An integrity-preserving technique for range queries over data streams in two-tier sensor networks

https://doi.org/10.1016/j.comnet.2022.109316

An Integrity-Preserving Technique for Range Queries over Data Streams in Two-Tier Sensor Networks

Francesco Buccafurri[a,*], Vincenzo De Angelis[a], Gianluca Lax[a]

[a] *University of Reggio Calabria, Via Università, 25, Reggio Calabria, Italy 89124*

Abstract

Two-tier sensor networks enable energy and computation saving due to the introduction of non-constrained storage nodes acting as intermediates between the sensors, which provide data, and the sink, which needs to access these data by submitting queries to the storage nodes. However, the storage nodes maintain a lot of data coming from different sensors and represent a single point of failure targeted by the attackers. In this scenario, robust guarantees about the integrity of the query result (i.e., completeness, freshness, and correctness) should be provided. This is a very well-known problem, traditionally faced through Merkle-Hash-Tree-based data structures. This paper presents a technique more efficient than the state-of-the-art methods for data insertions and deletions, supporting range queries over even non-temporal dimensions. Thus, the proposed solution appears suitable to the considered scenario, in which data can be updated with high frequency and insertions and deletions are executed by constrained devices.

Keywords: Two-tier Sensor Networks, Query Integrity, Range Queries, Data Streams.

1. Introduction

Two-tier sensor networks are widely investigated in the literature for the benefits in terms of energy and computational saving they provide [1, 2]. In the traditional one-tier sensor networks, data are collected by the *sensors* and sent, possibly through other sensors, to a node called *sink*. Since the sensors are involved in the transmission of data towards the sink, the energy consumption may result too high. Then, in a two-tier sensors network, the introduction of non-resource-constrained nodes with large storage capability, called *storage nodes*, allows us to reduce the effort required to the sensors.

In this model, the sensors send data directly to the storage nodes, that collect and store them. The sink retrieves the data by submitting some queries to the storage nodes.

Despite the benefits provided when adopting the two-tier architecture, new security problems arise since the storage node represents a point of failure targeted by an attacker.

The least severe assumption is that the only malicious goal of the attacker is to illegitimately draw from data some knowledge useful for its business or matter of commercial exchange. In this case, the attacker is defined *honest-but-curious*, and the security attribute compromised is *confidentiality*.

However, it is definitely realistic to consider more pessimistic threat models, in which the attacker behaves maliciously also during the executions of processes and protocols, thus attacking the *integrity* of data stored in the storage node and queried by the sink.

This paper is placed in the second case above, specifically focusing the attention on the problem of *query integrity* [3].

In a two-tier sensor network, the query integrity problem applies when the sink submits a query to the storage node. The problem is to guarantee three properties for the result of the query: *completeness* (no piece of information is omitted), *freshness* (the newest version of each data is returned), *correctness* (the values of returned data are not altered).

In the literature, many solutions have been proposed to face the query integrity problem [4, 5, 6, 7]. These approaches are based on the definition of new data structures including some extra information needed to verify the integrity of the query result.

However, the above research works are not tailored for sensor networks and solve a more general scenario in which the entity querying the data is represented by several clients and not just a single node (the sink).

In this case, public verifiability of query integrity must be provided and this enforces to have revocation mechanisms and to make assumptions on synchronizations.

On the contrary, in a two-tier sensor network, the general model is simplified, because only private verifiability is required and it can be obtained with a secret key shared between the sink and the sensors [8, 9].

Therefore, tailored solutions for query integrity in two-tier sensor networks are proposed in the literature [8, 10, 11, 12, 13]

However, as for queries, we restrict to the case of *range queries* [14], over any of the dimensions of multi-dimensional data. As matter of fact, range queries are typical queries in the domain of data streams (in which ranges can be temporal, spatial, or regarding some physical measure).

For range queries, the state-of-the-art solutions are based on

*Corresponding author
Email address: bucca@unirc.it (Francesco Buccafurri)

the notion of Merkle Hash tree (MHT) [15], which has the drawback of not being efficient in the case of very dynamic data.

The aim of our work is to find a solution that overcomes the limits of MHT-based approaches with respect to dynamic data. As a matter of fact, in this scenario, insertions and deletions are very frequent, and, importantly, all the extra operations (to update the structure used for integrity verification) must be done at the source, possibly by constrained devices.

Our approach improves the state-of-the-art techniques regarding the insertion from $O(log(n))$ to $O(1)$ (where n is the size of the stored data stream window), with no asymptotic price in terms of verification cost. This paper starts from the results given in [16], in which only the case of temporal range queries is considered and only the append operation is allowed as insertion. This study removes the above restrictions, by considering any possible attribute on which range queries can be performed. Consequently, the general case of insertion is handled, because the insertion of a new tuple into the data stream (i.e., an append operation with respect to the temporal dimension) corresponds to a generic insertion with respect to another dimension. Similar considerations can be done for cancellations from an arbitrary position of the dimension domain, which arise from the implementation of the sliding window mechanism over the data stream.

The structure of the paper is the following. In Section 2, we provide an overview of related work. A new data structure, called *integrity chain*, defined to maintain data and verify their integrity is presented in Section 3. In Section 4, this structure is applied to the two-tier sensor network architecture. The computational costs of the operations of our structure are discussed in Section 5. In Section 6, these costs are compared with the costs of MHT operations through an experimental analysis. Section 7 provides a security analysis to show how *correctness, freshness, and completeness* are guaranteed. Finally, in Section 8, we summarize the main outcomes that emerged in this study.

2. Related work

From a more general point of view, this paper can be contextualized in the scientific literature addressing the problem of secure outsourced data management over untrustworthy third party. In this context, one of the issues to face is known as *Provable Data Possession (PDP)* [17, 18, 19, 20, 21]. PDP techniques allow a client that has stored data at an untrusted server to verify that the server possesses the original data without retrieving it.

Another problem, more related to this paper, is *Public Auditability* [22, 23], which is about the possibility for the data owners to delegate a third-party auditor (TPA) to verify the integrity of data stored.

The exact context in which this paper falls is *Query Integrity* [3]. This problem applies when the data owner submits a query to the server storing the data and it should be able to verify if the returned result satisfies: *completeness* (no piece of information is omitted), *freshness* (the newest version of each data is returned), *correctness* (the values of returned data are not altered).

There are two different types of techniques proposed to face the query integrity problem: *probabilistic* and *deterministic* approaches (see [24] for a comprehensive overview).

In probabilistic solutions [25, 26, 27], the result is guaranteed with a certain degree of confidence. For example, in [25], the authors propose a probabilistic scheme for multi-dimensional range query that uses an ordered preserving hash-based function. For integrity verification, the authors define a new data structure called *local bit matrix* that includes information about the neighborhood of data. Another probabilistic solution that guarantees privacy as well as query integrity, is presented in [28].

Deterministic solutions [29, 30, 7] resolve the problem with no uncertainty. In these approaches, the server returns all data requested and a set of additional information called VO (*Verification Object*) that can be used to verify the integrity of returned data. In [7], a solution is presented that allows to perform aggregate queries.

The technique proposed in this paper is deterministic.

Several deterministic techniques [31, 32, 33] are based on Merkle Hash tree (MHT) structures [15], which actually represent the state of the art for this kind of techniques. When tree-like structures are adopted, logarithmic costs in the size of the entire database for data insertion/deletion derive, while verification is linear in the size of the range query. Among tree-based approaches, there is also a very recent proposal published in [23] which uses a ternary Merkle Hash Tree. However, this work deals with a more general problem including error localization with replica for data correction. Moreover, adopted cryptographic tools (i.e., signatures and bilinear maps) are more complex than those of our approach. As far as query integrity is concerned, the paper does not obtain better asymptotic costs than the state-of-the-art (MHT-based) techniques.

Besides Merkle-Hash-Tree-based approaches, signature-based schemes are proposed [34, 35, 36]. In these solutions, each tuple is signed and an aggregate signature is returned by the server storing the data. This reduces the size of the VO and improves the communication overhead. However, since these techniques require much more computational power than Merkle-Hash-Tree-based techniques, they do not outperform MHT-based techniques.

Even though the above proposal could be adopted in general scenarios, in this paper we consider a specific, but very relevant and well-studied, scenario that is two-tier sensor networks [1, 2].

In this scenario, we can take advantage from the fact that public verifiability (i.e., the possibility that more distributed parties can verify the integrity of the query result without sharing any secret key) is no longer needed (just the sink has to verify the integrity of data).

On the other hand, in these cases, it becomes critical to obtain efficient update operations since they are performed by resource-constrained devices.

In the literature, several proposals for query integrity in two-tier sensor networks are available [8, 10, 11, 12, 13].

However, among these solutions, we refer to those considering range queries [10, 37, 11, 12] (i.e., queries involving the

values between a lower bound and an upper-bound) in which the MHT-approach represents again the state of the art.

In this paper, we propose a technique that outperforms tree-based approaches, moving from $O(log(n))$ cost of insertion and deletion to $O(1)$ (where n is the size of the stored data), with no asymptotic price in terms of verification cost. This improvement can be very relevant whenever sensors produce data stream (highly dynamic data) and insertion/deletion operations must be secured at the source by constrained devices.

Finally, this paper takes origin from a proposal presented in [16] (and discussed also in [38] from an application point of view). However, this work represents a significant evolution of that initial idea in a number of directions. First, in [16], only range queries over the attribute *time* are allowed, while we treat general range queries (over any totally ordered attribute). Moreover, while [16] supports only *append* operations (which correspond to insertions of new tuples in a data stream), we allow, in this paper, *insert* and *delete* operations in a generic position of the attribute order. We observe that this feature makes the approach applicable to any situation, thus not just to data streams, but even when the case of data streams is considered, it allows us to perform range queries over any other attribute than the time. Indeed, the insertion of a new tuple into the data stream (i.e., an append operation with respect to the temporal attribute) corresponds to an insertion in a generic point of the domain of other attributes. Similar considerations can be done for deletions. When a sliding window mechanism over the data stream is implemented, tuples going out from the sliding window correspond to generic deletions with respect to the other attributes than the time.

Another advantage is that we reduce the verification cost and the VO size with respect to [16]. In fact, to build integrity chains, we compute the HMAC function not directly on the tuples, but on their digests. This means that the storage node, as extra information (i.e., VO), must return only digests (instead of entire tuples) together with some schema information (i.e., the markers). Still referring to extra information, this also reduces the computational effort of the sink to verify the query results, because HMAC must be applied on short messages (i.e., digests) instead of tuples. Finally, the submitted paper includes also a formal proof of security (not included in the conference paper).

3. A new data structure for efficient update operations

In this section, we describe the model allowing us to guarantee range-query *integrity*, that corresponds to the following three properties:

1. *Completeness*: All tuples involved in the queries have to be returned.
2. *Correctness*: The returned tuples have not to be corrupted.
3. *Freshness*: The newest version of tuples has to be returned

We consider a traditional two-tiered sensor network scenario with three actors:

- **Sensors**: Constrained devices that collect data and send them to the storage node with high frequency. As common in literature [8, 9], they are equipped with a secret key (stored in tamper-proof hardware) shared with the sink node.
- **Storage node**: The node that receives and stores data coming from the sensors and responds to the queries performed by the sink node.
- **Sink node**: The entity who submits range queries (over any attribute of the data) to the storage node and verifies the integrity of the result. It shares a secret key with the sensors. As will be clear in Section 4, the sink node also deletes data stored in the storage node.

This section is devoted to the description of a new data structure allowing efficient update operations.

The approach is based on the construction of one or more *integrity chains*, which are chains ordered on the attributes on which we want to perform range queries. These chains link tuples of the database through suitably generated message authentication codes (therefore, whose integrity can be checked by the sink node).

We enable a preliminary segmentation of the attribute domain into buckets which belongs to a common (not alterable) schema allowing us to minimize the amount of extra information the storage node must retrieve together with the result of the query in such a way that insertions and deletions (performed by the sensors and the sink, respectively) are very quick ($O(1)$) without paying a significant price in terms of result verification.

We start by giving some preliminary definitions.

We denote by $H(x)$, the application of a secure cryptography hash function (e.g., SHA256) on a message x and by HMAC(k, M), the application of the HMAC function [39] (with any underlying secure cryptographic hash function) with secret key k on a message M. For both H and HMAC, the output is generically called *digest*.

From now on, throughout the paper, consider given a database D composed of tuples with attribute schema $A = (a_1, a_2, ..., a_t)$, and domains, for each attribute $a \in A$, denoted by $U(a)$. W.l.o.g., we assume that $U(a)$ is discrete. Given a tuple t of the database, the value of the attribute $a \in A$ in this tuple is denoted by $val_t(a)$ ($\in U(a)$).

In the next definition, we define the *bucket schema* for a given database. This is done on a given attribute on which the sink wants to perform range queries, so we assume its domain is totally ordered. Like in histogram-based approaches [40, 41, 42], buckets can be obtained according to different rules, like *equi-depth*, *equi-width*, etc., but this issue is only affecting performance aspects and has been widely investigated in the literature also in the specific case of range queries [43, 44, 45]. For this reason, in this paper, we do not focus our attention on this aspect. The bucket schema entails also what we call *ghost schema*, which allows us to record in the integrity chains the deletions occurred in each bucket through dummy elements called *ghost elements*, in such a way that old versions of the database can be always distinguished by the sink. Importantly,

no information about the content of deleted tuples is included in the ghost elements which cannot be linked to deleted tuples by any party but the sink. This is a needed requirement for privacy reasons.

Definition 3.1. Given an attribute $a \in A$ with $U(a)$ totally ordered, a *bucket schema* B (of cardinality k) on a is a sequence $B = \langle b_1, b_2, \dots b_k \rangle$, such that:

1. $b_i \in U(a)$ for any $1 < i < k$ and $b_1, b_k \notin U(a)$;
2. $b_i < b_j$ for $1 < i < j < k$
3. The set $\hat{U}(a) = U(a) \cup \{b_1, b_k\}$ is totally ordered by extending the order of $U(a)$ in such a way that $b_1 < x < b_k$ for any $x \in U(a)$.

The elements $b \in B$ are named *boundaries*.

A *ghost schema* G (on B) is a sequence $G = \langle g_1, g_2, \dots g_{k-1} \rangle$, such that the set $\hat{U}(a) \cup \{(g_s, j_t) |\ j_t > 0, 1 \leq s < k, t > 0\}$ is totally ordered by extending the order of $\hat{U}(a)$ as follows: for each $1 \leq i \leq k-1$ and any positive number j, it holds that:

1. $x < (g_i, j) < b_{i+1}$, where x is the value of $U(a)$ preceding b_{i+1} in the total order of $\hat{U}(a)$.
2. $(g_i, j_1) < (g_i, j_2)$ for $j_1 < j_2$.

The elements $g \in G$ are named *ghost elements*.

Observe that both B and G are public and intentionally generable so that they should not be materialized and stored by any party. The next example shows a possible bucket schema with an associate ghost schema for a simple case.

Example 3.1. Let a be the score result of an Italian university student that has passed an exam. So, $U(a) = \{18, 19, \dots, 30\}$. A possible bucket schema is the following: $B = \langle b_1, b_2, b_3 \rangle = \langle$"$FirstValue$", 24, "$EndValue$"$\rangle$ with standard total order extended by "$EndValue$" > 30 and "$FirstValue$" < 18. A possible ghost schema is $G = \langle 1, 2 \rangle$ with total order extended as follows: "$FirstValue$" $< 18 < \dots < 23 < (1,1) < (1,2) < \dots < 24 < \dots < 30 < (2,1) < (2,2) < \dots <$ "$EndValue$".

Now, we are ready to formally define the data structure we use to guarantee query integrity, which is called *(integrity) chain*. This is done on a given ordered attribute, once a bucket schema is defined on this attribute.

Definition 3.2. Given a bucket schema $B = \langle b_1, b_2, \dots b_k \rangle$ on a given attribute $a \in A$ and a ghost schema $G = \langle g_1, g_2, \dots g_{k-1} \rangle$ on B, we define the *(n-integrity) chain (for D) in a* as the sequence of elements $C = (c_1, c_2, .., c_n)$ such that:

1. $c_n = \langle x, f_C(x) \rangle$ is such that $x = b_k$ and $f_C(x)$ is NULL;
2. for any $1 \leq i \leq n-1$, $c_i = \langle x, f_C(x) \rangle$, where (1) x is either a tuple of D or a boundary of B or $(g, i) \| d$, where g is a ghost element in G, i is a positive number and d is a digest, and (2) $f_C(x) = \text{HMAC}(k, H(x) \| H(x'))$, where x' is the left-most component of the element c_{i+1}, and k is a secret kept by the sensors and the sink;
3. given $c_i = \langle x, f_C(x) \rangle$ and $c_j = \langle x', f_C(x') \rangle$ such that $1 \leq i < j \leq n$, it holds that $val(x) \leq val(x')$, where (from now on):

$$val(x) = \begin{cases} val_x(a) & \text{if } x \text{ is a tuple} \\ x & \text{if } x \text{ is a boundary} \\ (g, i) & \text{if } x = (g, i) \| d,\ g \in G, i \in \mathbb{N}, \\ & \text{and } d \text{ is a digest} \end{cases}$$

4. $c_1 = \langle x, f_C(x) \rangle$ is such that $x = b_1$;
5. given $c_i = \langle x, f_C(x) \rangle$ and $c_j = \langle x', f_C(x') \rangle$ such that $i \neq j$ $(1 \leq i, j \leq n)$ and x, x' are boundaries, it holds $x \neq x'$;
6. given $c_i = \langle x, f_C(x) \rangle$ and $c_j = \langle x', f_C(x') \rangle$ such that x is a boundary and x' is a tuple and $x = val(x')$, it holds that $i < j$.

An element $c = \langle x, f_C(x) \rangle$ such that x is a boundary is called *marker*. If x is a tuple, the element c is said *secured-tuple* (*s-tuple*, for short), otherwise it is said *ghost-tuple* (*g-tuple*, for short). Two markers m and m' are *consecutive* if there is no other marker with boundary between the boundaries of m and m'. Given an element $c = \langle x, f_C(x) \rangle$, we define $val(c) = val(x)$ (see Definition 3.2).

Observe that, thanks to the definition of *val* on the elements of an integrity chain, Property 3 of Definition 3.2 can be also written as follows: For any c_i and c_j $(\in C)$ such that $i < j$, it holds that $val(c_i) \leq val(c_j)$.

Given a g-tuple $c_i = \langle x = (g, j) \| d, f_C(x) \rangle (\in C)$, we denote by $dgs(c_i) = d$.

The sequence of all the s-tuples and g-tuples c of C such that $val(c)$ is between the values of two consecutive markers is called *bucket*, and the values of the two markers are called *lower bound* and *upper bound* of the bucket, respectively. The bucket with boundaries (b_i, b_{i+1}) is said the *i-th bucket*.

In the next definition, we introduce the notion of *secured database*, which is the overall organization of data guaranteeing query integrity. It consists of the outsourced data to the storage node together with a number of integrity chains, one per dimension on which the sink submits queries. Each integrity chain has a state and the states of the integrity chains define the state of the secured database. First, we need the following preliminary definition:

Definition 3.3. A database D is said *nonced* if the attribute schema includes a dummy attribute whose value, in each tuple, is a *nonce*. The other attributes are called *actual*.

Observe that, in a nonced database in which deletion and insertion are allowed, if a tuple t is removed from the database, a tuple t' with the same value as t on all the attributes can never appear in the database, as they will differ at least for the nonce attribute value.

Definition 3.4. A *secured database* D_s consists of:

1. a nonced database D stored in the storage node;

2. at least one integrity chain C (stored in the storage node too) (in an actual attribute of D) with bucket schema B publicly available to the sensors, the sink, and the storage node (thus, not subject of integrity issues) and ghost schema G on B public available to the sink and the storage node;
3. for each chain C, a state S_C, which is a (k_C-1)-sequence of pairs of natural numbers $\langle (nop_1, j_1), \ldots, (nop_{k_C-1}, j_{k_C-1})\rangle$, each associated with a bucket, where k_C is the cardinality of the bucket schema of C. nop_i $(1 \leq i \leq k_C - 1)$ counts the number of insert and delete operations affecting the bucket with boundaries (b_i, b_{i+1}), while j_i counts the number of ghost tuples occurring in the corresponding bucket. From the state of the integrity chain, only the left-most element $(nop_i,\ 1 \leq i \leq k_C - 1)$ of each pair is materialized and stored by both the storage node and the sink.

Observe that the sensors neither need to maintain the nop values nor to access G.

We denote by NOP_C the sequence $\langle nop_1, \ldots, nop_{k_C-1}\rangle$ composed of the first elements of the pairs of S_C.

In the following definition, we define the operations allowed on a secured database. They are INSERT, DELETE, and RANGE_QUERY. INSERT and DELETE must modify the integrity chains, also by changing their state. The RANGE_QUERY result is given by the storage node to the sink together with some extra information taken by the associated integrity chain (i.e., VO). Thanks to this information, the sink can verify the integrity of the returned result.

Definition 3.5. On a secured database D_s, the following operations are allowed:

- **INSERT**: This operation receives a tuple t and inserts this tuple in the database also by updating the integrity chains and the state of each integrity chain of the database. This means that, for each integrity chain C of D_s, an s-tuple with t as left-most component must be inserted in the bucket with lower bound less than or equal to $val_t(a)$ and upper bound greater than $val_t(a)$ and the state S_C is updated by increasing by one the nop value corresponding to the involved bucket.
- **DELETE**: This operation receives a tuple t and deletes this tuple (if any) from the database also by updating the integrity chains and the state of the database. This means that, for each integrity chain C of D_s, the s-tuple with value t must be eliminated by the chain and a g-tuple $c = \langle (g_i, j) \| d, f_C((g_i, j)\|d)\rangle$ must be inserted, where (1) i is the bucket which t belongs to, (2) j is obtained by increasing by one the right-most element j_i of i-th pair of the state S_C, and d =HMAC$(k, H(t))$. S_C is then updated by increasing by one both nop_i and j_i corresponding to the i-th bucket (i.e., the bucket involved in the deletion).
- **RANGE_QUERY**: This operation receives a range query $Q_a(l, u)$, where l and u represent the bounds of an interval in the domain of the attribute a, on which there exists an integrity chain C in D_s. The aim of this operation is to retrieve the sequence of all the tuples t in D such that $l \leq val_t(a) \leq u$. The result of $Q_a(l, u)$ is a sequence $\bar{C}$ whose elements $\bar{c} = \langle x, y\rangle \in \bar{C}$ are such that: either $\bar{c}$ is either an s-tuple or a g-tuple or a marker, or x, y are digests.In the latter case, we call the element $\bar{c}$ as *digest tuple* (*d-tuple*, for short).

In the next definition, we introduce the notion of query result *verification-proof*, which is based on the satisfaction of four security invariants.

Definition 3.6. A range-query result $\bar{C} = (\bar{c}_1, \ldots, \bar{c}_k)$ is said *verification-proof* for the range query $Q_a(l, u)$ if the following *security invariants* hold:

SI1. There exist three indexes t', y', z' such that:

1. $1 \leq t' < y' \leq z' \leq k$.
2. given the bucket schema B on a, $\bar{c}_1$ is a marker with boundary $b_\alpha = max\{b_i \in B : b_i \leq l\}$.
3. for any $1 < i < t'$, $\bar{c}_i$ is a d-tuple.
4. for any $t' \leq i \leq y'$, $\bar{c}_i$ is not a d-tuple.
5. for any $y' < i < z'$, $\bar{c}_i$ is a d-tuple.
6. for any $z' \leq i < k$, $\bar{c}_i$ is a g-tuple.
7. given the bucket schema B on a, $\bar{c}_k$ is a marker with boundary $b_\beta = min\{b_i \in B : b_i > u\}$.
8. if $t' > 2$, $val(\bar{c}_1) \leq val(\bar{c}_{t'}) < l \leq val(\bar{c}_{t'+1})$
9. for any i, j such that $t' \leq i < j \leq y'$, $val(\bar{c}_i) \leq val(\bar{c}_j)$.
10. for any i, j such that $z' \leq i < j \leq k$, $val(\bar{c}_i) \leq val(\bar{c}_j)$.
11. $val(\bar{c}_{y'-1}) \leq u < val(\bar{c}_{y'}) \leq val(\bar{c}_{z'})$.

If the above conditions hold, we say that $\bar{C}$ is *well-formed* for $Q_a(l, u)$.

SI2. For any $1 \leq i \leq k-1$, $\bar{c}_i = \langle x, y\rangle$, $\bar{c}_{i+1} = \langle x', y'\rangle$, denoting (from now on) by $\bar{H}$ a function such that $\bar{H}(t) = t$ if t is a digest, $\bar{H}(t) = H(t)$ otherwise, it holds that:

$$y = \text{HMAC}(k, \bar{H}(x)\|\bar{H}(x'))$$

SI3. Given two consecutive markers $\bar{c}_i, \bar{c}_j$ $(1 \leq i < j \leq k)$ with boundaries b_v, b_{v+1} respectively, it holds: $nop_v = n_d(v) + n_s(v) + 2n_g(v)$, where $n_d(v), n_s(v), n_g(v)$ denote the number of d-tuples, s-tuples and g-tuples, respectively, occurring in the bucket.

SI4. Given two consecutive markers $\bar{c}_i, \bar{c}_j$ $(1 \leq i < j \leq k)$, denoting by $\bar{D} = \{dgs(\bar{c}_g) : \bar{c}_g$ is a g-tuple and $i < g < j\}$ and $\bar{V} = \{\text{HMAC}(k, \bar{H}(x)) : \bar{c}_s = \langle x, y\rangle$ is a d-tuple or an s-tuple and $i < s < j\}$, it holds that $\bar{D} \cap \bar{V} = \emptyset$.

In other words, SI1 states that $\bar{C}$ is a sequence of buckets (and markers that delimit such buckets) that starts with the marker with the highest boundary less or equal to l and ends with the marker with the lowest boundary greater than u. The first and last bucket may include some d-tuples that replace the s-tuples not involved in the range query but occurring in the bucket. These s-tuples need not be returned entirely but only their digests are enough to compute the HMAC functions and this reduces the number of sink-side hash computations. However,

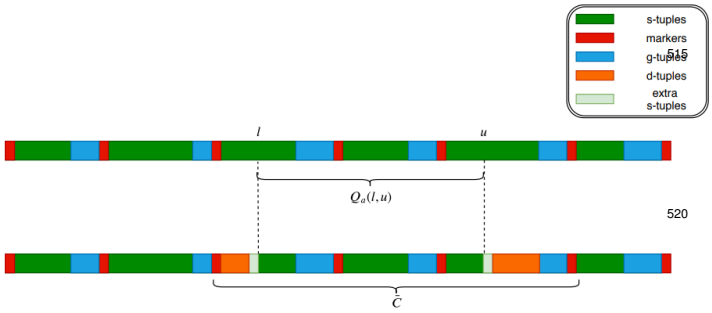

Figure 1: Graphical representation of integrity chain and result of a range query.

if d-tuples occur in the first bucket, then also an *extra* s-tuple must be returned, which corresponds to the tuple with the highest value less than l occurring in D. Similarly, if d-tuples occur in the last bucket, then also an *extra* s-tuple must be returned, which corresponds to the tuple with the lowest value greater than u occurring in D. The elements of $\bar{C}$ are ordered according to their values. For d-tuples, we mean that they are ordered according to the order of the corresponding tuples.

SI2 states that the right-most component of each element of $\bar{C}$ is computed correctly according to Property 2 of C (see Definition 3.2).

Regarding SI3, we recall that nop_v counts the number of INSERT and DELETE operations carried out on the bucket with boundaries (b_v, b_{v+1}). Thus, each s-tuple/d-tuple corresponds to an INSERT and each g-tuple corresponds to a DELETE of an s-tuple previously inserted (so, it has to be counted twice).

Finally, SI4 states that, for each bucket returned by the storage node, no tuple claimed as still occurring in the database (i.e., s-tuples and d-tuples) can appear among deleted tuples (i.e., can correspond to some g-tuple in the bucket).

In Fig. 1, an example of a range query on an integrity chain and the corresponding result are depicted.

As we prove in Section 7, for the sink, to receive a query result verification-proof guarantees that it cannot be compromised in terms of freshness, completeness, and correctness. Hence, to verify query integrity it suffices to check if the query result is verification-proof, that is all the security invariants hold.

The next step is to provide the previous definition in operational terms also by highlighting the messages that the sensors, the storage node, and the sink have to exchange. The three operations are then encoded in form of algorithms. In addition, a SET_UP operation is also defined. Specifically, in Algorithm 1, the initialization of the system is described and in Algorithms 2, 3, and 4 the functions INSERT, DELETE, and RANGE_QUERY are reported. In the following subsections, we provide a description of these algorithms. Therein, we denote by *sn*, *sk*, and *st* the sensor, the sink and the storage node, respectively.

3.1. SET_UP

In this subsection, we describe in detail the SET_UP operation, which is used to initialize the secured database D_s (considered initially empty).

First, *sk* selects the attributes on which range query will be performed. Let denote by n the number of such attributes. As a consequence, the secured database will contain n integrity chains (one for each attribute). For each chain (being creating), *sk* sets a bucket schema $B = \langle b_1, \ldots, b_k \rangle$ and a corresponding ghost schema G and makes public them (Lines 5-8 of Algorithm 1). Finally, for each boundary of B, *sk* builds a sequence of k digests $H = \langle h_1, \ldots, h_k \rangle$ and sends it to *st* (Lines 8-12). Each $h_i \in H \backslash \{h_k\}$ is computed as follows: h_i =HMAC$(k, H(b_i)||H(b_{i+1}))$ (Line 9) and h_k =NULL (Line 10). Finally, *st* retrieves B, H (Lines 17-18) and creates a k-integrity chain $C = (c_1, \ldots, c_k)$ (Line 21) where $c_i = \langle b_i, h_i \rangle$ for $1 \leq i \leq k$. To conclude the SETUP_UP phase, since no tuple has been added or deleted in/from C yet, the state is initialized as follows: $S_C = \langle (nop_1, j_1) \ldots, (nop_{k-1}, j_{k-1}) \rangle = \langle (0,0), \ldots, (0,0) \rangle$ (Line 22).

The proof that C is a *well-formed* chain in the sense of Definition 3.2, is provided in Appendix A.

3.2. INSERT

In this subsection, we describe in detail the INSERT operation, which inserts a tuple t on D_s. First, *sn* sends t to *st* (Line 2 of Algorithm 2). For each chain $C = (c_1, \ldots, c_n)$ with bucket schema B on a given attribute a, state S_C and ghost schema G on B, *st* (1) recovers the consecutive elements $c_i = \langle x, f_C(x) \rangle$ and $c_{i+1} = \langle x', f_C(x') \rangle$ such that $val(c_i) \leq val_t(a) < val(c_{i+1})$, (2) computes h_1=$H(x)$ and h_2=$H(x')$ and (3) sends them to *sn* (Lines 15-17). At this point,*sn* computes h_3=$H(t)$, h_4=HMAC$(k, h_1||h_3)$, h_5=HMAC$(k, h_3||h_2)$ and sends h_4, h_5 to *st* (Lines 5-7). This latter creates a new s-tuple $c = \langle t, h_5 \rangle$ (Line 19), updates the value of $f_C(x)$ to h_4 (Line 20) and creates a new chain $C' = (c'_1, \ldots, c'_{n+1})$ (Line 21) where:

$$c'_z = \begin{cases} c_z & \text{if } 1 \leq z \leq i \\ c & \text{if } z = i+1 \\ c_{z-1} & \text{if } i+1 < z \leq n+1 \end{cases}$$

Finally, it updates C with C' (Line 22).

The proof that C' is a *well-formed* chain in the sense of Definition 3.2, is provided in Appendix B.

To conclude this operation, *sn* finds (Line 8) (b_i, b_{i+1}) such that $b_i \leq val_t(a) \leq b_{i+1}$ (i.e., it finds the bucket where the s-tuple has been inserted). Then, it picks a random value R and sends R, b_i, h^* =HMAC$(k, R||b_i)$ to *sk* (line 9-10). The random is used to avoid reply attacks.

sk checks the correctness of h^* (Line 27) and increments by one the left-most element (nop_i) of the pair $(nop_i, j_i) \in S_C$ (Lines 29-30).

3.3. DELETE

In this subsection we describe in detail the operation DELETE which removes a tuple t on D_s.

First, *sk* sends t (or any unique key able to identify t) to *st* (Line 2 of Algorithm 3). For each chain $C = (c_1, \ldots, c_n)$ with bucket schema B on the attribute a, state S_C and ghost schema G on B, *st* retrieves $c_i = \langle t, f_C(t) \rangle$, $c_{i-1} = \langle x, f_C(x) \rangle$, and $c_{i+1} =$

$\langle x', f_C(x')\rangle$ and computes h_1=$H(x)$, h_2=$H(x')$ (Lines 24-25). At this point, *st* finds (b_v, b_{v+1}) such that $b_v \leq val_t(a) < b_{v+1}$, $c_k = \langle b_{v+1}, f_C(b_{v+1})\rangle$, and $c_{k-1} = \langle \bar{x}, f_C(\bar{x})\rangle$ (Lines 26-27). Then, it computes h_3 and h_4=$H(b_{v+1})$ (Lines 28-32) where:

$$h_3 = \begin{cases} H(\bar{x}) & \text{if } i \neq k-1 \\ \text{NULL} & \text{if } i = k-1 \end{cases}$$

Finally, *st* recovers, from S_C, the pair (nop_v, j_v) associated with (b_v, b_{v+1}) and sends to *sk* $(h_1, h_2, h_3, h_4, f_C(x), f_C(t), j_v)$ (Lines 33-34). At this point, *sk* computes $h^* = H(t)$ (Line 5) and checks that $f_C(x)$ is equal to HMAC$(h_1||h^*)$ and $f_C(t)$ is equal to HMAC$(h^*||h_2)$ (Lines 6-7). Then, *sk* computes h_5=HMAC(k, h^*) (Line 8), retrieves (b_v, b_{v+1}) and the ghost element g_v (Lines 9-10), computes $h_7 = H((g_v, j_v + 1)||h_5)$, h_6, h_8 =HMAC$(k, h_3||h_7)$, h_9 =HMAC$(k, h_7||h_4)$ (Lines 11-16), where:

$$h_6 = \begin{cases} \text{HMAC}(k, h_1||h_2) & \text{if } h_3 \neq \text{NULL } (i \neq k-1) \\ \text{HMAC}(k, h_1||h_7) & \text{otherwise} \end{cases}$$

and sends the digests (h_5, h_6, h_8, h_9) to *st* (Line 17). *st* retrieves g_v (Line 36) from G and creates a new g-tuple $c = \langle (g_v, j_v + 1)||h_5, h_9\rangle$ (Line 37), updates the value of $f_C(x)$ to h_6 and the value of $f_C(\bar{x})$ to h_8 (Line 38), and creates a new chain $C' = (c'_1, \ldots, c'_n)$ (Lines 39-42) where:

$$c'_z = \begin{cases} c_z & \text{if } 1 \leq z < i \vee k \leq z \leq n \\ c_{z+1} & \text{if } i \leq z < k-1 \\ c & \text{if } z = k-1 \end{cases}$$

Finally, it updates C with C' (Line 43) and changes the state S_C by increasing by one the value of j_v (Line 44).

The proof that C' is a *well-formed* chain in the sense of Definition 3.2, is provided in Appendix C.

Once the g-tuple is inserted, *sk* increments by one the left-most element nop_v of the pair $(nop_v, j_v) \in S_C$ (Line 19).

3.4. RANGE_QUERY

In this subsection, we describe in detail the RANGE_QUERY operation, which is used to retrieve the result of a range query $Q_a(l, u)$ and verify its integrity. First, *sk* submits to *st* the range query $Q_a(l, u)$ (Line 2 of Algorithm 4). *st* searches the chain $C = (c_1, \ldots, c_n)$ with bucket schema B on the attribute a and retrieves the elements b_α, b_β, c_t and c_y defined as: $b_\alpha = max\{b_i \in B : b_i \leq l\}$, $b_\beta = min\{b_i \in B : b_i > u\}$, $c_t = \langle b_\alpha, f_C(b_\alpha)\rangle$ and $c_y = \langle b_\beta, f_C(b_\beta)\rangle$ (Lines 8-10). Now, we consider two sets: $T = \{i \in N : t < i < y \wedge val(c_i) < l\}$ and $Y = \{i \in N : t < i < y \wedge val(c_i) > u\}$. *st* computes two indexes (Lines 11-15):

$$t' = \begin{cases} max\{T\} & \text{if } T \neq \emptyset \\ \text{t} & \text{otherwise} \end{cases}$$

$$y' = \begin{cases} min\{Y\} & \text{if } Y \neq \emptyset \\ \text{y} & \text{otherwise} \end{cases}$$

Finally, consider another set $Z = \{i \in N : y' \leq i < y \wedge c_i$ is a g-tuple$\}$. *st* computes the index (Lines 16-18):

$$z' = \begin{cases} min\{Z\} & \text{if } Z \neq \emptyset \\ \text{y} & \text{otherwise} \end{cases}$$

In other words, t' is the index of the element immediately preceding the first element requested by the range query. Similarly, y' is the index of the element immediately following the last element requested by the range query. Finally, z' is the index of the first g-tuple following the last element requested by the range query or it coincides with y.

At this point, *st* builds (Lines 19-22) the range query result $\bar{C} = (\bar{c}_1, \ldots, \bar{c}_{y-t+1})$ where, given the element $c_{t+i-1} = \langle x, f_C(x)\rangle$ such that $1 \leq i \leq y - t + 1$:

$$\bar{c}_i = \begin{cases} \langle H(x), f_C(x)\rangle & \text{if } 1 < i < t' - t + 1 \\ & \vee\ y' - t + 1 < i < z' - t + 1 \\ c_{t+i-1} & \text{otherwise} \end{cases}$$

Finally, $\bar{C}$ is sent to *sk* (Line 24). Now, *sk* invokes (Line 4) VERIFY, which is defined by Algorithm 5 and ensures $\bar{C}$ is verification-proof. This algorithm is based on the finite state automaton depicted in Fig. 2. It only represents the *structural* part of the SI1 invariant (see Definition 3.6), which correspond to conditions (1)–(7) of SI1.

The reading head of the automaton is simulated by the function readNext($\bar{C}$), which returns the next element of $\bar{C}$. We define the state transitions according to the type of element. In Fig. 2, the input symbols are $m_{in}, d, s, g, m, m_{fin}$ and denote that the next element of $\bar{C}$ is the first marker, a d-tuple, an s-tuple,a g-tuple, a marker (different from the first and from the last) and the last marker of $\bar{C}$, respectively. Algorithm 5 implements the automaton of Fig. 2 by enriching its state-transition functions in order to implement also the remaining part of the SI1 invariant (conditions (8)–(11)) and the invariants SI2, SI3, SI4. In detail, Lines 9-93 and Line 99 implement the invariant SI1. Line 97 implements SI2. Finally, Lines 101-112 implement SI3 and SI4. Then, if the algorithm VERIFY does not end in ERROR state, then SI1, SI2, SI3, and SI4 hold.

As it results from the operations described above, we remark that the only non-schema information the sink node has to keep to check query integrity is the set NOP_C (and not all the state S_C) of each integrity chain C. On the other hand, no information has to be maintained by the sensors since the bucket schema can be intentionally generated.

4. High-speed data streams in two-tiered sensor networks

In this section, we contextualize the model introduces in the previous section within the scenario of high-speed data streams in two-tiered sensor networks in which data are produced by constrained devices. Specifically, we show how data streams

Algorithm 1 SET_UP

procedure SINK NODE
  $n \leftarrow$ set the number of chains of D_s
  sendToStorageNode(n)
  for n times **do**
    set a bucket schema $B \leftarrow \langle b_1, b_2, \ldots, b_k \rangle$
    set a ghost schema $G \leftarrow \langle g_1, g_2, \ldots, g_{k-1} \rangle$ associated with B
    publish(B, G)
    for $1 \leq i \leq k-1$ **do**
      compute $h_i \leftarrow$HMAC($k, H(b_i) \| H(b_{i+1})$)
    $h_k \leftarrow$NULL
    $H \leftarrow \langle h_1, h_2, \ldots, h_k \rangle$
    sendToStorageNode(H)
  close
procedure STORAGE NODE
  $n \leftarrow$ receiveFromSinkNode()
  for n times **do**
    retrieve $B = \langle b_1, b_2, \ldots, b_k \rangle$
    $H = \langle h_1, h_2, \ldots, h_k \rangle \leftarrow$ receiveFromSinkNode()
    for all $1 \leq i \leq k$ **do**
      create the element $c_i \leftarrow \langle b_i, h_i \rangle$
    create the chain $C \leftarrow (c_1, c_2, \ldots, c_k)$
    create the state $S_C = \langle (nop_1, j_1), \ldots, (nop_{k-1}, j_{k-1}) \rangle \leftarrow \langle (0,0), \ldots, (0,0) \rangle$
  close

Algorithm 2 INSERT

procedure SENSOR
  sendToStorageNode(t)
  for all chain $C = (c_1, \ldots, c_n)$ with bucket schema B on a given attribute a, state S_C and ghost schema G on B **do**
    $h_1, h_2 \leftarrow$ receiveFromStorageNode()
    compute $h_3 \leftarrow H(t)$
    compute $h_4 \leftarrow$HMAC($k, h_1 \| h_3$) and $h_5 \leftarrow$ HMAC($k, h_3 \| h_2$)
    sendToStorageNode(h_4, h_5)
    retrieve, from B, (b_i, b_{i+1}) such that $b_i \leq val_t(a) < b_{i+1}$.
    Pick a random R and compute h^* =HMAC($k, R \| b_i$)
    sendToStorageNode(R, b_i, h^*)
  close
procedure STORAGE NODE
  $t \leftarrow$ receiveFromSensor()
  for all chain $C = (c_1, \ldots, c_n)$ with bucket schema B on a given attribute a and ghost schema G on B **do**
    retrieve, from C, $c_i = \langle x, f_C(x) \rangle$ and $c_{i+1} = \langle x', f_C(x') \rangle$ such that $val(c_i) \leq val_t(a) < val(c_{i+1})$
    compute $h_1 \leftarrow H(x)$ and $h_2 \leftarrow H(x')$
    sendToSensor(h_1, h_2)
    $(h_4, h_5) \leftarrow$ receiveFromSensor()
    create a new s-tuple $c \leftarrow \langle t, h_5 \rangle$
    update $f_C(x) \leftarrow h_4$
    create $C' = (c'_1, \ldots, c'_{n+1}) \leftarrow (c_1, \ldots, c_i, c, c_{i+1}, \ldots, c_n)$
    update $C \leftarrow C'$
  close
procedure SINK NODE
  for all chain $C = (c_1, \ldots, c_n)$ with bucket schema B on a given attribute a and ghost schema G on B **do**
    $R, b_i, h^* \leftarrow$ receiveFromSensor()
    if (HMAC($k, R \| b_i$) $\neq h^*$) **then**
      return ERROR
    retrieve, from NOP_C, nop_i
    update $nop_i \leftarrow nop_i + 1$
  close

Algorithm 3 DELETE

procedure SINK NODE
  sendToStorageNode(t)
  for all chain $C = (c_1, \ldots, c_n)$ with bucket schema B on a given attribute a, state S_C and ghost schema G on B **do**
    $h_1, h_2, h_3, h_4, f_C(x), f_C(t), j_v \leftarrow$ receiveFromStorageNode()
    compute $h^* \leftarrow H(t)$
    if (HMAC$(k, h_1 \| h^*) \neq f_C(x) \vee$ HMAC$(k, h^* \| h_2) \neq f_C(t)$) **then**
      return ERROR
    compute $h_5 \leftarrow$HMAC(k, h^*)
    retrieve, from B, (b_v, b_{v+1}) such that $b_v \leq val_t(a) < b_{v+1}$.
    retrieve, from G, the ghost g_v
    compute $h_7 \leftarrow H((g_v, j_v + 1) \| h_5)$
    if $h_3 \neq$NULL **then**
      compute $h_6 \leftarrow$ HMAC$(k, h_1 \| h_2)$.
    else
      compute $h_6 \leftarrow$ HMAC$(k, h_1 \| h_7)$.
    compute $h_8 \leftarrow$HMAC$(k, h_3 \| h_7)$, $h_9 \leftarrow$HMAC$(k, h_7 \| h_4)$
    sendToStorageNode(h_5, h_6, h_8, h_9)
    retrieve, from NOP_C, nop_v
    update $nop_v \leftarrow nop_v + 1$
  close
procedure STORAGE NODE
  $t \leftarrow$receiveFromSinkNode()
  for all chain $C = (c_1, \ldots, c_n)$ with bucket schema B on a given attribute a, state S_C and ghost schema G on B **do**
    retrieve, from C, $c_i = \langle t, f_C(t) \rangle$, $c_{i-1} = \langle x, f_C(x) \rangle$ and $c_{i+1} = \langle x', f_C(x') \rangle$.
    compute $h_1 \leftarrow H(x)$ and $h_2 \leftarrow H(x')$
    retrieve, from B, (b_v, b_{v+1}) such that $b_v \leq val_t(a) < b_{v+1}$.
    retrieve, from C, $c_k = \langle b_{v+1}, f_C(b_{v+1}) \rangle$ and $c_{k-1} = \langle \bar{x}, f_C(\bar{x}) \rangle$.
    if $i \neq k - 1$ **then**
      compute $h_3 \leftarrow H(\bar{x})$
    else
      $h_3 \leftarrow$ NULL
    compute $h_4 \leftarrow H(b_{v+1})$
    retrieve, from S_C, the pair (nop_v, j_v) associated with (b_v, b_{v+1}).
    sendToSinkNode($h_1, h_2, h_3, h_4, f_C(x), f_C(t), j_v$)
    $(h_5, h_6, h_8, h_9) \leftarrow$ receiveFromSinkNode()
    retrieve, from G, the ghost g_v
    create a new g-tuple $c \leftarrow \langle (g_v, j_v + 1) \| h_5, h_9 \rangle$
    update $f_C(x) \leftarrow h_6$, $f_C(\bar{x}) \leftarrow h_8$
    if $i \neq k - 1$ **then**
      create $C' = (c'_1, \ldots, c'_n) \leftarrow (c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_{k-1}, c, c_k, \ldots c_n)$
    else
      create $C' = (c'_1, \ldots, c'_n) \leftarrow (c_1, \ldots, c_{i-1}, c, c_{i+1}, \ldots c_n)$
    update $C \leftarrow C'$
    update $j_v \leftarrow j_v + 1$
  close

Algorithm 4 RANGE_QUERY

procedure SINK NODE
  sendToStorageNode($Q_a(l, u)$)
  $\bar{C} = (\bar{c}_1, \ldots, \bar{c}_k) \leftarrow$ receiveFromStorageNode()
  VERIFY($\bar{C}, Q_a(l, u)$)
  close
procedure STORAGE NODE
  $Q_a(l, u) \leftarrow$ receiveFromSinkNode()
  retrieve the chain $C = (c_1, \ldots, c_n)$ with bucket schema B on the attribute a
  retrieve, from B, (b_α, b_β) such that $b_\alpha = max\{b_i \in B : b_i \leq l\}$ and $b_\beta = min\{b_i \in B : b_i > u\}$
  retrieve, from C, $c_t = \langle b_\alpha, f_C(b_\alpha)\rangle$ and $c_y = \langle b_\beta, f_C(b_\beta)\rangle$
  $t' \leftarrow t$ $y' \leftarrow y$
  if $T = \{i \in N : t < i < y \wedge val(c_i) < l\} \neq \emptyset$ **then**
    $t' \leftarrow max\{T\}$
  if $Y = \{i \in N : t < i < y \wedge val(c_i) > u\} \neq \emptyset$ **then**
    $y' \leftarrow min\{Y\}$
  $z' \leftarrow y$
  if $Z = \{i \in N : y' < i < y \wedge c_i$ is a g-tuple $\} \neq \emptyset$ **then**
    $z' \leftarrow min\{Z\}$
  for $1 \leq i \leq y - t + 1$ **do**
    create the element $\bar{c}_i \leftarrow c_{t+i-1} = \langle x, f_C(x)\rangle$
    if $(1 < i < t' - t + 1) \vee (y' - t + 1 < i < z' - t + 1)$ **then**
      $\bar{c}_i \leftarrow \langle H(x), f_C(x)\rangle$
  $\bar{C} \leftarrow (\bar{c}_1, \ldots, \bar{c}_{y-t+1})$
  sendToStorageNode($\bar{C}$)
  close

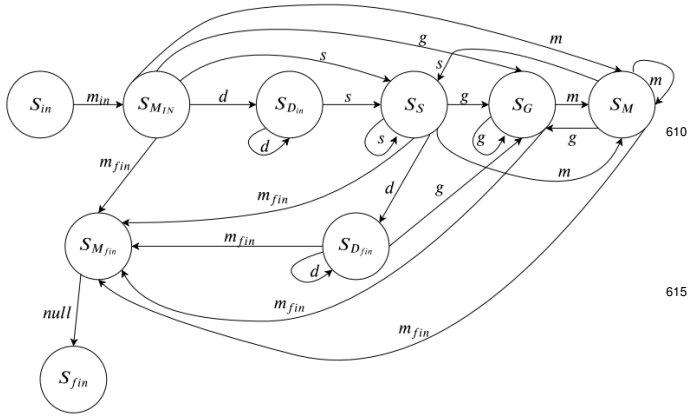

Figure 2: Finite state automaton underlying the verification of the query result procedure.

can be managed, as well as the integrity-preserving update operations. Consequently, we describe how range queries are performed on data streams so organized and how query integrity can be verified.

We consider several sensors that capture data from the surrounding environment. Those data, together with other information (e.g., a timestamp, the *id* of the sensor, etc.) are sent to the *storage node* to be stored. Each data can be thought as a tuple with several attributes, including the attribute *time*. For a given attribute (also different from the attribute time), the *sink node* can submit a range query to the storage node asking for all data whose values are within a given interval of this attribute. We are interested in guaranteeing *query integrity*, so that the result of the query returned by the storage node is correct, complete, and fresh.

We consider the case of low-powered sensors that collect and send data with very high frequency. As data must be secured at the source (thus, by constrained devices), the efficiency of the operations required to update the verification object becomes critical. On the contrary, the verification task is less critical since it is performed by the sink that owns much more computational power than the sensors. The same considerations of the sink apply for the storage node [10].

As often done in the field of data streams, we refer to a *sliding-window*-based approach, in which new data are added and the oldest are removed.

Now, we show how the model described in Section 3 is instantiated to this scenario.

Let $A = (a_1, \ldots, a_t)$ be the attributes of the tuples captured by the sensors. Let denote by a the attribute that represents the insertion timestamp of a tuple, and by D the data stream of tuples produced by the sensors. From now on, consider given a secured database D_s (built on D) containing an integrity chain C on the attribute a, with bucket schema $B = (b_1, \ldots, b_k)$ of cardinality k and ghost schema G. D_s may contain other integrity chains in addition to C, one for each attribute on which range queries are performed by the sink node.

Definition 4.1. Given a bucket schema $B = (b_1, \ldots, b_k)$ and a positive number s, a *sliding window* W (of cardinality w) *with shift parameter* s *on* B is a subsequence of B composed of all the w boundaries of B belonging to the interval $[b_s, b_{s+w-1}]$ such that: $1 \leq s < s + w - 1 \leq k$ (i.e., W is a *substring* of B).

From now on, we consider given a sliding window W on B (where, recall, B is the bucket schema of C) with shift parameter s.

In other words, the sliding window is updated when a new insertion saturates a bucket and a new bucket must be included

Algorithm 5 VERIFY

input: $\bar{C}, Q_a(l,u)$
retrieve, from B (on a), (b_α, b_β) such that $b_\alpha = max\{b_i \in B : b_i \leq l\}$ and $b_\beta = min\{b_i \in B : b_i > u\}$
$\bar{V} \leftarrow \emptyset$
$n\bar{o}p \leftarrow 0$
$S \leftarrow S_{IN}$
$c = \langle x', y' \rangle \leftarrow$ readNext($\bar{C}$)
$c_p = \langle x, y \rangle \leftarrow null$
val_1
if ($C.length < 2$) **then**
    $S \leftarrow S_{ERR}$
while ($S \neq S_{ERR} \wedge c \neq null$) **do**
    switch(S)
    case S_{IN}
    if (c is a marker and $val(c) = b_\alpha$) **then**
        $S \leftarrow S_{M_{in}}$
    else
        $S \leftarrow S_{ERR}$
    break
    case $S_{M_{in}}$
    if (c is a d-tuple) **then**
        $S \leftarrow S_{D_{in}}$
    if (c is an s-tuple $\wedge$ $val(c) \geq l$) **then**
        $S \leftarrow S_S$
    if (c is a g-tuple) **then**
        $S \leftarrow S_G$
    if (c is a marker and $val(c) = b_\beta$) **then**
        $S \leftarrow S_{M_{fin}}$
    if (c is a marker) **then**
        $S \leftarrow S_M$
    else
        $S \leftarrow S_{ERR}$
    break
    case $S_{D_{in}}$
    if (c is a d-tuple) **then**
        $S \leftarrow S_{D_{in}}$
    if (c is an s-tuple $\wedge$ $val(c) < l \wedge b_\alpha \leq val(c)$) **then**
        $S \leftarrow S_S$
    else
        $S \leftarrow S_{ERR}$
    break
    case S_S
    if (c is a d-tuple $\wedge val(c_p) > u$) **then**
        $val_1 \leftarrow val(c_p)$
        $S \leftarrow S_{D_{fin}}$
    if (c is an s-tuple $\wedge$ $val(c_p) \leq u$) **then**
        $S \leftarrow S_S$
    if (c is a g-tuple) **then**
        $S \leftarrow S_G$
    if (c is a marker and $val(c) = b_\beta$) **then**
        $S \leftarrow S_{M_{fin}}$
    if (c is a marker) **then**
        $S \leftarrow S_M$
    else
        $S \leftarrow S_{ERR}$
    break
    case S_G
    if (c is a g-tuple) **then**
        $S \leftarrow S_G$
    if (c is a marker and $val(c) = b_\beta$) **then**
        $S \leftarrow S_{M_{fin}}$
    if (c is a marker) **then**
        $S \leftarrow S_M$
    else
        $S \leftarrow S_{ERR}$
    break
    case $S_{M_{fin}}$
    if ($c = null$) **then**
        $S \leftarrow S_{FIN}$
    else
        $S \leftarrow S_{ERR}$
    break
    case $S_{D_{fin}}$
    if (c is a d-tuple) **then**
        $S \leftarrow S_{D_{fin}}$
    if (c is a g-tuple $\wedge val_1 < val(c)$) **then**
        $S \leftarrow S_G$
    if (c is a marker $\wedge val(c) = b_\beta \wedge val_1 < b_\beta$) **then**
        $S \leftarrow S_{M_{fin}}$
    else
        $S \leftarrow S_{ERR}$
    break
    $c_p \leftarrow c$
    $c \leftarrow$ readNext(C)
    if ($c \neq$ NULL) **then**
        if ($y \neq$HMAC($\bar{H}(x) || \bar{H}(x')$)) **then**
            $S \leftarrow S_{ERR}$
        if (c_p and c are not d-tuples $\wedge$ $val(c_p) > val(c)$) **then**
            $S \leftarrow S_{ERR}$
        if (c is an s-tuple $\vee$ c is a d-tuple) **then**
            $\bar{V}.add$(HMAC($k, \bar{H}(x')$))
            $n\bar{o}p \leftarrow n\bar{o}p + 1$
        if (c is a g-tuple) **then**
            $n\bar{o}p \leftarrow n\bar{o}p + 2$
            if ($dgs(c) \in \bar{V}$) **then**
                $S \leftarrow S_{ERR}$
        if (c is a marker) **then**
            retrieve the index γ such that $b_\gamma = val(c) \in B$
            retrieve, from NOP_C, nop_γ
            if $n\bar{o}p \neq nop_\gamma$ **then**
                $S \leftarrow S_{ERR}$
            $n\bar{o}p \leftarrow 0$
            $\bar{V} \leftarrow \emptyset$

into the sliding window by discarding the oldest one. This is done by increasing the shift parameter. The above mechanism corresponds to the insertion of new tuples done by the sensors. It could appear that the delete operation is not necessary, because the only removed tuples are those moving out from the sliding window. However, according to Definition 3.5, the presence of non-temporal integrity chains results in deleting tuples internal to buckets. This corresponds to a suitable re-definition of the INSERT operation given in Definition 3.5, as follows.

Definition 4.2. *INSERT_SW* This operation receives a tuple t and inserts this tuple into the database D_s as explained in the INSERT procedure of Definition 3.5. If $val_t(a)$ is greater than the greatest boundary contained in W, the sliding window shifts forward opportunely by updating the shift parameter s and all the tuples preceding the new smallest boundary of W are deleted, by calling the DELETE operation of Definition 3.5.

The pseudo-code of INSERT_SW is reported in Algorithm 6. First, the sensor invokes the INSERT method by passing as a parameter the tuple t (Line 3). In the chain C, which is ordered according to the insertion time of t, the tuple is appended. For any other chain, the position of the tuple depends on the value of the corresponding attribute. As effect of the INSERT operation the sink receives (Line 7) from the sensor the boundary b_i corresponding to the greatest boundary smaller than $val_t(a)$. Then it sets a new index $s^* = i + 2 - w$ (Line 8). Now, if $s \neq s^*$ then the sliding window has to shift forward, otherwise no further operation is required (Line 9). Suppose $s \neq s^*$, the sink finds the boundaries (b_s, b_{s^*}) and obtains from the storage node all the tuples t^* such that $b_s \leq val_{t^*}(a) < b_{s^*}$ by checking their integrity (Line 11). All these tuples are removed from D_s (Line 13) Finally, the shift parameter of W is updated from s to s^* (Line 14). This way, the sliding window shifts forward, that is, its first boundary becomes b_{s^*} and all the tuples which values precede b_{s^*} are removed from D_s.

To conclude this section, observe that the INSERT_SW operation can require a significant number of DELETE operations performed by the sink. Even though the sink has no particular issues from a computational point of view, our approach guarantees also very efficient DELETE operations.

5. Cost analysis

In this section, we provide a detailed analysis of the performance of the operations supported by our data structure. Specifically, we want to determine the computational costs in terms of number of *elementary* hash function applications that have to be computed to perform the operations defined above. For elementary application of the hash function H we mean the application of the function on a message with size not greater than the size of blocks on which the function is defined (according to the Merkle-Damgard scheme). From now on, we denote by l the size (in bits) of such block.

Definition 5.1. Given a hash function H, we define as *unitary cost (uc)*, the computational cost of H over a message of l bits.

Clearly, the computational cost of the application of H on a message of size x, is $C(x) = \frac{x}{l}\ uc$. From now on, uc is considered as default unitary cost, thus omitted.

In the following analysis, we do not consider the cost of concatenation, assignment, padding, and XOR, because they are negligible compared to the hash computations.

Moreover, we do not consider the hash computations performed by the storage node since it is not resource-constrained. Finally, to obtain a fair comparison, we assume that all hash computations are performed by using the same hash function H. Thus, H is also the hash function underlying HMAC.

5.1. Cost of HMAC

All operations defined above need the computation of the HMAC function. Now, we express the cost of HMAC in terms of uc.

Throughout this section, consider given a message M of size m and a hash function H (based on Merkle-Damgard scheme) that operates by dividing M into blocks of l bits and returns a digest of p bits. The definition of HMAC is:

$$HMAC(K, M) = H\left((K \oplus opad) || H((K \oplus ipad) || M)\right)$$

where: $ipad$= the byte 0x36 repeated $\frac{l}{8}$ times (l bits), $opad$= the byte 0x5c repeated $\frac{l}{8}$ times (l bits) and K is a secret key of arbitrary size $|K|$. We can distinguish two cases:

1. if $|K| \leq l$, then we add zero-padding to $|K| = l$;
2. if $|K| > l$, them we compute $H(K) = K'$ (the output is a string with size $t < l$) and then we add zero-padding again.

Since K is arbitrary, we assume that $|K| = l$, so that we can neglect the costs of the above padding operation. Moreover, as suggested by [39], it is possible to compute $K \oplus opad$ and $K \oplus ipad$ only once. Then, these values can be stored and used when required. Thus, we can neglect also the cost of the XOR operation.

We show that the cost of HMAC is: $C_{HMAC} = 3 + \frac{m}{l}$.

First, we evaluate the cost of $H((K \oplus ipad)||M)$. As $(K \oplus ipad)$ returns a block of size l, $((K \oplus ipad)||M)$ returns a block of size $l + m$. Therefore, $H((K \oplus ipad)||M)$ costs $\frac{l+m}{l} = 1 + \frac{m}{l}$. At this point, we can evaluate the cost of HMAC by considering that the hash function H operates on $(K \oplus opad)||H((K \oplus ipad)||M)$, which has size $l + p$. We can assume that, as it happens for common hash functions, $p < l$. Therefore, the message of size $l + p$ is padded to the size $2l$. Consequently, the overall cost is: $\frac{2l}{l} + 1 + \frac{m}{l} = 3 + \frac{m}{l}$.

5.2. Cost of operations

Now, we can determine the computational costs of the operations: INSERT, DELETE, RANGE_QUERY, INSERT_SW presented in Sections 3 and 4.

For simplicity, w.l.o.g., we consider a secured database D_s with only one integrity chain. We refer to Algorithms 2, 3, 4, 6. We denote by x the size of a generic tuple (assuming that such a size is the same for all tuples).

Moreover, we denote by p the size of the digests of H and assume to use a hash function for which $2p = l$ (e.g., SHA256).

Algorithm 6 INSERT_SW

```
1: procedure SENSOR
2:     Given a secured database D_s containing at least an integrity chain C on a with bucket schema B and a sliding window W with shift parameter s on B.
3:     INSERT of t in D_s
4:     close
5: procedure SINK NODE
6:     Given a secured database D_s containing at least an integrity chain C on a with bucket schema B and a sliding window W with shift parameter s on B.
7:     b_i ←receiveFromSensor()
8:     set s* ← i + 2 − w
9:     if s ≠ s* then
10:        retrieve, from B, (b_s, b_s*)
11:        C̄ ← RANGE_QUERY(Q_a(b_s, b_s*))
12:        for all s-tuple c̄_i = ⟨t, f_C(t)⟩ do
13:            DELETE t from D_s
14:        update s ← s*
15:    close
```

Finally, we assume that the size of $(g, j)\|d$, where g is a g-value and d is a digest, and boundaries is l (this assumption is realistic by considering the value l of common hash functions).

- INSERT: The only hash computations performed by the sensor are h_3, h_4, h_5 (see Algorithm 2, Lines 5-6). The computation of $H(t)$ can be performed with cost $\frac{x}{l}$. Instead, h_4, h_5 are computed by applying the HMAC function on the concatenation between two digests ($h_1\|h_3$ for h_4 and $h_3\|h_2$ for h_5, respectively). Thus, the input of the HMAC function has size $l = p + p$ and the cost of each HMAC is $3 + \frac{l}{l} = 4$. The total cost is then:

$$C_{INS} = \frac{x}{l} + 2 \cdot 4 = \frac{x}{l} + 8$$

- DELETE: We refer now to Algorithm 3. The reasoning is similar to the case of INSERT. First, the sink computes $h^* = H(t)$ (Line 5) with cost $\frac{x}{l}$. Then, to perform the check at Line 6, it applies, two times, the HMAC function with input of size $l = 2p$ and the cost is $4 \cdot 2 = 8$. At this point, it has to compute h_5, h_6, h_7, h_8, h_9. Regarding h_5, it is obtained by applying the HMAC function on h^* (Line 8). The input of HMAC has size p bits, but it is padded to the size of l bits, so, again, the cost of HMAC is 4. Regarding h_6, h_8, h_9 (Lines 12-16), as for the case of INSERT, HMAC is applied on the concatenation of two digests and the cost is 4. Finally, for h_7 (Line 11), the input of hash function H is $l = 2p$ and the cost is $\frac{l}{l} = 1$. The total cost is then:

$$C_{DEL} = \frac{x}{l} + 8 + 4 \cdot 4 + 1 = \frac{x}{l} + 25$$

- RANGE_QUERY: For this operation, the evaluation of the computational costs is a bit more complicated than the previous. In fact, it depends on the number of s-tuples, d-tuples, markers, and g-tuples involved in the range query. Given the sequence $\bar{C}$ (Algorithm 4, Line 3), we denote by $N(s)$, $N(m)$, $N(g)$, $N(d)$ the number of s-tuples, markers, g-tuples, and d-tuples, respectively, occurring in $\bar{C}$. We highlight that $N(s)$ are the actual tuples (except for at most two extra tuples) that the sink requires while the others $N(d) + N(m) + N(g)$ elements are additional information used to verify query integrity (i.e., VO). For each element of $\bar{C}$, the sink computes (at Line 97 of Algorithm 5) $\bar{H}(x)$ and $\bar{H}(x')$ and applies the function HMAC. However, $\bar{H}(x')$ can be used for the next iteration. Therefore, the sink applies the $\bar{H}$ function only once for each element $c \in \bar{C}$. The cost of $\bar{H}$ varies according to the type of the element c. Specifically, if c is a d-tuple, no hash computation is requested. If c is a marker or a g-tuple, the function $\bar{H}$ coincides with the hash function H and the input has a size equal to l, thus, the contribution to the total cost is $N(m) + N(g)$. Otherwise, if c is an s-tuple, again the function $\bar{H}$ coincides with H and the cost of H is $\frac{x}{l}$ per s-tuple. Thus, the overall contribution is $N(s)\frac{x}{l}$. Now, the for each elements of $\bar{C}$, except from the last, at Line 97, the sink computes an HMAC with input of size $2p = l$ with total cost of $4(N(s) + N(m) + N(g) + N(d) - 1)$. Finally, another HMAC is applied, for each s-tuple and d-tuple of $\bar{C}$ (Line 101), with input $p < l$. The cost of these HMAC is, by considering the padding operation to l bits as above, $4(N(s) + N(d))$. The total cost is then:

$$\begin{aligned} C_{RQ} = \; & N(s)\tfrac{x}{l} + (N(m) + N(g)) + 4(N(s) + N(d)) \\ & + 4(N(s) + N(d) + N(m) + N(g) - 1) \end{aligned} \tag{1}$$

We remark that in the proposed scenario, the operation RANGE_QUERY operation is performed by the sink that is not resource-constrained, thus it is not critical in our analysis.

- INSERT_SW: The sensors pay the cost of one INSERT and the cost of D DELETEs, where D is the number of tuples falling outside the sliding window.

$$C_{SW} = \frac{x}{l} + 8 + D\left(\frac{x}{l} + 25\right)$$

6. Performance Comparison

In this section, we perform a comparative analysis between our proposal and the Merkle-Hash-Tree-based solution that represents the state of the art. Even though there are different (even

recent) variants of MHT-based solutions, we refer to the basic one, by considering that the efficiency improvement of our approach is asymptotic. As we will see in the sequel of the section, we pass from $O(log(n))$ cost of *any* tree-based approach for insertion and deletion to $O(1)$ (where n is the size of the stored data), with no asymptotic price in terms of verification cost.

We remark that this is the core contribution of our approach aimed to *break down* the cost of insertion/deletion in those (relevant) cases in which their security must be guaranteed at the source (and thus, possibly, by constrained devices). Therefore, a little price in terms of exact operations (not asymptotic) of query verification can be tolerated, according to our goal, as verification is always performed by the sink that is not resource-constrained. The subsequent section of experiments aims to confirm analytical results by also studying the comparison when parameters change.

6.1. Merkle Hash Tree

In this section, we recall how a classical Merkle Hash Tree [15] works to guarantee query integrity. Data are divided into blocks (in our model the blocks coincide with the tuples of the database ordered according to their value). A Merkle Hash Tree (MHT) is a binary tree in which the leaves are the digests of these tuples. The rest of the nodes (internal nodes) are obtained by applying the hash function H on the concatenation of their two children. For example, in Fig. 3 the leaf associated with x_1 is $H_1 = H(x_1)$, the leaf associated with x_2 is $H_2 = H(x_2)$ and the internal node $H_{12} = H(H_1||H_2)$.

This data structure may be used to verify query integrity as follow. First, the root is digitally signed and publicly available (or it may be transferred through a trusted channel). Suppose now that the sink submits a query and receives, from the storage node, a set of tuples that have to be authenticated. For this purpose, the storage node has to return the tuples involved in the query, the two boundary tuples not included in the query, and a set of digests necessary to re-calculate the path from the leaves to the root. The two boundary tuples, extra s-tuples for our algorithm, are necessary to avoid that the storage node omits the first or the last tuples involved in the query. By using the digests, the sink is able to compute the root of the tree and to compare it with the trusted public root. If they match, the integrity of the query is verified. For example, by considering again Fig. 3, if we submit a query whose result is $\{x_4, x_5, x_6\}$ (in red), the storage node returns also the tuples $\{x_3, x_7\}$ (in green) and the set of digests $\{H_8, H_{12}\}$ (in blue). In fact, the root may be obtained with the following steps:

1. H_3=$H(x_3)$, H_4=$H(x_4)$, H_5=$H(x_5)$, H_6=$H(x_6)$, H_7=$H(x_7)$
2. H_{34}=$H(H_3||H_4)$, H_{56}=$H(H_5||H_6)$, H_{78}=$H(H_7||H_8)$
3. H_{14}=$H(H_{12}||H_{34})$, H_{58}=$H(H_{56}||H_{78})$
4. H_{root}=$H(H_{14}||H_{58})$

Now, we evaluate the computational cost of the INSERT, DELETE, and RANGE_QUERY operations, when an MHT is adopted. We denote by F the total number of leaves of the tree.

Independently of the version of MHT used, the insertion of a tuple of size x requires to compute its hash value (with cost $\frac{x}{l}$) to generate a new leaf of the tree and re-compute the digests that constitute the path from the leaf to the root (recall that l is the size of the elementary blocks on which H works). Since there are F leaves, the total cost is:

$$\bar{C}_{INS} = \frac{x}{l} + O(log(F))$$

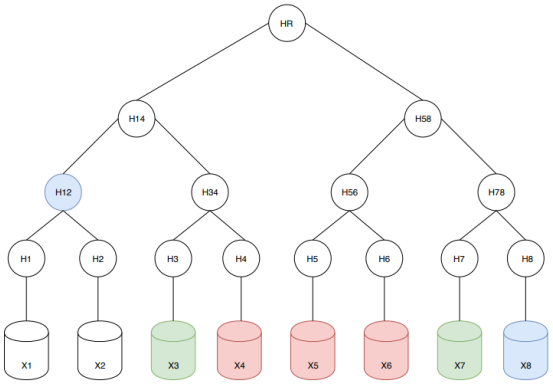

Figure 3: Range query in a Merkle Hash Tree (x_4, x_6).

For the deletion, instead of re-computing entirely the tree, when the sink removes a tuple, it substitutes such a tuple with a *dummy* tuple (which is similar to a g-tuple). Thus, F includes *regular* (i.e., not dummy) and dummy tuples. Now, if we assume that the size of a dummy tuple is equal to l, the sink applies the hash function H with unitary cost. Then, it re-computes the path to the root. The total cost is:

$$\bar{C}_{DEL} = 1 + O(log(F))$$

By comparing MHT to our approach, the number of dummy tuples is equal to the number of g-tuples since, for each deletion, a g-tuple is generated. Moreover, the regular tuples in the MHT correspond to the s-tuples in the corresponding integrity chain. Therefore, F represents also the total number of s-tuples and g-tuples occurring in the integrity chain.

Finally, regarding the verification procedure, the storage node returns a set of tuples that includes regular and dummy tuples. By using the notation of the previous section, the number of regular tuples is $N(s)$ (number of s-tuples actually requested by the sink) and the number of dummy tuples is $N(g)$ (number of g-tuples occurring in the result of the range query). Then, the sink computes H for each of these tuples. The cost of this computation is $\frac{x}{l}$ per regular tuple and 1 per dummy tuple (as discussed earlier). Finally, the digests resulting from the computation of H are concatenated as pairs and the function H is applied to these pairs. The input of H has size $l = p + p$. The hash function H is applied at least once per level. Thus, being $log(F)$ the number of levels of the tree, we obtain:

$$\tilde{C}_{RQ} = N(s)\frac{x}{l} + N(g) + O(log(F))$$

In conclusion, we note that for MHT-based approaches, the costs of INSERT, DELETE (and consequently INSERT_SW) increase as $log(F)$, while our approach is constant with respect to F and depends only on the size of the tuples.

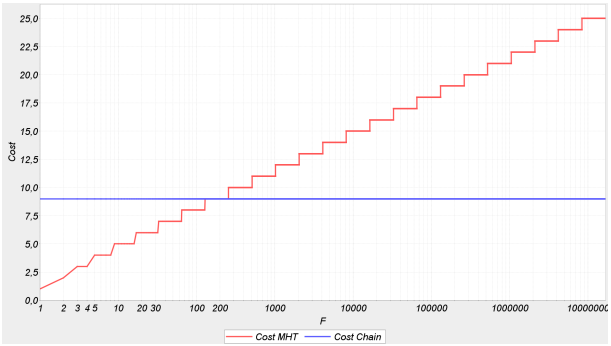

Figure 4: Insertion cost with x=64 bytes.

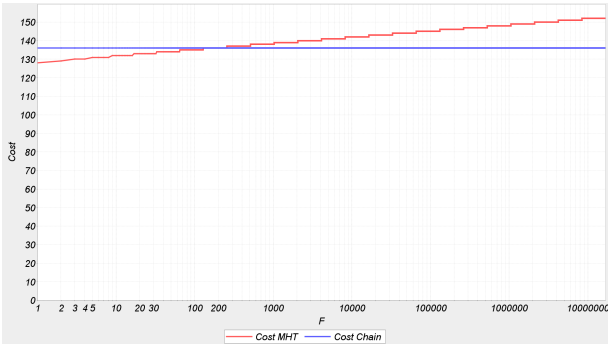

Figure 5: Insertion cost with x=8192 bytes.

Concerning verification, by considering range queries with size of the order of the size of buckets (coherently with our setting, we are not interested in punctual queries), on the basis of the above expression and the expression (1) given in Section 5, it is easy to see that, asymptotically, the verification is $O(N(s))$ for both techniques, as $N(m)$, $N(d)$ and $N(g)$ are all in $O(N(s))$.

6.2. Experiments

In order to validate our proposal, we perform an experimental analysis on synthetic data structures to evaluate insertion costs. In detail, we implemented a basic version of MHT (described in the previous subsection) and an integrity chain. Then, we study how the insertion cost of a tuple varies versus the total number F of elements (i.e., regular tuples plus dummy tuples or s-tuples plus g-tuples). We note that, for the integrity chain, the insertion cost is independent of the number of s-tuples, g-tuples, and markers. Similarly, in MHT, the presence of dummy tuples in F does not affect the insertion cost. Thus, we consider an MHT with F regular tuples and no dummy tuple and a chain with F s-tuples and no g-tuple (the markers occur but do not affect the cost). Finally, we evaluate the costs for different values of the size x of tuples: 64, 512, 2048, and 8192 bytes. Due to space limitations, we show only the results obtained for x=64 bytes and x=8192 bytes in Figures 4 and 5.

Clearly, the costs of insertion increase as the size of the tuples x increases for both the data structures. However, our approach overcomes the MHT approach for each value of x (64-512-2048-8192 bytes) and any plausible value of F. According to the results given in Sections 5 and 6, our approach has a constant cost, while MHT varies as $log(F)$.

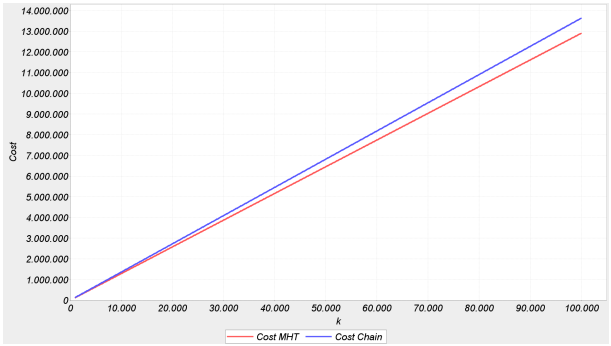

Figure 6: Verification cost of a range query of size k

Just to show that, apart from the asymptotic analysis given in the previous section, also exact costs do not penalize our solution, we perform a comparative analysis on the performance of the range query verification. We consider tuples with size x=8KB. For MHT, we consider 2^{24} regular tuples and 2^{20} dummy tuples. Similarly, for the integrity chain, we have 2^{24} s-tuples, 2^{20} g-tuples, and 2^{16} + 1 markers. Besides, we assume that the buckets of the chain contain the same number of s-tuples z = 256 and the same number of g-tuples n_g = 16. Thus, also in MHT, we assume there are on average 16 dummy tuples every 256 regular tuples.

This experiment evaluates the verification costs on range queries of size k, with k ranging from 1000 to 100000. As this cost depends on the start position of the query, we repeat the same query 10 times in different positions and calculate the mean value. The result of this experiment is reported in Fig. 6 and shows that MHT performs slightly better, but the performances of the two solutions are basically comparable.

7. Security analysis

The purpose of this section is to provide a security analysis of the model proposed in Section 3. The considered actors are the sensor, the sink, and the storage nodes. As in [8] and [10], in our adversary model, the sensor and the sink shares the secret key k used to compute the HMAC function, and the attacker is the storage node, for which we do not make any assumption about its trustworthiness. We observe that to guarantee query integrity does not mean that the storage node is not able to store wrong versions of the database, but that any anomalous portion of the database is detected by the sink node when required.

The basic *assumptions* we make are:

A1: the bucket schema B and the ghost schema G are public and not alterable;

A2: the sink maintains the set NOP_C for each chain C and they are not alterable by the attacker;

A3: the secret key k shared between the sink and the sensor is not guessable;

A4 the hash function used is unbreakable (pre-image resistant, second pre-image resistant, collision resistant).

We define the following *security compromises*:

Definition 7.1. $\mathbf{C_{frs}}$. The query result $\bar{C}$ of a range query $Q_a(l,u)$ is *compromised on freshness* if it differs from the actual query result and is obtained from an old (valid) integrity chain C_{old}.

$\mathbf{C_{crr}}$. The query result $\bar{C}$ of a range query $Q_a(l,u)$ is *compromised on correctness* if it differs from the actual query result and at least one of the following cases holds:

1. There exists an s-tuple $\bar{c^*} = \langle x^*, y^* \rangle \in \bar{C}$, such that x^* is obtained by the storage node by altering an original tuple x.
2. A new s-tuple $\bar{c^*} = \langle x^*, y^* \rangle$ is forged by the storage node and included in $\bar{C}$.

$\mathbf{C_{cmp}}$. Given tuple t of D with value between l and u, the query result $\bar{C}$ of a range query $Q_a(l,u)$ is *compromised on completeness* if it differs from the actual query result and there is no s-tuple in $\bar{C}$ with left-most component equal to t.

In the next definition, we state what query integrity formally means.

Definition 7.2. The query result $\bar{C}$ of a range query $Q_a(l,u)$ is *integrity-proven* if $\neg C_{frs} \wedge \neg C_{crr} \wedge \neg C_{cmp}$ holds.

We show now that the logical conjunction of the four invariants introduced in Definition 3.6 (i.e., the fact that the query result is verification-proof) is sufficient to prove that the query result is integrity-proven, and thus the security of our approach. In other words, this proves the effectiveness of the RANGE_QUERY operation as a way for the sink to have assurance about the integrity of the range-query results returned by the storage node.

Theorem 7.1. *The query result $\bar{C}$ of a range query $Q_a(l,u)$ is integrity-proven if it is verification-proof, according to Definition 3.6.*

Proof. By contradiction we prove that if $\bar{C}$ is not integrity-proven, then it is not verification-proof. Recall that, by Definition 3.6, $\bar{C}$ is verification-proof iff SI1 $\wedge$ SI2 $\wedge$ SI3 $\wedge$ SI4. Moreover, $\bar{C}$ is not integrity-proven iff $C_{frs} \vee C_{crr} \vee C_{cmp}$ holds.

First, suppose that C_{frs} holds. We show that C_{frs} implies that $\neg$ SI3 holds. Recall that, by definition of C_{frs}, the query result $\bar{C}$ is obtained from an old (valid) integrity chain C_{old}. Being a past integrity chain, C_{old} misses at least one INSERT or DELETE operation performed by the sensor or the sink, respectively. Suppose (b_v, b_{v+1}) are the boundaries delimiting the bucket involved in this missing operation. If an INSERT is omitted, then either an s-tuple or a d-tuple is missing from such bucket. Therefore, $nop_v = n_d(v) + n_s(v) + 2n_g(v) + 1$, being such bucket included in the result $\bar{C}$ (otherwise we can exclude that the query result $\bar{C}$ is different from the actual one) and due to Assumptions **A1** and **A2** guaranteeing that nop_v cannot be neither tampered nor associated to boundaries different from (b_v, b_{v+1}) by the adversary. Otherwise, if a DELETE is omitted, then an extra s-tuple (or d-tuple) occurs in the bucket while one g-tuple is missing. Therefore, $nop_v = n_d(v) + n_s(v) + 2n_g(v) - 1 + 2$, for the same reason as before. In both cases, the invariant SI3 (i.e., $nop_v = n_d(v) + n_s(v) + 2n_g(v)$) is invalidated, and this concludes this part of the proof.

Now, suppose that C_{Crr} holds. We show that C_{Crr} implies that $\neg$ SI2 $\vee\neg$ SI3 $\vee\neg$ SI4 holds. According to Definition 7.1, we have to consider two possible cases. The case (1) is that there exists an s-tuple $\bar{c^*} = \langle x^*, y^* \rangle \in \bar{C}$, such that x^* is obtained by the storage node by altering an original tuple x. To keep valid the invariant SI2, the storage node should be able to obtain the pair y_p=HMAC$(k, \bar{H}(x_p)\|\bar{H}(x^*))$, y^*=HMAC$(k, \bar{H}(x^*)\|\bar{H}(x_s))$, where x_s is the left-most component of the element following $\bar{c^*}$ and x_p is the left-most component of the element preceding $\bar{c^*}$ in the query result. By Assumptions **A3** and **A4** the storage node cannot forge them. The only possibilities for the storage node are: (i) to reuse HMACs computed by the sensor or sink in the legal execution of the protocol, (ii) to obtain HMACs by deceiving the sensor or the sink during the execution of the protocol. (i) occurs only if the storage node obtained y_p=HMAC$(k, \bar{H}(x_p)\|\bar{H}(x^*))$, y^*=HMAC$(k, \bar{H}(x^*)\|\bar{H}(x_s))$ because x^* has been previously inserted by the sensor in the database, and then removed the sink. Therefore, two cases may hold. The first is that the storage node does not tamper g-tuples. In this case, the invariant SI4 is invalidated, because of the presence of x^* in the query result. The other case is that the storage node removes the g-tuple corresponding to x^*, to keep SI4, but this invalidates SI3. Due to Assumptions **A3** and **A4**, no way to restore the valid number of g-tuples exists for the storage node in order to keep SI3. Now, we consider (ii). Concerning INSERT, the sensor computes HMACs on values that always include the tuple being inserted, therefore there is no way for the storage node to obtain neither y_p nor y^*. Concerning DELETE, considering that the storage node could keep old HMACs working as legal bypasses to implement the deletion of tuples, the storage node could take advantage of deletions to deceive the sink and obtain the aimed HMACs (i.e., y_p and y^*). But, the way in which DELETE is defined eliminates this possibility. Indeed, when a tuple t must be removed, the storage node provides the sink with $H(x)$, $f_C(x)$, $H(x')$ and $f_C(t)$ and the sink computes HMAC$(k, H(x)\|H(x'))$ by previously checking the correctness of the HMACs of the sub-chain including x, t, x'. If this sub-chain is the legal one, we fall into case (i). The only way for the storage node to return a correct but not legal sub-chain is to use an old version of this sub-chain. The storage node could have an advantage only if, in an old version of the database, either a sub-chain x_p, t, x^*, or a sub-chain x^*, t, x_s exists. In fact, in the first case, the sink computes y_p, in the second case computes y^* (which are the HMACs aimed by the storage node). Suppose that the first case occurs, that is the storage node finds an old sub-chain x_p, t, x^*. Two cases may happen. The first is that x^* is still in the database. In this case, being the old sub-chain not legal, other elements besides t are included between x_p and x^* in the legal sub-chain. Therefore, the attack on correctness results in a bypass excluding some tuples and then invalidating the invariant SI3. The other case is that x^* has been removed from the database. In this case, the attack on correctness could result in changing a tuple x with x^*, but this would invalidate the invariant SI4 because x^* corresponds to a g-tuple in the bucket.

A similar reasoning can be applied to the case in which a sub-chain x^*, t, x_s exists. The proof of case (1) is then concluded.

Now we consider case (2) in which the storage node forges a new s-tuple $\bar{c}^* = \langle x^*, y^* \rangle$ and includes it in the query result $\bar{C}$. Similarly to case (1), the only possibilities for the storage node to forge y^* (to keep valid the invariant SI2) are (i) and (ii) above. Therefore at least one between SI3 and SI4 is violated.

Thus, we proved that C_{Crr} implies that $\neg$ SI2 $\vee\neg$ SI3 $\vee\neg$ SI4 holds. It remains to prove that C_{cmp} implies that $\neg$ SI1 $\vee\neg$ SI2 $\vee\neg$ SI3 $\vee\neg$ SI4 holds.

Suppose now that C_{cmp} holds. By definition of C_{cmp}, an expected s-tuple is missing in $\bar{C}$. Due to Assumptions **A1** and **A2**, to keep valid the invariant SI3, the storage node should be able to replace the reduced number of s-tuples by increasing the number of g-tuples or d-tuples. The increment of s-tuples falls in the case of correctness compromise and has been already treated above.

First, consider the case of g-tuples. Since no old version of the chain can contain more g-tuples than the current version, the storage node should forge entirely the new g-tuples. By Assumptions **A3** and **A4**, HMACs cannot be forged, thus they only can be obtained as in (i) and (ii). Therefore, either SI2 or SI4 will be invalidated.

Now, we consider the case of d-tuples. We have to consider two cases: forged d-tuples or d-tuples obtained from s-tuples. In order to forge a d-tuple $c = \langle x, y \rangle$, due to Assumptions **A3** and **A4**, y cannot be forged, thus the only possibilities for the storage node to obtain a valid y are (i) and (ii). In this case either SI2 or SI4 will be invalidated.

Finally, a d-tuple can always be obtained from an s-tuple (i.e., the s-tuple missing in the query result) by replacing the left-most component with its digest. However, due to properties (3) and (5) of the Invariant SI1, the d-tuples have to be consecutive and must occur only in the left-most and in the right-most bucket of the query result. Therefore, the only s-tuples which can be transformed in d-tuples are the first s-tuple $\bar{c}_{t'}$ following the d-tuples in the first bucket and the last s-tuple $\bar{c}_{y'}$ preceding the d-tuples in the last bucket. However, due to the property (8) of SI1, $\bar{c}_{t'}$ must have value strictly less than l and the successive s-tuple $\bar{c}_{t'+1}$ must have value greater or equal to l, thus by replacing $\bar{c}_{t'}$ with a d-tuple, the Invariant SI1 is invalidated. A similar reasoning applies for $\bar{c}_{y'}$. The proof is then concluded and the theorem is proved.

8. Conclusion

In this paper, we propose a new solution to guarantee query integrity in a two-tier sensor network scenario. The focus of this work is to show that in this context, where insertion operations are very frequent, our approach outperforms the existing techniques. In order to accomplish this, we performed a comparative analysis between our solution and MHT approaches that represent the state of the art. From this analysis, it results that our technique has a constant cost of insertion against the logarithm cost of the MHT solution. Moreover, we do not have substantial differences in terms of computational cost in the verification phase. We stress that, typically, the critical operations are insertion and deletion, because their security must be guaranteed at the source (and thus, by constrained devices). In contrast, query verification is performed by the sink, that is not resource-constrained. Finally, in order to validate our results, we provided an experimental analysis on synthetic data structures and a formal security analysis that shows that query integrity is guaranteed if the query-result verification succeeds.

References

[1] A. Bari, S. Wazed, A. Jaekel, S. Bandyopadhyay, A genetic algorithm based approach for energy efficient routing in two-tiered sensor networks, Ad Hoc Networks 7 (4) (2009) 665–676.

[2] J. Pan, L. Cai, Y. T. Hou, Y. Shi, S. X. Shen, Optimal base-station locations in two-tiered wireless sensor networks, IEEE Transactions on Mobile Computing 4 (5) (2005) 458–473.

[3] P. Samarati, Data security and privacy in the cloud, in: International Conference on Information Security Practice and Experience, Springer, 2014, pp. 28–41.

[4] F. Li, M. Hadjieleftheriou, G. Kollios, L. Reyzin, Dynamic authenticated index structures for outsourced databases, in: Proceedings of the 2006 ACM SIGMOD international conference on Management of data, ACM, 2006, pp. 121–132.

[5] R. Sion, Query execution assurance for outsourced databases, in: Proceedings of the 31st international conference on Very large data bases, VLDB Endowment, 2005, pp. 601–612.

[6] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, C. Papamanthou, vsql: Verifying arbitrary sql queries over dynamic outsourced databases, in: 2017 IEEE Symposium on Security and Privacy (SP), IEEE, 2017, pp. 863–880.

[7] Q. Zheng, S. Xu, G. Ateniese, Efficient query integrity for outsourced dynamic databases, in: Proceedings of the 2012 ACM Workshop on Cloud Computing Security Workshop, CCSW '12, ACM, New York, NY, USA, 2012, pp. 71–82. doi:10.1145/2381913.2381927.
URL http://doi.acm.org/10.1145/2381913.2381927

[8] R. Li, A. X. Liu, S. Xiao, H. Xu, B. Bruhadeshwar, A. L. Wang, Privacy and integrity preserving top- k query processing for two-tiered sensor networks, IEEE/ACM Transactions on Networking 25 (4) (2017) 2334–2346. doi:10.1109/TNET.2017.2693364.

[9] H. Dai, G. Yang, X. Qin, Emqp: An energy-efficient privacy-preserving max/min query processing in tiered wireless sensor networks, International Journal of Distributed Sensor Networks 2013 (07 2013). doi:10.1155/2013/814892.

[10] F. Chen, A. X. Liu, Privacy- and integrity-preserving range queries in sensor networks, IEEE/ACM Transactions on Networking 20 (6) (2012) 1774–1787. doi:10.1109/TNET.2012.2188540.

[11] J. Bu, M. Yin, D. He, F. Xia, C. Chen, Sef: A secure, efficient, and flexible range query scheme in two-tiered sensor networks, International Journal of Distributed Sensor Networks 7 (1) (2011) 126407. arXiv:https://doi.org/10.1155/2011/126407, doi:10.1155/2011/126407.
URL https://doi.org/10.1155/2011/126407

[12] B. Sheng, Q. Li, Verifiable privacy-preserving range query in two-tiered sensor networks, in: IEEE INFOCOM 2008-The 27th Conference on Computer Communications, IEEE, 2008, pp. 46–50.

[13] X. Kui, J. Feng, X. Zhou, H. Du, X. Deng, P. Zhong, X. Ma, Securing top-k query processing in two-tiered sensor networks, Connection Science 33 (1) (2021) 62–80.

[14] M. McCarthy, Z. He, X. S. Wang, Evaluation of range queries with predicates on moving objects, IEEE Transactions on Knowledge and Data Engineering 26 (5) (2013) 1144–1157.

[15] R. C. Merkle, A certified digital signature, in: Advances in Cryptology—CRYPTO'89 Proceedings, Springer, 1989, pp. 218–238.

[16] F. Buccafurri, G. Lax, S. Nicolazzo, A. Nocera, Range query integrity in cloud data streams with efficient insertion, in: International Conference on Cryptology and Network Security, Springer, 2016, pp. 719–724.

[17] H. Wang, D. He, A. Fu, Q. Li, Q. Wang, Provable data possession with outsourced data transfer, IEEE Transactions on Services Computing (2019).

[18] C. Lin, Z. Shen, Q. Chen, F. T. Sheldon, A data integrity verification scheme in mobile cloud computing, Journal of Network and Computer Applications 77 (2017) 146–151.
[19] C. C. Erway, A. Küpçü, C. Papamanthou, R. Tamassia, Dynamic provable data possession, ACM Transactions on Information and System Security (TISSEC) 17 (4) (2015) 15.
[20] Y.-J. Ren, J. Shen, J. Wang, J. Han, S.-Y. Lee, Mutual verifiable provable data auditing in public cloud storage, Journal of Internet Technology 16 (2) (2015) 317–323.
[21] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, D. Song, Provable data possession at untrusted stores, in: Proceedings of the 14th ACM conference on Computer and communications security, Acm, 2007, pp. 598–609.
[22] Q. Wang, C. Wang, K. Ren, W. Lou, J. Li, Enabling public auditability and data dynamics for storage security in cloud computing, IEEE Transactions on Parallel and Distributed Systems 22 (5) (2011) 847–859 doi:10.1109/TPDS.2010.183.
[23] T. M, P. Varalakshmi, Enabling ternary hash tree based integrity verification for secure cloud data storage, IEEE Transactions on Knowledge and Data Engineering (2019) 1–1doi:10.1109/TKDE.2019.2922357.
[24] B. Zhang, B. Dong, W. H. Wang, Integrity authentication for sql query evaluation on outsourced databases: A survey, IEEE Transactions on Knowledge and Data Engineering (2019).
[25] F. Chen, A. X. Liu, Privacy and integrity preserving multi-dimensional range queries for cloud computing, in: 2014 IFIP Networking Conference, 2014, pp. 1–9. doi:10.1109/IFIPNetworking.2014.6857083.
[26] M. Xie, H. Wang, J. Yin, X. Meng, Integrity auditing of outsourced data, in: Proceedings of the 33rd international conference on Very large data bases, VLDB Endowment, 2007, pp. 782–793.
[27] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, P. Samarati, Efficient integrity checks for join queries in the cloud 1, Journal of Computer Security 24 (3) (2016) 347–378.
[28] W.-S. Ku, L. Hu, C. Shahabi, H. Wang, A query integrity assurance scheme for accessing outsourced spatial databases, Geoinformatica 17 (1) (2013) 97–124.
[29] F. Li, M. Hadjieleftheriou, G. Kollios, L. Reyzin, Authenticated index structures for aggregation queries, ACM Trans. on Information and System Security (TISSEC) 13 (4) (2010) 32.
[30] H. Pang, J. Zhang, K. Mouratidis, Scalable verification for outsourced dynamic databases, Proceedings of the VLDB Endowment 2 (1) (2009) 802–813.
[31] P. Devanbu, M. Gertz, C. Martel, S. G. Stubblebine, Authentic data publication over the internet, Journal of Computer Security 11 (3) (2003) 291–314.
[32] D. Ma, R. H. Deng, H. Pang, J. Zhou, Authenticating query results in data publishing, in: International conference on information and communications security, Springer, 2005, pp. 376–388.
[33] M. S. Niaz, G. Saake, Merkle hash tree based techniques for data integrity of outsourced data., in: GvD, 2015, pp. 66–71.
[34] H. Pang, A. Jain, K. Ramamritham, K.-L. Tan, Verifying completeness of relational query results in data publishing, in: Proceedings of the 2005 ACM SIGMOD international conference on Management of data, ACM, 2005, pp. 407–418.
[35] M. Narasimha, G. Tsudik, Authentication of outsourced databases using signature aggregation and chaining, in: International conference on database systems for advanced applications, Springer, 2006, pp. 420–436.
[36] W. Cheng, K.-L. Tan, Query assurance verification for outsourced multi-dimensional databases, Journal of Computer Security 17 (2009) 101–126. doi:10.3233/JCS-2009-0315.
[37] Y.-T. Tsou, C.-S. Lu, S.-Y. Kuo, Privacy- and integrity-preserving range query in wireless sensor networks, in: 2012 IEEE Global Communications Conference (GLOBECOM), 2012, pp. 328–334. doi:10.1109/GLOCOM.2012.6503134.
[38] F. Buccafurri, G. Lax, S. Nicolazzo, A. Nocera, Range query integrity in the cloud: the case of video surveillance, in: 2016 11th International Conference for Internet Technology and Secured Transactions (ICITST), IEEE, 2016, pp. 170–175.
[39] H. Krawczyk, R. Canetti, M. Bellare, Hmac: Keyed-hashing for message authentication (1997).
[40] V. Poosala, P. J. Haas, Y. E. Ioannidis, E. J. Shekita, Improved histograms for selectivity estimation of range predicates, in: ACM Sigmod Record, Vol. 25, ACM, 1996, pp. 294–305.
[41] Y. E. Ioannidis, V. Poosala, Balancing histogram optimality and practicality for query result size estimation, in: Acm Sigmod Record, Vol. 24, ACM, 1995, pp. 233–244.
[42] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. C. Sevcik, T. Suel, Optimal histograms with quality guarantees, in: VLDB, Vol. 98, 1998, pp. 24–27.
[43] F. Buccafurri, G. Lax, D. Saccà, L. Pontieri, D. Rosaci, Enhancing histograms by tree-like bucket indices, The VLDB Journal 17 (5) (2008) 1041–1061.
[44] F. Buccafurri, D. Rosaci, L. Pontieri, D. Saccà, Improving range query estimation on histograms, in: Proceedings 18th International Conference on Data Engineering, IEEE, 2002, pp. 628–638.
[45] S. Guha, N. Koudas, D. Srivastava, Fast algorithms for hierarchical range histogram construction, in: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, ACM, 2002, pp. 180–187.

Appendix A. SET_UP

We prove that C, as obtained in Section 3.1, is a *well-formed* chain in the sense of Definition 3.2.

First, since $c_k = \langle b_k, NULL\rangle$ and $c_1 = \langle b_1, h_1\rangle$, Properties 1 and 4 hold. Given two markers $c_i = \langle b_i, h_i\rangle$ and $c_j = \langle b_j, h_i\rangle$, $b_i \neq b_j$ for definition of B and Property 5 holds too. Trivially, there is no s-tuple in C, so also Property 6 holds. Now, for $1 \leq i \leq n-1$, given $c_i = \langle b_i, h_i\rangle$, $c_{i+1} = \langle b_{i+1}, h_{i+1}\rangle$, $h_i = \text{HMAC}(k, H(b_i)||H(b_{i+1}))$ and Propriety 2 is satisfied. Finally, given $c_i = \langle b_i, h_i\rangle$, $c_j = \langle b_j, h_j\rangle$, $val(c_i) = b_i < val(c_j) = b_j$ if $i < j$, for definition of B and also Property 3 is satisfied. Therefore, C is a well-formed chain.

Appendix B. INSERT

We prove that C', as obtained in Section 3.2, is a *well-formed* chain in the sense of Definition 3.2.

First, $c'_1 = c_1 = \langle b_1, f_C(b_1)\rangle$ and $c'_{n+1} = c_n = \langle b_k, f_C(b_k)\rangle$, thus Properties 1 and 4 are satisfied.

Concerning Property 5, given two markers c'_x and c'_y ($x \neq y$), we distinguish three cases:

1. $1 \leq x, y \leq i$ then $c'_x = c_x$ and $c'_y = c_y$. Since Property 5 is verified in C, then $val(c'_x) = val(c_x) \neq val(c_y) = val(c'_y)$ and Property 5 is satisfied also in C'.
2. $1 \leq x \leq i$ and $i+1 < y \leq n+1$ then $c'_x = c_x$ and $c'_y = c_{y-1}$. Since $y - 1 \neq x$ and the Property 5 is verified in C, then $val(c'_x) = val(c_x) \neq val(c_{y-1}) = val(c'_y)$ and Property 5 holds in C' too.
3. $i+1 < x, y \leq n+1$ then $c'_x = c_{x-1}$ and $c'_y = c_{y-1}$. By following the same reasoning as in case (1), it is easy to see that Property 5 holds in C'.

Concerning Property 6, given $c'_x, c'_y \in C$ such that c'_x is a marker, c'_y is an s-tuple and $val(c'_x) = val(c'_y)$:

1. if $1 \leq x, y \leq i$, then $c'_x = c_x$, $c'_y = c_y$. Since Property 6 is satisfied in C, then $x < y$ (i.e., Property 6 is satisfied in C',too).
2. if $i+1 < x, y \leq n+1$, then $c'_x = c_{x-1}$, $c'_y = c_{y-1}$. Since Property 6 is satisfied in C, then $x - 1 < y - 1$ and $x < y$ (i.e., Property 6 is satisfied in C', too).

3. By contradiction, if $1 \leq y \leq i$ and $i+1 < x \leq n+1$, then $c'_y = c_y$, $c'_x = c_{x-1}$, but $x-1 > y$ and this violates Property 6 in C, so this is not possible.
4. By contradiction, if $y = i+1$ and $i+1 < x \leq n+1$, then $c'_y = c$, $c'_x = c_{x-1}$. If $x-1 > i+1$ then $val(c_{x-1}) \geq val(c_{i+1})$, but $val(c'_y) = val(c'_x) = val(c) = val(c_{x-1})$, so $val(c) \geq val(c_{i+1})$ and this is impossible. If $x-1 = i+1$, then $val(c'_y) = val(c) = val(c_{x-1}) = val(c_{i+1})$ and again it is not possible. Therefore, the case $y = i+1 \leq i$ and $i+1 < x \leq n+1$ cannot occur.
5. in all the other cases, trivially, $x < y$ (i.e., Property 6 holds).

Concerning Property 2, given an element $c'_x \in C'$:

1. if $1 \leq x < i$, then $c'_x = c_x$ and $c'_{x+1} = c_{x+1}$, so since Property 2 is satisfied in C, it holds in C'.
2. if $x{=}i$, then $c'_x{=}c_x{=}c_i{=}\langle x, f_C(x)\rangle$ and $c'_{x+1}{=}c{=}\langle t, h_5\rangle$. Moreover, $f_C(x) = h_4{=}\text{HMAC}(k, h_1||h_3){=}\text{HMAC}(k, H(x)||H(t))$, so Property 2 is verified.
3. if $x{=}i{+}1$, then $c'_x{=}c{=}\langle t, h_5\rangle$ and $c'_{x+1}{=}c_x{=}c_{i+1}{=}\langle x', f_C(x')\rangle$ where $h_5{=}\text{HMAC}(k, h_3||h_2){=}\text{HMAC}(k, H(t)||H(x'))$ and Property 2 holds.
4. if $i+1 < x \leq n$, then $c'_x = c_{x-1}$ and $c'_{x+1} = c_x$. Since Property 2 is satisfied in C, it holds in C' too.

Regarding Property 3, again, we have several cases. Given $c'_x, c'_y \in C'$ such that $x < y$:

1. if $1 \leq x < y \leq i$ then $c'_x = c_x$, $c'_y = c_y$ and since Property 3 is satisfied in C, $val(c'_x) = val(c_x) \leq val(c_y) = val(c'_y)$ (i.e., Property 3 is verified).
2. if $1 \leq x \leq i$ and $y = i+1$ then $c'_x = c_x$ and $c'_y = c = \langle t, h_5\rangle$. If $x = i$, $val(c'_x) = val(c_x) = val(c_i) \leq val_t(a) = val(t) = val(c) = val(c'_y)$. Otherwise ($x < i$), $val(c'_x) = val(c_x) \leq val(c_i) \leq val(c) = val(c'_y)$ (thus, Property 3 is satisfied).
3. If $1 \leq x \leq i$ and $i+1 < y \leq n+1$ then $c'_x = c_x$, $c'_y = c_{y-1}$. Clearly $x < y-1$, thus $val(c'_x) = val(c_x) \leq val(c_{y-1}) = val(c'_y)$ (i.e., Property 3 holds).
4. If $x = i+1$ and $i+1 < y \leq n+1$ then $c'_x = c = \langle t, h_5\rangle$, $c'_y = c_{y-1}$. If $y = i+2$ then $y-1 = i+1$ and $val(c'_x) = val(c) \leq val(c_{i+1}) = val(c_{y-1}) = val(c'_y)$. Otherwise ($y > i+2$) then $y-1 > i+1$ and $val(c'_x) = val(c) \leq val(c_{i+1}) \leq val(c_{y-1}) = val(c'_y)$, (i.e., Property 3 is verified).
5. If $i+1 < x < y \leq n+1$ then $c'_x = c_{x-1}$, $c'_y = c_{y-1}$ and $val(c'_x) = val(c_{x-1}) \leq val(c_{y-1}) = val(c'_y)$ (thus, Property 3 holds).

Therefore, C' is a well-formed chain.

Appendix C. DELETE

We prove that C', as obtained in Section 3.3, is a *well-formed* chain in the sense of Definition 3.2.

First, $c'_1 = c_1 = \langle b_1, f_C(b_1)\rangle$ and $c'_n = c_n = \langle b_k, f_C(b_k)\rangle$, thus both Properties 1 and 4 hold.

Concerning Property 5, given two markers c'_x and c'_y ($x \neq y$), we distinguish 5 cases:

1. if $1 \leq x, y < i$ or $k \leq x, y \leq n$, then $c'_x = c_x$ and $c'_y = c_y$. Since Property 5 is satisfied in C, then $val(c'_x) = val(c_x) \neq val(c_y) = val(c'_y)$ and Property 5 is satisfied in C' too.
2. if $1 \leq x < i$ and $k \leq y \leq n$, then $c'_x = c_x$ and $c'_y = c_y$. Similarly to case (1), we can argue that Property 5 is verified in C'.
3. if $1 \leq x < i$ and $i \leq y < k-1$, then $c'_x = c_x$ and $c'_y = c_{y+1}$. Since $y+1 \neq x$ and Property 5 is satisfied in C, then $val(c'_x) = val(c_x) \neq val(c_{y+1}) = val(c'_y)$ and Property 5 is satisfied also in C'.
4. if $k \leq x \leq n$ and $i \leq y < k-1$, then $c'_x = c_x$ and $c'_y = c_{y+1}$. Since $y+1 \neq x$, again, we can apply the same considerations as in case (3).
5. if $i \leq x, y < k-1$ then $c'_x = c_{x+1}$ and $c'_y = c_{y+1}$. By following the same reasoning as in case (1), Property 5 holds in C'.

Regarding Property 6, given $c'_x, c'_y \in C$ such that c'_x is a marker, c'_y is an s-tuple and $val(c'_x) = val(c'_y)$:

1. if $1 \leq x, y < i$ or $k \leq x, y \leq n$, then $c'_x = c_x$, $c'_y = c_y$. Since Property 6 is satisfied in C, then $x < y$ (thus, Property 6 is satisfied in C' too).
2. if $i \leq x, y < k-1$, then $c'_x = c_{x+1}$, $c'_y = c_{y+1}$. Since Property 2 is satisfied in C, then $x+1 < y+1$ and $x < y$ (thus, Property 6 is satisfied in C' too).
3. by contradiction, if $1 \leq y < i$ and $k \leq x \leq n$, then $c'_y = c_y$, $c'_x = c_x$. Since $k > i$, then $x > y$ and this violates Property 6 in C, so this case is not possible.
4. by contradiction, if $1 \leq y < i$ and $i \leq x < k-1$, then $c'_y = c_y$, $c'_x = c_{x+1}$. Since $x+1 > y$, Property 6 in C is violated, thus this case is not possible.
5. by contradiction, if $i \leq y < k-1$ and $k \leq x \leq n$, then $c'_y = c_{y+1}$, $c'_x = c_x$. We have that $y+1 < k$, so $y+1 < x$. Again, this violates Property 6 in C, thus this case is not possible.
6. in all the other cases, it holds that $x < y$, so that Property 6 is trivially verified.

Regarding Property 2, given $c'_x \in C'$:

1. if $1 \leq x < i-1$, then $c'_x = c_x$ and $c'_{x+1} = c_{x+1}$. Since Property 2 is satisfied in C, it is also satisfied in C'.
2. if $x = i-1$, then $c'_x{=}c'_{i-1}{=}c_{i-1}{=}\langle x, f_C(x)\rangle$. If $i = k-1$, then $c'_{x+1}{=}c'_i{=}c'_{k-1}{=}c{=}\langle g_v||j_v+1||h_5, h_9\rangle$ and $f_C(x){=}h_6{=}\text{HMAC}(k, h_1||h_7){=}\text{HMAC}(k, H(x)||H(g_v||j_v{+}1||h_5))$ and thus, Property 2 is satisfied. Otherwise ($i \neq k-1$), $c'_{x+1} = c'_i = c_{i+1} = \langle x', f_C(x')\rangle$ and $f_C(x) = h_6 {=}\text{HMAC}(k, h_1||h_2){=}\text{HMAC}(k, H(x)||H(x'))$, thus Property 2 is satisfied.
3. if $i \leq x < k-2$, then $c'_x = c_{x+1}$ and $c'_{x+1} = c_{x+2}$. Similarly to case (1), Property 2 is satisfied.
4. if $x = k-2$ and $i \neq k-1$, then $c'_x{=}c'_{k-2}{=}c_{k-1}{=}\langle \bar{x}, f_C(\bar{x})\rangle$ and $c'_{x+1}{=}c'_{k-1}{=}c{=}\langle g_v||j_v+1||h_5, h_9\rangle$ where $f_C(\bar{x}) = h_8 {=}\text{HMAC}(k, h_3||h_7) {=}\text{HMAC}(k, H(\bar{x})||H(g_v||j_v+1||h_5))$, thus Property 2 is verified. (If $i = k-1$, we fall in case (2).)

5. if $x = k - 1$, then $c'_x = c'_{k-1} = c = \langle (g_v, j_v + 1) \| h_5, h_9 \rangle$ and $c'_{x+1} = c'_k = c_k = \langle b_{v+1}, f_C(b_{v+1}) \rangle$ where h_9=HMAC$(k, h_7 \| h_4)$=HMAC$(k, H((g_v, j_v+1) \| h_5) \| H(b_{v+1})$ (Property 2 verified)
6. if $k \leq x \leq n - 1$, then $c'_x = c_x$ and $c'_{x+1} = c_{x+1}$. Similarly to case (1), Property 2 is verified.

Regarding Property 3, given $c'_x, c'_y \in C'$ such that $x < y$:

1. if $1 \leq x < y < i$, then $c'_x = c_x$, $c'_y = c_y$ and $val(c'_x) = val(c_x) \leq val(c_y) = val(c'_y)$ (Property 3 is verified).
2. if $1 \leq x < i$ and $i \leq y < k - 1$, then $c'_x = c_x$, $c'_y = c_{y+1}$ and, since $x < y + 1$, $val(c'_x) = val(c_x) \leq val(c_{y+1}) = val(c'_y)$ (Property 3 is satisfied).
3. if $1 \leq x < i$ and $y = k - 1$ then $c'_x = c_x$, $c'_y = c = \langle (g_v, j_v + 1) \| h_5, h_9 \rangle$ and $val(c'_x) = val(c_x) \leq val(c_i) = val_t(a)$. By definition 3.1, $s < (g_v, j) < b_{v+1}$ for any j, where $s \in U(a)$ is the value preceding b_{v+1} in the total order of $\hat{U}(a)$. Since $val_t(a) \in U(a)$ and $val_t(a) < b_{v+1}$, then $val_t(a) < (g_v, j_v + 1) < b_{v+1}$. Thus, $val(c'_x) \leq val_t(a) < (g_v, j_v + 1) = val(c'_y)$ (Property 3 is verified).
4. if $1 \leq x < i$ and $k \leq y \leq n$ then $c'_x = c_x$, $c'_y = c_y$ and, as in the case (1), Property 3 is satisfied.
5. if $i \leq x < y < k - 1$ then both $c'_x = c_{x+1}$ and $c'_y = c_{y+1}$. Therefore, $val(c'_x) = val(c_{x+1}) \leq val(c_{y+1}) \leq val(c'_y)$ (Property 3 is verified).
6. if $i \leq x < k - 1$ and $y = k - 1$ then $c'_x = c_{x+1}$, $c'_y = c = \langle (g_v, j_v + 1) \| h_5, h_9 \rangle$. Since $i < x + 1 < k$, then $val(c_i) \leq val(c_{x+1}) \leq val(c_k) = b_{v+1}$. But $b_v \leq val(c_i) < b_{v+1}$, so $b_v \leq val(c_{x+1}) \leq b_{v+1}$. By contradiction, if $val(c_{x+1}) = b_{v+1} = val(c_k)$, then c_{x+1} cannot be a g-tuple (because its value $\in U(a)$). Moreover, c_{x+1} cannot be an s-tuple because, for Property 6, $x + 1$ should be greater than k. Finally, c_{x+1} cannot be even a marker because c_k and c_{x+1} would be two markers with the same boundary (in contrast with Property 5), thus $b_v \leq val(c_{x+1}) < b_{v+1}$. Again, suppose, by contradiction, that $c_{x+1} = \langle b_\alpha, f_C(b_\alpha) \rangle$ (i.e., c_{x+1} is a marker). Then, it would be $b_v \leq val(c_{x+1}) = b_\alpha < b_{v+1}$ and it is not possible for definition of B, so c_{x+1} is an s-tuple or a g-tuple. If c_{x+1} is an s-tuple, then $val(c_{x+1}) \in U(a)$ and $val(c_{x+1}) < b_{v+1}$. Thus, $val(c'_x) = val(c_{x+1}) < (g_v, j_v + 1) = val(c'_y) < b_{v+1}$ and Property 3 is satisfied. If c_{x+1} is a g-tuple, since $b_v \leq val(c_{x+1}) < b_{v+1}$, then $val(c_{x+1}) = (g_v, j)$ for some $j \leq j_v$. Thus, $val(c'_x) = val(c_{x+1}) = (g_v, j) < (g_v, j_v + 1) = val(c'_y)$ and Property 3 holds.
7. if $i \leq x < k - 1$ and $k \leq y \leq n$, then $c'_x = c_{x+1}$, $c'_y = c_y$ and, since $x + 1 < y$, $val(c'_x) = val(c_{x+1}) \leq val(c_y) = val(c'_y)$ (Property 3 holds).
8. if $x = k-1$ and $k \leq y \leq n$, then $c'_x = c = \langle (g_v, j_v+1) \| h_5, h_9 \rangle$, $c'_y = c_y$. If $y = k$, then $c'_y = c_k = \langle b_{v+1}, f_C(b_{v+1}) \rangle$, $val(c'_x) = val(c) = (g_v, j_v + 1) < b_{v+1} = val(c'_y)$ (Property 3 holds). Otherwise ($y > k$), $val(c'_x) = val(c) = (g_v, j_v + 1) < b_{v+1} = val(c_k) \leq val(c_y) = val(c'_y)$ (Property 3 holds).
9. if $k \leq x < y \leq n$, then $c'_x = c_x$, $c'_y = c_y$ and, as in case (1), Property 3 is satisfied.

Therefore, C' is a well-formed chain.