Publicly Verifiable Inner Product Evaluation over Outsourced Data Streams under Multiple Keys

Xuefeng Liu, Wenhai Sun, *Student Member, IEEE,* Hanyu Quan, Wenjing Lou, *Fellow, IEEE,* Yuqing Zhang, Hui Li, *Member, IEEE*

Abstract—Uploading data streams to a resource-rich cloud server for inner product evaluation, an essential building block in many popular stream applications (e.g., statistical monitoring), is appealing to many companies and individuals. On the other hand, verifying the result of the remote computation plays a crucial role in addressing the issue of trust. Since the outsourced data collection likely comes from multiple data sources, it is desired for the system to be able to pinpoint the originator of errors by allotting each data source a unique secret key, which requires the inner product verification to be performed under any two parties' different keys. However, the present solutions either depend on a single key assumption or powerful yet practically-inefficient fully homomorphic cryptosystems. In this paper, we focus on the more challenging multi-key scenario where data streams are uploaded by multiple data sources with distinct keys. We first present a novel homomorphic verifiable tag technique to publicly verify the outsourced inner product computation on the dynamic data streams, and then extend it to support the verification of matrix product computation. We prove the security of our scheme in the random oracle model. Moreover, the experimental result also shows the practicability of our design.

Index Terms—Data stream, Computation outsourcing, Storage outsourcing, Multiple keys, Public verifiability

1 Introduction

The past few years have witnessed the proliferation of streaming data generated by a variety of applications/systems, such as GPS, Internet traffic, asset tracking, wireless sensors, etc. Retaining a local copy of such exponentially-growing volume of data is becoming prohibitive for resource-constrained companies/organizations, let alone offering efficient and reliable query services on it.

Consider a stream-oriented service (e.g., market analysis, weather forecasting and traffic management), where *multiple* resource-constrained sources continuously collect or generate data streams, and outsource them to a powerful external server, e.g. cloud, for desired critical computations and storage savings. For example, using inner product computation over any two outsourced stock data streams from different sources for correlation analysis, a stock market trader is able to spot the arbitrage opportunities [1].

In spite of its merits, outsourcing naturally raises the issue of trust [2], [3], [4]. The third-party server may act maliciously due to insider/outsider attack, software/hardware malfunctions, intentional saving of computational resources, etc. Thus, it is desirable for clients to verify the computation result provided by the server. However, designing a verifiable computation scheme for the above example is not self-explanatory due to the following challenges.

First of all, the outsourced computation is data-sensitive, i.e., given forged data from a source, the final computation result will be erroneous even if the corresponding query is correctly processed by the server. Cryptography provides an off-the-shelf method to tackle this problem, namely, each data source may be equipped with a unique secret key to "sign" its data contribution, from which traceability is readily derived. However, the typical signature algorithm does not serve on purpose of verifiable multi-key computation. In deed, most of the existing verifiable computation schemes only focus on the single-key setting, i.e., data and its computation are outsourced from merely one contributor [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21] or from multiple contributors but with the same key [22]. On the other hand, we may resort to the powerful fully homomorphic encryption (FHE) but are hardly willing to use it in practice due to efficiency concern [23][24]. As a result, we are still striving to come up with a promising solution in such

- *X. Liu and H. Li are with the School of Cyber Engineering, Xidian University, China. X. Liu is also with the State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, China. E-mail: liuxf@mail.xidian.edu.cn, lihui@mail.xidian.edu.cn.*
- *H. Quan is with the School of Telecommunications Engineering, Xidian University, China. E-mail: hyquan@stu.xidian.edu.cn.*
- *W. Sun and W. Lou are with Virginia Polytechnic Institute and State University, Blacksburg, VA, USA. E-mail: {whsun,wjlou}@vt.edu.*
- *Corresponding author: Y. Zhang is with the national computer network intrusion protection center, University of Chinese Academy of Sciences and the School of Cyber Engineering, Xidian University, China. E-mail: zhangyq@ucas.ac.cn.*

a challenging multi-key setting.

Second, clients may not be in the same trust domain with data sources. A *keyless* client is hopefully able to conduct the result verification [5], [9], [15], [16], [17]. Hence, *public* verification property is more engaging here so as to allow any party devoid of secret keys with sources to check the outsourced computations.

Third, we must take the *efficiency* into account when realizing our design from both the viewpoints of computation and communication cost. In general, the verification cost is expected to be smaller than the initially outsourced computation, and constant communication overhead between client and server is favorable, independent of the number of data involved in the computation. Otherwise, the client may carry out the computation on her/his own.

Last but not the least, given potentially-unbounded data streams, it requires the outsourced functions to be evaluated over dynamic data. In other words, the involved data cannot be determined in advance. Therefore, how to *publicly* and *efficiently* verify the inner product evaluation over the outsourced data *streams* under *multiple keys* still remains an open problem.

Our contributions. In this paper, we introduce a novel homomorphic verifiable tag technique and design an efficient and publicly verifiable inner product computation scheme on the dynamic outsourced data stream under multiple keys. Our contributions are summarized as follows:

1) To the best of our knowledge, this is the first work that addresses the problem of verifiable delegation of inner product computation over (potentially unbounded) outsourced data streams under the *multi-key* setting. Specifically, we first present a publicly verifiable group-by sum algorithm, which servers as a building block for verifying the inner product of dynamic vectors under two different keys. Then, we extend the construction of the verifiable inner product computation to support matrix product from any two different sources.
2) Our scheme is efficient enough for practical use in terms of communication and computation overhead. Specifically, the size of the proof generated by the server to authenticate the computation result is constant, regardless of the input size n of the evaluated function. In addition, the verification overhead on the client side is constant for inner product querie[1]. For matrix product query, the verification cost is $O(n^2)$ in stark contrast to the super-quadratic computational complexity for matrix product.
3) Our scheme achieves the public verifiability, i.e., a *keyless* client is able to verify the computation results.
4) We formally define and prove the security of our scheme under the Computational Diffie-Hellman assumption [25] in the random oracle model.

Organization. The rest of the paper is organized as follows. Section 2 gives the related work. In section 3, we define the system model, design goals, proposed algorithms and security model. We present a group-by sum algorithm as a building block and introduce our verifiable inner product computation scheme in section 4, and an extension to matrix product in section 5, respectively. The security analysis is given in section 6, and we evaluate the performance of our scheme in section 7. Finally, section 8 concludes this paper.

2 Related Work

The problem of *verifying the outsourced algebraic computation* has attracted extensive attention in the past few years. These schemes can be divided into two categories: under single-key setting [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22] and under multi-key setting [23][24].

Single-key Setting. Fully homomorphic message authenticators [6], [7], [8] allow the holder of a public evaluation key to perform computations on previously authenticated data, in such a way that the produced proof can be used to certify the correctness of the computation. More precisely, with the knowledge of the secret key used to authenticate the original data, a client can verify the computation by checking the proof. For the asymmetric setting, Boneh and Freeman [9] proposed a realization of homomorphic signatures for bounded constant degree polynomials based on hard problems on ideal lattices. Although not all the above schemes are explicitly presented in the context of streaming data, they can be applied there under a *single-key setting*. In this scenario, the data source continually generates and outsources authenticated data values to a third-party server. Given the public key, the server can compute over these data and produce a proof, which enables the client to privately [6], [7], [8] or publicly [9] verify the computation result.

Our work is also related to a line of *verifiable* schemes [10], [11], [12], [13], [14], where a resource-constrained data source can outsource a computationally-intensive task to a third-party server and efficiently verify computation result. Recently, several works towards public verification either for specific classes of computations [15], [16] or for arbitrary computations [17] have been proposed. However, the outsourced data [15], [16] has to be a priori fixed. Another interesting line of works [18], [19], [20] considered a different setting for verifiable computation. In their models, the client needs to know the

1. Constant verification cost is achieved by a pre-computation in an offline phase.

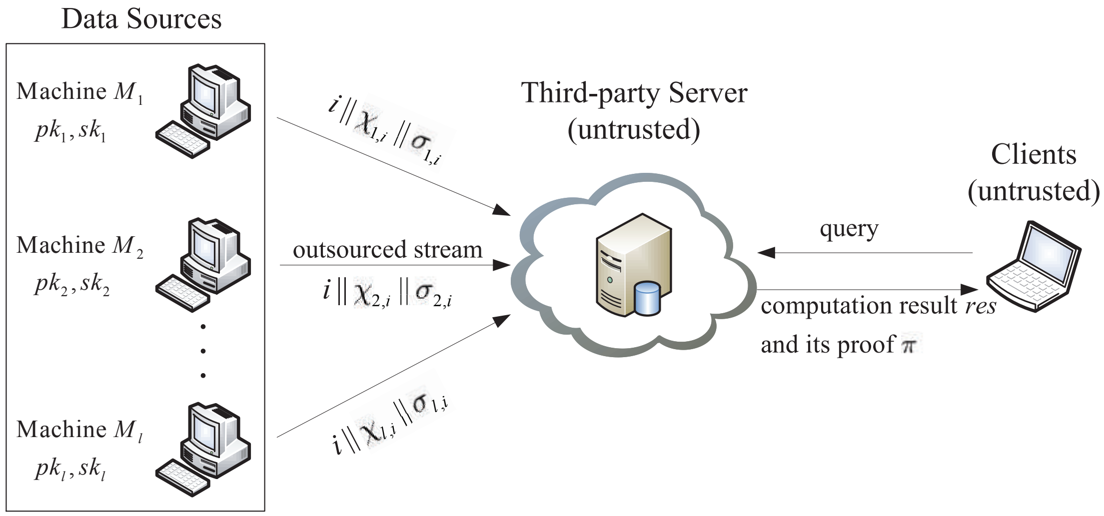


Fig. 1. System model

input of the outsourced computation and runs an interactive protocol with the server in order to verify the results. In memory delegation [21], the stream outsourcing was considered but with the restraint that the size of the steam has to be a priori bounded.

There are several works customized for the data stream outsourcing scenario. Specifically, a publicly verifiable grouped aggregation queries on outsourced data stream was proposed in [5]. In this work, clients are only allowed to query the server for the summation of a grouped data specified by the data source. A scheme of outsourced computations including group-by sum, inner product, matrix product with private verifiability was considered in [22]. Other works considering the verification of outsourced operations such as ranges and joins, were presented in [26], [27], [28], [29], [30], [31] .

Multi-key Setting. Recently, a multi-key non-interactive verifiable computation scheme was proposed in [23], followed by a stronger security guarantee scheme [24]. In their constructions, n computationally-weak users outsource to an untrusted server the computation of a function f over a series of joint inputs $(x_1^{(i)}, x_2^{(i)}, ..., x_n^{(i)})$ without interacting with each other, where i denotes the ith computation. In their schemes, after the generation of system parameters, data sources $P_j(j \in [1, n])$ outputs an encoded function f to the server. Then for the ith computation, P_j outsources the encoding of $x_j^{(i)}$ to the server and computes a secret $\tau_j^{(i)}$ for the verification. However, these schemes may not be applied to the stream setting since sources lost data control after the outsourcing and thus cannot generate the corresponding secrets for the verification. Besides, both of them based on FHE are not practically efficient. As shown in [32], it takes at least 30 seconds to run one bootstrapping operation of FHE for weaker security parameter on a high performance machine.

In this work, we consider publicly verifiable delegation of inner product computation over dynamic data streams under the *multi-key* setting. The proposed scheme is extremely lightweight for both data sources and clients.

3 Problem Formulation

3.1 System Model

We consider our system architecture as illustrated in Fig.1. There are a set of machines (data sources) $M_1, M_2, ..., M_l$, each of which owns a unique public and private key pair. These machines collect or generate potentially unbounded data streams and outsource them to a third-party server. We assume that these machines are not required to directly communicate with each other. More precisely, for a new data value $\mathcal{X}_{j,i}$ generated at time i, machine M_j $(1 \leq j \leq l)$ computes a homomorphic and publicly verifiable tag $\sigma_{j,i}$, and outsources a tuple $\{i, \mathcal{X}_{j,i}, \sigma_{j,i}\}$ to the server. The time measured in our scheme is discrete and increased with the arrival of a new tuple. In addition, we assume that the clocks of the data sources' machines, the server and the client are (at least loosely) synchronized. This requirement is inherent in most streaming applications [5], [22]. A client requests the server to compute inner product of any two machines' outsourced data streams by sending a corresponding query. Apart from the computation result res, the server also provides its proof π to the client. With π and some auxiliary information, the client is able to verify the correctness of the received computation result res.

We assume that the third-party server is *untrusted* because it sits outside of the trust domain of the sources. We also assume that clients are *untrusted* by the data sources, because they may be compromised, malicious, or collude with the server for financial incentives in practice. Therefore, the secret keys used by data sources to generate tags will not be transferred to clients for the result verification; otherwise, a malicious client with the private keys can collude with the server to modify the data and generate corresponding tags to deceive other clients. In this paper, we focus on the verification of the outsourced computation over public data streams, while sensitive data protection is outside the scope of our work.

3.2 Design Goals

Our scheme aims to achieve the following goals:

- **Multi-key setting**: Given different secret keys, multiple data sources can upload their data streams along with the respective verifiable homomorphic tags generated by the corresponding secret keys to the cloud. As such, no source can deny his/her contribution to the outsourced computations. In addition, the inner product evaluation can be performed over any two sources' outsourced streams, and the result can be verified using the associated tags.
- **Query flexibility**: The client should be free to choose any portion of the data streams as the input of the queried computation.
- **Public verifiability**: All the participants involved in the protocol should be able to *publicly* verify the outsourced computation results without sharing secret keys with data sources.
- **Efficiency**: More precisely, we expect that 1) the communication overhead between a client and the server is constant, i.e., independent of its input size of the queried computation, and that 2) verification overhead on the client side should be smaller than performing the outsourced computation by the client.

3.3 Algorithm Formulation

In this subsection, we provide the formal algorithm definition of our proposed scheme.

Definition 3.1. Our public verifiable inner product computation scheme includes a tuple of algorithms as follows:

- $\mathbf{KeyGen}(1^\kappa) \rightarrow (pk_j, sk_j)$: A probabilistic algorithm run by each machine M_j takes a security parameter κ as input, and outputs a public key pk_j and a secret key sk_j.
- $\mathbf{TagGen}(sk_j, i, \mathcal{X}_{j,i}) \rightarrow \sigma_{j,i}$: A (possibly) probabilistic algorithm run by machine M_j, takes as input its secret key sk_j, the current discrete time i and data $\mathcal{X}_{j,i}$, and outputs a publicly verifiable tag $\sigma_{j,i}$.
- $\mathbf{Evaluate}(\mathcal{F_{IP}}, \mathcal{X}_i, \mathcal{X}_j) \rightarrow res$: Let $\mathcal{X}_i = \{\mathcal{X}_{i,1}, \mathcal{X}_{i,2}, ..., \mathcal{X}_{i,n}\}$ and $\mathcal{X}_j = \{\mathcal{X}_{j,1}, \mathcal{X}_{j,2}, ..., \mathcal{X}_{j,n}\}$ denote the outsourced data streams of machines M_i and M_j, respectively. This deterministic algorithm is run by the server to compute the inner product of streams $\mathcal{X}_i$ and $\mathcal{X}_j$. It takes as inputs the inner product function $\mathcal{F_{IP}}$, two data streams $\mathcal{X}_i$ and $\mathcal{X}_j$, and outputs a computation result res.
- $\mathbf{GenProof}(\mathcal{F_{IP}}, \sigma_i, \sigma_j, \mathcal{X}_i, \mathcal{X}_j) \rightarrow \pi$: Let σ_i and σ_j denote the tag vectors for $\mathcal{X}_i$ and $\mathcal{X}_j$ generated by machine M_i and machine M_j, respectively. This algorithm is run by the server to generate a proof for the result res. It takes as input the inner product function $\mathcal{F_{IP}}$, two tag vectors σ_i and σ_j, as well as two data streams $\mathcal{X}_i$ and $\mathcal{X}_j$, and outputs a proof π.
- $\mathbf{CheckProof}(\mathcal{F_{IP}}, pk_i, pk_j, res, \pi) \rightarrow 0, 1$: A deterministic algorithm is run by the client to check the correctness of res. It takes as input the function $\mathcal{F_{IP}}$, two public keys pk_i and pk_j, the result res, as well as the proof π, and outputs 1 (accept) or 0 (reject).

Note that, **Evaluate** and **GenProof** can be combined together in our verifiable non-interactive inner product computation scheme. Here, we separate them to stress that they are two independent processes..

3.4 Security Definition

Definition 3.2. We state the security definition via the following experiment $\mathbf{Exp}_{\mathcal{A}}^{1^\kappa}$, which is a variation of the standard existential unforgeability under an adaptive chosen-message attack [33]. Intuitively, the experiment captures that an adversary cannot successfully construct a valid proof, unless it follows the client's query.

Setup: The challenger runs algorithm **KenGen** to generate a public key vector $\overrightarrow{pk} = (pk_1, pk_2, ..., pk_l)$ and a secret key vector $\overrightarrow{sk} = (sk_1, sk_2, ..., sk_l)$. The adversary $\mathcal{A}$ is given the public key vector $\overrightarrow{pk}$.

Query: The adversary $\mathcal{A}$ can adaptively query **TagGen** oracle for tags on the discrete time and the message of its choice. Specifically, $\mathcal{A}$ sends a tuple $(M_j, i, \mathcal{X}_{j,i})(1 \leq j \leq l)$ to the challenger. The challenger proceeds as follows: it first initializes an empty list L to record tuples $(M_j, i, \mathcal{X}_{j,i}, \sigma_{j,i})$. If (M_j, i) has not been queried before, the challenger runs algorithm $\mathbf{TagGen}(sk_j, i, \mathcal{X}_{j,i})$ and returns $\sigma_{j,i}$ to $\mathcal{A}$. In addition, the challenger adds a tuple $(M_j, i, \mathcal{X}_{j,i}, \sigma_{j,i})$ into the list L. If (M_j, i) has been queried before and $(M_j, i, \mathcal{X}_{j,i}) \in L$, the challenger retrieves $\sigma_{j,i}$ and returns it to $\mathcal{A}$. Otherwise, the challenger rejects this query.

Request: In this phase, a client requests the adversary $\mathcal{A}$ to evaluate the inner product of $\mathcal{X}_i$ and $\mathcal{X}_j$.

Forge: The adversary $\mathcal{A}$ outputs a tuple (res, π) with the restriction $res \neq \mathcal{X}_i \otimes \mathcal{X}_j$, where $\otimes$ denote the inner product operation.

If $\mathbf{CheckProof}(\mathcal{F_{IP}}, pk_i, pk_j, res, \pi)$ returns 1, then the adversary $\mathcal{A}$ wins this experiment.

KeyGen(1^κ) :
for $j = 1$ to l **do**
  choose a random number $sk_j = s_j \in Z_q^*$ as the secret key
  compute $pk_j = g^{s_j}$
  output (pk_j, sk_j)
end for

TagGen$(sk_j, i, \mathcal{X}_{j,i})$:
compute $\sigma_{j,i} = (g_1^{h_1(M_j,i)} g_2^{h_2(M_j,i)} g_3^{\mathcal{X}_{j,i}})^{sk_j}$
output $\sigma_{j,i}$

Evaluate$(\mathcal{F}_{\mathcal{GS}}, \mathcal{X}_j)$:
compute $res = \sum_{i\in\Delta} \mathcal{X}_{j,i}$
output res

GenProof$(\mathcal{F}_{\mathcal{GS}}, \sigma_j, \mathcal{X}_j)$:
compute $\pi = \prod_{i\in\Delta} \sigma_{j,i}$
output π

CheckProof$(\mathcal{F}_{\mathcal{GS}}, pk_j, res, \pi)$:
set $S_\Delta = (S_1, S_2)$
compute $S_1 = \sum_{i\in\Delta} h_1(M_j, i)$ and $S_2 = \sum_{i\in\Delta} h_2(M_j, i)$
if $(e(\pi, g) = e(g_1^{S_1} g_2^{S_2} g_3^{res}, pk_j))$ **then**
  output 1
else
  output 0
end if

Fig. 2. Publicly verifiable computation for group-by sum query

We say that a publicly verifiable computation on outsourced data stream scheme is secure, if for any probabilistic polynomial time adversary $\mathcal{A}$ the probability that $\mathcal{A}$ succeeds in the above experiment is negligible, i.e., $Pr[\mathbf{Exp}_A^{1^\kappa}(\mathcal{A}) = 1] \le negl(\kappa)$.

4 Our Construction

The public system parameters $\{e, G_1, G_2, q, g, g_1, g_2, g_3, h_1, h_2\}$ used in this work are defined as follows. G_1 and G_2 are two multiplicative cyclic groups of the same prime order q, and e denotes a bilinear map $G_1 \times G_1 \to G_2$ satisfying bilinearity, Non-degeneracy and computability [34]. $\{g, g_1, g_2, g_3\}$ are four generators randomly selected from group G_1. $h_1 : \{0,1\}^* \to Z_q^*$ and $h_2 : \{0,1\}^* \to Z_q^*$ represent two different collision-resistant hash functions, respectively. Let $f : Z_q^* \times \{0,1\}^* \times \{0,1\}^* \to Z_q^*$ be a pseudo-random function (PRF) and $f_\lambda(x, y)$ denote a PRF f with key λ on input (x, y).

Definition 4.1. The CDH Assumption [25]: Given $g, g^s, g_1 \in G_1$ for unknown $s \in Z_q^*$, no probabilistic polynomial-time algorithm can compute g_1^s with non-negligible advantage.

4.1 Building Block

Before introducing our construction for publicly verifiable inner product evaluation scheme, we first consider *a publicly verifiable group-by sum computation scheme over the outsourced dynamic stream under multiple keys* , which is of independent interest and serves as a building block for the verification of *inner product query*.

Specifically, we assume that machine M_j has outsourced the data stream $\mathcal{X}_j = \{\mathcal{X}_{j,1}, \mathcal{X}_{j,2}, ..., \mathcal{X}_{j,n}\}$ to the server. A client requests the server to compute the sum function $\mathcal{F}_{\mathcal{GS}}$ on a subset $\mathcal{X}_{j,\Delta}(\Delta \subseteq [1, n])$, i.e.,

$$res = \mathcal{F}_{\mathcal{GS}}(\mathcal{X}_{j,\Delta}) = \sum_{i\in\Delta} \mathcal{X}_{j,i} \tag{1}$$

We term such query a *group-by sum query*. The scheme for the public verification of a group-by sum query consists of five algorithms as shown in Fig.2, by substituting inner product function $\mathcal{F}_{\mathcal{IP}}$ with group-by sum function $\mathcal{F}_{\mathcal{GS}}$ in Definition 3.1.

The rationale behind this construction is straightforward. Machine M_j computes a homomorphic and publicly verifiable tag $\sigma_{j,i} = (g_1^{h_1(M_j,i)} g_2^{h_2(M_j,i)} g_3^{\mathcal{X}_{j,i}})^{sk_j}$ for $\mathcal{X}_{j,i}$. Given two tags $\sigma_{j,1}$ and $\sigma_{j,2}$, anyone can compute a tag $\sigma = \sigma_{j,1} \cdot \sigma_{j,2}$ for $\mathcal{X}_{j,1} + \mathcal{X}_{j,2}$. The value $\{M_j, i\}$ can be regarded as a one-time index of data $\mathcal{X}_{j,i}$ such that it will not be reused for computing other tags later. More precisely, machine $M_j (1 \le j \le l)$ runs algorithm **KeyGen** to generate a public/secret key pair (pk_j, sk_j) in setup phase. When a new data value $\mathcal{X}_{j,i}$ is collected or generated at time i, machine M_j runs algorithm **TagGen** to compute a tag $\sigma_{j,i}$ and outsources $(i, \mathcal{X}_{j,i}, \sigma_{j,i})$ to the server.

A client sends a group-by sum query $\{M_j, \Delta\}$ to the server for $res = \mathcal{F}_{\mathcal{GS}}(\mathcal{X}_{j,\Delta}) = \sum_{i\in\Delta} \mathcal{X}_{j,i}$. Upon receiving the request, the server calls algorithm **Evaluate** and **GenProof**, and then returns res, π to the client. Finally, the client runs algorithm **CheckProof** to check the validity of the computation result res.

Correctness. The correctness of the verification algorithm can be deduced from the following equation.

$$\begin{aligned} & e(\pi, g) \\ = \; & e(\textstyle\prod_{i\in\Delta} \sigma_{j,i}, g) \\ = \; & e(g_1^{\sum_{i\in\Delta} h_1(M_j,i)} g_2^{\sum_{i\in\Delta} h_2(M_j,i)} g_3^{\sum_{i\in\Delta} \mathcal{X}_{j,i}}, pk_j) \\ = \; & e(g_1^{S_1} g_2^{S_2} g_3^{res}, pk_j) \end{aligned} \tag{2}$$

Discussion. The outsourced computation is data-sensitive, i.e., given forged data from a source, the final computation result will be erroneous even if the corresponding query is correctly processed by the server. In our construction, each data source needs to attach its outsourced stream with tags. The server can check the validity of a tag by verifying whether equation $e(\sigma_{j,i}, g) = e(g_1^{h_1(M_j,i)} g_2^{h_2(M_j,i)}, g_3^{\mathcal{X}_{j,i}}, pk_j)$ holds. In section 6, we will prove that the tag is unforgeable, i.e., no source can deny his/her tags that have been outsourced to the server. Thus, given a disputed data value, we can trace back to the source with a corresponding tag.

Each data source needs to store only the private key and its identity, and the storage consumption is $O(\kappa + \log l)$, where κ is the security parameter and l denotes the number of sources. It takes machine M_j $O(n)$ modular exponentiations, $O(n)$ multiplications in G_1, and $O(n)$ hash operations to generate tags for a data stream $\mathcal{X}_j = \{\mathcal{X}_{j,1}, ..., \mathcal{X}_{j,n}\}$. Note that these tags are computed once and can be used for each query. Thus, the computation cost for each machine can be amortized over the future executions. The storage overhead for the tags at the server side includes $O(n)$ elements in G_1. To compute a proof, the server needs $O(n)$ multiplications in G_1. The proof is an element in G_1. Finally, the online burden at the client to verify the proof includes two parings, three modular exponentiations and two multiplications in G_1, since the auxiliary information S_Δ is independent of $\mathcal{X}_i$ and can be pre-computed.

Let us consider the case without outsourcing. Machine M_j needs to store $O(n)$ elements in Z_q^* for $\mathcal{X}_j$. When receiving a group-by sum query $\{M_j, \Delta\}$, machine M_i either performs the computation itself or transmits the data sets $\mathcal{X}_{i,j} (j \in \Delta)$ to the client. The former may incur a substantial computation overhead to M_i, because there are $O(2^n)$ possible Δ for $\mathcal{X}_j$. The communication cost is $O(n)$ when transmitting $\{\mathcal{X}_{j,i}\}_{i \in \Delta}$ to the client, and it takes $O(n)$ modular additions in Z_q^* for the client to compute $res = \sum_{i \in \Delta} \mathcal{X}_{j,i}$. Thus, there is a clear performance advantage for both data sources and client in the storage and computation outsourcing setting scenario.

4.2 Inner Product Query

Based on the group-by sum query described above, we present a publicly verifiable computation scheme for the *inner product query* over data streams with two different keys in this subsection. Specifically, any two machines M_1 and M_2 outsource the data stream $\mathcal{X}_1 = \{\mathcal{X}_{1,1}, \mathcal{X}_{1,2}, ..., \mathcal{X}_{1,n}\}$ and $\mathcal{X}_2 = \{\mathcal{X}_{2,1}, \mathcal{X}_{2,2}, ..., \mathcal{X}_{2,n}\}$ to the server, respectively. A client requests the server to compute the inner product function $\mathcal{F}_{\mathcal{IP}}$ on $\mathcal{X}_1$ and $\mathcal{X}_2$, i.e.,

$$res = \mathcal{F}_{\mathcal{IP}}(\mathcal{X}_1, \mathcal{X}_2) = \mathcal{X}_1 \otimes \mathcal{X}_2 = \sum_{i=1}^{n} \mathcal{X}_{1,i} \cdot \mathcal{X}_{2,i} \quad (3)$$

Fig.3 shows the concrete protocol.

The main idea behind this construction is as follows. Intuitively, $res = \sum_{i=1}^{n} \mathcal{X}_{1,i} \cdot \mathcal{X}_{2,i}$ is the sum of $\mathcal{X}_{1,i} \cdot \mathcal{X}_{2,i} (i \in [1, n])$. The server can generate a proof $\sigma_{1,i}^{\mathcal{X}_{2,i}}$ for data $\mathcal{X}_{1,i} \cdot \mathcal{X}_{2,i}$, and then aggregates these proofs into a whole one. Thus, the proof for the final result res is:

$$\begin{aligned} \pi_3 &= \prod_{i=1}^{n} \sigma_{1,i}^{\mathcal{X}_{2,i}} \\ &= (g_1^{\sum_{i=1}^{n} h_1(M_1,i)\mathcal{X}_{2,i}} g_2^{\sum_{i=1}^{n} h_2(M_1,i)\mathcal{X}_{2,i}} g_3^{res})^{sk_1} \end{aligned} \quad (4)$$

However, the client is still unable to check the correctness of res without the knowledge of $res_1 =$

KeyGen(1^κ) :
- **for** $j = 1$ to l **do**
  - choose a random number $sk_j = s_j \in Z_q^*$ as the secret key
  - compute $pk_j = g^{s_j}$
  - output (pk_j, sk_j)
- **end for**

TagGen$(sk_j, i, \mathcal{X}_{j,i})$:
- compute $\sigma_{j,i} = (g_1^{h_1(M_j,i)} g_2^{h_2(M_j,i)} g_3^{\mathcal{X}_{j,i}})^{sk_j}$
- output $\sigma_{j,i}$

Evaluate$(\mathcal{F}_{\mathcal{IP}}, \mathcal{X}_1, \mathcal{X}_2)$:
- compute $res = \mathcal{X}_1 \otimes \mathcal{X}_2 = \sum_{i=1}^{n} \mathcal{X}_{1,i} \cdot \mathcal{X}_{2,i}$
- output res

GenProof$(\mathcal{F}_{\mathcal{IP}}, \sigma_1, \sigma_2, \mathcal{X}_1, \mathcal{X}_2)$:
- compute $\pi_1 = \prod_{i=1}^{n} \sigma_{2,i}^{h_1(M_1,i)}$ and $\pi_2 = \prod_{i=1}^{n} \sigma_{2,i}^{h_2(M_1,i)}$
- compute $\pi_3 = \prod_{i=1}^{n} \sigma_{1,i}^{\mathcal{X}_{2,i}}$
- compute $res_1 = \sum_{i=1}^{n} h_1(M_1,i)\mathcal{X}_{2,i}$
- compute $res_2 = \sum_{i=1}^{n} h_2(M_1,i)\mathcal{X}_{2,i}$
- set $\pi = \{res_1, res_2, \pi_1, \pi_2, \pi_3\}$
- output π

CheckProof$(\mathcal{F}_{\mathcal{IP}}, pk_1, pk_2, res, \pi)$:
- set $S_\Delta = (S_{1,1}, S_{1,2}, S_{2,1}, S_{2,2})$
- compute $S_{1,1} = \sum_{i=1}^{n} h_1(M_1,i)h_1(M_2,i)$
- compute $S_{1,2} = \sum_{i=1}^{n} h_1(M_1,i)h_2(M_2,i)$
- compute $S_{2,1} = \sum_{i=1}^{n} h_2(M_1,i)h_1(M_2,i)$
- compute $S_{2,2} = \sum_{i=1}^{n} h_2(M_1,i)h_2(M_2,i)$
- **if** $(e(\pi_1, g) = e(g_1^{S_{1,1}} g_2^{S_{1,2}} g_3^{res_1}, pk_2)$,
  $e(\pi_2, g) = e(g_1^{S_{2,1}} g_2^{S_{2,2}} g_3^{res_2}, pk_2)$,
  $e(\pi_3, g) = e(g_1^{res_1} g_2^{res_2} g_3^{res}, pk_1))$ **then**
  - output 1
- **else**
  - output 0
- **end if**

Fig. 3. Publicly verifiable computation for inner product query

$\sum_{i=1}^{n} h_1(M_1,i)\mathcal{X}_{2,i}$ and $res_2 = \sum_{i=1}^{n} h_2(M_2,i)\mathcal{X}_{2,i}$. Then, the server can send (res_1, res_2) to the client along with their proofs (π_1, π_2) to guarantee their authenticity. Note that the auxiliary information S_Δ can be pre-computed to accelerate the verification process, because S_Δ is uncorrelated with $\mathcal{X}_1$ and $\mathcal{X}_2$.

Correctness. We prove the correctness of the verification algorithm according to the following three steps.

i. If res_1 is valid, then the equation $e(\pi_1, g) = e(g_1^{S_{1,1}} g_2^{S_{1,2}} g_3^{res_1}, pk_2)$ holds.

$$\begin{aligned} & e(\pi_1, g) \\ = \; & e(\prod_{i=1}^{n} \sigma_{2,i}^{h_1(M_1,i)}, g) \\ = \; & e(\prod_{i=1}^{n} (g_1^{h_1(M_2,i)} g_2^{h_2(M_2,i)} g_3^{\mathcal{X}_{2,i}})^{h_1(M_1,i)}, g^{sk_2}) \\ = \; & e(g_1^{S_{1,1}} g_2^{S_{1,2}} g_3^{res_1}, pk_2) \end{aligned} \quad (5)$$

ii. If res_2 is valid, then the equation $e(\pi_2, g) =$

$e(g_1^{S_{2,1}} g_2^{S_{2,2}} g_3^{res_2}, pk_2)$ holds.

$$\begin{aligned} & e(\pi_2, g) \\ = \ & e(\prod_{i=1}^{n} \sigma_{2,i}^{h_2(M_1,i)}, g) \\ = \ & e(\prod_{i=1}^{n} (g_1^{h_1(M_2,i)} g_2^{h_2(M_2,i)} g_3^{\mathcal{X}_{2,i}})^{h_2(M_1,i)}, g^{sk_2}) \\ = \ & e(g_1^{S_{2,1}} g_2^{S_{2,2}} g_3^{res_2}, pk_2) \end{aligned} \tag{6}$$

iii. If res is valid, then the equation $e(\pi_3, g) = e(g_1^{res_1} g_2^{res_2} g_3^{res}, pk_1)$ holds.

$$\begin{aligned} & e(\pi_3, g) \\ = \ & e(\prod_{i=1}^{n} \sigma_{1,i}^{\mathcal{X}_{2,i}}, g) \\ = \ & e(\prod_{i=1}^{n} g_1^{h_1(M_1,i)\mathcal{X}_{2,i}} g_2^{h_2(M_1,i)\mathcal{X}_{2,i}} g_3^{\mathcal{X}_{1,i} \cdot \mathcal{X}_{1,i}}, g^{sk_1}) \\ = \ & e(g_1^{res_1} g_2^{res_2} g_3^{res}, pk_1) \end{aligned} \tag{7}$$

Discussion. The storage size and computation overhead of each data source are the same as in the group-by sum case. To compute a proof π, the server needs $O(n)$ modular exponentiations in G_1, $O(n)$ modular multiplications in G_1, $O(n)$ hash operations, $O(n)$ modular additions and multiplications in Z_q^*. The proof includes two elements in Z_q^* and three elements in G_1. With the auxiliary information S_Δ, the computation cost for the client to verify the proof includes six parings, nine modular exponentiations and six multiplications in G_1.

As for the case without outsourcing, each machine M_j needs to store $O(n)$ elements in Z_q^* for $\mathcal{X}_j$. We assume that machines are not required to directly communicate with each other. Thus, a client need first receive $\mathcal{X}_1$ and $\mathcal{X}_2$ from M_1 and M_2 respectively, and then compute $\mathcal{X}_1 \otimes \mathcal{X}_2$ by himself/herself. The communication cost is $O(n)$, and the computation includes $O(n)$ modular additions and multiplications in Z_q^*. In contrast, it only incurs constant communication and computation overhead in the outsourcing case.

5 Matrix Product Query Extension

In this section, we extend the publicly verifiable inner product evaluation scheme to support *matrix product query* under the multi-key setting. Specifically, machine M_1 (M_2) generates a row vector $\overrightarrow{a_i}$ (a column vector $\overrightarrow{b_i}$) with m entries at time i and outsources it to the server. Let matrix A (B) denote the data stream outsourced by machine M_1 (respectively, M_2) up to the current time n, where

$$A = \begin{bmatrix} \overrightarrow{a_1} \\ \overrightarrow{a_2} \\ ... \\ \overrightarrow{a_n} \end{bmatrix}, B = [\overrightarrow{b_1}\overrightarrow{b_2}...\overrightarrow{b_n}]. \tag{8}$$

A client requests the server to compute the matrix product $\mathcal{F_{MP}} = A \times B$, i.e.,

$$A \times B = \begin{bmatrix} \overrightarrow{a_1} \otimes \overrightarrow{b_1} & \overrightarrow{a_1} \otimes \overrightarrow{b_2} & ... & \overrightarrow{a_1} \otimes \overrightarrow{b_n} \\ \overrightarrow{a_2} \otimes \overrightarrow{b_1} & \overrightarrow{a_2} \otimes \overrightarrow{b_2} & ... & \overrightarrow{a_2} \otimes \overrightarrow{b_n} \\ ... & ... & ... & ... \\ \overrightarrow{a_n} \otimes \overrightarrow{b_1} & \overrightarrow{a_n} \otimes \overrightarrow{b_2} & ... & \overrightarrow{a_n} \otimes \overrightarrow{b_n} \end{bmatrix} \tag{9}$$

In the above equation, $\overrightarrow{a_i} \otimes \overrightarrow{b_j}$ denotes the inner product of vectors $\overrightarrow{a_i}$ and $\overrightarrow{b_j}$.

To provide a proof of the matrix product computation, a possible approach is to directly extend the inner product verification algorithm. Let $res[i][j] = \overrightarrow{a_i} \otimes \overrightarrow{b_j}$ represent the (i^{th}, j^{th}) entry of the matrix $A \times B$. The server can first use the inner product algorithm to generate a proof $\pi_{i,j}$ for $res[i][j]$ and then send all the proofs $\pi_{i,j}(1 \le i \le n, 1 \le j \le n)$ to the client. However, this naive solution may be prohibitive as the proof size is $O(n^2)$ with large n.

In the following, we present a verification algorithm allowing the server to provide a fixed-size proof. The server first generates proofs for each entry of the matrix $A \times B$ and then combines these proofs together. Similar to the verification inner product query, the verifiable matrix product computation scheme includes the following phases.

$\textbf{KeyGen}(1^\kappa)$: Each machine M_i chooses a random number $sk_i = s_i \in Z_q^*$ as its secret key and computes the corresponding public key $pk_i = g^{sk_i}$.

$\textbf{TagGen}(sk_1, i, \overrightarrow{a_i})$: Machine M_1 uses this algorithm to generate tags for a vector $\overrightarrow{a_i} = (a_{i,1}, a_{i,2}, ..., a_{i,m})$ with m entries. Specifically, for $k = 1$ to m, machine M_1 computes

$$\mu_{i,k} = (g_1^{h_1(M_1,i,k)} g_2^{h_2(M_1,i,k)} g_3^{a_{i,k}})^{sk_1} \tag{10}$$

Finally, it sends $\overrightarrow{a_i}$ along with a row vector $\overrightarrow{\mu_i} = (\mu_{i,1}, \mu_{i,2}, ..., \mu_{i,m})$ to the server.

$\textbf{TagGen}(sk_2, j, \overrightarrow{b_j})$: Machine M_2 uses this algorithm to generate tags for a vector $\overrightarrow{b_j} = (b_{1,j}, b_{2,j}, ..., b_{m,j})^T$ with m entries. Specifically, for $k = 1$ to m, machine M_2 computes

$$\nu_{j,k} = (g_1^{h_1(M_2,j,k)} g_2^{h_2(M_2,j,k)} g_3^{b_{k,j}})^{sk_2} \tag{11}$$

Finally, it sends $\overrightarrow{b_i}$ along with a column vector $\overrightarrow{\nu_j} = (\nu_{j,1}, \nu_{j,2}, ..., \nu_{j,m})^T$ to the server.

$\textbf{Evaluate}(\mathcal{F}_{MP}, A, B)$: After receiving the matrix product query, the server computes $res = A \times B$ and returns the result res to the client.

After the receipt of res, the client chooses a random number $\lambda \in Z_q^*$ and sends it to the server.

$\textbf{GenProof}(\mathcal{F_{MP}}, \overrightarrow{\mu}, \overrightarrow{\nu}, A, B)$: Upon receiving λ, the server runs this algorithm to generate a proof π for the computation result as follows.

Step 1.The server computes a proof $\pi[i][j]$ for each entry $res[i][j]$ of $A \times B$ as follows.

$$\begin{cases} \pi[i][j]_1 = \prod_{k=1}^{m} \nu_{k,j}^{h_1(M_1,i,k)} \\ \pi[i][j]_2 = \prod_{k=1}^{m} \nu_{k,j}^{h_2(M_1,i,k)} \\ \pi[i][j]_3 = \prod_{k=1}^{m} \mu_{i,k}^{b_{k,j}} \\ res[i][j]_1 = \sum_{k=1}^{m} h_1(M_1,i,k) b_{k,j} \\ res[i][j]_2 = \sum_{k=1}^{m} h_2(M_1,i,k) b_{k,j} \end{cases} \tag{12}$$

Step 2.The server combines proofs $\pi[i][j](1 \le i \le n, 1 \le j \le n)$ together.

$$\begin{cases} \pi_1 = \prod_{i=1}^n \prod_{j=1}^n \pi[i][j]_1^{f_\lambda(i,j)} \\ \pi_2 = \prod_{i=1}^n \prod_{j=1}^n \pi[i][j]_2^{f_\lambda(i,j)} \\ \pi_3 = \prod_{i=1}^n \prod_{j=1}^n \pi[i][j]_3^{f_\lambda(i,j)} \\ res_1 = \sum_{i=1}^n \sum_{j=1}^n f_\lambda(i,j) res[i][j]_1 \\ res_2 = \sum_{i=1}^n \sum_{j=1}^n f_\lambda(i,j) res[i][j]_2 \end{cases} \quad (13)$$

In the end, the sever sends the proof $\pi = \{\pi_1, \pi_2, \pi_3, res_1, res_2\}$ to the client.

CheckProof$(\mathcal{F}_{\mathcal{MP}}, pk_1, pk_2, res, \pi)$: The client runs this algorithm to check the validity of the computation result.

Step 1.The client first computes auxiliary information for each entry $res[i][j]$. Note that the auxiliary information can be pre-computed to speed up the verification process, since it is independent of matrices A and B.

$$\begin{cases} S[i][j]_{1,1} = \sum_{k=1}^m h_1(M_1, i, k) h_1(M_2, j, k) \\ S[i][j]_{1,2} = \sum_{k=1}^m h_1(M_1, i, k) h_2(M_2, j, k) \\ S[i][j]_{2,1} = \sum_{k=1}^m h_2(M_1, i, k) h_1(M_2, j, k) \\ S[i][j]_{2,2} = \sum_{k=1}^m h_2(M_1, i, k) h_2(M_2, j, k) \end{cases} \quad (14)$$

$$\begin{cases} S_{1,1} = \sum_{i=1}^n \sum_{j=1}^n f_\lambda(i,j) S[i][j]_{1,1} \\ S_{1,2} = \sum_{i=1}^n \sum_{j=1}^n f_\lambda(i,j) S[i][j]_{1,2} \\ S_{2,1} = \sum_{i=1}^n \sum_{j=1}^n f_\lambda(i,j) S[i][j]_{2,1} \\ S_{2,2} = \sum_{i=1}^n \sum_{j=1}^n f_\lambda(i,j) S[i][j]_{2,2} \end{cases} \quad (15)$$

Step 2.Let $\omega = \sum_{i=1}^n \sum_{j=1}^n f_\lambda(i,j) res[i][j]$. If the following three equations hold, the client accepts the computation result res. Otherwise, the client rejects it.

$$\begin{cases} e(\pi_1, g) = e(g_1^{S_{1,1}} g_2^{S_{1,2}} g_3^{res_1}, pk_2) \\ e(\pi_2, g) = e(g_1^{S_{2,1}} g_2^{S_{2,2}} g_3^{res_2}, pk_2) \\ e(\pi_3, g) = e(g_1^{res_1} g_2^{res_2} g_3^{\omega}, pk_1) \end{cases} \quad (16)$$

Correctness. We prove the correctness of the verification algorithm in three steps.

i. If res_1 is valid, then the equation $e(\pi_1, g) = e(g_1^{S_{1,1}} g_2^{S_{1,2}} g_3^{res_1}, pk_2)$ holds.

$$\begin{aligned} & e(\pi_1, g) \\ = \; & e(\prod_{i=1}^n \prod_{j=1}^n \pi[i][j]_1^{f_\lambda(i,j)}, g) \\ = \; & e(\prod_{i=1}^n \prod_{j=1}^n \prod_{k=1}^m \nu_{k,j}^{h_1(M_1,i,k) f_\lambda(i,j)}, g) \\ = \; & e(\prod_{i=1}^n \prod_{j=1}^n (g_1^{S[i][j]_{1,1}} g_2^{S[i][j]_{1,2}} g_3^{res[i][j]_1})^{f_\lambda(i,j)}, pk_2) \\ = \; & e(g_1^{S_{1,1}} g_2^{S_{1,2}} g_3^{res_1}, pk_2) \end{aligned}$$

ii. If res_2 is valid, then the equation $e(\pi_2, g) = e(g_1^{S_{2,1}} g_2^{S_{2,2}} g_3^{res_2}, pk_2)$ holds.

$$\begin{aligned} & e(\pi_2, g) \\ = \; & e(\prod_{i=1}^n \prod_{j=1}^n \pi[i][j]_2^{f_\lambda(i,j)}, g) \\ = \; & e(\prod_{i=1}^n \prod_{j=1}^n \prod_{k=1}^m \nu_{k,j}^{h_2(M_1,i,k) f_\lambda(i,j)}, g) \\ = \; & e(\prod_{i=1}^n \prod_{j=1}^n (g_1^{S[i][j]_{2,1}} g_1^{S[i][j]_{2,2}} g_3^{res[i][j]_2})^{f_\lambda(i,j)}, pk_2) \\ = \; & e(g_1^{S_{2,1}} g_2^{S_{2,1}} g_3^{res_2}, pk_2) \end{aligned}$$

iii. If res is valid, then the equation $e(\pi_3, g) = e(g_1^{res_1} g_2^{res_2} g_3^{\omega}, pk_1)$ holds.

$$\begin{aligned} & e(\pi_3, g) \\ = \; & e(\prod_{i=1}^n \prod_{j=1}^n \pi[i][j]_3^{f_\lambda(i,j)}, g) \\ = \; & e(\prod_{i=1}^n \prod_{j=1}^n \prod_{k=1}^m \mu_{i,k}^{b_{k,j} f_\lambda(i,j)}, g) \\ = \; & e(\prod_{i=1}^n \prod_{j=1}^n (g_1^{res[i][j]_1} g_2^{res[i][j]_2} g_3^{res[i][j]})^{f_\lambda(i,j)}, pk_1) \\ = \; & e(g_1^{res_1} g_2^{res_2} g_3^{\omega}, pk_1) \end{aligned}$$

Discussion: The verification of matrix product is an interactive protocol since the client needs to send a challenge λ after receiving the result res. The server then provides a proof for res based on the challenge λ. Finally, the validity of res can be inspected through equation (16). We stress that λ cannot be transferred to the server before receiving res. Otherwise, given λ, the server can easily forge a result res' satisfying $\sum_{i=1}^n \sum_{j=1}^n f_\lambda(i,j) res'[i][j] = \sum_{i=1}^n \sum_{j=1}^n f_\lambda(i,j) res[i][j]$.

In the computation of inner product and matrix product, we evaluate functions over the entire outsourced streams. It is worth noting that the function can take any portion of the data streams as input.

Machine M_1 needs $O(mn)$ modular exponentiations, multiplications in G_1, and $O(mn)$ hash operations to generate tags for an $n \times m$ matrix A. Similar to the construction for group-by sum query, these tags are computed only once. The storage cost for the tags includes $O(mn)$ elements in G_1 at the server side. The auxiliary information $\pi_{i,j}(1 \le i \le n)(1 \le i \le n)$ for the generation of a proof at the server side include $O(n^2)$ elements in G_1 and $O(n^2)$ elements in Z_q^*, which has the same storage complexity with the computation result $A \times B$. In other words, the proof generation does not introduce the extra storage overhead. To compute a proof for $A \times B$, the server performs $O(mn^2)$ modular exponentiations, multiplications in G_1, $O(mn^2)$ modular additions, multiplications in Z_q^*, $O(mn)$ hash and $O(n^2)$ PRF operations. The proof π consists of three elements in G_1 and two elements in Z_q^*. Finally, the client performs six pairings, nine modular exponentiations, six modular multiplications in G_1, $O(n^2)$ modular additions and multiplications in Z_q^* to verify the proof. Without outsourcing, M_1 (M_2) has to store its matrix locally. The communication cost for transmitting a matrix includes $O(mn)$ elements in Z_q^*. Further more, the client compute $A \times B$ with super-quadratic complexity.

6 Security Analysis

In this section, we prove the security of the proposed scheme in the random oracle model.

Theorem 6.1. *Under the CDH assumption, the publicly verifiable computation scheme for group-by sum query is secure against an adaptive chosen-message attack in the random oracle model.*

Proof: The security definition of the publicly verifiable computation scheme for group-by sum query is similar to definition 3.2, except that adversary $\mathcal{A}$ forges a result $res \neq \sum_{i\in\Delta} \mathcal{X}_{j,i}$ and passes the verification. Now, we show how to construct an adversary $\mathcal{B}$ that uses $\mathcal{A}$ to solve the CDH problem. That is, given a CDH tuple (g, g^{s_j}, g_3), the adversary $\mathcal{B}$ is able to compute $g_3^{s_j}$ with non-negligible probability.

$\mathcal{B}$ simulates a publicly verifiable computation scheme with group-by sum query for $\mathcal{A}$ as follows.

Setup: The adversary $\mathcal{B}$ sets machine M_j's public key $pk_j = g^{s_j}$, $g_1 = g \cdot g_3^{\alpha}$ and $g_2 = (g \cdot g_3)^{\beta}$, where α and β are two random numbers in Z_q^*. The system parameters and the public key are given to the adversary $\mathcal{A}$.

Query: The adversary $\mathcal{A}$ adaptively queries $\mathcal{B}$ for tags on the discrete time and data of its choice. Specifically, $\mathcal{A}$ sends a tuple $(M_j, \mathcal{X}_{j,1}, 1)$ to $\mathcal{B}$. The algorithm $\mathcal{B}$ generates a tag $\sigma_{j,1}$ and sends it back to $\mathcal{A}$. $\mathcal{A}$ can continually make tag queries to $\mathcal{B}$ for the tags on $(M_j, \mathcal{X}_{j,2}, 2)$, $(M_j, \mathcal{X}_{j,3}, 3)$,..., $(M_j, \mathcal{X}_{j,n}, n)$ of its choice. The only restriction is that $\mathcal{A}$ cannot make tag queries for two different data values using the same discrete time i. $\mathcal{B}$ answers $\mathcal{A}$'s queries as follows:

$\mathcal{B}$ first initializes an empty list L to record the tuples $(M_j, \mathcal{X}_{j,i}, i, \gamma_{j,i}, \sigma_{j,i})$. After receiving a tag query, the adversary $\mathcal{B}$ processes the followings:

- If $(\mathcal{X}_{j,i}, i)$ has been queried before, $\mathcal{B}$ retrieves the tuple $\mathcal{X}_{j,i}, i, \gamma_{j,i}, \sigma_{j,i}$ from the list L and returns $\sigma_{j,i}$ to $\mathcal{A}$.
- If i has not been queried, $\mathcal{B}$ selects a random number $\gamma_{j,i}$ from Z_q^* and sets $\sigma_{j,i} = pk_j^{\gamma_{j,i}} = g^{s_j \cdot \gamma_{j,i}}$. Then $\mathcal{B}$ adds $(\mathcal{X}_{j,i}, i, \gamma_{j,i}, \sigma_{j,i})$ into the list L and returns $\sigma_{j,i}$ to $\mathcal{A}$.
- Otherwise, i.e., i has been queried but $(\mathcal{X}_{j,i}, i) \notin L$, $\mathcal{B}$ rejects this query.

In addition, $\mathcal{B}$ returns $h_1(M_j, i) = \frac{\mathcal{X}_{j,i}+\gamma_{j,i}}{1-\alpha}$ and $h_2(M_j, i) = \frac{\mathcal{X}_{j,i}+\alpha\gamma_{j,i}}{(\alpha-1)\beta}$ to $\mathcal{A}$ for the hash queries. We can observe that the tag $\sigma_{j,i} = g^{s_j \cdot \gamma_{j,i}}$ on $(\mathcal{X}_{j,i}, i)$ is valid under the public key $pk_j = g^{s_j}$, this is because of the following relationship.

$$
\begin{aligned}
& e(g_1^{h_1(M_j,i)} g_2^{h_2(M_j,i)} g_3^{\mathcal{X}_{j,i}}, pk_j) \\
= \; & e(g^{h_1(M_j,i)} g_3^{\alpha h_1(M_j,i)} (g \cdot g_3)^{\beta h_2(M_j,i)} g_3^{\mathcal{X}_{j,i}}, pk_j) \\
= \; & e(g^{\gamma_{j,i}}, g^{s_j}) \\
= \; & e(g^{s_j \cdot \gamma_{j,i}}, g)
\end{aligned}
\tag{17}
$$

Request: $\mathcal{B}$ requests the adversary $\mathcal{A}$ to compute $\sum_{i\in\Delta} \mathcal{X}_{j,i}$ by sending a time set Δ.

Forge: $\mathcal{A}$ returns a computation result res together with a proof π. Note that π is a valid proof that passes algorithm **CheckProof**, but $res \neq \sum_{i\in\Delta} \mathcal{X}_{j,i}$. Thus, we have $\pi = (g_1^{S_1} g_2^{S_2} g_3^{res})^{s_j}$, where $S_1 = \sum_{i\in\Delta} h_1(M_j, i)$ and $S_2 = \sum_{i\in\Delta} h_2(M_j, i)$. Let $res' = \sum_{i\in\Delta} \mathcal{X}_{j,i}$ be the real result, we can obtain

$$
\begin{aligned}
\pi = \; & (g_1^{S_1} g_2^{S_2} g_3^{res})^{s_j} \\
= \; & (g_1^{\sum_{i\in\Delta} h_1(M_j,i)} g_2^{\sum_{i\in\Delta} h_2(M_j,i)} g_3^{res})^{s_j} \\
= \; & ((g \cdot g_3^{\alpha})^{\sum_{i\in\Delta} h_1(M_j,i)} (g \cdot g_3)^{\beta \sum_{i\in\Delta} h_2(M_j,i)} g_3^{res})^{s_j} \\
= \; & (g^{\sum_{i\in\Delta} h_1(M_j,i) + \beta \sum_{i\in\Delta} h_2(M_j,i)} \\
& \cdot g_3^{\alpha \sum_{i\in\Delta} h_1(M_j,i) + \beta \sum_{i\in\Delta} h_2(M_j,i) + res})^{s_j} \\
= \; & (g^{\sum_{i\in\Delta} \gamma_{j,i}} g_3^{res - res'})^{s_j} \\
= \; & (g^{s_j \sum_{i\in\Delta} \gamma_{j,i}} g_3^{s_j (res - res')})^{s_j}
\end{aligned}
$$

Since $res' \neq res$, $\mathcal{B}$ can compute $g_3^{s_j} = (\frac{\pi}{g^{s_j \sum_{i\in\Delta} \gamma_{j,i}}})^{(res - res')^{-1}}$ from the above equation. The interactions of $\mathcal{A}$ with $\mathcal{B}$ are indistinguishable to $\mathcal{A}$ from interactions with an honest challenger in the experiment, as $\mathcal{B}$ chooses all parameters according to our scheme. Therefore, our scheme is secure against an adaptive chosen-message attack in the random oracle model under the CDH assumption. □

Theorem 6.2. *Under the CDH assumption, the public verifiable tag is unforgeable, i.e., no source can deny his/her tags that have been outsourced to the server.*

Proof: The proof is similar to that for **Theorem 6.1**, except that adversary $\mathcal{A}$ generates a valid tag $\sigma_{j,n+1} = (g_1^{r_1} g_2^{r_2} g_3^{\mathcal{X}_{j,n+1}})^{s_j}$ on data $\mathcal{X}_{j,n+1}$ at time $n+1$, where r_1 and r_2 are the random values returned to $\mathcal{A}$ for hash queries $h_1(M_j, n+1)$ and $h_2(M_j, n+1)$ in **Query** phase. Given $\sigma_{j,n+1}$, adversary $\mathcal{B}$ is able to compute $g_3^{s_j} = (\frac{\sigma_{j,n+1}}{g^{(r_1+r_2)s_j}})^{(\alpha r_1 + \beta r_2 + \mathcal{X}_{j,n+1})^{-1}}$, which contradicts the CDH assumption. Therefore, no source can deny his/her tags outsourced to the server. □

Before proving the security of our publicly verifiable computation scheme with *inner product query*, we give the following two lemmas.

Lemma 6.3. *If π_1 can pass the verification, then res_1 is valid.*

Proof: Given a CDH tuple (g, g^s, g_3), $\mathcal{B}$ simulates a publicly verifiable computation scheme with inner product query for $\mathcal{A}$ as follows.

Setup: The adversary $\mathcal{B}$ sets machine M_1's public key $pk_1 = g^s$, machine M_2's public key $pk_2 = g^{\delta s}$, $g_1 = g \cdot g_3^{\alpha}$ and $g_2 = (g \cdot g_3)^{\beta}$, where α, β and δ are three random numbers in Z_q^*. The system parameters and the public keys are given to the adversary $\mathcal{A}$.

Query: The adversary $\mathcal{A}$ adaptively queries $\mathcal{B}$ for tags on the discrete time and data of its choice.

For the query $(M_1, \mathcal{X}_{1,1}, 1)$,...,$(M_1, \mathcal{X}_{1,n}, n)$, $\mathcal{B}$ proceeds as follows:

$\mathcal{B}$ first initializes an empty list L_1 to record the tuples $(M_1, \mathcal{X}_{j,i}, i, \gamma_{1,i}, \sigma_{j,i})$. After receiving a tag query, the adversary $\mathcal{B}$ processes the followings:

- If $(M_1, \mathcal{X}_{1,i}, i)$ has been queried before, $\mathcal{B}$ retrieves the tuple $\gamma_{1,i}, \sigma_{1,i}$ from the list L_1 and returns $\sigma_{1,i}$ to $\mathcal{A}$.
- If i has not been queried, $\mathcal{B}$ selects a random number $\gamma_{1,i}$ from Z_q^* and sets $\sigma_{1,i} = pk_1^{\gamma_{1,i}} = g^{s\cdot\gamma_{1,i}}$. Then $\mathcal{B}$ adds $(M_1, \mathcal{X}_{1,i}, i, \gamma_{1,i}, \sigma_{1,i})$ into the list L_1 and returns $\sigma_{1,i}$ to $\mathcal{A}$.
- Otherwise, i.e., i has been queried but $(M_1, \mathcal{X}_{1,i}, i) \notin L_1$, $\mathcal{B}$ rejects this query.

In addition, $\mathcal{B}$ returns $h_1(M_1, i) = \frac{\mathcal{X}_{1,i}+\gamma_{1,i}}{1-\alpha}$ and $h_2(M_1, i) = \frac{\mathcal{X}_{1,i}+\alpha\gamma_{1,i}}{(\alpha-1)\beta}$ to $\mathcal{A}$ for the hash queries. We can observe that the tag $\sigma_{1,i} = g^{s\cdot\gamma_{1,i}}$ on $(\mathcal{X}_{1,i}, i)$ is valid under the public key $pk_1 = g^s$, this is because of the following relationship.

$$\begin{aligned}
& e(g_1^{h_1(M_1,i)} g_2^{h_2(M_1,i)} g_3^{\mathcal{X}_{1,i}}, pk_1) \\
= \ & e(g^{h_1(M_1,i)} g_3^{\alpha h_1(M_1,i)} (g\cdot g_3)^{\beta h_2(M_1,i)} g_3^{\mathcal{X}_{1,i}}, pk_1) \\
= \ & e(g^{\gamma_{1,i}}, g^s) \\
= \ & e(g^{s\cdot\gamma_{1,i}}, g)
\end{aligned} \tag{18}$$

For the query $(M_2, \mathcal{X}_{2,1}, 1)$,...,$(M_2, \mathcal{X}_{2,n}, n)$, $\mathcal{B}$ initializes an empty list L_2 to record the tuples $(M_2, \mathcal{X}_{2,i}, i, \gamma_{2,i}, \sigma_{2,i})$. When receiving a tag query, the adversary $\mathcal{B}$ processes as below:

- If $(M_2, \mathcal{X}_{2,i}, i)$ has been queried before, $\mathcal{B}$ retrieves the tuple $\gamma_{2,i}, \sigma_{2,i}$ from the list L_2 and returns $\sigma_{2,i}$ to $\mathcal{A}$.
- If i has not been queried, $\mathcal{B}$ selects a random number $\gamma_{2,i}$ from Z_q^* and computes $\sigma_{2,i} = pk_2^{\gamma_{2,i}} = g^{s\cdot\delta\cdot\gamma_{2,i}}$. Then algorithm $\mathcal{B}$ adds $(M_2, \mathcal{X}_{2,i}, i, \gamma_{2,i}, \sigma_{2,i})$ into the list L_2 and returns $\gamma_{2,i}$ to $\mathcal{A}$.
- Otherwise, i.e., i has been queried but $(M_2, \mathcal{X}_{2,i}, i) \notin L_2$, $\mathcal{B}$ rejects this query.

In addition, $\mathcal{B}$ returns $h_1(M_2, i) = \frac{\mathcal{X}_{2,i}+\gamma_{2,i}}{1-\alpha}$ and $h_2(M_2, i) = \frac{\mathcal{X}_{2,i}+\alpha\gamma_{2,i}}{(\alpha-1)\beta}$ to $\mathcal{A}$ for the hash queries. Similarly, we can observe that the tag $\sigma_{2,i} = g^{s\cdot\delta\cdot\gamma_{2,i}}$ on $(\mathcal{X}_{2,i}, i)$ is valid under the public key $pk_2 = g^{\delta s}$.
Request: $\mathcal{B}$ requests the adversary $\mathcal{A}$ to compute $\sum_{i=1}^n \mathcal{X}_{1,i}\cdot\mathcal{X}_{2,i}$.

Forge: $\mathcal{A}$ returns a computation result res together with a proof $\pi = res_1, res_2, \pi_1, \pi_2, \pi_3$.

Note that π_1 is a valid proof that passes algorithm **CheckProof**, but res_1 is a forged one. That is,

$$\begin{cases}
\pi_1 = (g_2^{S_{1,2}} g_1^{S_{1,1}+res_1})^{s\gamma} \\
S_{1,1} = \sum_{i=1}^n h_1(M_1,i)h_1(M_2,i) \\
S_{1,2} = \sum_{i=1}^n h_1(M_1,i)h_2(M_2,i) \\
res_1 \neq \sum_{i=1}^n h_1(M_1,i)\mathcal{X}_{2,i}
\end{cases} \tag{19}$$

Let $res_1^{'} = \sum_{i=1}^n h_1(M_1,i)\mathcal{X}_{2,i}$ be the real result, we can obtain

$$\begin{aligned}
\pi = \ & (g_1^{S_{1,1}} g_2^{S_{1,2}} g_3^{res})^{\delta s} \\
= \ & (g^{S_{1,1}+\beta S_{1,2}} g_3^{\alpha S_{1,1}+\beta S_{1,2}+res})^{\delta s} \\
= \ & (g^{\sum_{i=1}^n \frac{r_{2,i}(r_{1,i}+x_{1,i})}{1-\alpha}} g_3^{res-\sum_{i=1}^n \frac{x_{2,i}(r_{1,i}+x_{1,i})}{1-\alpha}})^{\delta s} \\
= \ & (g^{\sum_{i=1}^n \frac{r_{2,i}(r_{1,i}+x_{1,i})}{1-\alpha}} g_3^{res-\sum_{i=1}^n h_1(M_1,i)x_{1,i}})^{\delta s} \\
= \ & g^{\delta s\sum_{i=1}^n r_{2,i}h_1(M_1,i)} g_3^{(res-res^{'})\delta s}
\end{aligned}$$

Since $res^{'} \neq res$, $\mathcal{B}$ can compute $g_3^s = (\frac{\pi_1}{g^{\delta s\sum_{i=1}^n r_{2,i}h_1(M_1,i)}})^{(\delta(res-res^{'}))^{-1}}$ from the above equation. Obviously, this conflicts the CDH assumption. Similarly, if π_2 is valid, then the result res_2 is correct. □

Lemma 6.4. *If res_1, res_2 are valid and π_3 can pass the verification, then res is valid.*

Proof: The proof of this lemma directly follows the previous proofs. In the forge phase, the adversary $\mathcal{A}$ outputs a valid tuple (res, π) but with $res \neq \sum_{i=1}^n \mathcal{X}_{1,i}\cdot\mathcal{X}_{2,i}$. Thus, we have $\pi = (g_1^{res_1} g_2^{res_2} g_3^{res})^s$. Let $res^{'} = \sum_{i=1}^n \mathcal{X}_{1,i}\cdot\mathcal{X}_{2,i}$, then we have

$$\begin{aligned}
\pi = \ & (g_1^{res_1} g_2^{res_2} g_3^{res})^s \\
= \ & (g^{res_1} g_3^{\alpha\cdot res_1} g^{\beta\cdot res_2} g_3^{\beta\cdot res_2} g_3^{res})^s \\
= \ & g^{s(res_1+\beta\cdot res_2)} g_3^{s(\alpha\cdot res_1+\beta\cdot res_2+res)} \\
= \ & g^{s(res_1+\beta\cdot res_2)} g_3^{s(res-res^{'})}
\end{aligned}$$

Since $res^{'} \neq res$, $\mathcal{B}$ can compute $g_3^s = (\frac{\pi}{g^{s(res_1+\beta\cdot res_2)}})^{(res-res^{'})^{-1}}$ from the above equation. □

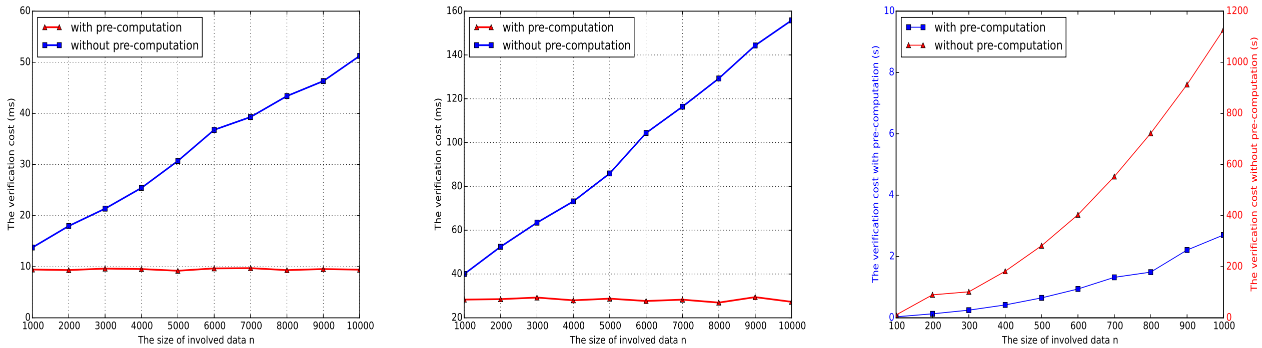
Theorem 6.5. *Under the CDH assumption, the publicly verifiable computation scheme for inner product query is secure against an adaptive chosen-message attack in the random oracle model.*

Proof: The desired security property can be proved directly from lemma 6.3 and lemma 6.4. □

Theorem 6.6. *Under the assumption that f is a PRF and CDH problem is hard, the publicly verifiable computation scheme for matrix product query is secure against an adaptive chosen-message attack in the random oracle model.*

Proof: In our construction of matrix product computation verification, the server first follows the computation of inner product verification to generate proofs for each entry of the matrix $A\times B$ and then combines these proofs together. Thus, we directly follows the proof of Theorem 5.4. The main difference is that we need to ensure that each entry of the computation result res should be true.

We assume that $k_{i,j} \in Z_q^*(1\leq i\leq n, 1\leq j\leq n)$ is generated via a truly random function $f^{'}$ instead of PRF f. By applying the same simulation shown in lemma 5.3 and lemma 5.4, we obtain that res_1, res_2 and $\omega = \sum_{i=1}^n\sum_{j=1}^n k_{i,j}\cdot res[i][j]$ are valid if the proof π passes the verification. Consider the following



(a) Group-by Sum Verification (b) Inner Product Verification (c) Matrix Product Verification

Fig. 4. Comparison of the verification costs between with pre-computation and without pre-computation

multivariate polynomial in finite field Z_q^*:

$$P(res) = k_{1,1} \cdot res[1][1] + ... + k_{n,n} \cdot res[n][n] - \omega \quad (20)$$

Note that adversary $\mathcal{A}$ forging res correctly is equivalent to finding res such that $P(res, \omega) = 0$. However, due to Lemma 1 in [35], for any (non-zero) multivariate polynomial P in Z_q^* of degree d (in our case $d = 1$) and randomly chosen $res[1][1], ..., res[n][n]$ with unknown coefficients $k_{1,1}, ..., k_{1,n}$, the probability that $P(res) = 0$ is $\frac{d}{q} = \frac{1}{q}$. Thus, the probability that adversary $\mathcal{A}$ forges a valid result res is negligible. □

7 Evaluation

This section evaluates the practical performance of our scheme. We conduct the computation at client-side by using JPBC library [36] in Eclipse 4.2 on a Windows 7 machine with 2.30 GHz Intel Core I7-3615QM. The cloud-side computation overhead is evaluated on an IBM System x3550 M4 machine. We choose type-A (symmetric) pairings with 80-bit security in our simulation, which results in the element in G_1 and Z_q^* to be 512-bit and 160-bit, respectively. Note that our scheme can also be implemented under the asymmetric pairings.

7.1 Storage

In our scheme, data sources store their public/private keys and system parameters locally while outsourcing all the data along with the corresponding tags to a third-party server. The size of a public and private key pair $(pk_j \in G_1, sk_j \in Z_q^*)$ is 84 bytes. The size of system parameters $\{\mathcal{G}, g, g_1, g_2, h_1(), h_2()\}$ is constant, regardless of data streams' size. The public keys of data sources dominates the client's storage. Assuming that there are 100 data sources in the system, the total storage on the client side is 6400 bytes. Thus, we observe that the storage overhead on data owners and clients are much smaller than the outsourced data streams.

7.2 Communication

We do not take the communication cost of query and the computation result into account, since they also occurs in the scenario without outsourcing. On receiving a computation query from the client, the cloud evaluates the corresponding function and generates a proof to ensure the validity of the computation result. The proof $\pi \in G_1$ for the group-by sum query is 64 bytes. For both inner product and matrix queries, the proofs $\pi = \{res_1, res_2, \pi_1, \pi_2, \pi_3\} \in Z_q^{*2} \times G_1^3$ are 232 bytes. Thus, the communication cost is constant in our scheme, regardless of the input size of the evaluated function.

7.3 Computation

Data source side. Generating a tag for a data value needs three exponentiation operations in G_1, two modular multiplications in G_1 and two hashes, which takes about 2.25 ms.

Client side. Figs 4.a and 4.b show the verification cost for group-by sum and inner product queries, respectively. Note that the auxiliary information S_Δ in the verification can be pre-computed, because they are only determined by S_Δ, i.e., independent of the outsourced data. Thus, with the aid of such pre-computation, the verification cost is constant, regardless of the input size n.

For simplicity, we consider the product of two $n \times n$ matrices, and the verification cost is shown in Fig 4.c. Similarly, a client can also pre-compute the auxiliary information $(S_{1,1}, S_{1,2}, S_{2,1}, S_{2,2})$, since these values are determined only by the indexes (i, j) and a PRF. The client needs six pairing operations, six exponentiation operations in G_1 and $O(n^2)$ modular addition and multiplication operations in Z_q^* to verify the validity of the result res.

Note that our construction significantly reduces the storage and computation burdens on the data sources and the clients due to our outsourcing model. Otherwise, machines M_1 and M_2 have to store the

TABLE 1
Computation Cost for Proof Generation (seconds)

Query	The number of involved data		
	1000	2000	3000
Group-by	0.016	0.033	0.049
Inner product	0.774	1.562	2.317

$O(n^2)$ entries of the matrices, and then send matrices A and B to the client, respectively. In addition, the client requires the super-quadratic amount of work to compute the matrix product.

Cloud side. To evaluate the performance of the cloud in our scheme, we measure its computation cost to generate proofs for client requests including group-by sum query and inner product query, where the number of n increases from 1000 to 3000. The results are given in TABLE 1. The cost for generating a proof for group-by sum query is extremely lightweight. This is because it only involves inexpensive multiplication operations in Z_q^*. On the other hand, exponentiation operations dominate the proof generation cost for the inner product query. Table 1 shows that it takes about 2.317 seconds even with a large number $n = 3000$. For simplicity, with matrix product query, we consider the multiplication of two $n \times n$ matrices. The values $\pi[i][j]_1, \pi[i][j]_2, \pi[i][j]_3, res[i][j]_1, res[i][j]_2$ computed once and used later for the same query, can be amortized over all future executions. Computing $\{\pi_1, \pi_2, \pi_3, res_1, res_2\}$ needs roughly 8.52 and 214.56 seconds for $n = 100$ and $n = 500$, respectively. Therefore, the overall performance at the cloud side is totally acceptable if we consider a more powerful cloud in practice.

8 Conclusion

In this paper, we introduce a novel homomorphic verifiable tag technique, and design an efficient and publicly verifiable inner product computation scheme on the dynamic outsourced data streams under multiple keys. We also extend the inner product scheme to support matrix product. Compared with the existing works under the single-key setting, our scheme aims at the more challenging *multi-key* scenario, i.e., it allows multiple data sources with different secret keys to upload their *endless data streams* and delegate the corresponding computations to a third party server, while the traceability can still be provided on demand. Furthermore, any *keyless* client is able to *publicly* verify the validity of the returned computation result. Security analysis shows that our scheme is provable secure under the CDH assumption in the random oracle model. Experimental results demonstrate that our protocol is practically efficient in terms of both communication and computation cost.

Acknowledgment

The authors would like to thank the editor and anonymous reviewers for their valuable suggestion to improve the quality of this manuscript. This research was supported by China Postdoctoral Science Foundation funded project (Grant No. 2014M562377), and National Natural Science Foundation of China (Grant No. 61402352, No. 61272481, No. 61572460).

References

[1] Y. Zhu and D. Shasha, "Statstream: Statistical monitoring of thousands of data streams in real time," in *Proceedings of the 28th international conference on Very Large Data Bases*. VLDB Endowment, 2002, pp. 358–369.
[2] W. Sun, X. Liu, W. Lou, Y. T. Hou, and H. Li, "Catch you if you lie to me: Efficient verifiable conjunctive keyword search over large dynamic encrypted cloud data," in *Computer Communications (INFOCOM), 2015 IEEE Conference on*. IEEE, 2015, pp. 2110–2118.
[3] X. Liu, Y. Zhang, B. Wang, and J. Yan, "Mona: secure multi-owner data sharing for dynamic groups in the cloud," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 6, pp. 1182–1191, 2013.
[4] L. Wei, H. Zhu, Z. Cao, X. Dong, W. Jia, Y. Chen, and A. V. Vasilakos, "Security and privacy for storage and computation in cloud computing," *Information Sciences*, vol. 258, pp. 371–386, 2014.
[5] S. Nath and R. Venkatesan, "Publicly verifiable grouped aggregation queries on outsourced data streams," in *International Conference on Data Engineering*. IEEE, 2013, pp. 517–528.
[6] D. Catalano and D. Fiore, "Practical homomorphic macs for arithmetic circuits," in *Advances in Cryptology–EUROCRYPT*. Springer, 2013, pp. 336–352.
[7] R. Gennaro and D. Wichs, "Fully homomorphic message authenticators," in *Advances in Cryptology-ASIACRYPT*. Springer, 2013, pp. 301–320.
[8] M. Backes, D. Fiore, and R. M. Reischuk, "Verifiable delegation of computation on outsourced data," in *ACM conference on Computer and communications security*. ACM, 2013, pp. 863–874.
[9] D. Boneh and D. M. Freeman, "Homomorphic signatures for polynomial functions," in *Advances in Cryptology–EUROCRYPT*. Springer, 2011, pp. 149–168.
[10] K.-M. Chung, Y. Kalai, and S. Vadhan, "Improved delegation of computation using fully homomorphic encryption," in *Advances in Cryptology–CRYPTO*. Springer, 2010, pp. 483–501.
[11] R. Gennaro, C. Gentry, and B. Parno, "Non-interactive verifiable computing: Outsourcing computation to untrusted workers," in *Advances in Cryptology–CRYPTO*. Springer, 2010, pp. 465–482.
[12] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum, "Delegating computation: interactive proofs for muggles," in *ACM symposium on Theory of computing*. ACM, 2008, pp. 113–122.
[13] J. R. Thaler, "Practical verified computation with streaming interactive proofs," Ph.D. dissertation, Harvard University, 2013.
[14] S. Benabbas, R. Gennaro, and Y. Vahlis, "Verifiable delegation of computation over large datasets," in *Advances in Cryptology–CRYPTO*. Springer, 2011, pp. 111–131.
[15] D. Fiore and R. Gennaro, "Publicly verifiable delegation of large polynomials and matrix computations, with applications," in *ACM conference on Computer and communications security*. ACM, 2012, pp. 501–512.
[16] B. Parno, M. Raykova, and V. Vaikuntanathan, "How to delegate and verify in public: Verifiable computation from attribute-based encryption," in *Theory of Cryptography*. Springer, 2012, pp. 422–439.
[17] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," in *IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 238–252.
[18] V. Vu, S. Setty, A. J. Blumberg, and M. Walfish, "A hybrid architecture for interactive verifiable computation," in *IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 223–237.
[19] S. T. Setty, V. Vu, N. Panpalia, B. Braun, A. J. Blumberg, and M. Walfish, "Taking proof-based verified computation a few steps closer to practicality." in *USENIX Security Symposium*, 2012, pp. 253–268.

[20] S. T. Setty, R. McPherson, A. J. Blumberg, and M. Walfish, "Making argument systems for outsourced computation practical (sometimes)." in *NDSS*, 2012.

[21] K.-M. Chung, Y. T. Kalai, F.-H. Liu, and R. Raz, "Memory delegation," in *Advances in Cryptology–CRYPTO*. Springer, 2011, pp. 151–168.

[22] S. Papadopoulos, G. Cormode, A. Deligiannakis, and M. Garofalakis, "Lightweight authentication of linear algebraic queries on data streams," in *International conference on Management of data*. ACM, 2013, pp. 881–892.

[23] S. G. Choi, J. Katz, R. Kumaresan, and C. Cid, "Multi-client non-interactive verifiable computation," in *Theory of Cryptography*. Springer, 2013, pp. 499–518.

[24] S. D. Gordon, J. Katz, F.-H. Liu, E. Shi, and H.-S. Zhou, "Multi-client verifiable computation with stronger security guarantees," in *Theory of Cryptography*. Springer, 2015, pp. 144–168.

[25] W. Diffie and M. E. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[26] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin, "Dynamic authenticated index structures for outsourced databases," in *ACM international conference on Management of data*. ACM, 2006, pp. 121–132.

[27] F. Li, K. Yi, M. Hadjieleftheriou, and G. Kollios, "Proof-infused streams: Enabling authentication of sliding window queries on streams," in *International conference on Very large data bases*. VLDB Endowment, 2007, pp. 147–158.

[28] H. Pang, A. Jain, K. Ramamritham, and K.-L. Tan, "Verifying completeness of relational query results in data publishing," in *International conference on Management of data*. ACM, 2005, pp. 407–418.

[29] H. Pang and K.-L. Tan, "Authenticating query results in edge computing," in *International Conference on Data Engineering*. IEEE, 2004, pp. 560–571.

[30] S. Papadopoulos, Y. Yang, and D. Papadias, "Cads: Continuous authentication on data streams," in *International conference on Very large data bases*. VLDB Endowment, 2007, pp. 135–146.

[31] Y. Yang, D. Papadias, S. Papadopoulos, and P. Kalnis, "Authenticated join processing in outsourced databases," in *International Conference on Management of data*. ACM, 2009, pp. 5–18.

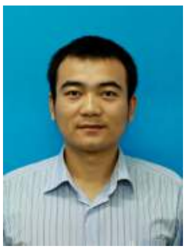
[32] C. Gentry and S. Halevi, "Implementing gentrys fully-homomorphic encryption scheme," in *Advances in Cryptology–EUROCRYPT*. Springer, 2011, pp. 129–148.

[33] J. Katz and Y. Lindell, *Introduction to modern cryptography: principles and protocols*. CRC Press, 2007.

[34] D. Boneh and M. Franklin, "Identity-based encryption from the weil pairing," in *Advances in CryptologyłCRYPTO 2001*. Springer, 2001, pp. 213–229.

[35] V. Shoup, "Lower bounds for discrete logarithms and related problems," in *Advances in CryptologyłEUROCRYPT*. Springer, 1997, pp. 256–266.

[36] A. De Caro and V. Iovino, "jpbc: Java pairing based cryptography," in *IEEE Symposium on Computers and Communications*. IEEE, 2011, pp. 850–855.



Xuefeng Liu is a postdoctor of Xidian University, China. He received his B.S. and Ph.D. degrees in information security from Xidian University, in 2007 and 2013, respectively. His research interests lie in the fields of cloud computing security and applied cryptography.

Wenhai Sun (S'14) received his BS degree in information security in 2007 and the PhD degree in cryptography in 2014 both from Xidian University, China. Since 2015 he has been a PhD candidate in the Department of Computer Science and a member of Complex Networks and Security Research Lab at Virginia Tech, USA. His research interests lie in the fields of cloud computing security, big data security, applied cryptography. He received the 8th ACM ASIACCS Distinguished Paper Award in 2013. He is a student member of the IEEE.

Hanyu Quan received the M.S. degree in cryptography and the B.S. degree in information security from Xidian University. He is currently working toward the Ph.D. degree in Xidian University. His research interests include the security and privacy issues in cloud computing and big data.

Wenjing Lou (M'03-SM'08-F'15) is a professor at Virginia Polytechnic Institute and State University. Prior to joining Virginia Tech in 2011, she was a faculty member at Worcester Polytechnic Institute from 2003 to 2011. She received her Ph.D. in Electrical and Computer Engineering at the University of Florida in 2003. Her current research interests are in cyber security, with emphases on wireless network security and data security and privacy in cloud computing. She was a recipient of the U.S. National Science Foundation CAREER award in 2008. She is a fellow of the IEEE.

Yuqing Zhang is a professor and supervisor of Ph.D. students of Graduate University of Chinese Academy of Sciences. He received his B.S. and M.S. degree in computer science from Xidian University, China, in 1987 and 1990 respectively. He received his Ph.D degree in Cryptography from Xidian University in 2000. His research interests include cryptography, wireless security and trust management.

Hui Li (M'10) received B.S. degree from Fudan University in 1990, M.S. and Ph.D. degrees from Xidian University in 1993 and 1998. In 2009, he was with Department of ECE, University of Waterloo as a visiting scholar. Since 2005, he has been a professor in the school of Telecommunications Engineering, Xidian University, China. His research interests are in the areas of cryptography, security of cloud computing,wireless network security and information theory. He served as TPC co-chair of ISPEC 2009 and IAS 2009, general co-chair of E-Forensic 2010, ProvSec 2011 and ISC 2011. He is a member of the IEEE.