IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received March 11, 2021, accepted May 6, 2021, date of publication May 12, 2021, date of current version May 25, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3079799

Integrity Checking for Aggregate Queries

SOMAYEH DOLATNEZHAD SAMARIN AND MORTEZA AMINI
Department of Computer Engineering, Sharif University of Technology, Tehran 1458889694, Iran
Corresponding author: Morteza Amini (amini@sharif.edu)

ABSTRACT With the advent of cloud computing and Internet of Things and delegation of data collection and aggregation to third parties, the results of the computations should be verified. In distributed models, there are multiple sources. Each source creates authenticators for the values and sends them to the aggregator. The aggregator combines the authenticated values and creates a verification object for verifying the computation/aggregation results. In this paper, we propose two constructions for verifying the results of countable and window-based countable functions. These constructions are useful for aggregate functions such as median, max/min, top-k/first-k, and range queries, where the distribution of values is not visible for sources but is visible to the aggregator. The proposed constructions are secure based on the RSA problem in the random oracle model and have the correctness and succinctness properties. Experimental results show that the communication and computation costs of the constructions are acceptable in practice and the proposed solution can be employed for real-world applications.

INDEX TERMS Integrity, cloud computing, secure delegation of computation, data aggregation.

I. INTRODUCTION

Data aggregation is the process of gathering data and extracting statistical information from them in a summary form. Nowadays, data aggregation is widely used in various applications including network traffic analysis, wireless sensor networks (WSNs), Internet of Things (IoT), and data stream management systems (DSMSs). In some applications, a large number of source nodes are distributed over a network while engaging in gathering data. Each source continuously generates data and sends them to a designated aggregator. By increasing the volume of the distributed data, the aggregator needs more storage, processing power, and network bandwidth. In such a way, data aggregation could be done by a powerful third-party system or more than one node.

With the widespread adoption of cloud computing platforms, enterprises or individuals may tend to outsource their storage and computation to the cloud. In the outsourcing of data aggregation, three main entities participate, (1) source nodes, (2) one or more external aggregators, and (3) an end-user. The aggregators receive values sent by all sources and execute registered queries (each query executes an aggregate function) on them and send the results to the end-user. The end-user can also be the owner of the sources and can distribute the sources over the network.

The associate editor coordinating the review of this manuscript and approving it for publication was Alessandra De Benedictis.

In the outsourcing scenario, the aggregators as external entities are not trusted and can compromise the confidentiality and integrity of the data and computations. Confidentiality is out of the scope of this paper, and this paper only focuses on the integrity of the data and computations. Actually, in this paper, we propose a solution for checking the integrity of the range and aggregate queries in a system with distributed sources.

Among the existing approaches proposed for different applications such as the DSMSs, WSNs, and IoT, homomorphic authenticators (homomorphic MACs or homomorphic signatures) are commonly used for checking the integrity of the aggregate queries. In most of the schemes proposed for homomorphic authenticators (in the single-source [1] or multi-source models [2]), each source generates an authenticated value and sends it to the aggregator. The aggregator executes aggregate queries on the authenticated values, generates a verification object (VO) for verifying the results, and sends the results and the verification object to the end-user.

In this paper, we also use a RSA-based homomorphic authenticator to authenticate the result of linear functions, statistical aggregates, and range queries on data collected from distributed sources. It should be noted that the system model considered in our paper is close to the system model considered in DSMSs. On the other hand, aggregate queries and window-based aggregate queries addressed in our paper are widely used in DSMSs.

TABLE 1. A detailed overview of the different methods for checking the integrity of the results in DSMSs.

Constructions	Model	Data Owner	Query Type			Window-based	False Positive	Cryptographic Tool
			Range	Aggregate	Linear Function (Sum)			
Li et al. [3]	SS	SO	✓	✓	✓	✓	✓	Merkle-Tree
Yi et al. [5]	SS	SO	-	-	✓	✓	✓	-
Nath et al. [6]	SS	SO	-	-	✓	✓	-	Group-based
Papadopoulos et al. [7]	DS	SO	-	-	✓	-	-	Pseudo-Random Function
Liu et al. [8]	DS	MO	-	-	✓	-	-	Bilinear Map
Wang et al. [9]	DS	MO	-	-	✓	-	-	Bilinear Map
Our proposed approach	DS	MO	✓	✓	✓	✓	-	RSA Signature

SS: Single Source, **DS**: Distributed Sources, **SO**: Single-Owner, **MO**: Multi-Owner.

Table 1 summarizes a detailed overview of the proposed solutions in DSMSs and our method. As we can see in this table, regarding the supported query type, the paper proposed by Li *et al.* [3] is the only solution that supports the range and statistical aggregate queries in DSMSs. They used the Merkle hash tree for verifying the integrity of results in a single source model. In their solution, the source collects b number of values, creates a Merkle tree using these values, and sends the signed root of the tree to the server. On the server-side, the server creates the Merkel tree for each b values, executes the query on the values fallen in the window with size $n > b$, and sends the results and the verification object to the verifier. For each tree that fall in the window, a separate verification object is created. The verifier may receive some additional values (when n is not a multiplication of b) and should filter the false positives according to the query. In their paper, they claim that their solution supports distributive aggregates, but they do not specify how the result of aggregate queries such as Top-k and Median that may fall in different trees are verified.

Although using the Merkle tree is a common method for verifying the range and aggregate queries, but in a distributed system, creating a unified Merkle tree with distributed leaf values is not possible. In this paper, we use the bucket partitioning concept to create the appropriate authenticators that can be combined on the server-side.

To the best of our knowledge, there is not any solution for verifying statistical aggregate queries such as max/min, top-k/first-k, range, and median in DSMSs with more than one source node. In these queries, the output of the query is a subset of the input values and the order of the input values, in computing the output, is also important. We call these queries countable queries, because we use the counting operation to verify the integrity of the results. We summarize our contributions as follows.

- We propose the first scheme for verifying all countable aggregate queries, especially the median (needs to verify the distribution of all input values) in DSMSs in a multi-source model with different data owners.
- We propose a general construction for verifying countable queries using the bucket partitioning algorithm, counting operation, and linear homomorphic authenticators. Actually, in our construction, we use an efficient linear homomorphic signature. This signature is based on the signature proposed by Gennaro *et al.* [4], which is secure in the random oracle model. In fact, the proposed construction for verifying countable queries depends on the construction defined for verifying linear functions. This makes the proof of the security and correctness of our construction dependent on the proof of security and correctness of the construction used to verify linear functions. We can also simply (with a few changes) replace the linear homomorphic authenticator used in our construction with another linear homomorphic authenticator that is multiplicatively homomorphic.
- To the best of our knowledge, our paper is the first paper that shows how we can use the bucket partitioning to accurately verify the integrity of the aggregate queries. Bucket partitioning is mainly used for executing range queries with encrypted bounds on the encrypted values.
- We provide formal security proofs for the succinctness, correctness, and security of the proposed constructions.

The rest of the paper is organized as follows. In Section II, related works are reviewed. In Section III, the model of the system and the threat model are described. We recall a few notions and standard preliminaries about the bucket partitioning and labeled programs in Section IV. The proposed scheme for verifying the integrity of the countable functions, its correctness and security definition is presented in Section V. Based on this scheme, three constructions are introduced in sections VI, VII, and VIII. The first construction is for authenticating linear functions and the second one extends the first one for authenticating countable queries. The last construction is proposed for window-based countable queries. Section IX describes the experimental results obtained by implementing the constructions, and, finally, Section X concludes the paper.

II. RELATED WORK

There are some research trends concentrated on the verification of computations such as (1) verifiable computations, (2) homomorphic authenticators and (3) methods proposed for verifying the results in specific applications such as outsourced database management system (DBMS), data stream management system (DSMS), wireless sensor network (WSN), and Internet of Things (IoT).

A. VERIFIABLE COMPUTATIONS

In the verifiable delegation of computations, a client wants to delegate the computation of a function to an untrusted party. The untrusted party tries to convince the client by providing a proof for the correctness of the results. The proof is generated using the cryptography and complexity theory primitives. Arora *et al.* [10] proposed the first method for the proof-based verifiable computation systems. Their method is completely theoretical and cannot be implemented in practice. In 2007, Ishai *et al.* [11] proposed a solution that significantly reduced the memory and timing complexity of this method. After that, many studies have been carried out to make these methods applicable in current computing systems [4], [12]–[14]. Nowadays, there are several projects where their prototypes have been implemented [12], [13], [15].

In all methods pointed above, the client uses the input values to verify the results. This approach is not applicable when the source and the verifier are not identical or the source does not store all values (in the case that the source and client are the same).

B. HOMOMORPHIC AUTHENTICATORS

Homomorphic authenticators (HAs) are used to authenticat the results of functions executed on more than one authenticated data. The scheme proposed for HAs are useful for systems that input values are not available for the verifier.

Schemes proposed for HAs are divided into two categories: homomorphic signatures and homomorphic MACs. In the homomorphic signatures, the verification key is public, while in the homomorphic MACs, the verification key is private. Each of these categories contains schemes for single-key and multi-key HAs. In the single-key HAs, a single private key is used to generate authenticators, and a single verification key is used to verifying the results. In the multi-key HAs, there are multiple sources that each of them has a different key. In these schemes, the verifier needs the verification keys of all sources.

The initial proposed single-key HA schemes authenticated the results of linear functions [4], [16]–[18]. After that, some schemes were proposed for bounded degree polynomial functions [19], [20] and general functions with bounded polynomial depth circuits [21], [22].

In 2016, Fior *et al.* [2] provided the first formal definition of a multi-key HA scheme. They used the standard lattice to propose a construction for multi-key homomorphic signature. They also introduced a multi-key homomorphic MAC which is based on a family of pseudo-random functions. At the same time, Lai *et al.* [23] proposed another scheme using SNARKs (Succinct, Non-Interactive Argument of Knowledge) and a digital signature for creating a multi-key homomorphic signature. Schabhüser *et al.* [24] proposed a construction for linear functions based on the bilinear pairing that has the context-hiding property. This property ensures that the verifier cannot obtain additional information about the inputs by observing the results. In this method, the succinctness of the proof and authenticators is related to the number of sources.

C. APPLICATION SPECIFIC METHODS

In the literature, there are some methods proposed for verifying the result of functions in specific applications. Some of these applications are verifiable outsourcing of databases and data-stream management systems, and verifiable data aggregation in WSNs.

1) OUTSOURCED DBMSS

In outsourced database management systems (DBMSs), verifying the integrity of queries is done using the authenticated data structures or probabilistic methods. The methods based on the authenticated data structures usually use a Merkle tree or chained signatures. A Merkle tree is a binary tree used for checking the membership of a value in a set. Since the size of the tree and the communication overhead grows as the number of the outsourced data increases, the methods use the Merkle tree [25]–[27] are not proper for checking the integrity of the complex and multi-attribute queries. Chained signatures are proposed for reducing the communication overhead of tree-based approaches. This approach is widely used for verifying the range queries [28]–[30]. In these methods, the signer creates a chained signature (by chaining the values of a column in different rows) for each value in the table. The server aggregates the signatures according to the query and creates a combined signature for the results.

2) IN-NETWORK AGGREGATION IN WSNS

In a wireless sensor network (WSN), data produced by different sensors are routed to the base station. For reducing the network traffic, data are aggregated in middle nodes and the results are sent to the base station. Therefore, verifying the integrity of the results is needed. For the sake of the little energy consumption, current solutions in this area commonly used replication [31]–[33] and probabilistic methods [34], [35]. In two-tier WSNs, the power of the aggregators is more than the source nodes, so the cryptography tools can also be used in these systems. The methods that use the cryptography primitives, provide the confidentiality and integrity of specific queries such as range, top-k, and max/min [36]–[38] and there is not a general approach for supporting a wide range of queries.

3) OUTSOURCED DSMSS

In 2007, Li *et al.* [3] proposed the first algorithm for verifying the result of queries executed on streams in data stream

management systems (DSMSs). They used Merkle-tree for verifying the integrity of the results, but their solution has some drawbacks. It does not support multi-source models and the authentication information that is generated by a source is dependent on the type of the query executed at the server (the verifier may receive some false positives). After that, the verification of the `GROUP BY SUM` and `Group By Count` queries were noticed by Yi *et al.* [5]. They used the algebraic and probabilistic methods to compute a small synopsis to check the correctness of the results. The source computes the synopsis according to the query executed on the server side and sends it directly to the verifier. In 2013, Nath and Venkatesan [6], extended this solution using the discrete logarithm problem to provide a publicly verifiable signature instead of the secret synopsis. In their method, the source node computes a set of combined signatures for all values in different groups. When the verifier needs to authenticate the result, it can request the combined signature to verify the results.

At the same time, Papadopoulos *et al.* [7] proposed the first method for verifying the result of linear algebraic queries such as sum, inner products, and matrix multiplications. In their model, data are gathered from different sources, but all sources are managed by the same owner. So, each source can create values and signatures on behalf of other sources. After that, Liu *et al.* [8] proposed a construction for verifying inner products and matrix multiplication on data streams that are gathered from multiple sources with different keys. Although they proposed the first publicly verifiable method in the multi-source model, but their method does not support statistical aggregate functions. Wang *et al.* [9] also demonstrated that the constructions proposed by Liu *et al.* [8] are not secure, explained the attacks on their proposed constructions, and improved the constructions to resist against the discovered attacks.

III. SYSTEM AND THREAT MODEL

The system model that we consider in this paper is shown in Fig. 1. In our model, we have three entities: (1) the distributed sources, (2) the aggregator, and (3) the user/verifier.

The system owner creates public parameters PP, secret-keys $\langle ssk, (sk_1, sk_2, \ldots, sk_{n_s})\rangle$, and a verification-key VK. The public parameters (PP) are given to all entities. The secret-key sk_i with $i \in [n_s]$ is given to the source i, the shared secret-key ssk is shared between all sources, and the verification-key is just available for the verifier. Since each source doesn't have the secret-key of other sources, we can assume that each source can be managed by a different data owner. Each source i computes the authentication information (AI_i) for its value (v_i) and sends v_i and AI_i to the aggregator. The aggregator collects the values and their authentication information from all sources and executes the registered aggregate query on them (the query is registered by the verifier). The aggregator computes the results (V_f) and creates a verification object (VO) for verifying the authenticity of them and sends the results and the verification object to the verifier. The verifier verifies the authenticity of the results by checking the verification object using the verification key.

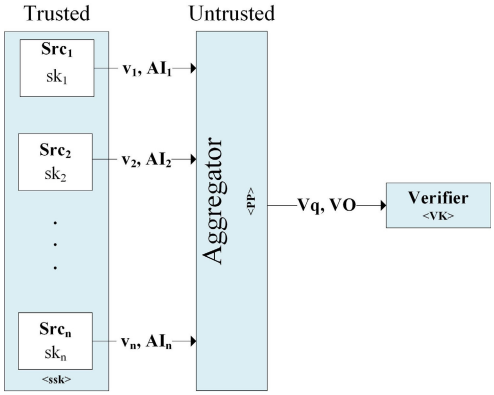

FIGURE 1. System model.

Threat Model: The privacy of values is not the subject of our paper, and we only concentrate on the integrity of data and computations. We assume that all sources are trusted and the aggregator is untrusted. We also assume the sources are not corrupted and cannot collude with the aggregator. The untrusted aggregator can initiate a pollution attack. In the pollution attack, the malicious server can deceive the verifier in various ways: (1) the untrusted server can change or remove some of the input values, (2) change the results, (3) produce random results, (4) execute the query on some values, not on all of them for saving its resource.

In such a model, the results of the query should be verified. So, the main problem is creating a verification object for the aggregation results to authenticate them and confront the mentioned malicious activities.

IV. PRELIMINARIES

Notation: Let λ be the security parameter and $[n] := \{1, \ldots, n\}$. The notation $s \xleftarrow{\$} S$ denotes uniformly sampling the value s from the set S. A function $negl(.)$ is said negligible in λ, if for every polynomial function p, there exists an integer N such that for all integer $n > N$, $negl(\lambda) < \frac{1}{p(n)}$. If A be a probabilistic algorithm (uses random coins), $y \leftarrow A(.)$ denotes assigning the output of the execution of A to the variable y. The list of the notations used in this paper is shown in Table 4.

Definition 1 (Bucket Partitioning [39]): Let the domain of the variable X is $D = [\alpha, \beta]$ where α and β are two positive integers. Bucket partitioning divides this domain to m buckets $B = \{\langle B_1, t_1\rangle, \ldots, \langle B_m, t_m\rangle\}$ where t_i is a tag assigned to the bucket B_i, and the following three conditions are hold:

1) $D = B_1 \cup B_2 \cup \ldots \cup B_m$
2) $\forall \langle B_i, t_i\rangle, \langle B_j, t_j\rangle \in B,\ B_i \neq B_j \Rightarrow B_i \bigcap B_j = \varnothing$
3) $\forall \langle B_i, t_i\rangle, \langle B_j, t_j\rangle \in B,\ B_i \neq B_j \Rightarrow t_i \neq t_j$

Two common preprocessing partitioning algorithms, which could be used for this purpose, are equal-width and equal-depth partitioning [40]. In the equal-width partitioning technique, the domain of values is divided into m equal

TABLE 2. List of frequently used symbols in this paper.

Symbol	Description
λ	Security parameter
n	Input size
n_s	Number of sources
PP	Public parameters
SP	Secret parameters
VK	Verification key
P	Partitioning information
sk_i	Secret-key of the source with identifier i
ssk	Shared secret-key
v	An input value
ℓ	Label of an input value
Δ	Dataset identifier
h	Hash function
m	Number of buckets
t_i	Tag of a bucket with the bucket number $i \in [m]$
AI	Authentication information
f	Countable function
C	The array contains the number of values in each bucket
S	The array contains the location of the result values in the buckets
VO	Verification object
V_f	Result set

size partitions (For example the domain $D = [0, 100]$ is divide to 5 equal-width partitions $[0, 20)$, $[20, 40)$, $[40, 60)$, $[60, 80)$, $[80, 100]$). In the equal-depth partitioning technique, the domain of values is divided into m partitions such that given S sample of values; each partition contains approximately $\frac{S}{m}$ number of values.

A. LABELED PROGRAMS

We recall the notion of labeled programs introduced by Gennaro and Wichs [22]. This notion is extended by Fiore *et al.* [2] for programs with inputs from more than one source.

The input program in the most homomorphic authenticator schemes is modeled as a labeled program. A labeled program $\mathcal{P} = \langle f, (\ell_1, \ldots, \ell n)\rangle$ consists of an n variate function $f : \mathcal{M}^n \rightarrow \mathcal{M}^{n'}$ and a set of labels $\ell_1, \ldots, \ell_n \in \{0, 1\}^*$. Labeled programs $\mathcal{P}_1$,…,$\mathcal{P}_N$ can be composed using a function $G : \mathcal{M}^N \rightarrow \mathcal{M}^{N'}$. The inputs of the composed program $\mathcal{P}^* = G(\mathcal{P}_1, \ldots, \mathcal{P}_N)$ are all distinct inputs of the labeled programs $\mathcal{P}_1, \ldots, \mathcal{P}_N$ (the inputs with the same labels are grouped together). For multi-key homomorphic authenticator schemes [2], the identity of the source (i.e. id) are added to the labels such that $\ell = (id, \tau)$ where τ is a tag. Actually, τ is a string to determine a data item in a set of inputs generated by the user with identity id.

Multi-Labeled Programs [41]: A multi-labeled program $\mathcal{P}_\Delta$ is a pair $(\mathcal{P}, \Delta)$, where $\mathcal{P}$ is a labeled program and $\Delta \in \{0, 1\}^*$ is a dataset identifier. Multi-labeled programs $\mathcal{P}_{1,\Delta}, \mathcal{P}_{2,\Delta}, \ldots, \mathcal{P}_{N,\Delta}$ with the same Δ, can also be composed using a function $G : \mathcal{M}^N \rightarrow \mathcal{M}^{N'}$ as $\mathcal{P}^*_\Delta = G(\mathcal{P}_1, \ldots, \mathcal{P}_N)$.

Definition 2 (Well-Defined Multi-Labeled Programs [20]): Let c be a constant value, $\Delta \in \{0, 1\}^*$ is a dataset identifier and L_Δ is a list of label and message pairs. A multi-labeled program $\mathcal{P}_\Delta = \langle f, (\ell_1, \ldots, \ell_n)\rangle$ is well defined with respect to the list L_Δ, if one of the following two conditions holds:

- for each label $(\ell_i, .) \in L_\Delta$, there exists message value v_i such that $(\ell_i, v_i) \in L_\Delta$.
- if there exist labels ℓ_i with $i \in [n]$, such that $(l_i, .) \notin L_\Delta$, then $f(v_{j(\ell_j, v_j) \in L_\Delta} \cup v'_{j(\ell_j, v'_j) \notin L_\Delta}) = c$, for all $v'_j \in \mathcal{M}$.

V. COUNTABLE QUERY AUTHENTICATION SCHEME

We define the syntax, correctness, security, and succinctness of the countable query authentication scheme or CQAS according to the schemes proposed by Boneh *et al.* [42] as follows.

*Definition 3 (*CQAS *Syntax):* An Countable Query Authentication Scheme is a collection of four polynomial-time algorithms $\Pi =$ (Setup, AlGen, Comb, Vrfy).

- $(PP, SP, VK) \leftarrow \mathsf{Setup}(1^\lambda, n, n_s)$
  Setup is a probabilistic polynomial-time (PPT) algorithm that is run by the owner of the system. It takes as input the security parameter λ, the input size n, and the number of sources n_s, and returns as output the public parameter PP, the secret parameters $SP = \langle (sk_1, \ldots, sk_{n_s}), ssk\rangle$ which contains the secret-key of all sources and a shared secret key between the trusted sources, and the verification-key VK. The public parameter is the default input to all other algorithms. This parameter determines a message space $\mathcal{M}$, an authenticator space $\mathcal{Y}$, a label space $\mathcal{L} = \mathcal{ID} \times \mathcal{T}$ (where $\mathcal{ID}$ is an identity space and $\mathcal{T}$ is a tag space), a set of admissible functions $F : \mathcal{M}^n \rightarrow \mathcal{M}^k$ with $k \leq n$, and other public values required in the constructions.
- $AI \leftarrow \mathsf{AlGen}_{ssk}(sk, v, \Delta, \ell = (id, \tau))$
  AlGen is a PPT algorithm, which is run by each source. It uses the shared secret-key ssk and takes as input the secret-key of the source sk, the message value v, the dataset identifier Δ, and a label of the message $\ell \in \mathcal{L}$, and returns as output the authentication information AI. It should be noted that for each newly generated value (or a set of values), a new dataset identifier is created.
- $VO := \mathsf{Comb}(f, \Delta, A)$
  Comb is a deterministic polynomial-time algorithm, which is run by the aggregator. It takes as input, the countable (aggregate or range) function $f \in F$, the set of authentication information $A = \{AI_1, \ldots, AI_n\}$ corresponding to the labels and values pairs $\{(\ell_i, v_i)\}_{i=1}^n$, and returns as output the verification object VO which validates the correctness of the result set $V_f = f(v_1, \ldots, v_n)$.
- $b := \mathsf{Vrfy}(\mathcal{P}, VK, V_f, VO, \Delta)$
  Vrfy is a deterministic polynomial-time algorithm that is run by the user/verifier. It takes as input, a labeled program $\mathcal{P}$ corresponding to the countable (aggregate or range) function f, the verification key VK, the result set V_f, the verification object VO and the dataset identifier Δ, and returns as output a bit b while $b = 1$ meaning a valid result and $b = 0$ meaning an invalid result.

*Definition 4 (*CQAS *Correctness):* We say CQAS is correct if for every sufficiently large security parameter λ, any

input size n, and any number of sources $n_s \in \mathbb{N}$ (polynomial in the security parameter λ), it has two properties:

1) *Authentication Correctness:* Authentication correctness means that all authentication information generated by all sources, are verified correctly. More precisely, let λ is a security parameter, n is the input size of the function f, and n_s is the number of sources contributed in the computation. Let $\mathsf{Setup}(1^\lambda, n, n_s)$ generates PP, $SP = \langle (sk_1, \ldots, sk_{n_s}), ssk \rangle$, and VK. Let for every source with identifier $id \in \mathcal{ID}$ and any value $v \in \mathcal{M}$ in any dataset $\Delta \in \mathbb{Z}_N$, and any label $\ell = (id, \tau) \in \mathcal{L}$, $AI \in \mathcal{Y}$ be the authentication information generated by running the algorithm $\mathsf{AlGen}_{ssk}(sk_{id}, v, \Delta, \ell)$. We say CQAS has the authentication correctness property if we have $\mathsf{Vrfy}(\mathcal{I}, VK, v, AI, \Delta) = 1$ where $\mathcal{I} = \langle I, \ell \rangle$ is an identity program such that $I(v) = v$.
2) *Evaluation Correctness:* Evaluation correctness means that the result of executing an admissible countable function on a set of authenticated values, is correct. In more formal, let λ is a security parameter, n is the input size of the function f, and n_s is the number of sources contributed in the computation. Let $\mathsf{Setup}(1^\lambda, n, n_s)$ generates PP, $SP = \langle (sk_1, \ldots, sk_{n_s}), ssk \rangle$, and VK. For a fixed dataset identifier $\Delta \in \mathbb{Z}_N$, a labeled program $\mathcal{P} = \langle f, (\ell_1, \ldots, \ell_\omega) \rangle$ with $\omega > 0$, corresponding to the function f, and all label, message and authenticator triples $\{(\ell_i = (id_i, \tau_i), v_i, AI_i)\}_{i=1}^{\omega} \in \mathcal{L} \times \mathcal{M} \times \mathcal{Y}$ where $AI_i = \mathsf{AlGen}_{ssk}(sk_{id_i}, v_i, \Delta, \ell_i)$, if for each i, $\mathsf{Vrfy}(\mathcal{I}, VK, v_i, AI_i, \Delta) = 1$, then we have $\mathsf{Vrfy}(\mathcal{P}, VK, V_f, VO, \Delta) = 1$ where $VO := \mathsf{Comb}(f, \Delta, \{AI_i\}_{i=1}^{\omega})$ and $V_f = f(v_1, , \ldots, v_\omega)$.

*Definition 5 (*CQAS *Security):* Let $\Pi = (\mathsf{Setup}, \mathsf{AlGen}, \mathsf{Comb}, \mathsf{Vrfy})$ be an countable query authentication scheme, and $\mathcal{A}$ be a probabilistic polynomial-time adversary. For defining the security of the scheme, we define a game $\exp^{\mathsf{CQAS}}_{\mathcal{A},\Pi}(\lambda)$ between the adversary $\mathcal{A}$ and the challenger $\mathcal{C}$ as follows:

- We assume the adversary $\mathcal{A}$ knows the labeled program $\mathcal{P} = \langle f, (\ell_1, \ldots, \ell_n) \rangle$, the security parameter λ, and chooses n_s, which is the number of sources.
- $\mathcal{C}$ runs $(PP, SP, VK) \leftarrow \mathsf{Setup}(1^\lambda, n, n_s)$ and sends the public parameter PP to $\mathcal{A}$.
- $\mathcal{A}$ is given the oracle access to the AlGen_{ssk} algorithm ($\mathcal{O}_{\mathsf{AlGen}}$) and can adaptively send requests of the for (ℓ, v, Δ) to $\mathcal{O}_{\mathsf{AlGen}}$. $\mathcal{C}$ assigns a set $V(\Delta)$ for each dataset Δ. This set contains tuples like (ℓ, v). When $\mathcal{A}$ sends his/her request, $\mathcal{C}$ examines the following conditions and sends AI to $\mathcal{A}$:
  - If $\mathcal{A}$ requests Δ for the first time, $\mathcal{C}$ defines $V(\Delta)$ and computes AI, and adds (ℓ, v) to $V(\Delta)$.
  - If $\mathcal{A}$ requests ℓ from the dataset Δ, for the first time, $\mathcal{C}$ computes AI and adds (ℓ, v) to $V(\Delta)$.
  - If $\mathcal{A}$ requests ℓ for the dataset Δ for the second or more time, $\mathcal{C}$ ignores it.

  Finally, $\mathcal{A}$ outputs a tuple $(\mathcal{P}^*, \Delta^*, V_f^*, VO^*)$ where $\mathcal{P}^* = \langle f^*, (\ell_1{}^*, \ldots, \ell_\omega{}^*) \rangle$.
- The adversary $\mathcal{A}$ wins the game, if and only if $\mathsf{Vrfy}(\mathcal{P}^*, VK, V_f^*, VO^*, \Delta^*) = 1$ and at least one of the following conditions hold:
  1) *Type-I Forgery:* Δ^* is a new dataset that is not queried before.
  2) *Type-II Forgery:* Δ^* is not a new dataset, $\mathcal{P}^*$ is well defined with respect to L_{Δ^*}, and $V_f^* \neq f^*(V(\Delta^*))$,
  3) *Type-III Forgery:* Δ^* is not a new dataset and $\mathcal{P}^*$ is not well defined with respect to L_{Δ^*}. In other words, the inputs of f^* contains a value v such that $(\ell, v) \notin V(\Delta^*)$. Namely, there is a value or identifier in $V(\Delta^*)$ which is not queried before and not appeared in L_{Δ^*}.

We say that scheme Π is secure if for all probabilistic polynomial-time adversaries $\mathcal{A}$, there is a negligible function negl(.) such that:

$$\Pr[\exp^{\mathsf{CQAS}}_{\mathcal{A},\Pi}(\lambda) = 1] \leq \mathrm{negl}(\lambda)$$

*Definition 6 (*CQAS *Succinctness):* Let λ is a security parameter, n is the input size of the function f, and n_s is the number of sources contributed in the computation. Let PP, $SP = \langle (sk_1, \ldots, sk_{n_s}), ssk \rangle$, and VK are generated by running the $\mathsf{Setup}(1^\lambda, n, n_s)$ algorithm. We say a countable query authentication scheme has the succinctness property, if and only if the size of the verification object VO, which is the output of the $\mathsf{Comb}(f, \Delta, \{AI_i\}_{i=1}^{n})$ algorithm such that each AI_i (for the value v_i with a label $\ell_i = (id_i, \tau_i)$) is generated by running $\mathsf{AlGen}_{ssk}(sk_{id_i}, v_i, \Delta, \ell_i)$, logarithmically depends on n, but possibly linearly in n_s. Namely, there is a fixed polynomial p such that $|VO| = p(\lambda, n_s, \log(n))$.

VI. LINEAR QUERY AUTHENTICATION (LQA)

For authenticating countable (aggregate or range) queries, we use the linear query authentication scheme as a building block. Linear functions have the general form $f(v_1, v_2, \ldots, v_n) = \sum_{i=1}^{n} \alpha_i v_i$. For authenticating linear queries, we use a homomorphic RSA-based signature which is secure in the random oracle model under the standard RSA assumption.

This signature is based on the homomorphic signature introduced by Gennaro *et al.* [4] for the network coding application. In their signature, a source splits a file into multiple parts, each part is a vector of values $(v_1, \ldots, v_n)$, signs each vector and sends them via the network to the receivers. The middle nodes in the network combine the received vectors (signatures) and send the results to the next node. Finally, the designate receiver reconstructs the file from the combined vectors and checks whether it is correct or not. They have considered the following assumptions in their constructions:

- The public-key is $(N, e, g_1, g_2, \ldots, g_n)$ where the pair (N, e) is the RSA public-key and $(g_1, g_3, \ldots, g_n)$ are random generators of a cyclic group QR_N (quadratic residues modulo N) where N is the product of two safe primes.
- The secret-key is (N, d), the RSA signature secret-key.

- The homomorphic hash function is defined as $\mathrm{H}: \mathbb{Z}^n \rightarrow QR_N$ such that $\mathrm{H}(v_1, v_2, \ldots, v_n) = \prod_{i=1}^{n} g_i^{v_i}$.

The definition of the basic homomorphic RSA-based signature proposed by Gennaro *et al.* [43] is as follows.

Definition 7 (Basic Homomorphic RSA-Based Signature [4]:) The homomorphic RSA-based signature of vector of values is defined as $sign(v_1, \ldots, v_n) = \left(\mathrm{H}(v_1, .., v_n)\right)^d \ mod\ N$. The homomorphic property is obvious such that $sign(\alpha v + \beta w) = sign(v)^{\alpha} sign(w)^{\beta}$ where $v = (v_1, v_2, \ldots, v_n)$ and $w = (w_1, w_2, \ldots, w_n)$.

For authenticating the values generated by each sources, we use the basic homomorphic RSA-based signature and make some changes in it. We choose the hash fuction $\mathrm{H}(v) = rg^v$ such that $sign(v) = \left(rg^v\right)^d \ mod\ N$, where $r \xleftarrow{\$} QR_N$ is a random value, and g is a random generator of a cyclic group QR_N (quadratic residue modulus N).

A. LQA CONSTRUCTION

According to the scheme $\Pi = (\mathsf{Setup}, \mathsf{AIGen}, \mathsf{Comb}, \mathsf{Vrfy})$ defined in Definition 3 and the RSA-based homomorphic signature proposed by Gennaro *et al.* [4], we define a lightweight linear query authentication (LQA) construction for authenticating the linear queries. This construction is used as a building block for the construction proposed for the countable queries authentication (CQA). It should be noted that our construction for authenticating countable queries is a generic construction and its security, correctness, and succinctness depend on the security, correctness, and succinctness of the LQA scheme. Therefore, any linear multiplicative homomorphic signature scheme that $sign(\alpha) \times sign(\beta) = sign(\alpha + \beta)$, and is compatible with Definition 3, can also be used in it. Let $\mathcal{P} = \langle f, (\ell_1, \ldots, \ell_n)\rangle$ is a labeled program for the linear function $f(v_1, v_2, \ldots, v_n) = \sum_{i=1}^{n} \alpha_i v_i$ where $\ell_i = (id_i, i)$.

1) $(PP, SP, VK) \leftarrow \mathsf{Setup}(1^{\lambda}, n, n_s)$: At the setup of the system, in the off-line phase, the Setup algorithm is run by the owner of the system. This algorithm determines the following parameters:

   Public parameters $PP = \langle (N, e), QR_N, g\rangle$:

   - (N, e) is an RSA public-key where N is the product of two primes p and q. e is a prime number such that $e > n\alpha \mathrm{M}$ where α is the maximum coefficient in the admissible linear functions and M is the maximum value that can be appeared in the exponent of g in creating authenticators in each source (e can be chosen to be a number of low Hamming weight to have an acceptable efficiency).
   - QR_N is the description of the cyclic group quadratic residue modulus N and g is a random generator of this group.

   Secret parameters $SP = \langle \left(sk_{id_1}, \ldots, sk_{id_{n_s}}\right), ssk\rangle$:

   - $\left(sk_{id_1}, \ldots, sk_{id_{n_s}}\right)$ is the secret key of all sources. For each source with identifier $id_i \in \mathcal{ID}$ such that $i \in [n_s]$, a random secret-key $sk_{id_i} \in_R \{0, 1\}^{\lambda}$ is opted.
   - $ssk = \langle N, d\rangle$ is the RSA private key that is shared between the trusted sources.

   Verification key $VK = (sk_{id_1}, \ldots, sk_{id_{n_s}})$.

   The public parameters are available for all entities in the system. In each source machine, a unique identifier, the secret key of the source, and the shared secret keys are set. We assume that the verification key is sent to the verifier via a secure channel.

2) $AI_i \leftarrow \mathsf{AIGen}_{ssk}(sk_{id}, v, \Delta, \ell)$: In the AIGen algorithm, for the newly generated value v belonging to the dataset Δ and the label (id, i) where $i \in [n]$, the source with identifier id, generates an authentication information AI_i as follows:

$$\sigma_i = \left(r_{id} \cdot g^v\right)^d \ mod\ N \tag{1}$$

   where $r_{id} = g^{sk_{id}\mathrm{h}(\Delta||i)} \ mod\ N$, and $\mathrm{h}: \{0, 1\}^* \rightarrow \mathbb{Z}_N$ is a hash function. The dataset identifier Δ can be a time-stamp or a synchronized sequence number between all sources. Finally, the authentication information $AI_i = \langle \ell, \sigma_i\rangle$ is sent to the aggregator.

3) $VO := \mathsf{Comb}(f, \Delta, \{AI_i\}_{i=1}^{n})$: The aggregator collects the authenticated values (belonging to the dataset Δ) and parses each authenticator AI_i with $i \in [n]$ to $\langle \ell = (id_i, i), \sigma_i\rangle$. Then, the aggregator executes the combine algorithm. This algorithm outputs the verification object $VO = \sigma'$ for the result $V_f = \sum_{i=1}^{n} \alpha_i v_i$ as follows:

$$\sigma' = \prod_{i=1}^{n} \sigma_i^{\alpha_i} \ mod\ N \tag{2}$$

4) $b := \mathsf{Vrfy}(\mathcal{P}, VK, V_f, VO, \Delta)$: The verification algorithm is executed as bellow:

$$F = g^{\sum_{i=1}^{n} sk_{id_i}\mathrm{h}(\Delta||i)\alpha_i}$$
$$(\sigma')^e \stackrel{?}{=} F \cdot g^{V_f} \ mod\ N \tag{3}$$

   It should be noted that the verifier registers the linear function on the aggregator and knows the labels and coefficients. For reducing the computation cost, for each linear function, computing F could be amortized by preprocessing.

B. CORRECTNESS AND SECURITY

For proving the correctness and security of LQA, we present the following theorems.

Theorem 1: LQA is correct.

Proof 1: According to Definition 4, we prove the authentication and evaluation correctness of LQA. We show that if all authenticators are created correctly and are combined according to the Comb algorithm, the verification algorithm outputs one. The complete proof of this theorem is provided in Appendix A.1.

Theorem 2: If the RSA problem is hard relative to the RSA-Gen algorithm and the hash function h is modeled as a random oracle, then the LQA construction is secure.

TABLE 3. Cost of different operations.

Notation	Description	Cost in practice
C_h	cost of the hash function h	0.11 ms
C_{Mexp}	cost of the modular exponentiation	17 ms
C_{Mmul}	cost of the modular multiplication	0.04 ms
C_{Madd}	cost of the modular addition	3 µs

Proof 2: For proving this theorem, we use Definition 5 and show that if the adversary $\mathcal{A}$ can do a forgery, the RSA problem is inverted. The complete proof of this theorem is provided in Appendix A.2.

C. SUCCINCTNESS AND PERFORMANCE

If the verifier receives all values and their authentication information (standard RSA digital signatures) directly from all sources, the communication overhead is $O\big(n(\lambda + \log N)\big)$ where n is the input size, λ is the maximum length of the input values, and N is the RSA modulus. In this construction, the size of the verification object is constant and it is equal to the size of the combined signature, which is at most $\log N$ bits. So, this construction has succinctness property as well.

The cost of different operations, which are obtained by some experiments in practice, is shown in Table 3.[1] In each source, the cost of creating an authenticator for each input value is $C_{src} = C_h + C_{Madd} + 2C_{Mmul} + C_{Mexp}$. The aggregator computes n modular exponentiations and multiplications and its cost is $C_{agg} = n(C_{Mmul} + C_{Mexp})$. The overhead in the verifier side in the amortized sense is $C_{vrf} = 2C_{Mexp} + C_{Mmul}$ and it is constant for different number of sources.

If the verifier receives all values and their standard RSA digital signatures directly from all sources, checks the validity of all authentication information, and computes the linear function, the computation cost will be $C_{vrf} = O(n(C_h + C_{Mexp}) + C_f)$ where $C_f = O(n)$ is the cost of computing a linear function.

VII. COUNTABLE QUERY AUTHENTICATION (CQA)

Let $\mathcal{P} = \langle f, (\ell_1, \ldots, \ell_n)\rangle$ is a labeled program for the countable aggregate function f. In countable aggregate functions such as median, max/min, range, and top-k/first-k, the result $V_f = f(v_1, \ldots, v_n)$ contains a subset of the values in the inputs ($V_f \subseteq \{v_1, \ldots, v_n\}$), and the order of the values is also important. In the median function, if the number of the input values be even, we assume without loss of generality, two middle values are returned in V_f.

The proposed construction for authenticating countable queries, named Countable Query Authentication or CQA, is based on the scheme $\Pi = (\mathsf{Setup}, \mathsf{AIGen}, \mathsf{Comb}, \mathsf{Vrfy})$ defined in Definition 3. The algorithms of this construction uses the algorithms of the LQA construction, a bucket partitioning algorithm, and a hash function $h : \{0,1\}^* \to \mathbb{Z}_N$. We use the bucket partitioning for determining the location of a value among the other values. The partitioning algorithm (such as the equal-width and equal-size algorithms) is used in the setup phase.

1) $(PP, SP, VK) \leftarrow \mathsf{Setup}(1^\lambda, n, n_s, R)$: The setup algorithm has some differences with the previous construction and it involves some other parameters related to the partitioning. It takes the domain $R = [\alpha, \beta]$ of the input values as the last argument and involves the following steps:

   Step 1: Execute the setup algorithm of LQA construction as

   $$(PP', SP', VK') \leftarrow LQA.\mathsf{Setup}(1^\lambda, n, n_s)$$

   Step 2: Execute the bucket partitioning algorithm on R as

   $$P \leftarrow Bucket\_Partitioning(R)$$

   P contains the number of buckets m, the domain of the values $[R_1, R_m]$, the intervals or buckets $[R_0, R_1)$, $[R_1, R_2), \ldots, [R_{m-1}, R_m]$, and random tags $\{t_1, \ldots, t_m\}$ of all buckets where $\forall i \in [m]$, $t_i \in_R \{0,1\}^\lambda$.

   Step 3: Set the parameters $PP = (PP', P)$, $SP = SP'$ and $VK = VK'$.

2) $AI_i \leftarrow \mathsf{AIGen}_{ssk}(sk_{id}, v, \Delta, \ell)$: In countable queries such as the median, the order of the values is important. So the verifier needs to verify the results and their orders. For this purpose, the AIGen algorithm creates $AI_i = \langle \ell = (id, i), \sigma_{B,i}, \sigma_{L,i}\rangle$ that contains two signatures for a given value v. One for authenticating the bucket of the value and another for authenticating the location of the value in the bucket. For authenticating the bucket of v, the bucket that the value belongs to it is found. Let t_j is the tag of this bucket where $j \in [m]$. The signature of the bucket is created as follows.

   $$\sigma_{B,i} = LQA.\mathsf{AIGen}_{ssk}(sk_{id}, h(t_j), \Delta, \ell = (id, i))$$

   This authenticator helps the verifier to investigate that the buckets of the results are correct or not. For authenticating the location of the value in the bucket, another authenticator is created. Let $[R_j, R_{j+1})$ is the interval of the bucket that v belongs to it. The source or signer determines the distance of the value from the beginning of the bucket as $dst = (v - R_j + 1)$ and then creates the following location signature.

   $$\sigma_{L,i} = LQA.\mathsf{AIGen}_{ssk}(sk_{id}, h(t_j||dst), \Delta, \ell = (id, i))$$

   Finally, the authentication information generated by the source i, for the value v_i is the tuple $AI_i = \langle \ell, \sigma_{B,i}, \sigma_{L,i}\rangle$.

3) $VO := \mathsf{Comb}(f, \Delta, \{AI_i\}_{i=1}^n)$: The aggregator gathers the values and authentication information sent by different sources and executes the countable labeled program $\mathcal{P} = \langle f, (\ell_1, \ldots, \ell_n)\rangle$ on the set of the label and value pairs $V(\Delta) = \{(\ell_1, v_1), \ldots, (\ell_n, v_n)\}$ belonging to the same dataset Δ, and finds the results such that $V_f = f(v_1, \ldots, v_n)$. For creating VO, the aggregator follows the steps described in Algorithm 1.

[1] The cost of different operations has been computed in the experimental environment explained in Section IX. Each operation was executed 1000 times and the average time of the operation was computed.

Algorithm 1 Combine Algorithm for Countable Queries

Step 0: Let $V(\Delta) = \{(\ell_1, v_1), \ldots, (\ell_n, v_n)\}$ is the set of the value and label pairs gathered from all sources, belonging to the same dataset Δ. Execute the countable aggregate function f on the input values and find the results such that $V_f = f(v_1, \ldots, v_n)$ where $V_f \subseteq \{v_1, \ldots, v_n\}$.

Step 1: Count the number of values in each bucket and create the vector $C = \langle c_1, c_2, \ldots, c_m \rangle$ where c_i is the number of values in the i-th bucket.

Step 2: Parse each authentication information AI_i to $((id_i, i), \sigma_{B,i}, \sigma_{L,i})$. Let $\mathcal{P}_C = \langle f_C, (\ell_1, \ldots., \ell_n) \rangle$ be a labeled program for the linear function $f_C(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i$ such that $x_i \in \mathbb{Z}_N$. All of the coefficients of f_C are set to one, and its labels are the same as the ones used in $\mathcal{P}$ (i.e. the labeled program of the countable aggregate function). Use the Comb algorithm of the LQA construction to combine the signatures of buckets as bellow:

$$\sigma_B = LQA.\mathsf{Comb}(f_C, \Delta, \{(\ell_i, \sigma_{B,i})\}_{i=1}^{n})$$

Step 3: Let *get_bucket* is a function that takes a value and partitioning information as inputs and outputs $bid \in [m]$ as the identifier of the bucket that the given value falls in it (this function is defined in Appendix B). Let the server finds the following sets:

- $B = \{t_j | \exists v \in V_f, j = get\_bucket(P, v)\}$ which contains the tag of all buckets that have at least one result value in it.
- $AL = \{(\ell_i, \sigma_{L,i}) | (\ell_i, v_i) \in V(\Delta) \wedge t_{get\_bucket(P,v_i)} \in B\}$ which contains the authentication information for the locations of the values fallen in the buckets that have at least one result value in it.
- $AB = \{(\ell_i, \sigma_{B,i}) | (\ell_i, v_i) \in V(\Delta) \wedge t_{get\_bucket(P,v_i)} \in B\}$ which contains the authentication information for the buckets of the values fallen in the buckets that have at least one result value in it.

Step 4: Define the set S such that $S = \{S_i | i \in [m] \wedge t_i \in B\}$. Each $S_i \in S$ (which is related to the bucket with the tag t_i) is defined as a vector $S_i = \{(j, s_j) | 0 \leq j < len \wedge 1 \leq s_j \leq \log(n)\}$ where $len = R_{i+1} - R_i + 1$ is the length of the bucket with the tag t_i and s_j denotes the number of values that their distance from the beginning of the bucket (i.e. R_i) is equal to j.

Step 5: Create a labeled program $\mathcal{P}_{AL} = \langle f_{AL}, (\{\ell_i\}_{(\ell_i, \sigma_{B,i}) \in AL}) \rangle$ for the linear function $f_{AL}(x_1, \ldots, x_{|AL|}) = \sum_{i=1}^{|AL|} x_i$; such that $x_i \in \mathbb{Z}_N$ and all of its coefficients are set to one. Combine the locations signatures using the Comb algorithm in the LQA construction as follows.

$$\sigma_L = LQA.\mathsf{Comb}(f_{AL}, \Delta, AL)$$

Step 6: Create a labeled program $\mathcal{P}_{AB} = \langle f_{AB}, (\{\ell_i\}_{(\ell_i, \sigma_{B,i}) \in AB}) \rangle$ for the linear function $f_{AB}(x_1, \ldots, x_{|AB|}) = \sum_{i=1}^{|AB|} x_i$; such that $x_i \in \mathbb{Z}_N$ and all of its coefficients are set to one. Combine these signatures using the Comb algorithm in the LQA construction as follows.

$$\sigma_{B'} = LQA.\mathsf{Comb}(f_{AB}, \Delta, AB)$$

Step 7: The verification object for verifying the result V_f is the tuple $VO = \langle C, S, \sigma_B, \sigma_L, \sigma_{B'} \rangle$, which is sent to the verifier.

4) $b := \mathsf{Vrfy}(\mathcal{P}, VK, V_f, VO, \Delta)$: When the verifier receives the results V_f and the verification object VO, parses the verification object to a tuple $\langle C, S, \sigma_B, \sigma_L, \sigma_{B'} \rangle$, and executes the algorithm Vrfy. The verifier needs to follow the steps shown in Algorithm 2 for verifying the correctness of the results.

A. CORRECTNESS AND SECURITY

As specified in the previous section, all algorithms of the CQA construction are based on the algorithms defined in the LQA construction. So, we can infer the correctness and security of the CQA construction according to the correctness and security of the LQA construction.

Theorem 3: If LQA is correct, then CQA is correct as well.

Proof 3: The proof of this theorem is obvious. In the CQA construction, both $\sigma_{B,i}$ and $\sigma_{L,i}$ are computed similar to the authenticators created in the LQA construction. Therefore, the proof of the authentication and evaluation correctness of CQA are the same as the proof provided for LQA with some differences that in these signatures, instead of the data values, we use the hash of some values and instead of the coefficients in the linear functions, the values in the sets C and S are used.

Theorem 4: If LQA is secure, then CQA is secure as well.

Proof 4: The proof of this theorem is also obvious. In the Step 2 and Step 3 of the verification algorithm in the CQA construction, we use the LQA.Vrfy algorithm to verify the signatures created using the LQA.Comb algorithm.

Algorithm 2 Verification Algorithm for Countable Queries

Step 1: Using the result values V_f, the vectors $C = (c_1, c_2, \ldots, c_m)$, $S = (S_1, \ldots, S_k)$, and considering the labeled program $\mathcal{P} = \langle f, (\ell_1, \ldots, \ell_n)\rangle$ for the aggregate function f, the verifier calls f_Validate function defined in Appendix B (f is replaced with the name of the aggregate function). This function checks that the values in the set V_f are valid or not. For example, for the max query, the last bucket that the number of the values fallen in it is not zero, is the bucket containing the result and the maximum value should be the last value in this bucket.

Step 2: If the results were compatible with C and S, the verifier checks the validity of the signature σ_B as follows.

1) The verifier computes $V' = \sum_{j=1}^{m} c_j h(t_j)$ using the vector C.
2) Then, the verifier executes the Vrfy algorithm in the LQA construction as follows.

$$b := LQA.\mathsf{Vrfy}(\mathcal{P}_C, VK, V', \sigma_B, \Delta)$$

where $\mathcal{P}_C = \langle f_C, (\ell_1, \ldots, \ell_n)\rangle$ is a labeled program for the linear function $f_C(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i$ such that $x_i \in \mathcal{M}$, all coefficients of this linear function are set to one, and its labels are the same as the labels used in $\mathcal{P}$ (the main aggregate program). In other words, the verifier checks the following equation is correct or not.

$$\sigma_B^e \stackrel{?}{=} \left(F \cdot g^{V'}\right) \bmod N$$

where $F = g^{\sum\limits_{(id_i, i)\in\mathcal{P}_C} sk_{id_i} h(\Delta||i)}$.

Step 3: If the previous two steps are passed, the verifier checks that σ_L is correct or not. For checking σ_L, two steps are needed:

1) The verifier computes $V'' = \sum_{S_i\in S}\big(-c_i \mathrm{h}(t_i) + \sum_{(j,s_j)\in S_i}(s_j.h(t_i||j))\big)$.
2) Then, executes the Vrfy algorithm in the LQA construction as follows.

$$b := LQA.\mathsf{Vrfy}(\mathcal{P}_{cst}, VK = \{0\}, V'', \sigma_L \cdot ({\sigma_{B'}}^{-1}), \Delta)$$

where $\mathcal{P}_{cst} = \langle f_{cst}, 0\rangle$ is a labeled program for the constant function $f_{cst} = c$. In other words, the verifier checks the following equation is correct or not.

$$(\sigma_L \cdot ({\sigma_{B'}}^{-1}))^e \stackrel{?}{=} g^{V''} \bmod N$$

For verifying σ_L, we use $\sigma_{B'}$ and compute $V'' = \sum_{S_i\in S}\big(-c_i \mathrm{h}(t_i) + \sum_{(j,s_j)\in S_i}(s_j.h(t_i||j))\big)$. Since the correctness of $\sum_{S_i\in S} c_i \mathrm{h}(t_i)$ is verified in Step 2 and in $\sigma_{B'}$ this summation is used, we can use $\sigma_{B'}$ and $g^{\sum_{S_i\in S} -c_i \mathrm{h}(t_i)}$ for verifying the identity of the sources that have at least a value in the result buckets. Therefore, since LQA is secure, CQA is secure as well.

B. SUCCINCTNESS AND PERFORMANCE

In the CQA construction, the verification object is $\langle C, S, \sigma_B, \sigma_L, \sigma_{B'}\rangle$ and its size is $O(Lm\log(n) + 3\log N)$ where $L \in \mathbb{Z}_N$ is the maximum size of a bucket length, n is the input size, and N is the RSA modulus.

If all sources directly send their values and authentication information (Standard RSA digital signatures) to the verifier, the overhead is $O(n(\lambda + \log N))$ where λ is the maximum size of the input values. We assume $Lm \ll n$, therefore, our construction has the succinctness property.

Let $C_f = O(k_1 n)$ is the cost of computing the function f with a constant coefficient k_1 which differs according to the aggregate function. The computation costs in each entity in our model, according to Table 3, are as follows:

- Each source finds the bucket of a value in $O(m)$ and creates the signatures in $C_{sig} = 3\mathrm{C_h} + 3\mathrm{C}_{Mmul} + 2\mathrm{C}_{Madd} + 2\mathrm{C}_{Mexp}$. So, the cost in each source is $\mathrm{C}_{src} = O(m + C_{sig})$.
- The aggregator computes the function f in C_f, and follows the steps mentioned in Algorithm 1. Totally, the cost in the aggregator side is $\mathrm{C}_{agg} = O\big(n(3\mathrm{C}_{Mmul} + m) + C_f\big)$ where n is the input size.
- The cost in the verifier side consists of the costs of all steps mentioned in Algorithm 2. In Step 1, checking the validity of the values received from the aggregator is done in $C_{validity} = O(k_2 m + L)$ where $1 \le k_2 \le n$ depending the type of the query and the size of the result set (In the Max/Min and Median queries $k_2 = 1$, in the Top-k/First-k queries $k_2 = k$, and in the range queries, it is at most n, when all the inputs are in the result set). We assume the costs of computing hashes and F are amortized by preprocessing. The cost of Step 2 is $O\big(m(\mathrm{C}_{Mmul} + \mathrm{C}_{add}) + 2\mathrm{C}_{Mexp}\big)$. The cost of final step

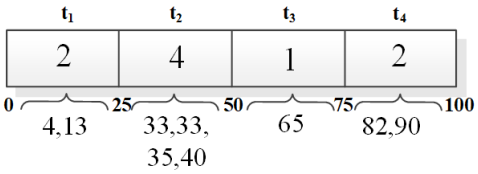

FIGURE 2. Partitioning of the input values.

is $O\big(mL(\mathrm{C}_{Madd} + \mathrm{C}_{Mmul}) + \mathrm{C}_{Mmul} + 2\mathrm{C}_{Mexp}\big)$. Overall, the cost in the verifier side is $\mathrm{C}_{vrf} = O\big(m(L(\mathrm{C}_{Madd} + \mathrm{C}_{Mmul}) + k_2) + 4\mathrm{C}_{Mexp}\big)$.

If all sources directly send their values and standard RSA digital signatures to the verifier, the computation cost in the verifier side would be $\mathrm{C}_{vrf} = O(n \times (\mathrm{C}_h + \mathrm{C}_{Mexp}) + C_f)$. It should be noted that C_{Mexp} and C_h are the most expensive operations in Table 3.

C. CASE STUDY: MEDIAN QUERY AUTHENTICATION

For a better understanding, we explain the construction proposed for authenticating countable queries with an example for median queries. A median function which is run by the aggregator is formally defined in Definition 8.

Definition 8 (Median of a Set [44]): Let $X = \{x_1, x_2, \ldots, x_n\}$ be a set of numbers and $S(x) = \sum_{i=1}^{n} |x - x_i|$. The real value x that minimize $S(x)$, is the median of X. In other word, if the values in the set X are sorted such that $x_1 \leq x_2 \leq \ldots \leq x_n$ we have

- if n is odd, namely $n = 2k - 1$ with $k \in \mathbb{N}$, then the median value is x_k.
- if n is even, namely $n = 2k$ with $k \in \mathbb{N}$, then the median value is $x = \frac{x_k + x_{k+1}}{2}$.

Let $n = n_s = 9$, $m = 4$, $R_1 = 0$, $R_4 = 100$ and the bucket intervals are $[1, 25)$, $[25, 50)$, $[50, 75)$, $[75, 100]$ which are partitioned with the equal-width binning algorithm. Let the tags of the buckets are $\{t_1, t_2, t_3, t_4\}$. We assign a label $\ell_i = (i, i)$ to the i-th source. The values created by all sources are {33, 4, 13, 90, 40, 65, 33, 35, 82} at the time-stamp τ (considered as the dataset identifier Δ), respectively. The partitioning of the inputs are shown in Fig. 2.

The signatures created by all sources are shown in Table 4. The aggregator collects the values and the signatures, finds the median value, and follows the steps mentioned in Algorithm 1.

- *Step 0:* $V_f = median(33, 4, 13, 90, 40, 65, 33, 35, 82) = 35$
- *Step 1:* $C = (2, 4, 1, 2)$
- *Step 2:* $\sigma_B = \prod_{i=1}^{9} \sigma_{B,i}$
- *Step 3:* Define the following arrays:
  - $B = \{t_2\}$
  - $AL = \{(\ell_1, \sigma_{L,1}), (\ell_5, \sigma_{L,5}), (\ell_7, \sigma_{L,7}), (\ell_8, \sigma_{L,8})\}$
  - $BL = \{(\ell_1, \sigma_{B,1}), (\ell_5, \sigma_{B,5}), (\ell_7, \sigma_{B,7}), (\ell_8, \sigma_{B,8})\}$
- *Step 4:* Define the array $S = \{S_2\}$ as $S_2 = \{(9, 2), (11, 1), (16, 1)\}$ where each entry in this array determines the number of values in each location of the second bucket.
- *Step 5:* Compute $\sigma_L = (\sigma_{L,1} \,.\, \sigma_{L,5} \,.\, \sigma_{L,7} \,.\, \sigma_{L,8}) \; mod \; N$
- *Step 6:* Compute $\sigma_{B'} = (\sigma_{B,1} \,.\, \sigma_{B,5} \,.\, \sigma_{B,7} \,.\, \sigma_{B,8}) \; mod \; N$

The verification object and the result are sent to the verifier. The verifier follows the steps mentioned in Algorithm 2 and verifies the result.

- *Step 1:* Run the Median_Validate(.) algorithm described in Appendix B. According to the array $C = (2, \mathbf{4}, 1, 2)$ and the median function, the result should be in the second bucket. As we can see $35 \in [25, 50)$. On the other hand, 35 is the third value of the second bucket. As we see $S_2[11] = 1$, two values exist before it, and one value exists after it which is compatible with $C[2]$. So, according to the set C and S, 35 is a valid median value.
- *Step 2:* Compute $V' = (2\mathrm{h}(t_1) + 4\mathrm{h}(t_2) + \mathrm{h}(t_3) + 2\mathrm{h}(t_4))$ and check that

$$\sigma_B{}^e \stackrel{?}{=} (g^{\sum_{i=1}^{9} \mathrm{h}(\tau||i)sk_i} \,.\, g^{V'})$$

- *Step 3:* Compute $V'' = -4\mathrm{h}(t_2) + 2\mathrm{h}(t_2||9) + \mathrm{h}(t_2||11) + \mathrm{h}(t_2||16)$ and check that

$$(\sigma_L \,.\, (\sigma_{B'}{}^{-1}))^e \stackrel{?}{=} g^{V''} \; mod \; N$$

VIII. WINDOW-BASED COUNTABLE QUERY AUTHENTICATION (WCQA)

In some applications like data stream management systems, source nodes generate values in a period and then send them to the aggregator. Each source can sign each value separately or create a combined signature for all values generated in a window, which reduces the communication overhead.

For generating a signature for all values in a window, we made some changes in the algorithms of the CQA construction and define a new construction called Window-based Countable Query Authentication or WCQA.

Let w_{id} with $id \in [n_s]$ is the number of values generated by the source with identifier id, and the generated values and the labels pairs belonging to the dataset Δ are $V_{id}(\Delta) = \{(\ell_{id,1}, v_{id,1}), \ldots, (\ell_{id,w_{id}}, v_{id,w_{id}})\}$.

1) $\langle PP, SP, VK\rangle \leftarrow \mathsf{Setup}(1^\lambda, n, n_s, R, W)$: In WCQA, the Setup algorithm takes additional inputs R and W, where R is the domain of the inputs and $W = (w_1, \ldots, w_{n_s})$ determines the number of values in the window for each sources such that $n = \sum_{i=1}^{n_s} w_i$. We assume the tags of the input values i.e. $(\tau_1, \ldots, \tau_n)$, are assigned to the sources, respectively. For example the array of tags $(\tau_1, \ldots, \tau_{w_1})$ and $(\tau_{w_1+1}, \ldots, \tau_{w_1+w_2})$ are assigned to the sources with identifiers 1 and 2, respectively. The setup algorithm executes $\langle PP', SP', VK'\rangle \leftarrow LQA.\mathsf{Setup}(1^\lambda, n, n_s)$ and $P \leftarrow Bucket\_Partitioning(R)$, then sets $PP = (PP', P, W)$, $SP = SP'$, and $VK = VK'$.
2) $AI \leftarrow \mathsf{AIGen}_{ssk}(sk_{id}, \{v_{id,i}\}_{i=1}^{w_{id}}, \Delta, \ell = (id, [\tau_s, \tau_e]))$: The AIGen algorithm generates σ_B and σ_L for the values belonging to the source with identifier id. It takes as input the secret key of the source, the set of values

TABLE 4. Bucket and location signatures of the generated values.

Source	Bucket Signature	Location Signature
src_1	$\sigma_{B,1} = \left(g^{sk_1 \mathrm{h}(\tau\|\|1)+\mathrm{h}(t_2)}\right)^d \bmod N$	$\sigma_{L,1} = \left(g^{sk_1 \mathrm{h}(\tau\|\|1)+\mathrm{h}(t_2\|\|9)}\right)^d \bmod N$
src_2	$\sigma_{B,2} = \left(g^{sk_2 \mathrm{h}(\tau\|\|2)+\mathrm{h}(t_1)}\right)^d \bmod N$	$\sigma_{L,2} = \left(g^{sk_2 \mathrm{h}(\tau\|\|2)+\mathrm{h}(t_1\|\|4)}\right)^d \bmod N$
src_3	$\sigma_{B,3} = \left(g^{sk_3 \mathrm{h}(\tau\|\|3)+\mathrm{h}(t_1)}\right)^d \bmod N$	$\sigma_{L,3} = \left(g^{sk_3 \mathrm{h}(\tau\|\|3)+\mathrm{h}(t_1\|\|13)}\right)^d \bmod N$
src_4	$\sigma_{B,4} = \left(g^{sk_4 \mathrm{h}(\tau\|\|4)+\mathrm{h}(t_4)}\right)^d \bmod N$	$\sigma_{L,4} = \left(g^{sk_4 \mathrm{h}(\tau\|\|4)+\mathrm{h}(t_4\|\|16)}\right)^d \bmod N$
src_5	$\sigma_{B,5} = \left(g^{sk_5 \mathrm{h}(\tau\|\|5)+\mathrm{h}(t_2)}\right)^d \bmod N$	$\sigma_{L,5} = \left(g^{sk_5 \mathrm{h}(\tau\|\|5)+\mathrm{h}(t_2\|\|16)}\right)^d \bmod N$
src_6	$\sigma_{B,6} = \left(g^{sk_6 \mathrm{h}(\tau\|\|6)+\mathrm{h}(t_3)}\right)^d \bmod N$	$\sigma_{L,6} = \left(g^{sk_6 \mathrm{h}(\tau\|\|6)+\mathrm{h}(t_3\|\|16)}\right)^d \bmod N$
src_7	$\sigma_{B,7} = \left(g^{sk_7 \mathrm{h}(\tau\|\|7)+\mathrm{h}(t_2)}\right)^d \bmod N$	$\sigma_{L,7} = \left(g^{sk_7 \mathrm{h}(\tau\|\|7)+\mathrm{h}(t_2\|\|9)}\right)^d \bmod N$
src_8	$\sigma_{B,8} = \left(g^{sk_8 \mathrm{h}(\tau\|\|8)+\mathrm{h}(t_2)}\right)^d \bmod N$	$\sigma_{L,8} = \left(g^{sk_8 \mathrm{h}(\tau\|\|8)+\mathrm{h}(t_2\|\|11)}\right)^d \bmod N$
src_9	$\sigma_{B,9} = \left(g^{sk_9 \mathrm{h}(\tau\|\|9)+\mathrm{h}(t_4)}\right)^d \bmod N$	$\sigma_{L,9} = \left(g^{sk_9 \mathrm{h}(\tau\|\|9)+\mathrm{h}(t_4\|\|8)}\right)^d \bmod N$

generated in a window, the data set identifier, and the labels of the values shown as an interval for the start and end of the tags. This algorithm determines the buckets of the values and their distances from the beginning of their buckets and computes the following signatures.

$$\sigma_B = \left(g^{sk_{id} \sum_{i=s}^{e} \mathrm{h}(\Delta||\tau_i) + \sum_{i=1}^{w_{id}} \mathrm{h}(t_{id,i})}\right)^d \bmod N$$
$$\sigma_L = \left(g^{sk_{id} \sum_{i=s}^{e} \mathrm{h}(\Delta||\tau_i) + \sum_{i=1}^{w_{id}} \mathrm{h}(t_{id,i}||dst_{id,i})}\right)^d \bmod N \quad (4)$$

where $t_{id,i} = t_{get\_bucket(P,v_{id,i})}$ and $dst_{id,i} = (v_{id,i} - R_{get\_bucket(P,v_{id,i})-1} + 1)$. Then, this source sends the tuple $\langle \ell = (id, [\tau_s, \tau_e]), \sigma_B, \sigma_L \rangle$ as the authentication information to the aggregator.

3) $VO := \mathsf{Comb}(f, \Delta, \{AI_i\}_{i=1}^{n_s})$: The aggregator executes the Comb algorithm on all values, following the same steps explained in Algorithm 1 and creates the verification object $VO = \langle C, S, \sigma_B, \sigma_L, \sigma_{B'} \rangle$. For verifying σ_L, an additional step should be added to the Comb algorithm. In this step, the aggregator computes a hash value h_{LB}. In σ_L, in addition to the values in the result buckets, all of the values generated by the sources that have a value in the result buckets are used. So, this step is for providing the additional information needed for verifying the results.
For each source with identifier id in AL, let $\{v_1, \ldots, v_{z_{id}}\}$ are all values generated by this source but not appeared in the result V_f. Let $\{(t_1, dst_1), \ldots, (t_{z_{id}}, dst_{z_{id}})\}$ determines the buckets of these values and their locations in the buckets. Using this set, the value h_{LB} is computed as follows.

$$\mathrm{h}_{LB} = \sum_{j=1}^{z_{id}} \mathrm{h}(t_j||dst_j) - \sum_{j=1}^{z_{id}} \mathrm{h}(t_j)$$

Finally the aggregator sends the verification object $VO = \langle C, S, \sigma_B, \sigma_L, \sigma_{B'}, \mathrm{h}_{LB} \rangle$ and the result set V_f to the verifier.

4) $b := \mathsf{Vrfy}(\mathcal{P}, VK, V_f, VO, \Delta)$: For verifying the results, the Vrfy algorithm follows the steps explained in Algorithm 2 but the last step of this algorithm is changed as follows.

Step 3: The verifier checks whether σ_L is correct or not. The verifier computes

$$V'' = \mathrm{h}_{LB} + \sum_{S_i \in S} \Big(- c_i \mathrm{h}(t_i) + \sum_{(s_j,j) \in S_i} (s_j.h(t_i||j)) \Big)$$

Then, executes the Vrfy algorithm in the LQA construction as follows.

$$b := LQA.\mathsf{Vrfy}(\mathcal{P}_{cst}, VK = \{0\}, V'', \sigma_L \,.\, ({\sigma_{B'}}^{-1}), \Delta)$$

where $\mathcal{P}_{cst} = \langle f_{cst}, 0 \rangle$ is a labeled program for the constant function $f_{cst} = c$. In other words, the verifier checks the following equation is correct or not.

$$(\sigma_L \,.\, {\sigma_{B'}}^{-1})^e \stackrel{?}{=} g^{V''} \bmod N$$

A. CORRECTNESS AND SECURITY

For proving the correctness and security of WCQA, two theorems are presented.

Theorem 5: WCQA is correct.

Proof 5: The authentication correctness of this construction is similar to the LQA construction with a difference that in this construction in the AlGen algorithm, instead of a value and a label, the sum of w_{id} number of hash values is used (w_{id} is the number of values generated by the source with identifier id). We prove that the evaluation of the authentication information received from all sources is also correct for both signatures created by the Comb algorithm. The complete proof of this theorem is provided in Appendix A.3

Theorem 6: If LQA is secure, then WCQA is secure as well.

Proof 6: The proof of this theorem is obvious according to the proof explained for the security of the CQA construction.

B. SUCCINCTNESS AND PERFORMANCE

In WCQA construction, let n be the total number of inputs. In this construction, the size of the verification object is $O(Lm \log(n) + 4 \log N)$, where $L \in \mathbb{Z}$ is the maximum size of a bucket length, n is the input size, and N is the RSA modulus. If all sources directly send their values and authentication information (standard RSA digital signature) to the verifier,

the overhead is $O(n(\lambda + \log(N)))$. So this construction has also the succinctness property.

Let the maximum length of a bucket is $L \in \mathbb{Z}_N$, $w \in \mathbb{Z}_N$ is the maximum number of values can be generated by a source, and $C_f = O(k_1 n)$ is the cost of computing the function f with constant value k_1 which differs depending on the chosen aggregate function. The computation costs in each entity in our model, according to Table 3, are as follows:

- Each source computes two signatures with the maximum cost of $C_{src} = 3wC_h + (3w + 2)\,C_{Madd} + 2C_{Mexp} + 3C_{Mmul}$.
- The cost in the aggregator is the sum of the cost of the aggregator in the CQA construction, which is $O\big(n(3C_{Mmul} + m) + C_f\big)$, and the cost of the final additional step, which is $O(2w(C_h + C_{Madd}))$. Totally, the cost in the aggregator side is $C_{agg} = O\big(n(3C_{Mmul} + m)) + 2w(C_h + C_{Madd}) + C_f\big)$.
- The verification cost is the same as the verification cost in the CQA construction, which is $O\big(m(L(C_{Madd} + C_{Mmul}) + k_2) + 4C_{Mexp}\big)$.

IX. IMPLEMENTATION AND EXPERIMENTAL RESULTS

We implemented our constructions in Java using JDK 1.8.0_151 and used UniCrypt library [45] for implementing the needed primitives and cryptographic operations. For the hash function h, we used the SHA256 algorithm. We also used the RSA-2048 as the underlying cryptosystem. All entities are deployed on the Ubuntu 15.10 operating system running on a machine with Intel Core i7- 2.7 GHz CPU and 10GB of RAM.

A. EVALUATION DATASETS

We used two real-world datasets including Intel Lab[11] and Speed Camera Violations in Chicago data portal[12] for evaluating the proposed constructions in data stream management systems. In Intel Lab dataset, 54 sensors collected 2.3 million records for one month. We used temperature values collected by the sensors during 03/01/2004 to 03/10/2004 (in other days, all sensors were not attended in recording the temperature). For the second dataset, more than 130 cameras measured the number of speed violations in a day. This dataset contains 176000 records from 01/01/2015 to 02/06/2019. We used the values recorded by 130 cameras during 01/01/2019 to 02/06/2019. We also generated a dataset with random values for 1000 number of sources.

At the initialization of the system, the Setup algorithm is run once and the generated parameters are used in different entities. The setup time is almost similar in all constructions. Generating two safe primes and the generator of the group QR_N is the most time consuming parts of the setup. In Table 5, the setup time for different number of sources are shown.

[11] http://db.csail.mit.edu/labdata/labdata.html

[12] https://data.cityofchicago.org/Transportation/Speed-Camera-Violations/hhkd-xvj4

TABLE 5. Average setup time.

	Number of Sources	Setup Time (s)
Dataset-1 (Intel Lab)	54	922
Dataset-2 (Speed Camera)	130	930
Dataset-3 (Random)	1000	1285

TABLE 6. LQA computaion costs.

	Dataset-1	Dataset-2	Dataset-3
Number of Sources	54	130	1000
AIGen Avg Time (ms)	19.3	19.1	19.2
Comb Avg Time (ms)	3.3	8.6	65.9
Vrfy Avg Time (ms)	20	18	19.2

TABLE 7. CQA computation and communication costs.

Number of Sources		Dataset-1 54	Dataset-2 130	Dataset-3 1000
Range	Avg Comb Time (ms)	2.6	18	965.7
	Avg Vrfy Time (ms)	61	65	66.32
	Avg VO size (Byte)	823	956	993
Median	Avg Comb Time (ms)	3.8	17.3	973.8
	Avg Vrfy Time (ms)	52.8	54	70.5
	Avg VO size (Byte)	823	895	897
Max	Avg Comb Time (ms)	3.9	14.5	983.4
	Avg Vrfy Time (ms)	63.1	46	65.44
	Avg VO size (Byte)	821	813	1052

B. EVALUATION OF LQA CONSTRUCTION

For the linear query authentication (LQA) construction, according to the number of sources, we have generated random coefficients α_i where $i \in [n_s]$ for the linear function $\sum_{i=1}^{n_s} \alpha_i v_i$. Table 6 illustrates the computation and communication costs evaluated in our experiments for different number of sources. In this construction the size of *AI* and *VO* are 288 and 256 bytes (RSA modulus) which are independent of the number of sources.

C. EVALUATION OF CQA CONSTRUCTION

According to the type of the values, for adjusting the distribution of the values among the buckets, different algorithms of partitioning can be used. We considered nine buckets for dataset-1 (containing values between 0 and 140), seven buckets for dataset-2 (containing values between 0 and 300), and four buckets for dataset-3 (containing values between 0 and 140). We measured the cost of the three queries including max, median, and range. The average time of AIGen is 39 millisecond and the size of the authentication information generated by each source is 544 bytes. The time of other algorithms is reported in Table 7.

We also measured the growth in the computation cost and the size of the verification object with respect to the number of the source in Figure 3. For this purpose, we created random values for different number of the source.

D. EVALUATION OF WCQA CONSTRUCTION

In the WCQA construction, each source generates w_i values and sends all of them with authentication information to the aggregator. We evaluated the computation and communication costs for the Intel lab dataset with different window sizes

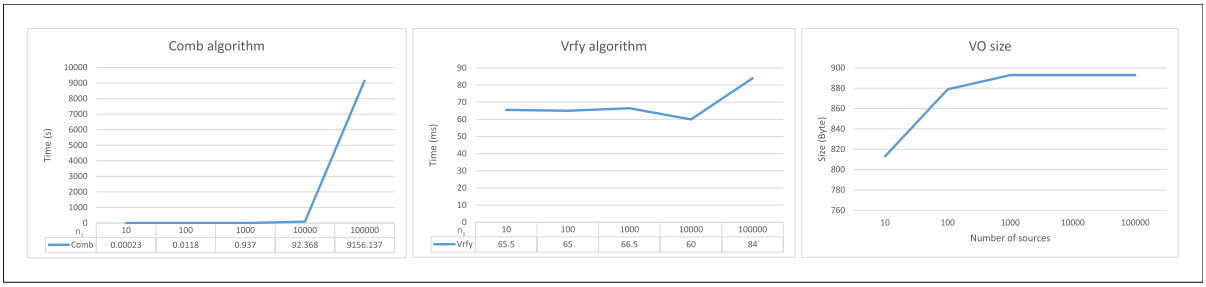

FIGURE 3. CQA computation and communication costs for different number of the sources.

FIGURE 4. WCQA computation and communication costs for different window sizes.

(we assume that all sources generate the same number of values). The evaluation results are shown in Figure 4.

E. DISCUSSION

The time complexity of the verification algorithm and size of the verification object are two main measures that determine the efficiency of the proposed solution in the outsourcing model. In the LQA construction, the verification time and the size of the VO are constant, independent of the number of sources. This is a great achievement because the linear functions can be used as a basic function for creating other functions that are used in our real-life applications.

For the second construction, CQA, if the system expert, partitions the domain of the values in such a way that the distribution of the values in all buckets are about uniform, the size of the verification object will be a minimal proportion of the size of all values. As we see in Table 7, the size of the verification object grows smoothly with the growth in the number of the sources and the verification time is almost the same for different numbers of sources and different types of queries.

The last construction, WCQA, is useful for applications process streams of values like aggregating sensor data in wireless sensor networks. As we see in Fig. 4, the growth in the verification time slightly depends on the total number of the values and the growth in the size of the verification object is acceptable for the real-world applications.

X. CONCLUSION

In this paper, we proposed constructions for linear, countable, and window-based countable queries, when data are collected from multiple sources. We first defined a lightweight linear query authentication construction (LQA) based on the RSA-based homomorphic signature, which is secure in the random oracle model.

Then we used the LQA construction to verify the results of the countable queries (such as max/min, top-k/first-k, range, and median), and window-based countable queries. In the proposed constructions for the countable and window-based countable queries, we utilized the bucket partitioning for verifying the distribution of the results among the all gathered values for aggregation. In these two constructions, the results

are verified by verifying the correctness of the result buckets and the location of the result values in these buckets.

We proved the correctness, succinctness, and security of our constructions and implemented them for experimenting their applicability in real-world systems. The experiment results show that the communication and computation costs of the method are acceptable in practice for real-world applications. It should be noted that the verification cost and the size of the verification object are constant in LQA, but they grow smoothly in CQA and WCQA constructions by increasing the number of gathered values.

As future works, we want to propose a solution for the corrupted sources, solve the collusion problem between the sources and aggregator, and propose a construction with the publicly verifiable verification object (by proposing a compiler that converts a multi-key linear homomorphic signature to a multi-key homomorphic signature for the countable queries, and design methods for verifying the results of the multi-attribute range queries.

APPENDIX A

A.1 PROOF OF THEOREM 1

The correctness of the LQA construction is verified according to Definition 4.

Authentication Correctness: For verifying a signature σ according to the equation (3), we have $\sigma^e = \left(g^{sk_{id}\mathrm{h}(\Delta||i)+v}\right)^{de} \bmod N$. As we know in the RSA signature "$de \bmod \phi(N) = 1$", so we have $\sigma^e = \left(g^{sk_{id}\mathrm{h}(\Delta||i)+v}\right) \bmod N$ which is correct according to the equation (3) and Definition 4.

Evaluation Correctness: The evaluation is correct because the created authenticators have the homomorphic property. The Vrfy algorithm in the LQA construction checks the following equality.

$$\begin{aligned}(\sigma')^e &= \left(\prod_{i=1}^{\omega} \sigma_i^{\alpha_i}\right)^e \bmod N \\ &= \prod_{i=1}^{\omega} \left(g^{sk_{id_i}\mathrm{h}(\Delta||\tau_i)+v_i}\right)^{\alpha_i} \bmod N \\ &= g^{\sum_{i=1}^{\omega}(sk_{id_i}\alpha_i\mathrm{h}(\Delta||\tau_i)+\alpha_i v_i)} \bmod N \\ &= \left(g^{\sum_{i=1}^{\omega} sk_{id_i}\alpha_i\mathrm{h}(\Delta||\tau_i)} g^{V_f}\right) \bmod N\end{aligned}$$

Which is correct according to the equation (3).

A.2 PROOF OF THEOREM 2

For proving Theorem 2, we show that if the adversary $\mathcal{A}$ can do a forgery, the RSA problem is inverted. Let λ is the security parameter and n is the number of inputs. For inverting the RSA problem, N, e, and $y \in_R \mathbb{Z}_N^*$ are given to the solver and the solver tries to find $x = y^{\frac{1}{e}} \bmod N$. We assume that the challenger $\mathcal{C}$ can solve the RSA problem if the adversary $\mathcal{A}$ can do a forgery in the proposed construction. For solving this problem, $\mathcal{C}$ invokes $\mathcal{A}$ as a subroutine and gives N and $g = y^2 \bmod N$ to it. $\mathcal{A}$ plays the game introduced in Definition 5 and sends the queries of the form $(\ell = (id, i), \Delta, v)$ to $\mathcal{C}$. Let $\sigma_\tau = (r_i g^v)^d \bmod N$, where $r_i \in_R QR_N$. For creating the authentication information for the value v, $\mathcal{C}$ computes the random value r_i as follows:

$$r_i = h_i^e . g^{-v}, \qquad h_i \in_R QR_N$$

Then, $\mathcal{C}$ computes the authentication information AI_i, which is equal to $\langle \ell, h_i \rangle$ and sends it to $\mathcal{A}$. Finally, $\mathcal{A}$ outputs a tuple $(\mathcal{P}^*, \Delta^*, V_f^*, VO^*)$, where $\mathcal{P}^* = \langle f^*, (\ell_1{}^*, \ldots, \ell_\omega{}^*) \rangle$ and $VO^* = \sigma'^*$ is a valid verification object for the result value V_f^* which is obtained by executing f on the values belonging to the dataset Δ^*. If $V_f^* \notin Span(v_1, \ldots, v_n)$ where the values $\{v_1, v_2, \ldots, v_n\}$ are exist in $S(\Delta^*)$, we have

$$\sigma'^* = \left(\prod_{i=1}^{n_s} h_i^{e.\alpha_i} . g^{-\sum_{i=1}^{n_s} \alpha_i v_i} . g^{V_f^*}\right)^d \bmod N$$

$$\sigma'^* = \prod_{i=1}^{n_s} h_i^{\alpha_i} . \left(g^{V_f^* - \sum_{i=1}^{n_s} \alpha_i v_i}\right)^d$$

Then, we have

$$\left(g^{V_f^* - \sum_{i=1}^{n_s} \alpha_i v_i}\right)^d = \frac{\sigma'^*}{\prod_{i=1}^{n_s} h_i^{\alpha_i}}$$

Let $\frac{\sigma'^*}{\prod_{i=1}^{n_s} h_i^{\alpha_i}} = z$, then we have $z^e = y^{2(V_f^* - \sum_{i=1}^{n_s} \alpha_i v_i)}$. As $\gcd(e, 2(V_f^* - \sum_{i=1}^{n_s} \alpha_i v_i)) = 1$, because e is a large prime, we have two integers a and b such that $1 = ae + b(2(V_f^* - \sum_{i=1}^{n_s} \alpha_i v_i))$. So, we can find a and b using the extended Euclidean algorithm, and compute $y^{\frac{1}{e}} = y^a z^b$.

A.3 PROOF OF THEOREM 5

The correctness of σ_B is obvious according to the previous constructions and we just prove the correctness of σ_L. We recall two sets $B = \{t_j | \exists v \in V_f, j = get\_bucket(P, v)\}$ and $AL = \{(\ell_i, \sigma_{L,i}) | (\ell_i, v_i) \in V(\Delta) \wedge t_{get\_bucket(P,v_i)} \in B\}$. The set B contains the tags of all buckets that have at least one result value in it, and the set AL contains the authentication information for the locations of the values fallen in the buckets that have at least one result value in it.

σ_L is the combination of all signatures exist in AL and is computed as $\sigma_L = \prod_{((id_i,i),\sigma_{L,i}) \in AL} \sigma_{L,i} \bmod N$. Let V is the set of values fall into the result buckets, and $\overline{V}$ is the set of values that don't fall into the result buckets, such that V and $\overline{V}$ include the values generated by all sources with identifier $((id, .), .) \in AL$. For a value v, let t_v and dst_v are the tag of the bucket that the value v belongs to it and its distance from the beginning of that bucket, respectively. For verifying σ_L according to the equation (4) and the verification object $VO = \langle C, S, \sigma_B, \sigma_L, \sigma_{B'}, \mathrm{h}_{LB} \rangle$, the following equations are valid and the combined signature σ_L is correct.

$$\begin{aligned}(\sigma_L)^e &= \prod_{(id_i,i) \in AL} {\sigma_{L,i}}^e \bmod N \\ &= \prod_{(id_i,i) \in AL} g^{sk_{id_i}\mathrm{h}(\Delta||i)} \\ &\quad \times \prod_{v \in \overline{V}} g^{\mathrm{h}\left(t_{get\_bucket(v)}||(v - R_{get\_bucket(v)-1}+1)\right)}\end{aligned}$$

$$\times \prod_{v \in V} g^{\mathrm{h}\left(t_{get\_bucket(v)} || (v - R_{get\_bucket(v)-1} + 1)\right)} \ mod\ N$$
$$= \sigma_{B'} \cdot g^{\mathrm{h}_{LB} - \sum_{S_i \in S} c_i \mathrm{h}(t_i)} \cdot \prod_{S_i \in S} g^{\sum_{(j,s_j) \in S_i} (s_j . \mathrm{h}(t_i || j))}$$

APPENDIX B

We explain the algorithms for verifying the median (Algorithm 2) and max (Algorithm 5) queries using the result set (V_f), the public partitioning parameter (P), and two tuples C and S which are part of the verification object (VO). We recall that P contains the number of the buckets (m), the domain of the values ($[R_0, R_m]$), the intervals or buckets ($[R_0, R_1)$, $[R_1, R_2)$, ..., $[R_{m-1}, R_m]$), and the tags ($\{t_1, \ldots, t_m\}$) of all buckets.

Algorithm 3 get_bucket Function

```
Function get_bucket(.) is
    Input: P,v
    Output: bnum
    for i = 1 To m do
        if v ≥ R_{i-1} and v < R_i then
            bnum = i;
        end
    end
end
```

Algorithm 4 Median Query Validation

```
Function Median_Validate(.) is
    Input: P, V_f, C, S
    Output: b
    Set b = 1, c_left = 0, c_right = 0, odd = false;
    if size(V_f) = 1 then
        j_l = get_bucket(P, V_f[0]);
        dst_l = V_f[0] − R_{(j_l−1)} + 1;
        j_r = j_l;
        dst_r = dst_l;
        odd = true;
    end
    else if size(V_f) = 2 then
        j_l = get_bucket(P, V_f[0]);
        dst_l = V_f − R_{(j_l−1)} + 1;
        j_r = get_bucket(P, V_f[1]);
        dst_r = V_f − R_{(j_r−1)} + 1;
    end
    for i = 1 To j_l − 1 do
        c_left += C[i];
    end
    for i = j_r + 1 To m do
        c_right += C[i];
    end
    for i = 0 To dst_l − 1 do
        if exists S_{j_l}.get(i) then
            c_left += S_{j_l}.get(i);
        end
    end
    for i = dst_r + 1 To R_{j_r} − R_{j_r−1} + 1 do
        if exists S_{j_r}.get(i) then
            c_right += S_{j_r}.get(i);
        end
    end
    d = |c_left − c_right|;
    r = (S_{j_r}.get(dst_r) + S_{j_l}.get(dst_l) − 2 − d) mod 2;
    if r ≠ 0 then
        b = 0;
    end
end
```

Algorithm 5 Max Query Validation

```
Function Max_Validate(.) is
    Input: P, max = V_f[0], C, S
    Output: b
    Set b = 1;
    j = get_bucket(P, max);
    dst = max − R_{(j−1)} + 1;
    if C[j] = 0 or S_j[dst] = 0 then
        b = 0;
    end
    else
        for ( i = j + 1 To m ) {
            if C[i] ≠ 0 then
                b = 0;
            end
        }
        for ( i = dst + 1 To R_{(j)} − R_{(j−1)} + 1 ) {
            if exists S_j.get(i) then
                b = 0;
            end
        }
    end
end
```

REFERENCES

[1] R. Johnson, L. Walsh, and M. Lamb, "Homomorphic signatures for digital photographs," in *Financial Cryptography and Data Security*, G. Danezis, Ed. Berlin, Germany: Springer, 2012, pp. 141–157.

[2] D. Fiore, A. Mitrokotsa, L. Nizzardo, and E. Pagnin, "Multi-key homomorphic authenticators," in *Advances in Cryptology—ASIACRYPT 2016*, J. H. Cheon and T. Takagi, Eds. Berlin, Germany: Springer, 2016, pp. 499–530.

[3] F. Li, K. Yi, M. Hadjieleftheriou, and G. Kollios, "Proof-infused streams: Enabling authentication of sliding window queries on streams," in *Proc. 33rd Int. Conf. Very Large Data Bases (VLDB)*, 2007, pp. 147–158.

[4] R. Gennaro, J. Katz, H. Krawczyk, and T. Rabin, "Secure network coding over the integers," in *Public Key Cryptography—PKC 2010*, P. Q. Nguyen and D. Pointcheval, Eds. Berlin, Germany: Springer, 2010, pp. 142–160.

[5] K. Yi, F. Li, G. Cormode, M. Hadjieleftheriou, G. Kollios, and D. Srivastava, "Small synopses for group-by query verification on outsourced data streams," *ACM Trans. Database Syst.*, vol. 34, no. 3, pp. 15:1–15:42, Sep. 2009.

[6] S. Nath and R. Venkatesan, "Publicly verifiable grouped aggregation queries on outsourced data streams," in *Proc. IEEE 29th Int. Conf. Data Eng. (ICDE)*, Apr. 2013, pp. 517–528.

[7] S. Papadopoulos, G. Cormode, A. Deligiannakis, and M. Garofalakis, "Lightweight authentication of linear algebraic queries on data streams," in *Proc. Int. Conf. Manage. Data (SIGMOD)*. New York, NY, USA: ACM, 2013, pp. 881–892.

[8] X. Liu, W. Sun, H. Quan, W. Lou, Y. Zhang, and H. Li, "Publicly verifiable inner product evaluation over outsourced data streams under multiple keys," *IEEE Trans. Services Comput.*, vol. 10, no. 5, pp. 826–838, Sep. 2017.

[9] X. A. Wang, Y. Liu, A. K. Sangaiah, and J. Zhang, "Improved publicly verifiable group sum evaluation over outsourced data streams in IoT setting," *Computing*, vol. 101, no. 7, pp. 773–790, Jul. 2019, doi: 10.1007/s00607-018-0641-6.

[10] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy, "Proof verification and the hardness of approximation problems," *J. ACM*, vol. 45, no. 3, pp. 501–555, May 1998.

[11] Y. Ishai, E. Kushilevitz, and R. Ostrovsky, "Efficient arguments without short PCPs," in *Proc. 22nd Annu. IEEE Conf. Comput. Complex. (CCC)*, Jun. 2007, pp. 278–291.
[12] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," in *Proc. IEEE Symp. Secur. Privacy*, May 2013, pp. 238–252.
[13] C. Costello, C. Fournet, J. Howell, M. Kohlweiss, B. Kreuter, M. Naehrig, B. Parno, and S. Zahur, "Geppetto: Versatile verifiable computation," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 253–270.
[14] Y. Kalai and O. Paneth, *Delegating RAM Computations*. Berlin, Germany: Springer, 2016, pp. 91–118.
[15] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish, "Verifying computations with state," in *Proc. 24th ACM Symp. Operating Syst. Princ.* New York, NY, USA: ACM, Nov. 2013, pp. 341–357.
[16] S. Agrawal, D. Boneh, X. Boyen, and D. M. Freeman, "Preventing pollution attacks in multi-source network coding," in *Public Key Cryptography—PKC 2010*, P. Q. Nguyen and D. Pointcheval, Eds. Berlin, Germany: Springer, 2010, pp. 161–176.
[17] D. Boneh and D. M. Freeman, "Linearly homomorphic signatures over binary fields and new tools for lattice-based signatures," in *Public Key Cryptography—PKC 2011*, D. Catalano, N. Fazio, R. Gennaro, and A. Nicolosi, Eds. Berlin, Germany: Springer, 2011, pp. 1–16.
[18] D. Catalano, D. Fiore, and B. Warinschi, "Efficient network coding signatures in the standard model," in *Public Key Cryptography—PKC 2012*, M. Fischlin, J. Buchmann, and M. Manulis, Eds. Berlin, Germany: Springer, 2012, pp. 680–696.
[19] D. Boneh and D. M. Freeman, "Homomorphic signatures for polynomial functions," in *Advances in Cryptology—EUROCRYPT 2011*, K. G. Paterson, Ed. Berlin, Germany: Springer, 2011, pp. 149–168.
[20] D. Catalano, D. Fiore, and B. Warinschi, "Homomorphic signatures with efficient verification for polynomial functions," in *Advances in Cryptology—CRYPTO 2014*, J. A. Garay and R. Gennaro, Eds. Berlin, Germany: Springer, 2014, pp. 371–389.
[21] S. Gorbunov, V. Vaikuntanathan, and D. Wichs, "Leveled fully homomorphic signatures from standard lattices," in *Proc. 47th Annu. ACM Symp. Theory Comput.* New York, NY, USA: ACM, Jun. 2015, pp. 469–477.
[22] R. Gennaro and D. Wichs, "Fully homomorphic message authenticators," in *Advances in Cryptology—ASIACRYPT 2013*, K. Sako and P. Sarkar, Eds. Berlin, Germany: Springer, 2013, pp. 301–320.
[23] R. W. F. Lai, R. K. H. Tai, H. W. H. Wong, and S. S. M. Chow, "Multi-key homomorphic signatures unforgeable under insider corruption," in *Advances in Cryptology—ASIACRYPT 2018*, T. Peyrin and S. Galbraith, Eds. Cham, Switzerland: Springer, 2018, pp. 465–492.
[24] L. Schabhüser, D. Butin, and J. Buchmann, "Context hiding multi-key linearly homomorphic authenticators," *IACR Cryptol. ePrint Arch.*, vol. 2018, p. 629, Mar. 2018.
[25] P. Devanbu, M. Gertz, C. Martel, and S. G. Stubblebine, "Authentic data publication over the Internet," *J. Comput. Secur.*, vol. 11, no. 3, pp. 291–314, Jul. 2003.
[26] M. T. Goodrich, R. Tamassia, and N. Triandopoulos, "Super-efficient verification of dynamic outsourced databases," in *Proc. Cryptopgraphers' Track RSA Conf. Topics Cryptol. (CT-RSA)*. Berlin, Germany: Springer-Verlag, 2008, pp. 407–424.
[27] H. Li, R. Lu, L. Zhou, B. Yang, and X. Shen, "An efficient merkle-tree-based authentication scheme for smart grid," *IEEE Syst. J.*, vol. 8, no. 2, pp. 655–663, Jun. 2014.
[28] M. Narasimha and G. Tsudik, "Authentication of outsourced databases using signature aggregation and chaining," in *Proc. 11th Int. Conf. Database Syst. Adv. Appl. (DASFAA)*. Berlin, Germany: Springer-Verlag, 2006, pp. 420–436.
[29] H. Pang, J. Zhang, and K. Mouratidis, "Scalable verification for outsourced dynamic databases," *Proc. VLDB Endowment*, vol. 2, no. 1, pp. 802–813, Aug. 2009.
[30] W. Song, B. Wang, Q. Wang, Z. Peng, and W. Lou, "Tell me the truth: Practically public authentication for outsourced databases with multi-user modification," *Inf. Sci.*, vol. 387, pp. 221–237, May 2017.
[31] T. Claveirole, M. D. de Amorim, M. Abdalla, and Y. Viniotis, "Securing wireless sensor networks against aggregator compromises," *IEEE Commun. Mag.*, vol. 46, no. 4, pp. 134–141, Apr. 2008.
[32] L. Hu and D. Evans, "Secure aggregation for wireless networks," in *Proc. Symp. Appl. Internet Workshops*, Jan. 2003, pp. 384–391.
[33] W. Du, J. Deng, Y. S. Han, and P. K. Varshney, "A witness-based approach for data fusion assurance in wireless sensor networks," in *Proc. IEEE Global Telecommun. Conf. (GLOBECOM)*, Dec. 2003, pp. 1435–1439.
[34] B. Przydatek, D. Song, and A. Perrig, "SIA: Secure information aggregation in sensor networks," in *Proc. 1st Int. Conf. Embedded Netw. Sensor Syst. (SenSys)*. New York, NY, USA: ACM, 2003, pp. 255–265.
[35] M. Garofalakis, J. M. Hellerstein, and P. Maniatis, "Proof sketches: Verifiable in-network aggregation," in *Proc. IEEE 23rd Int. Conf. Data Eng.*, Apr. 2007, pp. 996–1005.
[36] Y.-T. Tsou, C.-S. Lu, and S.-Y. Kuo, "SER: Secure and efficient retrieval for anonymous range query in wireless sensor networks," *Comput. Commun.*, vol. 108, pp. 1–16, Aug. 2017.
[37] C. M. Yu, G. K. Ni, I. Y. Chen, E. Gelenbe, and S. Y. Kuo, "Top-k query result completeness verification in tiered sensor networks," *IEEE Trans. Inf. Forensics Security*, vol. 9, no. 1, pp. 109–124, Jan. 2014.
[38] Y. Yao, N. Xiong, J. H. Park, L. Ma, and J. Liu, "Privacy-preserving max/min query in two-tiered wireless sensor networks," *Comput. Math. with Appl.*, vol. 65, no. 9, pp. 1318–1325, May 2013.
[39] H. Hacigümüş, B. Iyer, C. Li, and S. Mehrotra, "Executing SQL over encrypted data in the database-service-provider model," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*. New York, NY, USA: ACM, 2002, pp. 216–227.
[40] J. Dougherty, R. Kohavi, and M. Sahami, "Supervised and unsupervised discretization of continuous features," in *Machine Learning Proceedings 1995*, A. Prieditis and S. Russell, Eds. San Francisco, CA, USA: Morgan Kaufmann, 1995, pp. 194–202.
[41] M. Backes, D. Fiore, and R. M. Reischuk, "Verifiable delegation of computation on outsourced data," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*. New York, NY, USA: ACM, 2013, pp. 863–874.
[42] D. Boneh, D. Freeman, J. Katz, and B. Waters, "Signing a linear subspace: Signature schemes for network coding," in *Public Key Cryptography—PKC 2009*, S. Jarecki and G. Tsudik, Eds. Berlin, Germany: Springer, 2009, pp. 68–87.
[43] R. Gennaro, C. Gentry, and B. Parno, "Non-interactive verifiable computing: Outsourcing computation to untrusted workers," in *Proc. 30th Annu. Conf. Adv. Cryptol. (CRYPTO)*. Berlin, Germany: Springer-Verlag, 2010, pp. 465–482.
[44] D. Jackson, "Note on the median of a set of numbers," *Bull. Amer. Math. Soc.*, vol. 27, no. 4, pp. 160–164, 1921.
[45] P. Locher and R. Haenni, "A lightweight implementation of a shuffle proof for electronic voting systems," *Informatik*, vol. 44, no. 232, pp. 1391–1400, 2014.

SOMAYEH DOLATNEZHAD SAMARIN is currently pursuing the Ph.D. degree in information technology engineering (data security field) with the Department of Computer Engineering, Sharif University of Technology, Tehran, Iran. Her research interests include intrusion detection systems, database security, cloud computing security, and verifiable computations.

MORTEZA AMINI received the Ph.D. degree in software engineering (data security field) from the Sharif University of Technology, in 2010. He is currently an Associate Professor with the Department of Computer Engineering, Sharif University of Technology, Tehran, Iran. He is also one of the directors of Data and Network Security Laboratory (DNSL) with the Department of Computer Engineering. His research interests include database security, access control, intrusion detection systems, and formal methods in information security.