KV-Fresh: Freshness Authentication for Outsourced Multi-Version Key-Value Stores

Yidan Hu
Computer and Information Sciences
University of Delaware
Newark, DE, USA
yidanhu@udel.edu

Rui Zhang
Computer and Information Sciences
University of Delaware
Newark, DE, USA
ruizhang@udel.edu

Yanchao Zhang
Electrical, Computer and Energy Engineering
Arizona State University
Tempe, AZ, USA
yczhang@asu.edu

***Abstract*—Data outsourcing is a promising technical paradigm to facilitate cost-effective real-time data storage, processing, and dissemination. In such a system, a data owner proactively pushes a stream of data records to a third-party cloud server for storage, which in turn processes various types of queries from end users on the data owner's behalf. This paper considers outsourced multi-version key-value stores that have gained increasing popularity in recent years, where a critical security challenge is to ensure that the cloud server returns both authentic and fresh data in response to end users' queries. Despite several recent attempts on authenticating data freshness in outsourced key-value stores, they either incur excessively high communication cost or can only offer very limited real-time guarantee. To fill this gap, this paper introduces KV-Fresh, a novel freshness authentication scheme for outsourced key-value stores that offers strong real-time guarantee. KV-Fresh is designed based on a novel data structure, Linked Key Span Merkle Hash Tree, which enables highly efficient freshness proof by embedding chaining relationship among records generated at different time. Detailed simulation studies using a synthetic dataset generated from real data confirm the efficacy and efficiency of KV-Fresh.**

I. Introduction

Data outsourcing is a promising technical paradigm to facilitate cost-effective storage, processing, and dissemination of real-time data stream. In such a system, a data owner proactively pushes one or multiple high-volume data streams generated by distributed data sources to a third-party cloud server for storage and backup, which in turn processes various types of queries from many end users on the data owner's behalf. Examples of massive data streams include those collected by stock exchange service providers, online social networking service providers, critical infrastructure monitoring companies, and so on. In addition to higher availability and elasticity offered by cloud service provider, data outsourcing can also relieve the data owner from cumbersome management work and result in significant saving in operation cost.

This paper considers outsourced multi-version key-value store [1], [2], which is a non-SQL data model developed for storage, analysis and access of large volume of unstructured data. A key-value store is a database storing a collection of data records, each of which is a key-value pair that can be efficiently retrieved using the key. In a multi-version key-value store, the data value of a record has multiple versions, each of which is an updated value received at a different time. In comparison with traditional relational databases, key-value stores do not enforce any structure on the data and offers higher scalability, simpler designs, and higher availability. Examples of key-value stores include MongoDB, Amazon DynamoDB, Azure Cosmos DB, and so on. Key-value stores and other non-SQL databases have gained increasing popularity in recent years, of which the market is expected to reach 4.2 billion by 2020 [3].

Data outsourcing, unfortunately, poses critical security challenges in that cloud service providers cannot be fully trusted to faithfully provide query results to end users based on authentic and up-to-date data for various reasons. First, a compromised cloud server may provide forged data in response to end users' queries to mislead users into making incorrect decisions. For example, a cloud service provider may return forged data values in favor of the businesses with financial interests, and similar misbehavior have been widely reported in web search industry. Second, a cloud service provider may provide authentic but stale data, which is more subtle and difficult to detect. For example, a cloud service provider may purposefully drop some data for saving storage cost. Such misbehavior is particularly economically appealing if the data is of large volume and subjected to frequent update. In comparison to the first attack, this attack can also lead to bad decisions by end users but is more subtle and difficult to detect. These situations call for sound authentication techniques to ensure both authenticity and freshness of any query result returned by the cloud service provider.

Despite many efforts on authenticating outsourced query processing [4]–[24], authenticating data freshness is particularly challenging and has thus far received very limited attention. Common to existing solutions [25]–[29] is to divide the time into intervals and let the data owner generate a cryptographic proof for every key with no update in every interval. On receiving a query, the cloud server is required to return the most recent value for the queried key along with a freshness proof. While such approaches allow end users to verify the freshness of query results, the size of freshness proof is linear to the number of the intervals after the most recent update and thus inversely proportional to the length of the interval. As a result, existing solutions [25]–[29] either suffer from excessively high communication cost or can only

support limited real-time guarantee. For example, the state-of-art solution [29] can only support interval size in minutes. How to realize freshness authentication with strong real-time guarantee remains an open problem.

In this paper, we tackle this open challenge by introducing KV-Fresh, a novel freshness authentication mechanism for outsourced multi-version key-value store. We observe that the key to simultaneously achieve strong real-time guarantee and communication efficiency is to break the linear dependence between the size of freshness proof and the number of intervals after the latest update. Based on this observation, we introduce a novel data structure that embeds chaining relationship among updates in different intervals to realize efficient freshness proof. Built upon this novel data structure, KV-Fresh allows the cloud server to prove the freshness of query results by returning information for only a small number of intervals while skipping potentially many intervals in between. Our contributions in this paper can be summarized as follows.

- We identify a key limitation of existing solutions on freshness authentication that they either suffer from excessively high communication cost or can only support limited real-time guarantee.
- We propose a novel data structure that allows highly efficient proof of no update over a large of number of intervals.
- We introduce KV-Fresh, a novel freshness authentication mechanism for outsourced multi-version key-value stores that provides stronger real-time guarantee with low communication cost.
- We confirm the high efficiency of KV-Fresh via extensive simulation studies using a synthetic dataset generated from a real dataset. In particular, our simulation results show that KV-Fresh reduces the communication cost by up to 99.6% for proving data freshness and achieves up to nine times higher throughput in comparison with the state-of-art solution INCBM-TREE [29].

II. Related Work

Our work is mostly related to authenticating data freshness and existing solutions can be generally classified into two categories. The first category relies on the data owner to construct and maintain a proper data digest at the cloud server, such as a Merkle Hash tree or its variants [25]–[27], [30], [31]. These approaches require the data owner to maintain large local states about historical data or incur significant communication cost between the data owner and the cloud server. The second category [32]–[34] detects the cloud server's misbehavior through offline audit, which cannot guarantee data freshness in real-time. To authenticate data freshness in real time, Yang *et al.* introduced a design based on trusted computing hardware [28]. In [29], Tang *et al.* introduced INCBM-TREE, a data structure based on the Bloom filter and multi-level key-ordered Merkle hash tree. INCBM-TREE can only support relaxed real-time freshness check at the granularity of minute-based intervals, as the size of the freshness proof is inversely proportional to the interval length. Our work is mostly related to [29] and enables freshness verification at much smaller time granularity without using any trusted computing hardware.

Our work is also related to authenticating outsourced query processing [4], in which a data owner outsources its dataset to a third party service provider who is responsible for answering the data queries from end users on the data owner's behalf. Significant effort has been devoted to ensuring query integrity and completeness, i.e., a query result contains all the authentic data records satisfying the query. Various types of queries have been studied, including relational queries [5]–[7], range queries [8]–[12], top-k queries [13]–[17], skyline queries [18]–[21], kNN queries [22], [23], shortest-path queries [24], etc. A general framework is to let the data owner precompute some auxiliary information from its dataset, using cryptographic techniques to accompany its dataset, whereby the third party service provider can generate a proof in response to a user's query. None of these works consider the freshness of returned data records, and they are thus inapplicable to the problem addressed in this paper.

III. Problem Formulation

In this section, we introduce our system and adversary models and design goals.

A. System Model

We consider a data outsourcing system consisting of three parties: a data owner, a third-party cloud server, and many users. The data owner outsources a dataset in the form of a multi-version key-value store to the cloud server, which in turn answers data queries from users on the data owner's behalf.

The data owner maintains the key-value store at the cloud server by proactively pushing data updates to the cloud server as they become available. We assume that the keys can be ordered and denote by $\mathcal{K} = \{1, \ldots, |\mathcal{K}|\}$ the key space. The key-value store consists of a collection of data records, each of which contains a unique key $k \in \mathcal{K}$ and a data value that can have multiple versions received over different time. Each version corresponds to an update in the form of (k, v, t), where k is the key, v is the update value, and t is the timestamp indicating the time at which the update is issued.

Users access data records in the key-value store through the cloud server's GET API. Specifically, any user can issue a GET query as $Q(k, t_q)$, where k is the queried key and t_q is an optional parameter indicating the point of time up to which the data record is requested. On receiving query $Q(k, t_q)$, the cloud server needs to return the most recent data record for key k as of t_q. The absence of parameter t_q indicates that the user is asking for the most recent data record as of now. Extending our work to support other types of queries, e.g., range queries, is left as our future work.

B. Adversary Model

We assume that the data owner is trusted to faithfully perform all system operations. In contrast, the cloud server cannot be fully trusted and may launch the following two attacks. First, the cloud server may return forged or tampered

data records that do not belong to the data owner's dataset. Second, the cloud server may return authentic but stale data records in response to the user's GET query.

We assume that the communication channels between the data owner and the cloud server as well as between the cloud server and users are secured using standard techniques, e.g., TLS [35]. In addition, we also assume that the data owner cannot predict the keys that the user will query in advance.

C. Design Goals

Strict freshness verification—also referred to as real-time freshness check in [29]—requires the data owner to not only push authenticated data updates to the cloud sever as soon as there are available but also constantly inform the cloud server even if there is no update, which would result in prohibitive processing and communication cost. As in the state-of-art solution in [29], we seek to achieve relaxed real-time freshness verification. Specifically, we assume that time is divided into intervals of equal length, which means that the data owner pushes authenticated date updates to the cloud server on the interval basis. We assume that in every interval, every data object $k \in \mathcal{K}$ has either no or just one new updated value. Note that our proposed mechanism can be easily adopted to support multiple updated values in one interval.

In view of the aforementioned attacks, we aim to design a freshness authentication mechanism to allow a user to verify whether the query result returned by the cloud server satisfies the following two conditions.

- *Query-result integrity*: The returned data value v is indeed an updated value for key k from the data owner and has not been tampered with.
- *Query-result freshness*: There is no update for key k in any interval that starts after t and ends no later than t_q.

In other words, we aim to achieve relaxed real-time freshness verification because it cannot guarantee no update for key k in the interval that encloses t_q. The smaller the interval size, the stronger the real-time guarantee, and vice versa. We aim to support strong real-time guarantee with millisecond-based interval. Moreover, the mechanism should incur low update cost between the data owner and the cloud server and low communication and computation cost for proving data freshness.

IV. KV-Fresh

In this section, we first introduce two strawman approaches for freshness authentication followed by an overview of KV-Fresh. We then introduce a novel data structure that underpins KV-Fresh. Finally, we detail the design of KV-Fresh.

A. Two Strawman Approaches

We first introduce two strawman approaches to enable query-result freshness and integrity verification. The first approach is to let the data owner maintain the most recent update for every key and build a Merkle hash tree over all data records in every interval, some of which are updated in the current interval and the rest are copied from the previous interval. The data owner pushes the Merkle hash tree to the cloud server. With the Merkle hash tree constructed for every interval, the cloud server can prove the integrity and freshness of the query result. This approach incurs low communication cost for proving data freshness but excessively high update cost between the data owner and cloud server, as the data owner has to transmit information for every key even if many have no update in the short interval. In particular, the update cost between the data owner and the cloud server is linear to the size of the key space.

The second approach is to let the cloud server construct a Key-Ordered Merkle Hash Tree (KOMT) for every interval over only keys with update, where the absence of a key implicitly indicates that the most recent update for this key happened in one of the previous intervals. Given a batch of key-value records, the data owner sorts the records according to their keys and builds a Merkle hash tree over the sorted list. Doing so can minimize the communication cost between the data owner and the cloud server due to fewer leaf nodes in each KOMT. However, it still incurs high communication cost for proving data freshness if each key is updated infrequently, as the cloud server needs to prove that there is no update in possibly many intervals after the most recent update. More importantly, the number of intervals after the most recent update is inversely proportional to the size of interval, which means that strong real-time guarantee would incur significant communication cost for proving data freshness.

B. Overview Of KV-Fresh

KV-Fresh is designed to take the advantages of both approaches by striking a good balance between update cost and freshness proof size. In particular, the first strawman approach achieves low communication cost for proving data freshness by copying the most recent update to the Merkle hash tree constructed for the current interval. Doing so allows the cloud server to prove data freshness using the Merkle hash tree constructed for the current interval. In contrast, the second approach achieves low update cost between the data owner and the cloud server by greatly reducing the number of leaf nodes of the Merkle hash tree constructed for every interval. We find that the key to realize efficient freshness authentication with strong real-time guarantee is to simultaneously maintaining small Merkle hash tree size while realizing efficient proof of no update after the most recent update.

Based on the above observation, we introduce *Linked Key Span Merkle Hash Tree (LKS-MHT)*, a novel data structure to achieve small Merkle hash tree size in every interval while allowing efficient proof of no update in possibly many intervals. The key idea is to bundle adjacent keys with no update in one interval as a key block to reduce the number of leaf nodes. To enable efficient proof of no update over multiple intervals, each key block embeds the index of an earlier interval if none of the key in the block has update after the earlier interval. This allows the cloud server to skip possibly many intervals in between in the freshness proof. LKS-MHT can effectively break the linear dependence between the freshness proof size

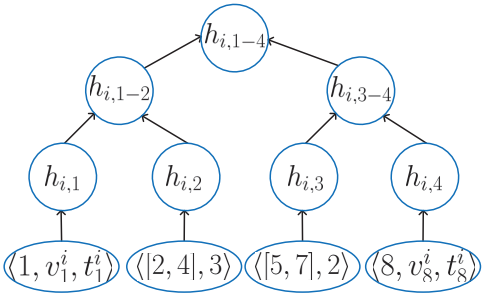


Fig. 1: An example of LKS-MHT.

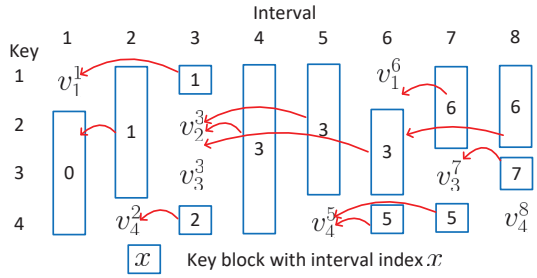


Fig. 2: Illustration of LKS-MHT-based freshness authentication

and the number of intervals with no update and thus enable highly efficient freshness authentication.

Under KV-Fresh, the data owner builds one LKS-MHT for every interval and pushes the LKS-MHT to the cloud server. The LKS-MHT contains information for every key in the key space, either an updated value received in the current interval or an index of an earlier interval, for which the LKS-MHT contains the most recent update or the index of another earlier interval. On receiving a GET query from the end user, the cloud server returns the LKS-MHT leaf node containing the queried key in the queried interval. If there is no update for the key in the queried interval, the cloud server further returns the LKS-MHT leaf node for the interval with index embedded in the leaf node of the queried interval. This process continuous until the most recent update for the queried key is found. In what follows, we first introduce LKS-MHT and its construction and then detail the operations of KV-Fresh.

C. LKS-MHT:Linked Key Span Merkle Hash Tree

We now introduce LKS-MHT, the data structure that underpins KV-Fresh. A LKS-MHT T_i is a binary tree constructed for each interval i with θ_i leaf nodes $L_{i,1}, \ldots, L_{i,\theta_i}$. Every leaf node $L_{i,j}, 1 \leq j \leq \theta_i$, consists of the following information.

(1) A key block $K_{i,j} = [l_{i,j}, r_{i,j}]$ with $l_{i,j}, r_{i,j} \in \mathcal{K}$ and $l_{i,j} \leq r_{i,j}$. If $l_{i,j} = r_{i,j}$, then $K_{i,j}$ represents a single key $l_{i,j}$.

(2.a) An interval index $\gamma_{i,j} \in \{0, \ldots, i-1\}$ that indicates that there is no update for any key in $K_{i,j}$ from interval $\gamma_{i,j}+1$ to i. In other words, the information about the most recent update for each key in $K_{i,j}$ can be found in interval $\gamma_{i,j}$ or earlier.

(2.b) Or an updated key value v_k^i along with timestamp t_k^i, if $K_{i,j}$ represents a single key k (i.e., $k = l_{i,j} = r_{i,j}$) which receives an update in interval i.

Given $L_{i,1}, \ldots, L_{i,\theta_i}$, the LKS-MHT is constructed similar to the traditional Merkle hash tree. In particular, we first calculate $h_{i,j} = H(L_{i,j})$ for all $1 \leq j \leq \theta_i$, where $H(\cdot)$ denotes a cryptographic hash function, e.g., SHA-256. We then computes every internal node as the hash of the concatenation of its two children. Note that if the number of leaf nodes is not a perfect power of two, some dummy leaf nodes need be introduced.

Fig. 1 shows an example of the LKS-MHT constructed for an interval i with the key space $\mathcal{K} = \{1, \ldots, 8\}$. The first leaf node corresponds to key $K_{i,1} = 1$ with the updated value v_1^i and timestamp t_1^i received in interval i; the second leaf node corresponds to a key block $K_{i,2} = [2, 4]$ and an interval index 3, meaning that the most recent information for keys in $[2, 4]$ can be found in interval 3 or earlier; the third leaf node corresponds a key block $K_{i,3} = [5, 7]$ and an interval index 2, meaning that the most recent information about any key in $[5, 7]$ can be found in interval 2 or earlier; and the last leaf node corresponds to key $K_{i,1} = 8$ with updated value v_8^i and timestamp t_8^i.

To see how LKS-MHT can be used to realize efficient freshness authentication, consider Fig. 2 as an example, where eight LKS-MHTs $T_1, \ldots, T_8$ are constructed for intervals 1 to 8 over key space $\mathcal{K} = \{1, 2, 3, 4\}$. Assume that the user issues a GET query as $\mathsf{Q}(2, t_q)$, where t_q is the end of interval 8. Since the most recent update for key 2 is v_2^3 received in interval 3, the cloud server needs to prove that there has been no update in intervals 4 to 8. To do so, the cloud server only needs to return the first leaf node in LKS-MHT T_8, which is a key block $[1, 2]$ and embeds an interval index 6, and the second leaf node in LKS-MHT T_6, which is a key block $[2, 3]$ and embeds an interval index 3, and the second leaf node in LKS-MHT T_3, which is a single key 2 with updated value v_2^3. As we can see, there is no need for the cloud server to return any information about intervals 4, 5, and 7.

D. LKS-MHT Construction

Now we discuss how to construct LKS-MHT T_i for each interval i, for which the key is to determine the set of key blocks with corresponding interval index. Denote by $\mathcal{K}_i \subseteq \mathcal{K}$ the subset of keys that receive updates in each interval $i \in \{1, 2, \ldots\}$. Without loss of generality, suppose $\mathcal{K}_i = \{k_{i,1}, k_{i,2}, \ldots, k_{i,\lambda_i}\}$, where $\lambda_i = |\mathcal{K}_i|$ and $k_{i,1} < k_{i,2} < \cdots < k_{i,\lambda_i}$.

1) The First Interval: It is straightforward to determine the leaf nodes of LKS-MHT T_1. We can see that the λ_1 keys $\mathcal{K}_1 = \{k_{1,1}, k_{1,2}, \ldots, k_{1,\lambda_1}\}$ partition the whole key space $\mathcal{K} = \{1, \ldots, K\}$ into $\lambda_1 + 1$ key blocks $B_1 = [1, k_{1,1} - 1]$, $B_2 = [k_{1,1} + 1, k_{1,2} - 1], \ldots, B_{\lambda_1+1} = [k_{1,\lambda_1} + 1, K]$. For simplicity, we assume that none of these key blocks are empty,

Algorithm 1: Construct candidate leaf nodes

input : Leaf nodes $L_{i-1,1}, \ldots, L_{i-1,\theta_{i-1}}$ and $\mathcal{K}_i$
output: An ordered list of candidate leaf nodes for T_i

1 $C_i \leftarrow$ emptylist;
2 **foreach** $j \in \{1, \ldots, \theta_{i-1}\}$ **do**
3 $K_{i,j} \leftarrow K_{i-1,j}$;
4 **if** $L_{i-1,j} = \langle k, v_k^{i-1}, t_k^{i-1} \rangle$ **then**
5 $\gamma_{i,j} = i - 1$;
6 **end**
7 **else if** $L_{i-1,j} = \langle [l_{i-1,j}, r_{i-1,j}], \gamma_{i-1,j} \rangle$ **then**
8 $\gamma_{i,j} = \gamma_{i-1,j}$;
9 **end**
10 Append $C_{i,j} = \langle K_{i,j}, \gamma_{i,j} \rangle$ to C_i;
11 **end**
12 **foreach** $k_{i,j} \in \mathcal{K}_i$ **do**
13 Find $C_{i,x} \in C_i$ such that $k_{i,j} \in K_{i,x}$;
14 Delete $C_{i,x}$ from C_i;
15 Insert $C_i^* = \langle k_{i,j}, v_{k_{i,j}}^i, t_{k_{i,j}}^i \rangle$ after $C_{i,x-1}$;
16 **if** $k_{i,j} > l_{i,x}$ **then**
17 Insert $\langle [l_{i,x}, k_{i,j} - 1], i \rangle$ before C_i^*;
18 **end**
19 **if** $k_{i,j} < r_{i,x}$ **then**
20 Insert $\langle [k_{i,j} + 1, r_{i,x}], i \rangle$ after C_i^*;
21 **end**
22 **end**
23 **return** C_i;

from which we can form $\theta_i = 2\lambda_1 + 1$ key blocks $\{K_{1,j}\}_{j=1}^{\theta_i}$, where

$$K_{1,j} = \begin{cases} B_{(j+1)/2}, & \text{if } j \text{ is odd,} \\ k_{1,j/2}, & \text{if } j \text{ is even,} \end{cases}$$

for all $1 \leq j \leq \theta_i$. We then create one leaf node $L_{1,j}$ for each key block $K_{1,j}$, where

$$L_{1,j} = \begin{cases} \langle B_{(j+1)/2}, 0 \rangle, & \text{if } j \text{ is odd,} \\ \langle k_{1,j/2}, v_{k_{j/2}}^1, t_{k_{j/2}}^1 \rangle, & \text{if } j \text{ is even.} \end{cases}$$

2) Subsequent Intervals: For every subsequent interval i ($i \geq 2$), the leaf nodes of T_i are determined jointly by the leaf nodes of T_{i-1} and $\mathcal{K}_i$ in two steps: (1) constructing a set of candidate leaf nodes and (2) determining the leaf nodes.

Candidate leaf nodes. First, we can obtain a set of candidate leaf nodes based on $L_{i-1,1}, \ldots, L_{i-1,\theta_{i-1}}$, and $\mathcal{K}_i$. Consider as an example a leaf node $L_{i-1,j}$ with key block $K_{i-1,j} = [l_{i-1,j}, r_{i-1,j}]$ and interval index $\gamma_{i-1,j} < i$. Assume that $|K_{i-1,j}| \geq 2$. If no key in $K_{i-1,j}$ receives any update in interval i, we create one candidate leaf node the same as $L_{i-1,j}$. Otherwise, we split $K_{i-1,j}$ into multiple non-overlapping key blocks and create one candidate leaf node from each of them. Each candidate leaf node either contains a key with update in interval i or a key block with no update that inherits the interval index $\gamma_{i-1,j}$ from $L_{i-1,j}$. For example, if a single key $k \in K_{i-1,j}$ is updated in interval i and $l_{i-1,j} < k < r_{i-1,j}$, we can split $K_{i-1,j}$ into three smaller candidate blocks and create three candidate leaf nodes: the first one with key block $[l_{i-1,j}, k-1]$ and the same interval index $\gamma_{i-1,j}$, the second one consisting of a single key k, the updated value v_k^i, and timestamp t_k^i, and the third one with key block $[k+1, r_{i-1,j}]$ and the same interval index $\gamma_{i-1,j}$.

We summarize the general procedure for constructing a list

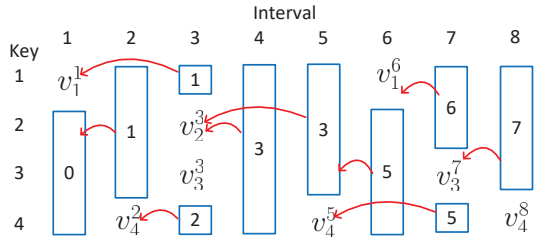


Fig. 3: An example of LKS-MHTs constructed under maximum merging.

of candidate leaf nodes in Algorithm 1, which takes a list of leaf nodes $L_{i-1,1}, \ldots, L_{i-1,\theta_{i-1}}$ of LKS-MHT T_{i-1} and $\mathcal{K}_i$ as input and outputs a sorted list of candidate leaf nodes C_i. The list of candidate leaf nodes is initialized as an empty list (Line 1). We then create one candidate leaf node from each leaf node $L_{i-1,j}$ where the interval index is set to $i-1$ if $L_{i-1,j}$ corresponds to a single key with update in interval $i-1$ or $\gamma_{i,j-1}$ if it corresponds to a key block (Lines 2 to 11). We then check every key $k_{i,j} \in \mathcal{K}_i$ to make necessary adjustment to the candidate leaf nodes (Lines 12 to 22). Specifically, for every $k_{i,j} \in \mathcal{K}_i$, we find the candidate leaf node $C_{i,x}$ whose key block encloses $k_{i,j}$ and replace $C_{i,x}$ with a new candidate leaf node $\langle k_{i,j}, v_{k_{i,j}}^i, t_{k_{i,j}}^i \rangle$, a candidate leaf node on the left if $k_{i,j} > l_{i,x}$, and a candidate leaf node on the right if $k_{i,j} < r_{i,x}$.

Leaf nodes. We now determine the leaf nodes for T_i from the candidate leaf nodes, for which the key is to merge some adjacent candidate leaf nodes to maintain a small number of leaf nodes. Without merging, the number of leaf nodes would increase monotonically at every interval and eventually reach $\mathcal{K}$, resulting in excessive update cost between the data owner and the cloud server similar to Strawman Approach 1.

Under what condition can adjacent candidate leaf nodes be merged? We observe that multiple adjacent candidate leaf nodes can be merged into one if none of the keys in the corresponding key blocks is updated in interval i. Specifically, for a group of adjacent candidate leaf nodes $C_{i,j}, \ldots, C_{i,j+s}$, if none of the keys in $\bigcup_{x=j}^{j+s} K_{i,x}$ is updated in interval i, then we can merge key blocks $K_{i,j}, \ldots, K_{i,s}$ into one and create a new leaf node as $\langle \bigcup_{x=j}^{j+s} K_{i,x}, i-1 \rangle$ that indicates that the most recent information about any key in $\bigcup_{x=j}^{j+s} K_{i,x}$ can be found in T_{i-1}.

Which adjacent candidate leaf nodes should be merged? A plausible answer is to merge every block of consecutive candidate leaf nodes into one leaf node to minimize the number of leaf nodes and thus the update cost. However, doing so would increase the size of freshness proof, as the cloud server needs to return information for more intervals. Fig. 3 shows an example of blindly merging all possible leaf nodes for 8 LKS-MHTs. Assume that the end user issues a GET query as $Q(2, t_q)$, where t_q is at the end of interval 8. The cloud server needs to return the first leaf node of T_8, which is a key block $[1, 3]$ and embeds an interval index 7, and the first leaf node in LKS-MHT T_7, which is a key block $[1, 2]$

and embeds an interval index 6, the second leaf node of T_6, which is a key block $[2, 4]$ and embeds an interval index 5, the first leaf node in LKS-MHT T_5, which is a key block $[1, 3]$ and embeds an interval index 3, and the second leaf node of T_2 which is a single key 2 with the updated value v_2^3. In comparison with the previous example shown in Fig. 2, the cloud server needs to return two more leaf nodes.

We first observe that some merging decisions can be made based on whether related keys have updates in the two intervals. Let $C_i = \langle C_{i,1}, \ldots, C_{i,\phi_i} \rangle$ be the list of candidate leaf nodes output by Algorithm 1, where ϕ_i is the number of candidate leaf nodes. We define b_j as the decision variable such that $b_j = 1$ if $C_{i,j}$ and $C_{i,j+1}$ are merged into one and 0 otherwise for all $1 \leq j \leq \phi_i - 1$. We find that b_j can be determined in the following two cases.

- **Case 1**: If either $C_{i,j}$ or $C_{i,j+1}$ corresponds to a single key with update in interval i, then $b_j = 0$, as the corresponding leaf node needs to record the update value and thus cannot be merged with other.
- **Case 2**: If $C_{i,j}$ and $C_{i,j+1}$ each correspond to a single key with update in interval $i-1$, i.e., $\gamma_{i,j} = \gamma_{i,j+1} = i-1$, then we should merge them into one leaf node, i.e., $b_j = 1$. Doing so can reduce the number of leaf nodes without increasing freshness proof size, because the cloud server needs to return the leaf node for at least one interval after the most recent update in interval $i-1$.

Based on the above observation, we define three index sets as $\Phi = \{1, \ldots, \phi_i - 1\}, \Phi_0 = \{j | j \in \Phi, K_{i,j} \in \mathcal{K}_i \vee K_{i,j+1} \in \mathcal{K}_i\}$ and $\Phi_1 = \{j | j \in \Phi, \gamma_{i,j} = \gamma_{i,j+1} = i-1\}$, where Φ_0 and Φ_1 correspond to the first and second cases, respectively. It follows that $b_j = 0$ for all $j \in \Phi_0$ and $b_j = 1$ for all $j \in \Phi_1$. In addition, we have $|\Phi_0| \leq |\mathcal{K}_i|$, as every key in $|\mathcal{K}_i|$ can introduce at most one element to Φ_0. We further note that if we let $b_j = 1$ for all $j \in \Phi \setminus \Phi_0$, i.e., merging every possible pair of candidate leaf nodes, then it would take $|\Phi| - |\Phi_0|$ merging operations and result in $\phi_i - (\phi_i - 1 - |\Phi_0|) = |\Phi_0| + 1$ leaf nodes. Therefore, the minimum number of leaf nodes that T_i could have is $|\Phi_0| + 1$.

We now formulate the remaining merging decisions as an optimization problem, in which we seek to minimize the expected size of freshness proof under the constraint of maximum number of leaf nodes. We observe that the size of freshness proof is linear to the number of intervals for which the cloud server needs to return a leaf node. Denote by $h_{k,i}$ and $h_{k,i-1}$ the numbers of leaf nodes the cloud server needs to return in response to queries $Q = (k, i)$ and $Q = (k, i-1)$, respectively, for all $k \in \mathcal{K}$. Also let p_k be the probability of key k being queried, where $\sum_{k \in \mathcal{K}} p_k = 1$. If every key is equally likely being queried, we then have $p_k = \frac{1}{K}$ for all $k \in \mathcal{K}$. The expected number of leaf nodes that the cloud server needs to return for freshness proof is given by

$$\mathbb{E}(h_i) = \sum_{k \in \mathcal{K}} p_k h_{k,i} = \sum_{k \in \mathcal{K}} p_k h_{k,i-1} + \sum_{k \in \mathcal{K}} p_k (h_{k,i} - h_{k,i-1}), \tag{1}$$

where $\mathbb{E}(\cdot)$ denotes expectation. Since merging decisions have no impact on the first term, minimizing $\mathbb{E}(h_i)$ is equivalent to minimizing $\sum_{k \in \mathcal{K}} p_k (h_{k,i} - h_{k,i-1})$.

Next, we analyze the relationship between $b_1, \ldots, b_{\phi_i - 1}$ and $\sum_{k \in \mathcal{K}} p_k (h_{k,i} - h_{k,i-1})$. First, we observe that $h_{k,i} - h_{k,i-1} = 1$ if key k belongs to a candidate leaf node that has been merged with another and 0 otherwise. We therefore seek to find a subset of candidate leaf nodes to be merged, denoted by $M_i \in C_i$, such that the $\sum_{C_{i,j} \in M_i} \sum_{k \in K_{i,j}} p_k$ is minimized. Let $\Phi' = \Phi \setminus (\Phi_0 \bigcup \Phi_1)$ and $\{b_j | j \in \Phi'\}$ be the remaining decision variables that need be determined. Further denote by $\Phi_1' = \{b_j = 1 | j \in \Phi'\}$ and $\Phi_0' = \{b_j = 0 | j \in \Phi'\}$ be the subsets of decision variables set to one and zero, respectively. Given Φ_1' and Φ_1, a candidate leaf node $C_{i,j}$ is merged with another, i.e., $C_{i,j} \in M_i$ if either $j-1$ or $j \in \Phi_1' \bigcup \Phi_1$. Let $\Pi = \{j | j - 1 \in \Phi_1' \bigcup \Phi_1 \vee j \in \Phi_1' \bigcup \Phi_1 \wedge j \in \Phi\}$. We have

$$\sum_{C_{i,j} \in M_i} \sum_{k \in K_{i,j}} p_k = \sum_{j \in \Pi} \sum_{k \in K_{i,j}} p_k.$$

Since Φ_1 is predetermined, we formulate the merging decisions as the following programming problem.

$$\begin{aligned} \text{minimize} \quad & f(\Phi_1') = \sum_{j \in \Pi} \sum_{k \in K_{i,j}} p_k \\ \text{subject to} \quad & \Phi_1' \subseteq \Phi', \\ & \phi_i - |\Phi_1 \bigcup \Phi_1'| \leq \max(\tau, |\Phi_0| + 1), \\ & b_j = 0, \forall j \in \Phi_0 \bigcup \Phi_0', \\ & b_j = 1, \forall j \in \Phi_1 \bigcup \Phi_1', \end{aligned} \tag{2}$$

where $\phi_i - |\Phi_1 \bigcup \Phi_1'|$ is the number of leaf nodes after $|\Phi_1 \bigcup \Phi_1'|$ merging operations and τ is a system parameter that limits the number of leaf nodes for every LKS-MHT and is usually set to be the larger the expected number of updates in each interval.

We now introduce an efficient greedy algorithm to solve the above optimization problem. Specifically, it is easy to see that objective function $f(\cdot)$ is non-negative and monotone. In addition, for any $\Phi_x \subseteq \Phi_y \subseteq \Phi'$ and $j \in \Phi' \setminus \Phi_y$ we have $f(\Phi_x \bigcup \{j\}) - f(\Phi_x) \geq f(\Phi_y \bigcup \{j\}) - f(\Phi_y)$, because the elements in $\Phi_y \setminus \Phi_x$ may have already caused $C_{i,j}$ or $C_{i,j+1}$ merged with another, resulting in smaller return from adding j. Therefore $f(\cdot)$ is also submodular. It is well known that for any function that is submodular, non-negative, and monotone, a greedy algorithm that selects the local optimal element at every step can output a solution with guaranteed approximation ratio of $1 - 1/e$, and no polynomial-time algorithm can achieve a better guarantee unless $P = NP$ [36].

Algorithm 2 shows the greedy algorithm. We first initialize the number of leaf nodes θ_i to $\phi_i - |\Phi_1|$, i.e., ϕ_i candidate nodes after $|\Phi_1|$ merging operations (Line 1). We then initialize Φ_1' to empty set and the set of remaining decision variables Φ' to $\Phi \setminus (\Phi_0 \bigcup \Phi_1)$. We then iteratively make the remaining merging decisions (Lines 4 to 9). In each iteration, we find $j^* \in \Phi'$ with the smallest $f(\Phi' \bigcup \{j^*\})$ and move j^* from Φ' to Φ_1'. This process continuous until the number of leaf nodes

Algorithm 2: Construct Leaf Nodes 1

input : Candidate leaf nodes $C_{i,1}, \ldots, C_{i,\phi_i}$, Φ, Φ_0, Φ_1, and τ
output: Φ'_1 and Φ'_0

$\theta_i \leftarrow \phi_i - |\Phi_1|$;
$\Phi'_1 \leftarrow \emptyset$;
$\Phi' \leftarrow \Phi \setminus (\Phi_0 \bigcup \Phi_1)$;
while $\theta_i > \max(\tau, |\Phi_0| + 1)$ **do**
  $j^* = \arg\min_{j \in \Phi'} f(\Phi' \bigcup \{j\})$;
  $\Phi'_1 \leftarrow \Phi'_1 \bigcup \{j^*\}$;
  $\Phi' \leftarrow \Phi' \setminus \{j^*\}$;
  $\theta_i \leftarrow \theta_i - 1$;
end
$\Phi'_0 \leftarrow \Phi' \setminus \Phi'_1$;
return Φ'_1 and Φ'_0;

θ_i reaches $\max(\tau, |\Phi_0| + 1)$. Finally, Φ'_1 and $\Phi'_0 = \Phi' \setminus \Phi'_1$ are output for constructing the leaf nodes for T_i.

E. Detailed Procedures

We now detail the procedures involved in KV-Fresh, which consists of three phases: *update preprocessing*, *query processing*, and *query-result verification*. We assume that the data owner has a public/private key pair that supports batch verification of digital signatures, such as RSA [37].

1) Update Preprocessing: Assume that the data owner receives data records $\{\langle v_k^i, t_k^i \rangle | k \in \mathcal{K}_i\}$ in each interval i for $i = 1, 2, \ldots$. At the end of each interval i, the data owner generates the leaf nodes $L_{i,1}, \ldots, L_{i,\theta}$ according to the procedures in Section IV-D1 if $i = 1$ or Section IV-D2 otherwise. The data owner then constructs an LKS-MHT T_i over $L_{i,1}, \ldots, L_{i,\theta}$. Let (n, e) and d be the data owner's RSA public/private key pair and R_i the root of T_i. The data owner signs the concatenation of interval index i and R_i as

$$s_i = H(i||R_i)^d \mod n. \tag{3}$$

Finally, the data owner sends all the leaf nodes $L_{i,1}, \ldots, L_{i,\theta_i}$ and its signature s_i to the cloud server, whereby the cloud server can compute all the intermediate nodes and root of T_i.

2) Query Processing: Assume that a data user issues a GET query $Q(k, t_q)$ asking for most recent update for key k as of the end of interval q_1. Also assume that v_k^i is the most recent update for key k received at time t_k^i in interval i, where $i \leq q_1$.

Given $T_1, \ldots, T_{q_1}$, the cloud server constructs the query result as follows. For every $x = 1, 2, \ldots$, the cloud server finds the leaf node L_{q_x,j_x} in LKS-MHT T_{q_x} such that $k \in K_{q_x,j_x}$. It follows that $L_{q_x,j_x} = \langle k, v_k^i, t_k^i \rangle$ if $q_x = i$ and $\langle K_{q_x,j_x}, \gamma_{q_x,j_x} \rangle$ otherwise. The cloud server returns

$$\mathsf{R}_x = \langle q_x, L_{q_x,j_x}, \mathcal{A}(R_{q_x}|L_{q_x,j_x}), s_{q_x} \rangle$$

as a partial query result, where R_{q_x} is the root of LKS-MHT T_{q_x}, and $\mathcal{A}(R_{q_x}|L_{q_x,j_x})$ is the set of internal nodes in T_{q_x} needed for computing root R_{q_x} from leaf node L_{q_x,j_x}. If $q_x > i$, then the cloud server set $q_{x+1} = \gamma_{q_x,j_x}$ and repeat the above process until $q_x = i$, i.e., the most recent update for key k received in interval i has been returned.

3) Query-Result Verification: Assume that the user has received the query result in the form of $\mathsf{R} = \langle \mathsf{R}_1, \ldots, \mathsf{R}_r \rangle$, where $\mathsf{R}_x = \langle q_x, L_{q_x,j_x}, \mathcal{A}(R_{q_x}|L_{q_x,j_x}), s_{q_x} \rangle$, for all $1 \leq x \leq r$. The data user first verifies the integrity of the query result. Specifically, for every $x = 1, \ldots, r$, the user first computes R_{q_x} from L_{q_x,j_x} using $\mathcal{A}(R_{q_x}|L_{q_x,j_x})$. It then verifies all r signatures in batch by checking whether

$$(\prod_{x=1}^{r} s_{q_x})^e \stackrel{?}{=} \prod_{x=1}^{r} H(q_x||R_{q_x}) \mod n,$$

where (n, e) is the data owner's RSA public key. If so, the user considers the query result authentic.

The data user also proceeds to verify the freshness of the query result using the interval indexes embedded in the returned leaf nodes. Assume that $q_1 > \cdots > q_s$. The user first checks if $q_s = q_1$, as the cloud server should always return one leaf node for the queried interval q_1. If so, the user further checks whether $q_{x+1} = \gamma_{q_x,j_x}$ for all $x = 1, \ldots, s-1$. Finally, the user verifies whether leaf node L_{q_x,j_x} contains the updated value v_k^i and timestamp t_k^i. If so, the user considers the query result fresh and accepts v_k^i as the most recent.

TABLE I: Default Settings

Para.	Val.	Description.
ϵ	10 ms	The interval size
$\|\mathcal{K}\|$	10,000	The number of keys
m	1,000	The number of intervals
τ	512	The maximal number of key blocks
$\|H(\cdot)\|$	256	The length of hash
$\|s_i\|$	1024	The length of data owner's signature

V. Performance Evaluation

In this section, we evaluate the performance of KV-Fresh.

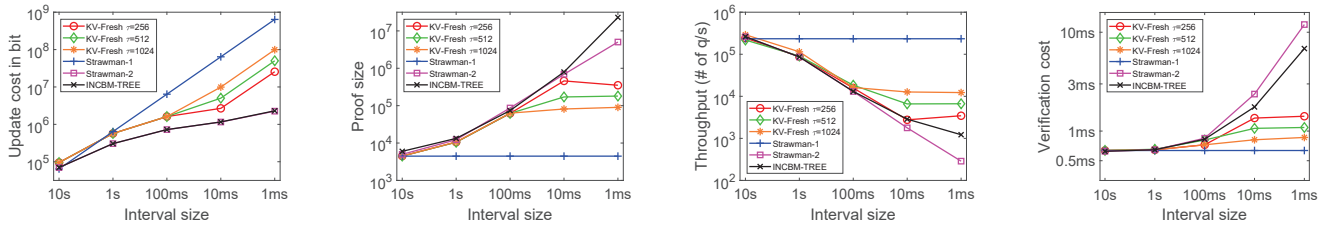
A. Dataset

We create a synthetic dataset from a TrueFax real-time currency conversion dataset [38] that includes tick-by-tick historical conversion rates for 16 major currency pairs with fractional pip spreads in millisecond detail. For our purpose, we take the currency conversion rate from EUR to USD from 12:00 am (GMT), January 2nd, 2019 to 03:46:40 pm (GMT) January 3rd, 2019. We divide the time period into 10,000 segments of 10 seconds. We treat the segment indexes as keys and the conversion rates as the updates. Our synthetic dataset consists 10,000 keys for a period of 10 seconds, and on average 131.55 keys receive updates for every 10 ms.

B. Simulation Settings

We implement KV-Fresh in Python and test it on a desktop with i7-6700 CPU, 16GB RAM and 64-bit Win10 operating system. We adopt the SHA-256 for the cryptographic hash function and RSA for digital signature. Table I summarizes our default settings unless mentioned otherwise.

We compare KV-Fresh with the state-of-art solution INCBM-TREE [29] as well as the Strawman-1 and Strawman-2 approaches discussed in Section IV-A using four metrics: (1) *update cost* which is number of extra bits per second



(a) Update cost vs. intervals size (b) Proof size vs. intervals size (c) Throughput vs. intervals size (d) Verification cost vs. intervals size

Fig. 4: Comparison of KV-Fresh, Strawman-1, Strawman-2, and INCBM-TREE with interval size varying from 10s to 1ms.

transmitted from the data owner to cloud server, (2) *proof size* which is the number of extra bits needed for proving the integrity and freshness for a query result, (3) *throughput* which is the number of queries processed by the cloud server per second, and (4) *verification time* which is the time needed for verifying a returned query result by the user.

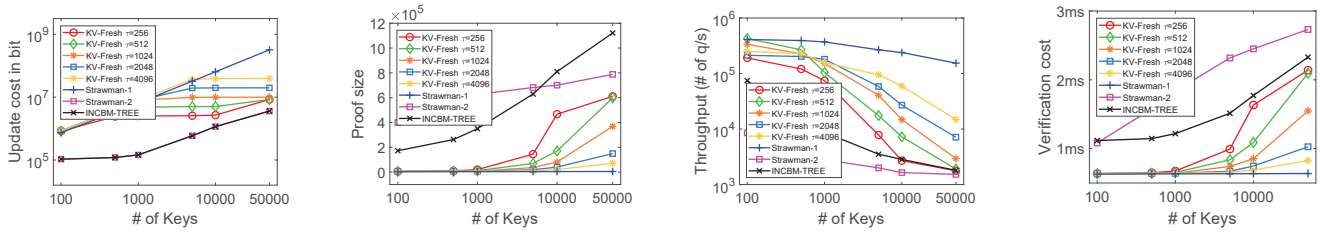
C. Simulation Results

We now report our simulation results where every point represents the average of 10,000 runs.

1) The Impact of Interval Size: Fig. 4(a) compares the update cost under Strawman-1, Strawman-2, INCBM-TREE, and KV-Fresh for $\tau = 256, 512$ and 1024 with interval size varying from 10s to 1ms. As we can see, the update cost per second increases as the interval sizes decreases under all four mechanisms. This is expected, as the number of intervals is inversely proportional to the interval size. Among the four mechanisms, Strawman-1 has the highest update cost when the interval size is smaller than 1s, as the data owner needs to send the most recent key-value record for every key in every interval. Strawman-2 and INCBM-TREE have the lowest update cost, as the data owner only sends keys with updates under both mechanisms. The update cost of KV-Fresh falls in the middle and increases much slower than that of Strawman-1. This is anticipated, as KV-Fresh requires the data owner to transmit updated key-value records and key block information with no update for every interval. Moreover, the smaller τ, the fewer leaf nodes of LKS-MHT in each interval, the lower update cost under KV-Fresh, and vice versa. We can see that even when the interval size is 1 ms, KV-Fresh with $\tau = 1024$ incurs an update cost of approximately 10^8 bits per second. In other words, a 100-Mbps link between the data owner and the cloud server suffices to support a key space of 10,000 keys, which makes KV-Fresh very practical.

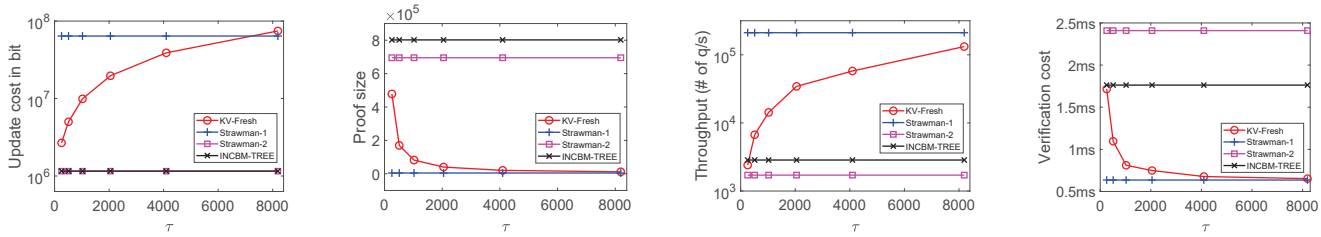
Fig. 4(b) shows the impact of interval size on the proof size of Strawman-1, Strawman-2, INCBM-TREE, and KV-Fresh. The proof size of Strawman-1 is not affected by the interval size and stays at 4460 bits. The proof sizes of the other three mechanisms all increase as the interval size decreases, except for KV-Fresh with $\tau = 256$. Among the other three, the proof sizes of Strawman-2 and INCBM-TREE grows the fastest are approximately inversely proportional to the interval size. The reason is that the data owner needs to prove that there is no update in every interval after the most recent update under both mechanisms. While INCBM-TREE employs a Bloom filter for efficient proof of no update, every Bloom filter covers only a constant number of intervals, and transmitting the Bloom filter incurs additional communication cost in comparison with Strawman-2. In contrast, the proof size under KV-Fresh grows much slower as the interval size decreases, because KV-Fresh allows the cloud server to skip potentially many intervals in the freshness proof. We can also see that the higher τ, the smaller the proof size when interval size reaches 10 ms and 1 ms. The reason is that the smaller the interval size, the fewer keys have updates in every interval, the fewer merging operations are needed for larger τ, and thus the fewer intervals, i.e., leaf nodes, need be returned under KV-Fresh. This is also the reason that we see the decrease in the proof size for KV-Fresh with $\tau = 256$ when interval size decreases from 10 ms to 1ms. In addition, we can see that KV-Fresh significantly outperforms INCBM-TREE when interval size is small. For example, when the interval size is 1 ms, the proof size under KV-Fresh with $\tau = 1024$ is approximately 90 Kb, which is less than 0.4% of the 22.9 Mb under INCBM-TREE.

Fig. 4(c) shows the throughput under Strawman-1, Strawman-2, INCBM-TREE, and KV-Fresh. The throughput under Strawman-1 is the highest and is not affected by the change in interval size. Among the other three, the throughput of Strawman-2 is the smallest, followed by INCBM-TREE. The reason is that the smaller the interval size, the more intervals after the most recent update, the more intervals the cloud server needs to process under Strawman-2 and INCBM-TREE, and vice versa. In contrast, the throughput of KV-Fresh initially declines as the interval size decreases from 10 s to 10 ms and then becomes stable or increases slightly as the interval size decreases from 10 ms to 1 ms. The reason for the initial decline is that when the interval size is large, most of the keys have updates in every interval, and the merging constraint is determined by $|\Phi_0|$ instead of τ, which results in excessive merging operations and more intervals that the cloud server needs to check. As the interval size further decreases, fewer and fewer keys have updates in each interval, which result in fewer merging operations and thus fewer intervals the cloud server needs to check. Generally speaking, in comparison with Strawman-2 and INCBM-TREE, KV-Fresh has similar throughput when the interval size is large while outperforms



(a) Update cost vs. $|\mathcal{K}|$ (b) Proof size vs. $|\mathcal{K}|$ (c) Throughput vs. $|\mathcal{K}|$ (d) Verification cost vs. $|\mathcal{K}|$

Fig. 5: Comparison of KV-Fresh, Strawman-1, Strawman-2, and INCBM-TREE with $|\mathcal{K}|$ varying from 100 to 50,000.



(a) Update cost vs. τ (b) Proof size vs. τ (c) Throughput vs. τ (d) Verification cost vs. τ

Fig. 6: Comparison of KV-Fresh, Strawman-1, Strawman-2, and INCBM-TREE with τ varying from 256 to 10,000.

Strawman-2 and INCBM-TREE by large margins when the interval size is small.

Fig. 4(d) compares the verification cost of the four mechanisms under different interval sizes. As we can see, the verification cost of Strawman-1 remains at 0.6357ms and is not affected by the change in interval size. The verification cost increases as the interval size decreases under all the other three mechanisms. Among them, KV-Fresh has the lowest verification cost and outperforms INCBM-TREE and Strawman-2 by large margins. The reason is that fewer leaf nodes need be returned under KV-Fresh than both INCBM-TREE and Strawman-2. These results demonstrate the significant advantages of KV-Fresh over other two mechanisms.

2) The Impact of the Number of Keys: Figs. 5(a) to 5(d) compares the performance of KV-Fresh, Strawman-1, Strawman-2 and INCBM-TREE with $|\mathcal{K}|$, i.e., the number of keys, varying from 100 to 50,000. As we can see from Fig. 5(a), the update costs of all schemes increase as the number of keys increase, which is anticipated. Moreover, the update cost of KV-Fresh is lower than that of Strawman-1 by a larger margin and higher than that of Strawman-2 and INCBM-TREE. Even for KV-Fresh with $\tau = 4096$, the update cost is approximately 3.9×10^7 bits per second, which is very practical for $\mathcal{K} = 50,000$ and 10-ms interval. Moreover, we can see from Fig. 5(b) that the proof size under all four mechanisms increase as $|\mathcal{K}|$ increases, as larger $|\mathcal{K}|$ leads to deeper MHT. Moreover, as $|\mathcal{K}|$ increases from 100 to 50,000, the proof size under KV-Fresh is always significantly smaller than that under Strawman-2 and INCBM-TREE. Similarly, Figs. 5(c) and 5(d) show that KV-Fresh achieves much higher throughput and lower verification cost than Strawman-2 and INCBM-TREE, because fewer leaf nodes need be returned under KV-Fresh than the other two.

3) The Impact of τ: Figs. 6(a) to 6(d) shows the performance of KV-Fresh with varying τ, where the performance of Strawman-1, Strawman-2 and INCBM-TREE are plotted for reference. Generally speaking, the larger τ, the higher update cost, the smaller proof size, the higher throughput, the smaller verification cost for KV-Fresh, and vice versa. In addition, the update cost, proof size, throughput, and verification cost of KV-Fresh are almost always between those under Strawman-1, Strawman-2, and INCBM-TREE, which is expected. While KV-Fresh incurs higher update cost than Strawman-2 and INCBM-TREE, it incurs much lower communication cost between the cloud server and the ender and smaller verification cost. Moreover, while update only happens between the data owner and the cloud server, the cloud server could serve potentially many users at the same time.

VI. Conclusion

In this paper, we have presented the design and evaluation of KV-Fresh, a novel freshness authentication scheme for outsourced multi-version key-value stores. KV-Fresh is built upon LKS-MHT, a novel data structure that allows efficient proof of no update over a potentially large number of intervals. Extensive simulation studies confirm that KV-Fresh can simultaneously achieve strong real-time guarantee and high communication efficiency.

Acknowledgement

The authors would like to thank the anonymous reviewers for their constructive comments and helpful advice. This work was supported in part by the US National Science Foundation under grants CNS-1933047, CNS-1718078, CNS-1651954 (CAREER), CNS-1700039, CNS-1824355, CNS-1619251, CNS-1514381, and CNS-1933069.

REFERENCES

[1] P. Felber, M. Pasin, . Rivière, V. Schiavoni, P. Sutra, F. Coelho, R. Oliveira, M. Matos, and R. Vilaça, "On the support of versioning in distributed key-value stores," in *IEEE SRDS*, Nara, Japan, Oct 2014, pp. 95–104.

[2] S. Bhattacherjee and A. Deshpande, "Rstore: A distributed multi-version document store," in *IEEE ICDE*, Paris, France, April 2018, pp. 389–400.

[3] "Nosql market is expected to reach 4.2 billion, globally, by 2020," https://www.alliedmarketresearch.com/press-release/NoSQL-market-is-expected-to-reach-4-2-billion-globally-by-2020-allied-market-research.html.

[4] H. Hacigumus, B. Iyer, and S. Mehrotra, "Providing database as a service," in *IEEE ICDE*, San Jose, CA, Feb 2002.

[5] M. Narasimha and G. Tsudik, "Authentication of outsourced databases using signature aggregation and chaining," in *DASFAA'06*, Singapore, Apr. 2006, pp. 420–436.

[6] H. Pang and K.-L. Tan, "Verifying completeness of relational query answers from online servers," *ACM Trans. Inf. Syst. Secur.*, vol. 11, no. 2, pp. 1–50, 2008.

[7] H. Pang, J. Zhang, and K. Mouratidis, "Scalable verification for outsourced dynamic databases," *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 802–813, 2009.

[8] J. Shi, R. Zhang, and Y. Zhang, "Secure range queries in tiered sensor networks," in *IEEE INFOCOM*, Rio de Janeiro, Brazil, Apr. 2009.

[9] R. Zhang, J. Shi, and Y. Zhang, "Secure multidimensional range queries in sensor networks," in *ACM MobiHoc'09*, New Orleans, LA, May 2009, pp. 197–206.

[10] Y. Yang, S. Papadopoulos, D. Papadias, and G. Kollios, "Authenticated indexing for outsourced spatial databases," *The VLDB Journal*, vol. 18, no. 3, pp. 631–648, Jun. 2009.

[11] H. Hu, J. Xu, Q. Chen, and Z. Yang, "Authenticating location-based services without compromising location privacy," in *ACM SIGMOD'12*, Scottsdale, AZ, May 2012, pp. 301–312.

[12] J. Shi, R. Zhang, and Y. Zhang, "A spatiotemporal approach for secure range queries in tiered sensor networks," *IEEE Transactions on Wireless Communications*, vol. 10, no. 1, pp. 264–273, Jan. 2011.

[13] R. Zhang, J. Shi, Y. Liu, and Y. Zhang, "Verifiable fine-grained top-k queries in tiered sensor networks," in *INFOCOM'10*, San Diego, CA, Mar. 2010.

[14] Q. Chen, H. Hu, and J. Xu, "Authenticating top-k queries in location-based services with confidentiality," *Proceedings of the VLDB Endowment*, vol. 7, no. 1, pp. 49–60, Sep. 2013.

[15] R. Zhang, Y. Zhang, and C. Zhang, "Secure top-k query processing via untrusted location-based service providers," in *IEEE INFOCOM*, Orlando, FL, Mar. 2012.

[16] R. Zhang, J. Shi, Y. Zhang, and X. Huang, "Secure top-k query processing in unattended tiered sensor networks," *IEEE Transactions on Vehicular Technology*, vol. 63, no. 9, pp. 4681–4693, Nov 2014.

[17] R. Zhang, J. Sun, Y. Zhang, and C. Zhang, "Secure spatial top-k query processing via untrusted location-based service providers," *IEEE Transactions on Dependable and Secure Computing*, vol. 12, no. 1, pp. 111–124, Jan 2015.

[18] X. Lin, J. Xu, and H. Hu, "Authentication of location-based skyline queries," in *CIKM*, New York, NY, Oct. 2011, pp. 1583–1588.

[24] M. L. Yiu, Y. Lin, and K. Mouratidis, "Efficient verification of shortest path search via authenticated hints," in *IEEE ICDE*, Long Beach, CA, Mar. 2010, pp. 237–248.

[19] X. Lin, J. Xu, and J. Gu, "Continuous skyline queries with integrity assurance in outsourced spatial databases," in *WAIM'12*, Harbin, China, Aug. 2012, pp. 114–126.

[20] X. Lin, J. Xu, H. Hu, and W.-C. Lee, "Authenticating location-based skyline queries in arbitrary subspaces," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 6, pp. 1479–1493, June 2014.

[21] W. Chen, M. Liu, R. Zhang, Y. Zhang, and S. Liu, "Secure outsourced skyline query processing via untrusted cloud service providers," in *IEEE INFOCOM*, April 2016, pp. 1–9.

[22] M. L. Yiu, E. Lo, and D. Yung, "Authentication of moving knn queries," in *IEEE ICDE*, Hannover, Germany, Apr. 2011, pp. 565–576.

[23] L. Hu, W.-S. Ku, S. Bakiras, and C. Shahabi, "Spatial query integrity with voronoi neighbors," *IEEE Transactions on Knowledge and Data Engineering*, vol. 25, no. 4, pp. 863–876, Apr. 2013.

[25] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin, "Dynamic authenticated index structures for outsourced databases," in *ACM SIGMOD'06*, Chicago, IL, 2006, pp. 121–132.

[26] F. Li, K. Yi, M. Hadjieleftheriou, and G. Kollios, "Proof-infused streams: Enabling authentication of sliding window queries on streams," in *VLDB*, Vienna, Austria, Sep. 2007, pp. 147–158.

[27] E. Stefanov, M. van Dijk, A. Juels, and A. Oprea, "Iris: A scalable cloud file system with efficient integrity checks," in *ACSAC*, Orlando, FL, December 2012, pp. 229–238.

[28] H.-J. Yang, V. Costan, N. Zeldovich, and S. Devadas, "Authenticated storage using small trusted hardware," in *CCSW*, Berlin, Germany, 2013, pp. 35–46.

[29] Y. Tang, T. Wang, L. Liu, X. Hu, and J. Jang, "Lightweight authentication of freshness in outsourced key-value stores," in *ACSAC*, New Orleans, LA, 2014, pp. 176–185.

[30] S. Papadopoulos, Y. Yang, and D. Papadias, "Cads: Continuous authentication on data streams," in *VLDB*, Vienna, Austria, Sep. 2007, pp. 135–146.

[31] M. T. Goodrich, C. Papamanthou, R. Tamassia, and N. Triandopoulos, "Athos: Efficient authentication of outsourced file systems," in *Information Security Conference*, Taipei, Taiwan, 2008, pp. 80–96.

[32] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten, "Sporc: Group collaboration using untrusted cloud resources," in *OSDI*, Vancouver, BC, Canada, Oct. 2010, pp. 337–350.

[33] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, "Cryptdb: Protecting confidentiality with encrypted query processing," in *SOSP*, Cascais, Portugal, Oct. 2011, pp. 85–100.

[34] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish, "Depot: Cloud storage with minimal trust," *ACM Trans. Comput. Syst.*, vol. 29, no. 4, pp. 12:1–12:38, Dec. 2011.

[35] T. Dierks and E. Rescorla, "The transport layer security (TLS) protocol," RFC 4346, Apr. 2006.

[36] A. Das and D. Kempe, "Algorithms for subset selection in linear regression," in *STOC'08*, Victoria, British Columbia, Canada, 2008, pp. 45–54.

[37] L. Harn, "Batch verifying multiple rsa digital signatures," *Electronics Letters*, vol. 34, no. 12, pp. 1219–1220, June 1998.

[38] TrueFax, "January 2019 historical tick-by-tick data," Downloaded from https://www.truefx.com/?page=download&description=january2019&dir=2019/2019-01, Jan 2019.