Revealing Every Story of Data in Blockchain Systems

Pingcheng Ruan
National University of Singapore
ruanpc@comp.nus.edu.sg

Tien Tuan Anh Dinh
Singapore University of Technology and Design
dinhtta@sutd.edu.sg

Qian Lin
National University of Singapore
linqian@comp.nus.edu.sg

Meihui Zhang
Beijing Institute of Technology
meihui_zhang@bit.edu.cn

Gang Chen
Zhejiang University
cg@zju.edu.cn

Beng Chin Ooi
National University of Singapore
ooibc@comp.nus.edu.sg

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 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 stores it in a Merkle tree. *LineageChain* provides a novel skip list index that supports efficient provenance queries. We have implemented *LineageChain* on top of Hyperledger Fabric and a blockchain-optimized storage system called ForkBase. We conduct extensive evaluation, demonstrating the benefits of *LineageChain*, its efficient query, and its small storage overhead.

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 is first used in Bitcoin [13] to store cryptocurrency transactions. Often referred to as decentralized ledger, blockchain ensures integrity (tamper evidence) of the complete transaction history. It is replicated over a peer-to-peer (P2P) network, and a distributed consensus protocol, for instance 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 [3], enables applications beyond cryptocurrencies by supporting for smart contracts. A smart contract has its states stored on the blockchain, and the states are modified via transactions that invoke the contract.

The management of data history, or data provenance, has been extensively studied in databases, and many systems have been designed to support provenance [7, 5, 4]. 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. However, this reconstruction can only be done during offline analysis. During contract execution (or runtime), no provenance information is accessible to smart contracts. This lack of runtime access to provenance therefore restricts the expressiveness of the computation logics that 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 tokens if B's average balance per day is more than t. It is not currently possible to 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. Beside performance overhead incurred from multiple interactions with the blockchain, this approach is not *safe*: it violates 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. This scenario can be caused by benign network conditions as well as by malicious attacks. In blockchains with native currencies, serializability violation can be exploited for Transaction-Ordering attacks that cause substantial financial loss to the users [11].

We design and implement *LineageChain*, a fine-grained, secure, and efficient provenance system for blockchains that enables a new class of smart contracts which can access provenance information at runtime. Although our goal is

 This is a minor revision of the paper entitled "Fine-Grained, Secure and Efficient Data Provenance on Blockchain Systems", published in the Proceedings of the VLDB Endowment, Vol. 12, No. 9, 975-988. 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 token management smart contract.

similar to that of existing works in adding provenance to databases, 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 efficient, not only to improve latency, but also to avoid degrading security [12]. To address this challenge, we design a novel skip list index optimized for provenance queries.

In summary, we make the following contributions:

- We present *LineageChain*, a system that efficiently captures fine-grained provenance for blockchains. It stores provenance securely, and exposes simple access interface to smart contracts.
- We present a novel index optimized for querying blockchain provenance. The index is similar to skip list, but is deterministic. Its performance is independent of the blockchain size.
- We implement *LineageChain* for Hyperledger Fabric v1.3 [3]. Our implementation builds on top of ForkBase, a blockchain-optimized storage [15]. Our experimental results demonstrate its benefits to provenance-dependent applications and its efficient query.

LineageChain is a component of our FabricSharp system [2], for which we improve Fabric'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 [8]. 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.

2. BLOCKCHAIN STATES ORGANIZATION

In this section, we discuss how the global states are organized in blockchains. [9] provides a comprehensive survey of blockchain design. There are three key requirements for building an index over the global states of a blockchain. We explain how they are met in Ethereum and Hyperledger. *LineageChain* also meets these requirements.

Tamper evidence. A user may 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. The index must provide a unique digest for the global states, so that blockchain nodes can quickly check if their states are the same.

Incremental update. The size of global states may be, but one block only updates a small portion of 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 necessary because of the immutability property of the blockchain which allows users to read any historical states. It is also important for block verification: when a new block is received that creates a fork, an old snapshot of the state is used for verification. Even when the blockchain allows no forks, snapshots enable roll-back when the received block is found to be invalid after execution.

Existing blockchains use indices that are based on Merkle tree. In particular, Ethereum implements Merkle Patricia Trie (MPT), and Hyperledger Fabric v0.6 implements Merkle Bucket Tree (MBT). In a Merkle tree, content of the parent node is recursively defined by those of the child nodes. A proof of integrity can be efficiently constructed without reading the entire tree. The Merkle tree meets the first requirement. It also meets the second requirement, because only the tree nodes affected by the update need to be changed. It meets the third requirement, because an update in the block recursively creates new tree nodes in the path affected by the change. And then new root then serves as index of the new snapshot, and is then included in the block header.

3. FINE-GRAINED PROVENANCE

In this section, we describe how we capture provenance, and the smart contract APIs for accessing it. We use as running example of 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 accounts (or addresses), *Addr1* and *Addr2*, are initialized with 100 tokens. Two transactions *Txn1* and *Txn2* transferring tokens between the two addresses are committed at block M and N respectively. The value of *Addr1* is 100 from block L 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

In *LineageChain*, every contract method can be made provenance-friendly via a *helper* method. In particular, dur-

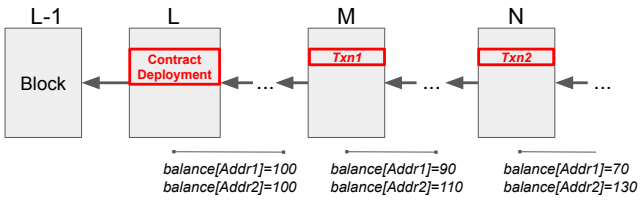


Figure 2: Content of the blockchain and *gState*.

```
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, which defines dependency between the sender identifier and recipient identifier.

ing transaction execution, *LineageChain* collects the identifiers and values of the accessed states, i.e., those 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 `prov_helper` method, 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[]);
```

`prov_helper` is defined by the contract developer, and it returns a set of dependencies based on the input read and write sets. Figure 3 shows an 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 $\{Addr2 : [Addr1]\}$.

3.2 Smart Contract APIs

Current smart contracts can only access the *latest* states. *LineageChain* provides access to the captured provenance via 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)`.

```
contract Token {
  ...
  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 the new APIs.

Figure 4 illustrates how the above APIs are used to express smart contract logics that are currently impossible, as shown in the `Blacklist` method. This will mark an address as blacklisted if one of its last 5 transactions is with a blacklisted address.

4. PROVENANCE STORAGE AND QUERY

In this section, we describe the design for storing and querying the captured provenance.

4.1 Storage

LineageChain enhances existing blockchain storage layer to provide efficient tracking and tamper evidence for the captured provenance. The key idea is to reorganize the flat leaf nodes in the original Merkle tree into a Merkle DAG.

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 trivial. 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 $k = Addr1$

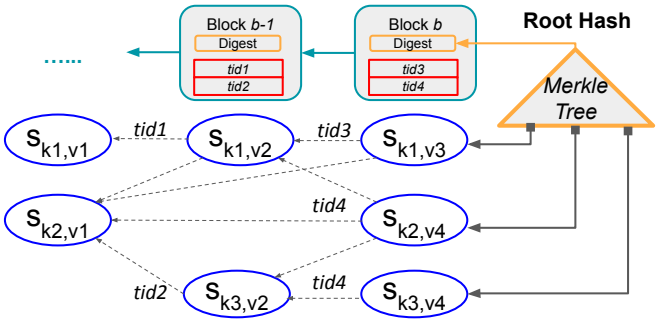


Figure 5: A Merkle DAG for storing provenance. s_{k_2,v_4} and s_{k_3,v_4} updated by the same transaction (tid_4), block b contains two transactions, tid_3 and tid_4. Its latest states are represented by the Merkle root.

and $v = M$, $s_{k,v} = 90$.

Definition 1. *A transaction, identified by tid which is strictly increasing, consumes a set of input states S^i_{tid} and produces 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. Finally, Property (4) means that every state update is unique.

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

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

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:*

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

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.

Forward tracking. One problem with 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 history of the state. However, the state can be read by future transaction, and as a consequence its forward dependencies cannot be determined at the time of update.

Fortunately, an important observation 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 history. Figure 5 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_2}. This is because when the transaction that outputs s_{k_2,v_4} is executed, it reads s_{k_1,v_2} 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 $F_{s_{k,v}}$, where

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

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

4.2 Efficient Query

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

Deterministic Append-only Skip List. We propose to build an index, called Deterministic Append-only Skip List (or DASL), on top of a state DAG to support fast version queries. The index has a skip list structure. It is designed for blockchains, exploiting the fact that the blockchain is append-only, and randomness is not well supported [6]. A DASL has two distinct properties compared to a normal skip list. First, it is append-only; that is, the index keys of the

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

Figure 6: A Node structure that captures a state $s_{k,v}$ with value val

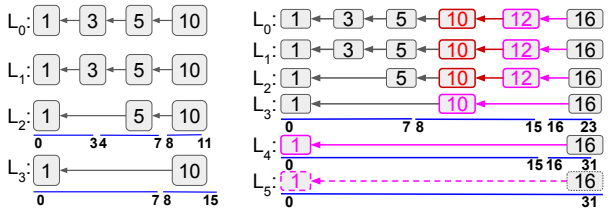


Figure 7: (a) A DASL containing versions 1, 3, 5 and 10. The base b is 2. The intervals for L_2 and L_3 are shown in blue lines. (b) The new DASL after appending version 12 and 16. L_4 is created when appending version 16. L_5 is created, then discarded.

appended entries, which are state versions, 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.

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^i_{j-1} and v^i_j 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^i_{j-1}, v^i_j is the smallest version in V_k such that:

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

Figure 6 shows how DASL is stored with the state in a data structure called Node. This structure (also referred to as node) contains the state version and value. A node belongs to multiple lists (or levels), hence it maintains a list of pointers to other nodes in each level as well as a list of the version numbers of pointed 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, and 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 $R^i_j = [jb^i, (j+1)b^i)$. Only the smallest version in V_k that falls in this range is included in the list. Figure 7(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

Algorithm 1: 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;
```

in order to 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 the time of v_q.

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 1 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, and then keep moving up the list level until the current node and the new node belong to the same interval (line 9 - 18). Figure 7(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 7(b), when appending version 16, all existing lists can be used. The algorithm then creates L_4 with node 1 and appends node 16 to it. It also creates a new level L_5, but subsequently discards it because node 16 will not be appended since it belongs to same interval of $[0, 32)$ with node 1.

4.3 Discussion

Our new Merkle DAG can be easily integrated to existing blockchain index structures. It meets the three requirements listed in Section 2. 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 preserve 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. 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.

DASL and Merkle DAG integration. Adding DASL to the Merkle DAG is straightforward. The node structure (Figure 6) is stored in the state entry (Definition 3). The node pointers are implemented as entry hashes. The Merkle tree structure remains unchanged.

Speed vs. storage. 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. 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 $2b\lceil \log_b d \rceil$. This is because a typical query consists of two stages: one going towards the lower levels, and the other going towards the upper level. Each stage involves traversing 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. It can be seen that b determines the tradeoff between the space overhead and query latency. Furthermore, DASL queries are more efficient for more recent versions, i.e. d are small, which is useful for smart contracts that rely on recent rather than old versions. Finally, the performance of such recent-version queries does not change as the state history grows.

5. PERFORMANCE EVALUATION

We implement *LineageChain* on top of Hyperledger Fabric v1.3. More details of the implemenation can be found in [14]. Figure 8 shows the software stack, highlighting the changes to the original Fabric's stack. We completely replace Fabric's storage layer with our implementation of the Merkle DAG and DASL index on top of ForkBase [15], a state-of-the-art blockchain storage system with efficient support for versioning. We instrument Fabric's 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 invokes the helper method after every successful contract execution.

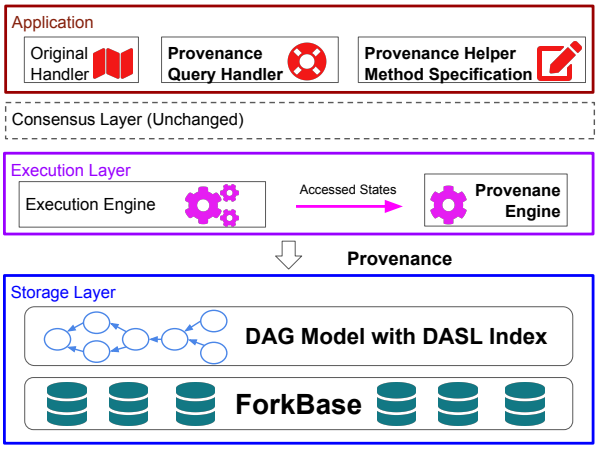


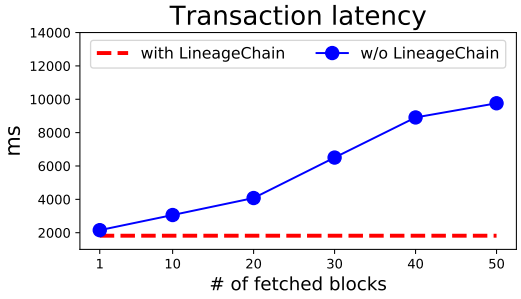
Figure 8: *LineageChain*'s software stack. The original storage layer is replaced with the implementation that supports fine-grained provenance. The original execution layer is instrumented with a provenance capture engine. The application layer contains the new helper method and provenance query APIs. The consensus layer is unchanged.

5.1 Methodologies

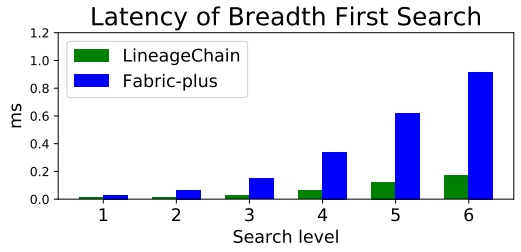
We evaluate *LineageChain* against two baselines. In the first baseline, called *Fabric-plus*, we directly store provenance information to Hyperledger's original storage and relies on its internal index to support provenance query. In the second baseline, called *LineageChain-lite*, we use ForkBase for storing state versions. This baseline has no support for multi-state dependency, and no DASL index. We use this to understand the index's performance.

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 [10] to populate the blockchain states with key-value tuples. We then measure the latencies of two queries: one retrieves a state at a specific block, and the other 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 impact of provenance on the overall blockchain performance. For this, we run the Smallbank benchmark on multiple nodes.



(a)



(b)

Figure 9: Performance of (a) a provenance-dependent blockchain application and (b) BFS Traversal latency

We measure the overall throughput, and analyze the cost breakdown to understand the overhead of provenance support.

Our experiments are run 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.

5.2 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.

Figure 9(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 gain brought by *LineageChain* for 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 than 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 10(a) illustrates the query latency with increasing block distance from the last block. It can be seen that when the distance is small, *LineageChain-lite* has the lowest latency. *LineageChain-lite* does not have DASL index, and as a consequence for this query it has to scan linearly from the latest version. As expected, the query is fast when the requested version is very recent because the number of read is small, but degrades the performance 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 *Fabric-plus* is independent of the block distance, because the query uses flat storage index directly. *LineageChain* outperforms both *LineageChain-lite* and *Fabric-plus*. Because of DASL, the query latency in *LineageChain* is low when the block distance is small. When the block distance increases, the latency increases only logarithmically, as opposed to linearly in *LineageChain-lite*.

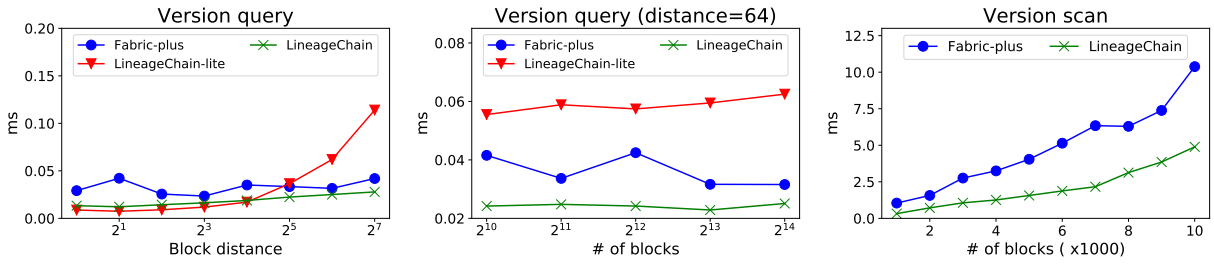
We repeat the experiment above while fixing the block distance to 64 and varying the total number of blocks. Figure 10(b) shows the results for the version query with increasing number of blocks. It can be seen that the query latency in both *LineageChain* and *Fabric-plus* remains roughly the same. 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 *Fabric-plus*, thanks to the index that reduces the number of entries needed to be read.

Next, we measure the latency for the operation that scans the entire version history of a given key. Figure 10(c) shows the scan latency with increasing number of blocks. For *Fabric-plus*, we first construct the key range and rely on the storage iterator for scanning. *LineageChain-lite* and *LineageChain* both use ForkBase iterator, and 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 *Fabric-plus* 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 and issue transactions that produce new phones. We perform a breadth-first search to retrieve all the dependencies of a phone. For this experiment, we only compare *Fabric-plus* and *LineageChain*, because *LineageChain-lite* does not support multi-state dependencies. Figure 9(b) shows the performance with varying search depths. The latency of both *Fabric-plus* and *LineageChain* grow exponentially with increasing depths, but *LineageChain* outperforms the baseline. It is because the index in *LineageChain* directly captures the dependencies, whereas each backtrack operation in *Fabric-plus* requires traversing on the storage index. As the number of queries increases with the search level, their performance gap widens.

Performance Overhead

Finally, we evaluate *LineageChain* overhead on Hyperledger Fabric v1.3. We use 16 nodes and vary the offer load by increasing the client's transaction rate. Figure 11 shows the performance overhead. At saturation, *LineageChain-lite* and *LineageChain* add less than 200ms in latency, compared to the original Fabric that has no provenance support. In contrast, *Fabric-plus* adds more than 1s. *LineageChain-lite*



(a) (b) (c)

Figure 10: 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).

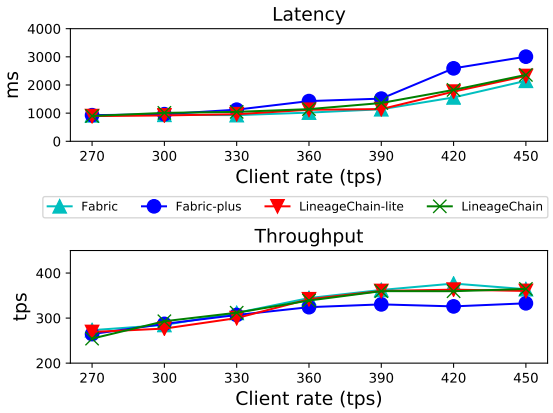


Figure 11: Performance on Fabric v1.3

and *LineageChain* reach similar throughput as the original Hyperledger, which is around 350tps. *Fabric-plus* peaks at around 330tps. These results demonstrate that *LineageChain*'s overhead over the original Fabric is small.

6. 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 exposes simple APIs to smart contracts, which enables a new class of provenance-dependent blockchain applications. Provenance is stored securely, and queries are efficient thanks to a novel skip list index. We implemented *LineageChain* on top of Hyperledger Fabric 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 *LineageChain* incurs small runtime overhead.

7. 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.

8. REFERENCES

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

[2] FabricSharp. `https://www.comp.nus.edu.sg/~dbsystem/fabricsharp/`.

[3] Hyperledger. `https://www.hyperledger.org`.

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

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

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

[14] P. Ruan, G. Chen, T. T. A. Dinh, Q. Lin, B. C. Ooi, and M. Zhang. Fine-grained, secure and efficient data provenance on blockchain systems. *Proceedings of the VLDB Endowment*, 12(9):975–988, 2019.

[15] 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.