Fog-Based Computing and Storage Offloading for Data Synchronization in IoT

Tian Wang, Jiyuan Zhou, Anfeng Liu, Md Zakirul Alam Bhuiyan, *Senior Member, IEEE*, Guojun Wang, and Weijia Jia, *Senior Member, IEEE*

***Abstract*—With the development of Internet of Things (IoT) technologies, increasingly many devices are connected, and large amounts of data are produced. By offloading the computing-intensive tasks to the edge devices, cloud-based storage technology has become the mainstream. However, if the end IoT devices send all of their data to the cloud, then data privacy becomes a great issue. In this paper, we propose a new architecture for data synchronization based on fog computing. By offloading part of computing and storage work to the fog servers, the data privacy can be guaranteed. Moreover, to decrease the communication cost and reduce the latency, we design a differential synchronization algorithm. Furthermore, we extend the method by introducing Reed–Solomon code for security consideration. We prove that our architecture and algorithm really have better performance than traditional cloud-based solutions in terms of both efficiency and security through a series of experiments.**

***Index Terms*—Cloud computing, cloud storage, computing offloading, differential synchronization, fog computing.**

I. Introduction

With the development of Internet of Things (IoT) technology, increasingly large amounts of electronic data are produced by numerous edge devices everyday [1]. Thus, there is an ever-increasing requirement for the collection and storage of these data [2]. Edge devices cannot satisfy this requirement because of their limited abilities. Fortunately, with the rise of cloud computing, the cloud storage technology has emerged in time. For the large storage capacity and computing ability of cloud computing, it can offer IoT devices with offloading services. Cloud-based synchronization is one of the core services in the cloud field. IoT devices increasingly prefer to synchronize all of their data to the cloud.

There are two main challenges in this schema: 1) security issues and 2) efficiency issues. On the one hand, if the cloud service provider owns the entire data, the data security cannot be ensured. Security issues have always been the primary focus in academic industrial sectors and include three aspects: 1) data privacy; 2) data integrity; and 3) data availability. There are many types of security threats, such as data loss, leakage of data, malicious user handling, wrong usage of cloud computing and its services, and hijacking of sessions while accessing data. Data security is always the most concerned part of users [3], [4]. On the other hand, the communication cost and latency between the cloud server and edge devices are intolerable in some delay-sensitive applications. For example, there are many synchronization tools, such as Microsofts ActiveSync and Botkinds AllwaySync. Their common disadvantage is that they always transfer the entire file even when a small change occurs. Obviously, this type of synchronization results in redundant communication and latency when the users frequently modify the data. In summary, the traditional data synchronization between IoT devices and cloud has the following shortcomings: first, the owner of IoT devices loses the operation abilities for private data, and second, frequent modifications generate a large amount of redundant data.

To solve these problems in traditional synchronization, we introduce fog computing, which is also called edge computing, as a middle computing layer. Fog computing is an extended computing model based on cloud computing; it is composed of many fog nodes with certain storage capacity and processing capability [5], [6]. It has been used in many burgeoning fields, such as sensor-cloud [7]. In this paper, we propose an architecture with three layers: 1) the edge layer; 2) fog layer; and 3) cloud layer. The fog server is set between the cloud and the edge devices, which is similar to a cache server. By offloading part of the computing and storage work to the fog server, the privacy of the data can be ensured. Furthermore, to reduce the communication cost and delay, we design a differential synchronization based on fog [8].

As the name implies, differential synchronizations check each update and only transfer the part of the file that has been changed. Remote Sync (RSYNC) is a traditional differential synchronization algorithm in Linux system operation [9]. By

Manuscript received May 15, 2018; revised July 27, 2018; accepted October 6, 2018. Date of publication October 15, 2018; date of current version June 19, 2019. This work was supported in part by grants from the National Natural Science Foundation of China (NSFC) under Grant No. 61872154, No. 61772148, and No. 61672441 and the Social Science Foundation of Fujian Province of China (No. FJ2018B038), the Natural Science Foundation of Fujian Province of China (No. 2018J01092), and the Fujian Provincial Outstanding Youth Scientific Research Personnel Training Program, and by FDCT/0007/2018/A1, FDCT/0007/2018/A1, DCT-MoST Joint-project No. (025/2015/AMJ), University of Macau Grant Nos: MYRG2018-00237-RTO, CPG2018-00032-FST and SRG2018-00111-FST of SAR Macau, China; Chinese National Research Fund (NSFC) Key Project No. 61532013, No. 61872239, and National China 973 Project No. 2015CB352401. *(Corresponding author: Tian Wang.)*

T. Wang and J. Zhou are with the College of Computer Science and Technology, Huaqiao University, Xiamen 361021, China (e-mail: cs_tianwang@163.com).

A. Liu is with the School of Information Science and Engineering, Central South University, Changsha 410083, China.

M. Z. A. Bhuiyan is with the Department of Computer and Information Sciences, Fordham University, New York, NY 10458 USA.

G. Wang is with the School of Computer Science and Educational Software, Guangzhou University, Guangzhou 510006, China.

W. Jia is with the Data Science Center, University of Macau, Macau SAR, China.

Digital Object Identifier 10.1109/JIOT.2018.2875915

introducing the fog, we can store the differential data and the corresponding information in the fog server. Thus, not every instance of synchronization sends a request to the cloud. Some trivial changes will be recorded in the fog until they exceed a particular threshold. When they reach the threshold, there is a total synchronization request from the fog server. In the mean time, the fog server will clear itself. The extra transmission between the fog server and the local machine is based on the wireless local area network (WLAN). The transmission rate of the WLAN is far higher than that of wide area network (WAN) [10]. In other words, the extra overhead is notably small. With the fog-based differential synchronization, our architecture has some benefits.

1) We can store a part of the data in the fog server for security.
2) The fog server can offload a certain amount of computation and storage that previously belongs to the cloud and user's devices.
3) The communication overload is minimized.

The main contributions of this paper can be summarized as follows.

1) We propose a new architecture for computation and storage offloading based on fog computing. This architecture can alleviate the burden of both end users and the cloud.
2) Fog-based synchronization algorithms are designed to minimize the communication cost and reduce the latency from the end users to the cloud.
3) We propose an advanced scheme with security consideration. Reed–Solomon code is introduced to protect the privacy of users.

The remainder of this paper is organized as follows. Section II reviews related research work, Section III elaborates the architecture, which we propose based on fog computing, in detail. Section IV introduces the algorithms in our architecture, Section V evaluates our architecture with several experiments, and Section VI concludes this paper.

II. Related Work

In this section, we will introduce some related research on data offloading. We will analyze and conclude them at the end of this section.

In the IoT field, edge devices cannot dispose the computing-complex task. They often offload these tasks to more powerful computing entities, e.g., cloud computing. Many studies focus on computing offloading in cloud computing in recent years [11], [12]. Chen *et al.* [13] proposed computation offload to clouds using aspect-oriented programming (COCA). COCA is a programming framework that enables smart phone application developers to easily offload part of the computation to the servers in the cloud. COCA works at the source level. By harnessing the power of aspect-oriented programming, COCA inserts the appropriate offloading code into the source code of the target application based on the result of static and dynamic profiling. However, in data synchronization, there are some differences. Data synchronization is the most common application in the synchronization field. Ramsey and Csirmaz [14] supposed that there are multiple replicas of a file system, e.g., one on a server, one on a computer at home, and one on a laptop. If one makes different changes at different replicas, the replicas no longer contain the same information. A file synchronizer makes them consistent again while preserving the changes. However, there are many challenges in the file synchronization system, such as the limit of low bandwidth. To solve this problem, most studies improve the differential synchronization based on RSYNC. Yan *et al.* [15] noted that RSYNC uses a single round of messages to solve this problem, whereas research has shown that significant additional savings in bandwidth are possible using multiple rounds. Thus, they propose a new and simple algorithm for file synchronization based on set reconciliation techniques over a slow network. They find that single-round protocols are preferable in scenarios with small files and large network latencies [16]. However, in the case of large collections and slow networks, it may be preferable to use multiple rounds to further reduce the communication costs [17]–[19]. Guo and Li [20] focused on the single-round case and propose an algorithm based on the use of set reconciliation techniques. The experiment results show that the communication cost of their algorithm is often significantly smaller than that of RSYNC, particularly for notably similar files. Suel *et al.* [21] focused on maintaining large replicated collections of files or documents in a distributed environment with limited bandwidth. They propose a framework for remote file synchronization and describe several new techniques that result in significant bandwidth savings. Their focus is on applications where notably large collections must be maintained over slow connections. They show that a prototype implementation of their framework and techniques achieves significant improvements over RSYNC. Other studies examine more specific applications of synchronization. For example, Lareida *et al.* [22] studied P2P-based approaches and present Box2Box, which is a new P2P file synchronization application that supports new features not present in BitTorrent-Sync [23]. The master-peer feature of Box2Box enables a user to deploy a high-availability peer on a user-controlled nano data center. This super peer is responsible for storing all users files. Another study focuses on video synchronization. Zhang *et al.* [24] designed a video-sync (VSYNC), which is a video file synchronization system that efficiently uses a bidirectional communication link to maintain up-to-date video sources at remote ends to a desired resolution and distortion level. By automatically detecting and transmitting only the differences among video files, VSYNC can avoid unnecessary retransmission of the entire video when there are only minor differences among the video copies.

With the development of cloud computing, more cloud-based data synchronization services arise in the cloud storage field. The combination with cloud generates new challenges, such as tight resource limits, silently corruption or leakage of user's data. Security is the most concerned issue in cloud field [25]. To solve these issues, Han *et al.* designed, implemented, and evaluated MetaSync, a secure and reliable data synchronization service using multiple cloud synchronization services as untrusted storage providers. MetaSync provides better availability and performance, stronger confidentiality and integrity, and larger storage [26].

In these studies, there are two main approaches: 1) peer-to-peer approach and 2) cloud-based approach. Uppoor *et al.* [27] proposed a new approach for efficient cloud-based synchronization of an arbitrary number of distributed file system hierarchies. This approach maintains the advantages of peer-to-peer synchronization with the cloud-based approach that stores a master replica online. The proposed system performs data synchronization in a peer-to-peer manner, which eliminates the cost and bandwidth concerns that arise in the cloud mastercopy approach. Regardless of the type of synchronization, the consistency of the user's file is the essential requirement. Paper [28] discusses this problem and provides a solution for consistency protocol in data synchronization filed. Bao *et al.* proposed a user-view-based file consistency protocol with a conflict-resolution mechanism for data synchronization service. They also implement their consistency protocol into a prototype called SyncViews and set up a LAN-based environment to evaluate its feasibility and performance. The results show that their protocol works well in data synchronization processes and outperforms iFolder when the synchronized file becomes notably large.

These studies consider different aspects of synchronization to improve its efficiency; most of them use differential synchronization to decrease the amount of new data. It is a really efficient method, but most existing studies ignore the computing overhead. Furthermore, few of them focus on cloud storage. Thus, in this paper, we will introduce a new architecture to solve the problems in traditional differential synchronization and offload the computation work of terminal devices and cloud server.

III. Architecture of Differential Synchronization With Fog Computing

A. Differential Synchronization

The increasing availability of always-on Internet connections have increased the demand for applications that enable multiple users to collaborate with one another in real time [29]. In the cloud storage field, users may often modify their files, which are stored in the cloud. If we use traditional methods and reupload the entire file, there will be severe wasting of resources, particularly when we dispose large-scale files [30]. To improve the storage efficiency, there is a simple idea: the users can only upload the particular part of data that has been changed. How to find the changed part is the most critical procedure. In practical applications, most differential algorithms use the slide window detection. The size of the window determines the accuracy of the detection results. A smaller window size corresponds to more precise detection results. After we obtain the changed part, we must reconstruct the new file.

In conclusion, the emergence of differential synchronization decreases the volume of upload data. As we mentioned, the traditional differential synchronization retains the following disadvantages.

1) The CSPs manage the user's data; thus, users do not actually control the physical storage of their data, which results in data security issues.
2) The cloud server must compute and detect during every time of request, as too many requests may reduce the performance of the cloud server.
3) There are some repeated operations when we continuously modify the same data, and thus, we need a new architecture of differential synchronization. Fortunately, fog computing can help us realize what we want.

B. Why Fog Computing

Fog computing was first proposed by Cisco's Bonomi in 2011 [31]. In Bonomi's view, compared to highly concentrated cloud computing, fog computing is closer to the edge network. Considering these characteristics, fog computing is more suitable to the applications that are sensitive to delay. Furthermore, compared to sensor nodes, fog nodes have a certain storage capacity and data processing, which can help the cloud servers to dispose some computation work [32].

Fog computing is commonly a hierarchical architecture with three layers: the topmost layer is the cloud computing layer, which has powerful storage capacity and computation capability; the middle layer is the fog computing layer; and the bottom is the wireless sensor network (WSN) layer [33]. The main work of this layer is collecting data and uploading it to the fog server. In this paper, the bottom layer is different from traditional fog computing [34]. We focus on the application instead of WSN. The introduction of fog computing can offload the burden of terminal devices and cloud server and improve the work efficiency. In our scheme, we use the fog computing model and adopt the three-layer structure. Furthermore, we replace the WSN layer by the users local machine.

C. Our Architecture

In this paper, we propose a fog-based differential synchronization in the IoT field. The introduction of fog computing establishes a hierarchical architecture. The right half of Fig. 1 shows three typical elements: 1) cloud servers; 2) fog servers; and 3) users' devices from top to bottom. Different operations of users will generate different requests. Every request causes a synchronization with information, which includes the differential data and related information. They will be temporarily uploaded to the fog server. The fog servers save this information within a certain size. Once the size exceeds a set threshold, the fog servers will upload them to the cloud servers for total synchronization. The left half of Fig. 1 illustrates the change in data level. The blue circle represents the old file. We assume that the user adds two blocks to the old file, which are marked in green. The next step in our architecture is to upload these new blocks to the fog server instead of directly uploading them to the cloud server. The fog server will decide whether to synchronize to the cloud.

Unlike the traditional differential synchronization, not every instance of synchronization sends a request to the cloud server in our architecture. On the one hand, by partially offloading the work of the terminal devices and cloud server to the fog server, the performance of the cloud can be improved. On the other hand, if a user repeatedly modifies a place, the traditional differential will repeatedly upload this part of data. However, this

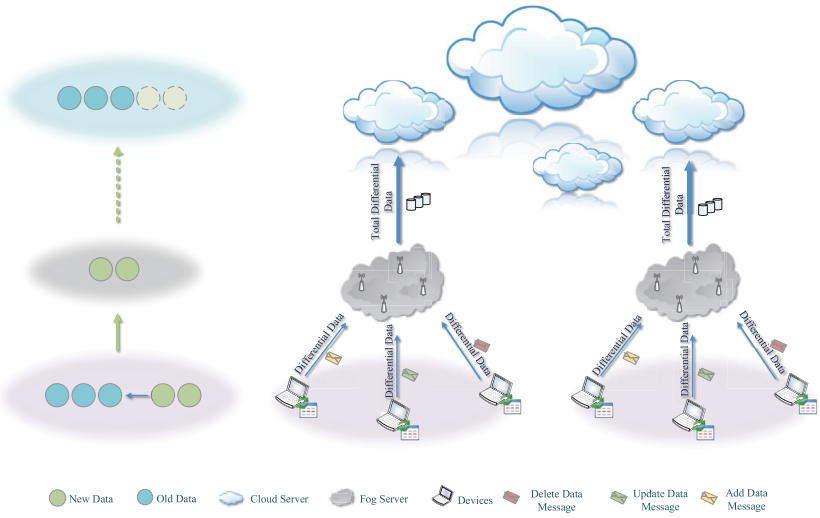


Fig. 1. Architecture of differential synchronization with fog computing.

situation can be improved in our architecture because not every time of modification will be synchronized to the cloud. We only upload the final version when the amount of data stored in the fog server reaches the threshold. These mechanisms are supported by our algorithms, which will be elaborated in the next section. In brief, our architecture includes three parts and has two main advantages compared to the traditional differential synchronization: 1) it can offload partial work to the fog and 2) it can further decrease the size of the data to be uploaded. We also considerate data security issues in cloud storage, which will be discussed in Section IV-C.

IV. Algorithm

In this section, we will introduce the solutions.

A. RSYNC

Before introducing our algorithm, we will introduce a traditional differential algorithm RSYNC, which is used in Linux systems as a data synchronization software. To facilitate understanding, we assume that there is a string "123abcdef" stored in the edge device and the cloud server. Then, the user changes it to "123ABCdefgh" and wants to synchronize it to the cloud. As shown in Fig. 2, after the cloud receives the sync request, it divides the string into three parts according to the window size, which is set as three bytes. We obtain three chunks (chunk[0]...chunk[2]) and compute the strong-check code and weak-check code for them. We take Ader32 as the weak-check code and MD5 as the strong-check code and save the results in a checksum table. Second, the cloud server transmits the table to the edge device. After the edge device receives the table, it checks the new string 123ABCdefgh according to the table.

In the "check new file," we take the same size window (three bytes) to check the string. In step one, the content in the widow is "123," which matches chunk[0]; thus, we record this information and the offset. In step two, the window slides to the forth byte. The content "ABC" cannot match any chunk, so the window slides to the next byte. Steps three

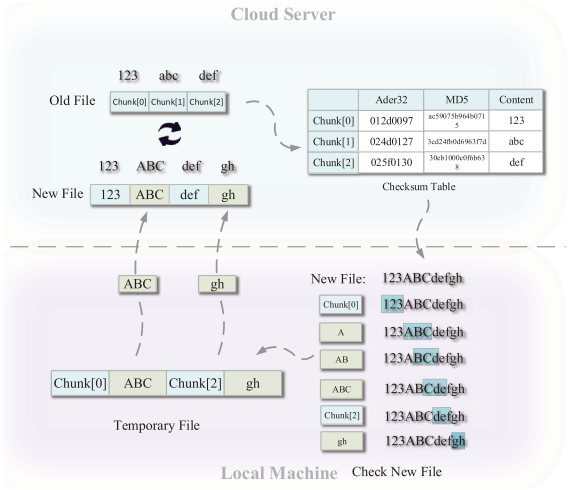


Fig. 2. Synchronization process with the RSYNC algorithm.

and four are identical to step two. In step five, chunk[2] is matched and recorded. In the final step, there is no matching. Thus, we obtain the matching information and new data ABC and "gh." Third, we upload this information to the cloud. Finally, the cloud server reconstructs 123ABCdefgh from the old string 123abcdef and the information from the user.

In this example, the amount of upload data is only 45% of the traditional synchronization. On the one hand, although we generate some table and related information, they are much less than the amount of the file. On the other hand, taking different check codes is a clever skill that can improve the computation efficiency. As we know, the computation of a strong check consumes much more resources than weak-check codes. However, the collision rate of a strong-check code is lower than that of a weak-check code. Thus, RSYNC compares the weak-check code first; if there are matching data, it uses a strong check to ensure it. This operation can relieve the work of the user and simultaneously ensure the accuracy. The algorithmic elaboration is listed in Algorithm 1.

Algorithm 1: RSYNC

Input: F_0,*window_size*, F_0 is the old file, which is stored in the cloud and user's mechine.
Output: F_n,F_n is the final version of file, which user synchronizes to the cloud.

```
for i = 1; i < n; i++ do
    Send_Request (CloudServer);
    Chunk[m] = Divide (F_n,window_size);
    for j = 0; j < m; m++ do
        Table1 ← Adler32 (Chunk[m]);
        Table2 ← MD5 (Chunk[m]);
    end
    Send (Table1, Table2) to Local;
    Slide_Check (F_n, Table1, window_size);
    if Matched then
        Check (F_n, Table2);
        if Matched then
            Record (Information, Newdata);
        else
            Slide_Check (next_position);
        end
    else
        Slide_Check (next_position);
    end
    Send (Information, Newdata) to cloud;
    Reconstruct (F_n, Information, Newdata);
end
```

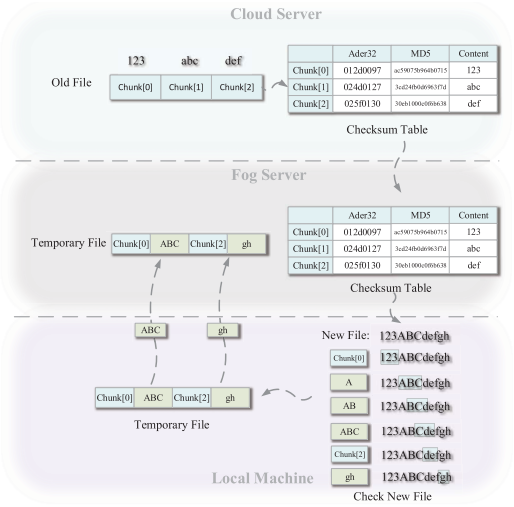


Fig. 3. FSYNC algorithm: not reaching the threshold.

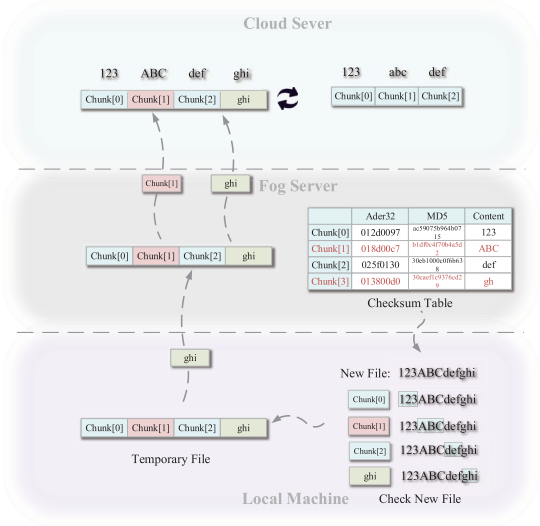


Fig. 4. FSYNC algorithm: reaching the threshold.

B. FSYNC

Although RSYNC has solved partial problems in the traditional cloud storage, it retains some deficiencies. There are too many times of requests when the edge device modifies F_0 to F_n. Every time of request will generate new data, although some of them are repeated, which causes a heavy load on the cloud server and lost of resources. As we mentioned, the fog server has a certain processing ability and storage capacity. Thus, we introduce the fog into our algorithm and design an Fog Sync (FSYNC), which can solve these problems. To facilitate understanding, we still assume that there is a string 123abcdef stored in the cloud server and edge device. Then, the user changes it to "123abcdefgh" and "123abcdefghi" step by step. Figs. 3 and 4 show how the FSYNC algorithm synchronizes this string to the cloud server step by step. In FSYNC, the fog server is deployed as a cache. The synchronization information is temporarily stored in the fog server and waits for the next update. When the total amount of data stored in the fog server reaches a threshold, FSYNC will perform a total synchronization.

Thus, there are two situations; the first situation is shown in Fig. 3. The first step is identical to RSYNC: after the cloud receives the sync request, it divides the string into three parts according to the window size. We obtain three chunks and compute their strong check code and weak check code. The results are saved in a checksum table and sent to the fog server. Second, the fog server saves a copy and sends the checksum table to the user. After the edge device receives the table, it checks the new string 123ABCdefgh step by step with a sliding window, which has been elaborated in RSYNC. Then, we obtain the matching information and new data ABC and gh. Third, the edge device uploads this information to the fog server. We assume that this time of synchronization does not reach the threshold; thus, the fog server does not upload this information to the cloud. It only updates the checksum table with the new information and generates a temporary file. We call this situation a local update, which indicates that we do not actually change the file in the cloud server. In other words, no request and data are sent to the cloud.

Another situation is shown in Fig. 4, where the edge device changes the string to "123ABCdefghi" based on the previous update. First, the user downloads the new checksum table from the fog server. Then, we check the new string with the new table. In this time, only "ghi" is the new data; thus, the edge device updates these data and the related information to the fog. Second, in the fog server, we assume that this time of change reaches the threshold; therefore, we must execute a

Algorithm 2: FSYNC

Input: F_0, *window_size*, *threshold*, F_0 is the old file, which is stored in the cloud and user's mechine.
Output: F_n,F_n is the final version of file, which user synchronizes to the cloud.

```
Chunk[m] = Divide (F0,window_size);
for j = 0; j < m; m++ do
    Table1 ← Adler32 (Chunk[m]);
    Table2 ← MD5 (Chunk[m]);
end
Send (Table1, Table2) to Fog;
for i = 1; i < n; i++ do
    Send_Request (FogServer);
    Compare (threshold);
    if Not Reach then
        Slide_Check (Fn, Table1, window_size);
        if Matched then
            Check (Fn, Table2);
            if Matched then
                Record (Information, Newdata);
            else
                Slide_Check (next_position);
            end
        else
            Slide_Check (next_position);
        end
        Record (Information, Newdata) in the Fog;
        Update (Table1, Table2);
    else
        Send (Information, Newdata) to cloud;
        Reconstruct (Fn, Information, Newdata);
        Clear the file in the Fog Server;
    end
end
```

total update. In the fog server, we compare the new data with the temporary file, which is generated in the last update. By comparison, we find that gh is replaced by ghi in the nearest update. Thus, we must only upload ABC, ghi, and some related information to the cloud. The final step is to reconstruct the string in the cloud server and clear the fog server. In this time of synchronization, only ABC and ghi are uploaded at last. However, if we use RSYNC, there are redundant data gh and a needless request. The algorithmic elaboration is shown in Algorithm 2.

The threshold is the maximal or minimal change rate. As shown in formula 1, in differential synchronization, a new file is composed with two parts: 1) new chunks and 2) old chunks. The new chunks are the differential data that are generated because of the file change; the old chunks are parts of the old file. As formula 2 shows, the change rate is the ratio of new chunks to the old chunks

$$\text{New\_file} = \text{new\_chunks} + \text{old\_chunks} \tag{1}$$

$$\text{Change\_rate} = \frac{\text{new\_chunks}}{\text{Old\_file}}. \tag{2}$$

As we mentioned, the threshold is used to decide whether to synchronize to the cloud server. We set the threshold as an interval [min, max]. As shown in formula 3, if the change rate is in this interval, the fog server will retain this synchronization till the next time. Otherwise, it will synchronize the file to the cloud server. The scale of the interval can be adjusted according to the requirement. We set the interval as [0.5, 1.5], which indicates that a half extra change will reach the threshold

$$f = \begin{cases} 1, & \text{Change\_rate} \geq \max \\ 0, & \min < \text{Chage\_rate} < \max \\ 1, & \text{Change\_rate} \leq \min . \end{cases} \tag{3}$$

We assume that the probability of exceeding the threshold after every modification is p. After n times of modifications and synchronization, the request times of RSYNC and FSYNC are

$$\text{RSYNC} = n \tag{4}$$

$$\text{FSYNC} = \sum_{i=1}^{n} f(p). \tag{5}$$

Formulas (4) and (5) show that the request time of FSYNC is always smaller than the request time of RSYNC as long as $p > 0$. Meanwhile, the amount of data also decreases in FSYNC. If we assume that there are m repeated modifications in n times of synchronization, the total amount of data that is uploaded to the cloud with RSYNC is $\sum_{i=1}^{n} \text{datasize}(i)$, whereas the total amount of data that is uploaded to the cloud with FSYNC is $\sum_{i=1}^{n-m} \text{datasize}(i)$. Obviously, $\sum_{i=1}^{n} \text{datasize}(i) > \sum_{i=1}^{n-m} \text{datasize}(i)$ as long as these synchronizations have repeated operations.

From the above analysis, we prove that FSYNC is theoretically more efficient than the traditional differential synchronization. The following question may arise: will the extra transmission between the fog server and the users in FSYNC increase the transmission time of synchronization? Of course not, the transmission between the fog server and users is always based on LAN, which has a higher speed than the WAN.

C. RS-FSYNC

By comparison, we find that FSYNC is more efficient than the traditional differential synchronization. However, an unsolved problem remains, which is the security issues. In the traditional storage schema, the users data are completely stored in the cloud servers. In other words, the users lose their right of control on data and face privacy leakage risk [35], [36]. Although encryption has been used in many studies, it is not absolutely effective [37]. In our new architecture, we can take advantage of the storage capacity of the fog server to solve this problem. To ensure the security of user's data, we add Reed–Solomon code into our algorithm. Reed–Solomon code is a type of erasure code that has always been used in the distributed storage field. The function of the Reed–Solomon code is correcting error by redundant data, which is generated by the original data. As shown in Fig. 5, first, we perform the mapping transformation on the file, so that each word of the file corresponds to a number in $GF(2^{\omega})$. After the mapping

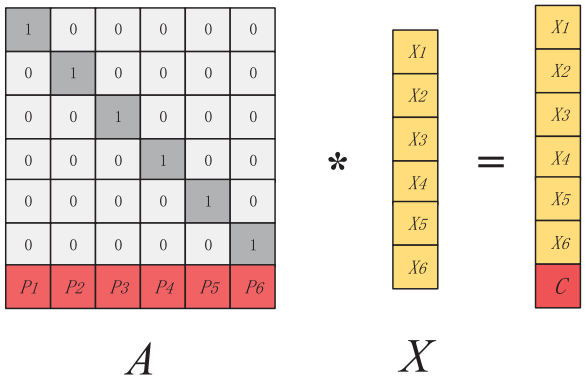


Fig. 5. Encoding illustration of Reed–Solomon code.

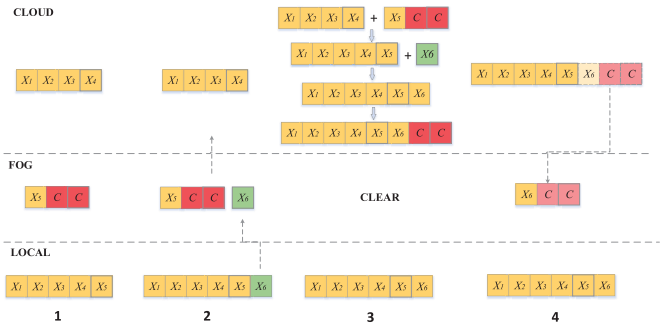


Fig. 6. Illustration of RS-FSYNC.

transformation, we obtain file matrix X. Then, we multiply file matrix X by the encoding matrix A. The multiplication generates k data blocks X_1 to X_6 and m redundant data blocks C. Reed–Solomon code has an important property: in the $k+m$ data blocks, if we have at least k data blocks, we can recover the original data with the encoding matrix. However, when the number of data blocks is less than k, it cannot be recovered. In RS-FSYNC, we use this property.

Based on the FSYNC algorithm, we make some changes to the file, which is stored in the cloud server. As shown in Fig. 6, in step 1, we perform encoding on the file and obtain X_1 to X_5 and two redundant data blocks C. We store these data blocks separately in the fog server and cloud server by the allocation in the figure. Three data blocks are stored in the fog server, and the number of data blocks is greater than the maximum limit of recovery. Thus, the cloud server provider or outside attacker cannot recover the file using the data stored in the cloud server. In other words, data security is ensured. Furthermore, the operations after updating are shown in steps 2–4. According to our FSYNC, if the new data X_6 do not reach the threshold, we upload them to the fog server. When the data in the fog server reach the threshold, we synchronize all of the data to the cloud server and clear the fog server. In the cloud server, we first reconstruct the original file, which is composed by $X_1 - X_5$. Then, we combine this file with the new data X_6 to obtain the newest file. Finally, we perform Reed–Solomon encoding on the newest file and obtain new $X_1 - X_6$ and two redundant data blocks C. We continue to store these data blocks separately in the fog server and cloud server by the allocation.

TABLE I
EXPERIMENT ENVIRONMENT

Operation System	Windows 10
CPU	Intel Core i7 2.50 GHz
Memory	8GB
Programming Language	C++/C#
Compiler	Visual Studio 2013 Professional
LAN Speed	100Mbps
WAN Speed	2Mbps

The specific allocation should satisfy formula 6, where parameter k is the number of blocks after the file is divided, parameter m is the number of redundant data blocks, and parameter r is the ratio of the data stored in the fog server to the data, which are stored in the cloud server. It is noteworthy that the redundant data are far smaller than the original data in size. Thus, we can nearly ignore their transmission overhead

$$\frac{m}{k+m} \leq \frac{k+m}{k} * r. \tag{6}$$

V. EVALUATION AND ANALYSIS

To prove the advantage of our architecture and algorithm, we evaluate them in different conditions and analyze the results. The experiment environment parameters are shown in Table I.

In our experiments, we often consider two situations of synchronization. In the first situation, the original file is 78.6 kB; then, we make some changes. After the first change, we obtain a new file (97.6 kB) and perform a synchronization. Then, we change the new file again and obtain a newer file (107 kB). Finally, we resynchronize it to the cloud (or fog). In the second situation, the original file is identical to that in the first situation (78.6 kB); then, we make some changes, obtain a new file (107 kB) and synchronize it to the cloud or fog. Then, we change the new file again and obtain a newer file (97.6 kB). Finally, we synchronize it again. In addition, the threshold for the fog server is set as [0.5, 1.5]. This interval presents the ratio of the new file size to the original file size. In other words, every change that exceeds 50% will reach the threshold and makes the fog server perform a total synchronization. To ensure the reliability of the experiment results, we test each case 100 times and take the average.

RSYNC is a widely used algorithm in differential synchronization. It has been accepted as the most representative differential synchronization algorithm for many years; thus, we select it for comparison. We also consider the traditional synchronization for comparison. In traditional synchronization, every change in data will generate a total upload of the data, which is a simple and convenient method. In both RSYNC and FSYNC, slide window detection is used. Different window sizes greatly affect the detection efficiency. In theory, a smaller window size makes the detection more accurate. A more accurate detection result implies less new data in the differential synchronization. However, a smaller window size corresponds to a lower processing efficiency. As shown in Fig. 7(a), we test the first step in the first situation with five different window sizes: 1/4, 1/6, 1/8, 1/10, and 1/20 of the test file. We find that

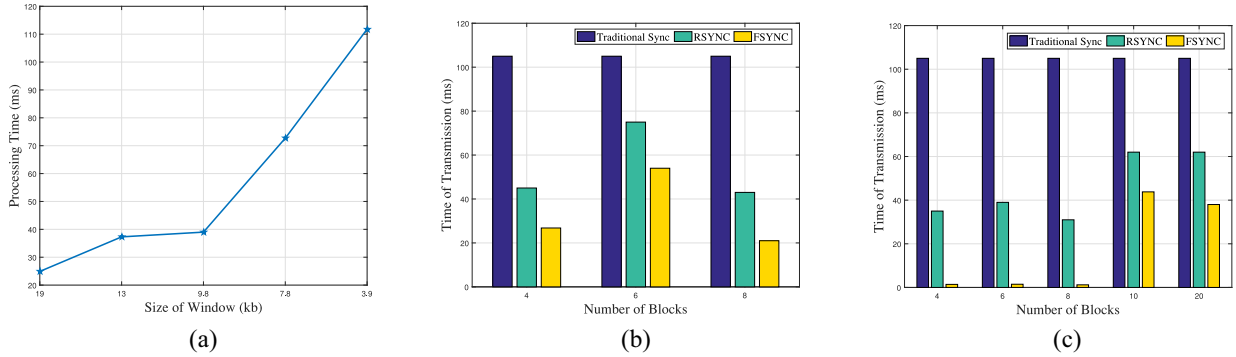


Fig. 7. Experiment results. (a) Effect of the window size on the processing time (78.6 kB→97.6 kB, 100 times). (b) Comparison of the transmission time in different synchronizations (78.6 kB→97.6 kB→107 kB, 100 times). (c) Comparison of the transmission time in different synchronizations (78.6 kB→107 kB→97.6 kB, 100 times).

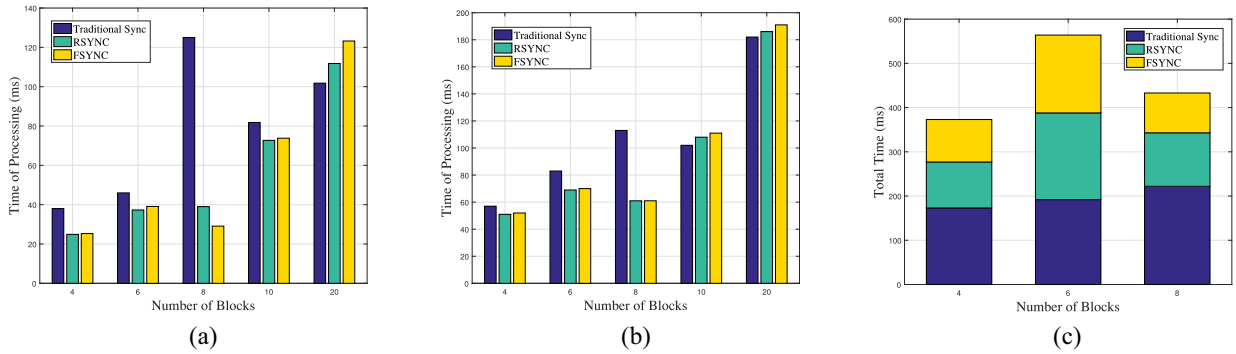


Fig. 8. Experiment results. (a) Comparison of processing time in different synchronizations (78.6 kB→97.6 kB→107 kB, 100 times). (b) Comparison of processing time in different synchronizations (78.6 kB→107 kB→97.6 kB, 100 times). (c) Total time of three synchronizations (78.6 kB→97.6 kB→107 kB, 100 times).

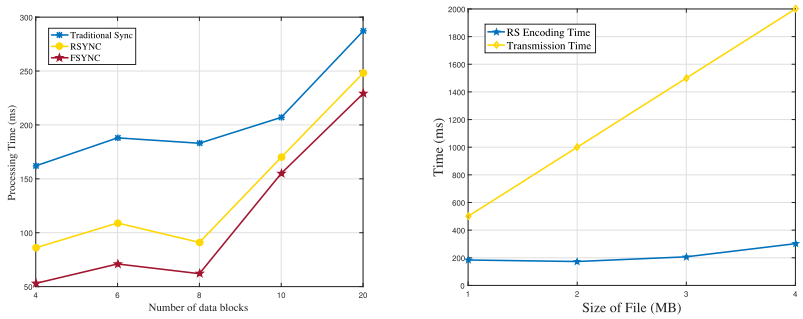
the processing time increases when the window size decreases. Therefore, in practical applications, a proper window size is particularly important. In the subsequent experiment, we do not take a fixed window size. We continue using 1/4, 1/6, 1/8, 1/10, and 1/20 of each file and denote them as "number of blocks" in the coordinate system.

Fig. 7(b) shows the transmission time of three synchronization methods with different window sizes in the first situation. In the three conditions, the traditional synchronization (blue) is always the longest because in every time of synchronization, the traditional method uploads the total file to the cloud, i.e., it uploads $97.6+107 = 204.6$ kB to the cloud every time. Thus, the results of the traditional Sync are identical in three conditions. Obviously, RSYNC (green) and FSYNC (yellow) save transmission time compared to the traditional Sync. As differential synchronizations, they do not need to upload the integrated file after every change. Moreover, the results show that FSYNC is more efficient than RSYNC because FSYNC further cuts the amount of new data by taking the fog server as a cache. Not every time of synchronization triggers a total synchronization. In situation one, when we change the file from 78.6 to 97.6 kB, this time of change does not reach the threshold, so the fog server temporarily saves the related information. After we change 78.6 to 97.6 kB, the change reaches the threshold. In the final synchronization, some repeated operations withdrawn in the last synchronization are saved. Thus, FSYNC transmit less data than RSYNC.

Fig. 7(c) shows the transmission time of three synchronizations with more window sizes in the second situation. In the five situations, the traditional Sync (blue) remains the longest one, followed by RSYNC (green), and FSYNC (yellow) is the shortest one. However, there are some differences in the first three situations. The transmission time of FSYNC is extremely short because in two times of changes, neither reach the threshold; thus, the new data are not uploaded to the cloud. To solve this problem, we need timestamps to perform forced synchronization. In practical application, we can set a time period, e.g., after seven days, if the fog server still cache new data, we perform forced synchronization regardless of whether the amount of new data reaches the threshold. However, this is not the research emphasis in this paper.

Fig. 8(a) and (b) shows the processing time of the three synchronizations in two different situations. We find that their time generally increases with the increase in number of blocks. This phenomenon is explained in Fig. 7. Except for individual exceptions, we find that the three synchronizations do not greatly differ from one another. FSYNC sometimes has the longest time, but it does not matter because the excessive overhead in processing is redeemable by saving time in transmission. To prove this statement, we test the total time, which includes the entire synchronization process.

Fig. 8(c) shows the total time of the three synchronizations in the first situation. The traditional Sync (blue) costs the most



(a) (b)

Fig. 9. Experiment results. (a) Total time of three synchronizations (78.6 kB→107 kB→97.6 kB, 100 times). (b) Comparison of RS encoding time and transmission time.

TABLE II
NUMBERS OF REQUESTS

Number of Blocks	Traditional	RSYNC	FSYNC
4	2	2	0
6	2	2	0
8	2	2	0
10	2	2	1
20	2	2	1

time because the total synchronization costs too much time in the transmission part. The RSYNC (green) is shorter than the traditional Sync. However, the FSYNC (yellow) is the shortest one because the transmission time accounts for a larger proportion in the total time. In other words, the overhead in the processing can be counteracted by the transmission.

Fig. 9(a) shows the total time of the three synchronizations in the second situation. These three synchronizations tend to increase with the increment of data blocks. In addition, the blue line, which represents the traditional approach, is the longest, and the yellow and red lines are behind it. Our algorithm FSYNC (yellow) is extremely fast in the continuous modification situation.

The above experiments results and analysis show that our architecture and FSYNC algorithm have better performance than the other algorithms. In addition, let us examine the numbers of requests to the cloud server. The test is in the second situation. The test result is shown in Table II, where two changes generate two requests in the traditional Sync and traditional differential Sync RSYNC. However, using the fog server, only one request is generated in FSYNC. In practical applications, FSYNC will sharply offload the work of the cloud.

In Section IV-C, we introduce Reed–Solomon into our algorithms and design a more secure algorithm RS-FSYNC. Will the introduction of RS coding affect the efficiency of synchronization? The answer is obvious: it will affect it but not too much. As we show in Fig. 9(b), we test four different file sizes of 1–4 MB. The blue line represents the RS encoding time, and the yellow line represents the transmission time. The time of RS encoding is far less than the transmission time. Furthermore, the growth rate of RS encoding is also smaller than the transmission time. When the file size increases, this gap will increase.

VI. CONCLUSION

Because of the increasing trend of IoT technology, the burden on users' devices and cloud server has become sharply heavier. By offloading all of the computing and storage to the cloud, the end users lose the operation abilities for private data. In this paper, we have designed a more efficient and secure cloud storage based on fog computing. By offloading part of the computing and storage work to the fog servers, the data privacy can be guaranteed. Moreover, to decrease the communication cost and reduce latency, we designed a differential synchronization algorithm. There is a common phenomenon that the new file is often slightly different from the previous version by one time of modification. If we upload the entire new file to the cloud server every time, the overheads will be huge. To solve this problem, previous studies focused on how to decrease the amount of new data [38]. Differential synchronization provides a feasible solution, but it increases the workload on the users' devices and the cloud server. By offloading part of the work to the fog server, the entire efficiency will be improved. We designed FSYNC and RS-FSYNC algorithms to support this architecture. The experiment results show that our architecture is feasible and has better performance than the other methods.

REFERENCES

[1] J. Ren, H. Guo, C. Xu, and Y. Zhang, "Serving at the edge: A scalable IoT architecture based on transparent computing," *IEEE Netw.*, vol. 31, no. 5, pp. 96–105, Aug. 2017.

[2] T. Li, S. Tian, A. Liu, H. Liu, and T. Pei, "DDSV: Optimizing delay and delivery ratio for multimedia big data collection in mobile sensing vehicles," *IEEE Internet Things J.*, vol. 5, no. 5, pp. 3474–3486, Oct. 2018.

[3] Z. Guan *et al.*, "Achieving efficient and secure data acquisition for cloud-supported Internet of Things in smart grid," *IEEE Internet Things J.*, vol. 4, no. 6, pp. 1934–1944, Dec. 2017.

[4] Z. Guan *et al.*, "Privacy-preserving and efficient aggregation based on blockchain for power grid communications in smart communities," *IEEE Commun. Mag.*, vol. 56, no. 7, pp. 82–88, 2018.

[5] K.-K. R. Choo, R. Lu, L. Chen, and X. Yi, "A foggy research future: Advances and future opportunities in fog computing research," *Future Gener. Comput. Syst.*, vol. 78, pp. 677–697, Jan. 2018.

[6] O. Osanaiye *et al.*, "From cloud to fog computing: A review and a conceptual live VM migration framework," *IEEE Access*, vol. 5, pp. 8284–8300, 2017.

[7] T. Wang *et al.*, "A comprehensive trustworthy data collection approach in sensor-cloud system," *IEEE Trans. Big Data*, to be published, doi: 10.1109/TBDATA.2018.2811501.

[8] D. Zhang, R. Shen, J. Ren, and Y. Zhang, "Delay-optimal proactive service framework for block-stream as a service," *IEEE Wireless Commun. Lett.*, vol. 7, no. 4, pp. 598–601, Aug. 2018.
[9] A. Tridgell and P. Mackerras. (1996). *The rsync Algorithm*. [Online]. Available: https://openresearch-repository.anu.edu.au/handle/1885/40765
[10] M. Huang, A. Liu, N. N. Xiong, T. Wang, and A. V. Vasilakos, "A low-latency communication scheme for mobile wireless sensor control systems," *IEEE Trans. Syst., Man, Cybern., Syst.*, vol. 49, no. 2, pp. 317–332, Feb. 2019.
[11] D. Zhang *et al.*, "Two time-scale resource management for green Internet of Things networks," *IEEE Internet Things J.*, vol. 6, no. 1, pp. 545–556, Feb. 2019.
[12] X. Peng *et al.*, "BOAT: A block-streaming app execution scheme for lightweight IoT devices," *IEEE Internet Things J.*, vol. 5, no. 3, pp. 1816–1829, Jun. 2018.
[13] H.-Y. Chen, Y.-H. Lin, and C.-M. Cheng, "COCA: Computation offload to clouds using AOP," in *Proc. IEEE/ACM Int. Symp. Clust. Cloud Grid Comput.*, Ottawa, ON, Canada, 2012, pp. 466–473.
[14] N. Ramsey and E. Csirmaz, "An algebraic approach to file synchronization," *ACM SIGSOFT Softw. Eng. Notes*, vol. 26, no. 5, pp. 175–185, 2001.
[15] H. Yan, U. Irmak, and T. Suel, "Algorithms for low-latency remote file synchronization," in *Proc. INFOCOM 27th Conf. Comput. Commun.*, 2008, pp. 156–160.
[16] U. Irmak, S. Mihaylov, and T. Suel, "Improved single-round protocols for remote file synchronization," in *Proc. INFOCOM 24th Annu. Joint Conf. IEEE Comput. Commun. Soc.*, vol. 3, 2005, pp. 1665–1676.
[17] G. Cormode, M. Paterson, S. C. Sahinalp, and U. Vishkin, "Communication complexity of document exchange," in *Proc. SODA*, 2000, pp. 197–206.
[18] T. Schwarz, R. W. Bowdidge, and W. A. Burkhard, "Low cost comparisons of file copies," in *Proc. 10th Int. Conf. Distrib. Comput. Syst.*, 1990, pp. 196–202.
[19] M. Z. A. Bhuiyan, J. Wu, G. Wang, T. Wang, and M. M. Hassan, "e-Sampling: Event-sensitive autonomous adaptive sensing and low-cost monitoring in networked sensing systems," *ACM Trans. Auton. Adapt. Syst.*, vol. 12, no. 1, p. 1, 2017.
[20] D. Guo and M. Li, "Set reconciliation via counting bloom filters," *IEEE Trans. Knowl. Data Eng.*, vol. 25, no. 10, pp. 2367–2380, Oct. 2013.
[21] T. Suel, P. Noel, and D. Trendafilov, "Improved file synchronization techniques for maintaining large replicated collections over slow networks," in *Proc. IEEE 20th Int. Conf. Data Eng.*, 2004, pp. 153–164.
[22] A. Lareida, T. Bocek, S. Golaszewski, C. Lüthold, and M. Weber, "Box2Box—A P2P-based file-sharing and synchronization application," in *Proc. IEEE 13th Int. Conf. Peer-to-Peer Comput. (P2P)*, 2013, pp. 1–2.
[23] BBC. (2013). *Megaupload File-Sharing Site Shut Down*. Accessed: May 10, 2018. [Online]. Available: http://www.bbc.co.uk/news/technology-16642369
[24] H. Zhang, C. Yeo, and K. Ramchandran, "VSYNC: Bandwidth-efficient and distortion-tolerant video file synchronization," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 22, no. 1, pp. 67–76, Jan. 2012.
[25] Q. Liu, G. Wang, X. Liu, T. Peng, and J. Wu, "Achieving reliable and secure services in cloud computing environments," *Comput. Elect. Eng.*, vol. 59, pp. 153–164, Apr. 2017.
[26] S. Han *et al.*, "MetaSync: Coordinating storage across multiple file synchronization services," *IEEE Internet Comput.*, vol. 20, no. 3, pp. 36–44, May/Jun. 2016.
[27] S. Uppoor, M. D. Flouris, and A. Bilas, "Cloud-based synchronization of distributed file system hierarchies," in *Proc. IEEE Int. Conf. Clust. Comput. Workshops Posters (CLUSTER WORKSHOPS)*, 2010, pp. 1–4.
[28] X. Bao *et al.*, "SyncViews: Toward consistent user views in cloud-based file synchronization services," in *Proc. IEEE 6th Annu. Chinagrid Conf. (ChinaGrid)*, 2011, pp. 89–96.
[29] N. Fraser, "Differential synchronization," in *Proc. 9th ACM Symp. Doc. Eng.*, 2009, pp. 13–20.
[30] D. Zhang *et al.*, "Utility-optimal resource management and allocation algorithm for energy harvesting cognitive radio sensor networks," *IEEE J. Sel. Areas Commun.*, vol. 34, no. 12, pp. 3552–3565, Dec. 2016.
[31] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the Internet of Things," in *Proc. ACM 1st Ed. MCC Workshop Mobile Cloud Comput.*, 2012, pp. 13–16.
[32] T. Wang *et al.*, "Fog-based storage technology to fight with cyber threat," *Future Gener. Comput. Syst.*, vol. 83, pp. 208–218, Jun. 2018.
[33] M. Z. A. Bhuiyan *et al.*, "Dependable structural health monitoring using wireless sensor networks," *IEEE Trans. Depend. Secure Comput.*, vol. 14, no. 4, pp. 363–376, Jul./Aug. 2017.
[34] T. Wang *et al.*, "Data collection from WSNs to the cloud based on mobile fog elements," *Future Gener. Comput. Syst.*, Jul. 2017, doi: 10.1016/j.future.2017.07.031.
[35] T. Wang *et al.*, "A three-layer privacy preserving cloud storage scheme based on computational intelligence in fog computing," *IEEE Trans. Emerg. Topics Comput. Intell.*, vol. 2, no. 1, pp. 3–12, Feb. 2018.
[36] C.-Z. Gao, Q. Cheng, P. He, W. Susilo, and J. Li, "Privacy-preserving naive bayes classifiers secure against the substitution-then-comparison attack," *Inf. Sci.*, vol. 444, pp. 72–88, May 2018.
[37] C.-Z. Gao, Q. Cheng, X. Li, and S.-B. Xia, "Cloud-assisted privacy-preserving profile-matching scheme under multiple keys in mobile social network," *Clust. Comput.*, pp. 1–9, Feb. 2018, doi: 10.1007/s10586-017-1649-y.
[38] T. Wang *et al.*, "Big data reduction for a smart city's critical infrastructural health monitoring," *IEEE Commun. Mag.*, vol. 56, no. 3, pp. 128–133, Mar. 2018.

Tian Wang received the B.Sc. and M.Sc. degrees in computer science from Central South University, Changsha, China, in 2004 and 2007, respectively, and the Ph.D. degree from the City University of Hong Kong, Hong Kong, in 2011.

He is currently a Professor with the National Huaqiao University of China, Quanzhou, China. His current research interests include wireless sensor networks, fog computing, and mobile computing.

Jiyuan Zhou received the B.S. degree from Tianjin Polytechnic University, Tianjin, China, in 2016. He is currently pursuing the master's degree with Huaqiao University, Quanzhou, China.

His current research interests include security in wireless networks, fog computing, and security in cloud storage.

Anfeng Liu received the M.Sc. and Ph.D. degrees in computer science from Central South University, Changsha, China, 2002 and 2005, respectively.

He is a Professor with the School of Information Science and Engineering, Central South University. His current research interests include cyber-physical systems, service network, and wireless sensor network.

Dr. Liu is a member (E200012141M) of the China Computer Federation.

Md Zakirul Alam Bhuiyan (M'09–SM'17) received the B.Sc. degree in computer science and technology from International Islamic University Chittagong, Chittagong, Bangladesh, in 2005, and the M.Eng. and Ph.D. degrees in computer science and technology from Central South University, Changsha, China, in 2009 and 2013, respectively.

He is currently an Assistant Professor (research) with the Department of Computer and Information Sciences, Fordham University, New York, NY, USA. He was a Post-Doctoral Fellow with Central South University, a Research Assistant with the Hong Kong Polytechnic University, Hong Kong, and a Software Engineer in industry. His current research interests include dependable cyber-physical systems, wireless sensor network applications, network security, and sensor-cloud computing.

Dr. Bhuiyan has served as a Managing Guest Editor, the Program Chair, the Workshop Chair, the Publicity Chair, a TPC member, and a Reviewer of international journals/conferences. He is a member of the ACM and the Center for Networked Computing.

Guojun Wang received the B.Sc. degree in geophysics and M.Sc. and Ph.D. degrees in computer science from Central South University, Changsha, China, in 1992, 1996, and 2002, respectively.

He is currently the Pearl River Scholarship Distinguished Professor with Guangzhou University, Guangzhou, China. He was a Professor with Central South University, a Visiting Scholar with Temple University, Philadelphia, PA, USA, and Florida Atlantic University, Boca Raton, FL, USA, a Visiting Researcher with the University of Aizu, Aizuwakamatsu, Japan, and a Research Fellow with Hong Kong Polytechnic University, Hong Kong. His current research interests include cloud computing, trusted computing, and information security.

Dr. Wang is a Distinguished Member of the CCF and a member of the ACM and IEICE.

Weijia Jia (M'97–SM'02) was born in 1957.

He is a Professor with the Data Science Center, University of Macau, Macau, China. His current research interests include next generation wireless communication, protocols, and heterogeneous networks.

Prof. Jia has served as an Editor or a Guest Editor for international journals and as the PC Chair or a member/keynote speaker for various prestigious international conferences. He is a member of the ACM and CCF.