Toward Secure and Scalable Computation in Internet of Things Data Applications

Xu Yuan, *Member, IEEE*, Xingliang Yuan, *Member, IEEE*, Baochun Li, *Fellow, IEEE*, and Cong Wang, *Senior Member, IEEE*

***Abstract*—The ever-growing of Internet of Things (IoT) data and the new spectrum of data applications have stimulated IoT clients to outsource their data to cloud servers or datacenters. Apart from storage service, the IoT clients also desires the servers to execute functional operations per client's request. In this paper, we aim to design the secure mechanisms that allow the IoT clients to outsource their encrypted data to geographically distributed servers while supporting homomorphic computation functions. We leverage the distributed index framework to disassemble and spread data evenly across geographically distributed servers while employing the key–value store as the underlying structure for fast data retrieval. To support computing over encrypted data, we customize Shamir's secret sharing into our mechanisms to design a tunable scheme for the adaption of different IoT application scenarios. In particular, we design three tunable protocols to achieve the effective additive homomorphic computations while approaching efficiency in terms of servers utilization, computation, and storage overhead. Even the designs focus on the additive computation, we show that it can be readily extended to other types of homomorphic computations as well as verifying the correctness of stored data. Based on the proposed protocols, we design system prototypes, deploy them in Amazon Web services, and evaluate our construction experimentally. Through experimental results, we show that our designs can achieve the efficiency in various perspectives.**

***Index Terms*—Cloud computing security, homomorphic computation, Internet of Things (IoT), key–value (KV) store.**

I. Introduction

THE application spectrum of Internet of Things (IoT) has been extensively spread to wide areas, such as healthcare, smart cities, intelligent transportation systems, and so forth [1], [2]. Accompanying with these applications is the rapid growth of the IoT data volumes, which have been explosively increasing year by year. However, IoT devices are typically small and portable with limited storage and computation capabilities. To handle the rapid growth of the data volumes and their applications, IoT devices themselves have reached a bottleneck which prevents the efficient data applications to enhance their functionalities. On the other hand, the prominent features of the cloud have attracted more IoT clients' interest to outsource their data storage and analytical services to the geographically distributed servers. These features, including high scalability and powerful computation resources, can provide much stable and efficient analytical services than a single IoT device [3]–[7]. Thus, outsourcing the IoT data and its applications into the remote cloud servers have been a promising method to extend the new IoT application ranges.

However, storing sensitive data over the remote servers has been inevitably raising concerns on data privacy and confidentiality. Before outsourcing, data should be initially encrypted with the appropriate algorithm so that data leakage can be minimized when reacting to clients' data analytical request. The general category of searchable symmetric encryption (SSE) has been proposed to enhance data privacy while preserving the efficient search privileges, such as keyword search and range search. But the design of SSE does not focus on the flexible data management on distributed servers. And also, one of the most popular general data analytics, i.e., homomorphic computation, has not been well explored in this category. To perform homomorphic computation, many designs (Yao's Garbled circuit [8], fully homomorphic encryption (FHE) [9], Paillier system [10], BGN [11]) based on the asymmetric homomorphic encryption have been proposed and applied to achieve secure computation over the encryption domain. However, the underlying algorithms in these designs are essentially based on public-key cryptographic mechanisms, which have been known with high computation complexity. Thus, these existing works cannot satisfy the efficiency requirements for computation over large volumes of data.

In this paper, we consider an IoT client wishes to store its data on distributed servers while performing the efficient homomorphic computation over these servers in a distributed manner. The design goals are toward the fast data retrieval and efficient computation with security guarantee at the distributed server sides. For distributing data, the distributed index framework (i.e., key–value (KV) store [3], [4], [12]–[14]) has been a flexible and effective solution to spread data evenly across

Manuscript received September 8, 2018; revised November 20, 2018; accepted December 17, 2018. Date of publication January 3, 2019; date of current version May 8, 2019. This work was supported in part by the Louisiana Board of Regents under Contract LEQSF(2018-21)-RD-A-24, in part by the Oceania Cyber Security Centre POC Scheme, in part by the Research Grants Council of Hong Kong under Grant CityU C1008-16G, and in part by the National Natural Science Foundation of China under Grant 61572412. *(Corresponding author: Xu Yuan.)*

X. Yuan is with the School of Computing and Informatics, University of Louisiana at Lafayette, Lafayette, LA 70503 USA (e-mail: xu.yuan@louisiana.edu).

X. Yuan is with the Faculty of Information Technology, Monash University, Clayton, VIC 3800, Australia (e-mail: xingliang.yuan@monash.edu).

B. Li is with the Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON M5S 3G4, Canada (e-mail: bli@ece.toronto.edu).

C. Wang is with the Department of Computer Science, City University of Hong Kong, Hong Kong (e-mail: congwang@cityu.edu.hk).

Digital Object Identifier 10.1109/JIOT.2018.2890728

geographically distributed servers, thus will be employed to serve as the underlying structure for data outsourcing in our design. The exemplary designs with distributed index framework have achieved the efficient search and strong security notions, but their functionalities are only limited to the simple query [13] and rich queries [14]. But, data analytic solutions, such as homomorphic computations, have not been well studied yet. The reason of these limitations is that the values are encrypted with standard encryption algorithms, i.e., AES, which does not hold the homomorphic computation properties.

The goal of this paper is to design new mechanisms to support distributed homomorphic computations by leveraging the distributed index framework and primitives cryptographic algorithms to adapt a much broader range of data applications. The Shamir's threshold secret sharing method with homomorphic computation properties will be employed as the underlying cryptographic algorithm to protect data privacy. We customize its design into our mechanism to implement homomorphic computation functions while satisfying the security and efficiency requirements. Due to the complexity of secure computations over encrypted domain, we cannot expect a single protocol to meet the satisfaction of all users, but multiple designs with the tradeoff in various perspectives seem to be a plausible approach so as to provide tunable solutions for users. Therefore, we will design multiple solutions with the tradeoff on storage overhead, communication overhead, and client's computation complexity.

Particularly, we design three protocols to achieve the tradeoff in different perspectives. We take the additive computation as a sample example to motivate our designs. Other homomorphic computations, such as subtraction and comparison, will be discussed later. Specifically, in Protocol I, for each data value (also called a secret), the client generates one share (based on Shamir's secret sharing) corresponding to each server. This enables fast data locating while computations can be performed on the respective server directly without the intercommunication among servers. All additive computation will be solely done at the server sides. Thus, Protocol I brings both less communication overhead and client side computation complexity. However, Protocol I has a hard requirement on the number of shares for each secret to be equal to the number of servers. This design will bring unnecessary storage overhead. In Protocol II, we allow client to generate an arbitrary number of shares and these shares will be randomly distributed to servers with a balanced distribution. In this protocol, since the required additive shares are randomly located on different servers, we enable two stages of additive computations. The first stage is on the servers, where the respective shares can be found and added together while the remaining shares will be sent to the client. The client will then perform the second stage computation once it receives all remaining shares. Obviously, this problem has less storage overhead and communication overhead, but brings the client side computation overhead. In Protocol III, we employ the same shares generation and distribution methods with less storage overhead. In addition, we remove the assistance of the client while all computations are solely done at the server sides in a fully distributed manner. We assume a search token will pass among all servers to find the corresponding shares for computation and only the aggregated results will be returned to the client. This will release the client side computation, but brings the communication overhead among distributed servers. In summary, the Protocol I achieves less communication overhead and client side computation complexity, but high storage overhead. The Protocol II achieves less storage and communication overhead but high client side computation complexity. The Protocol III achieves less storage overhead and client side computation complexity but higher interserver communication overhead. To validate the performance of our designs, we develop the system prototypes, deploy them on Amazon Web Services (AWS) EC2, and evaluate our constructions experimentally.

The remainder of this paper is organized as follows. In Section III, we introduce the background knowledge with primitive cryptographic concepts and algorithms that will be used in this paper. Section II discusses related work and Section IV describes our problem and system model of this paper. In Section V, we discuss different application scenarios and propose three corresponding protocols to adapt their applications. We extend our discussion into other types of homomorphic computation such as subtractive and comparison operations. Section VI presents the implementation and performance evaluation of our construction and Section VII concludes this paper.

II. Related Works

A. Encrypted Data Stores

In the literature, data stores that support querying over encrypted data draw much attention. CryptDB [15] proposed by Popa *et al.* is one of the first representative encrypted databases. It implemented a set of encryption schemes to support different SQL queries, respectively. Then Tu *et al.* [16] improved CryptDB by developing a customized query planner. After that, Pappas *et al.* [17] proposed BlindSeer to support arbitrary encrypted boolean queries. They proposed an encrypted B-tree index, where each query is encoded and evaluated via a garbled circuit-based secure two-party computation protocol. To address recent attacks (also known as inference attacks [18]) in this area, Poddar *et al.* [19] devised Arx which includes a range query protocol with the protection of data orders. Papadimitriou *et al.* [20] devised Seabed with a dedicated padding schema to hide the relations of data values. Recently, Yuan *et al.* [14] designed an encrypted KV store which can be deployed in a cluster and support parallel query processing. However, those systems consider a centralized or logically centralized server setting, which is unlike our setting. That is, all servers are individual computing and storage devices, which are not maintained by a central service provider.

B. Search Over Encrypted Data

Search as a ubiquitous function of data utilization is intensively studied in the encrypted domain. Searchable encryption [21] and order-preserving/revealing encryption [22], [23] are proposed to enable keyword search and range search, respectively. Those primitives only focus on the functions of search, and do not consider the computation after locating the matched encrypted data. Our design leverages the similar techniques of the above primitives for data search, i.e., secure one-way functions to protect the contents of search requests.

C. Homomorphic Encryption

Homomorphic encryption allows the server to compute directly over the encrypted data. In [9], the first FHE is constructed to enable arbitrary computations on ciphertexts. To improve efficiency, somewhat homomorphic encryption (SHE) is proposed to support limited numbers of computations [24]. However, FHE and SHE are still too heavy to be deployed in practice. Partially homomorphic encryption (PHE) like Paillier system [10] is widely adopted because of its efficiency, but it only supports singular operations. To further speedup, a symmetric-key-based PHE is implemented in [20] to replace Paillier for fast aggregation. However, this scheme is suitable for a static database, because the randomnesses involved in encryption/decryption need to be preset to achieve performance gains. When new data is added, affected data should be re-encrypted to preserve the efficiency.

D. Secure Multiparty Computation

There is a line of work on secure multiparty computation (SMC). As surveyed in [25], generic SMC can be classified into three directions, i.e., circuit-based, homomorphic encryption-based, and secret sharing-based schemes. Our design is related to secure sharing-based SMC schemes [26]–[28] (just to list a few). Note that existing schemes focus on the improvement the overall performance of SMC [26], applications in network analytics [27], or algorithms in high-dimensional data aggregation [28]. Unlike prior designs, our design leverages SMC to enable secure database queries. Besides, we consider several practical application scenarios and customize the protocols to serve different requirements.

III. Background Knowledge

In this section, we will give an overview of the cryptographic primitives that will be used in this paper.

A. Encrypted KV Store

We follow the construction of encrypted KV stores as proposed in [13], where the encrypted datasets can be stored as the KV pairs. We assume the client has a set of data value $\mathbf{v}$ while each data has an attribute l. The attribute and value pair (l, v) is considered as one KV pair, where attribute l is the search attribute. This KV pair should be protected when outsourcing to the remote servers. The search attribute l is kept safe by the secure pseudo-random function (PRF) which is denoted as l^*. The value v is encrypted via symmetric encryption and denoted as v^*. Then the entry of KV store is defined as

$$<l^*, v^*> = <P(K_l, l), \mathrm{Enc}(K_v, v)> \quad (1)$$

where $P(\cdot)$ is PRF function, K_l and K_v are the private keys. The consistent hashing function [29] could be employed with the input l^* to find the target server for $\langle l^*, v^*\rangle$. With consistent hashing, all documents can be stored across multiple servers with a balanced distribution.

B. Shamir's Threshold Secret Sharing

Shamir's (t, n) threshold secret sharing is a cryptographic algorithm to divide a secret value S into n shares in such a way that: 1) any t or more pieces of shares can recover S and 2) any $t-1$ or fewer shares cannot recover S.

To generate n shares, a random $t-1$ degree polynomial F will be chosen in the following form:

$$F(x) = S + a_1 x + a_2 x^2 + \cdots + a_{t-1} x^{t-1} \quad (2)$$

where $a_1, a_2, \ldots, a_{t-1}$ are distinct nonzero elements chosen independently from real number set $\mathcal{R}$ with uniform distribution. The input of the randomly uniform distributed x chosen from $\mathcal{R}$ will differentiate the shares. For example, to generate n shares for a secret S, we can randomly chosen $\{a_1, a_2, \ldots, a_{t-1}\}$ to form the $t-1$ degree polynomial function as shown in (2). A set of $\{x_1, x_2, \ldots, x_n\}$ will be randomly chosen as the entries of above polynomial function to output $F(x_1), F(x_2), \ldots, F(x_n)$. Here, $F(x_1), F(x_2), \ldots, F(x_n)$ are called the n shares respecting to the set of $\{x_1, x_2, \ldots, x_n\}$. With any t of $\{F(x_1), F(x_2), \ldots, F(x_n)\}$, we can use the Lagrange interpolation method to recover the S, i.e.,

$$S = \sum_{i=0}^{t} F(x_i) \prod_{1\le j\le t}^{j\ne i} \frac{x_j}{x_j - x_i}. \quad (3)$$

The advantages of Shamir's threshold secret sharing method can be summarized as follows.

1) Any adversary cannot recover the original secret if it can only get at most $t-1$ shares.
2) The original secret can still be recovered if at most $n-t$ shares are lost.

C. Homomorphic Computation

Homomorphic computation is a form of arithmetic calculations over encrypted information without decrypting the corresponding ciphertext. The computation results, when decrypted, should be the same as the calculation performed on the plaintext. For example, we have two ciphertexts $\mathrm{Enc}_k(m_1)$ and $\mathrm{Enc}_k(m_2)$ corresponding to plaintext m_1 and m_2, respectively. The additive homomorphic computation is shown as follows:

$$\mathrm{Enc}_k(m_1 + m_2) = \mathrm{Enc}_k(m_1) + \mathrm{Enc}_k(m_2). \quad (4)$$

When decrypted

$$\begin{aligned}\mathrm{Dec}_k(\mathrm{Enc}_k(m_1) + \mathrm{Enc}_k(m_2)) &= \mathrm{Dec}_k(\mathrm{Enc}_k(m_1 + m_2)) \\ &= m_1 + m_2. \end{aligned} \quad (5)$$

IV. Problem Statement

We consider that the IoT clients have a set of data values and applications. Due to the storage and computation limitations, IoT clients would like to outsource their data to remote servers and will leverage their powerful computation capabilities to perform the IoT data analytics. There are a set of key factors

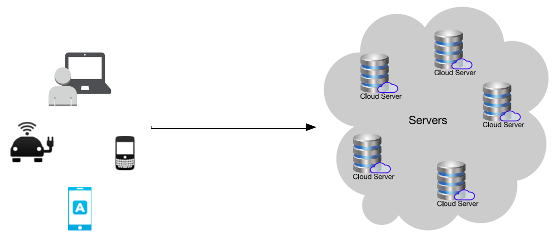


Fig. 1. System architecture.

that should be considered when outsourcing the data and its application to the remote servers, including but not limited to privacy, scalability, efficiency, overhead, and among others. The goal of this paper is to design scalable mechanisms, secure encryption schemes, and computation protocols to carry out efficient homomorphic computation for IoT data applications.

A. System Model

Our system architecture is shown as in Fig. 1. In this architecture, the IoT client has a private database consisting of a set of sensitive data values. Due to the computation and storage limitation, this client would like to outsource its database and computation to the geographically distributed servers. To better manage the database, we assume it is in the form of KV structure,[1] i.e., (key, value), where the key is a search attribute while the value is a corresponding data value. To distribute data across geographically located servers, this client will employ the distributed index framework (i.e., KV store) as the underlying structure to index sensitive data and build the local encrypted database for each target server. The local encrypted database on each server will be stored in the form of the encrypted index and value structures. With the local encrypted database, we will design various protocols to support the remote homomorphic computations.

There are three concerns at the client when outsourcing IoT data applications to the remote servers. First, this user does not wish to leak any sensitive information of the original database to remote servers or other adversaries. The confidentiality has to be preserved against the potential attackers. Namely, data has to be stored in a certain encryption form to achieve this goal. Second, the outsourced data, even in encrypted form, should be capable of supporting the analytical operations, such as additive, subtractive, sum, comparison, and so on, without leaking any sensitive information. Moreover, the outsourced data should tolerate the failure of the partial servers to maintain the completeness of the sensitive database.

To address above concerns, suitable secure encryption schemes need to be employed when outsourcing data into distributed servers, while efficient computation protocols should be designed to process user's data applications request. In our design, two cryptographic primitives, i.e., encrypted KV store and Shamir's secret sharing, will serve as the underlying structures to achieve our goal. We customize their characteristics into our design to meet the requirements on security, scalability, computation efficiency, and fault tolerance.

[1]If the data set is not in the KV structure, based on [13], we can transform it into the key and value format.

B. Threat Model

In this paper, the client's sensitive database is the target that should be protected. The client should never expose its encryption keys and encryption structure to the distributed servers or any other adversaries. We consider that the threats are from the semi-honest adversaries, who are interested in the database, but cannot delete or modify the database intentionally. Moreover, we assume the servers cannot collude with each other or be comprised by some adversaries to learn the encrypted data. However, they can monitor query and homomorphic computation protocols to learn the query attributes and the computation results. We do not consider the case where attackers can access the background information, i.e., the statistic information of the database, or the contents of query and computation results, thus the inference attack [18] and leakage-abuse attack [30] will not be considered in this paper. Moreover, we assume that the communication channels are authenticated and encrypted against eavesdropping.

V. Proposed Constructions

In this section, we describe our designs of homomorphic computation protocols in support of the IoT data applications. We take additive computation as an example to design the data outsourcing and computation protocols.

A. Overview

To outsource the sensitive data, we leverage the distributed index structure to encrypt the search attributes. The distributed index structure has the property of building local index which can help to locate data in a fast manner. To support additive homomorphic computation, there are several popular cryptographic primitives, such as Pallier Crypto, Gentry's Crypto, Shamir's secret sharing, and so on. Among these cryptographic primitives, the Shamir secret sharing is the most efficient one since it only includes additive operations rather than exponential and multiplicative (e.g., public-key encryption) as used in other systems. In our design, we customize the design of Shamir's secret sharing and distributed index framework into our protocols.

In particular, we consider three usage scenarios and design the corresponding protocols to adapt their applications. The first scenario is that the IoT client is a small portable device which has low memory and does not support large data computation. The storage service at the servers is cheap and the price charge is based on the number of servers that are running. The client can use all servers for storage but only a necessary number of machines instead of all of them while performing computation. The second scenario is that the storage service on the cloud servers are expensive and the client has the delay-sensitive applications. Thus, the client aims to minimize the data storage size on the servers, but would like to optimize the utilization of the servers with parallel computation to reduce the total computation latency. The third scenario is that the client's application is not sensitive to the latency and it does not have powerful computation capability.

With these scenarios, we design the corresponding protocols to adapt their applications. The Protocol I is a simple design,

```
Protocol I: Index Building and Data Encryption
1. Data: Private key $K$, $DB$, $x_1, x_2, \cdots, x_N$, k, N.
2. begin:
3.   client:
4.   For each $(C, V) \in DB$ do
5.      Randomly generate coefficients $a_1, a_2, \cdots, a_{(k-1)}$
6.      Generate Shares $V_1, V_2, \cdots, V_N$ for secret $V$
           corresponding to $x_1, x_2, \cdots, x_N$
7.      for each $V_i$ do
8.         for each $c = 1, \cdots, N$
9.            $label \leftarrow F(k, C||c)$
10.           send $(label, V_c)$ to the server $c$
11.        end for
12.     end for
13.  end for
```

Fig. 2. Procedure of index building and data encryption in Protocol I.

```
Protocol I: Additive Homomorphic Computation Protocol
1. Request: Comp$(k, C_1, C_2, +)$.
2. Data: Private key $k$; attributes $C_1$ and $C_2$
3. Begin:
4.   client:
5.   for each $c \leq N$
6.      $label_c^1 \leftarrow F(K, C_1||c)$
7.      $label_c^2 \leftarrow F(K, C_2||c)$
8.   send all $(label_c^1, label_c^2, +)$ to server $c$
9.   server:
10.  for server $i \in [1, N]$
11.     $V_i^1 \leftarrow Get(label_i^1)$
12.     $V_i^2 \leftarrow Get(label_i^2)$
13.     send $(V_i^1 + V_i^2)$ to client
14.  The client recovers $V_1 + V_2$ with $\{V_i^1 + V_i^2 | i = 1 \ to \ N\}$
15.  Return: $m_1 + m_2$.
```

Fig. 3. Search and computation procedure in Protocol I.

which generates one share corresponding to each server. In this design, the IoT client can quickly locate shares and aggregate the related shares together on each server, but it requires the number of shares generated for each secret to be equal of the number of servers even some of them will not be used for computation. The serious storage overhead will be caused by this protocol. In Protocol II, we allow the user to generate an arbitrary number of shares and these shares will be randomly distributed to servers with the balanced distribution. In this protocol, since the required additive shares are randomly located on different servers, two stages are enabled for additive computations. The first stage is on the server sides. Once the servers find the respected shares, they can add them together and send the remaining shares to the user. The user will then perform the second stage computation once it receives all remaining shares. Finally, we design the Protocol III, which achieves the same space efficiency as in Protocol II, but the computation protocol is fully distributed. In the computational protocol, we remove the client's assistance and let the search token to pass among servers to find the corresponding items for computation. Besides, only the computation results will be returned to the client. In the following of this section, we present the details of each proposed protocol.

B. Protocol I: Basic Design

We now describe our first protocol to adapt the first application scenario. As mentioned earlier, a distributed index framework will be utilized to index the search attributes and Shamir's secret sharing will be advocated to encrypt the respective values to protect data confidentiality and privacy. We assume the data is in the form of KV pair. For each KV pair (C, V), we randomly select a x_i value corresponding to each server i, and employ Shamir's threshold secret sharing [i.e., (2)] to generate N shares, i.e., $V_1, V_2, \ldots, V_N$, where N is the number of servers and each V_i corresponding to one x_i. In this process, we set t as the threshold for recovering the original secret. With each V_i, we use the distributed index framework to encrypt its search attributes. To distinguish each V_i from the same secret, an incremental number c_i is padded to the search attributes, i.e., c_i is incremented from 1 to N respected to each share V_i. With the search attribute, the search token will be generated by masking it with a one-way cryptographic hash function as $F(K, C||c_1), F(K, C||c_2), \ldots, F(K, C||c_N)$ corresponding to $V_1, V_2, \ldots, V_N$, respectively, where K is the private key. Each KV pair, i.e., $(F(K, C||c_i), V_i)$ for $i = 1, 2, \ldots, N$ will be distributed to its target server i. Note here, shares from different secrets that are corresponding to the same x store on the same server. This storage strategy can help each server locally perform additive homomorphic computation since they are corresponding to the same x. Fig. 2 depicts the details of data outsourcing procedure, including index building and data encryption.

1) Additive Homomorphic Computation Protocol: Based on our designed encryption protocol, we now discuss the corresponding computation protocol. To simplify our explanation, we use two values m_1 and m_2 as an example to introduce our computation protocol. We assume C_1 and C_2 are the search attributes of m_1 and m_2, respectively. To perform the additive homomorphic computation for m_1 and m_2, the client generates the search tokens $(F(K, C_1||c_i), F(K, C_2||c_i))$ corresponding to each server $1 \leq i \leq N$ by using the one-way hash function $F(\cdot)$. Each search tokens and computation request will be sent to the respective server i. Upon receiving the search token and computation request, each server i performs the search operation to find the corresponding values V_i^1 and V_i^2. Based on our outsourcing protocol, the V_i^1 and V_i^2 are corresponding to the same x on the same server. This enables each server to perform the additive computation for V_i^1 and V_i^2 (i.e., $V_i = V_i^1 + V_i^2$) and the computational results will be returned to the client. With all received additive results $V_1, V_2, \ldots, V_N$, this client can recover $m_1 + m_2$ with the interpolation method [see (3)].

In this protocol, since any t shares are enough to recover the original secret, it is not necessary to send the computation request to all servers. It is enough for a client to generate any t search token for m_1 and m_2 and send them to t corresponded servers to perform the additive computation. As seen, this design can tolerate the failure of at most $N-t$ servers since the shares on any of the remaining servers (at least t) are sufficient to recover the original data or perform the homomorphic computation. Fig. 3 shows the details of Protocol I.

```
Protocol II: Index Building and Data Encryption
1. Data: Private key $K$, $DB$, $x_1, x_2, \cdots, x_q$, k, N.
2. begin:
3.   client:
4.   for each $(C, V) \in DB$ do
5.     Randomly generate coefficient $a_1, a_2, \cdots, a_k$
6.     Generate Shares $V_1, V_2, \cdots, V_q$ for secret $V$
         corresponding to $x_1, x_2, \cdots, x_q$
7.     for $c = 1$ to $q$
8.       $label \leftarrow F(K, C||c)$
9.       $i \leftarrow route(label)$
10.      send $(label, V_c)$ to the server $i$
11.      c++;
12.    end for
```

Fig. 4. Build index and encrypt data in Protocol II.

The advantages of this protocol is that the shares from different secrets corresponding to the same x can be quickly located since they are storing on the same sever. The computational operations can be directly performed on the same server without the interserver communication, which will cause less communication overhead and client computation complexity. But, this protocol obviously brings huge storage overheard due to that the number of shares for each secret has to be the same as the number of servers. This overhead will linearly increase with the number of servers.

C. Protocol II: Client-Assisted Space-Efficient Scheme

In this protocol, we design a client-assisted space efficient scheme with the aim to remove the hard requirement of the number of shares from N to an arbitrary number q. That is, we allow the client to generate an arbitrary number of shares for each secret instead of N as shown in Protocol I. To balance the distribution, we employ the consistent hashing function at the client side to route data evenly across all servers. Note here, shares that are corresponding to the same x will be distributed across all servers. The challenge arising here is, to perform homomorphic computation, how to locate the shares corresponding to the same x. In our design, we grant the client assisting to identify the shares corresponding to the same x.

The index building and data encryption protocol is similar to Protocol I (see Fig. 2) with the differences that this protocol only generates q (can be an arbitrary number less than N but greater than 2) shares for each secret. This q number of shares will be routed to target servers based on the consistent hashing function with the input of the encrypted index. The detailed designs of the index building and data encryption procedure are shown in Fig. 4. Note here, the number of shares can be an arbitrary value, thus this protocol is flexible to reduce the storage space which achieves higher space efficiency when comparing to Protocol I as shown in Section V-B.

1) Additive Homomorphic Computation Protocol: In this protocol, since the shares are distributed across different servers, the main difficulty is to identify the shares that are corresponding to the same x value. In our design, we grant client to assist the computation procedure. We also take m_1 and m_2 as an example for ease of explanation. The client generates two groups of search tokens $S_1 = \{((F(K, C_1||c_i), \text{mask}(x_i))|1 \leq i \leq q\}$, and $S_2 = \{((F(K, C_2||c_i), \text{mask}(x_i))|1 \leq i \leq q\}$ and send them to the respective servers. For search tokens corresponding to each x_j, client will initialize a value $E_j = 0$. This value is used to aggregate the shares corresponding to the same x_j.

```
Protocol II: Additive Homomorphic Computation Protocol
1. Request: Comp$(k, C_1, C_2, +)$.
2. Data: Private key $K$; attributes $C_1$ and $C_2$
3. Return: $m_1 + m_2$
4.   client:
5.   For $c \leq N$
6.     $E_c = 0$
7.     $mask(x_i) \leftarrow F(x_c)$
8.     $label_1 = F(K, C_1||c)$
9.     $label_2 = F(K, C_2||c)$
10.    $S_1 \leftarrow S_1 \cup \{label_1, mask(x_i)\}$
11.    $S_2 \leftarrow S_1 \cup \{label_2, mask(x_i)\}$
12.  end for
13.  identify subset $S_1^i$ and $S_2^i$ for each server $i$
14.  server:
15.    at each server $i$:
16.      for each $(label, mask(x)) \in S_1^i$
17.        $R_1^i \leftarrow R_1^i \cup \{(mask(x), get(label)\}$
18.      end for
19.      for each $(label, mask(x)) \in S_2^i$
20.        $R_2^i \leftarrow R_2^i \cup \{(mask(x), get(label)\}$
21.      end for
22.    $R_i(x) \leftarrow R_1^i(x) \oplus R_2^i(x)$
23.    remove involved terms in $R_1^i$ and $R_2^i$
24.    send $R_i(x)$ to the client
25.    Send $R_1^i$ and $R_2^i$ to the client
26.  client:
27.    for each $c \leq k$
28.    $E_c = E_c + \sum_{i=1}^{N}(R_i(x_c) + R_i^1(x) \oplus R_i^2(x_c))$
29.  Client recovers $m_1 + m_2$
```

Fig. 5. Search and computation procedure in Protocol II.

Search tokens S_1 and S_2 will be sent to all servers in parallel. Each server i will find the search results R_i^1 and R_i^2 corresponding to S_1 and S_2, respectively, and also identify the search results with the same x by checking the value of mask(x). For the ease of expression, we define $R_i^1(x) \oplus R_i^2(x)$ as the set of additive results from the items in R_i^1 and R_i^2 corresponding to the same x. Each server i will perform the computation of $R_i(x) = R_i^1(x) \oplus R_i^2(x)$ and remove related items from R_i^1 and R_i^2 after that. The additive results $R_i(x)$ as well as R_i^1 and R_i^2 will be sent back to the client. After receiving the $R_i(x)$, R_i^1, and R_i^2, the client will perform the computation $E_j = E_j + \sum_{i=1}^{N}(R_i(x_j) + R_i^1(x) \oplus R_i^2(x_j))$ for R values that are corresponding to the same x_j. Till now, $E_1, \ldots, E_k$ are the aggregated results of q shares. With $E_1, E_2, \ldots, E_k$, client can recover the $m_1 + m_2$ with the interpolation method in (3). The detailed protocol design is shown in Fig. 5.

From this design, we can see this protocol removes the hard requirement of the shares number (which is equaling to the server numbers), which thus relieves the storage overhead at the servers side, but brings some level of computation complexity at the client side.

D. Protocol III: Fully Distributed Space-Efficient Scheme

In Protocol I, there is a hard requirement for the number of shares generated for each secret, i.e., the number of shares of each secret should be equal to the number of servers. In Protocol II, we allow the client to generate an arbitrary number of shares for each secret instead of the N (N is the number of servers). However, it requires the client to perform some level of computation which brings the computation burden to client.

In this protocol, we consider the scenario that a client does not have enough power to assist the computation and the distributed operation is required at the servers side but the efficient storage overhead is still required. We aim to design a fully distributed computation protocol to complete the computation procedure. This protocol allows all computations to be done at the servers side in a distributed manner. The challenge here is similar as that in the Protocol II. That is, how to locate the shares corresponding to the same x, since they may be stored across different servers in order to achieve the balanced distribution. To address this challenge, we assume search tokens will be passed among all servers following a token ring to find the items corresponding to the same x for addition. The index building, data encryption, and outsourcing stages are the same as in Protocol II as shown in Fig. 4.

1) Additive Homomorphic Computation Protocol: To perform the computation, our protocol has to identify two search tokens corresponding to the same x. The client first sends x along with search tokens to each server so that it can check whether two searched results are corresponding to the same x. To protect the privacy of x_i, we use one-way cryptographic hash function to mask x_i, i.e., $\text{mask}(x_i)$. Similarly as in the Protocol II, client also generates two groups of search tokens $S_1 = \{((F(k, C_1||c_i), \text{mask}(x_i))|1 \le i \le q\}$ and $S_2 = \{((F(k, C_2||c_i), \text{mask}(x_i))|1 \le i \le q\}$ for secrets m_1 and m_2, respectively. The client will divide S_1 and S_2 into the subset of groups S_1^i and S_2^i, respectively, for $1 \le i \le N$, based on the servers that search token will be routed to. After receiving a subgroup of search token, each server i performs parallel search operations to find the set of shares, assuming R_i^1 and R_i^2. Each server i will perform $R_i^1(x) \oplus R_i^2(x)$ to find the additive results for the items in $R_i^1(x)$ and $R_i^2(x)$ corresponding to the same x, and also remove all relevant items from $R_i^1(x)$ and $R_i^2(x)$.

To perform the computation for remaining items in R_i^1 and R_i^2 that do not contain items corresponding to the same x, we assume all servers form a token ring and a token will be generated at one server [31]. Such a token will be passed from this server to others one by one along the token ring. That is, the first server sends R_1^1 and R_1^2 to the second server. The second server (assume server 2) will perform the computation on $R_1^1(x) \oplus R_2^2(x)$ and $R_2^1(x) \oplus R_1^2(x)$, send results to the client, and remove all relevant items from $R_1^1(x)$, $R_2^1(x)$, $R_1^2(x)$, and $R_2^2(x)$. Then, we let $R_2^1(x) := R_1^1(x) \cup R_2^1(x)$ and $R_2^2(x) := R_1^2(x) \cup R_2^2(x)$. After that, the second server sends $R_2^1(x)$ and $R_2^2(x)$ to the next server along the token ring and repeats the same process till the last server. At each server, the following checking proceed will be executed: $\text{mask}(x^{i-1}) - \text{mask}(x^i) = 0$ or not, assuming one item in R_{i-1} has $\text{mask}(x^{i-1})$ and one item in R_i has $\text{mask}(x^i)$, where x^i represents the x in R^i. If it is zero, the server adds the respected shares together, sends it back to the client, and removes the respective items in both R_{i-1} and R_i. Otherwise, these two shares do not correspond to the same x. This procedure terminates if R_{i-1} is an empty set; otherwise, we let $R_i := R_{i-1} \cap R_i$, and send it to the next server. Obviously, this protocol needs less storage overhead and low client computation complexity, but requires the intercommunication among servers, which thus bring high communication overhead.

The designs in Protocols I, II, and III are only for the additive homomorphic computation of two values, but they can be easily applied to more than two values. We omit their discussions here to conserve space.

E. Discussion

1) Security: The security in our proposed protocols includes two parts: a) search security and b) data security. We assume distributed servers are semi-honest adversaries and the designed protocols should be against potential attacks from them. For search security, since we use the one-way hash function to mask the search keys, there do not exist probabilistic polynomial-time distinguishers to derive the search key. Thus, search tokens are strongly protected to guarantee the search security.

For data security, we use Shamir's secret sharing to encrypt data, thus the data security strength depends on the number of shares that can be learned by a server. In Protocols I and II, each server can only obtain one share of each data which will not leak any sensitive information of the original secret. Thus, both Protocols I and II can protect the data security well. In Protocol III, the aggregation of shares can be passed to next sever till the last server. This will bring potential risk where the last server may get all t aggregated shares and reach the required threshold to recover additive result. Compared to Protocols I and II, Protocol III will leak more data information. However, even with t aggregated shares, the servers still cannot recover the additive results since the x values are kept as the secret keys at the client. To avoid this leakage and be robust to the case of occasionally leaking the x values, we can mask the original secret. That is, instead of storing shares of the original secret m_1, we will mask it with $F(D)$, where D is a constant private value and $F(\cdot)$ is a one-way hash function. Thus, the stored share at the server side will be $m_1 + F(D)$. This $F(D)$ will be held at the client as a private key and will never be exposed to others. Even if in the worse case, the last server can recover $m_1 + m_2 + F(D)$, but it still cannot remove $F(D)$ to learn $m_1 + m_2$. But for the client, once it receives shares, it can first recover $m_1 + m_2 + F(D)$ and then cancel $F(D)$ to get the $m_1 + m_2$.

2) Extensions: The above design can be easily extended to comparison or subtraction operations for any two values. To support the subtractive operation in above three protocols, servers can perform subtraction over the shares corresponding to the same x after receiving search tokens. A minor change is needed in Protocols I, II, and III by replacing all additive into subtraction operations. The comparison operation can follow the same procedure as in the subtraction protocol. That is, after we recover the results from subtraction protocol, we can compare it with zero to check which one is larger.

3) Verification: Verification [32] is important to check the integrity of shares that are stored on each server to guarantee the correctness of original data. After data is outsourced to remote servers, we should periodically verify the integrity of the original data. The Paul Feldman's verification scheme [33] is a plausible solution and can be embedded into our design to support verification. In the data encryption stage, we choose a generator g belonging to a cyclic group G as a private system parameter. In order to verify the correctness of the stored shares on one server, client will generate additional items stored on servers for verification.

Accompanying with index building and data encryption as shown in Section V-C, the client also generates sequences of verifiable values, $f(a_i)$, corresponding to each coefficient in the polynomial function of Shamir' secret sharing [i.e., (2)], i.e., $f(a_i) = g^{a_i}$. All these verifiable values will be stored on servers accompanying with each key, i.e., $(F(K, C_1||c_i), \{f(a_i)|i = 1, \dots, q\}, V_i)$. The private parameter g will be stored at client for verification. To verify the integrity of one share corresponding to one x, client can perform the search to find the $\{f(a_i)|i = 1, \dots, q\}$ and V_i. Then it compares g^{V_i} with $f(a_0) * f(a_1)^x * f(a_2)^{x^2} \cdots f(a_q)^{x^t}$ to check the equality. The client can perform the following computation: $F(x) = f(a_0) * f(a_1)^x * f(a_2)^{x^2} \cdots f(a_q)^{x^t} = \prod_{j=0}^{t} g((a_j)x^j) = g^{\sum_{j=0}^{t} a_j x^j}$. It is easy to see that if the stored share is correct, then $g^{V_i} = F(x)$, since we have $V_i = \sum_{j=0}^{t} a_j x^j$ from (2).

VI. Experimental Evaluation

A. Implementation Environment

In this section, we conduct experiments to evaluate the performance of our computation protocols as proposed in Section V. Especially, we will focus on data storage and additive computational time costs under Protocols I, II, and III. We design system prototypes for the proposed protocols and implement them in AWS for performance evaluation. In our experiments, one AWS t2-small and a cluster of AWS m4-xlarge instances are created where the t2-small instance serves as the client and all other m4-xlarge instances serve as the distributed servers. For both t2-small and m4-xlarge instances, the Ubuntu Server 16.04 LTS is installed. The configurations of t2-small instance includes 1vCPUs (2.4 GHz, Intel Xeon E5-2676v3), 2 GB RAM memory and 8 GB SSD while the configurations of m4-xlarge instances include 4 vCPUs (2.4 GHz, Intel Xeon E5-2676v3), 16 GB RAM memory and 8 GB SSD. The bandwidth between any two instances is 1 Gb/s. To support both the remote data retrieval and computation between any two instances, the software framework Apache Thrift (v0.9.3) is employed to implement the remote procedure call. OpenSSL (v1.0.2a) is used for the cryptographic build blocks, in which HMAC-SHA2 is called to implement the one-way hash function. Redis 3.2.0 is installed on each instance to store the data (keys and values). Based on this experimental environment, we will conduct various experiments to show the performance of proposed protocols by varying share amounts, threshold value, dataset size, and server numbers.

TABLE I
Computation Time of Shamir's Method and Paillar Cryptosystem With Various Data Sizes

Data sizes	Shamir's Method	Paillier Crptosystem
1×10^5	0.00269s	0.240397s
1×10^6	0.026009s	2.3770068s
1×10^7	0.261838s	23.735413s
1×10^8	2.5949s	237.095208s

B. Comparison of Shamir's Method and Paillier Cryptosystem

Before conducting our experiments, we show the comparison of the computation cost between Shamir's secret sharing method and Paillier cryptosystem to provide the powerful argument of advocating Shamir's method in term of computation efficiency in our protocols. We implement both Shamir's method and Paillier cryptosystem in a single server and conduct experiments of additive homomorphic computation with various data sizes, i.e., 1×10^4, 1×10^5, 1×10^6, and 1×10^7 data records. Table I shows the computation time (including both the homomorphic computation time and decryption time) of Shamir's method and Paillier cryptosystem. The first column represents the number of data records which varies from 1×10^5 to 1×10^7. The second and third columns represent the computation time under the Shamir's method and Paillier cryptosystem, respectively. From this table, we conclude that Shamir's method has much higher computation efficiency than Paillier cryptosystem. This demonstrates the advantages of using Shamir's method in our protocols.

C. Comparison of Storage Overhead

We now compare the storage overhead caused by our Protocols I, II, and III. One instance is launched as the client who randomly generates a set of data including 1×10^5 data records, and another eight instances are launched as the distributed servers which store the encrypted data and perform the necessary computation per user's request. Our proposed protocols are implemented and deployed on these instances. We set the Shamir's threshold as 3. With the input of 1×10^5 data records, the generated shares will be distributed to respective servers. Since the encryption protocol and storage methods are same as in Protocols II and III, here we only compare the storage overhead between Protocols I and II.

In Protocol I, since this protocol has a hard requirement to have the number of shares be equal to the number of servers, the number of shares for each data is 8. The storage overhead is 8.61 Megabytes on each server. In Protocol II, we monitor the storage overhead by varying the number of shares. Table II shows the storage sizes on each server with the shares amounts varying from 3 to 8. By comparing Protocol I (storage size is 8.61 MB), we can see the storage sizes in Protocol II can be much smaller than that in Protocol II. In addition, it also has a great advantage of adjusting share amounts to reduce storage size. For example, when the share number is set to 3, the storage overhead is only in the range of [3.32M, 3.36M]

TABLE II
STORAGE OVERHEAD ON EACH SERVER BY VARYING THE NUMBER OF SHARES UNDER PROTOCOL II. "S_N" REPRESENTS "SERVER N"

Shares	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8
3	3.35 M	3.35 M	3.32 M	3.36 M	3.32 M	3.33 M	3.36 M	3.36 M
4	4.31 M	4.31 M	4.28 M	4.31 M	4.24 M	4.29 M	4.32 M	4.31 M
5	5.26 M	5.25 M	5.23 M	5.26 M	5.21 M	5.24 M	5.27 M	5.27 M
6	6.70 M	6.69 M	6.69 M	6.71 M	6.70 M	6.74 M	6.74 M	6.73 M
7	7.65 M	7.65 M	7.64 M	7.67 M	7.63 M	7.66 M	7.69 M	7.67 M
8	8.66 M	8.60 M	8.60 M	8.62 M	8.59 M	8.61 M	8.63 M	8.63 M

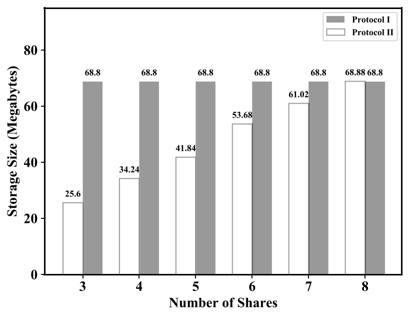


Fig. 6. Total storage overhead over all eight servers by varying the number of shares under Protocols I and II.

on each server, which can save more than 60% storage space when comparing to Protocol I.

To show that our results are consistent with various data volumes, we randomly generate 100 group of data, and each group includes 1×10^5 data records. We calculate the total storage overhead over all servers under Protocols I and II. Fig. 6 shows the total storage overhead among all servers with various amounts of shares. The white bars and black bars represent the total storage overhead on all eight servers under Protocols I and II, respectively. From this figure, we conclude that Protocol II is flexible to adjust storage overhead by setting appropriate share amounts.

D. Comparison of Computation latency

Now we compare the computation latency among Protocols I, II, and III by varying the threshold values, data sizes, and server amounts, respectively. Here, the computation latency consists of the search time, additive homomorphic computation time, data decryption time, and network delay.

1) Varying the Threshold Value: We first consider the impact of the threshold on computation latency in each protocol. One client and eight servers are launched in this experiments. The client randomly generates 100 groups of data with each one includes 1×10^5 data records.

Fig. 7 shows the average time cost (i.e., computation latency) over 100 groups of 1×10^5 data records by varying the threshold from 2 to 8 under Protocols I, II, and III. In this figure, the dashed line represents the computation latency under the Protocol I. As shown, the time cost is 3.9 s in this protocol and this value does not change with the varieties of threshold values. The reason is that the number of data volumes on each server are same which thus leads to the same search and computation time cost on each server. Since search and computation operations among all servers are parallel, even we

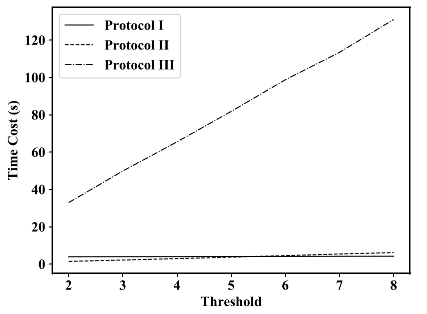


Fig. 7. Comparison of the computation latency among Protocols I, II, and III by varying the number of shares.

increase the threshold (equal to the number of servers to perform additive computation), the time cost is still determined by a single server, which will not impact the total time cost.

The solid line represents the additive homomorphic computation latency in Protocol II. The computation latency increases from 1.48 s to 6.22 s when varying the threshold value from 2 to 8. The reason is that Protocol II can always utilize computation resources of all distributed servers and perform parallel operations. Thus, the computation load on each server in Protocol II is lower than that in Protocol I, which leads to smaller computation latency. This demonstrates the importance of appropriate design of homomorphic computation protocol over distributed servers, which has the significant impact on the computation latency.

The dotted line shows the computation latency from Protocol III, which increases from 33.04 s to 131.36 s when varying the threshold from 2 to 8. We can see the time cost in this protocol is much higher than that from both Protocols I and II. The reason is that search and computation operations in Protocol III sequence among all servers while paralleling in Protocols I and II. Thus, the time cost in Protocol III cannot compete with Protocols I and II due to that the interserver communications disable the parallel operations.

2) Varying the Data Size: We now compare the computation latency in Protocols I, II, and III by varying the data sizes. In this experiment, we set the threshold to 3 while the data records are increasing from 1×10^5 to 1×10^6. For each data size, we randomly generate 100 groups of data to calculate the average computation latency.

Fig. 8(a)–(c) shows the average computation latency in Protocols I, II, and III, respectively. From these figures, we can see the computation time increases linearly with the data sizes of each protocol.

3) Varying the Number of Servers: We vary the server amounts from 3 to 8 to show their impacts on the computation latency. The data size and threshold are fixed to 1×10^5 and 3, respectively. Fig. 9 shows the computation latency under three protocols, where the dashed line, solid line, and dotted line represent the Protocols I, II, and III respectively. From this figure, we can see that the increasing of server amounts has no impact on the computational latency in Protocol I. The reason is that the computation latency is determined by the data size and the computational capability of three servers. This is because Protocol I can only utilize

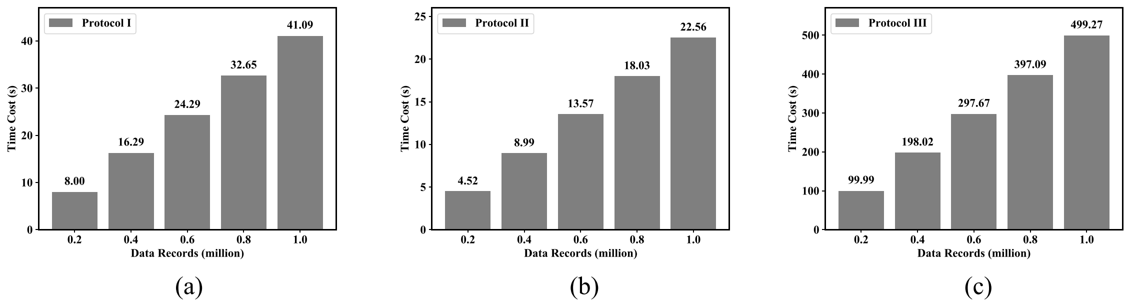


Fig. 8. Computation latency with various data records. (a) Protocol I. (b) Protocol II. (c) Protocol III.

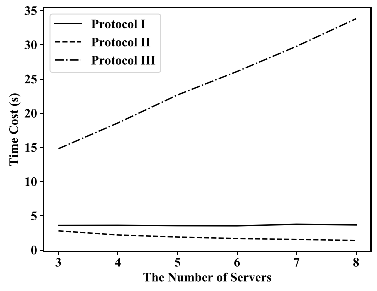


Fig. 9. Comparison of time cost with various number of servers.

three servers which is the same as the threshold value. The computation latency in Protocol II decreases with the increment of server amounts from 3 to 8, since the computation load will decrease on each single server. Thus, the computation time which is determined by a single server due to the parallel computation will decrease. On the other hand, the computation latency from Protocol III increases with the number of servers due to the sequence operation among all servers. With the increase of server amounts, this protocol needs to traverse more servers where on each single server, it needs to search the entire database to find the respective data stored on this server and perform the necessary computation. Thus, comparing Protocols I and II, Protocol III causes higher delay.

VII. Conclusion

In this paper, we designed secure homomorphic computation protocol to enable IoT clients to outsource their data and computation services to the geographically distributed servers. We employed the distributed index framework and Shamir's secret sharing to design three protocols to adapt various application scenarios while achieving efficiency in different perspectives, i.e., server utilization, computation efficiency, and storage overhead. Specifically, the Protocol I is simple and easy to be implemented. It achieves less communication overhead and client side computation complexity, but high storage overhead. The Protocol II achieves less storage and communication overhead but high client side computation complexity. The Protocol III achieves less storage overhead and client side computation complexity but high interserver communication overhead. These three protocols are formed into a tunable mechanism to adapt different IoT application scenarios. All these designs can satisfy the security guarantee and also can be readily extended to other types of homomorphic computations. The proposed protocols were implemented on Amazon EC2 to evaluate their performance experimentally.

References

[1] *State of the Market: Internet of Things 2016*, Verizon, New York, NY, USA, 2016. [Online]. Available: https://www.verizon.com/about/sites/default/files/state-of-the-internet-of-things-market-report-2016.pdf

[2] Q. Zou, L. Ni, Q. Wang, Q. Li, and S. Wang, "Robust gait recognition by integrating inertial and RGBD sensors," *IEEE Trans. Cybern.*, vol. 48, no. 4, pp. 1136–1150, Apr. 2018.

[3] R. Escriva, B. Wong, and E. G. Sirer, "HyperDex: A distributed, searchable key-value store," in *Proc. ACM Conf. Appl. Technol. Archit. Protocols Comput. Commun. (SIGCOMM)*, Helsinki, Finland, 2012, pp. 25–36.

[4] J. Ousterhout *et al.*, "The RAMCloud storage system," *ACM Trans. Comput. Syst. (TOCS)*, vol. 33, no. 3, 2015, Art. no. 7.

[5] Q. Wang *et al.*, "Privacy-preserving collaborative model learning: The case of word vector training," *IEEE Trans. Knowl. Data Eng.*, vol. 30, no. 12, pp. 2381–2393, Dec. 2018.

[6] Q. Wang *et al.*, "Real-time and spatio-temporal crowd-sourced social network data publishing with differential privacy," *IEEE Trans. Depend. Secure Comput.*, vol. 15, no. 4, pp. 591–606, Jul./Aug. 2018.

[7] M. Du, Q. Wang, M. He, and J. Weng, "Privacy-preserving indexing and query processing for secure dynamic cloud storage," *IEEE Trans. Inf. Forensics Security*, vol. 13, no. 9, pp. 2320–2332, Sep. 2018.

[8] A. C.-C. Yao, "How to generate and exchange secrets," in *Proc. IEEE Annu. Symp. Found. Comput. Sci.*, Toronto, ON, Canada, 1986, pp. 162–167.

[9] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proc. ACM Symp. Theory Comput. (STOC)*, Bethesda, MD, USA, 2009, pp. 169–178.

[10] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Proc. Int. Conf. Theory Appl. Cryptograph. Techn. (EUROCRYPT)*, Prague, Czechia, 1999, pp. 223–238.

[11] D. Boneh, E.-J. Goh, and K. Nissim, "Evaluating 2-DNF formulas on ciphertexts," in *Proc. Theory Cryptography*, Cambridge, MA, USA, 2005, pp. 325–341.

[12] (2015). *A Rock-Solid, High Performance Database That Provides NOSQL and SQL Access*. [Online]. Available: https://foundationdb.com/

[13] X. Yuan, X. Wang, C. Wang, C. Qian, and J. Lin, "Building an encrypted, distributed, and searchable key-value store," in *Proc. ACM Asia Conf. Comput. Commun. Security (ASIACCS)*, Xi'an, China, 2016, pp. 547–558.

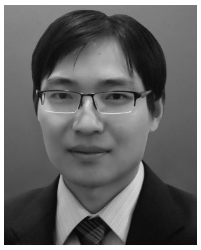
[14] X. Yuan *et al.*, "EncKV: An encrypted key-value store with rich queries," in *Proc. ACM Asia Conf. Comput. Commun. Security (ASIACCS)*, Abu Dhabi, UAE, 2017, pp. 423–435.

[15] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: Protecting confidentiality with encrypted query processing," in *Proc. ACM Symp. Oper. Syst. Principles (SOSP)*, Cascais, Portugal, 2011, pp. 85–100.

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

[17] V. Pappas *et al.*, "Blind seer: A scalable private DBMS," in *Proc. IEEE Symp. Security Privacy (S&P)*, San Jose, CA, USA, 2014, pp. 359–374.

[18] M. Naveed, S. Kamara, and C. V. Wright, "Inference attacks on property-preserving encrypted databases," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security (CCS)*, Denver, CO, USA, 2015, pp. 644–655.
[19] R. Poddar, T. Boelter, and R. A. Popa, "Arx: A strongly encrypted database system," Cryptol. ePrint Archive, Rep. 2016/591, 2016. [Online]. Available: http://eprint.iacr.org/2016/591
[20] A. Papadimitriou *et al.*, "Big data analytics over encrypted datasets with seabed," in *Proc. USENIX Conf. Oper. Syst. Design Implement. (OSDI)*, 2016, pp. 587–602.
[21] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: Improved definitions and efficient constructions," in *Proc. ACM Conf. Comput. Commun. Security (CCS)*, 2006, pp. 79–88.
[22] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill, "Order-preserving symmetric encryption," in *Proc. Annu. Int. Conf. Theory Appl. Cryptograph. Techn. (EUROCRYPT)*, 2009, pp. 224–241.
[23] K. Lewi and D. J. Wu, "Order-revealing encryption: New constructions, applications, and lower bounds," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security (CCS)*, Vienna, Austria, 2016, pp. 1167–1178.
[24] M. Naehrig, K. Lauter, and V. Vaikuntanathan, "Can homomorphic encryption be practical?" in *Proc. ACM Workshop Cloud Comput. Security*, Chicago, IL, USA, 2011, pp. 113–124.
[25] Y. Lindell and B. Pinkas, "Secure multiparty computation for privacy-preserving data mining," *J. Privacy Confidentiality*, vol. 1, no. 1, pp. 59–98, 2009.
[26] D. Bogdanov, S. Laur, and J. Willemson, "Sharemind: A framework for fast privacy-preserving computations," in *Proc. Eur. Symp. Res. Comput. Security (ESORICS)*, 2008, pp. 192–206.
[27] M. Burkhart, M. Strasser, D. Many, and X. Dimitropoulos, "SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics," in *Proc. USENIX Conf. Security (USENIX)*, Washington, DC, USA, 2010, p. 15.
[28] K. Bonawitz *et al.*, "Practical secure aggregation for privacy-preserving machine learning," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security (CCS)*, Dallas, TX, USA, 2017, pp. 1175–1191.
[29] D. Karger *et al.*, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web," in *Proc. Annu. ACM Symp. Theory Comput. (STOC)*, El Paso, TX, USA, 1997, pp. 654–663.
[30] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-abuse attacks against searchable encryption," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security (CCS)*, Denver, CO, USA, 2015, pp. 668–679.
[31] N. A. Lynch, *Distributed Algorithms*. San Francisco, CA, USA: Elsevier, 1996.
[32] J. Zhu *et al.*, "Enabling generic, verifiable, and secure data search in cloud services," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 8, pp. 1721–1735, Aug. 2018.
[33] P. Feldman, "A practical scheme for non-interactive verifiable secret sharing," in *Proc. Annu. Symp. Found. Comput. Sci.*, Los Angeles, CA, USA, 1987, pp. 427–438.

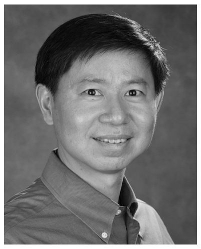


Xu Yuan (S'13–M'16) received the B.S. degree from the Department of Information Security, Nankai University, Tianjin, China, in 2009, and the Ph.D. degree from the Bradley Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University, Blacksburg, VA, USA, in 2016.

From 2016 to 2017, he was a Post-Doctoral Fellow of electrical and computer engineering with the University of Toronto, Toronto, ON, Canada. He is currently an Assistant Professor with the School of Computing and Informatics, University of Louisiana at Lafayette, Lafayette, LA, USA. His current research interests include cloud computing security, algorithm design and optimization for spectrum sharing, coexistence, and cognitive radio networks.

Xingliang Yuan (S'15–M'18) received the B.S. degree in electrical engineering from the Nanjing University of Posts and Telecommunications, Nanjing, China, in 2008, the M.S. degree in electrical engineering from the Illinois Institute of Technology, Chicago, IL, USA, in 2009, and the Ph.D. degree in computer science from the City University of Hong Kong, Hong Kong, in 2016.

He is currently a Lecturer with the Faculty of Information Technology, Monash University, Clayton, VIC, Australia. His current research interests include cloud security, privacy-aware computing, and secure networked systems.



Baochun Li (F'15) received the B.E. degree from the Department of Computer Science and Technology, Tsinghua University, Beijing, China, in 1995, and the M.S. and Ph.D. degrees from the Department of Computer Science, University of Illinois at Urbana–Champaign, Urbana, IL, USA, in 1997 and 2000, respectively.

Since 2000, he has been with the Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON, Canada, where he is currently a Professor. He holds the Nortel Networks Junior Chair of Network Architecture and Services from 2003 to 2005 and has been the Bell Canada Endowed Chair of Computer Engineering since 2005. His current research interests include cloud computing, large-scale data processing, computer networking, and distributed systems.

Dr. Li was a recipient of the IEEE Communications Society Leonard G. Abraham Award in the field of communications systems in 2000, the Multimedia Communications Best Paper Award from the IEEE Communications Society in 2009, and the University of Toronto McLean Award. He is a member of the ACM.

Cong Wang (SM'17) received the B.E. degree in electronic information engineering and the M.E. degree in communication and information system from Wuhan University, Wuhan, China, and the Ph.D. degree in electrical and computer engineering from the Illinois Institute of Technology, Chicago, IL, USA.

He is currently an Associate Professor with the Department of Computer Science, City University of Hong Kong, Hong Kong. His research has been supported by multiple government research fund agencies, including the National Natural Science Foundation of China, the Hong Kong Research Grants Council, and the Hong Kong Innovation and Technology Commission. His current research interests include data and computation outsourcing security in the context of cloud computing, network security in emerging Internet architecture, multimedia security and its applications, and privacy-enhancing technologies in the context of big data and Internet of Things.

Dr. Wang was a recipient of the President's Award from the City University of Hong Kong in 2016. He was a co-recipient of the Best Student Paper Award of CHINACOM 2009, IEEE MSN 2015, and IEEE ICDCS 2017. He has been serving as the TPC Co-Chair for a number of IEEE conferences/workshops. He is a member of the ACM.