SlimArchive: A Lightweight Architecture for Ethereum Archive Nodes

Hang Feng, Yufeng Hu, and Yinghan Kou, *Zhejiang University;*
Runhuai Li and Jianfeng Zhu, *BlockSec;* Lei Wu and Yajin Zhou, *Zhejiang University*

https://www.usenix.org/conference/atc24/presentation/feng-hang

This paper is included in the Proceedings of the
2024 USENIX Annual Technical Conference.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-41-0

Open access to the Proceedings of the
2024 USENIX Annual Technical Conference
is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

SlimArchive: A Lightweight Architecture for Ethereum Archive Nodes

Hang Feng
Zhejiang University
h_feng@zju.edu.cn

Yufeng Hu
Zhejiang University
yufenghu@zju.edu.cn

Yinghan Kou
Zhejiang University
yinghan_kou@zju.edu.cn

Runhuai Li
BlockSec
rhli@blocksec.com

Jianfeng Zhu
BlockSec
jfzhu@blocksec.com

Lei Wu
Zhejiang University
lei_wu@zju.edu.cn

Yajin Zhou*
Zhejiang University
yajin_zhou@zju.edu.cn

Abstract

With the rapid development of Ethereum, archive nodes that record all historical states have become a critical component of the infrastructure. However, current archive nodes suffer enormous storage requirements and poor performance due to the inefficient authenticated Merkle Patricia Trie and coarse-grained state granularity.

This paper presents a lightweight and high-performance architecture for Ethereum archive nodes to address the two limitations mentioned earlier. The core idea of our approach is to maintain compacted, flattened, and fine-grained (i.e., transaction-level) historical states by flattening the minimum state changes of each transaction required for the world state. Our method maintains an archive node with minimum storage requirements while providing high-performance state access. We have implemented a prototype system named SLIMARCHIVE for Ethereum. The evaluation results demonstrate that our approach reduces storage requirements by 98.1%, improves state access throughput by 19.0×, and speeds up transaction execution by an average of 1112.5×, compared to vanilla Geth.

1 Introduction

Since its inception in 2015, Ethereum has emerged as one of the most prominent public blockchains. Its capability to execute smart contracts has considerably facilitated the development of Decentralized Applications (DApps), including Cryptocurrency [21], Decentralized Finance (DeFi) [53], Non-Fungible Token (NFT) [26], and more.

Ethereum functions as a transaction-driven finite state machine [54]. Each time a block is mined, Ethereum nodes use the Ethereum Virtual Machine (EVM) to execute transactions in the block and advance the world state, i.e., the granularity of the state transition is block-level. Ethereum nodes are categorized as non-archive (light or full) and archive nodes [4]. Non-archive nodes prune historical states as the blockchain progresses, whereas archive nodes persist the world state snapshot at each block, thereby maintaining all historical states.

Archive nodes, which maintain billions of historical transactions and their resulting states, have become a vital infrastructure component in both academia and industry. They can be used in multiple scenarios, e.g., smart contract and transaction testing. For example, archive nodes offer multiple JSON-RPC APIs [5, 12, 13] for analyzing historical transactions and states. Security researchers can analyze the execution traces of historical transactions, which depend on historical states, to identify suspicious behaviors and attack patterns [35, 44, 55, 64]. Developers can employ fuzz testing on smart contracts under historical states to uncover vulnerabilities [59]. Arbitrage researchers can enhance their trading strategies by back-testing them through simulations of transactions based on historical states [39, 51]. Besides, historical states enable temporal analysis [65], such as analyzing the historical balance changes of a particular account. Additionally, certain Extract-Transform-Load (ETL) works [19–21, 34, 55, 63, 66] collect and analyze historical data on Ethereum to gain insightful knowledge.

Unfortunately, current archive nodes suffer enormous storage requirements and poor performance [31, 40, 47, 52, 56]. The storage consumption of an archive node implemented by Go-Ethereum (Geth, [11]), the official and most popular Ethereum execution client, has reached over 16 TB. The synchronization of an archive node also takes up a considerable amount of time (Table 2 in Section 5.1). Additionally, the performance of state access is extremely low [29, 55]. Several works [23, 32, 33] point out that state access consumes the majority of the transaction execution time.

These problems arise from the inefficient authenticated Merkle Patricia Trie (MPT) and coarse-grained state granularity (Section 3). First, Ethereum employs the authenticated MPT to maintain a world state that enables untrusted nodes to generate Merkle proofs to convince users of query results [33]. However, MPTs also introduce excessive intermediate data (non-leaf nodes) that consume extra storage. Besides, since data are stored in leaf nodes, each state access requires searching the MPT from root to leaf, i.e., MPTs suffer

*Corresponding author

from read/write amplification. Furthermore, the granularity of historical states is a block. However, this coarse-grained block-level granularity introduces much computation overhead when accessing the state at a transaction with a high position in the block.

Key Insights. While most works [23, 33, 43, 50, 57, 61, 62] focus on optimizing the authenticated structure of blockchain data, we observe that the authentication (Merkle proofs) of historical states is not required in most real-world usage scenarios (Section 3.1). Therefore, under these circumstances, we can replace the complicated MPT with a compacted and flattened data model for historical states to reduce intermediate data and simplify state access. Moreover, we observe that the Ethereum state transition is composed of the sequential execution of transactions by the EVM, i.e., the state transition granularity is a transaction at a lower layer (Section 3.2). This mechanism allows refining the granularity of historical states to the transaction level, thus avoiding overhead introduced by the coarse-grained state granularity.

Our Method. Consequently, we propose SLIMARCHIVE, a lightweight and high-performance architecture for Ethereum archive nodes (Section 4). Based on the basic principle that state changes alone are sufficient to recover the state at any particular time point, the core idea of SLIMARCHIVE is to maintain *compacted*, *flattened*, and *fine-grained* historical states by *flattening the minimum state changes of each transaction required for the world state*. Our approach maintains historical states with minimum storage while providing high performance. Specifically, the high-level system design includes the following three perspectives:

- **Minimizing recorded data.** The EVM produces massive data during transaction execution, but not all are required for an archive node. To minimize unnecessary data storage, we scrutinize the operations that mutate the Ethereum world state and collect the minimum changed states required for maintaining historical states. Specifically, we ignore the authentication data and EVM's runtime states (e.g., stack and memory). For each transaction, we only preserve the post-state of account and storage (see Section 2.2) changes.
- **Extracting transaction-level state changes.** Since Ethereum performs block-level state transition, we cannot directly obtain transactions' state changes. To address this challenge, we first hook each transaction execution. Except for transaction execution, some consensus procedures [28] also mutate the world state. To ensure completeness, we formalize consensus procedures as virtual transactions. We instrument the EVM and collect (virtual) transactions' state changes. Ultimately, we break down a block-level state transition into a series of transaction-level state changes.
- **Flattening state changes to benefit from the key-value store.** As state changes are multidimensional, they cannot be stored directly in a key-value store. To address this issue, we propose a flattened data structure called *state-temporal archive*. Each key-value pair contains the minimum but complete data of one state change. Our novel data structure effectively minimizes the intermediate data. Additionally, the transaction-level state changes are arranged chronologically. We convert querying the historical state to seeking the last state change that occurred before the specified time, which is ultra-efficient in key-value stores.

Trade-offs and Practical Significance. SLIMARCHIVE trades off data authentication for better performance and improved cost-effectiveness (Section 6). Unlike standard Ethereum nodes that provide cryptographic proofs for data, SLIMARCHIVE necessitates users' trust in its responses. Although this approach has potential limitations, it offers an efficient alternative for most real-world scenarios where authentication for historical states is unnecessary, or where archive nodes are considered trustworthy. Furthermore, it contributes to the diversity of the Ethereum client ecosystem, providing users with an additional option for utilizing archive nodes.

We have implemented a prototype based on Geth for Ethereum. We evaluate SLIMARCHIVE from various perspectives (Section 5). Compared to the baseline (Geth), SLIMARCHIVE reduces storage requirements by 98.1%, improves state access throughput by 19.0×, and speeds up transaction execution by an average of 1112.5×. We also conduct additional experiments showing that SLIMARCHIVE consumes fewer resources (memory and disk) to access historical states. The evaluation results demonstrate that our approach efficiently reduces storage requirements and improves performance compared to state-of-the-art solutions. Finally, it is worth noting that our idea could also be applied to other EVM-compatible blockchains.

Contributions. In summary, we make the following main contributions in this paper.

- **Systematic Investigation.** We thoroughly investigate current Ethereum archive node implementations and empirically study their limitations. Our observations motivate the design of our approach.
- **New Architecture.** We propose SLIMARCHIVE, a lightweight and high-performance architecture for Ethereum. To the best of our knowledge, SLIMARCHIVE is the first implementation of archive nodes that maintain transaction-level historical states.
- **Efficient Data Structure.** We introduce a novel flattened data structure called *state-temporal archive* that benefits from the key-value store for historical states. It minimizes storage requirements and accelerates state access.
- **Comprehensive Evaluation.** We evaluate SLIMARCHIVE from various perspectives. The results demonstrate that our approach achieves significant resource savings and performance improvements.

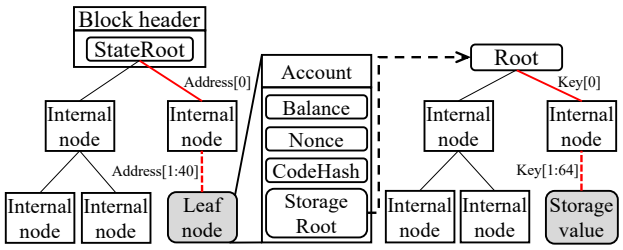


Figure 1: The Ethereum World State Tree.

2 Background

2.1 Ethereum Blockchain

Ethereum represents a chain of blocks. A block contains an ordered sequence of transactions sent by users. Each time a new block is constructed by the miner and broadcasted to the network, it will be processed and validated by all the nodes in the P2P network. An Ethereum node is made up of two layers: the consensus layer and the execution layer [7, 8].

Consensus Layer. The consensus layer handles the mining and syncing progresses of the blockchain. It determines the valid miner and synchronizes blocks from peers to ensure the blockchain's growth and consistency among all peers.

Execution Layer. The execution layer is responsible for blockchain state transition. It handles block processing and world state management. The block processing is the sequential execution of transactions. A transaction is instructions signed using a user's private key. The execution of transactions will transform the Ethereum world state. Transactions are executed in the EVM [54]. Except for the world state, the EVM maintains four runtime data regions: stack, memory, call data, and return data. Stack and memory are used for intermediate data. Call data is the input payload of the transaction. Return data is the output of the execution result. Note that these data regions are volatile and will be cleared after the execution of each transaction. This paper focuses on state management optimization for the execution layer.

2.2 Ethereum State

Account-based Model. Ethereum maintains an account-based world state. There are two types of accounts in Ethereum: Externally-Owned Accounts (EOAs) and contract accounts. EOAs are controlled by users' private keys. Contract accounts are controlled by their smart contract bytecode. Each account is identified by a unique address. An account consists of four properties: `Balance`, `Nonce`, `CodeHash`, `StorageRoot`. `Balance` represents the amount of asset (Ether) owned in the account. `Nonce` refers to the number of transactions issued by the account so far. `CodeHash` and `StorageRoot` are only used for contract accounts. `CodeHash` represents the hash of the contract bytecode. A contract account also has private and persistent storage, a mapping between 256-bit bytes. Thus, each contract account has up to 2^{256} storage slots. Each contract's storage is encoded as an MPT. `StorageRoot` refers to the root of the storage tree.

Merkle Patricia Trie. Ethereum uses the MPT [14, 54] (a variant of the Merkle tree) to maintain the world state. The MPT is a radix tree (trie) that provides cryptographic verification. Data are stored in leaf nodes. Hash-pointers link parents and children nodes. The MPT allows efficient proof generation and verification. Ethereum nodes can provide the leaf node's Merkle proof, which includes all nodes on the path from the leaf node to the root node, along with their siblings. Users can compute the corresponding state root using the Merkle proof and compare it with the state root recorded locally, thereby verifying the integrity of the leaf node data.

Ethereum World State Tree. As Figure 1 depicts, Ethereum maintains two types of MPTs: the state trie (left part) and the storage trie (right part). Each block maintains the `StateRoot` of the state trie that records the account states; Each contract account maintains the `StorageRoot` of the storage trie that records the contract's storage states. Accounts and storage are stored in the leaf nodes (the gray rects in Figure 1). The MPT is a 16-radix trie. The path is a sequence of hex digits representing the address or slot key. To save disk space, The MPT merges a node with its child if the node has only one child. Thus, the MPT in reality has a lower depth than a full MPT. In Ethereum, state access requires searching the MPT from root to leaf (the red line in Figure 1).

Each time a block is processed, the dirty states are flushed to the leaf nodes, and the MPT updates from dirty leaf nodes to the root to construct a new MPT. Since the state transition is performed at each block, the granularity of historical states is at the block level. There are two modes according to the garbage collection (GC) policy for historical states in Ethereum: *archive mode* and *non-archive mode*. Non-archive mode only preserves recent (latest 128 blocks) MPTs, while historical MPTs are pruned. Archive mode persists the MPT at each block. Most blockchain systems use key-value stores for persistent data. In Ethereum MPTs, the key is the node hash; the value is the serialized node data.

3 Limitations and Solutions

In this section, we investigate Geth, the standard and most widely used Ethereum execution client. We first synchronize an archive node with the initial 14M blocks and a full node at block height 18M (August 2023). Then, we identify and empirically study two limitations, i.e., inefficient authenticated MPT and coarse-grained state granularity, that contribute to the storage and performance problems. Finally, we propose our solutions based on our insights.

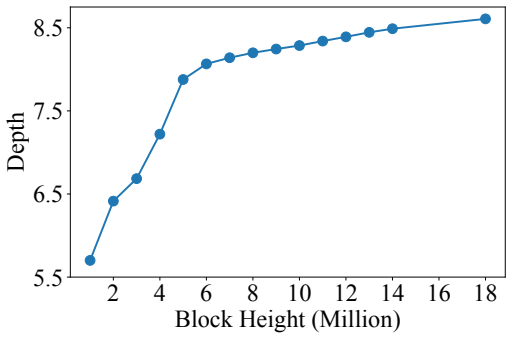


Figure 2: The average leaf node depth of MPTs at different block heights.

3.1 Inefficient Authenticated MPT

Current Ethereum nodes employ MPTs to maintain historical states. This design originates from the data authentication requirements [33], i.e., the nodes could provide Merkle proofs of query results to convince users. However, the authenticated MPT suffers inherited *excessive intermediate data* reducing effective storage utilization, and *read/write amplification* slowing state access.

Excessive Intermediate Data. Recall the MPT described in Section 2.2. Only leaf nodes store states, while the intermediate data (internal nodes) that serve as proofs do not contribute to the world state. We analyze the state trie at block 18M. The MPT has 217.9M leaf nodes, but the count of intermediate nodes reaches 293.8M. The states with a size of 14.9 GB produce an MPT with a total size of 40.6 GB. In other words, The storage consumed by the intermediate data is 1.72 times the size of the world state. *The storage utilization (state size / MPT size) is only* 36.7%.

Read/Write Amplification. Since states are stored in leaf nodes, state access requires searching the MPT from the root to the leaf node. Thus, each state access is amplified to d database operations, where d in the order of $O(\log n)$ is the depth of the leaf node. To empirically study how the MPT amplifies state access, we measure the average depth of leaf nodes of current Ethereum state tries at different block heights. Figure 2 demonstrates that the depth increases as Ethereum progresses. As of block 18M, the average leaf node depth reaches 8.6, which implies *each state access will be amplified to an average of* 8.6 *database operations*.

Our Finding. In summary, data authentication introduces significant storage overhead and performance loss in archive nodes. We argue that this authentication mechanism is unnecessary across many scenarios, particularly for archive nodes. As discussed in Section 1, historical states are primarily used for testing and analytical purposes, where performance is critical. However, generating and verifying Merkle proofs carries a high price (e.g., network overhead and multiple hash operations) for nodes and users. Thus, in many cases, users will not ask archive nodes to provide Merkle proofs of historical states. For example, a researcher conducting large-scale historical transaction profiling would likely deploy a local archive node for enhanced performance rather than authentication. Infura, one of the largest blockchain service providers in the industry, states that the most commonly used RPC methods are those that do not require Merkle proofs,[1] e.g., `eth_getBalance` and `eth_call`, indicating that the authentication of blockchain states is rarely used in the current Ethereum ecosystem. Luo et al. [37] also point out that two of the most frequent trust models for users to read data from the blockchain, in reality, are trusting third-party nodes and deploying self-owned nodes. In these trust models, data authentication is also unnecessary.

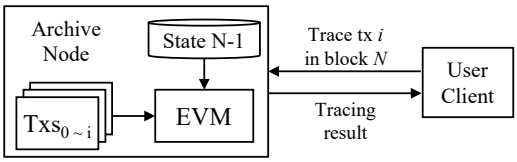


Figure 3: An example of tracing transaction i in block N. The execution from transaction 0 to $i-1$ is pre-processing. Only the execution of transaction i is effective.

> **Insight 1.** Data authentication of historical states is **not** required in most real-world usage scenarios.

Our Solution. Consequently, to achieve a more efficient and cost-effective archive node when data authentication is not required, we could employ a compacted and flattened data model instead of the complicated MPT for historical states to minimize intermediate data and simplify state access.

3.2 Coarse-Grained State Granularity

The Ethereum protocol defines that the world state updates at the end of each block. Thus, the granularity of Ethereum's historical states is a block. However, the block-level historical states fail to provide the intra-block state, i.e., the historical state at a transaction. To retrieve the state at a specific transaction, it is necessary to re-execute all the transactions before the target transaction in the block to recover the world state. We define the execution of these transactions as **pre-processing**. Figure 3 shows an example of this inefficient historical state access. To trace transaction i in block N, all precedent transactions (index ranges from 0 to $i-1$) in the block must be executed as the pre-processing. Only the execution of transaction i is **effective** and outputs the tracing result. This substantially increases the system overhead as many unnecessary transactions are executed [29, 55].

[1] https://www.infura.io/blog/post/ethereum-rpcs-methods.

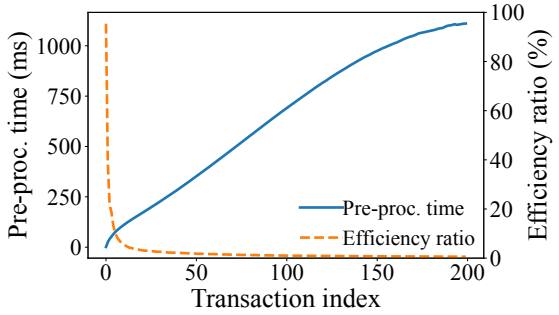


Figure 4: The pre-processing time and efficiency ratio of transaction tracing with Geth.

To better understand the overhead incurred by the block-level historical states, we sample 10K blocks with block numbers ranging from 13M to 14M and process approximately 1.76M transactions. We define the **efficiency ratio** as the ratio of the effective execution time to the total (pre-processing and effective execution) time. The effective execution time is 3.4 hours, while the pre-processing cost reaches 297.0 hours. Consequently, the efficiency ratio is only 1.1%, indicating that 98.9% *of the total execution time is spent on the pre-processing*. Furthermore, we analyze the pre-processing time and efficiency ratio under different transaction indexes. Figure 4 illustrates that as the transaction index reaches 200, the pre-processing cost exceeds 1 second, and the efficiency ratio approaches near zero.

Our Finding. The coarse-grained block-level historical states drastically undermine the efficiency of archive nodes. Fortunately, we observe that the processing of a block is composed of the sequential execution of transactions by the EVM.

> **Insight 2.** Thus, while Ethereum appears to have block-level state transition externally, at the low-level execution layer (EVM), the **basic unit** causing state transition is the **transaction**, i.e., the granularity of state transition at the execution layer is a transaction.

Our Solution. Consequently, we could refine the granularity of historical states to a transaction at the execution layer to eliminate the overhead caused by the pre-processing.

4 SLIMARCHIVE

4.1 Overview

Objectives. SLIMARCHIVE is a novel architecture for Ethereum archive nodes, designed to address the issues discussed in Section 3. It aims to achieve the following three objectives. First, SLIMARCHIVE is designed to be *lightweight*, requiring minimal computation and storage resources. Second, SLIMARCHIVE is designed to be *flexible*, providing access to historical states at the transaction-level granularity. Third, SLIMARCHIVE is designed to be *high-performance*, providing performant state access and transaction execution.

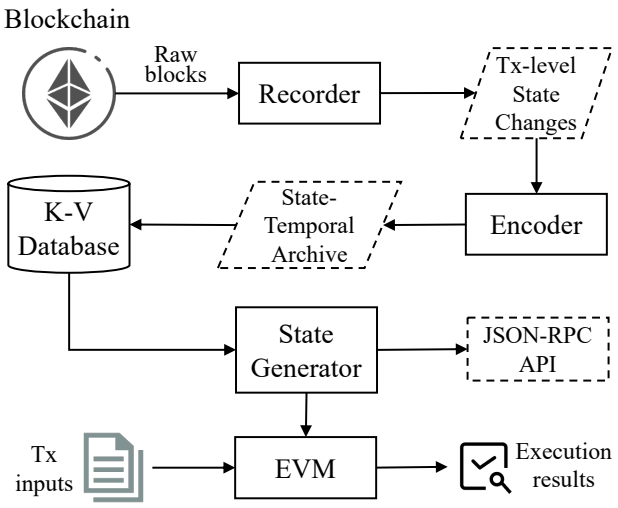


Figure 5: Components and workflow of SLIMARCHIVE.

Figure 5 shows the components and workflow of SLIMARCHIVE. SLIMARCHIVE consists of three key components: The *recorder* is an instrumented EVM that collects state changes of each transaction when synchronizing new blocks. The *encoder* encodes the state changes into the *state-temporal archive*, which is the flattened representation of historical states at the transaction-level granularity. The *state-temporal archive* is persisted in the key-value store. The *state generator* loads the *state-temporal archive* and recovers the world state specified by the address, storage key, and transaction position (block number and transaction index). The EVM retrieves states from the *state generator* and executes input transactions. External users can also query historical states by the JSON-RPC APIs [5].

4.2 Recorder

The *recorder* is an instrumented EVM that collects transaction-level state changes during block processing. To minimize recorded data, the *recorder* collects only the necessary changed states to be updated to the world state. To reduce the redundancy, the *recorder* records only the post-state (write value) of each transaction. Except for transaction execution, there exist other actions that also mutate the world state. To ensure the completeness of historical states, we propose the concept of the *virtual transaction*: All actions that mutate the world state outside the transaction execution are considered virtual transactions. The *recorder* also collects state changes of virtual transactions. In brief, the *recorder* collects each transaction's minimum and complete state changes.

Transaction-Level State Changes. Specifically, the *recorder* collects four types of changed states after executing each transaction, i.e., account, storage, code, and deleted states. First, the account state is in the format of the 2-tuple: `(Address, SlimAccount)`. The `SlimAccount` is a struct consisting of three fields, i.e., `Balance`, `Nonce`, and `CodeHash`, characterizing the account state. Second, the storage state is in the format of the 3-tuple: `(Address, SlotKey, SlotValue)`, which represents the slot value specified by the address and slot key. Third, the code state is in the format of the 2-tuple: `(CodeHash, Bytecode)`, which represents the bytecode with the corresponding `CodeHash` of the created contract account. Finally, the deleted state is a set of addresses indicating the deleted contract accounts after the transaction. The contract account could self-destruct and clear its account and storage states. The deleted state is used to help recover the storage state in Section 4.4. The transaction metadata, including block number, hash, and transaction index, is also recorded to maintain the temporal information. The authentication data (e.g., `StorageRoot`) is not maintained. Besides, EVM's volatile runtime data, including memory, stack, call data, and return value, are also discarded.

State Change Collection. The *recorder* leverages the state *journal* of Geth to collect state changes during the execution of each transaction.The *journal* is a slice of operation logs that tracks all state writing operations during transaction execution. After the execution of each transaction, the *recorder* collects dirty states from the *journal*. Note that the writing logs in the *journal* do not ensure that the state is changed after the transaction execution. For example, two opposite writing operations do not change the state after the transaction. But they are tracked in the *journal*. Thus, the deduplication must be performed by comparing the pre- and post-states.

Typically, A block finalization process is performed at the end of each block (after the execution of all transactions) for consensus purposes (e.g., mining rewards distribution [6] and staking withdrawals [10]). As such, the block finalization process may also mutate the world state. To maintain the completeness and correctness of historical states, the *recorder* perceives the block finalization process as a virtual transaction at the end of the block and collects the state changes of the block finalization process.

The genesis block (the block number is 0) is a special block to hard-code the initial states. The above approach can not collect the state changes of the genesis block because the block is never processed. Thus, to collect the state changes of the genesis block, we also perceive the initialization process as a virtual transaction. We hook the blockchain initialization to collect the state changes in the genesis block.

In summary, the recorder outputs four types of state changes for each (virtual) transaction as follows:

- The account state is in the format of `(Address, SlimAccount)`, which represents the current account state (except for storage) specified by the account address.

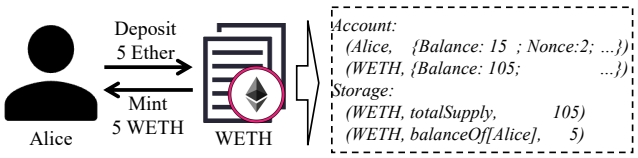


Figure 6: An example of transaction-level state changes. The left part is the deposit transaction to WETH. The right part is the consequent state changes.

- The storage state is in the format of `(Address, SlotKey, SlotValue)`, which represents the current slot value specified by the account address and the slot key.
- The code state is in the format of `(CodeHash, Bytecode)`, which represents the newly deployed contract bytecode addressed by its code hash.
- The deleted state is a set of addresses that represents the deleted contract accounts.

Example State Changes. Figure 6 shows an example of the transaction-level state changes (right part) of a deposit transaction (left part) to WETH. WETH [16] is an ERC-20 token [2] that is pegged to Ether at a 1:1 ratio. Users can deposit Ether to the token contract. The contract will mint equivalent WETH tokens to users. Alice transfers five Ether to WETH and gets five WETH tokens back in the deposit transaction. The transaction alters the account state of Alice and WETH. Besides, two storage slots of WETH are modified by the transaction: The `totalSupply` indicates the total number of minted tokens. The `balanceOf[Alice]` indicates Alice's WETH balance. Note that the transaction fee is omitted in the example for the sake of simplification.

4.3 Encoder

The *encoder* encodes the transaction-level state changes into the *state-temporal archive*, a flattened data structure that minimizes intermediate data and simplifies state access. Note that it is hard to directly store the state changes in the key-value store for the following reasons. First, the key-value store only supports the 2-tuple (key-value) format data, but the state changes have more dimensions, such as the address, slot key, block number, transaction index, and state value. Thus, the multidimensional transaction-level state changes are not supported by the key-value store. Besides, the limited query functions of the key-value store undermine the accessibility of historical states [52]. For example, the query of the current balance of an account can be expressed in a SQL query statement *select balance from account_state where address = [addr] order by blockNumber limit 1*. However, such query language is not supported by the key-value store. Thus, to persist historical states while enabling efficient access, the *encoder* encodes state changes so that they can be accessed

Table 1: The encoding rules of state-temporal archive.

State Type	Key					Value
	State Key			Temporal Key		
Account	Account Flag	Address		Block Number	Tx Index	RLP(SlimAccount)
Deleted	Deleted Flag					Empty String
Storage	Storage Flag	Address	Slot Key			Slot Value
Code	Code Flag	Code Hash		N/A		Contract Bytecode

from the key-value store. Specifically, the *encoder* flattens state changes and arranges them chronologically. Compared to the standard MPT, the flattened structure eliminates intermediate data; the chronological order enables efficient state access. We propose a new data structure named *state-temporal archive* for historical states to achieve the above goals.

State-Temporal Archive. We first define two operations. The `append()` is the concatenation for multiple byte strings, and the `RLPEncode()` means the Recursive Length Prefix [54] encoding that serializes a struct (e.g., the `SlimAccount`) to the corresponding byte string.

To facilitate the encoding, we divide each state change into three parts: state key, temporal key, and state value. The first part is the state key that represents **which** state is changed. The state key is defined in Equation 1. A state flag is inserted at the head of the state key to distinguish different types of state changes. The state ID for account and deleted states is the address. For storage and code states, the state ID is `append(Address, SlotKey)` and `CodeHash`, respectively.

$$StateKey = Append(StateFlag, StateID) \quad (1)$$

The second part is the temporal key. It represents **when** the state change occurred. We use the big-endian block number and transaction index to represent the timestamp. The temporal key is defined in Equation 2, where `NumberBytes` and `IndexBytes` represent the 6-byte block number and 2-byte transaction index, respectively.

$$TemporalKey = Append(NumberBytes, IndexBytes) \quad (2)$$

The last part is the state value. It represents **what** the state is after the transaction. The state value for the account, storage, and code is `RLPEncode(SlimAccount)`, `SlotValue`, and `Bytecode`, respectively. The deleted state is a set of addresses to mark the deleted accounts. We define an empty string as the state value of the deleted state.

Given the 3-part formalization, the *encoder* encodes each state change into the *state-temporal archive* in the format of key-value pairs. The mapping relations are as follows:

- **For the types of account, deleted and storage:**

$$Append(StateKey, TemporalKey) \rightarrow StateValue$$

- **For the type of code:**

$$StateKey \rightarrow StateValue$$

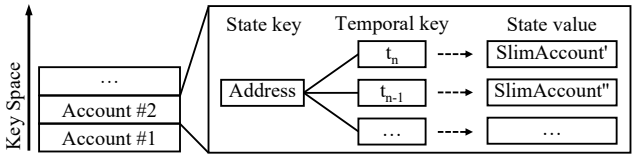


Figure 7: An example of *state-temporal archive*. The historical accounts are sorted in the chronological order.

The state-temporal key, consisting of two parts, is sufficient to locate the historical state. The prefix is the state key. The suffix is the temporal key. Note that since the bytecode is addressed only by the code hash, the temporal key is unnecessary for the code type. Table 1 shows the detailed encoding rules of the *state-temporal archive*. Note that both state key and temporal key are fixed-length for each type of state.

Flattened Historical States. The *state-temporal archive* is a flattened data structure for the transaction-level historical states. Each key-value pair contains the **minimum but complete** data (which, when, and what) to describe one state change of an entity. Since the key-value store arranges keys in lexicographic order, the dual-part, aligned state-temporal key maintains a **partially chronological order** of the state changes. The prefix state key clusters the state changes for each entity. Meanwhile, the suffix temporal key guarantees the chronological order within the clustered state changes. Given a time point, searching for the most recent state change of an entity becomes straightforward.

Figure 7 shows an example of *state-temporal archive* for account states. *No intermediate key-value pairs unrelated to the historical states are produced.* It is possible to scan an entity's state changes based on chronological order.

4.4 State Generator

The *state generator* recovers historical states from the flattened *state-temporal archive*. Specifically, it takes the state key and temporal key as inputs and returns the corresponding historical state. The code state is only addressed by the state key (`CodeHash`). For account and storage states, the state key represents the entity for which the state is to be queried, whereas the temporal key represents the specific time at which the state is to be queried. To be compatible with the stan-

Algorithm 1: Historical State Query Algorithm

Input: State key: *stateKey*;
Temporal key: *temporalKey*.
Output: The historical state and modification time.

```
1  Function StateAt(stateKey, temporalKey):
2      lower ← stateKey
3      upper ← Append(stateKey, temporalKey)
4      iter ← NewIterator(db, lower, upper)
5      SeekLast(iter)
6      if IsValid(iter) then
7          modifyTime ← ExtractTemporal(iter.Key)
8          return (iter.Value, modifyTime)
9      else
10         return (null, null)
11     end
12 end
```

dard Ethereum archive node, the block-level historical state is represented as the state at the block's first transaction (the transaction index is 0). The EVM reads states from the *state generator* when executing transactions. Besides, users can also access historical states directly via the JSON-RPCs.

As the *state-temporal archive* stores all state changes, querying the state at a specified time point equals querying the last state change that occurred before the time point. Since state changes are in chronological order, searching the target state change is feasible by seeking the greatest key smaller than the corresponding state-temporal key, which can be completed by a **single** database range scan operation. Note that the search scope is constrained by the state key as the prefix. The range scan of the key-value store is highly efficient [18].

Algorithm 1 describes the core function `StateAt` utilized for querying historical states. `SeekLast` means moving the iterator to the last key-value pair. `ExtractTemporal` means extracting the temporal part from the key via string slicing. The `StateAt` function takes the state and temporal keys as input parameters. The outputs are the last state change and corresponding modification time (in the format of the temporal key). The function initializes an iterator with the lower bound as the state key and the upper bound as the state-temporal key for iteration. Then, seeking the last key-value pair will retrieve the target state and corresponding modification time.

Account Query. Given the address and timestamp (block number and transaction index), the *state generator* constructs the corresponding state key and temporal key for the account. Then, searching the last account change via the `StateAt` function will retrieve the target account state.

Storage Query. Since the self-destruct operation will clear the storage of the contract account, searching only the storage changes is insufficient to fetch the historical storage state. For example, if a contract self-destructs after the slot write, searching storage changes will get the writing value. However, the real state of the slot is the empty value. Thus, to fetch the correct historical storage state, the `state generator` searches both the latest storage change and the latest deleted time. Only the storage change after the delete operation is the valid storage state. Otherwise, the storage state is the empty value.

4.5 Implementation Details

State Validation. It is necessary to validate states during synchronization since the peers in the network are untrusted. This involves comparing the state root of the local world state tree with that recorded in the received block. To facilitate this validation process, SLIMARCHIVE maintains the latest MPTs to compute state roots during synchronization.

Chain Reorganization. Chain reorganization (reorg or rollback) is performed when the local chain diverges from the canonical chain, i.e., chain forks [46, 48, 54]. When this happens, the forked blocks (and their corresponding states) are discarded. The blocks from the fork point to the latest block in the canonical chain are processed to re-establish the world state. Typically, a reorg involves only a few blocks. To prepare for potential reorgs, a Geth full node [9] maintains the MPTs of the most recent 128 blocks, thereby enabling a reorg of up to 128 blocks. SLIMARCHIVE employs a similar strategy to Geth to tackle chain reorgs, i.e., buffering the *state-temporal archive* of the latest 128 blocks in memory. Should a reorg occur, we discard the outdated data and re-generate the corresponding canonical *state-temporal archive*. States that exceed the most recent 128 blocks could be considered immutable and are persisted to disk. Besides, SLIMARCHIVE also maintains the most recent 128 MPTs to enable state validation when chain reorg occurs.

Storage Engine. We employ Pebble [15], a Log-Structured Merge-Tree (LSM-Tree, [41])-based key-value store, as our storage engine for several reasons. First, the *state-temporal archive* is straightforward and aligned. Simple query interfaces (i.e., get and scan) could fulfill the requirements in the Ethereum context. Thus, key-value stores are well-suited for these scenarios and generally outperform other storage engines like relational databases. Second, key-value stores like Pebble can be embedded in progress without external software dependencies. This integration reduces the system's complexity. Lastly, by choosing Pebble—a decision also made by Geth—we avoid additional complexity and overhead.

We have implemented a prototype of SLIMARCHIVE based on the Geth v1.15.5. It is worth noting that although the implementation is for Ethereum, the concepts and technical solutions of SLIMARCHIVE can be applied to other EVM-compatible blockchains, e.g., Binance Smart Chain and Polygon, since they employ similar mechanisms to maintain the world state and execute transactions as Ethereum.

Table 2: Comparison of the time (hours) spent on the historical state generation. Geth does not have the aggregation stage.

	Blocks	Agg.	Persisting	Total
Geth	0-14M	N/A	961.9	961.9
Erigon	0-18M	15.7	29.9	45.6
SLIMARCHIVE	**0-18M**	**11.0**	**3.7**	**14.7**

5 Evaluation

We first evaluate the performance of SLIMARCHIVE from various perspectives, including *archive node synchronization*, *historical state access*, and *transaction execution*. Then, we evaluate the *storage overhead* introduced by refining the state granularity. Finally, we validate the *correctness* of SLIMARCHIVE. We also evaluate Geth and Erigon [3] for comparison. All experiments are carried out using real-world Ethereum workloads.

Environment Setup. We conduct experiments using a physical machine equipped with the CPUs of Intel(R) Xeon(R) Silver 4214R CPU @ 2.40GHz with 24 cores × 2 sockets, 377 GB memory, 4 × 3.84 TB SAMSUNG PM893 SATA-6.0 SSDs. The operating system is Ubuntu 22.04.2 LTS.

5.1 Archive Node Synchronization

First, we investigate the time and storage consumption required for archive node synchronization. Since the archive node records all historical states, it is necessary to synchronize the blockchain from the genesis block. We perform the synchronization of the initial 18M blocks as of August 26, 2023, for both SLIMARCHIVE and Erigon. Due to the extensive computational and disk requirements, Geth's synchronization process can only proceed up to block height 14M as of January 13, 2022.

Time Cost. The synchronization of archive nodes involves many steps, such as transaction execution, result verification, and state update. However, we only focus on state updates that generate historical states since the others are outside the scope of this work. We break down the state update into two stages. The first is the in-memory data aggregation, which collects and encodes the dirty states in memory. The second is data persisting that writes the new world state to disk. Note that maintaining the latest MPTs should not be included in comparison since all Ethereum implementations must perform this step. Thus, Geth does not have the aggregation stage as it directly persists the MPT to disk without extra transformation like SLIMARCHIVE and Erigon. Table 2 shows the time spent on generating the historical states. SLIMARCHIVE outperforms Geth and Erigon in both data aggregation and persisting stages. Geth spent 961.9 hours on the initial 14M blocks. SLIMARCHIVE and Erigon used 14.7 and 45.6 hours for the initial 18M blocks, respectively. SLIMARCHIVE exceeds a 65.4× speedup and achieves a 3.1× speedup compared with Geth and Erigon, respectively.

Table 3: Comparison of the storage usage for the historical states. The block number ranges from 0 to 18M.

	Geth	Erigon	**SLIMARCHIVE**
Storage (GB)	14,041.5	791.5	**267.6**

Storage Usage. Similar to the calculation of time cost, we only consider the storage used for historical states, while others, like blocks, transactions, and logs, are not included. Table 3 shows the comparison of the storage usage of historical states. Geth and Erigon consume 14,041.5 GB and 791.5 GB storage to save all historical states, respectively.[2] However, SLIMARCHIVE only needs 267.6 GB storage to store the transaction-level *state-temporal archive*. SLIMARCHIVE reduces the storage requirements by 98.1% and 66.2% compared with Geth and Erigon, respectively.

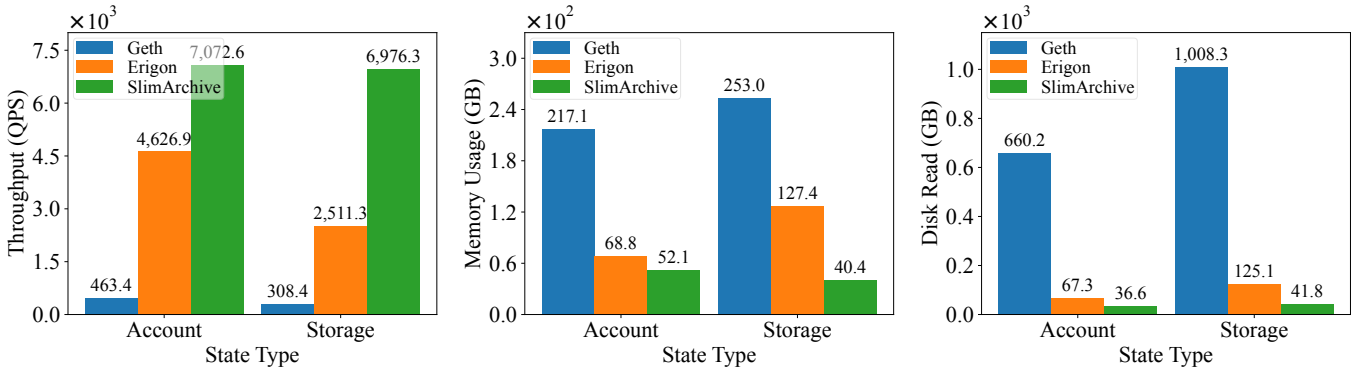
Summary. The time cost and storage usage demonstrate that SLIMARCHIVE effectively reduces the computation and storage resources required for historical states. Unlike Geth and Erigon, SLIMARCHIVE saves only the changed states using a compacted data structure, thus economizing on computation and storage by minimizing the intermediate data.

5.2 Historical State Access

Archive nodes provide the fundamental function of accessing historical states. Users can query historical `Balance`, `Nonce`, `Code`, and `Storage` of accounts via JSON-RPC APIs. Therefore, we examine the access performance of these four types of historical states. It is worth noting that SLIMARCHIVE, along with Geth and Erigon, encodes `Balance`, `Nonce`, and `CodeHash` into a single account object using the RLP serialization. Thus, we evaluate two query interfaces: `GetAccountAt(address, number)` and `GetStorageAt(address, key, number)` for accessing historical accounts and storage. Specifically, we measure the throughput and system overhead (memory usage and disk reading) of the two query interfaces. The metrics for the throughput, memory overhead, and disk overhead are the Query Per Second (QPS), the maximum Resident Memory Size (RES), and the `read_bytes` field of the I/O description file of the process, respectively.

Workload. We randomly select 10M addresses and 10M (address, storage key) pairs, each of which represents a state accessed during the processing of the initial 14M blocks. Then, we generate a random block number ranging from 0 to 14M for each state. These are the inputs for the query processes.

[2] Geth storage is estimated by subtracting full node size from archive node size, based on data from Etherscan: https://etherscan.io/charts.



(a) Access throughput. **(b) Memory overhead.** **(c) Disk overhead.**

Figure 8: Comparison of the throughput and system overhead of historical state access.

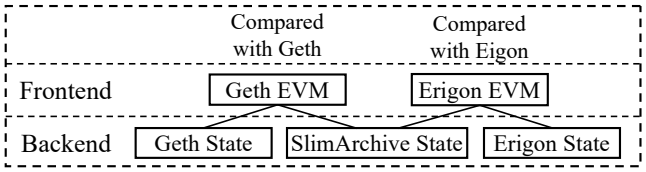


Figure 9: The methodology of evaluating transaction execution: controlling the frontend for each comparison.

Figure 8 shows the throughput and the system overhead when querying the historical states of the 10M accounts and the 10M storage slots.

Throughput. SLIMARCHIVE improves the account query throughput by 15.3× and 1.5× compared with Geth and Erigon, respectively. For the storage type, the improvements are 22.6× and 2.8×, respectively. SLIMARCHIVE achieves a total (account and storage) throughput of 7,024.1 QPS and is 19.0× and 2.2× higher than Geth and Erigon, respectively. SLIMARCHIVE achieves higher throughput.

Memory Overhead. For the account query, SLIMARCHIVE utilizes only 24.0% and 75.7% of the memory compared with Geth and Erigon, respectively. For the storage query, the percentages are even less, i.e., 15.9% and 31.7%. SLIMARCHIVE utilizes less memory.

Disk Overhead. When querying accounts, SLIMARCHIVE reduces the I/O size to 5.5% and 54.4% compared with Geth and Erigon, respectively. While for the storage query, the percentages are 4.1% and 33.4%. SLIMARCHIVE requires less I/O operations.

Summary. SLIMARCHIVE outperforms Geth and Erigon in all metrics, as it requires less system overhead and delivers higher throughput. The reasons are as follows. To retrieve the state, Geth needs to traverse the state tree from the root to the leaf. The traversals of MPTs demand substantial disk lookups and lead to lower throughput and higher system overhead.

Table 4: Transaction execution speedups (×) over Geth and Erigon achieved by SLIMARCHIVE.

Speedup over	Mean	Median
Geth	1112.5	672.2
Erigon	109.4	60.9

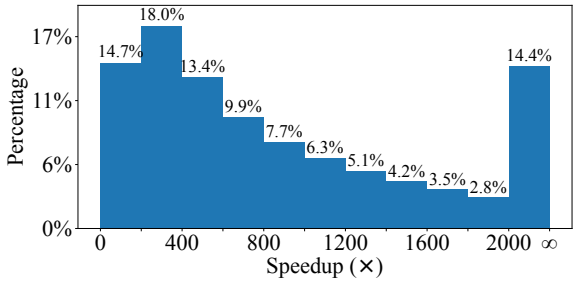
Although Erigon optimizes the processing, it still requires multiple disk lookups. In contrast, SLIMARCHIVE employs a flattened data structure for historical states. Compared with Geth and Erigon, SLIMARCHIVE performs minimum disk lookups when retrieving the historical state.

5.3 Transaction Execution

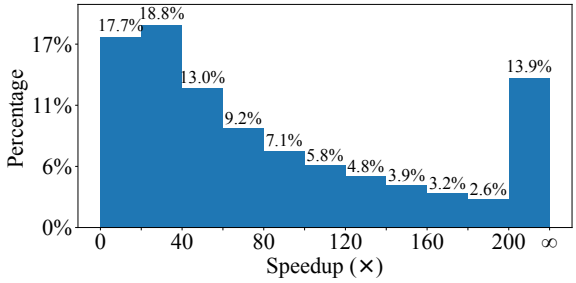
To highlight the effectiveness and efficiency of our transaction-level archive node, we evaluate the performance of transaction execution based on historical states. Specifically, we measure the time spent on executing historical transactions.

We formalize transaction execution as a `frontend-backend` scheme, where the frontend is the EVM that executes instructions, and the backend is the state management component that provides states for the EVM. Because Erigon and Geth employ different designs and implementations of the EVM, our comparison experiments should use the same EVMs as those in Geth and Erigon to control variables. Figure 9 illustrates our methodology: we control the frontend (EVM) during each comparison. We have also implemented the query interfaces for the Erigon EVM to access states from SLIMARCHIVE. We sample 10K blocks from the 13M-th block to the 14M-th block as the workloads and execute a total of 1,762,188 transactions.

Table 4 presents the overall transaction execution speedups. The mean and median speedups achieved by SLIMARCHIVE over Geth are 1112.5× and 672.2×, respectively. Compared



(a) Speedup over Geth.



(b) Speedup over Erigon.

Figure 10: Transaction execution speedup distribution. The y-axis is the percentage of transactions.

to Erigon, the speedups are 109.4× and 60.9×, respectively. We also present the speedup distribution in Figure 10. Compared to Geth, 85.3% and 14.4% of the transactions are accelerated exceeding 200× and 2,000×, respectively. In comparison to Erigon, the proportions are 82.3% and 13.9% for the speedups of 20× and 200×, respectively.

Furthermore, we analyze the correlation between the transaction index and the execution speedup achieved by SLIMARCHIVE. As shown in Figure 11, SLIMARCHIVE achieves higher speedups as the transaction index increases. When the transaction index reaches 300, the average speedups over Geth and Erigon reach about 3,100× and 300×, respectively.

Summary. Thanks to the transaction-level historical states, executing transactions no longer requires the extremely time-consuming pre-processing stage, as in Geth and Erigon. SLIMARCHIVE remarkably increases the speed of transaction execution by several orders of magnitude, demonstrating the effectiveness and efficiency of the fine-grained, transaction-level historical states.

5.4 Storage Overhead

Refining the granularity of historical states from block level to transaction level introduces two types of storage overhead. The first is the 2-byte transaction index in the temporal key; The second lies in the intra-block state changes. For instance, consider an account changed by n transactions in a block.

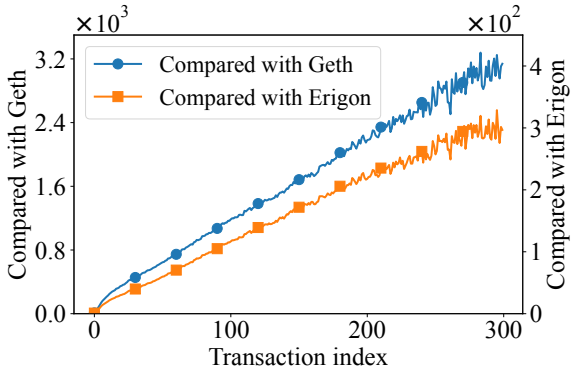


Figure 11: The positive correlation between transaction index and execution speedup. The y-axes are the average execution speedups over baselines achieved by SLIMARCHIVE.

The block-level archive node produces only one state change, while the transaction-level archive node produces n state changes. We also evaluate the storage consumption of SLIMARCHIVE under block-level granularity. The block-level *state-temporal archive* consumes 194.3 GB storage, revealing that refining the granularity carries a storage overhead of 73.3 GB. Since the transaction-level historical states remarkably improve transaction execution performance (by 1112.5×), and the compacted and flattened data structure significantly reduces the overall storage (by 98.1%), we believe the storage overhead is acceptable.

5.5 Correctness Validation

We first validate the correctness of block-level historical states by performing a run-time state consistency check during synchronization. The EVM will track all state modifications during the block processing. The dirty states will be updated to the Ethereum MPT at the end of each block. We compare the updated states with that generated by SLIMARCHIVE before block commitment. No inconsistent state exception is raised during the synchronization of the initial 18M blocks. Additionally, Experiment 5.2 also validates the correctness. We compare the outputs of SLIMARCHIVE, Geth, and Erigon. All the query results are identical.

We also validate the correctness of our transaction-level historical states by re-executing historical transactions. We compare the execution results (e.g., outputs, gas usage, and logs) of re-executed transactions and real-world transactions. If our transaction-level historical states are misrecorded, the re-execution will produce different results. We re-execute all (2.07B) transactions of the 18M blocks. All the results are identical to those of the real-world transactions, demonstrating the correctness of SLIMARCHIVE.

6 Discussion

The Intention of Data Authentication in Ethereum. The authenticated MPT allows users who maintain the state roots to verify the integrity of states retrieved from untrusted blockchain nodes [33]. This mechanism is primarily utilized by light nodes, which store only the state roots and retrieve state data from untrusted full or archive nodes that maintain the standard MPT structure. When a light node distrusts external nodes, it requests the state along with corresponding Merkle proof and validates the response using the self-owned state root. In summary, the MPT enables light nodes to verify the integrity of data retrieved from untrusted nodes.

Trade-offs, Limitations, and Trust Model. SLIMARCHIVE simplifies blockchain archive nodes. Our method aims to meet the requirements for more efficient archive nodes in most real-world usage scenarios where data authentication is not required. Thus, SLIMARCHIVE sacrifices the authentication data (MPTs) to reduce storage and improve performance. Our system is not appropriate for scenarios that require authentication of historical states, such as when light nodes, distrusting SLIMARCHIVE, request historical states. Unlike the standard Ethereum execution client, Geth, SLIMARCHIVE requires users' trust when accessing historical states.

Practical Significance. Despite the loss of authentication functionality, SLIMARCHIVE still holds practical significance for the following reasons. Within the current Ethereum ecosystem, different types of nodes (such as light, full, and archive nodes) exist to serve various tasks. Nevertheless, there are few solutions tailored for scenarios in which the authentication for historical states is unnecessary, yet these scenarios constitute the majority of real-world applications. To address this gap, we propose our cost-efficient and high-performance solution. SLIMARCHIVE is not intended to replace existing systems. Instead, our system could complement existing solutions. For instance, service providers like Infura could maintain both types of archive nodes to balance common proof-unrelated and rare proof-related queries, thereby improving overall service efficiency. Additionally, users who wish to deploy cost-effective local archive nodes could employ our method. Therefore, we believe that the trade-offs made by SLIMARCHIVE are reasonable considering practical demands.

Security and Data Integrity. As introduced in Section 4.5, SLIMARCHIVE adopts a strategy similar to that of a Geth full node [9] for state validation and chain reorganization during synchronization. Thus, SLIMARCHIVE could operate independently (i.e., without trusted third parties) within the blockchain network, providing the same level of security as a standard Ethereum full node has. Compared with a standard archive node, SLIMARCHIVE only prunes the MPTs of historical states, while other blockchain data, e.g., blocks, transactions, and logs, are preserved. The only data loss is the authentication information of historical states, which, as discussed earlier, is not required in many scenarios.

7 Related Work

Blockchain Storage Optimization. Many works optimize the blockchain storage layer as Ethereum scales up.

Erigon [3] optimizes Ethereum's historical states by maintaining three distinct tables: the read set that records the initial states required for processing each block, the read set's inverted index that maps from the address (or slot key) to a list of block numbers where the corresponding state is loaded to execute transactions, and the map that records the current states of accounts and storage. State access in Erigon involves two or three steps: First, locate the state by the inverted index. Second, retrieve the state by the location from the read set. Third, read the current state if the state is not recorded in the read set. Although Erigon simplifies the data model of historical states, it still introduces significant intermediate data (e.g., the inverted index). Besides, each state access is amplified to two table lookups. Moreover, Erigon also suffers from coarse-grained block-level granularity. In contrast, SLIMARCHIVE produces no intermediate data and requires only one database operation to retrieve the transaction-level historical state.

Some works focus on optimizing the authenticated layer for blockchain full nodes. Li et al. [33] propose LVMT, a novel storage framework designed to overcome the performance bottlenecks caused by the standard MPT. LVMT uses a multi-layer Authenticated Multipoint evaluation Tree (AMT, a vector commitment protocol) to compute the commitment (a.k.a state root). In contrast to the standard Ethereum MPT, where updating state root has a time complexity of $O(\log n)$, LVMT achieves a constant time for generating commitments. However, LVMT is incompatible with current EVM-compatible blockchains due to the fundamental difference in state commitments' computations. Zhang et al. [61] propose COLE, a column-based learned method to address the limitations inherent in the MPT. By storing each entity's historical states in a column and indexing these states using learned models, COLE enhances both the efficiency of data storage and the speed of query processing within blockchain systems. Choi et al. [23] use three-layer MPTs to maintain the world state, i.e., one on-disk MPT maintaining the world state at a checkpoint and two in-memory MPTs holding dirty states. The in-memory MPTs serve as caches that record recently visited data, thus reducing I/O operations when executing transactions. The in-memory MPTs are periodically (e.g., every $1,000$ blocks) merged into the on-disk MPT. However, this approach is unsuitable for archive nodes, as archive nodes require persisting the world state at each block rather than periodically.

Chen et al. [22] propose Block-LSM, an Ethereum-oriented key-value store leveraging the ordered nature of Ethereum (i.e., block sequence) to improve the synchronization performance. Block-LSM uses a shared prefix scheme and several semantic-orientated memory buffers specifically for Ethereum data to reduce the write amplification [45] and improve the access throughput of the key-value store.

Many works [17, 24, 36, 42, 58, 67] attempt to compress or prune the blockchain data to reduce the storage requirements. However, these methods also have limitations, such as incomplete blockchain states and inefficient data processing, which compromise the usability and applicability of blockchain [27, 52]. Sharding [25, 30, 38, 43, 49, 60] is another popular method utilized to scale blockchain. It partitions blockchain data and transaction tasks into multiple entities to alleviate the pressure on a single node and increase the system throughput. However, this approach introduces extra system overhead, e.g., network and storage [27, 52].

Although the above-mentioned systems attempt to optimize blockchain storage from various angles, they have inherent limitations, as discussed. Moreover, they fail to address the issue of how to minimize the storage without compromising the Ethereum states' availability and an archive node's functionalities. In contrast, SLIMARCHIVE preserves the functionality of an archive node while simultaneously optimizing the storage consumption. However, it's worth noting that SLIMARCHIVE and the methods above solve the problems from different perspectives and complement each other.

Transaction Re-execution. Kim et al. [29] propose an off-the-chain execution environment that records the initial context (i.e., substate) of each historical transaction before it is executed, allowing historical transactions to be re-executed in isolation and parallel based on their substate. EthScope [55] instruments an Ethereum full node to aggregate intermediate data for each transaction and stores these aggregation results in ElasticSearch [1]. It uses independent EVMs (i.e., replay engines) to re-execute historical transactions with the intermediate states. The above two methods employ similar strategies, i.e., they collect the complete initial states required for executing each transaction and re-execute historical transactions in isolated environments. Additionally, the prefetch of the initial states is conducted to accelerate execution. However, both EthScope and [29] are domain-specific, i.e., they focus on only the re-execution of historical transactions. As they preserve only the initial states required for each transaction execution, they fail to serve as archive nodes, which require additional capabilities to query blockchain historical states at any specified time and simulate transactions.

Historical Data Exploration. Many data-centric systems focus on extracting, loading, and transforming historical blockchain data for various tasks. Several works [19–21, 34, 66] improve data acquisition methods, facilitating extensive Ethereum historical data exploration. EthScope [55] and TxSpector [63] extract the intermediate data of historical transactions for locating suspicious behaviors at scale. Zhou et al. [68] analyze historical transactions to measure the evaluation of real-world attacks. Zhao et al. [65] investigate the evolution of the Ethereum network from a temporal perspective. As historical data provide significant insights, SLIMARCHIVE could serve as the data source for the above works as it records all the historical states.

8 Conclusion

This paper presents the design and implementation of SLIMARCHIVE, a lightweight and high-performance architecture that addresses the storage and performance problems of current Ethereum archive nodes. SLIMARCHIVE flattens the minimum state changes of each transaction required for the world state, achieving maintaining compacted, flattened, and fine-grained historical states. The evaluation results demonstrate that our approach significantly reduces storage consumption (by 98.1%) and highly improves performance (19.0× for state access and 1112.5× for transaction execution).

Acknowledgments

We thank all anonymous reviewers for their invaluable comments. This work is partially supported by the National Key R&D Program of China (No. 2022YFE0113200), the National Natural Science Foundation of China (NSFC) under Grant 62172360, U21A20464. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of funding agencies.

References

[1] Elasticsearch. https://www.elastic.co/guide/index.html. [Online; accessed December-2023].

[2] Erc-20: Token standard. https://eips.ethereum.org/EIPS/eip-20/. [Online; accessed December-2023].

[3] Erigon client. https://github.com/ledgerwatch/erigon/. [Online; accessed December-2023].

[4] Ethereum archive node. https://ethereum.org/en/developers/docs/nodes-and-clients/archive-nodes/. [Online; accessed December-2023].

[5] Ethereum json-rpc api. https://ethereum.org/en/developers/docs/apis/json-rpc/. [Online; accessed December-2023].

[6] Ethereum mining rewards. https://ethereum.org/zh-tw/developers/docs/consensus-mechanisms/pow/mining/. [Online; accessed December-2023].

[7] Ethereum network layers. https://ethereum.org/en/developers/docs/networking-layer/. [Online; accessed December-2023].

[8] Ethereum node architecture. https://ethereum.org/en/developers/docs/nodes-and-clients/node-architecture/. [Online; accessed December-2023].

[9] Ethereum nodes and clients. https://ethereum.org/en/developers/docs/nodes-and-clients/. [Online; accessed December-2023].

[10] Ethereum staking withdrawals. https://ethereum.org/en/staking/withdrawals/. [Online; accessed December-2023].

[11] Go-ethereum (geth) client. https://github.com/ethereum/go-ethereum/. [Online; accessed December-2023].

[12] Go-ethereum (geth) debug namespace. https://geth.ethereum.org/docs/interacting-with-geth/rpc/ns-debug/. [Online; accessed December-2023].

[13] Go-ethereum (geth) evm tracing. https://geth.ethereum.org/docs/developers/evm-tracing. [Online; accessed December-2023].

[14] Merkle patricia trie. https://ethereum.org/en/developers/docs/data-structures-and-encoding/patricia-merkle-trie/. [Online; accessed December-2023].

[15] Pebble key-value store. https://github.com/cockroachdb/pebble. [Online; accessed December-2023].

[16] Wrapped ether (weth). https://etherscan.io/address/0x4f26ffbe5f04ed43630fdc30a87638d53d0b0876. [Online; accessed December-2023].

[17] JD Bruce. The mini-blockchain scheme. *White paper*, 1:1–10, 2014.

[18] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[19] Ting Chen, Zihao Li, Yufei Zhang, Xiapu Luo, Ang Chen, Kun Yang, Bin Hu, Tong Zhu, Shifang Deng, Teng Hu, et al. Dataether: Data exploration framework for ethereum. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1369–1380. IEEE, 2019.

[20] Ting Chen, Zihao Li, Yuxiao Zhu, Jiachi Chen, Xiapu Luo, John Chi-Shing Lui, Xiaodong Lin, and Xiaosong Zhang. Understanding ethereum via graph analysis. *ACM Transactions on Internet Technology (TOIT)*, 20(2):1–32, 2020.

[21] Ting Chen, Yufei Zhang, Zihao Li, Xiapu Luo, Ting Wang, Rong Cao, Xiuzhuo Xiao, and Xiaosong Zhang. Tokenscope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, pages 1503–1520, 2019.

[22] Zehao Chen, Bingzhe Li, Xiaojun Cai, Zhiping Jia, Zhaoyan Shen, Yi Wang, and Zili Shao. Block-lsm: An ether-aware block-ordered lsm-tree based key-value storage engine. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*, pages 25–32. IEEE, 2021.

[23] Jemin Andrew Choi, Sidi Mohamed Beillahi, Peilun Li, Andreas Veneris, and Fan Long. Lmpts: Eliminating storage bottlenecks for processing blockchain transactions. In *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–9, 2022.

[24] Xiaohai Dai, Jiang Xiao, Wenhui Yang, Chaofan Wang, and Hai Jin. Jidar: A jigsaw-like data reduction approach without trust assumptions for bitcoin system. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1317–1326. IEEE, 2019.

[25] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 international conference on management of data*, pages 123–140, 2019.

[26] Dipanjan Das, Priyanka Bose, Nicola Ruaro, Christopher Kruegel, and Giovanni Vigna. Understanding security issues in the nft ecosystem. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 667–681, 2022.

[27] Xing Fan, Baoning Niu, and Zhenliang Liu. Scalable blockchain storage systems: research progress and models. *Computing*, 104(6):1497–1524, 2022.

[28] Arthur Gervais, Ghassan O Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. On the security and performance of proof of work blockchains. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 3–16, 2016.

[29] Yeonsoo Kim, Seongho Jeong, Kamil Jezek, Bernd Burgstaller, and Bernhard Scholz. An off-the-chain execution environment for scalable testing and profiling of smart contracts. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 565–579, 2021.

[30] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE symposium on security and privacy (SP)*, pages 583–598. IEEE, 2018.

[31] John Kolb, Moustafa AbdelBaky, Randy H Katz, and David E Culler. Core concepts, challenges, and future directions in blockchain: A centralized tutorial. *ACM Computing Surveys (CSUR)*, 53(1):1–39, 2020.

[32] Ao Li, Jemin Andrew Choi, and Fan Long. Securing smart contract with runtime validation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 438–453, 2020.

[33] Chenxing Li, Sidi Mohamed Beillahi, Guang Yang, Ming Wu, Wei Xu, and Fan Long. LVMT: An efficient authenticated storage for blockchain. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 135–153, 2023.

[34] Yang Li, Kai Zheng, Ying Yan, Qi Liu, and Xiaofang Zhou. Etherql: a query layer for blockchain system. In *Database Systems for Advanced Applications: 22nd International Conference, DASFAA 2017, Suzhou, China, March 27-30, 2017, Proceedings, Part II 22*, pages 556–567. Springer, 2017.

[35] Zihao Li, Jianfeng Li, Zheyuan He, Xiapu Luo, Ting Wang, Xiaoze Ni, Wenwu Yang, Xi Chen, and Ting Chen. Demystifying defi mev activities in flashbots bundle. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 165–179, 2023.

[36] Yinqiu Liu, Kun Wang, Yun Lin, and Wenyao Xu. Lightchain: a lightweight blockchain system for industrial internet of things. *IEEE Transactions on Industrial Informatics*, 15(6):3571–3581, 2019.

[37] Zhongtang Luo, Rohan Murukutla, and Aniket Kate. Last mile of blockchains: Rpc and node-as-a-service. In *2022 IEEE 4th International Conference on Trust, Privacy and Security in Intelligent Systems, and Applications (TPS-ISA)*, pages 305–311, 2022.

[38] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 17–30, 2016.

[39] Robert McLaughlin, Christopher Kruegel, and Giovanni Vigna. A large scale study of the ethereum arbitrage ecosystem. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3295–3312, 2023.

[40] Gianmaria Del Monte, Diego Pennino, and Maurizio Pizzonia. Scaling blockchains without giving up decentralization and security: A solution to the blockchain scalability trilemma. In *Proceedings of the 3rd Workshop on Cryptocurrencies and Blockchains for Distributed Systems*, pages 71–76, 2020.

[41] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33:351–385, 1996.

[42] Asutosh Palai, Meet Vora, and Aashaka Shah. Empowering light nodes in blockchains with block summarization. In *2018 9th IFIP international conference on new technologies, mobility and security (NTMS)*, pages 1–5. IEEE, 2018.

[43] Soujanya Ponnapalli, Aashaka Shah, Souvik Banerjee, Dahlia Malkhi, Amy Tai, Vijay Chidambaram, and Michael Wei. RainBlock: Faster transaction processing in public blockchains. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 333–347, 2021.

[44] Kaihua Qin, Liyi Zhou, and Arthur Gervais. Quantifying blockchain extractable value: How dark is the forest? In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 198–214. IEEE, 2022.

[45] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 497–514, 2017.

[46] Fabian Ritz and Alf Zugenmaier. The impact of uncle rewards on selfish mining in ethereum. In *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 50–57. IEEE, 2018.

[47] Abdurrashid Ibrahim Sanka and Ray CC Cheung. A systematic review of blockchain scalability: Issues, solutions, analysis and future research. *Journal of Network and Computer Applications*, 195:103232, 2021.

[48] Caspar Schwarz-Schilling, Joachim Neu, Barnabé Monnot, Aditya Asgaonkar, Ertem Nusret Tas, and David Tse. Three attacks on proof-of-stake ethereum. In *International Conference on Financial Cryptography and Data Security*, pages 560–576. Springer, 2022.

[49] Yuechen Tao, Bo Li, Jingjie Jiang, Hok Chu Ng, Cong Wang, and Baochun Li. On sharding open blockchains with smart contracts. In *2020 IEEE 36th international conference on data engineering (ICDE)*, pages 1357–1368. IEEE, 2020.

[50] Haixin Wang, Cheng Xu, Ce Zhang, Jianliang Xu, Zhe Peng, and Jian Pei. vchain+: Optimizing verifiable blockchain boolean range queries. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 1927–1940. IEEE, 2022.

[51] Ye Wang, Yan Chen, Haotian Wu, Liyi Zhou, Shuiguang Deng, and Roger Wattenhofer. Cyclic arbitrage in decentralized exchanges. In *Companion Proceedings of the Web Conference 2022*, pages 12–19, 2022.

[52] Qian Wei, Bingzhe Li, Wanli Chang, Zhiping Jia, Zhaoyan Shen, and Zili Shao. A survey of blockchain data management systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 21(3):1–28, 2022.

[53] Sam Werner, Daniel Perez, Lewis Gudgeon, Ariah Klages-Mundt, Dominik Harz, and William Knottenbelt. Sok: Decentralized finance (defi). In *Proceedings of the 4th ACM Conference on Advances in Financial Technologies*, pages 30–46, 2022.

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

[55] Siwei Wu, Lei Wu, Yajin Zhou, Runhuai Li, Zhi Wang, Xiapu Luo, Cong Wang, and Kui Ren. Time-travel investigation: toward building a scalable attack detection framework on ethereum. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(3):1–33, 2022.

[56] Junfeng Xie, F Richard Yu, Tao Huang, Renchao Xie, Jiang Liu, and Yunjie Liu. A survey on the scalability of blockchain systems. *IEEE network*, 33(5):166–173, 2019.

[57] Cheng Xu, Ce Zhang, and Jianliang Xu. vchain: Enabling verifiable boolean range queries over blockchain databases. In *Proceedings of the 2019 international conference on management of data*, pages 141–158, 2019.

[58] Yibin Xu. Section-blockchain: A storage reduced blockchain protocol, the foundation of an autotrophic decentralized storage architecture. In *2018 23rd International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 115–125. IEEE, 2018.

[59] Mingxi Ye, Yuhong Nan, Zibin Zheng, Dongpeng Wu, and Huizhong Li. Detecting state inconsistency bugs in dapps via on-chain transaction replay and fuzzing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 298–309, 2023.

[60] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 931–948, 2018.

[61] Ce Zhang, Cheng Xu, Haibo Hu, and Jianliang Xu. COLE: A column-based learned storage for blockchain systems. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, pages 329–345, 2024.

[62] Ce Zhang, Cheng Xu, Haixin Wang, Jianliang Xu, and Byron Choi. Authenticated keyword search in scalable hybrid-storage blockchains. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 996–1007, 2021.

[63] Mengya Zhang, Xiaokuan Zhang, Yinqian Zhang, and Zhiqiang Lin. Txspector: Uncovering attacks in ethereum from transactions. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2775–2792, 2020.

[64] Zhuo Zhang, Zhiqiang Lin, Marcelo Morales, Xiangyu Zhang, and Kaiyuan Zhang. Your exploit is mine: Instantly synthesizing counterattack smart contract. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1757–1774, 2023.

[65] Lin Zhao, Sourav Sen Gupta, Arijit Khan, and Robby Luo. Temporal analysis of the entire ethereum blockchain network. In *Proceedings of the Web Conference 2021*, pages 2258–2269, 2021.

[66] Peilin Zheng, Zibin Zheng, Jiajing Wu, and Hong-Ning Dai. Xblock-eth: Extracting and exploring blockchain data from ethereum. *IEEE Open Journal of the Computer Society*, 1:95–106, 2020.

[67] Qiuhong Zheng, Yi Li, Ping Chen, and Xinghua Dong. An innovative ipfs-based storage model for blockchain. In *2018 IEEE/WIC/ACM international conference on web intelligence (WI)*, pages 704–708. IEEE, 2018.

[68] Shunfan Zhou, Malte Möser, Zhemin Yang, Ben Adida, Thorsten Holz, Jie Xiang, Steven Goldfeder, Yinzhi Cao, Martin Plattner, Xiaojun Qin, et al. An ever-evolving game: Evaluation of real-world attacks and defenses in ethereum ecosystem. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2793–2810, 2020.