Authentication of Freshness for Outsourced Multi-Version Key-Value Stores

Regular paper

Yuzhe Tang[†] Ting Wang[‡] Xin Hu[‡] Jiyong Jang[‡] Ling Liu[†] Peter Pietzuch[♯]

[†]*Georgia Institute of Technology, Atlanta GA, USA, Email:* {yztang@, lingliu@cc.}gatech.edu

[‡]*IBM Research Center, Yorktown Heights NY, USA, Email:* {tingwang, xinhu, jjang}@us.ibm.com

[♯]*Imperial College London, UK, Email:* prp@doc.ic.ac.uk

***Abstract*—Data outsourcing offers cost-effective computing power to manage massive data streams and reliable access to data. For example, data owners can forward their data to clouds, and the clouds provide data mirroring, backup, and online access services to end users. However, outsourcing data to untrusted clouds requires data authentication and query integrity to remain in the control of the data owners and users.**

In this paper, we address this problem specifically for multi-version key-value data that is subject to continuous updates under the constraints of data integrity, data authenticity, and "freshness" (i.e., ensuring that the value returned for a key is the latest version). We detail this problem and propose INCBM-TREE, a novel construct delivering freshness and authenticity.

Compared to existing work, we provide a solution that offers (i) lightweight signing and verification on massive data update streams for data owners and users (e.g., allowing for small memory footprint and CPU usage on mobile user devices), (ii) integrity of both real-time and historic data, and (iii) support for both real-time and periodic data publication. Extensive benchmark evaluations demonstrate that INCBM-TREE achieves more throughput (in an order of magnitude) for data stream authentication than existing work. For data owners and end users that have limited computing power, INCBM-TREE can be a practical solution to authenticate the freshness of outsourced data while reaping the benefits of broadly available cloud services.

I. INTRODUCTION

In the big data era, data sources generate data of large variety, volume, and at a high arrival rate. Such intensive data streams are widely observed in system logs, network monitoring logs, social application logs, and many others. In order to efficiently digest the large data streams, which can be beyond a regular data owner's computing capability, outsourcing data and computation to clouds becomes a promising approach. Clouds can provide sufficient storage and computing capabilities with the help of large data centers and scalable networked software. By delegating processing, storing, and query serving of data streams to a third-party service, the data outsourcing paradigm not only relieves a data owner of the cumbersome management work but also saves significant operational cost.

For example, stock exchange service providers, social networking companies, and network monitoring companies can benefit from outsourcing their streaming data to clouds. In a stock exchange market, stock buyers and sellers make deals based on the changing price. To identify stock market trends, stock buyers may frequently consult exchange providers about historical and real-time stock prices. With a large number of stocks, the footprint of the stock price data would easily grow out of a regular company's computing capability or its IT budget. Moreover, as there are more and more stock brokers in the market, it requires huge computing power to serve such a large customer base. Another example is a social networking website where the stream of social application events arrive at a high rate. At an online auction website, bid proposals are continuously generated, which can easily exceed the limit of the server capability of the operating company. Big data streams can be also observed in a network monitoring scenario where a company monitors its real-time network traffic.

Despite its advantages, data outsourcing causes issues of trust, because the cloud, being operated by a third-party entity, is not fully trustworthy. A cloud company could deliver incomplete query results to save computation cost or even maliciously manipulate data for financial incentives, e.g., to gain an unfair advantage by colluding with a data user competing with the rest. Therefore, it is imperative for a data owner to protect data authenticity and freshness when outsourcing its data to a third-party cloud.

It is crucial to assure *temporal freshness* of data, i.e., obtain proofs that the server does not omit the latest data nor return out-of-date data. Especially when the value of the data is subject to continuous updates, it is not sufficient to guarantee only the *correctness* of data because a data client[1] expects to obtain the "freshest" data. For example, a data user can query the latest price of a stock, the latest bid towards the purchase of a product in an auction, or the sensor readings of any monitored attributes at a specific time.

While a large corpus of work including [1], [2], [3], [4], [5], [6], [7] focused on authentication of dynamic data, efficient freshness authentication still remains as a challenging and understudied problem. Freshness authentication essentially requires signing the relationships between a data version and any time point when the version is valid. There are two main approaches to sign the relationship. First, a key-based signing approach is used for authenticating

[1]We interchangeably use a data client and a data user in the paper.

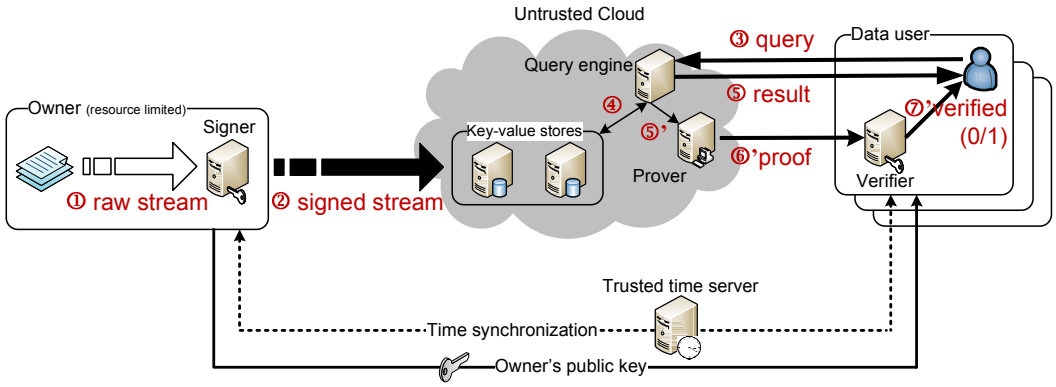


Figure 1: System of outsourced key-value stores

dynamic data [3], [8] and aggregated information [9], [10]. The approach periodically signs the latest value with the current time so that freshness of the value is guaranteed at the granularity of the signing time interval. Even though this approach offers a data client with efficient verification, it is impractical for intensive data streams because it requires a data owner to keep a local copy of the entire dataset. Second, a time-based signing approach is used for authenticating data streams [11]. The approach signs a sequence of incoming data, and provides a data owner with efficient signing since it does not require a local copy of the entire dataset. This approach, however, could impose significant verification overhead on a data client since all previous data have to be retrieved for verification. Due to the limitation, freshness authentication is provided only for recent data within a sliding window [11].

In this paper, we propose a novel authentication framework for multi-version key-value data streams. We note that freshness verification overhead in existing work is high because it lacks the capability to perform an efficient *non-membership* test, e.g., there were no updates on the value of key k within last 5 minutes. We formalize the problem in §II and introduce a novel construct INCBM-TREE in §III to address the problem. INCBM-TREE uses a Bloom filter [12] for freshness authentication while enabling lightweight signing and optimized verification. Conceptually, INCBM-TREE is a Merkle Hash tree (MHT) [13] that embeds a hierarchy of Bloom filters (BFs). In INCBM-TREE, a MHT signs and protects authenticity of data streams along with their associated BFs, whereas BFs provide efficient verification for version freshness. Furthermore, we design INCBM-TREE in such a way that it can be incrementally constructed and maintained so that signing data stream can be done efficiently without accessing historical data. We summarize the comparison of INCBM-TREE with existing work in Table I, where +/– denotes that an approach support/does not support a feature.

In summary, our main contributions are as follows.

Table I: Comparing INCBM-TREE with prior work

Data Model	Approaches	Version freshness	Lightweight signing	Efficient verification
Stream	[11]	+	+	–
Agg	[9], [10]	–	+	+
Dynamic Data	[8], [3]	+	–	–
	INCBM-TREE	+	+	+

- To the best of our knowledge, we are the first to efficiently solve the problem of *outsourcing multi-version key-value stores with verifiable freshness*, enabling devices with limited computation capabilities to leverage cloud-based data management while relying on the freshness and authenticity of the outsourced data.
- We design a generic Put/Get interface for outsourced data retrieval, which differentiates itself from sliding window queries [11] and aggregation queries [10], [9] on outsourced data.
- We define the authenticity property in the new problem setting.
- We propose a novel construct INCBM-TREE to authenticate version freshness, which dramatically reduces freshness verification and does not incur disk I/O overhead.
- We evaluate or implementation of INCBM-TREE and our results confirm that it applies to generic key-value stores, offering more throughput (in an order of magnitude) for data stream authentication than existing work.

II. PROBLEM FORMULATION

A. System Model

Figure 1 illustrates our system model. In this ecosystem, there are three parties in different administrative domains:

- a data owner, e.g., a small technology start-up company
- a data user, e.g., a customer of the company
- a public cloud, offering data storage and management

Note that both the data owner and the data user do not fully trust a public cloud as it is operated by a third-party company with its own interests, partially conflicting with those of the small start-up company.

In this scenario, the data owner outsources a key-value dataset to a public cloud, and each stream unit is an update to a key-value record in the dataset, submitted by the data owner to the public cloud. In order to provide data authenticity, a data owner employs a *signer* to sign the raw data stream before publishing it to a cloud (steps ① and ②). The data user typically uses the Put/Get interface to access the key-value dataset.

The data user issues Get queries to the outsourced key-value store in the public cloud. The usual query path returns a result of interests (steps ③, ④, and ⑤), and an additional verification path (steps ⑤', ⑥', and ⑦'). A prover in a cloud composes a proof and delivers it to the data user's *verifier*, which verifies the authenticity of the query results for the data user. We assume that the data user knows the public key of the data owner, and that the signer and the verifier are time-synchronized (e.g., using a trusted network time services) for freshness authentication. Details of the authentication framework will be discussed in §II-E.

B. Data Model

Our data model is a multi-version key-value data stream. We require that the signature for the data is publicly verifiable. Without losing generality, we consider a basic key-value data model where each object has a unique key k, a value v, and an associated timestamp ts. v can be updated multiple times. This basic data model can be easily extended to support more advanced key-value models, such as the column-family model, by multiplying the data model for each column.

1) Query Model: We focus on the selection query with time awareness. Given key k and timestamp ts, the query returns the latest version of the object associated with key k by ts. The API, modeled after the generic Put/Get interface in key-value stores [14], [15], [16], [17], is defined as follows.

$$\mathsf{VGet}(k, ts) \rightarrow \langle v, ts_{\text{latest}} \rangle \cup \pi(k, v, ts_{\text{latest}}, ts)$$

Here, π is a proof presented by a cloud to a data user. If unspecified, the default value of ts is the current timestamp ts_{now}. This API is primitive to many selection SQL queries and can add time-awareness. Our model considers both continuous and one-time queries.

C. Threat Model

We assume a threat model where neither the data user nor the public cloud service is trusted by the data owner. We assume that the adversary (e.g., having compromised the cloud service) may deliberately conceal certain versions of data by excluding them from responses to data users. The adversary may also manipulate data values before presenting them to the querying user. Further, we assume that an adversarial cloud service may collude with malicious data users, trying to obtain the data owner's secret key so that it can forge a proof for compromised data. In our solution, we use public-key signature so that only public key is released to the data user and cloud.

1) Security Goals: Based on our query and threat models, there are two desirable properties to be guaranteed.

Definition 2.1: Given a query key k in our data model, the cloud may return multiple versions updated at different points in time $\{\langle v, ts \rangle\}$. For a timestamp ts_{query}, a version $\langle v, ts \rangle$ is

- *fresh* if and only if $\langle v, ts \rangle$ is the latest version updated before ts_{query} and
- *correct* if and only if $\langle v, ts \rangle$ is indeed a version that belongs to the key k and was submitted by the data owner.

D. Cryptographic Essentials

We introduce a set of cryptographic tools used throughout the remaining of the paper.

1) Hash Function: A hash function $H(\cdot)$ takes a variable-length input $\mathtt{x}$ and generates a fixed-length output $\mathtt{y} = H(\mathtt{x})$. A hash can be efficiently computed. Hash functions used in this work are assumed to be collision resistant, which means that it is computationally infeasible to find two different inputs $\mathtt{x} \neq \mathtt{x}'$ such that $H(\mathtt{x}) = H(\mathtt{x}')$, e.g., SHA1.

2) Digital Signature: A public-key digital signature scheme provides data integrity and ownership of the signed data. After creating a pair of keys (SK, PK), the signer keeps the secret key SK, and publishes the public key PK. A signature $sig(\mathtt{x}, SK)$ is produced for a message $\mathtt{x}$, and a recipient of the message can verify its integrity and the ownership of $\mathtt{x}$ by using $sig(\mathtt{x}, SK)$ and PK. Note that signing is typically much more expensive than hashing.

3) Merkle Tree: A Merkle hash tree (MHT) [13] is a method of collectively authenticating a set of objects. Specifically, being a binary balanced tree structure, each leaf node of MHT corresponds to the hash of an object, and each non-leaf node corresponds to the hash (digest) of the concatenation of its direct children's hashes. The root node corresponds to the digital signature of the root digest. To prove the authenticity of any object, the prover provides the verifier with the object itself and the digests of the siblings of the nodes that lie in the path from the root. By iteratively computing and concatenating the appropriate hashes, the verifier can then recompute the root digest and verify its correctness using its digital signature.

4) KOMT*:* A key-ordered Merkle tree (KOMT) is an approach of using a Merkle tree to sign data batches, e.g. for authenticating data streams. Given a batch of data records with key attributes, a KOMT sorts the data based on the key and hashes each leaf node with its two neighboring nodes,

i.e., the predecessor and the successor in the sorted order. This allows to easily verify whether a range of keys or a specific key is in the current dataset or not.

E. Authentication Framework

Based on our data model, we now describe the authentication framework. As illustrated in Figure 1, the stream of data updates is generated by the data owner and is signed by a signer in the owner domain (step ①). The signer signs the streaming updates in a batch. Given a batch of updates b in the stream, the signer builds a digest structure dd (e.g. a Merkle tree), and signs the digest $\mathtt{sig}(dd, SK)$. In particular, the digest of the batch dd includes the root of the constructed digest structure d, the current timestamp ts_{end}, and the timestamp of the last batch ts_{start}. That is, $dd = ts_{\text{start}} \| ts_{\text{end}} \| d$. The signature of the batch is published along with the raw data stream to a cloud (step ②). Upon receiving the signed updates stream, the cloud materializes them in key-value stores. To accommodate the intensive data stream, a write-optimized key-value store (e.g., BigTable [14], HBase [15], Cassandra [16] and LevelDB [17]) can be used. These key-value stores optimize the write operations by their append-only designs which typically results in the co-existence of multiple versions given a single data record in the store. In the signed data stream, the raw data updates are applied to a base table in the key-value store, and the signatures are stored in a meta-data table. The digest structures, which are not transmitted to a cloud for the sake of bandwidth efficiency, are reconstructed on the cloud side from the raw data updates, and then stored in another meta-data table. The following interface shows how to persist key-value data and meta-data in the store.

$$\begin{aligned} \text{BaseTable.}\mathsf{Put}(k, v, ts) & \qquad (1) \\ \text{BaseTable.}\mathsf{Get}(k, ts) \quad & \rightarrow \quad \langle v, ts_{\text{latest}} \rangle \\ \text{SigTable.}\mathsf{Put}(sig, ts_{\text{start}}, ts_{\text{end}}) & \\ \text{SigTable.}\mathsf{Get}(ts) \quad & \rightarrow \quad sig \\ \text{DigTable.}\mathsf{Put}(ts_{\text{start}}, ts_{\text{end}}, d) & \\ \text{DigTable.}\mathsf{Get}(ts) \quad & \rightarrow \quad d \end{aligned}$$

While the data owner sends the data streams to the cloud, a data user can issue a $\mathsf{VGet}(k,ts)$ query to the cloud (step ③). A query engine server in a cloud processes the query by interacting with the key-value store (step ④) and returns the result to the user (step ⑤). At the same time, the query engine prompts the prover to prepare the proof for the query result (step ⑤'). The proof includes signatures and a specific digest structure that links signatures to the result data. After the proof is sent to the verifier in the data user domain (step ⑥'), the verifier validates the result and provides the authenticity test result to the user (step ⑦').

Given a query and its result $\mathsf{VGet}(k,ts) \rightarrow \langle v, ts_{\text{latest}} \rangle$, the proof $\pi(k, v, ts_{\text{latest}}, ts)$ needs to satisfy two properties: the *correctness* ensuring that $\langle v, ts_{\text{latest}} \rangle$ is a valid version published by the data owner, and the *freshness* ensuring that the returned version $\langle v, ts_{\text{latest}} \rangle$ is indeed the latest version of key k as of time ts. Proving the freshness of v requires verifying that there is no version update of key k between ts_{latest} and ts. A naive proof includes all the data updates or versions within the time interval with their signatures to the verifier. This is too expensive for massive data streams, especially if for keys that are updated infrequently. In this work, we make the key observation that a version freshness verification is essentially equivalent to a "non-membership" test between a time interval, i.e., there doesn't exist a version $\langle v', ts' \rangle$ such that $ts_{\text{latest}} < ts' \leq ts$.

We assume that the signer and the verifier are time synchronized so that the freshness can be verified[2]. This assumption can be satisfied by using a trusted time server (e.g., NIST Internet Time Server[3]). In addition, we assume that data owners are independent of each other, that is, different stream owners have their own signature keys and independently publish the streaming data to a cloud. Different owners' streaming data is stored and served independently by the cloud service. This is a common scenario in outsourced cloud operations for privacy reasons.

Our framework for authenticating key-value stores is more flexible and lightweight than existing stream/data authentication frameworks in an outsourced database. Compared to stream authentication [11] where a sliding window query is supported only on recent data, our framework allows access to both recent and historic data. Compared to dynamic data authentication [8], [3], our framework does not require local materialization of the whole dataset (or its latest snapshot) on the data owner side, thereby being more lightweight and practical in the streaming scenario.

III. IncBM-Tree

A. The Need for Large Signing Batch

In our authentication framework, a big batch size is critical to efficient verification. For a fixed amount of streaming data, the larger batch is signed at a time, the fewer times of signature verification is required. We have conducted a performance study of KoMT. Figure 2 presents a preview for verification performance under different key distributions (e.g. the uniform and Zipfian distributions). The use of a larger batch results in orders of magnitude faster verification.

1) A Baseline Approach: Limited memory space of the owner is the primary factor that prevents data batch from increasing infinitely. A straightforward approach for a data

[2]Consider a simple example for necessity of timing server and time synchronization, where a data user's time is on $12/04/2013$ while owner's time is on $12/03/2013$. Then the owner can sign the latest version up to $12/03/2013$ while the user may think that it is yesterday's version, but not today.

[3]http://tf.nist.gov/tf-cgi/servers.cgi

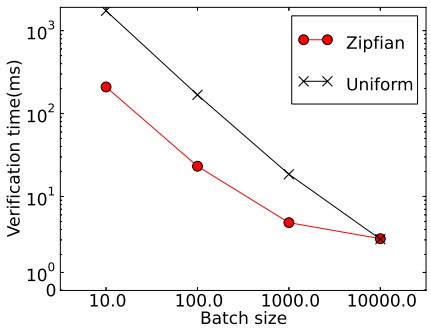


Figure 2: Verification performance of KOMT

batch setting that is larger than the owner memory is to spill the data on to disk and load it back upon the signing time. At the signing time, the data in multiple spill files will be merged and sorted based on key. This approach, termed on-disk KOMT, serves as a baseline in this paper. It can have a data batch as large as the owner disk can accommodate. However, the main drawback is that it involves expensive disk IO when the owner ingests the intensive data stream; it could significantly slow down the sustainable throughput to ingest the stream. This motivates us to seek a lightweight authentication structure that can fit into memory yet still be able to sign a large batch of data.

B. INCBM-TREE: *Design and Structure*

Our basic idea is to compress the data in memory as much as possible before the digest and signature are constructed. Based on the observation that a Bloom filter is a summary for a (non)-membership test, we propose to combine a Bloom filter with a Merkle tree for efficient verification of both correctness and freshness. The structure of INCBM-TREE is illustrated in Figure 3. Comparing to the traditional Merkle tree, each tree node in INCBM-TREE maintains not only a hash digest but also a digest value that summarizes

Table II: Notations

System model symbols
k: key
v: value
ts: timestamp
r: stream arrival rate
e: user tolerable delay
s_r: record size
S: memory size
INCBM-TREE symbols
b: batch size
b_r: batch size in real-time publication
b_b: batch size in periodic publication
C_{sig}: cost of single digital signature
C_{dig}: per-record cost of building digest
d: the digest of a stream data batch
q: ratio of INCBM-TREE to KOMT

the key set in the subtree rooted at the node. For a key set of the subtree, the digest includes a Bloom filter and an interval of both upper and lower bounds. For instance, a leaf node 6 maintains a Bloom filter BF_6 summarizing key 1 on node 12 and a hash digest h_6. Given node 3 which is the parent of two leaf nodes (node 6 and node 7), its digest is a Bloom filter of union of its children nodes' Bloom filters, namely $BF_3 = BF_6 \cup BF_7$. The range digest is simply with lower bound 1 and upper bound 23, namely $R_3 = [1, 23]$. It comes from merging the ranges from its two children, that is, $[1, 23] = [1, 12] \cup [15, 23]$. The hash digest is the hash value of concatenation of all children's hashes, the range digest, and the Bloom filters, that is, $h_3 = H(h_6 \| h_7 \| BF_3 \| R_3)$. INCBM-TREE uses the following constructs.

$$\begin{aligned} R(\text{node}) &= R(\text{left\_child}) \cup R(\text{right\_child}) \\ BF(\text{node}) &= BF(\text{left\_child}) \cup BF(\text{right\_child}) \\ h(\text{node}) &= H(h(\text{left\_child}) \| h(\text{right\_child}) \| BF(\text{node}) \| R(\text{node})) \end{aligned} \quad (2)$$

In a INCBM-TREE, the Bloom filter at every level is of the same length. The error rate E of a bloom filter can be estimated using equation $E = (1 - e^{-\frac{Kx}{m}})^K$, where K is the number of hashes used in the Bloom filter and m is the length of the array used to store bits in Bloom filter, x is the number of values of the data set summarized in Bloom filter. Given a pre-defined batch size b and a tolerable error bound E_b, we can set the length of the Bloom filter l as follows.

$$E = (1 - e^{-\frac{Kb}{m}})^K \quad (3)$$

$$\Rightarrow m = \frac{Kb}{-\ln(1 - {E_b}^{1/K})} \quad (4)$$

For the root node, the error rate is $E = E_b$. For an internal node, the number of leaf nodes in the subtree is smaller than b. Therefore, its actual error rate is bounded by $E < E_b$.

1) Security Property:

Theorem 3.1: The INCBM-TREE root node can authenticate any bloom filter in the tree structure.

Proof: The proof of security is based on the infeasibility of finding two different Bloom filters BF_1 and BF_2 such that $H(\ldots BF_1 \| \ldots) = H(\ldots BF_2 \| \ldots)$. If this is feasible, then there exist two values, $v_1 = \ldots BF_1 \| \ldots$ and $v_2 = \ldots BF_2 \| \ldots$, such that $H(v_1) = H(v_2)$, which clearly contradicts the fact that H is a collision resistant secure hash. ■

C. Proof Construction and Verification

The INCBM-TREE is used to construct proof to verify a result of query VGet. We start the description of proof construction by an ideal case where a bloom filter is without error. Following the example in Figure 3, to provide a proof for freshness on key 98, it suffices to return only two nodes, that is, node 5 and node 3. Because bloom filter BF_3 can

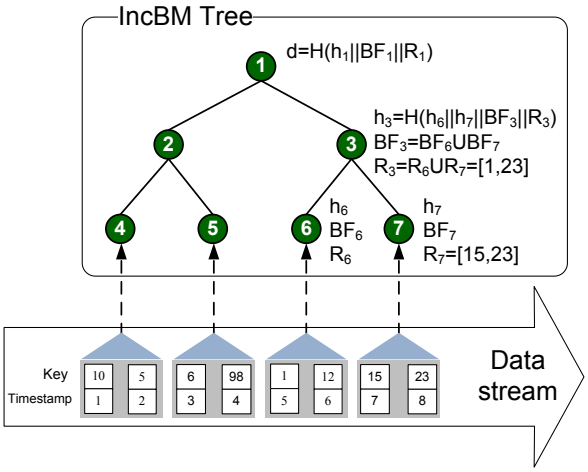


Figure 3: INCBM-TREE structure

test the non-membership of key 98 in the subtree at node 3, and digest h_3 can be used to verify authenticity of BF_3.

In reality, a Bloom filter can have a false positive. This implies that when a key k is not in a set, a Bloom filter may claim the false membership that k is in the set. In this case, our strategy is to go down INCBM-TREE by one level. For instance, when BF_3 can not verify the non-membership of key k, we use the two children's Bloom filters, BF_6 and BF_7, which collectively verify the non-membership. By using multiple lower-level Bloom filters, the chance of correctly verifying the non-membership becomes higher. In an extreme case when all the internal nodes' bloom filters fail to confirm the non-membership and one reaches a leaf node, the non-membership or membership can be verified correctly by going through each element inside the leaf node. In practice, the INCBM-TREE is built on top of the key-ordered Merkle tree which can verify the non-membership by returning an authentication path that covers the queried key[11], [3]. Given a key-ordered Merkle tree of height c, this authentication path involves with at most c hash values. By this way, it can always guarantee error-free verification (due to the collision resistance of a hash function).

1) Cost Analysis: Although descending the INCBM-TREE guarantee the error-free of non-membership test, it inevitably incurs extra verification cost. However, as we will show below, the probability of descending the INCBM-TREE decreases exponentially and given a small error rate bound, the extra cost is expected to be a constant

Because descending one level down only happens when a parent node is erroneous for answering a non-membership test, it occurs with a probability equal to the error rate of the node's bloom filter. By enforcing all Bloom filters in INCBM-TREE is of constant error rate E, we can have the fact that descending N levels down occurs with probability E^N.

Theorem 3.2: In an INCBM-TREE with a Bloom filter, an error rate bounded by E ($E \ll 0.5$) at different tree levels, the extra cost X is expected to be a constant value, as follows.

$$X = \frac{1-E}{1-2E} \tag{5}$$

Here, cost X is defined to be the number of tree nodes for error-free verification.

Proof: Suppose the expected cost of a tree node at level l ($l = 0$ for leaf nodes) is X_l. There are two cases for query processing: 1) The tree node's Bloom filter can correctly answer the non-membership query, and 2) The tree node can not. For the first case, it occurs with probability $1 - E$, since a Bloom filter's error rate is E. When the query cost is 1 for a single node, its contribution to the overall expected cost is $(1 - E) \cdot 1$. For the second case, it occurs with a Bloom filter's error rate E. And the query evaluation needs to descend into the tree node's direct children at level $l - 1$. The cost should be equal to the sum of expected costs at all the children nodes. Suppose each tree node has 2 children, the contribution of the second case to the overall expected cost is $E \cdot 2X_{l-1}$. Overall, we have the following and drive a closed-form for the expected cost.

$$\begin{aligned} X_l &= 2E \cdot X_{l-1} + (1-E) \\ &= (2E)^2 \cdot X_{l-2} + (1-E)(1+2E) \\ &= (2E)^3 \cdot X_{l-3} + (1-E)[1+2E+(2E)^2] \\ &\dots \\ &= (2E)^l \cdot X_0 + (1-E)[1+2E+(2E)^2+\cdots+(2E)^{l-1}] \\ &= \frac{1-E}{1-2E} + (2E)^l\big(c + \frac{1-E}{1-2E}\big) \end{aligned} \tag{6}$$

The last step is due to that $X_0 = c$. Because $E \ll 0.5$, we can approximate $X_l \to X = \frac{1-E}{1-2E}$. ■

The implication of this theorem is that when E grows large, the tree node would not be very useful in terms of non-membership test. For example, when $E \to 0.5$, $X \to \infty$, making the INCBM-TREE useless. Recall that in Equation 3 E increases with the number of data records b (also the batch size). The small error rate E is one of the factors that constrain the growth of a INCBM-TREE during its construction and limits its batch size. We describe the digest construction using INCBM-TREE in the next.

D. Digest Construction

In our framework, the data stream is signed on the batch digest constructed using the root node of INCBM-TREE. We describe the incremental construction process of an INCBM-TREE. Algorithm 1 illustrates the construction process; the incoming data stream is partitioned to small batches and a KOMT digest is constructed for each batch[4]. Then, the

[4]KOMT is used only for small batch of data; it is designed so because of relative inefficiency of INCBM-TREE with small batch – a bloom filter for very few records can be a waste of space.

constructed KOMT digests are put into the leaf level of an grow INCBM-TREE. In particular, merging two INCBM-TREE nodes is based on calculation in Equation 2. If it succeeds in merging, the merged node is promoted to one level up and replaces the former node. Then the same merging process repeats, until either it reaches the root node or there is no former node at the current tree level. During construction, it only needs to maintain the "frontier" nodes, but not interior nodes inside the tree, as shown in Figure 4a. Depending on certain conditions, the final batch digest is produced by exporting the hash of the highest root. There are different conditions to trigger a growing INCBM-TREE to be exported and signed; it could be the hitting of the memory limit, and/or crossing of the error rate lower bound of INCBM-TREE.

Algorithm 1 IncBuild(Key-value batch s, incubator p)

```
currentNode←keyOrderedMerkleDigest(s)
l ← 0
node← p.removeTreeNode(l)
loopnode≠NULL
    currentNode← merge(currentNode, node)
    l ← l + 1
    node← p.removeTreeNode(l)
end loop
p.insertTreeNode(currentNode, l)
if p.overflow() then
    p.exportINCBM-TREE()
end if
```

E. Implementation and Integration

In implementation, we integrate INCBM-TREE with KOMT. The memory is utilized 1) to hold the data to construct KOMT of small batches, and 2) to hold the Bloom filters of the constructed KOMTs. The proportion of INCBM-TREE to KOMT denoted by q can be of different values, ranging from 0 to 1. In practice, the choice of value q should be bounded by the available memory size. Since an INCBM-TREE can essentially build arbitrarily large batches (bounded only exponentially due to the incremental construction), we do not flush data from memory to a disk in an INCBM-TREE yet still be able to construct the digest of the same large batch.

1) Cost Analysis: We analyze the possible batch size that can be achieved by a limited memory size S. As a starting point, we first consider the case that the memory is all dedicated to a KOMT; in this case, the memory can accommodate batch of size $b_\mathrm{b} = \frac{S}{s_r}$. In our system, a KOMT only occupies $\frac{q}{q+1}$ of the whole memory, leading to the batch of KOMTs be able to host $\frac{b_\mathrm{b} \cdot q}{q+1}$ records. An INCBM-TREE occupies the rest of memory $\frac{S}{q+1}$. Recall that each leaf node in the INCBM-TREE corresponds to the dataset of an INCBM-TREE. Suppose the compression ratio of INCBM-TREE leaf node is p; in other words, given a dataset of size $\frac{S \cdot q}{q+1}$, the INCBM-TREE leaf node is of size $\frac{S \cdot q}{q+1}/p$. Then the second part of memory can host up to $\frac{\frac{S}{q+1}}{\frac{S \cdot q}{q+1}/p} = \frac{p}{q}$ nodes of the INCBM-TREE. Based on the property of incremental construction, the INCBM-TREE can grow up to accommodating $2^{\frac{p}{q}}$ leaf nodes, or equivalently, a KOMT. Therefore, the total batch size of an INCBM-TREE is up to $2^{\frac{p}{q}} \cdot \frac{b_\mathrm{b} \cdot q}{q+1}$ records.

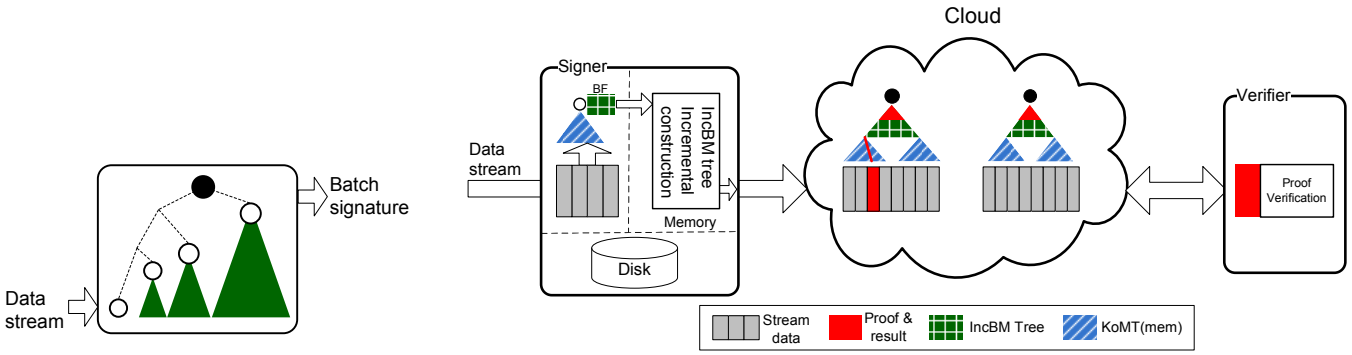
We further analyze the expected query latency. For an INCBM-TREE node with an error rate E, the expected cost is $X = \frac{1-E}{1-2E}$. The latency is $N/(2^{\frac{p}{q}} \frac{q}{q+1} b_\mathrm{b}) \cdot \frac{1-E}{1-2E}$. For the pure KOMT, it is $N/b_\mathrm{b} \cdot \log b_\mathrm{b}$.

F. Real-time Publication

In addition to selection queries, we also consider the model of real-time stream publication for continuous queries. In this scenario, particularly, in the presence of big data stream and powerful clouds, the system bottleneck would be on the owner side which is of limited resources. Especially, the rate of how fast the stream can be signed at the owner side determines how much real-time data can be delivered to the data users. For intensive data streams, a saturating scenario may occur when streaming data arrives at a higher rate than what the data owner's system can handle. The system is bounded by CPU utilization due to the need of frequent signing (by using a batch size to achieve real-time availability) and it is desirable to have a lightweight signing approach.

To handle intensive data streams, we have two design goals in our framework: 1) real-time availability of authenticated streaming data, 2) efficient verification on both real-time and historical data. To be specific, as data update stream comes into the system continuously, we want to minimize and if possible to bound the duration between the time when it is generated and the time when it is stored in a cloud and made available to data users. This real-time processing is important to many latency sensitive applications, ranging from marketing in stock exchange to real-time social news mining. For verification efficiency, the verification cost is dominated by that of historical or static data that are updated long time ago. To authenticate freshness of such data, the proof need to be presented in a way that enables verification all the way from an old time to now or a recent time.

To approach the design goals, our key observation is that batch size is critical to both the real-time-ness and verification overhead. On the one hand, a small batch size reduces the time lapse between data generation and the time when the data become available. On the other hand, a large batch size means high verification efficiency especially for historical data (which will be elaborated in §III-A). Therefore, in this work, we propose a *multi-granularity signing framework*, in which the raw update stream is batched and signed twice, respectively in small batches for real-time publication and in large batches for efficient verification.



(a) INCBM-TREE incremental construction

(b) System of outsourced stream with INCBM-TREE

Figure 4: INCBM-TREE operations and systems

The following two sections respectively describe the design and system of our framework for real-time and periodic publications. The system implementation and deployment of these two components are flexible; they can be co-located in a single machine or two separate machines.[5]

1) Cost Optimization: We formulate the per-record signing cost for optimized computation. We first consider the cost model of our framework. To sign a data stream the cost includes the cost of building digest $\overline{C}_{\text{dig}}(b)$. It can be described as

$$\overline{C}_{\text{dig}}(b) = C_{\text{dig}} \cdot b \log b, \tag{7}$$

where C_{dig} is the cost of a hash operation and $b \log b$ is the cost of building a (binary) Merkle tree on b records and sorting. Given l records in a stream with a configured batch size b, the overall signing cost C_{publ} consists of the cost to build the digests, namely $\frac{l}{b}\overline{C}_{\text{dig}}$, and the cost of applying digital signatures, namely $\frac{l}{b}C_{\text{sig}}$. Note $\frac{l}{b}$ is the number of batches for l records in the stream. By plugging in Equation 7, we have the per-record signing cost $y = \frac{C_{\text{publ}}(x,n)}{l}$ as following,

$$y = \frac{C_{\text{sig}}}{b} + C_{\text{dig}} \log b \tag{8}$$

Our goal is to minimize the per-record signing cost y. We can have following formula based on the first-order condition for local optimum.

$$\left(\frac{C_{\text{sig}}}{b_{\text{opt}}} + C_{\text{dig}} \log b_{\text{opt}}\right)'_b = 0 \tag{9}$$

$$\Rightarrow \begin{cases} b_{\text{opt}} = \frac{C_{\text{sig}}}{C_{\text{dig}}} \\ y_{\text{opt}} = C_{\text{dig}}(1 + \log \frac{C_{\text{sig}}}{C_{\text{dig}}}) \end{cases}$$

2) Real-time Stream Publication: We consider the problem of signing data stream in real time. Formally, we assume the end user has a maximal tolerable delay e for signing. The stream has an arrival rate r. This yields the upper bound of batch size of real-time signing, denoted by b_{r}, as follows.

$$b_{\text{r}} \leq r \cdot e \tag{10}$$

By combining Equation 9 and Equation 10, we can have the following batch size setting.

$$b_{\text{r}} = \begin{cases} b_{\text{opt}} = \frac{C_{\text{sig}}}{C_{\text{dig}}} & \text{, if } b_{\text{opt}} \leq r \cdot e \\ r \cdot e & \text{, otherwise} \end{cases} \tag{11}$$

[5]Currently we do not address the interference of these two system components when co-deployed on a single machine, which is the focus of our future work.

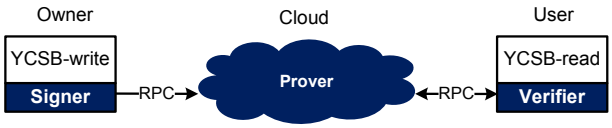


Figure 5: Evaluation system overview

IV. EVALUATION

A. Implementation

We implemented a stream authentication system using INCBM-TREE to evaluate the applicability of INCBM-TREE to generic key-value stores and the performance of INCBM-TREE. Our implementation is mostly based on the cryptographic library provided by JAVA (i.e. `javax.crypto.*`), and utilized RSA as signature schemes. Our implementation also includes various digest structures, such as KOMT and INCBM-TREE, and main system components for verification, such as a signer on the data owner side, a verifier on the data user side, and a prover on the cloud side.

To evaluate the end-to-end stream authentication performance, we built a client-sever system as depicted in Figure 5. We utilized Yahoo! Cloud Serving Benchmark (YCSB)[6], an industrial standard benchmarking tool, to simulate key-value workloads for comprehensive performance measurement.

[6]https://github.com/brianfrankcooper/YCSB/wiki

In particular, we used a write-only workload generator with the provided key distributions to simulate the stream source of web applications used in Yahoo!. We also used a read-only workload generator to simulate data users posing queries to a cloud. Since a cloud is typically not a performance bottleneck, all the data in our experiments in a cloud is processed on a memory. To bridge clients with a remote cloud, we used an RPC library based on Google's ProtoBuf and JBoss's Netty. In order to make the RPC cost less intrusive to our evaluation results, we chose to invoke multiple operations, e.g., verifications of multiple queries in a single RPC call.

1) Equipments: We conducted our experiments in Emulab. The experiments were performed on two machines: a weak machine for clients (i.e., a data owner and a data user) and a powerful machine for a server (i.e., a cloud). The powerful machine was equipped with one 2.4 GHz 64-bit Quad Core Xeon processor (with hyper-threading support) and 12 GB RAM. The less powerful machine was equipped with 3 GHz processor, 2 GB RAM, and 300 GB disk.

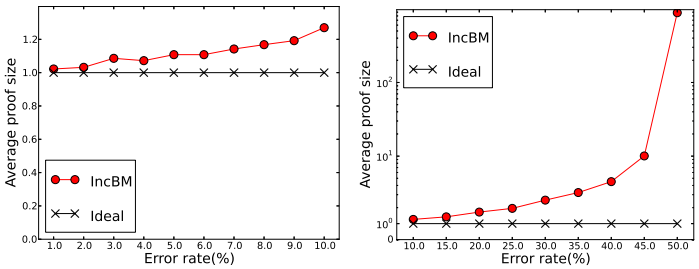
2) Comparison: In our evaluation, we assumed that the batch size was too large to fit into a data owner's memory because a data owner would have a limited computing power in a typical data outsourcing scenario. A data owner would need to flush the streaming data to a disk when a memory cannot hold the entire stream.

We considered KoMT as the baseline for performance comparison in that KoMT has been widely used in prior steam authentication work [11], [8]. We compared the performance of IncBM-Tree to the performance of the on-disk KoMT. On-disk KoMT is a variant of KoMT to support a large batch size; however, it may need to retrieve the data from a disk and perform a merge sort to sign all the data. We set the same batch size for IncBM-Tree for fair performance comparison. However, note that IncBM-Tree can be constructed incrementally without holding entire data before signing. In our experiments, we evaluated two different IncBM-Tree to KoMT ratios, e.g., 0.1 and 0.2. We fixed the error rate of Bloom filters as 0.1.

B. Micro-benchmark

1) Proof Construction Cost: We measured the proof construction cost for non-membership by varying the error rate of Bloom filters in IncBM-Tree. We changed the size of a Bloom filter to change error rates. Under each setting, we repeated the experiments for 1000 times and plotted the average of the proof sizes in Figure 6. With small error rates (e.g., $< 10\%$), the proof size was very small, e.g., slightly above 1. With large error rates (e.g., $\geq 10\%$), the proof size exponentially increased as shown in Figure 6b. The result was consistent with our cost analysis discussed in Equation 5.

2) Bandwidth Cost of Publication: In order to evaluate the bandwidth cost of publication, we measured the size of

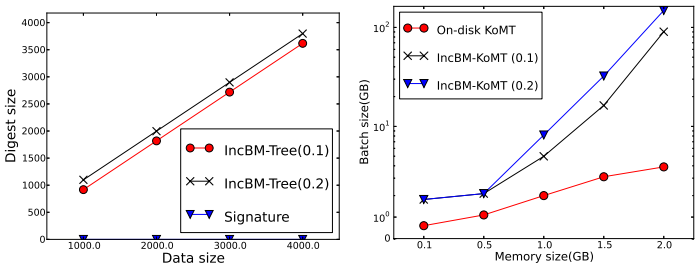


(a) Small error rate (b) Large error rate

Figure 6: Effect of the error rate of a Bloom filter on the constructed proof size (Y axis in a log scale)

constructed digests. A digest size matters a lot to a cloud where a digest has to be fully materialized. We measured the digest size for IncBM-Tree while varying the data size. We also included the size of digital signatures as comparison points as shown in Figure 7a. A digest size, measured by the number of hash values, increased linearly to the data size, measured by the number of records in the stream. A digest was significantly larger than a signature, which supported our design choice where we did not transmit a digest to a cloud.

3) Maximal Batch Size: We measured the maximal batch size that can be achieved under the constraint of varying memory sizes. As depicted in Figure 7b, the maximal batch size of KoMT was linearly bounded by the memory size while the maximal batch size of IncBM-Tree was exponentially bounded. With different ratios q, the maximal batch size differs. A large value of q (e.g., $q = 0.2$) means more space is dedicated to IncBM-Tree and a big batch can be constructed.



(a) Digest size (b) Maximal batch size

Figure 7: IncBM-Tree digest size and maximal batch size

C. Write Performance

We comprehensively evaluated the write performance when publishing stream with large batches. A large batch size is desirable for a better performance as discussed in §III-A.

1) Experiment Setup: We measured the performance of ingesting a data stream. In our setup, 300 million key-value pairs were produced by YCSB's workload-A generator [18] under a Zipf key distribution, and were fed into the owner-side signer. While driving the workload to the system, we measured the sustained throughput and latency over the course of time. In a simulated environment, we only measured the overhead from digest construction and signature generation. We saturated the system by setting the targeted throughput to be higher than sustainable throughput. We fixed the compression ratio of a Bloom filter as 50, that is, the size of a Bloom filter was less than $\frac{1}{50}$ of the data size.

2) Time-series Results: The time-series result was reported in Figure 8a. We did not include the initial data loading stage to exclude unstable system factors, e.g., cold cache. While the throughput of IncBM-Tree remained stable and high, the throughput of KoMT fluctuated along the timeline. At the valley points, KoMT was performing heavy disk I/Os to flush overflowing data and to load data from a disk to a memory for signing. Due to the reason, the average throughput of KoMT was lower than that of IncBM-Tree.

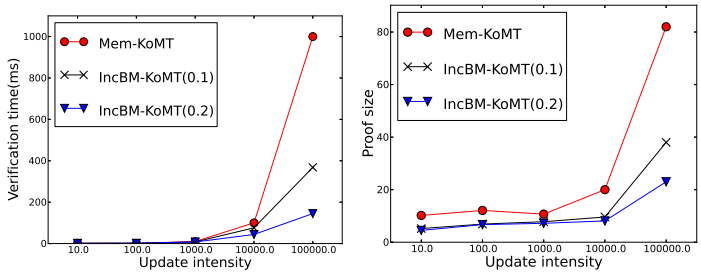
3) Average Throughput: We repeated the above primitive experiments multiple times under different settings and reported their average. We first varied batch sizes under a fixed memory size 0.5GB so that tested batch sizes were always bigger than memory sizes. Figure 8b reports the throughput. IncBM-Tree achieved an order of magnitude higher average throughput than on-disk KoMT. As the batch size increased, the throughput of KoMT decreased because more disk accesses were required for signing. The throughput of IncBM-Tree remained almost the same across various batch sizes because of the incremental digest construction. IncBM-Tree achieved much higher throughput than KoMT due to the pure memory operations without disk I/O. We then varied the memory size under the fixed batch size of 4GB. As described in Figure 8c, the throughput of KoMT increased with larger memory size mainly because of fewer disk I/Os. The throughput of IncBM-Tree were remained stable with different memory sizes.

D. Query Performance

1) Experiment Setup: We further conducted experiments to measure the query performance, specifically the verification cost. In our setup, we used the write workload generator to populate data to a cloud, and the cloud rebuilt digest structures, e.g., KoMT and IncBM-Tree. Then, we used YCSB with a read-only configuration to generate query workload to the cloud. The query keys were generated by YCSB workload-A generator. Upon receiving queries, a prover prepared query results with proofs in a cloud. We were not interested in the performance of a cloud because a cloud is typically not a performance bottleneck; thus we were mainly concerned with the performance of the client-side, e.g., the user-side verification cost.

A data user issued a series of read requests to retrieves proofs from a cloud and verify them. We used a YCSB framework to measure the elapsed time. During the experiments, we varied an update intensity that determined the time interval between the query time ts and the time of the last update ts_{latest}.

2) Verification Performance: We compared the verification performance of IncBM-Tree with the performance of on-memory KoMT with a small batch size. We used the verification time and the proof size, measured by the number of hashes or signatures included in a proof, to evaluate the verification performance. In term of the verification time (Figure 9a) and the proof size (Figure 9b), IncBM-Tree outperformed KoMT. The result demonstrated the non-membership test of IncBM-Tree was highly efficient for freshness verification. On the contrary, KoMT could utilize only a small batch size due to memory constraint. The performance gap between IncBM-Tree and on-memory KoMT was little in Figure 9b because on-memory KoMT could return more signatures than IncBM-Tree with the same proof size.

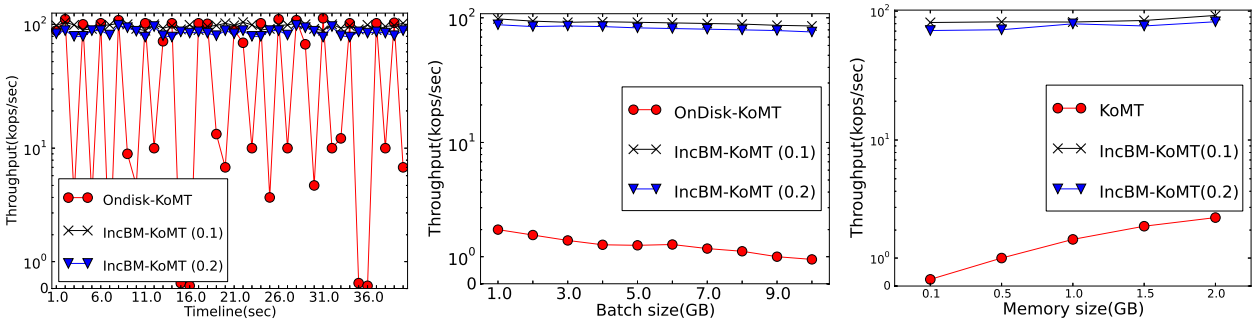


(a) Time (b) Proof size

Figure 9: Verification performance

V. Related Work

Security issues such as data integrity and confidentiality have become increasingly concerned in cloud computing and in an outsourced database scenario. Hacigumus et al. introduced general security issues in an outsourced database [19]. There has been many emerging work addressing the confidentiality issue in outsourced data [20], [21], [22]. Our work is complementary as we address data integrity and authenticity. Recently, several work has been proposed to address the query authentication for outsourced databases [1], [2], [3], [4], [5], [6], [7]. Their proposed methods were specialized to specific data types (e.g., raw data or meta-data) and specific data models (e.g., a database or data streams). Early work focused on a traditional database model and often required a data owner to keep a full local copy of the original data for data updates. By contrast, our work, which also focuses



(a) Time series (b) Varying batch sizes (c) Varying memory sizes

Figure 8: Stream write performance

on authentication of queries, allow the querier to verify the correctness of query results while minimizing data owner's computation requirements.

Existing work often employed collision-resistant hash functions and digital signatures to construct verifiable objects that were returned along with query results to a data client for the verification purpose. For example, signature chaining was used to construct authentication information on top of data structures, such as KD-tree and R tree to guarantee query completeness and authenticity [5], [1].

Merkle hash tree (MHT) [23] is Another commonly used authentication structure. Specifically, Devanbu et al. [2] presented a general framework to publish data based on a Merkle tree that allows a data client to authenticate query answers. Pang and Tan [7] combined a Merkle tree with a B-tree where each internal node was associated with a signed digest that were derived from all tuples in the subtree. This allowed a verification object to be constructed regardless of tree sizes and be efficiently verified by edge servers. Yang et al. [6] proposed a similar approach by embedding a Merkle tree into a R-tree to support fast query processing and verification. Merkle trees have been considered to used to authenticate (non)-membership [24], [25], [26], yet most of them consider a static dataset scenario. Li et al. [3] discussed dynamic scenarios where a data owner was allowed to update database records. However, due to the use of a Merkle tree, updating database records entailed expensive overhead of revoking and recomputing digital signatures.

More recently, as a streaming data model becomes increasingly popular in the big data era, authenticating data streams has attracted significant attentions from research communities. To address unique challenges, such as data freshness and lightweight authentication required by the intensive streaming models, several approaches were proposed including [27], [28], [11], [8]. For example, proof-infused stream [11] considered a streaming data model of continuous data updates and window-based queries. It addressed data freshness only on sliding window queries with various predicates (e.g., ranges on multiple data keys). Result freshness of streaming data was discussed in [9] in the context of global aggregations instead of fine-grained (or per-version) selection queries. CAT [29] tackled a query model close to our work, but freshness was provided by expensive update-in-place actions which was not applicable to a intensive stream scenario. CADS [8] built KoMT on top of ToMT (a variant of KoMT to support temporal completeness); however it still required recomputing the hash values and digital signatures.

The main difference of IncBM-Tree from previous approaches is the support of lightweight authentication of intensive data streams with freshness (temporal completeness) guarantees. Most existing work ensured freshness of query results via either coarse-grained window-based approach which only guaranteed data freshness for recent time window [11] or revoking and resigning of the entire dataset which made them less desirable for an data streaming scenario [3], [8].

VI. Conclusion

In this paper, we highlighted and articulated the problem of providing data freshness assurance for outsourced multi-version key-value stores. We proposed IncBM-Tree, a novel authentication structure which offers a set of desirable properties in stream authentication: 1) lightweight for both data owners and end clients, 2) optimized for intensive data update streams, and 3) adaptive to varying delay tolerance. Through extensive benchmark evaluation, we demonstrated IncBM-Tree provided throughput improvement (in an order of magnitude) for data stream authentication than existing work. The superior performance makes our approach applicable particularly to data owners and clients with weak computational capabilities, which is typical for outsourcing scenarios.

References

[1] W. Cheng, H. Pang, and K.-L. Tan, "Authenticating multi-dimensional query results in data publishing," in *Proceedings*

of the 20th IFIP WG 11.3 Working Conference on Data and Applications Security, ser. DBSEC'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 60–73. [Online]. Available: http://dx.doi.org/10.1007/11805588_5

[2] P. Devanbu, M. Gertz, C. Martel, and S. G. Stubblebine, "Authentic data publication over the internet," *Journal of Computer Security*, vol. 11, p. 2003, 2003.

[3] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin, "Dynamic authenticated index structures for outsourced databases," in *SIGMOD Conference*, 2006, pp. 121–132.

[4] E. Mykletun, M. Narasimha, and G. Tsudik, "Authentication and integrity in outsourced databases," *Trans. Storage*, vol. 2, no. 2, pp. 107–138, May 2006. [Online]. Available: http://doi.acm.org/10.1145/1149976.1149977

[5] M. Narasimha and G. Tsudik, "Dsac: integrity for outsourced databases with signature aggregation and chaining." in *CIKM*, O. Herzog, H.-J. Schek, N. Fuhr, A. Chowdhury, and W. Teiken, Eds. ACM, 2005, pp. 235–236. [Online]. Available: http://dblp.uni-trier.de/db/conf/cikm/cikm2005.html#NarasimhaT05

[6] 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, June 2009. [Online]. Available: http://dx.doi.org/10.1007/s00778-008-0113-2

[7] H. Pang and K.-L. Tan, "Authenticating query results in edge computing," in *Proceedings of the 20th International Conference on Data Engineering*, ser. ICDE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 560–. [Online]. Available: http://dl.acm.org/citation.cfm?id=977401.978163

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

[9] G. Cormode, A. Deligiannakis, M. Garofalakis, and S. Papadopoulos, "Lightweight authentication of linear algebraic queries on data streams," in *SIGMOD*, 2013.

[10] S. Nath and R. Venkatesan, "Publicly verifiable grouped aggregation queries on outsourced data streams," in *ICDE*, 2013, pp. 517–528.

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

[12] B. H. Bloom, "Space/Time trade-offs in hash coding with allowable errors," *Communications of the Association for Computing Machinery*, vol. 13, no. 7, 1970.

[13] R. C. Merkle, "A certified digital signature," in *Proceedings on Advances in Cryptology*, ser. CRYPTO '89, 1989.

[14] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, "Bigtable: A distributed storage system for structured data (awarded best paper!)," in *OSDI*, 2006, pp. 205–218.

[15] "http://hbase.apache.org/."

[16] "http://cassandra.apache.org/."

[17] "http://code.google.com/p/leveldb/."

[18] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *SoCC*, 2010, pp. 143–154.

[19] H. Hacigümüs, S. Mehrotra, and B. R. Iyer, "Providing database as a service," in *ICDE*, R. Agrawal and K. R. Dittrich, Eds. IEEE Computer Society, 2002, pp. 29–38.

[20] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich, "Processing analytical queries over encrypted data," *PVLDB*, vol. 6, no. 5, pp. 289–300, 2013.

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

[22] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu, "Plutus: Scalable secure file sharing on untrusted storage," in *FAST*, 2003.

[23] R. C. Merkle, "Protocols for public key cryptosystems," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 1980, pp. 122–134.

[24] M. Blum, W. S. Evans, P. Gemmell, S. Kannan, and M. Naor, "Checking the correctness of memories," *Algorithmica*, vol. 12, no. 2/3, pp. 225–244, 1994.

[25] C. U. Martel, G. Nuckolls, P. T. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine, "A general model for authenticated data structures," *Algorithmica*, vol. 39, no. 1, pp. 21–41, 2004.

[26] C. Papamanthou, R. Tamassia, and N. Triandopoulos, "Authenticated hash tables," in *ACM Conference on Computer and Communications Security*, 2008, pp. 437–448.

[27] A. Chakrabarti, G. Cormode, and A. McGregor, "Annotations in data streams," in *Automata, Languages and Programming*. Springer, 2009, pp. 222–234.

[28] G. Cormode, J. Thaler, and K. Yi, "Verifying computations with streaming interactive proofs." *PVLDB*, vol. 5, no. 1, pp. 25–36, 2011. [Online]. Available: http://dblp.uni-trier.de/db/journals/pvldb/pvldb5.html#CormodeTY11

[29] D. Schröder and H. Schröder, "Verifiable data streaming," in *ACM Conference on Computer and Communications Security*, 2012, pp. 953–964.