an one-bit *restored flag* to the account state, whose default value is 0, yet is set to 1 by the verifier when an account is restored. Figure 5 depicts the flag bit for each checkpoint that has the account. So, if the flag for P_{M2} is 0 as in Figure 5 (a), meaning a legitimate crumb account, P_{M1} and P_{M2} can be merged and restored, with the flag for the restored account set to 1. If the flag for P_{M2} is 1 as in Figure 5 (b), the verifier will reject cheating already-restored account as a crumb account.

4.2.3 Security Property. The security of Ethanos can be stated as follows. First, if a restore transaction includes a false proof of an account (e.g., void proof for a checkpoint where the account exists), the verifier can detect and fail the restoration (e.g., the cheating attempt in Section 4.2.1).

Second, if a restore transaction includes only valid proofs, a Merkle proof for a checkpoint means one of three cases: a *restored account* if the restored flag at the checkpoint is 1, an *active account* if the restored flag is 0, and a *crumb account* if the restored flag is 0 and if there is a restored or active account ahead of it in the restore transaction. Then, there are only four checkpoint states for an account: active (a), restored (r), crumb (c), and void (v). Now, Ethanos has the following security property: "a restore transaction cannot generate an invalid state". A valid state generated by a restore transaction can be expressed using a regular expression $(a^+|r^+)v^+(c^+v)^*v^*$. If a restore transaction includes proofs that can generate a string of this regular expression, it can be executed by Ethanos; otherwise, it is rejected (we assume that if the last proof is a crumb state, there is an additional void state next to it, to make the regular expression simple without affecting the correctness). For example, *avvvcccvv* in Figure 6 (a) is valid, but *avvvvvrvv* in Figure 6 (b) is invalid because r follows v (*aav* is still valid although it include two consecutive a's, since Ethanos will correctly restore the account only from the second a). Since Ethanos rejects any string not belonging to the regular expression, an attacker has no choice but to follow the regular expression, but it only generates a correct state.

4.3 Synchronization

Ethanos has two synchronization modes. One is *archive sync*, the same as full sync in Ethereum. It downloads all blocks and replays/validates all transactions from the genesis block to the current block to reproduce the current state trie.

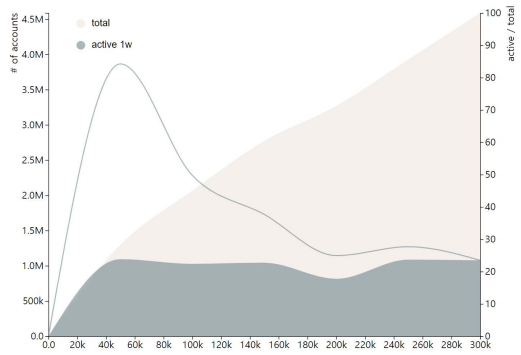


Figure 6. The number of active accounts (left axis) and its ratio (right axis) in our experiment with 300K blocks.

The other is *compact sync*, the same as fast sync in Ethereum. It downloads the state trie of the pivot block and replays only the transactions from the pivot block to the current block to reproduce the current state trie. Compact sync also downloads the state trie of the latest checkpoint to use it as cached state trie, and omits downloading old transactions from the genesis block to the pivot block unlike fast sync since they are not replayed anyway; the client can request old transactions to other archive nodes if needed [13]. Although compact sync should download two state tries, both tries include only the active accounts in the current and previous epochs, so they will be small (around 220MB in our experiment).

4.4 Fee for Restore Transaction

Bootstrapping with sharply-reduced state tries and drastically-fewer transactions will allow an ordinary client to function as a full node to verify transactions. It will also lower the space and time barrier to become a miner, increasing the number of miners. More miners and less security concerns would lead the miners to reduce the fee for the restore transaction, enough for the users willingly to pay. Actually, the users will pay a fee to restore their accounts if the accounts are more valuable than the fee. Archive nodes who maintain old state tries/transactions to create the restore transaction or provide old transaction on demand should also get the fee. Supply and demand dynamics will determine the number of archive nodes/miners and the fee, such that if the number of those nodes increases, the fee will decrease, and vice versa.

5 Evaluation

We evaluated Ethanos with 300,000 blocks of Ethereum from the 7.0 million block to the 7.3 million block, where 30,575,237 transactions execute updating 4,588,065 accounts for two months in early 2019. At the 7.0 million block, Ethereum blocks are packed with many transactions among many active accounts, enough to show a mature behavior as a popular account-based blockchain. Our experiment can be interpreted as starting an Ethanos blockchain with Ethereum's 7.0 million block as its genesis block, for the 300K blocks of

Data type	Geth	Ethanos	Diff
Headers	110.26	119.98	+9.72
Bodies	3,220.84	3,852.75	+631.91
Receipts	1,035.47	1,042.14	+6.67
Difficulties	6.78	6.73	-0.05
Block number -> hash	12.14	12.04	-0.1
Block hash -> number	11.73	11.73	0
Transaction Index	1,019.24	1,006.83	-12.41
Bloombit index	6.13	6.13	0
Trie nodes	46,333.43	40,342.62	-5,990.81
Trie preimages	275.66	275.66	0
Total	52,031.68	46,676.61	-5,355.07

Table 2. Storage size (MB) comparison of *archive sync* between *Geth* and *Ethanos*.

Ethereum transactions (for comparison with Ethereum, we also made Ethereum start from the 7.0 million block).

If we compute the number of one-week active accounts and its ratio during this period, as we did in Figure 2, there exist about 1 million active accounts with the ratio of 23.5% at the 300K block, as shown in Figure 6. Since there are many active accounts and transactions, our result would quickly exhibit a mature behavior of Ethanos although it is obtained only with first 300K blocks. On the other hand, since the real ratio of active accounts will be much smaller (e.g., less than 3% at the 7.0 million block of the original Ethereum as in Figure 2) and the real storage data of a few years will be much larger than that of two months, the real impact of Ethanos will be much higher than the ones reported here.

Running real transactions on the real Geth client to evaluate Ethanos is challenging, which is another contribution of this paper. For example, we must lower the mining difficulty to create blocks fast, but it is not simple to pack each block with exactly the same transactions in the same order as in Ethereum. So we control the miner using web3.js library to pause/resume mining. Also, the private key of each sender is unknown, so how to handle it is an issue. The experiment is time-consuming; it took two weeks to run 300K blocks. On the other hand, it is essential to evaluate Ethanos as realistically as possible since we can evaluate if sweeping dormant accounts is really beneficial, and how much the benefit is, only with real transactions running on real Geth code.

We performed the experiment in two steps. First, we pre-simulate transactions to classify the accounts and generate a *transaction list*, where we find points to add restore transactions. That is, starting with the account balance at the 7.0 million block, we simulate the account activities to see what accounts are saved at each checkpoint, what accounts are accessed at each epoch, and if an account accessed by a transaction is available in the previous checkpoint, thus being active in the current epoch, and if not, whether it should be a new account, a crumb account, or a restored account. For a restored account, we mark a position in the transaction list

	fast sync		compact sync	
Data type	Geth	Ethanos	Geth	Ethanos
Headers	105.04	114.19	105.04	114.19
Bodies	3,075.72	3,629.04	6.43	6.64
Receipts	986.78	992.77	5.44	5.45
Difficulties	5.91	5.92	5.91	5.92
Block number -> hash	10.78	10.77	10.78	10.77
Block hash -> number	11.04	11.04	11.04	11.04
Transaction Index	993.77	983.03	0.19	0.19
Bloombit index	0	0	0	0
Trie nodes	831.83	220.03	832.84	220.03
Trie preimages	0	0	0	0
Total	6,020.87	5,966.79	977.67	374.23

Table 3. Storage size (MB) comparison of *fast sync* and *compact sync* between *Geth* and *Ethanos* at the 7^{th} checkpoint.

where we will create a restore transaction (we cannot create it yet since no blocks are built, so cannot compute proofs).

Then, we run the modified Geth client to run transactions, create restore transactions, build state tries, mine blocks, and perform synchronizations, exactly as Ethereum does. Since we should start with an empty state trie (σ_ϕ) where the balance of any new account is set to zero, we make the value of each transaction be zero to avoid exceptions incurred by insufficient balances (in Ethereum we can start the genesis block by initializing *genesis.json* file with the initial balance of the accounts, but we found it fails if there are more than one million accounts). Although only zero values are transferred, our pre-simulation profiles every account and finds the correct position for restore transaction, so our experimental result would be almost the same as when running the real Ethereum transactions for Ethanos, especially for the the storage size and the synchronization time.

Other experimental constraints are as follows. We made all transactions as *delegated transactions*, transactions performed on behalf of the real accounts, because we cannot make a signature of each transaction. This increases each transaction size by 20 bytes. We treat contract account (CA) as externally owned account (EOA), such that a transaction involved with smart contract to send Ether is regarded as a regular transaction. An *internal transaction*, a transaction that a smart contract invokes, is not a regular transaction, thus being ignored. We will discuss how to restore CA later.

We implemented Ethanos based on Geth v1.9 client and compared with it for evaluation. We set the epoch size to 40,320 blocks ($\approx$ 1 week), which made seven sweeps (checkpoints) during the 300,000 blocks and added 198,121 restore transactions to the blockchain data. Also, the size of bloom filter at each checkpoint block is 512 KB. To reduce the impact of network fluctuation on synchronization time and the peer-drop failures, we connect the client who bootstraps and the host who provides the bootstrap data via Ethernet. The host has i7-9700K 3.60GHz CPU, 64GB RAM, 6TB SSD, and the client has i7-7700 3.6GHz CPU, 16GB RAM, 1TB SSD.

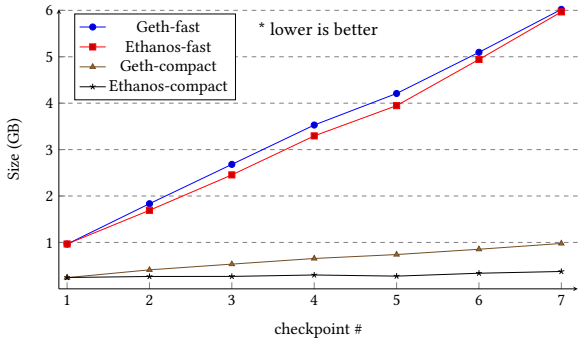


Figure 7. Data size of *fast sync* and *compact sync* for *Geth* and *Ethanos* at each checkpoint.

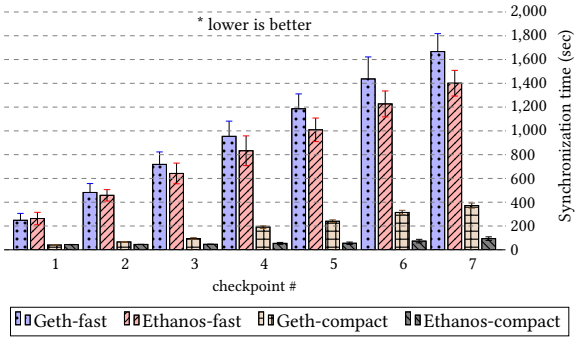


Figure 8. Synchronization time of *fast sync* and *compact sync* between *Geth* and *Ethanos* at each checkpoint.

5.1 Storage Size

5.1.1 Archive Sync. Geth archive sync downloads and replays all transactions from the genesis block to the current block, which reproduces the current state trie (MPT) as well as all intermediate trie nodes. Ethanos archive sync does the same, except that it empties the state trie when a new epoch starts, and executes the restore transactions additionally. Table 2 compares the total data size of archive sync for Geth and Ethanos from the genesis block to the 300K block, which are 52.0GB and 46.7GB, respectively. It also compares the size for each data type. This result means that Ethanos reduces the total blockchain size by 5.3GB (more than 10%) and the biggest contributor is the reduction of the *Trie nodes* sizes by almost 6GB. It should be noted that both Geth and Ethereum have the same number of accounts. They have the same number of updates for each account, which adds new trie nodes in the trie. The only difference is that Geth maintains a single huge trie composed of all accounts (4.5M accounts at 300K block in Figure 6), while Ethanos maintains a small trie at each epoch composed of only active accounts during the epoch (≈1.0M accounts). When an account is updated, the depth to the account in Ethanos is likely to be shorter than that in Geth (since Geth would require more branch nodes for a more detailed distinction of addresses), adding fewer new nodes. The difference of the trie node sizes will get bigger as the blockchain grows, since the number of all accounts increases sharply while the number of active accounts are bounded. Among other data types, bloom filters increased the size of block headers (*Headers*) by 10MB, and restore transactions increased transaction sizes (*Bodies*, *Receipts*) some, yet minor compared to the state trie size increase.

5.1.2 Fast and Compact Sync. Fast sync downloads all transactions but replays only the transactions from the pivot block to the current block, based on the state trie of the pivot block (also downloaded). Compact sync does the same except that it omits downloading the transactions occurred before the pivot block. We evaluated both syncs for Ethanos compared to Geth. For simpler comparison between Geth and Ethanos, we performed fast sync and compact sync at the $(checkpoint + 64)^{th}$ block so that the checkpoint becomes the pivot block. Then, the only difference between Geth and Ethanos is if the snapshot of the state trie of the pivot block is composed of all accounts up to the checkpoint (Geth) or only active accounts during the checkpoint's epoch (Ethanos).

Table 3 compares the storage size of Geth and Ethanos for fast sync and compact sync at the the 7^{th} checkpoint. For both syncs, Geth has a state trie size (*Trie nodes*) of 831MB, but Ethanos reduces it to 220MB. On the other hand, Ethanos increases the transaction size (*Bodies*, *Receipts*) and the block header size (*Headers*). Overall, the reduction of trie size by Ethanos has little impact on the total storage size of fast sync (6.0GB vs. 5.9GB), since the transaction size clearly dominates the trie size when only 300K blocks have passed from the genesis block. However, it has a much higher impact on the total storage size of compact sync (977MB vs. 374MB) after it cuts the transaction size. Again, it should be noted that this result is obtained only with the 300K blocks of two months. Figure 7 depicts the total storage size of fast sync and compact sync for Geth and Ethanos at each checkpoint.

The most important result is that the Ethanos state trie size to download for fast sync or compact sync remains around 220MB, even if the blockchain grows with its number of accounts and transactions being skyrocketed. Since the blocks mined in our experiments are already filled with the maximum number of transactions mined in today's Ethereum, and since only the active accounts during the one-week epoch comprises the state trie, the trie size is unlikely to far exceed this size, as long as the number of one-week active accounts remain around one million (which is true as of Jan 2020). For compact sync, even if we make a sync near the end of an epoch, thus downloading the last checkpoint's state trie of 220MB as well as the current pivot block's compact sync storage of 374 MB, the total storage for compact sync will be 594 MB, reasonable for a client to bootstrap as a full node. The almost-constant trend can also be found in Figure 7.

5.2 Synchronization Time

Figure 8 shows the synchronization time of the four cases at each checkpoint: *Geth-fast*, *Ethanos-fast*, *Geth-compact*, and

Ethanos-compact. We performed each experiment 15 times and obtain the average with the standard deviation, as shown in the graph. Since we connect the host and the client directly via Ethernet, we can measure the impact of downloading reduced state tries and reduced transactions more precisely, not just the impact of downloading less data, but downloading in a different way (i.e., traverse-transfer-rebuild for the state tries and batch-download for the transaction data).

Figure 8 shows that as we pass more checkpoints, Ethanos becomes tangibly faster than Geth in both fast sync and compact sync, similar to the data size trend in Figure 7. However, there is one thing to note. If we compare Geth-fast and Ethanos-fast at the epoch 7, the data size is almost the same, but the synchronization time of Ethanos is 17% shorter. Similarly, the data size of Ethanos-compact is half of Geth-compact, yet the synchronization time is 75% shorter. The difference of 290 seconds between Geth and Ethanos in both cases must be caused by the difference of the state trie size (around 610MB), which means that downloading the state trie works at 475 sec/GB. Similarly, if we compare Geth-fast and Geth-compact, or Ethanos-fast and Ethanos-compact, and divide the time difference by the data size difference excluding state trie size, we can see that downloading the transaction data (and other minor data) works at 234~257 sec/GB. This indicates that downloading state trie is almost twice slower than downloading other data, thus a bottleneck, so downsizing the state trie by Ethanos is beneficial. Again, this is a result only after 300K blocks, and if we synchronize after a few million blocks, downloading a huge state trie would be much slower, making an even larger benefit.

Another important result in Figure 8 is that the synchronization time of Ethanos-compact is tiny compared to others. Also, it increases quite slowly, similarly to the data size increase of Figure 7, enough to allow any client to quickly bootstrap as a full node.

5.3 Account Types at Each Checkpoint

We now examine the behavior of the Ethanos-based blockchain in detail. For each checkpoint, we classified the type of each account accessed during its epoch into four types: *cached*, *new*, *crumb*, and *restored*. When an account is used by a transaction for the first time in an epoch, it is *new* if it has never been used from the genesis block to the current block, *cached* if it is available in the state trie of the last checkpoint, *crumb* if it is not cached but used as a receiver, and *restored* if it is not cached but used as a sender with no enough balance, thus being preceded by a restore transaction. In Figure 9, all 1.2 million accounts at checkpoint 1 are new since the genesis block has an empty state (σ_ϕ). Among these accounts, 0.35 million accounts are cached at checkpoint 2 while the rest become dormant. There are no crumb or restore accounts at this checkpoint, because all accounts accessed in the first epoch are available in the state trie cached at checkpoint 1. At checkpoint 3, since the first cached state trie is replaced

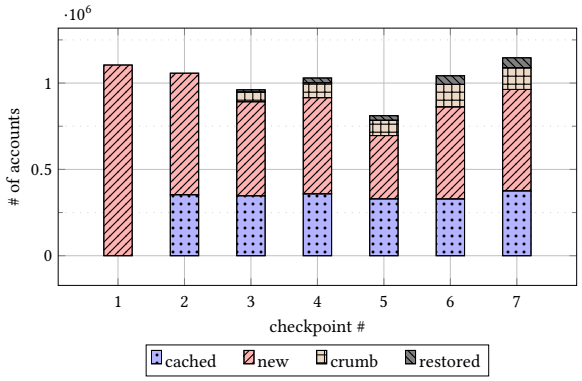


Figure 9. Number and type of accounts at each checkpoint.

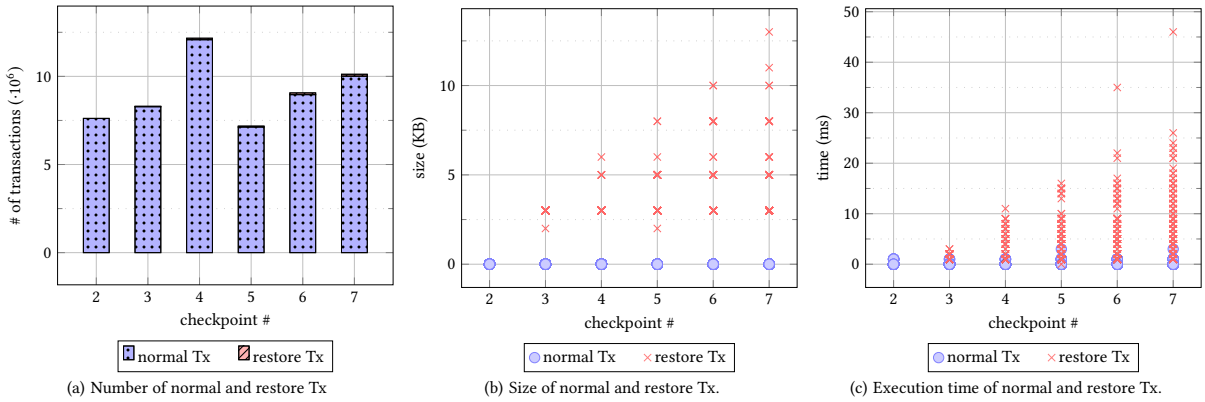
by the second state trie of checkpoint 2, crumb and restored accounts begin to appear. The graph shows that there are far fewer restored accounts than crumb accounts, meaning that many dormant accounts could be reactivated as crumb accounts without restoration. This makes the restored accounts take less than 5% of the total accounts. If users understand the operating mechanisms of Ethanos, the ratio of crumb and restored accounts will decline. The number of cached accounts remains almost constant in all checkpoints, implying that many accounts are reused across checkpoints.

5.4 Overhead of Restore Transactions

We measured the number, the size, and the execution time of restore transactions compared to normal transactions. Figure 10(a) depicts the number of restore transactions and normal transactions executed in each epoch. It indicates that restore transactions are fractionally executed, taking less than 1.5% of all transactions.

For the size and the execution time experiments, since the number of transactions differs among the epochs and the number of restore transactions is too small, we randomly sampled 200 normal transactions and 200 restore transactions for each epoch, and measured the size and the execution time. Figure 10(b) shows the size of normal and restore transactions, and the markers become darker in proportion to the number. The size of normal transactions is constant (0.12KB), but the size of a restore transaction is over 3KB and diverse since a restore transaction must include at least one Merkle proof ($\approx$ 3KB) and multiple Void proofs. More proofs would increase the size of a restore transaction, but we found that bloom filters reduced the size by more than half.

Figure 10(c) depicts the execution time of normal transaction and restore transaction. The execution time for a normal transaction is under one millisecond on average, whereas the time for a restore transaction is much longer because we must visit all checkpoints until we meet the last active checkpoint, searching the bloom filter and validating Merkle/Void proofs at each checkpoint. Since 98.4% of accounts are reused within 6 months as seen in Table 1, restoration with a huge



(a) Number of normal and restore Tx (b) Size of normal and restore Tx. (c) Execution time of normal and restore Tx.

Figure 10. Comparisons of normal and restore transaction (Tx).

overhead would be rare, and there are many ways to reduce it (e.g., smart contract for automatic reactivation service).

We estimated how restore transactions affect the blockchain performance using the data at the 7^{th} epoch. Here, a block executes 248 normal transactions and 2.7 restore transactions, on average, whose estimated execution time using Figure 10(c) is 99.2 and 10.5 milliseconds, respectively. So, restore transactions would increase the average execution time of a transaction by 10%. However, the block interval in Ethereum is 15 seconds, so the increase of the transaction time is trivial compared to the mining time.

5.5 Evaluation of Epoch Length

We evaluated the epoch length. A longer epoch is expected to produce larger state tries, increasing the bootstrapping overhead, but lower the frequency of the restore transactions, reducing the restoration overhead. For this experiment, we extended the trace to 864,000 blocks from previous 300,000 blocks, which include the period of May 2019 when Ethereum was more active due to its price rise. We evaluated four epoch lengths: three days, one week, two weeks, and one month. They made 50, 20, 10, and 5 epochs, respectively, during the 864,000 blocks. We performed the experiment by running the pre-simulation to obtain the number/distribution of active accounts in each epoch as well as the number of restore transactions executed in each epoch (pre-simulation is enough for this evaluation since running the Geth client would take too long, yet producing no more meaningful result).

Figure 11 shows the number of active accounts at each checkpoint for the four epoch lengths (we saw the data for a weekly epoch during 300,000 blocks in Figure 9). For every epoch length, the number of active accounts moves up and down depending on the activity of Ethereum, and reaches a peak around the 7.8 million block in May 2019 by attracting many investors. We found that when the epoch length increases, the number of active accounts in each epoch also increases, but less proportionally (e.g., the number of active accounts of one-month epoch is around 6 times more than that of three-day epoch, not 10 times). Actually, for the last epoch when the number of active accounts is peak, the state trie size of three-day epoch was 138MB, and it was 278MB (one week), 454MB (two weeks), and 817MB (one month), increasing just 86%, 70%, and 60% compared to the increase ratio of the epoch length. This is due to the transaction workload highly skewed towards recent accounts, which tends to reuse active accounts rather than creating new ones, especially in longer epoch lengths. Since the state trie dominates the downloading space/time for compact sync, these sizes will represent the bootstrapping overhead of epoch lengths.

Figure 12 shows the number of restore transactions at each epoch. It also depicts the cumulative number for five intervals (e.g., the bar at checkpoint 50 for the three-day epoch means the cumulative number between checkpoint 40 and 50), so the sum of the five bars is the total number of restore transactions during the 864,000 blocks. As expected, larger state tries of longer epoch lengths obviate restore transactions that would otherwise be required, decreasing the total number, yet the decrement is also not proportional (e.g., the number of restore transactions of one-month epoch is around 1/5 of that of three-day epoch, not 1/10). The accounts restored in longer epochs are also restored in shorter epochs, which appears to reduce the decrease ratio. The drop from two-week to one-month is notably more pronounced than others.

To decide an appropriate epoch length we need to consider both the bootstrapping overhead and the restoration overhead, especially the higher restoration overhead of smart contracts; see Section 6.1. One-month epoch seems cost-effective due to its relatively smaller increase of the state trie size and larger decrease of the restoration frequency. For mobile clients, however, bootstrapping with 817MB is rather heavy, so one-week epoch seems appropriate until the network speed and the storage size increase in the future (two-week epoch is not efficient since it does not help much in reducing the restoration frequency). We may employ a *variable* epoch length adaptive to the trie size such that we make a checkpoint when the current trie size reaches some threshold, rather than when a fixed epoch length elapses.

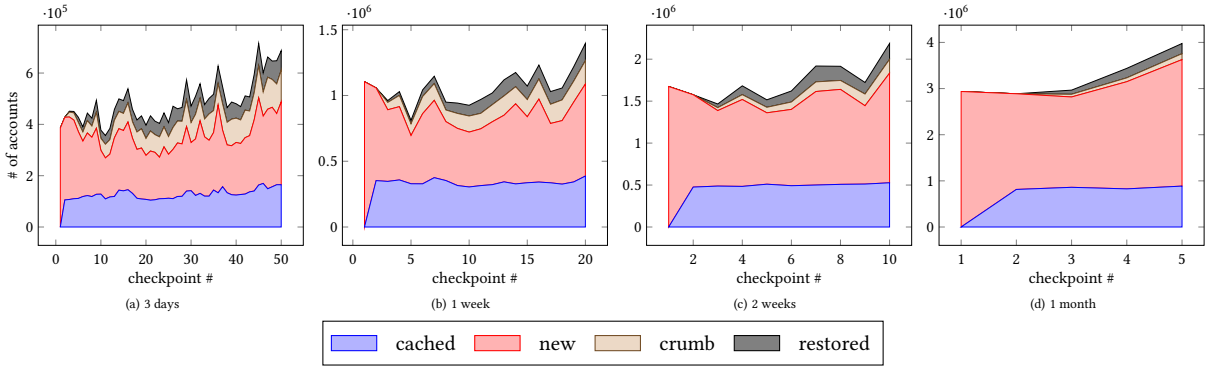


Figure 11. Number of active accounts and their types at each checkpoint for four epoch lengths.

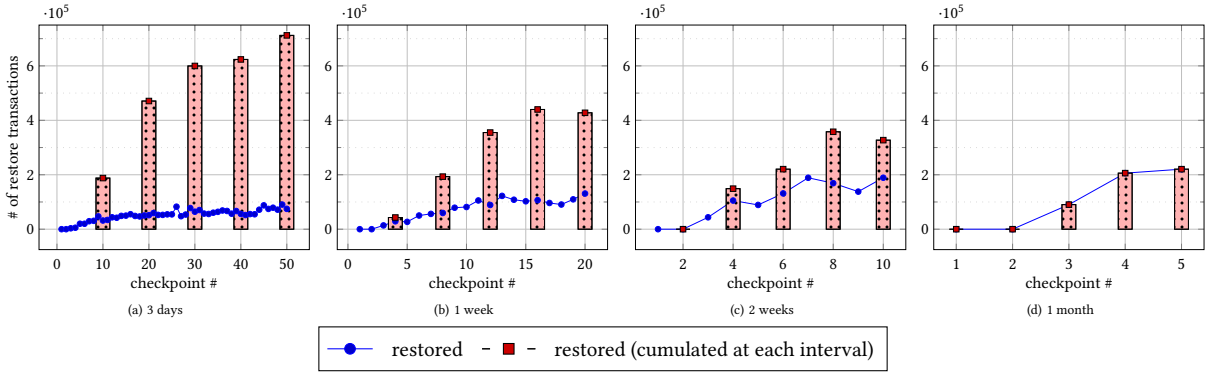


Figure 12. Number of restore transactions and their cumulative number at each checkpoint for four epoch lengths.

6 Discussions

This section discusses two additional issues of Ethanos, restoration of smart contracts and downloading of block headers.

6.1 Restoration of Smart Contract

Although we did not deal with a smart contract account (CA) in this paper, there is nothing special about restoring CA except that its restore transaction might be larger, since it must include a snapshot of the *program state* of the CA at the last active checkpoint. The largest space overhead of the snapshot is mainly due to the ownership record of assets such as *token* based on the ERC-20 protocol. If each record takes 52 bytes (20 for account address + 32 for balance), one million users would make 52MB, so it is large. There are three issues to consider for the restoration of CA.

First, unlike regular EOA accounts, the state of a smart contract in active service is updated by many users continuously, or by the owner of the CA, thus unlikely to be dormant (e.g., many popular smart contracts like CryptoKitties [27], MakerDao [28], Compound [29], or Yearn [30], monitor and call themselves regularly). On the other hand, a smart contract once being out of service would be out of service forever, so a dormant smart contract would never be restored again. Ethanos can sweep most of these dormant, heavy smart contracts (many of them are for testing, yet not *self-destructed*), making the state trie much lighter.

Second, even when a dormant CA were reactivated, there cannot be a crumb that has a program state, so we do not have to consider merging the program state of the original CA and those of the crumb CAs. That is, the address of a CA is calculated by hashing the RLP-encoded array of deployer's address and its nonce value (i.e., transaction count): $Address = keccak256(rlp.encode([deployer, nonce]))[12:]$ ($keccak256$ is the Ethereum hashing function). Since the nonce value increases monotonically, it is impossible to deploy a new crumb CA with the same address. So, a crumb CA can be created only by a value (Ether) transferred to the address as an EOA, meaning that a crumb CA cannot have its own program state, but only the balance. Therefore, we can restore a CA by restoring the snapshot of the last program state of the original CA, and then by merging the balances of the CA and the crumb CAs, as we did with EOA.

Finally, we discuss how to serialize the program state to a snapshot and restore it. The program state to be saved in the snapshot includes only the *state variables*, and they are stored in the database with an Merkle Patricia Trie (MPT) structure called the *storage trie*, similar to the state trie. The address is composed of the position for a singleton variable or a hash value computed with the key and the position ($keccak256(key \cdot position)$) for dynamically-sized arrays or *mapping*s, and the data is located at the leaf of the MPT. Since the position of each variable is statically decided and

inferable from the code, the MPT is deterministic. A restore transaction for a CA should include the snapshot of the storage trie for its restoration. However, except for top three CAs of Ethereum, there are fewer than 0.8 million users [31], so it would not be impractical to save the MPT. To reduce the space overhead, we may apply the Ethanos idea recursively to the storage trie. That is, if a token owner wants to restore a dormant CA to run a token transaction, it can restore only its token record from the mapping variable (and other involved token records and singleton variables). Since the owner knows the keys (account addresses) while the variable positions are fixed, the restore transaction can save only the data extracted by a Geth API *getStorageAt()* without any storage trie nodes, reducing the space overhead. We need a more elaborate protocol, which will be discussed elsewhere.

6.2 Reducing Download Size of Block headers

After Ethanos-compact sharply reduces the state trie size and the number transactions to download, the *block headers* can be a bottleneck. For example, the total storage needed by fast sync in Jan 2020 is around 235GB, where the size of block headers is around 3GB, so bootstrapping with 3GB is still burdensome. We can exploit the idea of *NIPoPoW* [14], which reduces their download size without affecting the security. It downloads only a sample of block headers of *super blocks*, blocks with a higher mining difficulty, and chain them together using the hash of the previous super block. These block headers are much harder to forfeit than regular ones, so verifying only these block headers at synchronization would have a similar effect of verifying all block headers.

Now, remember that Ethanos requires the block headers of the checkpoints to confirm Merkle/Void proofs and save the bloom filters. Instead of choosing a checkpoint at the exact, one-week epoch boundary of around 40,320 blocks, we propose choosing a checkpoint *near* the boundary, at a super block. Fortunately, super blocks are known to appear periodically [32, 33], so we can choose a checkpoint in around one-week period. Then, we can download only 226 block headers if we synchronize in Jan 2020 when Ethereum has 9.1M blocks (9.1M / 40,320), and their size is 84KB (226 * 370 bytes). This means that the total download data for synchronization can be below 1GB (Ethanos-compact download size of around 600MB plus the NIPoPoW download size of 0.92MB accommodating 100M blocks).

7 Related Work

There have been a few *Ethereum improvement proposals (EIPs)*, aimed to reduce the state trie size by eliminating empty accounts or accounts with negligible balances, such as EIP-158 [34], EIP-161 [35], and EIP-168 [36]. EIP-158 defined an *empty account* whose *nonce* and *balance* are zero with empty *codeHash* and *storageRoot*, and allows miners to delete them to save storage. EIP-161 in the *spurious dragon* hard fork [37] covers several edge cases of EIP-158, but they occur sporadically, so the impact is minimal. EIP-168 proposed a more aggressive optimization by removing *dust accounts* whose balance is less than the transaction fee, but this proposal is still under discussion because of the security concerns.

Vault takes a similar approach to reduce the bootstrapping time and storage [38] for *Algorand* blockchain [6]. Vault observed that 38% of EOA on Ethereum has no balance, but Ethereum cannot remove them due to the risk of the replay attack. Vault introduced an expiration date to transactions so that it can safely remove those accounts whose balance is zero and whose last transaction has expired. The experimental results show that the state size could be reduced to 3.1GB (one shard) when the state size of Ethereum is 5GB, when making a synchronization after executing 500 million transactions. Unlike EIPs or Vault, Ethanos removes dormant accounts occupying more than 95% of the accounts, which significantly reduces storage size and bootstrapping time, and restores them if needed. Furthermore, Ethanos is a generalized solution , while Vault is only applicable to Algorand.

Utreexo [39] reduces Bitcoin's *UTXO set*, analogous to Ethereum's state trie. It represents a UTXO set with a hash-based accumulator similar to a Merkle tree, and saves the root hash in the block. This allows a node to verify a transaction without a full UTXO set, given a proof that a UTXO is in the accumulator. Utreexo caches the accumulator data of recent UTXOs in a node, reducing the proof size. Compared to Ethanos, reducing UTXO set with recent UTXOs is similar to reducing state trie with active accounts, so is spending old UXTO with a proof to restoring dormant accounts.

8 Summary

State-based blockchain suffers from data explosion. We proposed Ethanos, which periodically sweep dormant accounts, rebuild the state trie only with active accounts, and restore dormant accounts when needed. Ethanos can reduce the state trie size to about 220MB, so when combined with optimizations that omit old transactions and less-important block headers, it is plausible to bootstrap with less than 1GB, within a few hundred seconds. This not only increases the number of full nodes/miners, thus improving blockchain security, but allows occasional on-line users to be a full node whenever transmitting transactions, thus improving user security. Ethanos is applicable to any state-based blockchains (even Ethereum 2.0), especially those aiming high transactions-per-second, which suffer more from state explosion.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2020R1A2B5B02001845). The ICT at Seoul National University provided research facilities for this study.

References

[1] Vitalik Buterin et al. Ethereum white paper. *GitHub repository*, 1:22–23, 2013.

[2] Satoshi Nakamoto et al. Bitcoin: A peer-to-peer electronic cash system. 2008.

[3] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual International Cryptology Conference*, pages 357–388. Springer, 2017.

[4] Gavin Wood. Polkadot: Vision for a heterogeneous multi-chain framework. *White Paper*, 2016.

[5] A Yakovenko. Solana: A new architecture for a high performance blockchain, 2018.

[6] Yossi Gilad et al. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68. ACM, 2017.

[7] Etherscan. Ethereum full node sync archive chart. https://etherscan.io/chartsync/chainarchive, cited Oct 2020.

[8] EthHub. Running an ethereum node. https://docs.ethhub.io/using-ethereum/running-an-ethereum-node/, cited Oct 2020.

[9] Consensys. Infura. https://infura.io/, cited Oct 2020.

[10] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[11] Ethereum Foundation. Go-ethereum. https://geth.ethereum.org/, cited Oct 2020.

[12] Vaibhav Saini. Getting deep into geth: Why syncing ethereum node is slow. https://hackernoon.com/getting-deep-into-geth-why-syncing-ethereum-node-is-slow-1edb04f9dc5, 2018.

[13] Vitalik Buterin. Personal communication. We discussed whether all nodes bootstrapped with fast sync must have all transactions, and concluded no if archive nodes can provide them on demand, 2019.

[14] Aggelos Kiayias et al. Non-interactive proofs of proof-of-work. In *International Conference on Financial Cryptography and Data Security*, pages 505–522. Springer, 2020.

[15] Alan Demers et al. Epidemic algorithms for replicated database maintenance. *ACM SIGOPS Operating Systems Review*, 22(1):8–32, 1988.

[16] Ethereum. Ethash. https://github.com/ethereum/wiki/wiki/Ethash, cited Oct 2020.

[17] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *Annual International Cryptology Conference*, pages 139–147. Springer, 1992.

[18] Mustafa Al-Bassam et al. Fraud proofs: Maximising light client security and scaling blockchains with dishonest majorities. *arXiv preprint arXiv:1809.09044*, 2018.

[19] Viktor Leis et al. The adaptive radix tree: Artful indexing for main-memory databases. In *IEEE 29th International Conference on Data Engineering*, volume 13, pages 38–49, 2013.

[20] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*, pages 369–378. Springer, 1987.

[21] Sanjay Ghemawat and Jeff Dean. leveldb. https://github.com/google/leveldb, cited Oct 2020.

[22] Xiaoyao Qian. *Improved authenticated data structures for blockchain synchronization.* Master's thesis, University of Illinois at Urbana-Champaign, 2018.

[23] Péter Szilágyi. Fast synchronization algorithm. https://github.com/ethereum/go-ethereum/pull/1889, cited Oct 2020.

[24] Péter Szilágyi. State Trie. https://twitter.com/peter_szilagyi/status/1166642710763266050. He estimated the state trie size 54GB for 359M trie nodes on Aug 2019 when the fast sync size was 172GB. On Jan 2020 when the fast sync size was 235GB, there were 475M state trie nodes. Our measurement of average node size is around 150 bytes (same as his size above), so we estimated the state trie size 71GB.

[25] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[26] Gavin Wood. Eip-169: Dust account replay security. https://github.com/ethereum/EIPs/issues/169, 2016.

[27] Dapper Labs. Cryptokitties. https://www.cryptokitties.co, cited Oct 2020.

[28] Maker Foundation. Makerdao. https://makerdao.com, cited Oct 2020.

[29] Compound Labs. Compound. https://compound.finance, cited Oct 2020.

[30] Andre Cronje. Yearn finance. https://yearn.finance, cited Oct 2020.

[31] Etherscan. The number of holders in each erc-20 token. https://etherscan.io/tokens?sort=holders&order=desc, cited Oct 2020.

[32] Andrew Miller. The high-value-hash highway. https://bitcointalk.org/index.php?topic=98986.0, 2012.

[33] Andrew Miller. Re: The high-value-hash highway. https://bitcointalk.org/index.php?topic=98986.msg1109747#msg1109747, 2012.

[34] Vitalik Buterin. Eip-158: State clearing. https://eips.ethereum.org/EIPS/eip-158, 2016.

[35] Gavin Wood. Eip-161: State trie clearing(invariant-preserving alternative). https://eips.ethereum.org/EIPS/eip-161, 2016.

[36] Gavin Wood. Eip-168: Kill dust accounts. https://github.com/ethereum/EIPs/issues/168, 2016.

[37] Alex Beregszaszi. Eip-607: Spurious dragon. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-607.md, 2017.

[38] Derek Leung et al. Vault: Fast bootstrapping for the algorand cryptocurrency. In *In 26th Annual Network and Distributed System Security Symposium*, 2019.

[39] Thaddeus Dryja. Utreexo: A dynamic hash-based accumulator optimized for the bitcoin utxo set. *The International Association for Cryptologic Research. ePrint Archive*, 2019:611, 2019.