Lightweight Authentication of Freshness in Outsourced Key-Value Stores

Yuzhe Tang
Syracuse University
Syracuse, NY USA
ytang100@syr.edu

Ting Wang
IBM Research Center
Yorktown Heights, NY USA
tingwang@us.ibm.com

Ling Liu
Georgia Tech
Atlanta, GA USA
ling.liu@cc.gatech.edu

Xin Hu
IBM Research Center
Yorktown Heights, NY USA
huxin@us.ibm.com

Jiyong Jang
IBM Research Center
Yorktown Heights, NY USA
jjang@us.ibm.com

ABSTRACT

Data outsourcing offers cost-effective computing power to manage massive data streams and reliable access to data. 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 authenticity and query integrity to remain in the control of the data owners and users.

In this paper, we address the authenticated data-outsourcing 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 for a low-budget IT department), (ii) immediate authentication of data freshness, (iii) support of authentication in the presence of both real-time and historical data accesses. Extensive benchmark evaluations demonstrate that INCBM-TREE achieves higher 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.

1. 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 from 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, 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 user expects to obtain the "freshest" data. For example, in the online stock exchange, a broker is interested in the latest price of a stock.

While there are different types of data authenticity (e.g. basic data integrity, completeness for range queries [17, 16], etc) in various outsourced systems, data freshness is particularly challenging to authenticate. To break the data freshness, malicious clouds can simply return stale but properly signed data versions. The problem of freshness authentication can be complicated by the presence of 1) big data and 2) accessibility to historical data, as observed in many cloud applications. In order to ensure the users of data freshness, it is critical for the owner to remember the latest data

ACSAC '14, December 08 - 12 2014, New Orleans, LA, USA

http://dx.doi.org/10.1145/2664243.2664244 .

Table 1: Comparing INCBM-TREE with prior work

Outsourced systems	Approaches	Big data	Historical access	Real-time verification	Efficient signing
Data stores	CloudProof [26], SPORC [12]	+	–	–	+
	Iris [30], Athos [13]	+	–	+	–
Streams	CADS [25], ProofInfused [17]	–	–	+	+
Highly dynamic stores	INCBM-TREE	+	+	+	+

acknowledged by the cloud; in the case of big data, such "latest snapshot" of the whole dataset may be simply too big to maintain locally, making it impractical for the owner to remember. On the other hand, accessing historical data (i.e. not only the latest snapshot) is required by many workloads; for example, predictive analysis needs to look at the history to predict the future, or consistency control protocols need to know the latest version at a historical time. Authenticating freshness in the presence of historical data access could even increase the size of local state to maintain on the owner side, because it needs to maintain not only the snapshot but also the full update history of the dataset. In this perspective, we summarize the existing work in Table 1 where symbol +/– denotes that an approach support/does not support a feature. In the table, we consider two additional features for freshness authentication: 3) real-time detection and 4) data write efficiency. Existing audit-based approaches [27, 12, 19] rely on an offline process to detect the violation; such approaches can not guarantee the freshness in real time. Existing real-time detection approaches, such as Iris [30], require an up-to-date local state (e.g. root hash of a remote Merkle tree) on the owner side, which can be expensive to maintain (i.e. each data update requires to read a partial Merkle tree from the cloud and then to locally update the root).

In this paper, we propose a novel authentication framework for multi-version key-value data stores. Our freshness authentication is based on the owner's promises to frequently sign the updates within certain time interval; such assumption is practical in the context of intensive data updates. In this framework, we formalize the problem of freshness authentication to be a non-membership test; for instance, the freshness of a data record updated 5 minutes ago can be authenticated by the non-membership fact that there were no updates of the data in the last 5 minutes. We formally describe our problem and authentication framework in §2 and §3, and introduce a novel construct INCBM-TREE in §4 to address the non-membership test problem. An INCBM-TREE uses a Bloom filter [7] for freshness authentication while enabling lightweight signing and optimized verification. Conceptually, an INCBM-TREE is a Merkle Hash tree (MHT) [21] that embeds a hierarchy of Bloom filters (BFs). In an INCBM-TREE, an MHT signs and protects authenticity of data streams along with their associated BFs, whereas BFs are used in non-membership tests for verifying 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 reading the historical data from the cloud.

In summary, our main contributions are as follows.

- To the best of our knowledge, we are the first to solve the problem of *efficiently outsourcing multi-version key-value stores with verifiable version freshness*, enabling devices with limited computation capabilities to leverage cloud-based data management while ensuring the freshness and authenticity of the outsourced data.
- We propose a novel construct INCBM-TREE to authenticate version freshness, which dramatically reduces freshness verification overhead and signs the data stream efficiently with small memory footprint.
- We evaluate the 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.

2. PROBLEM FORMULATION

2.1 System Model

Our system model, illustrated in Figure 1, considers a data outsourcing scenario; it involves a single data owner (e.g., a small enterprise) and multiple data users (e.g., employees or customers of the enterprise), who are bridged through a public cloud.

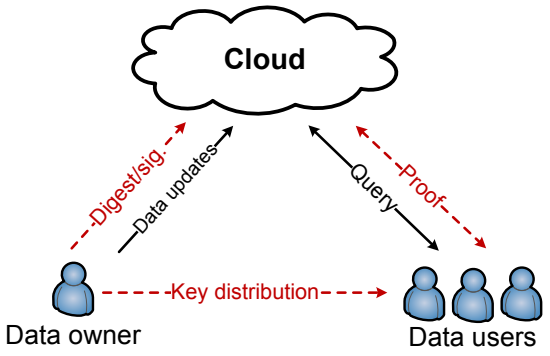


Figure 1: System model and authentication framework

Specifically, in this scenario the basic data flow (as black solid arrows in Figure 1) starts from the data source [1], which generates the big data stream. The semantics of the data stream is that each stream unit is an update to a record in the outsourced dataset in the cloud. The data updates are uploaded to the cloud and applied there.

The data users issue queries against the outsourced data in the cloud. In this work, we assume an SaaS/PaaS cloud model (i.e. Software/Platform-as-a-Service); the cloud provides a managed storage service accessible through the Put/Get API, such as the Amazon S3 service and many other key-value stores [9, 2, 1].[2] With the API, we consider a key-value data model where each object has a unique key k, and multiple overwriting values $\{v\}$, each of which is associated with a unique timestamp t. The cloud storage API is formally described as follows. Note that the timestamp t_q is optional in Get; when absent, it means that Get needs to return the newest version as of now (i.e. the query time).

$$\mathsf{Put}(k, v, t) \rightarrow \text{ACK} \quad (1)$$

$$\mathsf{Get}(k[, t_q]) \rightarrow \langle v, t \rangle \quad (2)$$

[1]We use "data owner" and "data source" interchangeably.

[2]This paper focuses on simple data access, while the support and optimization for advanced queries (e.g. aggregations) can be future work of this paper.

2.2 Threat Model

We assume the minimal trust in our system that data users only trust the data owner.

The public cloud, administrated in a third-party domain, is untrusted. In our model, the cloud can do anything with the outsourced data at its proposal. To break data freshness, the cloud can launch a replay attack [18] to return properly signed but stale data when processing the Get queries. The incentives of mounting such attacks could be many, such as to save computation costs. While the malicious behavior could occur, our authentication framework (as will be described below) guarantees that such behavior can not occur without being noticed by the data users.

Based on our query and threat models, there are specifically two desirable security properties to be guaranteed.

DEFINITION 2.1. *Given a query Get$(k[,t_q])$, the result returned from the cloud, say $\langle v,t\rangle$, is*

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

Table 2: Notations

k: key	v: value
t: timestamp	B: record size
S: memory size	b: KOMT batch size
q: memory ratio for KOMT	l: INCBM-TREE depth
E: error rate for non-membership test	
E_b: bound for E	r: reporting time interval

3. AUTHENTICATION FRAMEWORK

We have described the basic data flow in our system model, and in this section we describe an extra data flow for authenticating this basic flow. The key-value data stream, after being generated from the data source, is locally digested and then signed. As shown by the red dashed arrow in Figure 1, the digest and signatures are uploaded to the cloud. After issuing a query request and receiving the result from the cloud, the data user needs proofs to verify the result authenticity; such proofs are constructed by the cloud. To make the users be able to verify, we assume a key-distribution channel from the owner to the users.

3.1 Flexibility

In our framework, the data source is in the enterprise domain, while the data users can be flexible, either inside or outside the domain. For example, if a data user is an employee of the enterprise, s/he is inside the domain, and if the user is an enterprise customer, s/he can be outside the domain. For the users inside the enterprise domain, the authentication framework may use symmetric key to sign and verify the data for performance concerns; such keys are shared securely between the data source and users in the domain. If the data needs to be shared with the users outside the enterprise domain, then the framework can use the public key infrastructure to authenticate the data and to distribute the public key to the users. The users outside the enterprise domain may maliciously collaborate with the public cloud to forge a proof for compromised data. It is thus necessary to use the public keys which gives users only the ability to verify the data, but not the one to sign the data.

3.2 Freshness Authentication and Non-Membership Test

Our framework primarily focuses on the freshness authentication[3]. Given the design of Get API, it is required to authenticate the result freshness for now or for a historical time t_q. In particular, for freshness as of now, the challenge comes from that the time "now" is constantly evolving; the freshness of every record needs to be constantly reported, even when there is no update on the record. Our framework supports to authenticate relaxed real-time freshness at a time interval of r minutes (e.g. $r = 1$ or $r = 0.1$). We assume that the owner promises to report the authenticated data updates every r minutes. This is a reasonable assumption, and it does not incur dramatically huge overhead onto the owner; in our system, the owner (e.g. an enterprise) is ingesting a high-rate data stream and unlike some other offline owners [12] it needs to be $24/7$ online anyway. In addition, we assume that the physical time on all participating parties (including the data owner, the cloud and data users) is synchronized. Such synchronization can be realized by a reliable trusted time service[4]. Under the promise and assumption, the user can easily authenticate data freshness relaxed by a r-minute interval; that is, the absence of a proper freshness proof for data more than r minutes ago directly means the misbehavior of the cloud. The cause of such misbehavior could be various; the cloud may be unreliable and (unintentionally) lose the data outsourced by the owner, or the cloud may deliberately neglect the fresh data for economic reasons.[5]

The above mechanism of promised frequent reporting provides sufficient conditions to support that a behaving cloud *can* prove the data freshness. In the following, we briefly describe *how* data freshness is verified. Proving the freshness of data $\langle v,t\rangle$ as of time t_q (or now) is equivalent to saying that no newer version $\langle v',t'\rangle$ exists such that $t' \in (t,t_q]$. Formally, given that the owner's data stream is signed in time intervals, it needs to authenticate two facts:

- **Membership of** $\langle k,v,t\rangle$: The outsourced data that is reported in a time interval covering the queried time t indeed contains the key-value data of key k.
- **Non-membership of** $\langle k,v',t'\rangle$: The outsourced data whose time interval falls in $(t,t_q]$ (i.e. the interval's older time bound postdates t and newer time bound predates or equals t_q) does not contain any key-value data of key k.

It is thus essential to design a digest structure that enables both the membership and non-membership tests, which is described in the next section.

4. INCBM-TREE BASED DIGEST

In this section, we describe our digest. We will first overview the digest, then specifically describe the proposed INCBM-TREE, and then the materialization of the digest in our authentication framework. After introducing the mechanism, we will describe policies to optimize the system performance. At the end, we will describe the system implementation and deployment.

4.1 Multi-Level Digest

To enable the (non)-membership test, a baseline digest is to use the Key-ordered Merkle Hash Tree, or KOMT [21, 17, 16, 30, 13].

[3]Correctness can be easily authenticated using the standard techniques, such as MAC or digital signatures [16].

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

[5]We consider the use of a strongly consistent data store in the cloud; that is, the data, once acknowledged being successfully written to the cloud, is made immediately available to the subsequent queries. Real-world data stores of strong consistency include BigTable [9], HBase [2], and Cassandra [1] under certain configuration.

Given a batch of b key-value records, the KOMT digest is constructed by sorting the records based on data key and then building an MHT on top of the ordered list. The KOMT is not well suited for our scenario of outsourcing intensive data streams, due to performance concerns. Specifically, KOMT's performance depends on the size of the data batch, b. With one setting of b, KOMT may achieve efficiency in either signing or verification, but not both (e.g. for a small b, there will be a large number of batches and digests which increases the number of non-membership tests for verification; or for a large b, the owner may need to hold a huge local state which grows out of its memory space).

In this paper, we propose a multi-level digest for efficient key-value data authentication. Our digest is a hierarchical structure of four levels (from the bottom level to the top): 1) a first-level KOMT which is built for each (small) time interval r and serves for the purposes of real-time reporting, 2) a second-level KOMT with batch size b configured based on available memory size, 3) INCBM-TREE that is built on top of the second-level KOMT; it uses Bloom filters to index non-membership information across KOMT's, 4) the chain of root hashes of the previously digested INCBM-TREEs. In the following, we specifically describe the structure of proposed INCBM-TREE.

Design goals. INCBM-TREE's design considers the presence of both historical and real-time data access. In this context, its design goals are: (1) to minimize the amount of data stored on the data-owner side for lightweight data signing, (2) to reduce the proof size for efficient verification of data freshness on the data-user side,

4.2 INCBM-TREE Structure

The basic idea behind the INCBM-TREE is that a Bloom filter can summarize a data set in a space-efficient way and thus facilitates the non-membership test. The structure of an INCBM-TREE is illustrated in Figure 2. Comparing to the traditional Merkle tree, each tree node in the INCBM-TREE maintains not only a hash digest but also a digest value that summarizes the key set in the subtree rooted at the node. For a key set of the subtree, the digest includes a Bloom filter and a value range. For instance, a leaf node 6 maintains a Bloom filter BF_6 summarizing key 1 and key 12 under the node 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 $BF_3 = BF_6 \cup BF_7$. Considering numeric data, 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, $[1, 12] \cup [15, 23]$ (note that there are no data in $[12, 15]$). In total, the digest of the tree node is the hash value of concatenation of all its children's hashes, the range digest, and the Bloom filters, that is, $h_3 = H(h_6 \| h_7 \| BF_3 \| R_3)$. Note that in an INCBM-TREE, Bloom filters at different levels are of the same length. Formally, the INCBM-TREE uses the following constructs.

$$\begin{aligned} R(\text{node}) &= R(\text{left\_child}) \cup R(\text{right\_child}) \\ BF(\text{node}) &= BF(\text{left\_child}) \cup BF(\text{right\_child}) \\ h(\text{node}) &= H(h(\text{left\_child}) \| h(\text{right\_child}) \| BF(\text{node}) \| R(\text{node})) \end{aligned} \quad (3)$$

Security property

THEOREM 4.1. *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 $H(\dots BF_1 \| \dots) = H(\dots BF_2 \| \dots)$. If this is feasible, then it is easy to find two values, $v_1 = \dots BF_1 \| \dots$ and $v_2 = \dots BF_2 \| \dots$, such that $H(v_1) = H(v_2)$, which contradicts the fact that H is a collision resistant secure hash. □

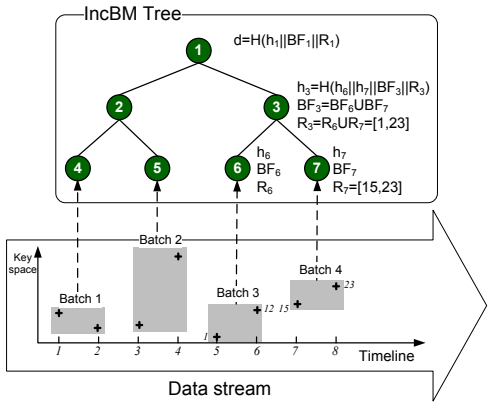


Figure 2: INCBM-TREE structure

4.3 Digest Materialization in the Framework

The digest structure is materialized in our authentication framework at two places: 1) The data owner locally maintains a partial INCBM-TREE as its local state for the signing purpose, 2) The cloud maintains a full copy of the INCBM-TREE for the query verification. In the following, we will follow the authentication data flow to describe the functioning of the materialized INCBM-TREE, that is, the owner-side digest construction and the cloud-side proof construction.

4.3.1 Digest Construction at Owner

The data owner digests a data update stream by maintaining a data batch in memory. Based on the buffered data batch, KOMT is constructed at two levels of time granularity, that is, every r minutes (for level-1 KOMT) and whenever the available memory space is consumed (for level-2 KOMT). At either time point, the owner would sort the buffered key-value data based on key and on the ordered list builds an MHT. The produced root hash for level-1 KOMT is reported to the cloud, while that for level 2 KOMT is added to INCBM-TREE as part of a new leaf node.

For INCBM-TREE construction, upon adding each new KOMT root, the owner maintains the Bloom filter of the current data batch. The maintenance of INCBM-TREE is incremental as illustrated in Algorithm 1; the newly created leaf node in INCBM-TREE would trigger a series of merging actions – If it finds that there are two nodes at the same level, those two nodes will be merged into one node and the new node will be promoted one level up. The process would construct a growing INCBM-TREE; it stops growing when it produces a single node and the error rate of Bloom filter of this node exceeds an error bound denoted by E_b. After a root of the INCBM-TREE is generated, it is chained with previously signed INCBM-TREE roots (i.e. chained in order of time). The signed INCBM-TREE root is then uploaded to the cloud. Note that a key-value record is digested and signed twice, one by level-1 KOMT (for real-time freshness authentication) and the other by the INCBM-TREE (for efficient freshness authentication to historical big data).

An example: We consider $b = 2$, which means KOMT is built every 2 records. At time point 2, a new KOMT is built based on data batch 1 of the first two records. The KOMT's root node is signed and uploaded to the cloud. The root node is also included in a new leaf node of the INCBM-TREE, that is, node 4. Then at time point 4, a similar KOMT is built on the next two records. After

it builds a new KOMT and uploads its root node, a new leaf of INCBM-TREE is created and then merged with node 4 to node 2. Similarly, at time point 6 and 8, two new KOMT's are built based on the new data batches and two new leaves are inserted into the INCBM-TREE. Particularly, at time point 6 it generates a new leaf node 6, and at time point 8 it generates another leaf which triggers the merging of node 6 and node 2 to node 1. At this point of time, only node 1 is maintained in the local state of the owner.

In general, during the construction of the INCBM-TREE, only the "frontier" part of the tree is maintained locally. Figure 3 illustrates a snapshot of the local state maintained by the data owner. The owner's local state consists of the full data copy of current batch for constructing two-level KOMT's, the frontier part of INCBM-TREE and a serials of previous signed roots of INCBM-TREE's.

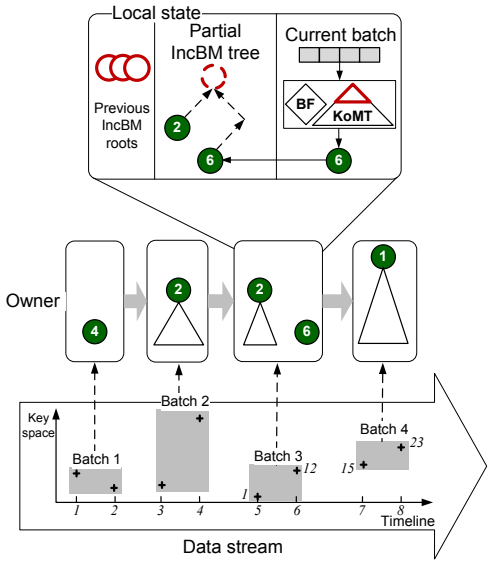


Figure 3: Incremental INCBM-TREE construction on key-value stream: the red shapes are digests that are signed and uploaded to the cloud.

Algorithm 1 IncBuild(Key-value batch s)

```
1: currentNode←keyOrderedMerkleDigest(s)
2: l ← 0
3: node← removeTreeNode(l)
4: loop node≠NULL
5:     currentNode← merge(currentNode, node)
6:     l ← l + 1
7:     node← removeTreeNode(l)
8: end loop
9: insertTreeNode(currentNode, l)
10: if E >= E_b then
11:     uploadINCBM-TREEToCloud()
12: end if
```

4.3.2 Proof Construction and Verification

To see how a proof is constructed and verified, we need to first take a look at the data and digest maintained by the cloud. At any point of time, the cloud maintains data of three kinds: 1) data without any digest; such data must be uploaded to the cloud within the last r minutes, given our real-time reporting mechanism, 2) data with a digest being level-1 KOMT, and 3) data with two digests, being level-1 KOMT and INCBM-TREE. While data 1) can not be authenticated due to lack of digest and signatures, data 2) can be authenticated by the traditional MHT's authentication path [17]. We assume the size of data 2) is small enough to fit into memory and freshness authentication can be efficiently performed. In the following, we hence focus on describing the mechanism for verifying freshness regarding data 3).

For data 3), the INCBM-TREE is used to construct proofs to verify a result of Get query. More specifically, the authenticated Bloom filters can be used to efficiently test the (non)-membership of a particular data key, as required for authenticating the data freshness. Following the example in Figure 2, to prove the freshness of key 98 at time point 8, it suffices to return only two nodes, that is, node 5 and node 3. Because Bloom filter BF_3 can test the non-membership of key 98 in time interval $[5, 8]$ (i.e. node 3), and digest h_3 can be used to verify authenticity of BF_3.

In reality, a Bloom filter can have error in its (non)-membership test. This implies that when a key k is not in a set, a Bloom filter may falsely claim the 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 where all the internal nodes fail to confirm the non-membership, it resorts to the KOMT beneath the INCBM-TREE to test the non-membership or membership. Traditionally, the KOMT can verify the non-membership by returning an authentication path that covers the queried key[17, 16]. By this means, it can always guarantee the error-free verification of data freshness.

4.3.3 Cost Estimation

We first define the error rate of non-membership test, $E(l)$, which is the probability that a non-membership test can fail using a BF at tree level l. We will deduce the mathematical formula of $E(l)$ later.

Recall that our digest construct enforces that $E(l = *) < E_b$. Based on this fact we can estimate the upper bounds of traversing the INCBM-TREE. Specifically, in the traversal, descending one level down only happens when a parent node fails to test non-membership, which occurs with probability $E(l)$. Since $E(l) < E_b$, we can have the fact that descending N levels down occurs with a probability smaller than E_b^N.

THEOREM 4.2. *In an* INCBM-TREE *where the error rates of any Bloom filters at different levels are bounded by E_b, the extra cost X is expected to be the following.*

$$X = \frac{1 - E_b}{1 - 2E_b} + (2E_b)^l \left(E_b \log b - 2E_b \cdot \frac{1 - E_b}{1 - 2E_b}\right) \quad (4)$$

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. Considering the traversal process of non-membership test, there are two cases: 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 $1 - E_b$, since a Bloom filter's error rate is E_b. When the query cost is 1 for a single node, its contribution to the overall expected cost is $(1 - E_b) \cdot 1$. For the second case, it occurs with a Bloom filter's error rate E_b. 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 $E_b \cdot 2X_{l-1}$. Overall, we have the following and drive a closed-form for the expected cost.

$$\begin{aligned} X_l &= 2E_b \cdot X_{l-1} + (1 - E_b) \\ &= (2E_b)^2 \cdot X_{l-2} + (1 - E_b)(1 + 2E_b) \\ &... \\ &= (2E_b)^l \cdot X_0 + (1 - E_b)[1 + 2E_b + (2E_b)^2 \\ &\quad + \cdots + (2E_b)^{l-1}] \\ &= \frac{1 - E_b}{1 - 2E_b} + (2E_b)^l \big(X_0 - \frac{1 - E_b}{1 - 2E_b}\big) \\ &= \frac{1 - E_b}{1 - 2E_b} + (2E_b)^l \big(E_b \log b - 2E_b \cdot \frac{1 - E_b}{1 - 2E_b}\big) \end{aligned}$$

The last step is due to that $X_0 = E_b \log b + (1 - E_b)$. □

The implication of this theorem is that if we can bound the error rate of all BFs in the INCBM-TREE, we can also bound the cost of non-membership test largely based on the value of E_b. In practice, we set $E_b < 0.5$, then we can have $X_l \approx \frac{1-E_b}{1-2E_b}$ for a reasonably large l.

4.4 Effective System Configuration

In this part, we analyze our authentication framework and give suggestions on configurations to make the framework effective.

We start by deducing the formula of non-membership error rate $E(l)$. For a BF at level l in the INCBM-TREE, the node contains no more than $b \cdot 2^l$ key-value records. The BF's error rate (for membership test) is $E' = (1 - e^{-\frac{b \cdot 2^l}{m}})$ (here we use a single hash function)[23]. For key-value data, we consider a key domain of cardinality 2^{B_k}; that is, B_k is the necessary number of bits for storing a key. Thus, the error rate for non-membership test is the ratio of the likelihood that the error case occurs (i.e. $E' \cdot b2^l/(1 - E')$) to the likelihood that a non-membership test happens (i.e. $E' \cdot b2^l/(1 - E') + 2^{B_k} - b2^l$).

$$\begin{aligned} E(l) &= \frac{E' \cdot b2^l/(1 - E')}{E' \cdot b2^l/(1 - E') + 2^{B_k} - b2^l} \qquad (5) \\ &\approx (e^{\frac{b}{m}2^l} - 1) \cdot b \cdot 2^{l - B_k} \\ &= (e^{\frac{q}{1-q}\frac{l}{B}2^l} - 1) \cdot \frac{S}{B} q \cdot 2^{l - B_k} < \frac{1}{2} \end{aligned}$$

Here, we used approximation that $2^B_k \gg 1$ (e.g. $B_k = 8000$ for a one-KB key). In the last step, we used system parameter q which is the ratio of memory allocated to host KOMT (with the rest of memory to host INCBM-TREE); thus, we can apply $b = Sq/B$ and $m \cdot l = S(1 - q)$ in the equation.

From the equation, we can see that 1) $E(l)$ is a monotonic increasing function to variable l; thus the root node would have the largest error rate among all tree nodes and it suffices to control the root in order to bound the error rates of all nodes in INCBM-TREE. 2) E is very sensitive to increasing the value of l. Such E value is a major factor that limits the INCBM-TREE from growing large and deep.

Setting the depth for INCBM-TREE (i.e. l) is an art. While a deep INCBM-TREE is good for increasing the chance to have a hit on the INCBM-TREE during the proof construction, it may hurt the verification efficiency and contribute to a large proof size when it misses. A rule of thumb is to set l to be a medium value; in our experiment setting, $l = 7$ works well in performance.

4.5 System Implementation

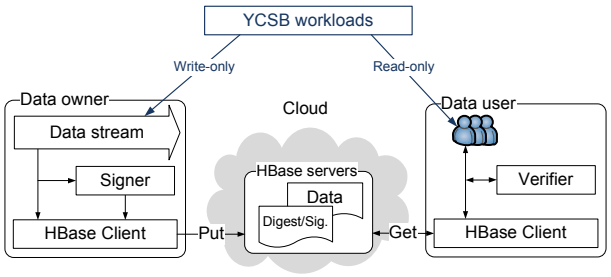


Figure 4: Prototype on HBase and experimental setup

We implemented a proof-of-concept prototype for our authentication framework. We choose HBase [2] to be the substrate system, because 1) HBase is a representative key-value store, which is widely used for many cloud storage services, 2) HBase is optimized toward the write performance; it is designed by a log-structured merge tree [24], which lends itself to persisting the high-rate data stream,3) HBase attains strong consistency, a feature that is required for the freshness authentication by our framework.

The basic data flow in the HBase system is following. The data source installs an HBase client, and through the Put API submits the key-value data to the cloud. The data is then stored in a (base) table in the HBase server cluster. A data user installs the HBase client and through an HBase Get API requests to retrieve the outsourced data in the cloud.

To hook our authentication data flow, we made several system modifications and configurations. First, we attached the signer component to the Put path at the owner side; this is done by instrumenting the HBase client library; we emit every key-value data in the Put path to the signer component, which then builds the digest as was previously described. The digest is submitted to the cloud (more specifically to the "digest table" in the cloud as described below) by the HBase Put call. Note that we only submit the signed root node to the cloud, and leave everything else (e.g. the digest structure in the INCBM-TREE) local; such digests can be recomputed from the outsourced raw data in the cloud. Second, the HBase server in the cloud maintains two tables, one for the base data and the other for the meta-data including the digests and signatures. The digest table is sharded based on the generation time. Third, for proof construction, a baseline implementation is to coordinate the proof construction by data users; after receiving a Get result the user could submit a separate Get request to the digest table. For performance concern and saving the extra communications to the cloud, we implemented an alternate design that automatically triggers the proof construction upon the base table receiving a Get request. This is implemented by using HBase's CoProcessor interface [3] which allows a programmer to attach external actions to HBase's internal events (e.g. receiving a Get call). The proof is constructed by consulting the digest table which stores the full history of authenticated digest structures. After being constructed, the proof is encoded in the Get result to the user. Fourth, after receiving the Get result, the data user parses it and extracts the encoded proof, which is then sent to the verifier component to check the authenticity of the query result based on the owner's public key. The verifier then notifies the end user of the final authentication result. We illustrate our system implementation in Figure 4.

The internal of the authentication framework is implemented based on the cryptographic library in Java (i.e. `javax.crypto.*`). In particular, we used RSA for digital signature.

5. EVALUATION

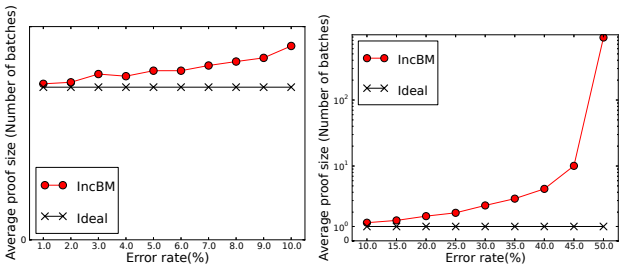
We first describe our simulation result to study the performance of the INCBM-TREE, and then report the experimental result based real system setup.

5.1 Simulation Study

We first conducted simulation-based performance study to justify the hierarchical design of the INCBM-TREE.

5.1.1 Proof Construction Cost

The extra cost for proof construction/verification plays a key role in INCBM-TREE's overall efficiency. We first studied the verification cost based on Equation 4. Particularly, we varied the error rate of Bloom filters in INCBM-TREE (by changing their sizes). Under each setting, we repeated the experiments for 100 times and plotted the average proof size in Figure 5. 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 5b. The result was consistent with our cost analysis in the sense that any Bloom filter should not have error rate too large (e.g. 10%).



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

Figure 5: Effect of the error rate of a Bloom filter on the constructed proof size (Y axis in a log scale)

5.1.2 Non-membership Test Cost

We then compared INCBM-TREE with two alternate designs, KOMT-Only which only maintains KOMT, and KOMT-None which only maintains INCBM-TREE without KOMT. For simulation, we considered all approaches being memory-resident. Instead of the proof size, we used a metric Z in this experiment which measures the per-record cost during a non-membership test. This metric is more suitable for comparison since it is independent to the data and query distribution. For INCBM-TREE, we used the following formula to calculate Z in our simulator. Here, b_1 denotes the batch size used in the INCBM-TREE approach.

$$Z_1 = X/(b_1 \cdot 2^l) \quad (6)$$
$$= \frac{1}{b_1 \cdot 2^l}[\frac{1-E_b}{1-2E_b} + (2E_b)^l(E_b \log b_1 - 2E_b\frac{1-E_b}{1-2E_b})]$$

For KOMT-Only, metric Z is,

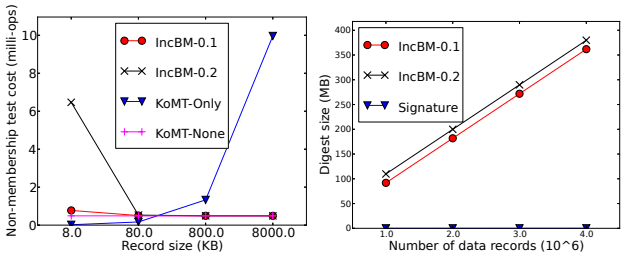
$$Z_2 = \frac{\log b_2}{b_2} = \frac{B}{S} \cdot \log \frac{S}{B} \quad (7)$$

Here, we used two facts: 1) A non-membership test in an MHT of b_2 leaf nodes requires an authentication path from a leaf to the root, which costs $\log b_2$ operations, 2) The KOMT-Only design uses all memory space to store the data batch of KOMT, yielding $b_2 = \frac{S}{B}$.

For the KOMT-None approach, it can be thought as a special case of INCBM-TREE when the KOMT is built on individual data records. By plugging $b_3 = 1$ in Equation 6, we have the following.

$$Z_3 = \frac{1}{2^l}\frac{1-E_b}{1-2E_b}[1-(2E_b)^{l+1}] \quad (8)$$

In our simulation, we used the memory size 1 GB = $10^9 \times 8$ bits, the size of key-value pairs ranging from 1 KB, 10 KB, 10^2 KB to 10^3 KB. We set the INCBM-TREE depth $l = 7$ for both INCBM-TREE and KOMT-None approach; such value is the largest one that obeys our constraints as in Equation 5. For the INCBM-TREE, we consider the different memory ratios, $q = 0.1$ and $q = 0.2$. We report the value of metric Z in the unit of Milli-ops (i.e. 10^{-3} operations) per record. We show our simulation result in Figure 6a. In the result, it is easy to see that approaches using BF, including INCBM-TREE and KOMT-None, incur fewer costs for non-membership tests than KOMT-Only. When the key-value records grow large, the cost of KOMT increases quickly. This result confirms the approximate-linear relationship between metric Z_2 and record size B as in Equation 7 (note that $S \gg B$). For INCBM-TREE, its cost stays small because of the compression effect of BF used in INCBM-TREE.



(a) Non-membership test cost (b) Digest size

Figure 6: INCBM-TREE performance study

5.1.3 Digest Size

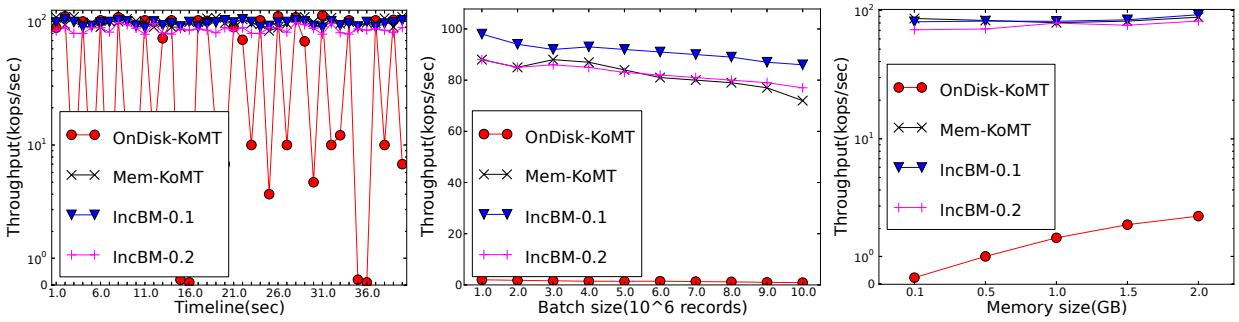
In order to evaluate the bandwidth cost of outsourcing, we measured the size of constructed digests. A digest size matters a lot to a cloud where a digest has to be fully stored there for proof construction. We measured the digest size for INCBM-TREE while varying the number of data records. We also included the size of digital signatures as comparison points as shown in Figure 6b. 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.

5.2 Real Experiment Platform Setup

Based on our prototype system, we used YCSB to drive read-/write workloads, as shown in Figure 4. YCSB[6] is an industrial strength benchmarking tool to simulate various key-value workloads in the cloud. In experiments, we launched two YCSB instances, one to generate write-only workload to the owner, and the other to generate read-only workload to the users.

We deployed our experiment system in Emulab [31]. The data owner and users were set up on two separate machines while the

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



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

Figure 7: Stream write performance

HBase server cluster run on 10 machines for the cloud storage. In the HBase cluster, we installed HBase on top of HDFS. We used one machine for the master node (i.e. HBase HMaster and HDFS namenode); the rest 9 machines were used as the slaves. We used default software configurations for both HBase and HDFS. All machines were homogeneous with 3 GHz CPU and 2 GB RAM.

5.3 Write Performance

Based on the real platform, we first evaluated the write performance of INCBM-TREE. We describe our specific experiment setup and then report the evaluation result.

5.3.1 *Experiment Setup*

We measured the performance of ingesting a data stream. In our setup, 300 million key-value pairs were produced by YCSB; we used a write-only configuration to generate the workloads under a Zipf distribution. The data were driven into the owner-side signer; in this process, we measured the sustained throughput and latency. We saturated the system by setting the targeted throughput to be higher than estimated sustainable throughput.

For comparison, we considered the KOMT-Only design, as it is used by prior work [17, 25]. Particularly, we considered two realizations of KOMT-Only: One was memory resident, and the other was on disk. The memory-only KOMT occupied the whole memory space; that is, the batch size is exactly equal to the memory space. By contrast, the on-disk KOMT may spill data to and retrieve data from disk, and we set its batch size to be three times of the memory space. For INCBM-TREE, it always resided in memory and we tested with memory ratio $q = 0.1$ and $q = 0.2$. Here, one might argue that it is unfair to compare the memory-resident INCBM-TREE with on-disk KOMT. It is a fair setting as we will see that both approaches achieve similar performance in verification – The fact that the on-disk KOMT needs more memory footprint and needs to spill data to disk would show the superiority of the INCBM-TREE design. Likewise, we consider both the in-memory KOMT and in-memory INCBM-TREE; by this way, their write performances are similar (i.e. without disk IO upon signing) and we can then fairly compare them in terms of the verification costs.

5.3.2 *Time-series Results*

The time-series result along the data ingesting process was reported in Figure 7a. 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 time-line. At the valley points, KOMT was performing heavy disk I/Os to flush overflowing data and to load data from disk to memory for signing. Due to the reason, the average throughput of KOMT was lower than that of INCBM-TREE.

5.3.3 *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 7b reports the throughput. INCBM-TREE achieved an order of magnitude higher average throughput than the 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 7c, the throughput of KOMT increased with larger memory size mainly because of fewer disk I/Os. The throughput of INCBM-TREE remained stable with different memory sizes.

5.4 Query Performance

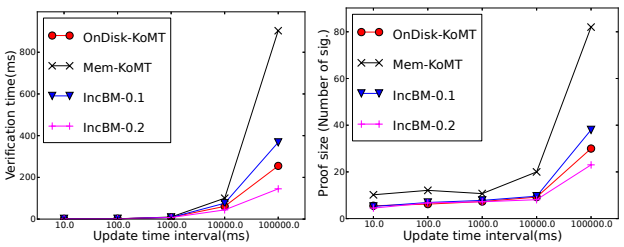
5.4.1 *Experiment Setup*

We further conducted experiments to measure the query verification cost. Before the experiment, we set up the cloud system in advance; we deployed the HBase cluster and populated it using a pre-materialized key-value dataset (which was generated earlier using YCSB). The cloud maintained full copies of different authentication structures (e.g. KOMT and INCBM-TREE), which followed the setting described previously. To conduct the experiments, we drove a read-only YCSB workload (more specifically, the workload-C in YCSB) into the system through data users (as in Figure 4). Recall that the cloud constructs the proof for the query result, and the user verifies the result based on the proof. Here, since we were not interested in the performance of the cloud part (which is typically not the system bottleneck in an outsourcing scenario), we were mainly concerned with the client performance, e.g. the user-side verification cost.

5.4.2 *Verification Performance*

In our query model (i.e. Get (k, t)), different keys k's are updated with different time intervals – While some keys are very fre-

quently updated, other keys are not. Such update frequency plays a key role in determining the verification cost, and we used it as the parameter in our experiment. As for the metric, we measured the proof size and actual verification time. The proof size is captured in the unit of batch numbers. We report our experiment result in Figure 8a for verification time and Figure 8b for the proof size. It is clear to see that both metrics increase linearly with the update time interval, which is expected. Because the older the last update is (as is usual for records updated in larger intervals) the longer history it has to dig into for the non-membership test. For different approaches, INCBM-TREE outperformed the rest, because it uses BF to index the non-membership information across KOMT batches and improves the test efficiency. In particular, on-disk KOMT achieves similar proof size with INCBM-TREE. However, this efficiency for on-disk KOMT comes at the expenses of much higher write overhead and lower write throughput as revealed in Figure 7b.



(a) Time (b) Proof size

Figure 8: Verification performance

6. RELATED WORK

6.1 Data Outsourcing and Security Issues

In the age of cloud computing, data outsourcing has become a popular practice to save the operational overhead. As most cloud services are provided in the public domain and by a third-party operating company, security issues become an inevitable concern in practice. Hacigumus et al. introduced general security issues in an outsourced database scenario [14]. Most security concerns can be attributed to data confidentiality and/or authenticity. While there is a large body of research work for information confidentiality in the public cloud (e.g. querying encrypted data [27]), the concern of our work is on the data authenticity aspect.

Work for authenticated data outsourcing can be classified from the system perspective. There are various outsourced data systems, ranging from pure computations (e.g. MapReduce [28, 8]), to pure storage, and to a hybrid (e.g. a database management system). In particular, the outsourced dynamic databases [16, 25] consider outsourcing data that is subject to updates from the data owners and serves SQL queries to the data users. The outsourced data streams [17, 10, 29] consider a continuous data stream as input and serves stream-specific queries, such as aggregations, continuous queries, etc. Comparing to the databases, data stream system is less concerned about storing data and enabling access to historical big data; instead, it is more interested in processing and serving relatively small data in memory (e.g. recent data within a time window). In the system community, outsourced file systems and key-value stores are considered. While file systems deal with sequential data reads/writes in addition to the random ones, key-value stores are more primitive, typically dealing the very basic Put/Get API. The key-value stores are scalable and highly available, which lends themselves to providing cloud services; there is a trend to superimpose various high-level functions (e.g. databases [5]) on top of key-value stores. Authentication is also studied in the context of P2P networks [15]. In their systems, multiple peers are located in different domains, which makes them mutually untrusted. This model is different from a key-value store, where all nodes are located in a single domain, namely the cloud domain.

In different systems, authenticity has different meanings. For a single data unit (e.g. a data blob), authenticity or data integrity requires that the content can not be changed by the cloud without the notice of end users. For a faulty cloud storage, PDP or provable data possession [6] is authenticity on the data-possession information. When a range query is considered, completeness [17, 16] is an important property that assures that no valid data in the queried range is missed in the result. Our work considers authentication of version freshness in outsourced key-value stores; such freshness is specific to the multi-version Put/Get API exposed by many key-value stores.

6.2 Freshness Authentication

To authenticate data freshness, existing digest structures mainly rely on two approaches; updating an MHT [30, 13] or offline auditing authenticated operation histories (by hash chains) [27, 12].

In the updating-MHT approach, the owner stores a local state for the remote MHT in the cloud. The local state could be the root hash of the MHT [30, 13] or the full copy [16, 20, 21]. Given data updates, the owner needs to update and sign the local state before sending the authenticated updates to the cloud. In this approach, authenticating the updates is inefficient as it either needs to maintain huge on-disk local data as in the full-copy variant or needs the owner to read back authenticated information from the cloud for updates as in the root-hash variant.

The other approach for freshness authentication is to audit operation histories from both sides of the users and cloud; at the end of each epoch, the trusted owner collects the operation histories from the cloud (which is assumed to be willing to collaborate) and data users, and then by comparing the histories, freshness can be authenticated or a violation is detected. By using hash chain and user-supplied randomness, the history information from the cloud can be authenticated so that it can effectively prevent the cloud from forging the history. The audit-based freshness authentication assumes logging responsibility on the user side and can only detect the violation in an offline and delayed manner (i.e. at the end of each epoch).

6.2.1 Systems

CloudProof [26] addresses the authentication of Put/Get in an untrusted key-value store service (e.g. Amazon S3). In the system, the trusted owner is offline in the sense that it is not present in any active Put/Get path from the data users. This design relieves the owner from being $24/7$ online. Data freshness requires a Get to return the latest available data written by Put. CloudProof audits the authenticated history to provide the freshness guarantee.

SPORC [12] supports multi-user collaborative applications on a P2P alike system with a thin server. The sole responsibility of the server is to coordinate and order the operations on the object shared by multiple users. In the system, users or logged-in clients trust each other but they do not trust the server. SPORC authenticates fork consistency (or variants) on multi-version data in a multi-user context. The proposed technique for that is to authenticate the committed write history of a user using hash chain and to verify the new writes by directly contacting the original client. In a similar setting, Depot [19] verifies the freshness of data obtained from the cloud by periodically communicating the trusted peer clients.

In the database community, various data management systems consider dynamic dataset updated by a stream of data writes. To guarantee data freshness under this dynamic setting, existing work [25, 16] relies on the traditional mechanisms for document certificate validation [22], such as publishing revoked signatures and signing with expiration time (and resign upon expiration).

To authenticate freshness in real-time, prior work [32] assumes a trusted computing base in the cloud server. Iris [30] considers the outsourcing of an enterprise file system to the cloud; the reads and writes are issued from the users of the same enterprise domain and the trusted enterprise is present in every read/write path. In the presence of updates to file block, Iris addresses the freshness problem – it is the latest version of the file block in the cloud that should be returned. Iris updates the MHT for freshness; it is guaranteed by maintaining the latest version numbers of each file block in a Merkle tree and locally maintaining its root hash. Persistent authenticated dictionaries or PADs [11, 4] are a performance-optimized data structure to authenticate multi-version data with key-completeness in the presence of historical access. As PADs do not particularly address the freshness authentication problem, they are complementary with INCBM-TREE.

7. 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 intensive stream authentication: 1) lightweight for both data owners and end users, 2) optimized for intensive data update streams, and 3) immediate authentication of data freshness in the presence of real-time and historical data accesses. Through extensive benchmark evaluation, we demonstrated that 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.

Acknowledgment

Yuzhe Tang and Ling Liu were partially supported by the National Science Foundation under Grants IIS-0905493, CNS-1115375, IIP-1230740, and a grant from Intel ISTC on Cloud Computing. This research was partially sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defense and was accomplished under Agreement Number W911NF-06-3-0001. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the ofiňĄcial policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defense or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

8. REFERENCES

[1] http://cassandra.apache.org/.
[2] http://hbase.apache.org/.
[3] https://blogs.apache.org/hbase/entry/coprocessor_introduction.
[4] A. Anagnostopoulos, M. T. Goodrich, and R. Tamassia. Persistent authenticated dictionaries and their applications. In *Information Security ISC 2001*, pages 379–393, 2001.
[5] M. Armbrust, K. Curtis, T. Kraska, A. Fox, M. J. Franklin, and D. A. Patterson. Piql: Success-tolerant query processing in the cloud. *PVLDB*, 5(3):181–192, 2011.
[6] G. Ateniese, R. C. Burns, R. Curtmola, J. Herring, L. Kissner, Z. N. J. Peterson, and D. X. Song. Provable data possession at untrusted stores. In *ACM CCS*, pages 598–609, 2007.
[7] B. H. Bloom. Space/Time trade-offs in hash coding with allowable errors. *Communications of the Association for Computing Machinery*, 13(7), 1970.
[8] B. Braun, A. J. Feldman, Z. Ren, S. T. V. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *SOSP*, pages 341–357, 2013.
[9] 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*, pages 205–218, 2006.
[10] G. Cormode, A. Deligiannakis, M. Garofalakis, and S. Papadopoulos. Lightweight authentication of linear algebraic queries on data streams. In *SIGMOD*, 2013.
[11] S. A. Crosby and D. S. Wallach. Super-efficient aggregating history-independent persistent authenticated dictionaries. In *ESORICS 2009*, pages 671–688, 2009.
[12] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. Sporc: Group collaboration using untrusted cloud resources. In *OSDI*, pages 337–350, 2010.
[13] M. T. Goodrich, C. Papamanthou, R. Tamassia, and N. Triandopoulos. Athos: Efficient authentication of outsourced file systems. In *ISC*, pages 80–96, 2008.
[14] H. Hacigümüs, S. Mehrotra, and B. R. Iyer. Providing database as a service. In *ICDE*, pages 29–38, 2002.
[15] A. Kapadia and N. Triandopoulos. Halo: High-assurance locate for distributed hash tables. In *NDSS*, 2008.
[16] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *SIGMOD Conference*, pages 121–132, 2006.
[17] F. Li, K. Yi, M. Hadjieleftheriou, and G. Kollios. Proof-infused streams: Enabling authentication of sliding window queries on streams. In *VLDB*, pages 147–158, 2007.
[18] J. Li, M. N. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (sundr). In *OSDI*, pages 121–136, 2004.
[19] P. Mahajan, S. T. V. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. In *OSDI*, pages 307–322, 2010.
[20] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39(1):21–41, Jan. 2004.
[21] R. C. Merkle. A certified digital signature. In *Proceedings on Advances in Cryptology*, CRYPTO '89, 1989.
[22] S. Micali. Efficient certificate revocation. 1996.
[23] M. Mitzenmacher and E. Upfal. *Probability and computing - randomized algorithms and probabilistic analysis*. Cambridge University Press, 2005.
[24] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, 1996.
[25] S. Papadopoulos, Y. Yang, and D. Papadias. Cads: Continuous authentication on data streams. In *VLDB*, pages 135–146, 2007.
[26] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang. Enabling security in cloud storage slas with cloudproof. USENIX-ATC'11, pages 31–31, 2011.
[27] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *SOSP*, pages 85–100, 2011.
[28] I. Roy, S. T. V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and privacy for mapreduce. NSDI'10, pages 20–20, Berkeley, CA, USA, 2010. USENIX Association.
[29] D. Schröder and H. Schröder. Verifiable data streaming. In *ACM CCS*, pages 953–964, 2012.
[30] E. Stefanov, M. van Dijk, A. Juels, and A. Oprea. Iris: a scalable cloud file system with efficient integrity checks. In *ACSAC*, pages 229–238, 2012.
[31] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI*, 2002.
[32] H.-J. Yang, V. Costan, N. Zeldovich, and S. Devadas. Authenticated storage using small trusted hardware. In *CCSW*, pages 35–46, 2013.