Freshness Authentication for Outsourced Multi-Version Key-Value Stores

Yidan Hu, *Member, IEEE,* Xin Yao, *Member, IEEE* Rui Zhang, *Member, IEEE* and Yanchao Zhang, *Fellow, IEEE*

Abstract—Data outsourcing is a promising technical paradigm to facilitate cost-effective real-time data storage, processing, and dissemination. In data outsourcing, 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. However, the popular outsourced multi-version key-value stores pose a critical security challenge that a third-party cloud server cannot be fully trusted to return both authentic and fresh data in response to end users' queries. Although several recent attempts have been made 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 for both point query and range query. 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. Extensive simulation studies using a synthetic dataset generated from real data confirm the efficacy and efficiency of KV-Fresh.

Index Terms—Freshness authentication, data outsourcing, multi-version key-value store

1 Introduction

Data outsourcing is a promising technical paradigm to facilitate cost-effective real-time data 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. Doing so can relieve the data owner from cumbersome management work and result in significant saving in operation cost. Data outsourcing can also provide a more efficient query processing service for end users because of the higher availability and elasticity offered by cloud service providers.

This paper considers a data outsourcing system with a multi-version key-value store [1], [2] to store, analyze and access the large volume of unstructured data streams. A key-value store is a non-SQL 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. Key-value stores outperform traditional relationship database with higher scalability, simpler designs, and higher availability. Key-value stores and other non-SQL databases have gained increasing popularity in recent years, such as MongoDB, Amazon DynamoDB, Azure Cosmos DB, and so on. The market of non-SQL database is expected to reach 4.2 billion by 2020 [3].

Despite offering many advantages, data outsourcing also 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 return forged data in response to end users' queries to mislead users into making incorrect decisions. For example, a cloud service provider may intentionally delete data or return forged data in favor of the businesses with financial interests [4]. 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 [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], authenticating data freshness poses unique challenges and has thus far received very limited attention. Common to existing solutions [16], [17], [18], [19], [20] is to divide the time into intervals and let the data owner generate a cryptographic proof for every key with no update in every

- *Y. Hu is with the Department of Computing Security, Rochester Institute of Technology, Rochester, Rochester, NY 14623 E-mail: yidan.hu@rit.edu.*
- *R. Zhang is with the Department of Computer and Information Sciences, University of Delaware, Newark, DE, 19716. E-mail: ruizhang@udel.edu.*
- *X. Yao is with the School of Computer Science and Engineering, Central South University, Changhsa, China. E-mail: xinyao@csu.edu.cn.*
- *Y. Zhang is with the School of Electrical, Computer and Energy Engineering, Arizona State University, Tempe, AZ, 85287. E-mail: yczhang@asu.edu*
- *R. Zhang and X. Yao are both corresponding authors.*

Manuscript received XXX XX, 2020; revised XXX XX, XXX.

interval. On receiving a point 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 [16], [17], [18], [19], [20] either suffer from excessively high communication cost or can only support limited real-time guarantee. For example, the state-of-art solution [20] can only support interval size in minutes. Moreover, none of the existing solutions [16], [17], [18], [19], [20] can support efficient range queries. While a range query can be implemented by multiple point queries, doing so would incur a communication cost proportional to the number of keys in the queried range. There is thus a pressing need to develop efficient mechanisms for freshness authentication with strong real-time guarantee while supporting both point and range queries.

In this paper, we tackle this open challenge by introducing KV-Fresh, a novel freshness authentication mechanism for outsourced multi-version key-value store supporting both point query and range query. 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 for both point query and range query.
- 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 [20].

The rest of the paper is structured as follows. Section 2 discusses the related work. Section 3 presents the system and adversary models and design goals. Section 4 introduces a novel data structure, LKS-MHT and proposes an efficient freshness authentication mechanism, KV-Fresh built upon LKS-MHT. We evaluate the performance of KV-Fresh in Section 5 and finally conclude this paper in Section 6. .

2 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 such as a Merkle Hash tree or its variants at the cloud server. In each interval, the data owner sends an updated data digest to the cloud server by recomputing the root of the Merkle hash tree and sign its root, which would require the data owner either maintain a full copy of the local data [16], [18], [21] or download records [17], [22]. The second category [23], [24], [25] 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 [19]. In [20], 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 [20] and enables freshness verification at much smaller time granularity.

Our work is also related to authenticating outsourced query processing, where a data owner outsources its dataset to a third party service provider which in turns answers data queries from end users on the data owner's behalf. Significant efforts have been devoted to ensuring query integrity and completeness, i.e., the query result contains all the intact data records satisfying a query. Various types of queries have been studied, including relational queries [5], [6], [7], [26], [27], [28], [29], range queries [8], [9], [30], top-k queries [13], [14], [15], [31], [32], skyline queries [10], [11], [12], [33], kNN queries [34], [35], [36], shortest-path queries [37], etc. None of these works consider the freshness of returned data records, and they are thus inapplicable to the problem addressed in this paper.

Our work is also loosely related to the lines of research on verifiable database (VDB) and secure storage outsourcing. VDB seeks to provide a resource-limited data owner the capability of storing a large database on a cloud server, retrieving and updating any data record in a verifiable way. Benabbas *et al.* [38] presented the first VDB scheme based on verifiable delegation of polynomials, which cannot support public verification, i.e., only the data owner can verify the proof returned by the cloud server. More recently, vector commitment and its variants [39], [40], [41], [42], [43] were proposed to support efficient public verification. In secure storage outsourcing, a data owner outsources the storage a large database on an untrusted cloud server and can verify that the server possesses the original data without any tampering. For example, Ateniese *et al.* [44] presented a Provable Data Possession (PDP) scheme to prove the integrity and ownership of clients' data without downloading data. As another example, Erway *et al.* [45] extended the PDP model and proposed dynamic PDP to support provable updates of stored data. Moreover, Zhu *et al.* [46] introduced

a cooperative PDP scheme to support verifiable cooperative storage over multiple cloud servers. None of these schemes can be directly applied to freshness authentication as any user other than the data owner can detect whether cloud server returns authentic but stale data records.

3 Problem Formulation

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

3.1 System Model

We consider a data outsourcing system consisting of three parties: a data owner, a third-party cloud server, and many end users. The data owner outsources a multi-version key-value store to the cloud server, which in turn answers data queries from end 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 that supports both point query and range query. Specifically, a point query is represented 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. Moreover, a range query is modeled by $Q([l, r], t_q)$, where $1 \leq l < r \leq |\mathcal{K}|$ and $[l, r]$ denotes the range of keys being queried. On receiving query $Q([l, r], t_q)$, the cloud server needs to return the most recent data records for every key $k \in [l, r]$ as of t_q. It is easy to see that point query is a special case of range query where $l = r$. For both point queries and range queries, the absence of the optional parameter t_q indicates that the user is asking for the most recent data record for a specific key or the most up-to-date records for a set of keys belonging to the key range as of now.

3.2 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 point or range 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 [47]. In addition, we also assume that the data owner cannot predict the keys that the user will query in advance.

3.3 Design Goals

Strict freshness verification—also referred to as real-time freshness check in [20]—requires the cloud server 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 [20], 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 data updates to the cloud server on the interval basis. To ease the presentation, 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 two 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*: for each queried key k, the returned data value v is indeed an updated value from the data owner and has not been tampered with.
- *Query-result freshness*: for each queried key k, there is no update in any interval that starts after t and ends before or exactly at 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 and low communication and computation costs. In particular, the mechanism should incur low update cost between the data owner and the cloud server as well as low communication and computation cost for proving data freshness.

4 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 and its construction. Finally, we detail the design of KV-Fresh.

4.1 Two Strawman Approaches

We first introduce two strawman approaches to enable query-result integrity and freshness verification.

Strawman Approach 1. The first approach adopts a similar idea [16], [18], [21], which lets 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3172380, IEEE Transactions on Dependable and Secure Computing

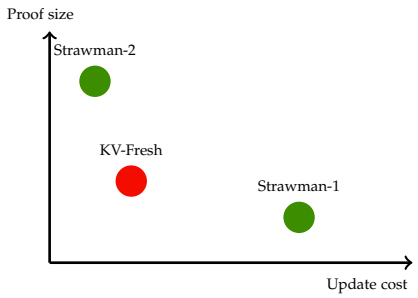


Fig. 1. Comparison of Strawman Approach 1, Strawman Approach 2 and KV-Fresh

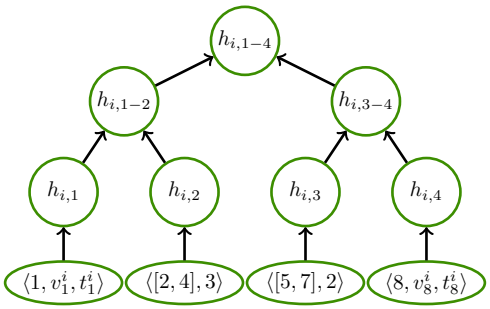


Fig. 2. An example of LKS-MHT.

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. **Strawman Approach 2**. 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, i.e., small interval size, would incur significant communication cost for proving data freshness. INCBM-TREE [20] can be viewed as a variant of Strawman Approach 2.

4.2 Overview Of KV-Fresh

KV-Fresh is designed to strike a good balance between the update cost between data owner and cloud service provider and the size of freshness proof by taking the advantages of both strawman approaches. In particular, Strawman Approach 1 achieves small freshness proof size by copying the most recent value of every key to the Merkle hash tree constructed in each interval. Doing so allows the cloud server to prove data freshness using only the Merkle hash tree constructed for the current interval. On the other hand, Strawman Approach 2 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. KV-Fresh realizes efficient freshness authentication with strong real-time guarantee by simultaneously maintaining a small Merkle hash tree size while realizing efficient proof of no update in possibly many intervals after the most recent update. Fig. 1 shows a comparison of KV-Fresh and the two strawman approaches in terms of the proof size and update cost.

KV-Fresh is built upon *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 behind the LKS-MHT is to bundle adjacent keys with no update in one interval as a key block to limit the number of leaf nodes and reduce the update cost between data owner and cloud server in comparison with Strawman Approach 1. 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 received any update after the earlier interval, which allows the cloud server to skip potentially many intervals in between in the freshness proof. LKS-MHT can effectively break the linear dependence between the freshness proof size and the number of intervals with no update and thus significantly reduce the size of freshness proof in comparison with Strawman Approach 2.

Under KV-Fresh, the data owner constructs one LKS-MHT in every interval, which 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. The data owner signs the LKS-MHT and pushes the LKS-MHT along with its signature to the cloud server. On receiving a query from the end user, the cloud server returns a list of LKS-MHT leaf nodes containing the queried key. The chaining relationship embedded in the returned leaf nodes allows the user the verify both the integrity and freshness of the query result. In what follows, we first introduce LKS-MHT and its construction and then detail the operations of KV-Fresh.

4.3 LKS-MHT:Linked Key Span Merkle Hash Tree

We now introduce LKS-MHT, the data structure that underpins KV-Fresh. An 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 fields.

(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.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3172380, IEEE Transactions on Dependable and Secure Computing

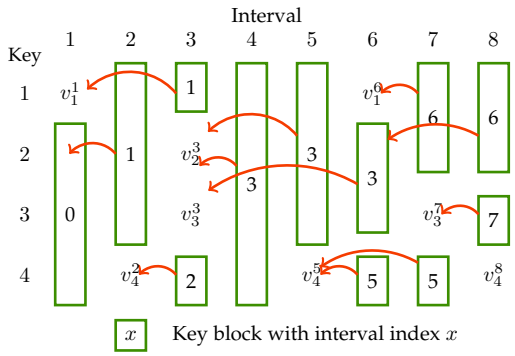


Fig. 3. Illustration of LKS-MHT-based freshness authentication

(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 such as 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. 2 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. 3 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.

In the next two subsections, we introduce how to construct the LKS-MHT for the first interval and the subsequent intervals, respectively.

4.4 LKS-MHT Construction in the First Interval

We first show how to construct LKS-MHT T_i for the first interval i ($i = 1$). Denote by $\mathcal{K}_1 \subseteq \mathcal{K}$ the subset of keys that receive updates in the first interval. Without loss of generality, suppose $\mathcal{K}_1 = \{k_{1,1}, k_{1,2}, \ldots, k_{1,\lambda_1}\}$, where $\lambda_1 = |\mathcal{K}_1|$ and $k_{1,1} < k_{1,2} < \cdots < k_{1,\lambda_1}$. We can see that the λ_1 keys, $\mathcal{K}_1 = \{k_{1,1}, k_{1,2}, \ldots, k_{1,\lambda_1}\}$, split the whole key space $\mathcal{K} = \{1, \ldots, K\}$ into λ_1+1 key blocks without update, $B_1 = [1, k_{1,1} - 1]$, $B_2 = [k_{1,1} + 1, k_{1,2} - 1], \ldots, B_{\lambda_1+1} = [k_{1,\lambda_1} + 1, K]$. For simplicity, we assume that none of these key blocks are empty, 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

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

4.5 LKS-MHT Construction in Subsequent Intervals

We now discuss how to construct LKS-MHT T_i for the subsequent interval i ($i \geq 2$), for which the key question is to determine the set of key blocks with corresponding interval index. Let $\mathcal{K}_i = \{k_{i,1}, k_{i,2}, \ldots, k_{i,\lambda_i}\}$ be the subset of keys that have received updates in the subsequent interval i, where $\lambda_i = |\mathcal{K}_i|$ and $k_{i,1} < k_{i,2} < \cdots < k_{i,\lambda_i}$. For every subsequent interval i, 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 with a single key k and 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 of candidate leaf nodes in Appendix A of the supplement file.

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 into one 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3172380, IEEE Transactions on Dependable and Secure Computing

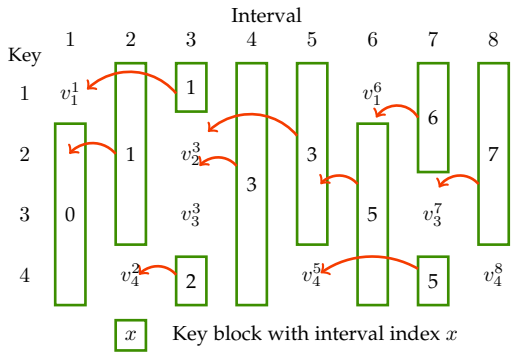


Fig. 4. An example of LKS-MHTs constructed under maximum merging.

cost between the data owner and the cloud server as in 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 $C_{i,j}, \ldots, C_{i,j+s}$ for some $s \geq 1$, if none of the keys in their respective key blocks $\bigcup_{x=j}^{j+s} K_{i,x}$ have received any update 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 $\langle \bigcup_{x=j}^{j+s} K_{i,x}, i-1 \rangle$, which 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 group of consecutive candidate leaf nodes into one to minimize the number of leaf nodes and thus the update cost between the data owner and the cloud server. However, doing so would increase the size of freshness proof, as the cloud server needs to return information for more intervals. Fig. 4 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. 3, 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 predetermined in the following two cases.

- **Case 1**: If either $C_{i,j}$ or $C_{i,j+1}$ corresponds to a single key that has received an 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 the other.
- **Case 2**: If neither $C_{i,j}$ nor $C_{i,j+1}$ each correspond to a single key that has received an update in interval $i-1$, i.e., $|K_{i,j}| = |K_{i,j+1}| = 1$ and $\gamma_{i,j} = \gamma_{i,j+1} = i-1$, then we should merge them into one, 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 observations, 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 $\Phi_1 = \{j | j \in \Phi, |K_{i,j}| = |K_{i,j+1}| = 1, \gamma_{i,j} = \gamma_{i,j+1} = i-1\}$, where Φ_0 and Φ_1 correspond to the first and second cases, respectively. In other words, $b_j = 0$ for all $j \in \Phi_0$ and $b_j = 1$ for all $j \in \Phi_1$. We further note that if we set $b_j = 1$ 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 the number of remaining leaf nodes is given by

$$\begin{aligned} \phi_i - (|\Phi| - |\Phi_0|) &= \phi_i - (\phi_i - 1 - |\Phi_0|) \\ &= |\Phi_0| + 1 \ . \end{aligned}$$

Therefore, the minimum number of leaf nodes that T_i can have is $|\Phi_0| + 1$.

We make the remaining merging decisions through an optimization approach. In what follows, we introduce two optimization problem formulations with different objective functions and present their solutions.

4.5.1 Formulation 1: Expected Freshness Proof Size Minimization

Our first formulation aims to minimize the expected size of freshness proof under the constraint of the 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 in response to a point query. Let $h_{k,i}$ and $h_{k,i-1}$ denote 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 each key k being queried, where $\sum_{k \in \mathcal{K}} p_k = 1$. If every key is equally likely being queried, we then have $p_k = 1/\mathcal{K}$ for all $k \in \mathcal{K}$. Let $\triangle h_k = h_{k,i} - h_{k,i-1}$ 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

$$\begin{aligned} \mathbb{E}(h_i) &= \sum_{k \in \mathcal{K}} p_k h_{k,i} \\ &= \sum_{k \in \mathcal{K}} p_k h_{k,i-1} + \sum_{k \in \mathcal{K}} p_k \triangle h_k \ , \end{aligned} \tag{1}$$

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

Next, we analyze the relationship between decision variables $b_1, \ldots, b_{\phi_i - 1}$ and $\sum_{k \in \mathcal{K}} p_k \triangle h_k$. First, we observe that $\triangle h_k = 1$ if key k belongs to a candidate leaf node being merged with another adjacent one and 0 otherwise. Let

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3172380, IEEE Transactions on Dependable and Secure Computing

$\Phi' = \Phi \setminus (\Phi_0 \bigcup \Phi_1)$ 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 $\Phi'_0 = \{b_j = 0 | j \in \Phi'\}$ 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 one 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

$$\sum_{k \in \mathcal{K}} p_k \triangle h_k = \sum_{j \in \Pi} \sum_{k \in K_{i,j}} p_k,$$

where $K_{i,j}$ is the key block of $C_{i,j}$.

Let $f(\Phi'_1) = \sum_{j \in \Pi} \sum_{k \in K_{i,j}} p_k$. We formulate the merging decisions as the following programming problem.

$$\begin{aligned} \text{minimize} \quad & f(\Phi'_1) \\ \text{subject to} \quad & \Phi'_1 \subseteq \Phi', \\ & \phi_i - |\Phi_1 \bigcup \Phi'_1| \leq \max(\tau, |\Phi_0| + 1), \\ & b_j = 0, \forall j \in \Phi_0 \bigcup \Phi'_0, \\ & b_j = 1, \forall j \in \Phi_1 \bigcup \Phi'_1, \end{aligned} \tag{2}$$

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 usually set to be the larger the expected number of updates in each interval. Also note that parameter τ serves as an upper bound for the number of LKS-MHT leaf nodes in every interval as the average number of keys with update in each interval is inversely proportional to the size of the interval.

We now introduce an efficient greedy algorithm to solve the above optimization problem with guaranteed approximation ratio. We can see that the objective function $f : 2^{\Phi'} \rightarrow \mathbb{R}$ is a set function, and the following theorem characterizes its properties.

Theorem 1. *The objective function $f(\cdot)$ in Eq. (2) is non-negative, submodular, and monotone.*

We give the proof in Appendix B of the supplement file.

A well known result is that for any objective function that is non-negative, submodular, and monotone, a greedy algorithm that iteratively 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 $P = NP$ [48].

We now detail the greedy algorithm for the merging decision in Algorithm 1. 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 $f(\Phi' \{j^*\})$ and move j^* from Φ' to Φ'_1. This process continuous until the number of leaf nodes 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 LKS-MHT T_i.

Algorithm 1: Minimizing Expected Proof Size

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

1 $\theta_i \leftarrow \phi_i - |\Phi_1|$;
2 $\Phi'_1 \leftarrow \emptyset$;
3 $\Phi' \leftarrow \Phi \setminus (\Phi_0 \bigcup \Phi_1)$;
4 **while** $\theta_i > \max(\tau, |\Phi_0| + 1)$ **do**
5 $\quad j^* = \arg\min_{j \in \Phi'} f(\Phi' \bigcup \{j\})$;
6 $\quad \Phi'_1 \leftarrow \Phi'_1 \bigcup \{j^*\}$;
7 $\quad \Phi' \leftarrow \Phi' \setminus \{j^*\}$;
8 $\quad \theta_i \leftarrow \theta_i - 1$;
9 **end**
10 $\Phi'_0 \leftarrow \Phi' \setminus \Phi'_1$;
11 **return** Φ'_1 and Φ'_0;

4.5.2 Formulation 2: Minimizing Maximal Size of Freshness Proof

Our second formulation seeks to minimize the maximal freshness proof size among all keys, i.e., $\max_{k \in \mathcal{K}} \{h_{k,i}\}$, under the constraint of the maximal number of leaf nodes. Note that this would require the data owner to keep track of $\{h_{k,i} | k \in \mathcal{K}\}$. Again let $h_{k,i}$ and $h_{k,i-1}$ be the number of leaf nodes that need be returned in response to queries $Q = (k, i)$ and $Q = (k, i-1)$, respectively, for all $k \in \mathcal{K}$. Recall that $\mathcal{K}_i \subseteq \mathcal{K}$ is the subset of keys that receive an update in interval i. It follows that $h_{k,i} = 1$ for all $k \in \mathcal{K}_i$. Since $h_{k,i} \geq 1$ for all $k \in \mathcal{K}$, we have

$$\max_{k \in \mathcal{K}} \{h_{k,i}\} = \max_{k \in \mathcal{K} \setminus \mathcal{K}_i} \{h_{k,i}\}. \tag{3}$$

Let $C_i^- = \{C_{i,j} | K_{i,j} \bigcap \mathcal{K}_i = \emptyset\}$, i.e., $C_{i,j}$ contains no key that receives an update in interval i. For every candidate leaf node $C_{i,j} \in C_i^-$, denote its maximum freshness proof size in response to $Q = (k, i-1)$ and $Q = (k, i)$ by $m_{i-1,j} = \max_{k \in K_{i,j}} \{h_{k,i-1}\}$ and $m_{i,j} = \max_{k \in K_{i,j}} \{h_{k,i}\}$, respectively. It follows that

$$\begin{aligned} \max_{k \in \mathcal{K} \setminus \mathcal{K}_i} \{h_{k,i}\} &= \max_{C_{i,j} \in C_i^-} \{m_{i,j}\} \\ &= \max_{C_{i,j} \in C_i^-} \{m_{i-1,j} + \triangle m_j\}, \end{aligned} \tag{4}$$

where $\triangle m_j = m_{i,j} - m_{i-1,j}$ for all $C_{i,j} \in C_i^-$.

We now analyze the relationship between decision variables $b_1, \ldots, b_{\phi_i - 1}$ and $\max_{C_{i,j} \in C_i^-} \{m_{i-1,j} + \triangle m_j\}$. Similar to Formulation 1, $\triangle m_j = 1$ if the candidate leaf node $C_{i,j}$ is merged with another and 0 otherwise. Again let $\Phi' = \Phi \setminus (\Phi_0 \bigcup \Phi_1)$ and $\{b_j | j \in \Phi'\}$ be the remaining decision variables that need be determined. Also let $\Phi'_1 = \{b_j = 1 | j \in \Phi'\}$ 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 one 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

$$\max_{C_{i,j} \in C_i^-} \{m_{i,j}\} = \max(\{m_{i-1,j} + 1\}_{j \in \Pi}, \{m_{i-1,j}\}_{j \in \Phi \setminus \Pi}). \tag{5}$$

Let $g(\Phi'_1) = \max(\{m_{i-1,j} + 1\}_{j \in \Pi}, \{m_{i-1,j}\}_{j \in \Phi \setminus \Pi})$. We formulate the remaining merging decisions as the following

optimization problem.

$$\begin{aligned} \text{minimize} \quad & g(\Phi_1') \\ \text{subject to} \quad & \Phi_1' \subseteq \Phi', \\ & \phi_i - |\Phi_1 \bigcup \Phi_1'| \leq \max(\tau, |\Phi_0| + 1), \\ & b_j = 0, \forall j \in \Phi_0 \bigcup \Phi_0', \\ & b_j = 1, \forall j \in \Phi_1 \bigcup \Phi_1'. \end{aligned} \tag{6}$$

The following theorem shows that the objective function $g(\cdot)$ is also non-negative, submodular, and monotone, for which the proof is given in Appendix B of the supplement file.

Theorem 2. *The objective function $g(\cdot)$ in Eq. (6) is non-negative, submodular, and monotone.*

We give the proof in Appendix C of the supplement file.

The above optimization problem can be solved with an efficient greedy algorithm. While choosing the local optimal with the smallest $g(\cdot)$ can lead to an efficient greedy algorithm with guaranteed approximation ratio as in the first formulation, we notice that there may be multiple choices with the same minimal $g(\cdot)$ in each step. We therefore further prioritize the merging decision that involves the new candidate leaf nodes with the smallest key block size. We detail the procedure for making the merging decisions in Algorithm **??** given in Appendix D.

4.6 Point Query Processing

We now detail the procedure of KV-Fresh for point queries, 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 [49].

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 $L_{i,1}, \ldots, L_{i,\theta_i}$ according to the procedures presented in Section 4.4 if $i = 1$ or Section 4.5 otherwise. The data owner then constructs an LKS-MHT T_i over $L_{i,1}, \ldots, L_{i,\theta_i}$. Let $pk = (n, e)$ and $sk = (d)$ be the data owner's RSA public/private key pair and R_i the root of T_i. The data owner computes

$$s_i = H(i||R_i)^d \mod n. \tag{7}$$

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.

Note that if no key receives any update in interval i, the data owner can simply resign the root of the LKS-HMT T_{i-1}, i.e., R_{i-1}, in concatenation with the new interval index i as in Eq. (7) and sends the signature to the cloud server, which results in an update cost of $O(1)$.

Query Processing. Assume that a data user issues a point query $Q(k, t_q)$ asking for the most recent data record 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 $i \leq q_1$.

The cloud server constructs the query result in a recursive fashion. Specifically, the cloud server first finds the leaf node L_{q_1,j_1} in LKS-MHT T_{q_1} such that $k \in K_{q_1,j_1}$. There are two cases. First, if $i = q_1$, then we have $L_{q_1,j_1} = \langle k, v_k^i, t_k^i \rangle$, i.e., L_{q_1,j_1} contains the most recent value for key k. Second, if $i < q_1$, then we $L_{q_1,j_1} = \langle K_{q_1,j_1}, \gamma_{q_1,j_1} \rangle$, i.e., L_{q_1,j_1} points to an earlier interval γ_{q_1,j_1}, and the cloud server continues to check LKS-MHT L_{q_2}. In general, 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

$$\mathsf{R}_x = \langle q_x, L_{q_x,j_x}, \mathcal{A}(R_{q_x}|L_{q_x,j_x}), s_{q_x} \rangle, \tag{8}$$

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 is found.

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 then verifies 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 $q_s = q_1$ because the cloud server should always return the 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_s,j_s} contains the updated value v_k^i and timestamp t_k^i. If so, the user considers the query result fresh.

Theorem 3. *KV-Fresh can detect any inauthentic and/or stale point query result.*

We provide the proof in Appendix C.

4.7 Range Query Processing

We now discuss how to extend the above solution into range query. A straightforward solution is to convert any range query into multiple point queries with each corresponding to one unique queried key, which would result in a proof size approximately linear to the size of query range. Our key observation is that the point query responses of adjacent queried keys have large overlap and can be merged to significantly reduce the communication cost. In what follows, we detail the procedure of query processing and query-result verification as the procedure update preprocessing remains the same.

Query Processing. Assume that the cloud server receives a range query $Q([l, r], t_q)$ asking for the most recent data record for every key $k \in [l, r]$ as of the end of interval q.

Also assume that v_k is the most recent update received at time t_k^i in interval i_k, where $i_k \leq q$ for all $k \in [l, r]$).

The cloud server first generates a point query result for every queried key $k \in [l, r]$. Let $\mathsf{R}^k = \{\mathsf{R}_1^k, \ldots, \mathsf{R}_{r_k}^k\}$ be the query result for each queried key $k \in [l, r]$, where r_k is the number of partial query results and

$$\mathsf{R}_x^k = \langle q_x^k, L_{q_x, j_x}^k, \mathcal{A}(R_{q_x^k}^k | L_{q_x, j_x}^k), s_{q_x^k} \rangle, \tag{9}$$

for all $1 \leq x \leq r_k$. It is easy to see that $q_1^k = q$ for all $l \leq k \leq r$ as the query result for every queried key must contain the information about interval q.

Given all the partial query results $\{\mathsf{R}_x^k | l \leq k \leq r, 1 \leq x \leq r_k\}$, the cloud server constructs the final query result by eliminates any duplicate leaf nodes. First, the cloud server sorts $\{\mathsf{R}_x^k | l \leq k \leq r\}$ first according to interval index q_x^k and then key k such that partial query results for the adjacent keys and the same interval appear next to each other. The cloud server then identifies and eliminates any duplicate partial query results for the same interval. Second, the cloud server merges all remaining the partial query results into one for every interval that appears in $\{\mathsf{R}_x^k | l \leq k \leq r\}$. Specifically, let $i^* = \min_{k \in [l,r]}\{i_k\}$ be the earliest interval with the most recent update for any queried key. For every interval $j \in [i^*, q]$ with at least one partial query result, the cloud server constructs an aggregated partial query result as follows. Let $\mathcal{K}_j^{[l,r]} \in [l, r]$ be the subset of keys that have partial query results for interval j. For each $k \in \mathcal{K}^{[l,r]}$, let its partial query result for interval j be

$$\mathsf{R}^k = \langle j, L_j^k, \mathcal{A}(R^k | L_j^k), s_j \rangle, \tag{10}$$

where we omit a part of the subscript to simplify the notation. We can see that $\{L_j^k\}_{k \in [l,r]}$ is a subset of LKS-MHT T_j's leaf nodes. The cloud server constructs an aggregate query result for interval j as

$$\mathsf{R}_j^{[l,r]} = \langle j, \{L_j^k | k \in \mathcal{K}_j^{[l,r]}\}, \mathcal{A}(R^k | \{L_j^k | k \in \mathcal{K}_j^{[l,r]}\}), s_j \rangle, \tag{11}$$

where $\mathcal{A}(R^k | \{L_j^k | k \in \mathcal{K}_j^{[l,r]}\})$ is the union of the subsets of internal nodes of LKS-MHT T_j needed to compute the root R^k from L_j^k for all $k \in \mathcal{K}^{[l,r]}$.

Query-Result Verification. The verification of a range query result is essentially the same as verifying multiple point query results. In particular, the only difference between the query processing in the two cases is that the cloud server eliminates the duplicated information among multiple point query results, so all the information needed for verifying the integrity and freshness of individual point query results are included in the range query result. We omit the details here due to space limitation.

Theorem 4. *KV-Fresh can detect any inauthentic and/or stale range query result.*

The proof is provided in the Appendix C.

5 Performance Evaluation

In this section, we evaluate the performance of KV-Fresh via extensive simulation studies using a real dataset.

TABLE 1
Default Simulation 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	1024	The maximal number of key blocks
$\|H(\cdot)\|$	256	The length of hash
$\|s_i\|$	1024	The length of data owner's signature

5.1 Dataset

We create a synthetic dataset from a TrueFax real-time currency conversion dataset [50] 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.

5.2 Simulation Settings

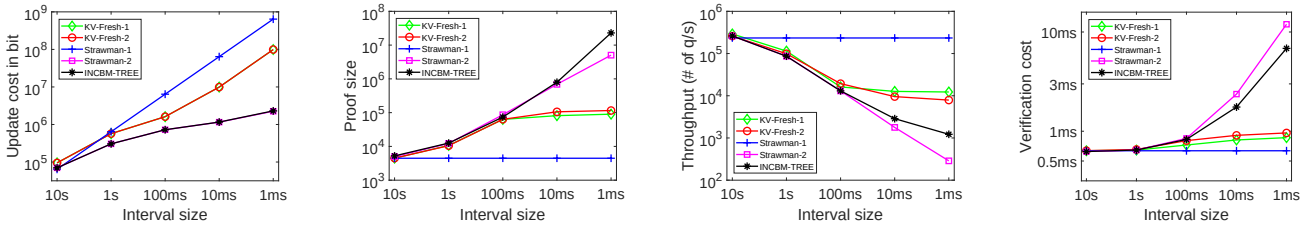
We implement a prototype of KV-Fresh in Python and deploy it on three desktops connected by 100 Mbps links, which act as the data owner, cloud server, and the user, respectively. Each desktop has a i7-6700 CPU, 16GB RAM and 64-bit Win10 operating system. We adopt the SHA-256 for the cryptographic hash function and the RSA for digital signature. Table 1 summarizes our default settings unless mentioned otherwise.

For point query, we compare KV-Fresh with the state-of-art solution INCBM-TREE [20] as well as the Strawman Approach 1 and Strawman Approach 2 approaches introduced in Section 4.1 using four performance metrics: (1) *update cost* which is number of extra bits per second transmitted from the data owner to cloud server,i.e., additional communication cost between the data owner and cloud server, (2) *proof size* which is the number of extra bits needed for proving the integrity and freshness for a query result, i.e., additional communication cost between the cloud server and user, (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. While the throughput and verification time may vary across different platforms, our goal is to provide a fair comparison among the four schemes under the same setting.

5.3 Simulation Results for Point Queries

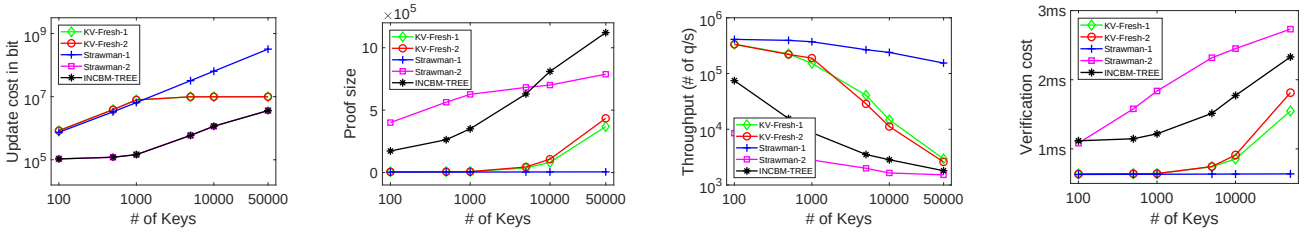
We now report our simulation results for point queries where every point in the following figures represents the average over 10,000 runs each with a distinct random seed. We refer to the two formulations discussed in Sections 4.5.1 and 4.5.2 as KV-Fresh-1 and KV-Fresh-2, respectively.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3172380, IEEE Transactions on Dependable and Secure Computing



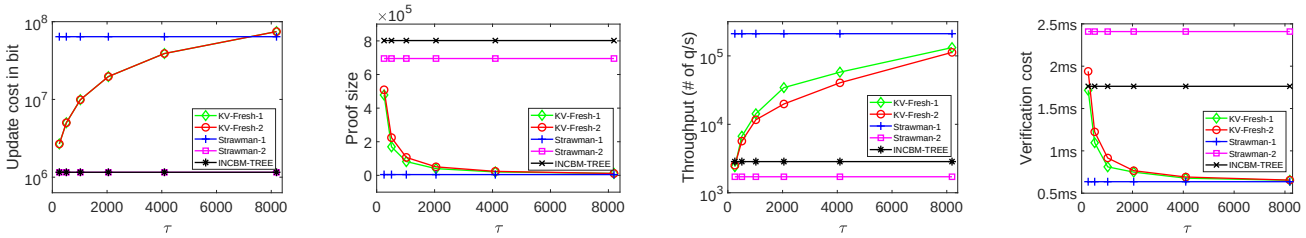
(a) Update cost vs. intervals size (b) Proof size vs. intervals size (c) Throughput vs. intervals size (d) Verification cost vs. intervals size

Fig. 5. Comparison of KV-Fresh, Strawman-1, Strawman-2, and INCBM-TREE with interval size varying from 10s to 1ms.



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

Fig. 6. 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. 7. Comparison of KV-Fresh, Strawman-1, Strawman-2, and INCBM-TREE with τ varying from 256 to 10,000.

5.3.1 *The Impact of Interval Size*

Fig. 5(a) compares the update cost under Strawman-1, Strawman-2, INCBM-TREE, KV-Fresh-1 and KV-Fresh-2 with interval size varying from 10 s to 1 ms, respectively. As we can see, the update cost per second increases as the interval sizes decreases under all mechanisms. This is expected, as the number of intervals is inversely proportional to the interval size. Among the five mechanisms, Strawman-1 has the highest update cost when the interval size is smaller than 1 s, 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 costs of KV-Fresh-1 and KV-Fresh-2 fall in the middle and increase much slower than that of Strawman-1. This is anticipated, as both KV-Fresh-1 and KV-Fresh-2 require the data owner to send only updated key-value records and key block information with no update for every interval. Moreover, when the interval size is 1 ms, both KV-Fresh-1 and KV-Fresh-2 incur 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. 5(b) shows the impact of interval size on the proof size of Strawman-1, Strawman-2, INCBM-TREE, KV-Fresh-1, and KV-Fresh-2. The proof size of Strawman-1 is not affected by the interval size and stays at 4460 bits. The proof sizes of the other four mechanisms all increase as the interval size decreases. Among the them, the proof sizes of Strawman-2 and INCBM-TREE grow the fastest and 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 the 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. In contrast, the proof sizes under KV-Fresh-1 and KV-Fresh-2 grow much slower as the interval size decreases, because both KV-Fresh-1 and KV-Fresh-2 allow the cloud server to skip potentially many intervals in the freshness proof. We can also see that the proof size of KV-Fresh-1 is slight lower than that of KV-Fresh-2, which is anticipated as KV-Fresh-1 aims to minimizing the expected size of freshness proof and the proof size in Fig. 5(b) is the average over 10,000 runs.

In addition, we can see that KV-Fresh outperforms INCBM-TREE by a large margin when the interval size is small. For example, when the interval size is 1 ms, the proof sizes under KV-Fresh-1 and KV-Fresh-2 are approximately 90 Kb and 115 Kb, respectively, which are less than 0.4% and 0.5% of the 22.9 Mb under INCBM-TREE, respectively.

Fig. 5(c) compares the throughput under Strawman-1, Strawman-2, INCBM-TREE, KV-Fresh-1, and KV-Fresh-2. We can see that the throughput under Strawman-1 is the highest and not affected by the change in interval size. Among the other four, 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 on average, 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-1 and KV-Fresh-2 initially decline as the interval size decreases from 10 s to 10 ms and then become stable or decrease 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. Moreover, KV-Fresh-1 outperforms KV-Fresh-2 with higher average throughput due to its merging decision policy, which aims to minimizing the expected proof size. Generally speaking, in comparison with Strawman-2 and INCBM-TREE, both KV-Fresh-1 and KV-Fresh-2 have similar throughput when the interval size is large while outperform Strawman-2 and INCBM-TREE by large margins when the interval size is small. For example, when the interval size is 1 ms, KV-Fresh-1 achieves 9.05 and 41.75 times higher throughput than INCBM-TREE and Strawman-2, respectively.

Fig. 5(d) compares the verification cost of the five 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 four mechanisms. Among them, KV-Fresh-1 and KV-Fresh-2 both outperform INCBM-TREE and Strawman-2 by large margins. The reason is that fewer leaf nodes need be returned under either KV-Fresh-1 or KV-Fresh-1 than both INCBM-TREE and Strawman-2. For example, when interval size is 1 ms, it takes 0.86 ms and 0.96 ms to verify a query result under KV-Fresh-1 and KV-Fresh-2, respectively, while Strawman-2 and INCBM-TREE require 11.96 ms and 6.84 ms, respectively. These results demonstrate the significant advantages of KV-Fresh over other two mechanisms.

5.3.2 The Impact of the Number of Keys

Figs. 6(a) to 6(d) compare the performance of KV-Fresh-1, KV-Fresh-2, Strawman-1, Strawman-2 and INCBM-TREE with $|\mathcal{K}|$, i.e., the total number of keys, varying from 100 to 50,000. As we can see from Fig. 6(a), the update costs of all schemes increase as the number of keys increase, which is anticipated. Moreover, the update cost of KV-Fresh-1 and KV-Fresh-2 are lower than that of Strawman-1 by a larger margin but higher than that of Strawman-2 and INCBM-TREE. More importantly, even when the $|\mathcal{K}|$ is 50,000, the update costs of KV-Fresh-1 and KV-Fresh-2 are both approximately 3.9×10^7 bits per second, which is very practical for 10-ms interval. From Fig. 6(b), we can see that the proof sizes under all mechanisms increase as $|\mathcal{K}|$ increases, as a larger $|\mathcal{K}|$ leads to a deeper MHT. Moreover, as $|\mathcal{K}|$ increases from 100 to 50,000, the proof sizes under KV-Fresh-1 and KV-Fresh-2 are always significantly smaller than those under Strawman-2 and INCBM-TREE. Similarly, Figs. 6(c) and 6(d) show that both KV-Fresh-1 and KV-Fresh-2 achieve much higher throughput and lower verification cost than Strawman-2 and INCBM-TREE because fewer leaf nodes need be returned under KV-Fresh-1 and KV-Fresh-2 than the other two.

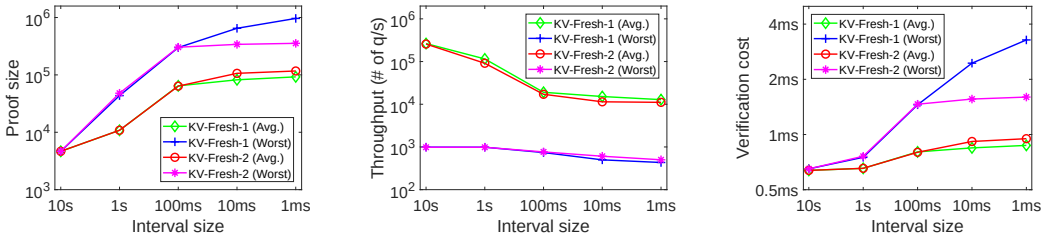
5.3.3 The Impact of τ

Figs. 7(a) to 7(d) show the performance of KV-Fresh-1 and KV-Fresh-2 with τ varying from 256 to 8192, where the performance of Strawman-1, Strawman-2 and INCBM-TREE are not affected by τ and only plotted for reference. Generally speaking, the larger τ, the higher the update cost, the smaller proof size, the higher throughput, the smaller verification cost for both KV-Fresh-1 and KV-Fresh-2, and vice versa. In addition, the update cost, proof size, throughput, and verification cost under KV-Fresh-1 and KV-Fresh-2 are almost always between those under Strawman-1 and those under Strawman-2 and INCBM-TREE, which is expected. While KV-Fresh-1 and KV-Fresh-2 incur higher update cost than Strawman-2 and INCBM-TREE, they incur much lower communication cost between the cloud server and the user and smaller verification cost at the user. Moreover, while update only happens between the data owner and the cloud server, the cloud server needs to serve potentially many users at the same time.

5.4 Comparison between KV-Fresh-1 and KV-Fresh-2

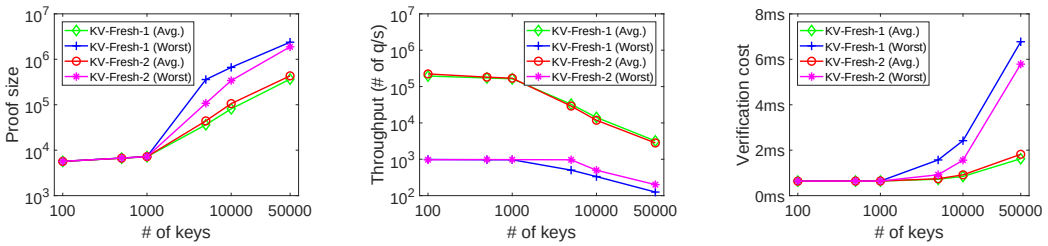
Fig. 8(a) and Fig. 8(c) compare the performance of KV-Fresh-1 and KV-Fresh-2 with interval size varying from 10 s to 1 ms, where KV-Fresh-1 (Avg.) and KV-Fresh-2 (Avg.) represent the average results of 10,000 runs and KV-Fresh-1 (Worst) and KV-Fresh-2 (Worst) represent the worst case among the 10,000 runs under KV-Fresh-1 and KV-Fresh-2, respectively. As we can see from Fig. 8(a), as the interval size decreases, both the average and the largest proof sizes increase under both KV-Fresh-1 and KV-Fresh-2, which is expected. More importantly, KV-Fresh-1 achieves smaller average proof size but larger proof size under the worst case. The reason is that KV-Fresh-1 and KV-Fresh-2 are designed to minimize the expected and maximum proof sizes, respectively. Fig. 8(b) shows that as the interval size increases, both the average and maximum proof sizes initially decrease followed by stable or decrease slightly due to the same reason in Fig. 5(c). We also observe that KV-Fresh-1 achieves higher average throughout but lower worst-case throughput than KV-Fresh-2. From Fig. 8(c), we can see that KV-Fresh-2 incurs a slightly higher average verification cost than KV-Fresh-1 for the same reason. More importantly, the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3172380, IEEE Transactions on Dependable and Secure Computing



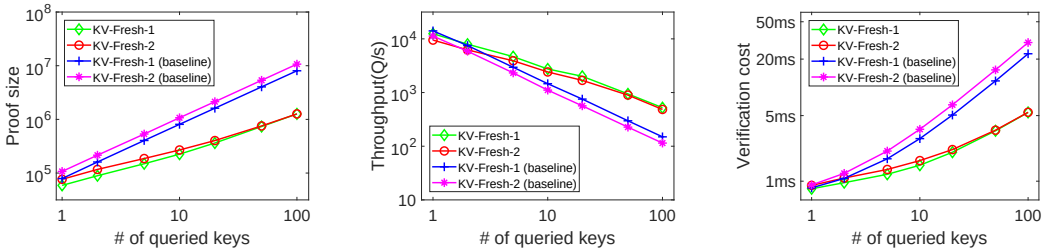
(a) Proof size vs. interval size (b) Throughput vs. interval size (c) Verification cost vs. interval size

Fig. 8. Comparison of KV-Fresh-1 and KV-Fresh-2 with interval size varying from 10s to 1ms.



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

Fig. 9. Comparison of KV-Fresh-1 and KV-Fresh-2 with $|\mathcal{K}|$ varying from 100 to 50,000.



(a) Proof size vs. size of query range (b) Throughput vs. size of query range (c) Verification cost vs. size of query range

Fig. 10. Comparison of KV-Fresh-1 and KV-Fresh-2 with the size of query range varying from 1 to 100.

worst-case verification cost under KV-Fresh-2 is significantly lower than that of KV-Fresh-1. Moreover, we can see that the gap between the average and worst-case verification costs grows as the interval size decreases. The reason is that when the interval size is large, many keys receive updates in each interval on average and the terminal condition for merging is mainly determined by τ, so there are very few merging opportunities to demonstrate the difference between KV-Fresh-1 and KV-Fresh-2. As the interval size decreases, the terminal condition is gradually determined by τ, and different merging decision have large impact on the average and worst-case verification costs, which leads to the increased gap between the two mechanisms.

Fig. 9(a) and Fig. 9(c) compare the average and worst-case performance of KV-Fresh-1 and KV-Fresh-2 with $|\mathcal{K}|$ varying from 100 to 50,000. Generally speaking, the larger $|\mathcal{K}|$, the larger proof size, the lower throughout and the higher verification cost for both the average and worst-case under the two mechanisms. Moreover, KV-Fresh-1 outperforms KV-Fresh-2 in terms of average proof size, throughout and verification cost, while KV-Fresh-2 has better worst-case performance. In addition, we can see from Fig. 9(c) that the gap between the average and the worst-case performance increases as $|\mathcal{K}|$ increase from 100 to 50,000. For example, the difference between KV-Fresh-1 (Avg.) and KV-Fresh-1 (Worst) grows from 0.84 ms to 5.1 ms when the $|\mathcal{K}|$ increases from 5,000 to 50,000.

5.5 Simulation Results for Range Queries

Since INCBM-TREE [20] is not directly applicable to range queries, we compare KV-Fresh with a baseline solution, referred to as *KV-Fresh-baseline*, which processes a range query as multiple independent point as in KV-Fresh. For the performance evaluation, we still use the metrics proof size, throughput, and verification time but omit the metric update cost as they share the same update preprocessing procedure.

Fig. 10(a) shows the impact of the size of query range on the proof size under KV-Fresh-1, KV-Fresh-2, KV-Fresh-1-baseline, and KV-Fresh-2-baseline. We can see that the

proof size increases as the number of queried keys increases under all four mechanisms, which is expected. Moreover, both KV-Fresh-1 and KV-Fresh-2 incur a much smaller size of proof than corresponding KV-Fresh-1-baseline and KV-Fresh-2-baseline. For example, when the number of queried keys is 10, KV-Fresh-1 and KV-Fresh-2 reduce the proof size of KV-Fresh-1-baseline and KV-Fresh-2-baseline by 72% and 74.9%, respectively. As another example, when the number of queried keys is 100, KV-Fresh-1 and KV-Fresh-2 reduce the proof size of KV-Fresh-1-baseline and KV-Fresh-2-baseline by 84% and 88% times, respectively. The reason is that the two baseline solutions treat a range query as multiple independent point queries for which the query results have large overlap. In contrast, both KV-Fresh-1 and KV-Fresh-2 eliminate such redundancy in the query result, resulting in significant reduction in the freshness proof size and thus higher communication and computation efficiency.

Fig. 10(b) compares the throughput of KV-Fresh-1, KV-Fresh-2, KV-Fresh-1-baseline, and KV-Fresh-2-baseline with the number of queried keys varying from 1 to 100. We can see that the throughput under all four mechanisms decreases as the number of queried keys increase, which is expected as it takes longer time to process a range query with a larger query range size. Moreover, both KV-Fresh-1 and KV-Fresh-2 outperform corresponding KV-Fresh-1-baseline and KV-Fresh-2-baseline, especially when the size of query range is large, as they both treat a range query as a whole instead of multiple independent point queries. For example, when the size of query range is 100, KV-Fresh-1 can process 522 range queries in one second, while KV-Fresh-1-baseline can only process 149 range queries.

Fig. 10(c) shows the verification cost of KV-Fresh-1, KV-Fresh-2, KV-Fresh-1-baseline, and KV-Fresh-2-baseline with different sizes of query range. We can see that the verification cost of the all mechanisms sharply increase as the number of queried keys increases. Similar to Fig. 10(a) and Fig. 10(b), both KV-Fresh-1 and KV-Fresh-2 outperform corresponding KV-Fresh-1-baseline and KV-Fresh-2-baseline in terms of verification cost, which is expected. These results further confirm the high efficiency of KV-Fresh in processing range queries.

6 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. Specifically, 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. We also propose two merging decision to fulfill the LKS-MHT construction. KV-Fresh supports both point query and range query. Extensive simulation studies confirm that KV-Fresh can always simultaneously achieve strong real-time guarantee and high communication efficiency.

References

[1] P. Felber, M. Pasin, . Rivire, V. Schiavoni, P. Sutra, F. Coelho, R. Oliveira, M. Matos, and R. Vilaa, "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] "Baidu workers arrested for 'deleting posts for money'," https://www.bbc.com/news/technology-19149185.

[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] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] M. Narasimha and G. Tsudik, "Dsac: Integrity for outsourced databases with signature aggregation and chaining," ser. ACM CIKM'05, Otc. 2005, p. 235236.

[27] E. Mykletun, M. Narasimha, and G. Tsudik, "Authentication and integrity in outsourced databases," *ACM Transactions on Storage*, vol. 2, no. 2, pp. 107–138, May 2006.

[28] A. A. Yavuz, "Immutable authentication and integrity schemes for outsourced databases," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 1, pp. 69–82, 2018.

[29] B. Zhang, B. Dong, and W. H. Wang, "Integrity authentication for sql query evaluation on outsourced databases: A survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 33, no. 4, pp. 1601–1618, 2021.

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

[31] 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.

[32] D. Wu, B. Choi, J. Xu, and C. S. Jensen, "Authentication of moving top-k spatial keyword queries," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 4, pp. 922–935, 2015.

[33] 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.

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

[35] 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.

[36] Y. Jing, L. Hu, W.-S. Ku, and C. Shahabi, "Authentication of k nearest neighbor query on road networks," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 6, pp. 1494–1506, 2014.

[37] 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.

[38] S. Benabbas, R. Gennaro, and Y. Vahlis, "Verifiable delegation of computation over large datasets," ser. Advances in Cryptology – CRYPTO 2011, 2011, pp. 111–131.

[39] D. Catalano and D. Fiore, "Vector commitments and their applications," ser. Public-Key Cryptography – PKC 2013, 2013, pp. 55–72.

[40] X. Chen, H. Li, J. Li, Q. Wang, X. Huang, W. Susilo, and Y. Xiang, "Publicly verifiable databases with all efficient updating operations," *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–1, 2020.

[41] M. Miao, J. Ma, X. Huang, and Q. Wang, "Efficient verifiable databases with insertion/deletion operations from delegating polynomial functions," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 2, pp. 511–520, 2018.

[42] X. Chen, J. Li, X. Huang, J. Ma, and W. Lou, "New publicly verifiable databases with efficient updates," *IEEE Transactions on Dependable and Secure Computing*, vol. 12, no. 5, pp. 546–556, 2015.

[43] X. Chen, J. Li, J. Weng, J. Ma, and W. Lou, "Verifiable computation over large database with incremental updates," *IEEE Transactions on Computers*, vol. 65, no. 10, pp. 3184–3195, 2016.

[44] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song, "Provable data possession at untrusted stores," ser. CCS '07, 2007, p. 598609.

[45] C. C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia, "Dynamic provable data possession," *ACM Trans. Inf. Syst. Secur.*, vol. 17, no. 4, Apr 2015.

[46] Y. Zhu, H. Hu, G.-J. Ahn, and M. Yu, "Cooperative provable data possession for integrity verification in multicloud storage," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 12, pp. 2231–2244, 2012.

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

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

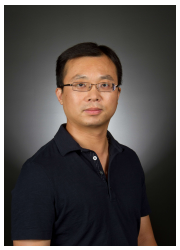
[49] L. Harn, "Batch verifying multiple rsa digital signatures," *Electronics Letters*, vol. 34, no. 12, pp. 1219–1220, June 1998.

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

Yidan Hu received the B.E. and M.E. both in Computer Science from Hangzhou Dianzi University in 2013 and 2016, respectively, and the PhD degree in Computer Science from University of Delaware in 2021. She is currently an Assistant Professor in the Department of Computing Security at the Rochester Institute of Technology. Her primary research interests are security and privacy in networked and distributed systems, wireless networking, and mobile computing.

Xin Yao received the B.S. in Computer Science from Xidian University in 2011, the M.S. in Software Engineering and the Ph.D. in Computer Science and Technology from Hunan University in 2013 and 2018, respectively. From 2015 to 2017, he worked as a visiting scholar at Arizona State University. He is currently an assistant professor at Central South University. His research interests include security and privacy issues in social network, Internet of things, cloud computing and big data.

Rui Zhang received the B.E. in Communication Engineering and the M.E. in Communication and Information System from Huazhong University of Science and Technology, China, in 2001 and 2005, respectively, and the PhD degree in electrical engineering from the Arizona State University, in 2013. He has been an Assistant Professor in the Department of Computer and Information Sciences Department at University of Delaware since 2016. Prior to joining UDel, he had been an Assistant Professor in the Department of Electrical Engineering at the University of Hawaii from 2013 to 2016. His primary research interests are security and privacy issues in wireless networks, mobile crowdsourcing, mobile systems for disabled people, cloud computing, and social networks. He received the US NSF CAREER Award in 2017.



Yanchao Zhang received the B.E. in Computer Science and Technology from Nanjing University of Posts and Telecommunications in 1999, the M.E. in Computer Science and Technology from Beijing University of Posts and Telecommunications in 2002, and the Ph.D. in Electrical and Computer Engineering from the University of Florida in 2006. He is a Professor in School of Electrical, Computer and Energy Engineering at Arizona State University. His primary research interests are security and privacy issues in computer and networked systems, with current focus areas in emerging wireless networks, mobile crowdsourcing, Internet-of-Things, social networking and computing, wireless/mobile systems for disabled people, big data analytics, mobile/wearable devices, and wireless/mobile health. He has been on the editorial boards of IEEE Transactions on Mobile Computing, IEEE Wireless Communications, IEEE Transactions on Control of Network Systems, and IEEE Transactions on Vehicular Technology. He received the US NSF CAREER Award in 2009 and is a Fellow of IEEE.