Fine-Grained, Secure and Efficient Data Provenance on Blockchain Systems

Pingcheng Ruan[†], Gang Chen[§], Tien Tuan Anh Dinh[†], Qian Lin[†], Beng Chin Ooi[†], Meihui Zhang[‡]
[†]*National University of Singapore* [§]*Zhejiang University* [‡]*Beijing Institute of Technology*
[†]{ruanpc, dinhtta, linqian, ooibc}@comp.nus.edu.sg [§]cg@zju.edu.cn [‡]meihui_zhang@bit.edu.cn

ABSTRACT

The success of Bitcoin and other cryptocurrencies bring enormous interest to blockchains. A blockchain system implements a tamper-evident ledger for recording transactions that modify some global states. The system captures entire evolution history of the states. The management of that history, also known as data provenance or lineage, has been studied extensively in database systems. However, querying data history in existing blockchains can only be done by replaying all transactions. This approach is applicable to large-scale, offline analysis, but is not suitable for online transaction processing.

We present *LineageChain*, a fine-grained, secure and efficient provenance system for blockchains. *LineageChain* exposes provenance information to smart contracts via simple and elegant interfaces, thereby enabling a new class of blockchain applications whose execution logics depend on provenance information at runtime. *LineageChain* captures provenance during contract execution, and efficiently stores it in a Merkle tree. *LineageChain* provides a novel skip list index designed for supporting efficient provenance query processing. We have implemented *LineageChain* on top of Hyperledger and a blockchain-optimized storage system called ForkBase. Our extensive evaluation of *LineageChain* demonstrates its benefits to the new class of blockchain applications, its efficient query, and its small storage overhead.

PVLDB Reference Format:
Pingcheng Ruan, Gang Chen, Tien Tuan Anh Dinh, Qian Lin, Beng Chin Ooi, Meihui Zhang. Fine-Grained, Secure and Efficient Data Provenance on Blockchain Systems. *PVLDB*, 12(9): 975-988, 2019.
DOI: https://doi.org/10.14778/3329772.3329775

1. INTRODUCTION

Blockchains are capturing attention from both academia and industry. A blockchain is a chain of blocks, in which each block contains many transactions and is linked with the previous block via a hash pointer. It was firstly introduced in Bitcoin [27], where Satoshi Nakamoto employs it to batch cryptocurrency transactions. Often referred to as decentralized ledger, the chain ensures integrity of the complete transaction history. It is replicated over a peer-to-peer (P2P) network, and a distributed consensus protocol, namely Proof-of-Work (PoW), is used to ensure that honest nodes in the network have the same ledger. More recent blockchains, for instance Ethereum [1] and Hyperledger [2], extend the original design to support applications beyond cryptocurrencies. In particular, they add smart contracts which encode arbitrary, Turing-complete computation on top of the blockchain. A smart contract has its states stored on the blockchain, and the states are modified via transactions that invoke the contract.

Blockchains are disrupting many industries, including finance [34, 29], supply chain [24, 35], and healthcare [4]. These industries are exploiting two distinct advantages of blockchains over traditional data management systems. First, a blockchain is decentralized, which allows mutually distrusting parties to manage the data together instead of trusting a single party. Second, the blockchain provides integrity protection (tamper evidence) to all transactions recorded in the ledger. In other words, the complete transaction history is secure.

The management of data history, or data provenance, has been extensively studied in databases, and many systems have been designed to support provenance [13, 14, 8, 30, 5, 36]. In the context of blockchain, there is explicit, but only coarse-grained support for data provenance. In particular, the blockchain can be seen as having some states (with known initial values), and every transaction moves the system to new states. The evolution history of the states (or provenance) can be securely and completely reconstructed by replaying all transactions. This reconstruction can be done during offline analysis. During contract execution (or runtime), however, no provenance information is safely available to smart contracts. In other words, smart contracts cannot access historical blockchain states in a tamper-evident manner. The lack of secure, fine-grained, runtime access to provenance therefore restricts the expressiveness of the business logic the contract can encode.

Consider an example smart contract shown in Figure 1, which contains a method for transferring a number of tokens from one user to another. Suppose user A wants to send tokens to B based on the latter's historical balance in recent months. For example, A only sends token if B's average balance per day is more than t. It is not currently possible to

Proceedings of the VLDB Endowment, Vol. 12, No. 9
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3329772.3329775

```
contract Token {
    method Transfer(sender, recipient, amount) {
        bal1 = gState[sender];
        bal2 = gState[recipient];
        if (amount < bal1) {
            gState[sender] = bal1 - amount;
            gState[recipient] = bal2 + amount;
} } }
```

Figure 1: A smart contract that manages for token management.

write a contract method for this operation. To work around this, A needs to first compute the historical balance of B by querying and replaying all on-chain transactions, then based on the result issues the `Transfer` transaction. Besides performance overhead incurred from multiple interactions with the blockchain, this approach is not *safe*: it fails to achieve transaction serializability. In particular, suppose A issues the `Transfer` transaction tx based on its computation of B's historical balance. But before tx is received by the blockchain, another transaction is committed such that B's average balance becomes $t' < t$. Consequently, when tx is later committed, it will have been based on stale state, and therefore fails to meet the intended business logic. In blockchains with native currencies, serializability violation can be exploited for Transaction-Ordering attacks that cause substantial financial loss to the users [25].

In this paper, we design and implement a fine-grained, secure and efficient provenance system for blockchains, called *LineageChain*. In particular, we aim to enable a new class of smart contracts that can access provenance information at runtime. Although our goal is similar to that of existing works in adding provenance to databases [5, 35, 31], we face three unique challenges due to the nature of blockchain. First, there is a lack of data operators whose semantics capture provenance in the form of input-output dependency. More specifically, for general data management workloads (i.e., non-cryptocurrency), current blockchains expose only generic operators, for example, `put` and `get` of key-value tuples. These operators do not have input-output dependency. In contrast, relational databases operators such as `map`, `join`, `union`, are defined as relations between input and output, which clearly capture their dependencies. To overcome this lack of provenance-friendly operators, we instrument blockchain runtime to record read-write dependency of all the states used in any contract invocation, which is then passed to a user-defined method that specifies which dependency to be persisted.

The second challenge is that blockchains assume an adversarial environment, therefore any captured provenance must be made tamper evident. To address this, we store provenance in a Merkle tree data structure that also allows for efficient verification. The final challenge is to ensure that provenance queries are fast, because a large execution overhead is undesirable due to the Verifier's Dilemma [26]. To address this challenge, we design a novel skip list index that is optimized for provenance queries. The index incurs small storage overhead, and its performance is independent of the number of blocks in the blockchain.

In summary, we make the following contributions:

- We introduce a system, called *LineageChain* that efficiently captures fine-grained provenance for blockchains. It stores provenance securely, and exposes simple access interface to smart contracts.
- We design a novel index optimized for querying blockchain provenance. The index incurs small storage overhead, and its performance is independent of the blockchain size. It is adapted from the skip list but we completely remove the randomness to fit for deterministic blockchains.
- We implement *LineageChain* for Hyperledger [2]. Our implementation builds on top of ForkBase, a blockchain-optimized storage [37]. We conduct extensive evaluation of *LineageChain*. The results demonstrate its benefits to provenance-dependent applications, and its efficient query and small storage overhead.

LineageChain is a component of our Hyperledger++ system [3], for which we improve Hyperledger's execution and storage layer for the secure runtime provenance support. Elsewhere, we have addressed the consensus bottleneck by applying sharding efficiently and exploiting trusted hardware to scale out system horizontally, to substantially improve the system throughput [15]. We have also improved the storage efficiency by designing a tamper-evident storage engine that supports efficient forking called Forkbase. We are currently incorporating smart contract verification to enhance the correctness of smart contracts.

The remainder of the paper is organized as follows. Section 2 provides background on blockchains. Section 3 describes our design for capturing provenance, and the interface exposed to smart contracts. Section 4 discusses how we store provenance, and Section 5 describes our new index. Section 6 presents our implementation. Section 7 reports the performance of *LineageChain*. Section 8 discusses related work, and Section 9 concludes this work.

2. BACKGROUND AND OVERVIEW

In this section, we present relevant background on blockchain systems [18, 7], and design choices that affect index structure requirements. Following which, we present an overview of *LineageChain*.

2.1 Blockchain Systems

A blockchain system consists of multiple nodes that do not trust each other. Current blockchains can be broadly classified as permissionless or permissioned. In the former, any node can join or leave the network. In the latter, membership is strictly controlled, and a node must be authenticated and granted permission to join the network.

Consensus. Except for a few permissioned blockchains with high auditability, most blockchains assume the Byzantine failure model, in which faulty nodes behave arbitrarily. Under this hostile environment. They use a Byzantine Fault Tolerance (BFT) consensus protocol to ensure that honest nodes agree on the same states. Examples of BFT protocols include Proof-of-work (PoW) which is used in Bitcoin [27], and PBFT [11] which is used in Hyperledger. Classic BFT protocols, such as PBFT, guarantee that honest nodes have the identical chain of blocks. PoW and its variants, on the other hand, allow for inconsistency because the chain can have forks. These protocols handle forks by deterministically selecting one branch over the other. For example, in PoW the longest branch is selected.

Data model. Different blockchains adopt different data models for their states. Bitcoin' states are unspent coins modeled as Unspent Transaction Outputs (UTXOs), which consists of outputs of transactions that have not been used as inputs to another transaction. The UTXO model lends itself to simple transaction verification, because nodes only need to check that the transaction output has not been used in the past. More recent blockchains, namely Ethereum and Hyperledger, support general states that can be modified arbitrarily by smart contracts. They adopt an account-based data model, in which each account has its own local states stored on the blockchain. A smart contract transaction can write arbitrary data to the storage. This flexible data model comes at the cost of integrity protection and verification of the account states. In this paper, we focus on the account-based data model.

Block structure. A block in the blockchain stores the transactions and the global states. The block header contains the following fields.

- `PreviousBlockHash`: reference to the previous block in the chain.
- `Nonce`: used for checking validity of the block. In PoW consensus, `Nonce` is the solution to the PoW puzzle.
- `TransactionDigest`: used for integrity protection of the list of transactions in the block.
- `StateDigest`: used for integrity protection of the global states after executing the block transactions.

Both `TransactionDigest` and `StateDigest` are Merkle-tree roots. They allow for efficient block transfer, in which the block headers and block content can be decoupled and transferred separately. In addition, they enable efficient verification of transactions and states.

Algorithm 1: Block verification in blockchain

```
   Input: A block Blk received from the network.
   Input: The global state gState, which maps state
          identifiers to their values.
   Output: True if the block is valid, False otherwise.
   // Step 1: Verify Nonce (PoW only).
 1 if checkNonce(Blk.Header.Nonce) then
 2 |   return False;
   // Step 2: Verify transactions.
 3 txnDigest = computeDigest(Blk.Transactions);
 4 if txnDigest != Blk.Header.TransactionDigest then
 5 |   return False;
   // Step 3: Tentatively execute transactions.
 6 oldState = gState;
 7 allUpdates = [];
 8 for txn in Blk.Header.Transactions do
   |   // Buffer the changes
 9 |   updates = execute(txn);
10 |   allUpdates.Append(update);
11 gState.apply(allUpdates);
   // Step 4: Verify new state.
12 stateDigest = computeDigest(state);
13 if stateDigest != Blk.Header.StateDigest then
   |   // Rollback
14 |   gState = oldState;
15 |   return False;
16 else
17 |   return True;
```

Block verification. Algorithm 1 illustrates how a node uses the block header to verify if a block it receives from the network is valid. If the block is valid, it is appended to the chain. When PoW is used for consensus, the node first checks if `Nonce` is the correct solution to the PoW puzzle. This step is skipped if a deterministic consensus protocol, such as PBFT, is used. Next, it checks if the list of transactions has not been tampered with, by computing and verifying `TransactionDigest` from the list. It then tentatively re-executes the included transactions. During execution, the states are accessed via some index structures. After the execution, the node checks if the resulting states match with `StateDigest`. If they do, the block is considered valid and the new states are committed to the storage. Otherwise, the states are rolled back to those before execution. We note that Algorithm 1 takes as input an object *gState* that represents the global states. If the blockchain does not have forks, e.g., Hyperledger, this object is the *latest* states. However, when there are forks, e.g., in Ethereum, this object may refer to the global states at a point in the past.

2.2 State Organization

The most important feature of blockchain is the guarantee of data integrity, which implies that the global states must be tamper evident. The block verification algorithm above is crucial for the security of blockchain. We note that the algorithm requires access to all history snapshots of the states, as well as the ability to update the states in batch. These requirements present new challenges in designing an index structure for organizing blockchain states. In particular, traditional database indices such as B+ tree cannot be used. We now elaborate on the requirements for a blockchain index, and explain how they are met in Ethereum and Hyperledger. *LineageChain* builds on existing blockchain indices to ensure security for the captured provenance.

Tamper evidence. A user may want to read some states without downloading and executing all the transactions. Thus, the index structure must be able to generate an integrity proof for any state. In addition, the index must provide a unique digest for the global states, so that nodes can quickly check if the post-execution states are identical across the network.

Incremental update. The size of global states in a typical blockchain application is large, but one block only updates a small part of the states. For example, some states may be updated at every block, whereas other may be updated much more infrequently. Because the index must be updated at every block, it must be efficient at handling incremental updates.

Snapshot. A snapshot of the index, as well as of the global states, must be made at every block. This is crucial to realize the immutability property of blockchain which allows users to read any historical states. It is also important for block verification. As explained earlier, when a new block is received that creates a fork, an old snapshot of the state must be used as input for verification. Even when the blockchain allows no forks, snapshots enable roll-back when the received block is found to be invalid after execution (step 4 in Algorithm 1).

Existing blockchains use indices that are based on Merkle tree. In particular, Ethereum implements Merkle Patricia Trie (MPT), and Hyperledger implements Merkle Bucket

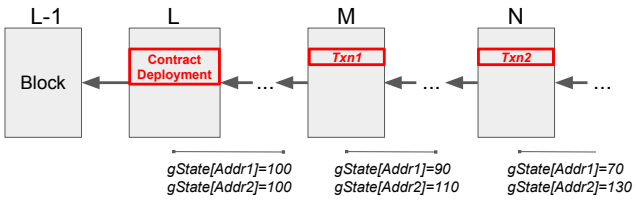


Figure 2: The example ledger with the corresponding *gState* between the block interval

Tree (MBT). In a Merkle tree, content of the parent node is recursively defined by those of the child nodes. The root node uniquely identifies the content of all the leaf nodes. A proof of integrity can be efficiently constructed without reading the entire tree. Therefore, the Merkle tree meets the first requirement. This structure is also suitable for incremental updates (second requirement), because only the nodes affected by the update need to be changed. To support efficient snapshots, an update in the Merkle tree recursively creates new nodes in the path affected by the change. The new root then serves as index of the new snapshot, and is then included in the block header.

2.3 LineageChain Overview

Given a smart contract on an existing blockchain, *LineageChain* enriches it with fine-grained, secure and efficient provenance as follows. First, the contract can implement a helper method to define the exact provenance information to be captured at every contract invocation. By default, all read-write dependencies of all the states are recorded. Second, new methods can be added that make use of provenance at runtime. As far as a contract developer is concerned, these are the only two changes from the existing, non-provenance blockchain. The captured information is then stored in an enhanced blockchain storage that ensures efficient tracking and tamper evidence of provenance. On top of this storage, we build a skip list index to support fast provenance queries. These changes to the blockchain storage are invisible to the contract developer.

3. FINE-GRAINED PROVENANCE

In this section, we describe our approach to capture provenance during smart contract execution. We present APIs that allow the contract to query provenance at runtime.

Running example. Throughout this section, we use as running example the token smart contract shown in Figure 1. Figure 2 depicts how the global states are modified by the contract. In particular, the contract is deployed at block L^{th} in the blockchain. Two addresses *Addr1* and *Addr2* are initialized with 100 tokens. Two transactions *Txn1* and *Txn2* that transfer tokens between the two addresses are committed at block M and N respectively. The value of *Addr1* is 100 from block R to block $M-1$, 90 from block M to $N-1$, and 70 from block N. The global state *gState* is essentially a map of addresses to their values.

3.1 Capturing Provenance

Blockchains support only a small set of data operators for general workloads, namely `read` and `write`. These operators are not provenance friendly, in the sense that they do not capture any data association (input-output dependency). In contrast, relational databases or big data systems have many provenance-friendly operators, such as `map`, `reduce` and `join`, whose semantics meaningfully capture the association. For instance, the output of `join` is clearly derived from (or is dependent on) the input data.

```
contract Token {
    method Transfer(...){...} // as above
    method prov_helper(name, reads, writes) {
      if name == "Transfer" {
        for (id,value) in writes {
            if (reads[id] < value) {
                recipient = id;
            } else {sender = id; }
        }
        // dependency list with a
        // single element.
        dep = [sender];
        return {recipient:dep};
      }
      ...
    }
}
```

Figure 3: The provenance helper method for *Token* contract. It defines dependency between the sender identifier and recipient identifier. This method is invoke after every invocation of the Token contract.

In *LineageChain*, every contract method can be made provenance-friendly via a *helper* method. More specifically, during transaction execution, *LineageChain* collects the identifiers and values of the accessed states, i.e., ones used in `read` and `write` operations. The results are a read set `reads` and write set `writes`. For *Txn1*, `reads` = $\{Addr1 : 100, Addr2 : 100\}$, and `writes` = $\{Addr1 : 90, Addr2 : 110\}$. After the execution finishes, these sets are passed to a user-defined method `prov_helper`, together with the name of the contract method. `prov_helper` has the following signature:

```
method prov_helper(name: string,
                   reads: map(string, byte[]),
                   writes: map(string, byte[]))
       returns map(string, string[]);
```

It returns a set of dependencies based on the input read and write sets. Figure 3 shows implementation of the helper method for the Token contract. It first computes the identifier of the sender and recipient from the read and write sets. Specifically, the identifier whose value in `writes` is lower than that in `reads` is the sender, and the opposite is true for the recipient. It then returns a dependency set of a single element: the recipient-sender dependency. In our example, for *Txn1*, this method returns $\{Addr2 : [Addr1]\}$

LineageChain ensures that `prov_helper` is invoked immediately after every successfully contract execution. If the method is left empty, *LineageChain* uses all identifiers in the read set as dependency of each identifier in the write set. Interested readers may observe that the vanilla Hyperledger already computes for the read/write set during the endorsement phase. Orthogonal to ours, they are internally used for the concurrency control to achieve one-copy serializability. Instead, we allow contract developers to capture for their application-level provenance.

3.2 Smart Contract APIs

Current smart contracts can only safely access the *latest* blockchain state. In Hyperledger, for example, the `get(k)` operation returns the last value of k that is written or being batched. In Ethereum, on the other hand, when a smart contract reads a value of k at block b, the system considers the snapshot of states at block $b-1$ as the latest states. Although there may exist a block $b' > b$ on a different branch, the smart contract always treats what returned from the storage layer as the latest state.

The main limitation of the current APIs is that the smart contract cannot tamper-evidently read previous values of a state. Instead, the contract has to explicit track historical versions, for example by maintaining a list of versions for every state. This approach is costly both in terms of storage and computation. *LineageChain* addresses this limitation with three additional smart contract APIs.

- `Hist(stateID, [blockNum])`: returns the tuple `(val, blkStart, txnID)` where `val` is the value of `stateID` at block `blockNum`. If `blockNum` is not specified, the latest block is used. `txID` is the transaction that sets `stateID` to `val`, and `blkStart` is the block number at which `txID` is executed.
- `Backward(stateID, blkNum)`: returns a list of tuples `(depStateID, depBlkNum)` where `depStateID` is the dependency state of `stateID` at block `blkNum`. `depBlkNum` is the block number at which the value of `depStateID` is set. In our example, `Backward(Addr2, N)` returns `(Addr1, M)`.
- `Forward(stateID, blkNum)`: similar to the `Backward` API, but returns the states of which `stateID` is a dependency. For example, `Forward(Addr1, L)` returnss `(Addr2, M)`.

Figure 4 demonstrates how the above APIs are used to express smart contract logics that are currently impossible in the secure manner. (The vanilla Hyperledger optionally provides a *historyDB* for the historical query. But implemented from the flat storage, it does not provide the tamper-evidence guarantee, which is our major contribution.) We add two additional methods to the original contract, both of which use the new APIs. The `Refund` method examines an account's average balance in the recent month and makes the refund accordingly. The `Blacklist` method marks an address as blacklisted if one of its last 5 transactions is with a blacklisted address.

```
contract Token {
  ...
  method Refund(addr) {
    blk := last block in the ledger
    first_blk := first block in this month
    sum = count = 0;
    while (first_blk < blk) {
      val, startBlk, txnID = Hist(addr, blk);
      blk = startBlk - 1;
      sum += val;
      count += 1;
    }
    avg = sum / count;
    refund_amount := refund amount based on avg
    gState[addr] += refund_amount;
  }

  method Blacklist(addr) {
    blk := last block in the ledger
    blacklisted = false;
    iterate 5 times {
      val, startBlk, txnID = Hist(addr, blk);
      for (depAddr, depBlk)
            in (Backward(addr, startBlk)
             or Forward(addr, startBlk)) {
        if depAddr in gState["blacklist"] {
          gState["blacklist"].append(addr);
          return;
        }
      }
      blk = startBlk - 1;
    }
  }
}
```

Figure 4: Smart contract with provenance-dependent methods.

4. SECURE PROVENANCE STORAGE

In this section, we discuss how *LineageChain* enhances existing blockchain storage layer to provide efficient tracking and tamper evidence for the captured provenance. Our key insight is to reorganize the flat leaf nodes in the original Merkle tree into a Merkle DAG. We first describe the Merkle DAG structure, then discuss its properties. Finally, we explain how to exploit the blockchain execution model to support forward provenance tracking.

4.1 Merkle DAG

Let k be the unique identifier of a blockchain state, whose evolution history is expected to be tracked. Let v be the unique version number that identifies the state in its evolution history. When the state at version v is updated, the new version v' is strictly greater than v. In *LineageChain*, we directly use the block number as its version v. Let $s_{k,v}$ denote the value of the state with identifier k at version v. We drop the subscripts if the meaning of k and v are not important. For any $k \neq k'$ and $v \neq v'$, $s_{k,v}$ and $s_{k',v'}$ represent the values of two different states at different versions. s_k^b represents the state value with identifier k at its latest version before block b. In our example, for $k = Addr1$ and $v = M$, $s_{k,v} = 90$.

Definition 1. *A transaction, identified by tid which is strictly increasing, reads a set of input states S^i_{tid} and updates a set of output states S^o_{tid}. A valid transaction satisfies the following properties:*

$$\forall s_{k_1,v_1}, s_{k_2,v_2} \in S^o_{tid}. \quad k_1 \neq k_2 \wedge v_1 = v_2 \tag{1}$$

$$\forall s_{k_1,v_1} \in S^i_{tid}, s_{k_2,v_2} \in S^o_{tid}. \quad v_1 < v_2 \tag{2}$$

$$\forall s_{k,v} \in S^i_{tid}, s_{k,v'} \in S^i_{tid'}. \quad tid < tid' \Rightarrow v \leq v'. \tag{3}$$

$$tid \neq tid' \Rightarrow S^o_{tid} \cap S^o_{tid'} = \emptyset \tag{4}$$

Property (1) means that the versions of all output states of a transaction are identical, because they are updated by the same transaction in the same block. Property (2) implies the version of any input state is strictly lower than that of the output version. This makes sense because the blockchain establishes a total order over the transactions, and because the input states can only be updated in previous transactions. Property (3) specifies that, for all the states with the same identifier, the input of later transactions can never have an earlier version. This ensures the input state of any transaction must be up-to-date during execution time. Finally, Property (4) means that every state update is unique.

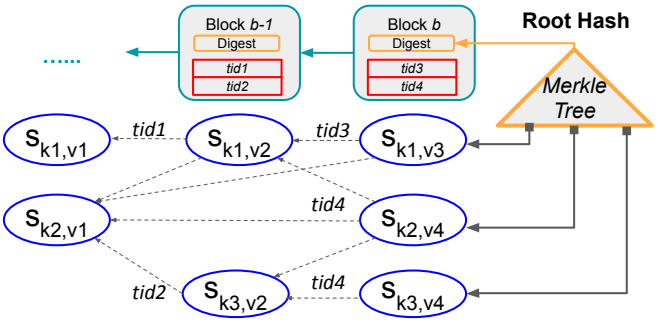


Figure 5: A Merkle DAG for storing provenance. s_{k_2,v_4} and s_{k_3,v_4} updated by the same transaction (tid_4), but their dependencies are different. b contains two transactions, tid_3 and tid_4. Its latest states include s_{k_1,v_3}, s_{k_2,v_4}, s_{k_3,v_4}, from which a Merkle tree is built.

Definition 2. *The dependency of state s is a subset of the input states of the transaction that outputs s. More specifically:*

$$dep(s) \subset S^i_{tid} \text{ where } s \in S^o_{tid}.$$

We note that *dep*, which is returned by `prov_helper` method, is only a subset of the read set.

Definition 3. *The entry $E_{s_{k,v}}$ of the state $s_{k,v}$ is a tuple containing the current version, the state value, and the hashes of the entries of its dependent state. More specifically:*

$$E_{s_{k,v}} = \langle v, s_{k,v}, \{hash(E_{s'}) | s' \in dep(s_{k,v})\}\rangle$$

An entry uniquely identifies a state. In *LineageChain*, we associate each entry with its corresponding hash.

Definition 4. *The set of latest states at block b, denoted as $S_{latest,b}$ is defined as:*

$$S_{latest,b} = \bigcup_k \{s^b_k\}$$

Let U_b be the updated states in block b. We can compute $S_{latest,b}$ by recursively combining U_b with $S_{latest,b-1} \setminus U_b$.

Definition 5. *χ_b is the root of a Merkle tree built on the map S_b where*

$$S_b = \{k : hash(E_{s^b_k}) | \forall s^b_k \in S_{latest,b}\}.$$

LineageChain stores χ_b as the state digest in the block header.

4.2 Discussion

Our new Merkle DAG can be easily integrated to existing blockchain index structures. In particular, existing Merkle index such as MPT stores state values directly at the leaves, whereas the Merkle DAG in *LineageChain* stores the entry hashes of the latest state versions at the leaves. By adding one more level of indirection, we maintain the three properties of the index (tamper evidence, incremental update and snapshot), while enhancing it with the ability to traverse the DAG to extract fine-grained provenance information.

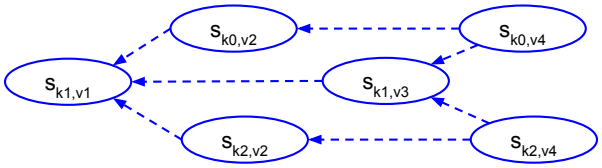


Figure 6: Forward tracking support. After s_{k_1,v_1} is updated, there can only be s_{k_0,v_2} and s_{k_2,v_2} that are dependent on s_{k_1,v_1}. Future states can only depend on s_{k_1,v_3}. Forward pointers of s_{k_1,v_1} are stored in the entry of s_{k_1,v_3}.

Recall that the state entry hash captures the entire evolution history of the state. Since this hash is protected by the Merkle index for tamper evidence, so is the state history. In other words, we add integrity protection for provenance without any extra cost to the index structure. For example, suppose a client wants to read a specific version of a state, it first reads the state entry hash at the latest block. This read operation can be verified against tampering, as in existing blockchains. Next, the client traverses the DAG from this hash to read the required version. Because the DAG is tamper evident, the integrity of the read version is guaranteed.

4.3 Support for Forward Tracking

One problem of the above DAG model is that it does not support forward tracking, because the hash pointers only reference *backward* dependencies. When a state is updated, these backward dependencies are permanently established, so that they belong to the immutable derivation history of the state. However, the state can be read by future transaction, therefore its forward dependencies cannot be determined at the time of update.

Our key insight here is that only forward dependencies of the *latest state* are mutable. Once the state is updated, due to the execution model of blockchain smart contract, in which the latest state is always read, forward dependencies of the previous state version becomes permanent. As a result, they can be included into the derivation history. Figure 6 illustrates an example, in which forward dependencies of s_{k_1,v_1} becomes fixed when the state is updated to s_{k_1,v_3}. This is because when the transaction that outputs s_{k_0,v_4} is executed, it reads s_{k_1,v_3} instead of s_{k_1,v_1}.

In *LineageChain*, for each state $s_{k,v}$ at its latest version, we buffer a list of forward pointers to the entries whose dependencies include $s_{k,v}$. We refer to this list as $F_{s_{k,v}}$, which is defined more precisely as follows:

$$F_{s_{k,v}} = \{hash(E_{s'}) | s_{k,v} \in dep(s')\}$$

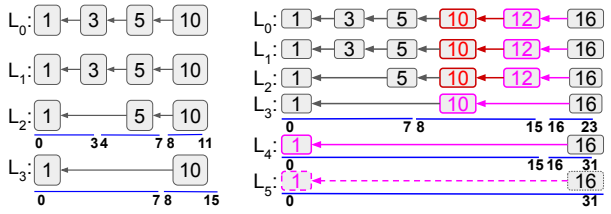
When the state is updated to $s_{k,v'}$ for $v' > v$, we store $F_{s_{k,v}}$ at the entry of $s_{k,v'}$.

5. EFFICIENT PROVENANCE QUERIES

The Merkle DAG structure supports efficient access to the latest state version, since the state index at block b contains pointers to all the latest versions at this block. To read the latest version of s, one simply reads χ_b, follows the index to the entry for s, and then reads the state value from the entry. However, querying an arbitrary version in the DAG is inefficient, because one has to start at the DAG head

```
struct Node {
  Version v;
  Value val;
  List<Version> pre_versions;
  List<Node*> pre_nodes;
}
```

Figure 7: A Node structure that captures a state $s_{k,v}$ with value val



(a) (b)

Figure 8: (a) A DASL containing versions 1, 3, 5 and 10. The base b is 2. The intervals for L_2 and L_3 are shown in blue lines. (b) The new DASL after appending version 12 and 16. L_4 is created when appending version 16. L_5 is created, then discarded.

and traverse a long the edges towards the requested version. Supporting fast version queries is important when the user wants to examine the state history only from a specific version (for auditing purposes, for example). It is also important for provenance-dependent smart contracts because such queries directly affect contract execution time.

In this section, we describe a novel index that facilitates fast version queries. The index is designed for permissioned blockchains. We discuss its efficiency and how to extend it to permisionless blockchains.

5.1 Deterministic Append-Only Skip List

We propose to build an index on top of a state DAG to enable fast version queries. The index has a skip list structure, which we call Deterministic Append-only Skip List (or DASL). It is designed for blockchains, exploiting the fact that the blockchain is append-only, and randomness is not well supported [10]. More specifically, a DASL has two distinct properties compared to a normal skip list. First, it is append-only. The index keys of the appended items, which are versions in our case, are strictly increasing. Second, it is deterministic, that is, the index structure is uniquely determined by the values of the appended items, unlike a stochastic skip list. For ease of explanation, we assume that version numbers are positive integers.

Definition 6. *Let $V_k = \langle v_0, v_1, ...\rangle$ be the sequence of version numbers of states with identifier k, in which $v_i < v_j$ for all $i < j$. A DASL index for k consists of N linked lists $L_0, L_1, .., L_{N-1}$. Let v_j^{i-1} and v_j^i be the versions in the $(j-1)^{th}$ and j^{th} node of list L_i. Let b be the base number, a system-wide parameter. The content of L_i is constructed as follows:*

1) $v_0 \in L_i$

2) Given v_{j-1}^i, v_j^i is the smallest version in V_k such that:

$$\left\lfloor \frac{v_{j-1}^i}{b^i} \right\rfloor < \left\lfloor \frac{v_j^i}{b^i} \right\rfloor \quad (5)$$

Figure 7 shows how DASL is stored with the state in a data structure called Node. This structure (also referred to as node) consists of the state version and value. A node belongs to multiple lists (or levels), hence it maintains a list of pointers to some version numbers, and another list of pointers to other nodes. Both lists are of size N, and the i^{th} entry of a list points to the previous version (or the previous node) of this node in level L_i. For the same key, the version number uniquely identifies the node, hence we use version numbers to refer to the corresponding nodes.

We can view a list as consisting of continuous, non-overlapping intervals of certain sizes. In particular, the j^{th} interval of L_i represents the range $R_j^i = [jb^i, (j+1)b^i)$. Only the smallest version in V_k that falls in this range is included in the list. Figure 8(a) gives an example of a DASL structure with $b = 2$. It can be seen that when the version numbers are sparsely distributed, the lists at lower levels are identical. In this case, b can be increased to create larger intervals which can reduce the overlapping among lower-level lists.

A DASL and a skip list share two properties. First, if a version number appears in L_i, it also appears in L_j where $j < i$. Second, with $b = 2$, suppose the last level that a version appears in is i, then this version's preceding neighbour in L_i appears in L_j where $j > i$. Given these properties, a query for a version in the DASL is executed in the same way as in the skip list. More specifically, the query traverses a high-level list as much as possible, starting from the last version in the last list. It moves to a lower level only if the preceding version in the current list is strictly smaller than the requested version. In DASL, the query for version v_q returns the largest version $v \in V_k$ such that $v \leq v_q$ (the inequality occurs when v_q does not exist). This result represents the value of the state which is visible at time of v_q.

Algorithm 2: DASL Append

Input: version v and last node *last*
Output: previous versions and nodes

```
1  level=0; // list level
2  pre_versions = [];
3  pre_nodes = [];
4  finish = false ;
5  cur = last ;
6  while not finish do
7      l = cur->pre_versions.size() ;
8      if l > 0 then
9          for j=level; j<l; ++j do
10             if cur->version / b^j < v / b^j then
11                 pre_versions.append(cur->version);
12                 pre_nodes.append(cur);
13             else
14                 finish = true;
15                 break;
16         if not finish then
17             cur = cur->pre_versions[l-1];
18             level = l
19     else
           /* We have reached the last level        */
20         finish = true;
21         while cur->version / b^level < v / b^level do
22             ++level;
23             pre_versions.append(cur->version);
24             pre_nodes.append(cur);
25 return pre_version, pre_nodes;
```

We now describe how a new node is appended to DASL. The challenge is to determine the lists that should include the new node. Algorithm 2 details the steps that find the lists, and subsequently the previous versions, of the new node. The key idea is to start from the last node in L_0, then keep increasing the list level until the current node and the new node belong to the same interval (line 9 - 18). Figure 8(b) shows the result of appending a node with version 12 to the original DASL. The algorithm starts at node 10 and moves up to list L_1 and L_2. It stops at L_3 because in this level node 10 and 12 belong to the same interval, i.e., $[8, 16)$. Thus, the new node is appended to list L_0 to L_2. When the algorithm reaches the last level and is still able to append, it creates a new level where node 0 is the first entry and repeats the process (line 21 - 24). In Figure 8(b), when appending version 16, all existing lists can be used. The algorithm then creates L_4 with node 1 and appends the node 16 to it. It also creates level L_5, but then discards it because node 16 will not be appended since it belongs to same interval of $[0, 32)$ with node 1.

5.2 Discussion

Integrating to Merkle DAG The DASL is integrated to the Merkle DAG as follows. The node structure (Figure 7) is stored in the state entry (Definition 3). The node pointers are implemented entry hashes. The Merkle tree structure remains unchanged.

Speed-storage tradeoff As a skip list variant, DASL shares the same lineage space complexity and logarithmic query time complexity. Suppose there are v^* number of versions and the base of DASL is b. The maximum number of required pointers is $\frac{b(v^*-1)}{b-1}$. (There are at most $\lceil \log_b v^* \rceil$ levels and the i-th level takes at most $\lceil \frac{v^*}{b^i} \rceil - 1$ pointers.) Suppose the queried version is v^q and the query distance $d = v^* - v^q$, the maximum number of hops in such query is capped at $2b\lceil \log_b d \rceil$. (A query traversal from the end will undergo two stages, one stage towards lower levels and the other stage back towards upper levels. In each stage, the traversal will take at most b hops on the same list before moving to the next level. And there are at most $\lceil \log_b d \rceil$ levels to traverse.) Hence, b controls the tradeoff between the space overhead and query delay. One benefit of this property is that DASL queries favor more recent versions, i.e. d are small, which is useful for smart contracts that work use recent rather than old versions. Another benefit is that the performance of such recent-version queries does not change as the state history grows.

Extending to permissionless blockchains. We note that DASL incurs storage overhead. The version query also incurs some computation cost, even though it is more efficient with DASL than without it. These costs may be small enough such that they do not affect the performance of a permissioned blockchain, as we demonstrate in Section 7. However, they need to be carefully managed in a permisionless blockchain where any overhead directly translates to monetary cost to the miners. In particular, any additional cost to the miner triggers the Verifier Dilemma [26] and compromises the incentive mechanism of the blockchain.

A malicious user could issue a transaction that references a very old version. Reading earlier versions is more expensive, because there are more hops involved. This overhead is born by all nodes in the network, since every node in the network has to execute the same transaction. Current public blockchains prevent such denial-of-service attack by explicitly charging a fee for each operation in the transaction. In Ethereum, the transaction owner pays for the resource consumption in *gas*. A transaction that writes more data or consumes more CPUs has to pay more *gas*. Thus, rational users are deterred from running too complex transactions on the blockchain.

As DASL consumes resources, its costs must be explicitly accounted for in permissionless blockchains. More specifically, during deployment, the contract owner specifies which states require DASL support. Alternatively, DASL support can be automatically inferred from the contract's source code. The deployment fee should reflect this extra storage cost for DASL. If the fee is too high, the owner can lower it by increasing b. During contract execution, the execution engine must charge the cost of DASL queries to the transaction fee. In particular, a query that requires more hops to find the requested version incurs a higher transaction fee. Users may empirically estimate the hops (as well as the cost) based on the above-derived theoretical upper bound.

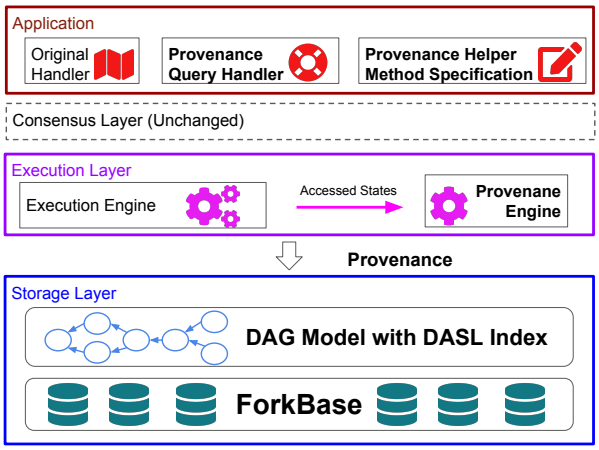


Figure 9: *LineageChain*'s software stack. The original storage layer is replaced with the implementation that supports fine-grained provenance. The original execution layer is instrumented with a provenance capture engine. The application layer contains the new helper method and provenance query APIs. The consensus layer is unchanged.

6. IMPLEMENTATION

In this section, we present our implementation of *LineageChain* based on Hyperledger. We implement *LineageChain* both on Hyperledger v0.6 and v1.3. Although the two blockchains follow different designs, they share the same four logical layers: storage, execution, consensus and application layer [19]. Figure 9 depicts these layers and highlights the changes we make to the original Hyperledger's stack. In particular, we completely replace Hyperledger's storage layer with our implementation of the Merkle DAG and DASL index. This new storage is built on top of ForkBase [37], a state-of-the-art blockchain storage system with support for version tracking. We instrument the original execution engine to record read and write sets during contract execution. At the application layer we add a new helper method and three provenance APIs. The execution engine

Algorithm 3: Provenance update and digest Computation

```
1  fb.Map<id,vid> latest;
2  String branch = "default";
   // Buffered forward pointers
3  Map<id, List<vid» forward;
   Input: id, version, value of the updated state
   Input: ids of the dependent states, dep_ids
4  Function Update(id, version, value, dep_ids):
       /* Backward pointers                          */
5      List<vid> back_vids;
6      for dep_id in dep_ids do
7          back_vid = latest[dep_id];
8          back_vids.push_back(back_vid);
       /* Forward pointers                           */
9      forward_vids = forward[id];
10
       /* Retrieve pointer to last DASL node         */
11     last_vid = latest[id];
       /* Refer DASL Append in Algorithm 2           */
12     pre_versions, pre_vids = DaslAppend(version,
         last_vid);
13     node = new DaslNode{version, pre_versions,
         pre_vids} ;
14     meta = Serialize(back_vids, forward_vids, node) ;
15
       /* Store the updated value                    */
16     new_vid = fb.Put(id, branch, value, meta);
17
       /* Update forward pointers                    */
18     for dep_id in dep_ids do
19         forward[dep_id].push_back(new_vid);
20     forward[id].Clear();
21     latest[id] = new_vid;
22
   Output: The state digest for the committed block
23 Function ComputeDigest():
24     latest_vid = fb.Put("state", branch, latest, nil);
25     return latest_vid;
```

is modified to invoke the helper method after every successful contract execution.

6.1 Storage Layer Implementation

Instead of implementing the storage layer from scratch, we leverage ForkBase for its support of version tracking. We exploit three properties of ForkBase in *LineageChain*. The first is the fork semantics, with which the application can specify a *branch* for the update. Given a branch, ForkBase provides access to the latest value. The second property is tamper evidence, in which every update returns a tamper-evident identifier *vid* which captures the entire evolution history of the updated value. A data object in ForkBase is uniquely identified by the key and *vid*. In *LineageChain*, *vid* is used as the entry hash in the DAG, and as the pointer in DASL. The third property is the rich set of built-in data types including map, list and set.

Algorithm 3 details our implementation for updating states and computing the global state digest when a block is being committed. The update function is invoked with a new state and a list of dependencies. We first prepare the list of backward pointers by retrieving the latest *vids* of the dependent states (line 5-8). Next, we build the pointers for the DASL index, then retrieve the forward pointers of the previous state (line 9). The metadata from these steps are serialized and stored together with the updated value in ForkBase (line 14-16). The result is a new *vid* for the update, which is appended to the list of forward pointers of every dependent state (line 18-20). *vid* is now the latest version (line 21).

The global states are stored in a map object in ForkBase. To compute the global state digest, we simply update the map object with the new *vids* computed for this block. The update operation of the map object, which is built as a Merkle tree in ForkBase, returns a digest *latest_vid* which is then included to the block header. This digest provides tamper evidence for the evolution histories of all the states up to the current block.

6.2 Application and Execution Layer Implementation

In Hyperledger, users write their smart contracts by implementing the `Chaincode` interface. Given a *chaincode*, the execution engine triggers the `Init` and `Invoke` method during deployment and invocation respectively. Both methods take as input an instance of `ChaincodeStubInterface` which supplies relevant context, such as access to the ledger states, to the smart contract.

We add the helper method, called `ProvHelper`, to the `Chaincode` interface. This method's signature, and how to write user-defined provenance rules with it, are explained in Section 3. The execution engine intercepts `PutState` and `GetStates` during execution to record the read set and the write set. It invokes `ProvHelper` when the execution finishes, passing it the `ChaincodeStubInterface`, the name of the method, and the recorded read and write sets. The three new provenance APIs, namely `Hist`, `Backward` and `Forward`, are added to `ChaincodeStubInterface` and therefore are accessible to all contract methods.

7. PERFORMANCE EVALUATION

7.1 Baselines and Experimental Setup

We evaluate *LineageChain* against two baselines. In the first baseline, called *Hyperledger+*, we directly store provenance information to Hyperledger's original storage, i.e., RocksDB, and relies on RocksDB's internal index to support provenance query. In the second baseline, called *LineageChain–*, we use ForkBase for the storage of state versions. This baseline does not support multi-state dependency, nor does it have DASL index. We use this to evaluate the storage overhead for the index and for tracking multi-state dependency, and performance benefits of the index.

We perform three sets of experiments. First, we evaluate the performance of *LineageChain* for provenance-dependent blockchain applications. We compare it against the approach that queries provenance offline before issuing blockchain transactions. Second, we evaluate the performance of provenance queries in *LineageChain* on a single machine. For single-state version queries, we use the YCSB benchmark provided in BLOCKBENCH [19] to populate the blockchain states with key-value tuples. We then measure the latencies of two queries: one that retrieves a state at a specific block, and one that iterates over the state history. For multi-state dependency tracking, we implement a

contract for a supply chain application In this application, a phone is assembled from intermediary components which are made from other components or raw material. The supply chain creates a DAG representing the derivation history of a phone. The maximum depth of the DAG is 6. We generate synthetic data for this contract, and examine the latency of the operation that uses `Backtrack` to retrieve dependencies of a given phone.

In the third set of experiments, we evaluate the provenance impact on the overall performance of the blockchain in the distributed setting. To this end, we run the Smallbank benchmark in BLOCKBENCH on multiple nodes. We measure the overall throughput, and analyze the cost breakdown to understand the overhead of provenance support.

Our experiments are conducted on a local clusters of 16 nodes. Each node is equipped with E5-1650 3.5GHz CPU, 32GB RAM, and 2TB hard disk. The nodes are connected via 1Gbps Ethernet. The results reported are based on Hyperledger v0.6, unless stated otherwise.

7.2 Experimental Results

Provenance-dependent applications

We implement a simple provenance-dependent blockchain application by modifying the YCSB benchmark in BLOCKBENCH such that the update operation depends on historical values. With *LineageChain*, the contract has direct access to the provenance information, and the client remains the same as in the original YCSB. Without *LineageChain*, the client is modified such that it reads B latest blocks before issuing transactions. B represents how far behind the client is to the latest states.

We run the experiments using the *LineageChain* implementation on Hyperledger v1.3, with 16 nodes and 1 client. Figure 10(a) shows transaction latency with varying B. It can be seen that with *LineageChain*, the latency remains almost constant because the client does not have to fetch any block for the provenance query. In contrast, without *LineageChain*, the latency increases linearly with B. This demonstrates the performance benefit of having access to provenance information at runtime.

Provenance queries

We first create 500 key-value tuples and then continuously issue update transactions until there are more 10k blocks in the ledger. Each block contains 500 transactions. We then execute a query for the values of a key at different block numbers. Figure 11(a) illustrates the query latency with increasing block distance from the last block. It can be seen that when the distance is small, *LineageChain–* has the lowest latency. *LineageChain–* does not have DASL index, hence for this query it performs linear scan from the latest version. Therefore, it is fast when the requested version is very recent because the number of read is small. However, its performance degrades quickly as the distance increases. In particular, when the block distance reaches 128, the query is 4× slower than *LineageChain*. We observe that the query latency in *Hyperledger+* is independent of the block distance. It is because the query uses RocksDB index directly. *LineageChain* outperforms both *LineageChain–* and *Hyperledger+*. Due to DASL, the query latency in *LineageChain*

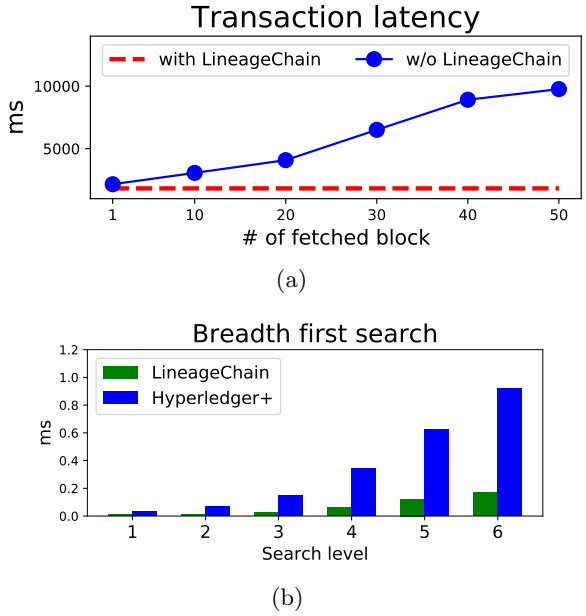


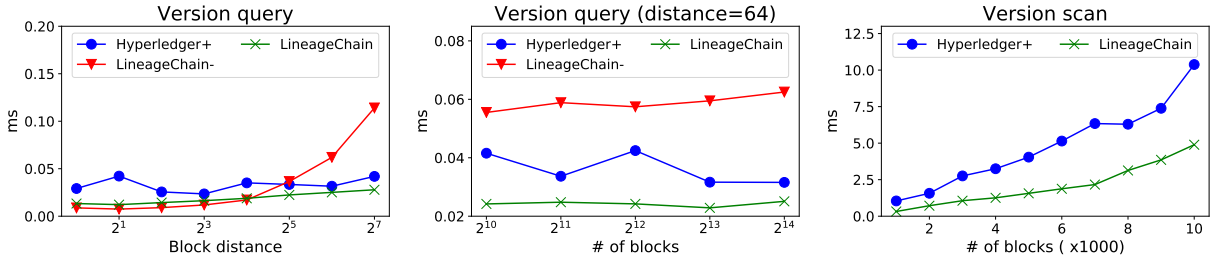
Figure 10: Performance of (a) a provenance-dependent blockchain application and (b) BFS Traversal latency

is low when the block distance is small. When the block distance increases, the latency in *LineageChain* increases only logarithmically, as opposed to linearly in *LineageChain–*.

We repeat the experiment above while fixing the block distance to 64 and varying the total number of blocks. Figure 11(b) shows the results for the version query when the number of block increases. It can be seen that the query latency in both *LineageChain* and *Hyperledger+* remains roughly the same. In other words, the performance of version queries in these systems are independent of the block numbers, which is due to the DAG data model that tracks state versions. *LineageChain* outperforms *Hyperledger+* thanks to the index that reduces the number of hops needed to be read. An interesting observation is that the latency of *Hyperledger+* fluctuates significantly. We attribute this fluctuation to RocksDB's log-structure-merge tree index, in which the requested version may reside at different levels of the tree when the total number of block increases.

Next, we measure the latency for the operation that scan the entire version history of a given key. Figure 11(c) shows the scan latency with increasing number of blocks. For *Hyperledger+*, we first construct the key range and rely on RocksDB iterator for the scanning. *LineageChain–* and *LineageChain* both use ForkBase iterator, therefore they have the same performance. As the number of block increases, the version history becomes longer which accounts for the linear increase in latency in both systems. However, *LineageChain* outperforms *Hyperledger+* by a constant factor. We attribute this difference to ForkBase's optimizations for version tracking.

Finally, we evaluate the query performance with multi-state dependency. We populate the blockchain states with raw materials and issue transactions that create new phones. We perform a breadth-first search to retrieve all the dependencies of a phone. For this experiment, we only compare *Hyperledger+* and *LineageChain*, because *LineageChain–* does not support multi-state dependencies. Figure 10(b)



(a) (b) (c)

Figure 11: Latency of the version query on YCSB with increasing block distance (a) and increasing number of blocks (b). Latency of the version scan with increasing block number (c).

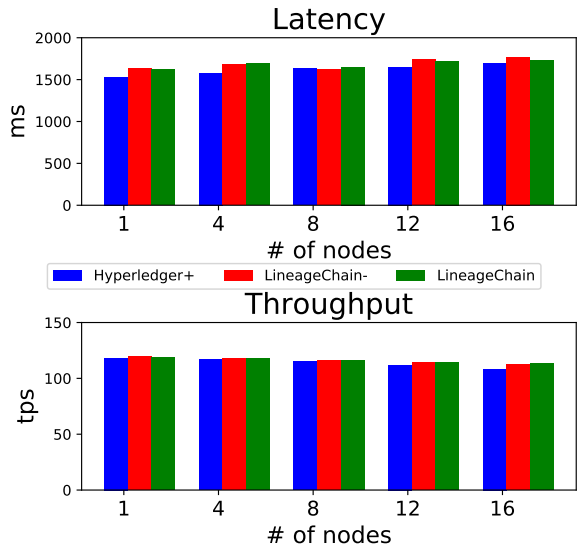


Figure 12: Performance under Smallbank workload.

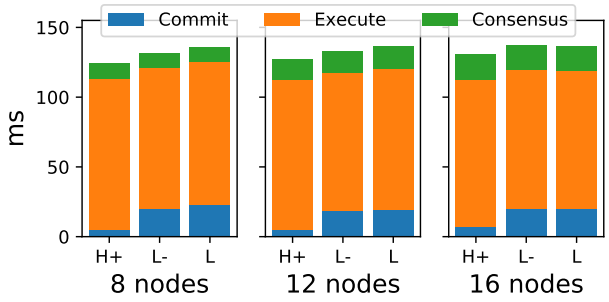


Figure 13: Latency breakdown. *H+*, *L-*, *L* stands for *Hyperledger+*, *LineageChain–* and *LineageChain* respectively.

shows the performance with varying search depths, in which the latency of both *Hyperledger+* and *LineageChain* grow exponentially with increasing depths. However, *LineageChain* outperforms the baseline. It is because the index in *LineageChain* directly captures the dependencies, whereas each backtrack operation in *Hyperledger+* requires traversing on RocksDB index. As the number of queries increases with the search level, their performance gap accumulates.

LineageChain overhead

We run the Smallbank benchmark with a single client and increasing number of nodes. The client uses 16 threads and issues 16 transactions per second. We examine the overhead of provenance support on the overall performance of the blockchain.

Figure 12 shows the transaction latency and overall throughput. We do not observe significant differences between *LineageChain* and the other baselines. In particular, the transaction latency is around 1.2s, and the throughput decreases from roughly 118tps to 108tps.

We break down the block latency into three components: consensus latency (the block proposal phase), execution latency (when transactions are executed), and commit latency (when the states are committed to the storage). Figure 13 shows the detailed breakdown when the number of nodes is 8, 12 and 16. It can be seen that the execution phase accounts for a majority of the cost. There is no significant difference in block execution time of *Hyperledger+* and *LineageChain*. Hence, the overhead of the provenance capture engine is negligible.

The block commit latency in *LineageChain–* and *LineageChain* are 2× higher than that of *Hyperledger+*. It is due to ForkBase's computation to update its internal data structures during commit. However, the block commit phase accounts for less than 20% of the total block latency. We further note that the transaction latency is in order of seconds, whereas a block latency is in in order of hundreds of milliseconds. Thus, any extra overhead incurred by provenance support does not result in any visible differences in the user-perceived performance of the blockchain.

Next, we examine the breakdown in storage cost of *LineageChain*. Figure 14 shows the storage breakdown with varying number of blocks. The block size is fixed at 500 transactions per block. Figure 14 shows the breakdown with varying block sizes, while fixing the number of blocks at 1000. It can be seen that the storage size grows linearly with the number of blocks and block sizes. We observe that the block content accounts for the majority the storage cost. In contrast, the extra provenance and DASL index account for only $2-4\%$ of the total space consumption. In *LineageChain*, the DASL pointers are implemented as 20-byte *vid* strings. A new state will at most add $D + log(N)$ new pointers where N is the number of previous versions and D is the number of dependent states. When compared with kilobyte-sized blocks, the storage consumption of these pointers is not significant. As a result, the storage overhead of DASL is small.

Finally, we evaluate *LineageChain* overhead on Hyperledger v1.3. This version of Hyperledger uses a different

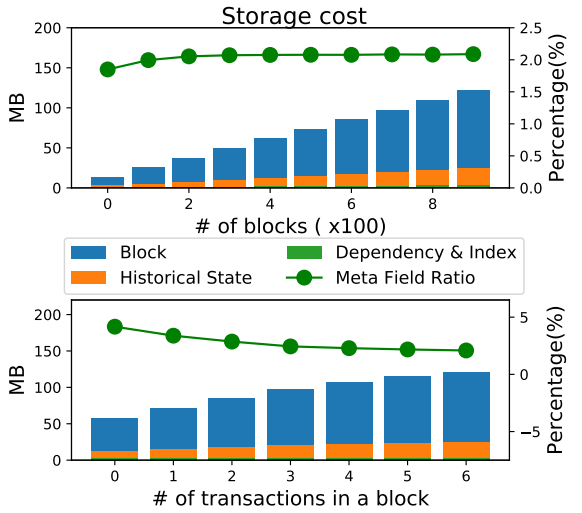


Figure 14: Storage cost breakdown.

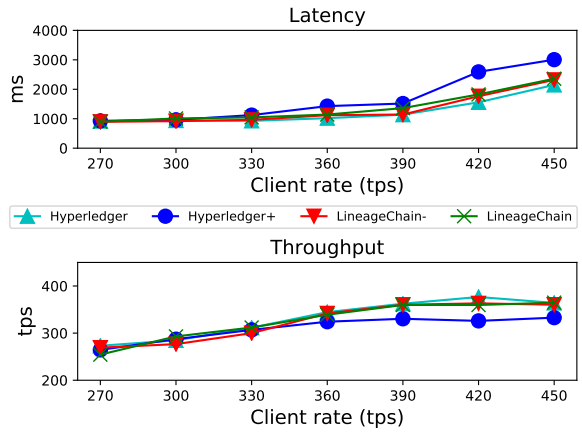


Figure 15: Performance on Hyperledger v1.3

consensus protocol and more simplified data model than v0.6. We use 16 nodes and vary the offer load by increasing the client's transaction rate. Figure 15 shows the performance overhead. At saturation, *LineageChain–* and *LineageChain* add less than 200*ms* in latency, compared to the original Hyperledger that has no provenance support. In contrast, *Hyperledger+* adds more than 1*s*. *LineageChain–* and *LineageChain* reach similar throughput as the original Hyperledger, which is around 350tps. *Hyperledger+* peaks at around 330tps. These results demonstrate that *LineageChain*'s overhead over the original Hyperledger is small.

8. RELATED WORK

Data Provenance Data provenance has been studied extensively in database systems. Support for provenance has been added to a wide range of systems, from relational databases [13, 14, 8], collaborative data sharing systems [23, 33], to big data platforms [30, 5, 36]. However, these systems have different requirements on provenance and therefore face different challenges. As a result, their provenance solutions are ad-hoc and do not generalize well. In relational databases, Peter Buneman et al. [9] propose an approach for query provenance, in which they identify two types of provenance for databases: "why" and "where" provenance. In Hadoop and other MapReduce systems, data derivation graphs are established based on the data flow from mappers to reducers [21]. Titian [22] instruments Spark's original RDDs to capture data transformation. Other works exploit provenance in specific application domains, such as networks [12], language processing [17] and interactive workflow applications [31]. They focus on improving provenance storage and query efficiency.

Our work shares the same spirit as the above. We add provenance capabilities to blockchains, which enables a new class of blockchain applications. We design a new data model and index that are optimized for blockchains. Our system addresses the unique challenges raised by the decentralization nature of blockchains.

Blockchain Most research in blockchain systems focuses on security of permissionless blockchains. One open challenge is to improve incentive compatibility. A protocol is incentive compatible if honest participants get rewards proportional to its contributions to the network. Bitcoin, for example, is not incentive compatible [20, 32, 28]. Another challenge is to improve smart contracts security. Recent vulnerabilities in Ethereum that resulted in substantial financial loss has prompted many efforts in finding bugs and securing smart contracts with programming language techniques [6, 16, 25]. As a data management system, blockchain has poor performance. BLOCKBENCH [19] compares several blockchains with in-memory databases and shows orders of magnitude difference in transaction throughput.

Our work aims to enrich blockchain capabilities by introducing provenance. It is orthogonal to those above that aim to improve security and performance of the consensus protocol. The most closely related work to ours is ForkBase [37]. In fact, current *LineageChain* implementation is built on top of ForkBase to leverage its version tracking capability. However, our novelties over ForkBase include multi-state dependency tracking, efficient index, and rich APIs for accessing provenance at runtime. Another similar recent work is VChain [38], where researchers achieve the integrity of boolean range queries over the historical data on blockchain databases. But different from ours, they are optimizing for the offline analytical query. Instead, we extend the provenance support to blockchain online transactions.

9. CONCLUSIONS

In this paper, we presented *LineageChain*, a fine-grained, secure and efficient provenance system for blockchains. The system efficiently captures provenance information during runtime and stores it in a secure storage. It exposes simple APIs to smart contracts, which enables a new class of provenance-dependent blockchain applications. Provenance queries are efficient in *LineageChain*, thanks to a novel skip list index. We implemented *LineageChain* on top of Hyperledger and benchmarked it against several baselines. The results show the benefits of *LineageChain* in supporting rich, provenance-dependent applications. They demonstrate that provenance queries are efficient, and that the system incurs small storage overhead.

10. ACKNOWLEDGMENTS

This research is supported by Singapore Ministry of Education Academic Research Fund Tier 3 under MOE's official grant number MOE2017-T3-1-007.

11. REFERENCES

[1] Ethereum. https://www.ethereum.org.

[2] Hyperledger. https://www.hyperledger.org.

[3] Hyperledger++. https://www.comp.nus.edu.sg/~dbsystem/hyperledger++/index.html.

[4] Medilot. https://medilot.com.

[5] S. Akoush, R. Sohan, and A. Hopper. Hadoopprov: Towards provenance as a first class citizen in mapreduce. In *TaPP*, 2013.

[6] N. Atzei, M. Bartoletti, and T. Cimoli. A survey of attacks on ethereum smart contracts (sok). In *Principles of Security and Trust*, pages 164–186. Springer, 2017.

[7] S. Bano, A. Sonnino, M. Al-Bassam, S. Azouvi, P. McCorry, S. Meiklejohn, and G. Danezis. Consensus in the age of blockchain. https://arxiv.org/abs/1711.03936, 2018.

[8] P. Buneman, A. Chapman, and J. Cheney. Provenance management in curated databases. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 539–550. ACM, 2006.

[9] P. Buneman, S. Khanna, and T. Wang-Chiew. Why and where: A characterization of data provenance. In *International conference on database theory*, pages 316–330. Springer, 2001.

[10] C. Cachin, S. Schubert, and M. Vukolić. Non-determinism in byzantine fault-tolerant replication. *arXiv preprint arXiv:1603.07351*, 2016.

[11] M. Castro, B. Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[12] C. Chen, H. T. Lehri, L. Kuan Loh, A. Alur, L. Jia, B. T. Loo, and W. Zhou. Distributed provenance compression. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 203–218. ACM, 2017.

[13] J. Cheney, L. Chiticariu, W.-C. Tan, et al. Provenance in databases: Why, how, and where. *Foundations and Trends® in Databases*, 1(4):379–474, 2009.

[14] L. Chiticariu, W.-C. Tan, and G. Vijayvargiya. Dbnotes: a post-it system for relational databases based on provenance. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 942–944. ACM, 2005.

[15] H. Dang, T. T. A. Dinh, D. Loghin, E.-C. Chang, Q. Lin, and B. C. Ooi. Towards scaling blockchain systems via sharding. *arXiv preprint arXiv:1804.00399*, 2018.

[16] K. Delmolino, M. Arnett, A. Kosba, A. Miller, and E. Shi. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *International Conference on Financial Cryptography and Data Security*, pages 79–94. Springer, 2016.

[17] D. Deutch, N. Frost, and A. Gilad. Provenance for natural language queries. *PVLDB*, 10(5):577–588, 2017.

[18] T. T. A. Dinh, R. Liu, M. Zhang, G. Chen, B. C. Ooi, and J. Wang. Untangling blockchain: A data processing view of blockchain systems. *IEEE Transactions on Knowledge and Data Engineering*, 30(7):1366–1385, 2018.

[19] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan. Blockbench: A framework for analyzing private blockchains. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1085–1100. ACM, 2017.

[20] I. Eyal and E. G. Sirer. Majority is not enough: Bitcoin mining is vulnerable. *Communications of the ACM*, 61(7):95–102, 2018.

[21] R. Ikeda, H. Park, and J. Widom. Provenance for generalized map and reduce workflows. 2011.

[22] M. Interlandi, K. Shah, S. D. Tetali, M. A. Gulzar, S. Yoo, M. Kim, T. Millstein, and T. Condie. Titian: Data provenance support in spark. *PVLDB*, 9(3):216–227, 2015.

[23] Z. G. Ives, T. J. Green, G. Karvounarakis, N. E. Taylor, V. Tannen, P. P. Talukdar, M. Jacob, and F. Pereira. The orchestra collaborative data sharing system. *ACM Sigmod Record*, 37(3):26–32, 2008.

[24] K. Korpela, J. Hallikas, and T. Dahlberg. Digital supply chain transformation toward blockchain integration. In *proceedings of the 50th Hawaii international conference on system sciences*, 2017.

[25] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 254–269. ACM, 2016.

[26] L. Luu, J. Teutsch, R. Kulkarni, and P. Saxena. Demystifying incentives in the consensus computer. In *CCS*, 2015.

[27] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2009.

[28] K. Nayak, S. Kumar, A. Miller, and E. Shi. Stubborn mining: Generalizing selfish mining and combining with an eclipse attack. In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*, pages 305–320. IEEE, 2016.

[29] Q. K. Nguyen. Blockchain-a financial technology for future sustainable development. In *2016 3rd International Conference on Green Technology and Sustainable Development (GTSD)*, pages 51–54. IEEE, 2016.

[30] H. Park, R. Ikeda, and J. Widom. Ramp: A system for capturing and tracing provenance in mapreduce workflows. 2011.

[31] F. Psallidas and E. Wu. Smoke: Fine-grained lineage at interactive speed. *PVLDB*, 11(6):719–732, 2018.

[32] A. Sapirshtein, Y. Sompolinsky, and A. Zohar. Optimal selfish mining strategies in bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 515–532. Springer, 2016.

[33] Y. L. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *ACM Sigmod Record*, 34(3):31–36, 2005.

[34] A. Tapscott and D. Tapscott. How blockchain is changing finance. *Harvard Business Review*, 1(9), 2017.

[35] F. Tian. An agri-food supply chain traceability system for china based on rfid & blockchain technology. In *Service Systems and Service Management (ICSSSM), 2016 13th International Conference on*, pages 1–6. IEEE, 2016.

[36] J. Wang, D. Crawl, S. Purawat, M. Nguyen, and I. Altintas. Big data provenance: Challenges, state of the art and opportunities. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 2509–2516. IEEE, 2015.

[37] S. Wang, T. T. A. Dinh, Q. Lin, Z. Xie, M. Zhang, Q. Cai, G. Chen, B. C. Ooi, and P. Ruan. Forkbase: An efficient storage engine for blockchain and forkable applications. *PVLDB*, 11(10):1137–1150, 2018.

[38] C. Xu, C. Zhang, and J. Xu. vchain: Enabling verifiable boolean range queries over blockchain databases. *arXiv preprint arXiv:1812.02386*, 2018.