Secure Sharing of Private Locations through Homomorphic Bloom Filters

Yunhe Feng, Zheng Lu, Qing Cao
University of Tennessee, Knoxville
Email: {yunhefeng, zlu12, cao}@utk.edu

***Abstract*—Location information is becoming increasingly popular in online social networks, vehicle networks, and online games. In this paper, we develop a distributed protocol that allows one party to determine, in a private and secure manner, whether or not the trajectory of a second party has an intersection with specific locations of interest. Our design is fully flexible, meaning that each user is able to specify what kind of datasets they would like to make visible, and be queried by other users. The methodology is based on developing a generalized set membership check approach, using an advanced data structure called the bloom filter. To demonstrate its feasibility and usability, we offer three working prototypes, which are implemented on the open-source homomorphic libraries. Our preliminary results illustrate the performance and overhead of the proposed approaches as well as the security of the protocol designs.**

***Keywords*—Location sharing; Bloom filter; Homomorphic encryption; Location security; Location privacy**

I. Introduction

Location data is becoming increasingly popular in online social networks, vehicle networks, and online games. Companies like Google and Tesla have heavily invested in autonomous vehicles that make driving decisions based on real-time highly accurate location coordinates. On smartphones, more users are sharing location data, ranging from ride-sharing apps to geographically enhanced games, such as the Pokemon Go, among others. In these scenarios, one big concern is the privacy of users, as unexpected leaks of users' location trajectories will allow potentially malicious attacks to gain advantages in the real world [1]. For example, by knowing what time a user leaves and returns home each day, a potential third-party attacker could identify the time periods during which the user's home is not occupied. The attacks range from mild inconveniences such as privacy leaks to much more serious attacks such as break-ins.

One significant challenge on keeping location data secure is that we should not trust servers as safe against attacks [2]. Indeed, the significant increase on hacking activities against servers in recent years means that if we store non-encrypted data on servers, we will face the risks of data leakage when the servers are compromised. Perhaps paradoxically, when building apps that involve location data, servers typically perform extensive computation on the location traces, making it necessary for the servers to be able to decrypt data as needed. For example, Facebook servers need users' non-encrypted locations to find nearby friends. Mobile service provider need users' rough location to analyze smartphone usage behaviors [3]. Consequently, we ask, is it possible for us to maintain the security of location traces while *allowing* servers to perform location-specific computations, such as calculating intersections?

DOI 10.1109/BDS/HPSC/IDS18.2018.00034

In this paper, we develop a secure computation framework on location data where the servers have no knowledge of its original data, i.e., the servers do not keep the private keys to decrypt data. In this way, compromised servers will not cause leaked user data. Our work is motivated and enabled by the recently developed fully homomorphic encryptions, where recent progress demonstrated it is feasible to perform meaningful and predictable computations on encrypted data without decrypting them first [4], [5]. Although existing primitives primarily support simple operations such as bit-wise operators, additions, and multiplications, in this work, we build on these homomorphic techniques, and expand them to support location-specific calculations. Specifically, we focus on one commonly used building-block operation in location based data processing: computing the location intersections.

Formally, this computational operation assumes that two users, Alice and Bob, want to find out if their location datasets (e.g., collections of singleton locations, or trajectories, or areas within specified boundaries) have intersections. Figure 1 shows an overview of the proposed framework, where the actions taken by Alice, Bob, and the cloud aggregation server are illustrated. The application model works as follows: Alice first publishes her location datasets that are encrypted using her public key, via the cloud aggregation server, so that Bob does not get access to the plain data directly. Instead, Bob can build a query based on his location sets and send it to the aggregation server. Note that as Bob also cares about his privacy, his query is also encrypted by Alice's public key. To increase security, before being encrypted, both Alice's location dataset and Bob's query are inserted into an advanced data structure called the bloom filter [6], which hashes all location data into binary strings. In this way, even when the cloud aggregation server is compromised and the secret key of Alice is hacked at the same time, the plain location data is still not leaked. Next, the aggregation server performs the *matching* step, where Alice's encrypted datasets are matched against Bob's query, and a (still encrypted) computational result is returned to Alice. Alice can decide whether there is an intersection between the incoming query and her datasets by decrypting the result using her own private key. If there is an intersection, Alice is probably able to send additional information to Bob.

We now make a few remarks on this computational model. First, by design, this model ensures that the location datasets are fully secure, as Alice will only publish the hashed datasets after encryption. On the side of Bob, the query location data is also pre-processed using hash functions, and then encrypted using Alice's public key. Hence, Alice and Bob never need

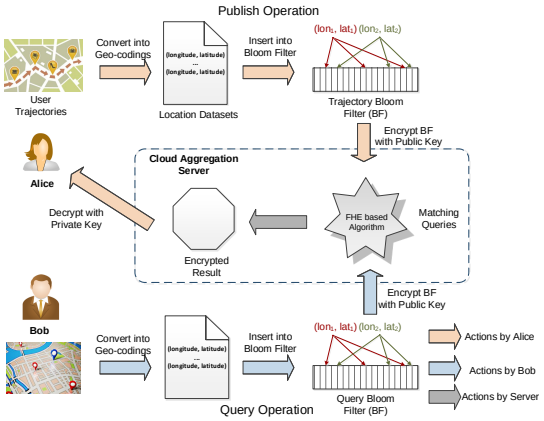

Fig. 1. Computational model architecture

to share the plaintext of their location trajectories directly. Second, this computational model is particularly secure against compromised servers, as the aggregation server does not keep any plaintext of location trajectories. Finally, only queries that return positive results (i.e., there are intersections) will lead to further interactions between Alice and Bob. For queries that do not return intersections, Alice and Bob do not learn about each other's locations.

The contributions of this paper are as follows. First, we design and propose a fully homomorphic version of the bloom filter. Second, we develop multiple optimized protocols that allow one party to determine, in a private and secure manner, whether or not a second party's trajectory has an intersection with the first party. Our design is fully flexible, meaning that each user is able to specify what kind of datasets (e.g., trajectories, areas, and isolated points) they would like to make visible, and be queried by other users. Third, we offer working prototypes based on the open-source homomorphic libraries. Additionally, our preliminary evaluation results on a real-world dataset, which is collected in a city area on users' smartphones, demonstrate the feasibility of the proposed approaches as well as the security of the protocol designs.

The remaining of this paper is organized as follows. We first review the related work in Section II, and then we present the problem formulation and the protocol designs in Section III. The analysis of security is discussed in Section IV, followed by the evaluation results in Section V. Finally, we conclude the paper in Section VI.

II. Related Work

Previous work on keeping location secure has investigated multiple directions. The first direction, which is called k-anonymous obfuscation [7], tries to hide the true locations of users by obfuscating them to the granularity of larger cells. Such methods, though privacy-aware, make it harder to develop applications that require the precise locations of users. Another method is through statistical methods [8], which add random noise to the samples of individual users, but keep the global statistical parameters to be more or less reliable. Again, such methods are only suitable for large-scale statistical needs, but are not useful where one user's data needs to be exploited for application needs, e.g., ride sharing and navigations.

Our proposed approaches fall under the third direction that performs encryption and decryption methods on the location data in order to achieve location security and privacy even when servers may be compromised. One recently developed encryption scheme suitable for this purpose is the homomorphic encryption, which aims to support complex processing on the encrypted data without decrypting them first, yet still yielding results that, once decrypted, are meaningful and correct. The homomorphic encryption has widely inspired applications [9], [10] of cloud computing.

Existing work in this direction has attempted to apply homomorphic computing techniques to process location data. For example, the work in [11] utilized partially homomorphic encryption to develop a privacy-preserving location proximity protocol called the InnerCircle. A more recent work [12] adopted homomorphic encryption to determine an optimal meeting location for a group of users. Note that all of these works are implemented based on partially homomorphic encryption methods that only support limited types of operations, i.e., either addition or multiplication, but not both.

A common task needed in location data processing is related to checking intersections between private sets. To this end, previous work has taken advantage of a data structure called the bloom filter, which supports constant time checkings on set memberships. For example, the work in [13] proposed the garbled bloom filters, based on which the oblivious Bloom intersection is performed to check the private set intersections. Another work [14] used the bloom filters to represent the location tags to conduct private proximity tests. In [15], the bloom filter was combined with partially homomorphic (Goldwasser−Micali) encryption to design an outsourced private set intersection protocol. Because partially homomorphic encryption only supports either the multiplication operation or addition operation, the work [15] had to rely on additional methods, such as the Sander Young Yung technique [16], to mimic the second operation but with a failure probability.

In contrast to these existing efforts, our work is directly motivated by the recent progress to develop fully homomorphic encryption schemes, such as the well-known scheme developed by Gentry in 2009 [4]. Compared with partially homomorphic encryption, this scheme is far more powerful because it supports both multiplication and addition operations simultaneously and allows for arbitrary computations on the encrypted data principally [17]. Inspired by Gentry's work [4], several practically feasible fully homomorphic encryption schemes have been developed [5], [18], [19]. Therefore, we build our protocol on top of existing fully homomorphic computing libraries, but we note that future developments of better paradigms will lead to lower computing cost and overhead, as well as better security in our system.

III. Framework Design

A. Assumptions

Our system design consists of three parties: Alice, Bob, and the aggregation server. In a typical application scenario, Alice first selectively publishes her location datasets after encrypting

them with her public key. Later, another user Bob sends his location queries encrypted by Alice's public key to the aggregation server. The server is responsible for performing computation tasks on the encrypted data, and sends the (still encrypted) results to Alice. Finally, Alice decides if there is an intersection by decrypting the results via her own private key. We assume that Alice and Bob are usually not malicious. They will follow the protocol correctly, but once the protocol has ended they can perform any computation they want on the information (encrypted or otherwise). If Alice finds out that there is an intersection with the trajectory of the query, the further interactions between Alice and Bob are out of the scope of this protocol.

B. Background

As shown in Figure 1, the homomorphic encryption method and the bloom filter are the two core components in our protocol. Next, we give a little background information about the two components.

Homomorphic Encryption: This method takes a foundational role in our system. Briefly, in a fully homomorphic encryption scheme, the following equations hold true:

$$m_1 + m_2 = D(E(m_1) + E(m_2)) \tag{1}$$

$$m_1 * m_2 = D(E(m_1) * E(m_2)) \tag{2}$$

In these equations, $E()$ represents the encryption operation, and $D()$ represents the decryption operation. The computation that is performed with encrypted values can be translated to operations in the plain text domain. This allows parties to perform blind computation on encrypted values.

Bloom Filter: We use standard notations on bloom filters to represent their operations, as follows:

- m: the size (total number of bits) of the bloom filter;
- k: the number of hash functions;
- n: the number of elements inserted in the bloom filter;
- p: the false positive probability of the bloom filter;
- t: the number of bits flipped to one.

The probability of false positives p given a parameter setting (m, k, n) can be calculated as:

$$p \approx (1 - e^{\frac{-kn}{m}})^k \tag{3}$$

For a given bloom filter size m and the number of inserted elements n, the number of hash functions k that minimizes the false positive is:

$$k = \frac{m}{n} \ln 2 \tag{4}$$

For a given number of inserted elements n and the desired false positive p, the required number of bits m is:

$$m = -\frac{n \ln p}{(\ln 2)^2} \tag{5}$$

We use these results later for parameter settings and the performance analysis of our protocol in Section V.

C. Main Ideas on an Ideal Protocol

The main idea for an ideal protocol works as follows. The user Alice first inserts all locations that she wants to publish into a configured bloom filter BF_A, by hashing each of these locations k times into BF_A, and flipping corresponding bits as 1. After all locations are inserted, suppose that t bits at index $a_1, a_2, \ldots, a_t$ in BF_A have been set as 1s. Our next goal is to encrypt the bloom filter BF_A properly. Specifically, we observe that its flipped bits can be represented as a polynomial:

$$f(x) = \prod_{i=1}^{t}(x - a_i) \tag{6}$$

Alice then sends the encrypted polynomial $f(x)$ either in the product form or the expanded form to the aggregation server. Alice also sends the details on the k hash functions, and the configuration parameters for BF_A. This is necessary as such information will be later used by Bob for encryption needs. Even though Alice sends the hash functions, as the $f(x)$ is encrypted with the public key, the aggregation server has no way to learn which bits have been set as 1s in BF_A.

The aggregation server next waits for the incoming queries. When doing the query, Bob does not send the raw location data to the server. Instead, he will first construct a new bloom filter BF_B by obtaining the k hash functions and BF_A's configuration parameters from the server. Then the queried location is hashed into BF_B by flipping k bits at index $b_1, b_2, \ldots, b_k$. Next, Bob encrypts the k indices as $E(b_1), E(b_2), \ldots, E(b_k)$ using Alice's public key, and sends them to the server to check whether BF_A has the corresponding bits flipped as 1s by evaluating $f(x)$ in the encrypted form as $E(f(E(b_i)))$. We know that, based on the nature of this encryption method, the following equation also holds true:

$$D(E(f(E(b_i)))) = f(b_i) \tag{7}$$

Furthermore, if the bit at index b_i in BF_A has been flipped to 1 by Alice earlier, this evaluation result must be 0 based on the product nature of the polynomial. Therefore, if we construct another polynomial as $\sum_{i=1}^{k} E(f(E(b_i))^2)$ (we use square is to prevent the evaluation may sometimes lead to negative numbers), we have:

$$H(b) = D(\sum_{i=1}^{k} E(f(E(b_i))^2)) = \sum_{i=1}^{k} f(b_i)^2 \tag{8}$$

Observe that in this equation, only when all bits at the k indices in BF_A have been flipped to 1, the result $H(b)$ will be 0. If $H(b)$ is not 0, then it means that the query is not in the bloom filter of Alice. We note that the multiplication steps for computing $f(b_i)^2$ is t (based on the factorized form of $f(x)$). So the total time of multiplication involved in computing $H(b)$ is $O(k*t)$. A large t, like 200, makes such a multiplication impractical, because the fast increased size of ciphertexts worsens the computation overhead greatly. Some techniques, such as relinearization [20] and approximate eigenvector method [21], have been proposed to balance the multiplication computation capacity and computation overhead, but they still do not work efficiently on a huge multiplication depth in real applications. Therefore, we design the following optimizations towards

practical protocols containing a greatly reduced number of multiplication operations. With these optimizations, the protocols can be made practical. We call these protocols *practical optimizations* for this reason.

D. Optimization 1: A Lightweight Homomorphic Bloom Filter

We now describe the first working design using fewer additions and multiplications compared with the idealized design above. Specifically, it requires k additions, no multiplication and one decryption to speed up the query. The pseudocode is shown in Algorithm 1. Alice first inserts her locations into a bloom filter BF_A using k hash functions, then encrypts the value of every bit $v_1, v_2, \ldots, v_m$ in BF_A using her public key PK. The $E(v)$ is then submitted to the aggregation server. Next, Bob hashes his queried location into BF_B, a bloom filter configured with the same parameters as BF_A, where the bits at index $b_1, b_2, \ldots, b_k$ are flipped to 1s. These k indices are submitted to the server in plaintext, which we think is still safe because the chosen k cryptographic hash functions ensure that the server can hardly compute the plain text from hashed values reversely. The server then sums all elements in $E(v)$ whose index is in b. At last, the server subtracts k, i.e., the length of b, from the sum and returns the ciphertext result to Alice for decryption using her secret key SK. The whole evaluation and decryption can be expressed as follows:

$$plain\_result = D(\sum_{j=1}^{k} E(v_{b_j}) - k) \tag{9}$$

If $plain\_result$ equals 0, it implies that all values of v_{b_j} are same and equal 1. In other words, all bits at index b_j in BF_A are flipped as 1s. So, we can say Bob interacts Alice with a certain probability of false positive expressed by Equation 3. Otherwise, they do not intersect at the queried location 100%.

Algorithm 1 A Lightweight Design of the Homomorphic Bloom Filter

Alice: Bloom Filter Encryption
$v \leftarrow$ all bits in BF_A
for $i = 1 \rightarrow m$ **do** ▷ m is the size of BF_A
  $E(v_i) \leftarrow$ encrypt v_i using PK
end for
send $E(v)$ to the aggregation server

Bob: Query Encryption
$b \leftarrow$ the indices of bits flipped to 1s in BF_B
send b to the aggregation server

Sever: Evaluation
$sum \leftarrow 0$
for $j = 1 \rightarrow k$ **do**
  $sum \leftarrow sum + E(v_{b_j})$
end for
$cipher\_result \leftarrow sum - k$
send $cipher\_result$ to Alice

Alice: Result Decryption
$plain\_result \leftarrow$ decrypt $cipher\_result$ using SK

E. Optimization 2: An Improved Homomorphic Bloom Filter

To further improve the security of Bob's data in optimization O1, we propose an improved design as shown in Algorithm 2. Like optimization O1, Alice first inserts her locations into a bloom filter BF_A, where t bits at index $a_1, a_2, \ldots, a_t$ have been set as 1s. Then Alice uses her public key PK to encrypt each negated a_i as the encrypted $E(-a_i)$, and submits them to the aggregation server. After launching a query, Bob first requests Alice's PK and her k hash functions from the aggregation server. Then Bob hashes his queried location into BF_B with flipping bits at index $b_1, b_2, \ldots, b_k$ to 1s. The indices are then encrypted as $E(b_1), E(b_2), \ldots, E(b_k)$ by PK and sent to the server. Once receiving the encrypted query from Bob, the server starts the evaluation based on $E(-a)$ and $E(b)$. Specifically, for each $E(b_j)$, the server performs $E(b_j) + E(-a_i)$, where $i = 1, 2, \ldots, t$, and gets t encrypted results which are then decrypted as $D(E(b_j) + E(-a_i))$ respectively by Alice using her secret key SK. Once one of the t encrypted results of b_j equals 0, it means the bit at index b_j in BF_A is set as 1. If all bits at index $b_1, b_2, \ldots, b_k$ in BF_A are set as 1s, we can conclude Bob has an interaction with Alice at a certain confidence level which is determined by BF_A's rate of false positives (we analyze the impact of this false positive rate later). Otherwise, Bob has no intersection with Alice at the queried location 100%. Note that in this design, the information of Bob might still be leaked to Alice no matter whether they have trajectory intersections or not, because Alice can obtain the value of b_j by:

$$b_j = \frac{\sum_{i=1}^{t}[D(E(b_j) + E(-a_i)) + a_i]}{t} \tag{10}$$

To address this problem, we introduce the randomness during the evaluation by multiplying $E(b_j) + E(-a_i)$ by a random positive integer z. Thus, if $D(E(b_j)+E(-a_i))$ equals 0, $D((E(b_j) + E(-a_i)) * z)$ still equals 0, while other values vary dramatically.

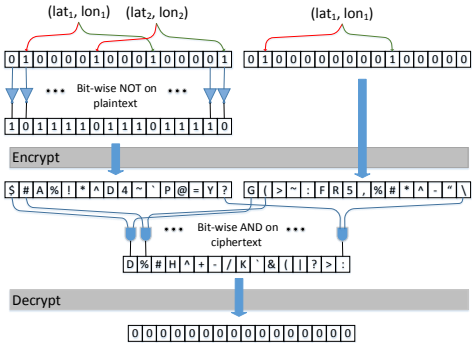

Fig. 2. Bloom filter encrypts a query with the bit-wise encryption scheme

F. Optimization 3: A Bit-wise Homomorphic Bloom Filter

Although the optimization O1 and O2 are feasible, they are built entirely on top of the underlying homomorphic encryption scheme, and treat such a scheme as a black box. The advantage of doing this is that even if we change the implementation of the homomorphic encryption, these designs still work without any modification.

However, we also observe that the recently proposed homomorphic encryption schemes are usually based on bit-wise operations, a feature that is highly similar to the underlying scheme of bloom filters. This similarity reveals an opportunity for cross-layer optimization. Specifically, a large number of homomorphic encryption schemes encrypt data in a bit-wise

Algorithm 2 An Improved Design of the Homomorphic Bloom Filter

Alice: Bloom Filter Encryption
$a \leftarrow$ the indices where bits are flipped to 1s in BF_A
for $i = 1 \rightarrow t$ **do**
  $E(-a_i) \leftarrow$ encrypt $-a_i$ using PK
end for
send $E(-a)$ to the aggregation server

Bob: Query Encryption
$b \leftarrow$ the indices where bits are flipped to 1s in BF_B
for $j = 1 \rightarrow k$ **do**
  $E(b_j) \leftarrow$ encrypt b_j using PK
end for
send $E(b)$ to the aggregation server

Sever: Evaluation
for $j = 1 \rightarrow k$ **do**
  for $i = 1 \rightarrow t$ **do**
    $cipher\_result_{ij} \leftarrow (E(b_j) + E(-a_i)) * z$
  end for
end for
send $cipher\_result$ to Alice

Alice: Result Decryption
for $j = 1 \rightarrow k$ **do**
  $flag \leftarrow FALSE$
  for $i = 1 \rightarrow t$ **do**
    $plain\_result_{ij} \leftarrow$ decrypt $cipher\_result_{ij}$ using SK
    if $plain\_result_{ij} = 0$ **then**
      $flag \leftarrow TRUE$
      break
    end if
  end for
  if $flag = FALSE$ **then**
    return no intersection
  end if
end for
return intersection exists

manner, i.e., each bit in the plaintext is encrypted as a separate ciphertext. Later, the computation is represented as a boolean circuit with XOR and AND gates, where the input is the ciphertext for each encrypted bit. As this mechanism breaks down an arbitrary computation into bit operations, it may lead to highly complex circuits. However, in our case, after transforming the original query operation that involves several arithmetic operations into the query operation based on bloom filter, the query operation can be easily decomposed to a single bit-wise operation.

We represent the bloom filter as a bit array, which can in turn be represented as an array of integers. We then use the integer array as encryption input, and are able to naturally obtain the encrypted bloom filter without using the polynomial methods as developed in the previous subsections. Specifically, we re-design the query operation as follows. Suppose Alice constructs a bloom filter BF_A with all her location dataset. Bob also constructs a bloom filter BF_B with only the location he wants to query. To check whether the location queried by Bob exists in Alice's location set, we only need to check if all bits flipped as 1s in BF_B are also set as 1s in BF_A, which can be decomposed into two bit-wise operations. First, perform a bit-wise AND operation between the two bloom filters to get a bit array containing wanted bits in BF_A. Then, perform a bit-wise XOR operation between BF_B and the resulting bit array from the last step to check whether all wanted bits are set as 1. Therefore, the query operation can be expressed as:

$$query\_result = BF_A \& BF_B \oplus BF_B \tag{11}$$

If the queried location of Bob exists in Alice's location set, the $query\_result$ in Equation 11 will be zero. (Equation 11 will fail when both bloom filters are empty, i.e., both Alice and Bob have no trajectory, but this can be easily prevented.) Furthermore, the operation can be simplified as:

$$query\_result = \neg BF_A \& BF_B \tag{12}$$

Note that the bit-wise NOT operation only requires one input, thus can be performed locally before the encryption occurs. This can further reduce the required operation overhead under encryption form.

With Equation 5, we can calculate the optimal m given a certain number of elements n and required false positive level p. We use l to represent the length of an integer. Then the system needs to encrypt $\lceil m/l \rceil$ integers. During the query, the system needs to calculate encrypted AND operation for $\lceil m/l \rceil * l$ bit. In the last step, the system needs to decrypt $\lceil m/l \rceil$ integers to get the result. Before the computation, both the Alice and Bob need to send their bloom filter to the server which is $\lceil m/l \rceil$ integers. After the computation has been done, $\lceil m/l \rceil$ integers need to be sent back as the result.

IV. Security Analysis

As the goal of our system is to provide secure and private protocols for users to share location data without leaking sensitive information, we now analyze the security of this system in the presence of *semi-honest adversaries* in the perspectives of Alice, Bob, and the aggregation server.

A. Alice's Privacy and Actions

Observe that in our protocol designs, as long as Alice correctly encrypts her data with the public key, her security and privacy is well protected. Therefore, users will not be discouraged from adopting this service and publishing their location data due to privacy concerns.

On the other hand, Alice is able to decrypt the results received from the aggregation server only when there are intersections. In optimizations O1 and O3, Alice only gets random integers unless it is zero, indicating that there exists an intersection. In optimization O2, we introduced intentional randomness during the evaluation to prevent Alice from identifying Bob's position.

B. Bob's Privacy and Actions

As discussed in Section IV-A, all of the optimizations O1 to O3 reveal no Bob's location data to Alice. Next, we consider whether Bob's location data is leaked to the aggregation server. In optimizations O2 and O3, Bob submits his query to the server in an encrypted way, which guarantees the server has no access to Bob's data. In optimization O1, the server may only obtain Bob's positions of the bits flipped as 1 but not his location data in plaintext. Strong cryptographic but

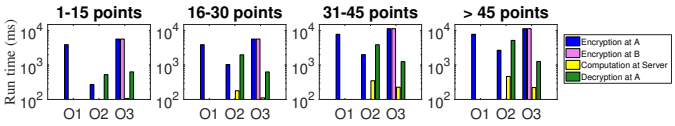

Fig. 3. Computation overhead

computationally intensive hash functions, such as password-based key derivation functions, can make the server's brute-force attacks on Bob's location data less effective.

C. The Aggregation Server's Actions

We assume that the messages between Alice, Bob, and the server are encrypted, so the communication channel can be secure. For optimization O1, the server's potential brute-force attack on Bob has been presented in Section IV-B. For optimizations O2 and O3, even if the server is compromised and is able to read and write all messages, information is not leaked to the server due to the very nature of the homomorphic encryption: all computations are based on encrypted data, not plain text.

On the other hand, the server is indeed able to log the IP addresses and user accounts of the users. But we consider this not a security problem as this is the standard operations of social networks and apps. Finally, the server may sabotage the system by refusing to act as the intermediate server. In practice, this does not often happen as servers lack the motivations to block users arbitrarily.

V. Evaluation

In this section, we report experimental results of our homomorphic bloom filter protocols based on a real-world smartphone mobile dataset. First, we briefly describe the dataset and the experimental setup. Then, the parameter settings of the protocols are presented. Finally, we compare the computation and communication overhead of different protocols.

A. Datasets and Experimental Setup

Our dataset, provided by a telecommunication carrier in China, consists of thousands of cellular data access records in a small city. More specifically, this dataset includes 7607 unique users, 121 cell towers, and was collected by from 6:00 pm to 9:00 pm on a Sunday in 2014. Each mobile data access record contains the coordinate of the cell tower with which the user communicates. A sequence of records of a user can be viewed as her history trajectory. We randomly pick users and generate queries for evaluation. All experiments are performed on a PC with a 3.10GHz Xeon E3-1220 processor, and 32GB memory, running Red Hat 4.8.5.

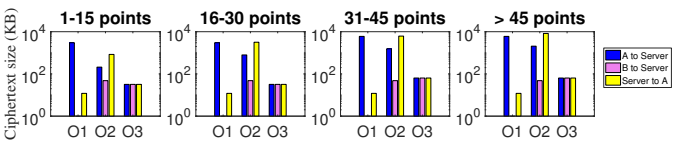

Fig. 4. Communication overhead

B. Parameter Settings

We group the users in our dataset into 4 categories, by the number of unique towers they visited: group 1 (1-15 points), group 2 (16-30 points), group 3 (31-45 points), group 4 (above 45 points). The default false positive probability is set as 0.1. According to Equation 4 and Equation 5, we use a 250-bit bloom filter with 4 hash function for group 1 & 2, a 500-bit bloom filter with 4 hash function for group 3 & 4.

We implement the optimization O1 and O2 on an integer encryption system (SEAL) [22], and the optimization O3 on a binary encryption system based on [18]. In SEAL, we set polynomial modulus as "$1x\hat{}1024+1$", coefficient modulus as $FFFFFFF00001$ and plaintext modulus as 256. Under this setting, we calculate the size of a freshly encrypted integer in SEAL as $1025 * 48 * 2 = 98400$ bits, where 1025 is the number of coefficients in one polynomial, 48 is the number of bits per coefficient occupies, and 2 is the size of the array of polynomials. To minimize the difference of the size of the ciphertext, for the binary encryption system, the security parameter λ is set as 4, and each bit in the plaintext is encrypted into a 1024-bit wide ciphertext.

C. Computation Overhead

The computation overhead can be decomposed into three parts: the encryption time, the query computation time and decryption time, where the encryption happens at both Alice and Bob. We randomly perform 20 queries for each group of users and compare the computation overhead of all three optimizations in Figure 3.

As we can see, in optimization O1, the encryption overhead at Alice dominates the overall computation overhead, as Alice requires to encrypt every bit in her bloom filter BF_A. In optimization O2, the encryption overhead at Alice is much larger than encryption overhead at Bob, because the number of integers to be encrypted is determined by the number of flipped 1s in the bloom filter. This also explains why the encryption overhead increases as the number of location points increase. Since optimization O2 incurs large intermediate results to be transmitted back to Alice, the decryption overhead (almost 5 seconds) is high. The computation overhead of optimization O3 can reach 24 seconds in total (11 seconds for encryption), which is the highest among all three protocols. We think it is mainly caused by the setting of the security parameter λ in our evaluation. The binary homomorphic encryption library we used requires λ to be set as a power of two. To keep the size of ciphertext generated by this library and the SEAL on the same scale, the λ is set as 4 in our experiments. If we minimize the λ to 2, both the computation overhead and the ciphertext size decline exponentially.

D. Communication Overhead

The communication overhead mainly consists of three parts: the ciphertext submitted to the server by Alice, the ciphertext submitted to the server by Bob, and the computation results sent from the server to Alice. We use the size of the ciphertext to be transmitted as the main measure for communication

overhead. We compare the communication overhead of all three optimizations in Figure 4.

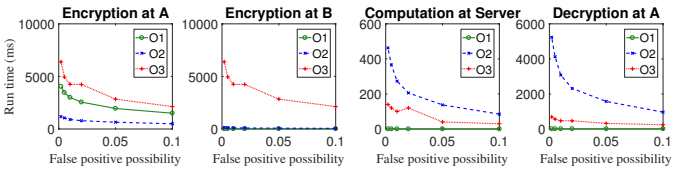

Fig. 5. Accuracy vs computation overhead

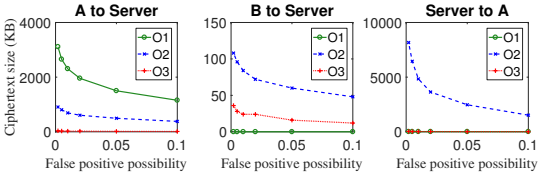

Fig. 6. Accuracy vs communication overhead

As we can see that in optimization O1, most of the overhead happens at sending ciphertext from Alice to the server, as each bit in the bloom filter is treated as an integer. Thus, optimizations O2 and O3 beat optimization O1 regarding the communication overhead generated by the data transmission from Alice to the server. For optimization O2, besides the heavy communication overhead at transmitting ciphertext from Alice to the server, a large number of encrypted intermediate results sent from the server to Alice also cause a high communication overhead. Optimization O3, which adopts a bit-wise encryption scheme, has the lowest communication overhead with only tens of KB in size. All three parts of communication overhead for optimization O3 keep the equal, as the bloom filters of Alice and Bob are always treated as integer arrays with the same size in the bit-wise computations.

E. Protocol Accuracy

As the bloom filter has possible false positives, we take the users in group 3 (31-45 points) as an example to demonstrate the computation overhead in Figure 5 and communication overhead in Figure 6 given different false positive possibility p. We set the corresponding number of hash functions k and the size of bloom filter m based on Equation 4 and Equation 5.

For optimization O1, as the encryption at Alice and transmission of data from Alice to server dominate the computation overhead and communication overhead respectively, they both decrease as p increases, whereas other overheads are quite stable. For optimization O2, both computation overhead and communication overhead decrease as p increases with the exception that the encryption overhead at Bob keeps relatively stable. Similar to optimizations O1 and O2, optimization O3 also shows a trend of decreasing overhead as p increases.

VI. Conclusion

In this paper, we investigate the problem on how to keep users' location data secure, while at the same time allowing servers to perform basic operations on the data. We demonstrate secure computation primitives on location data where the servers should have no knowledge of the plaintext, i.e., the servers should not even keep the private keys to decrypt data. In this way, compromised servers will not cause leaked user data. We have proposed a generalized secure set membership check framework based on the recently developed homomorphic encryption and an advanced data structure called the bloom filter. We also use real-world datasets to evaluate the proposed practical prototypes, which are implemented on the open-source homomorphic libraries. The preliminary results demonstrate the feasibility and efficiency of the proposed approach as well as the security of the protocol designs.

Acknowledgements

We thank the anonymous reviewers for their insightful comments. The work reported in this paper was supported in part by the grant USDA NIFA 2017-67007-26150.

References

[1] EPIC, "Location privacy issues and suggestions." https://epic.org/privacy/location_privacy/.

[2] R. L. Krutz and R. D. Vines, *Cloud security: A comprehensive guide to secure cloud computing*. Wiley Publishing, 2010.

[3] Z. Lu, Y. Feng, W. Zhou, X. Li, and Q. Cao, "Inferring correlation between user mobility and app usage in massive coarse-grained data traces," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 1, no. 4, p. 153, 2018.

[4] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, STOC '09, (New York, NY, USA), pp. 169–178, ACM, 2009.

[5] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption.," *IACR Cryptology ePrint Archive*, vol. 2012, p. 144, 2012.

[6] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[7] F. Casino, J. Domingo-Ferrer, C. Patsakis, D. Puig, and A. Solanas, "A k-anonymous approach to privacy preserving collaborative filtering," *Journal of Computer and System Sciences*, vol. 81, no. 6, pp. 1000–1011, 2015.

[8] D. E. Seidl, G. Paulus, P. Jankowski, and M. Regenfelder, "Spatial obfuscation methods for privacy protection of household-level data," *Applied Geography*, vol. 63, pp. 253–263, 2015.

[9] D. Liu, E. Bertino, and X. Yi, "Privacy of outsourced k-means clustering," in *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pp. 123–134, ACM, 2014.

[10] Y. Li, K. Gai, L. Qiu, M. Qiu, and H. Zhao, "Intelligent cryptography approach for secure distributed big data storage in cloud computing," *Information Sciences*, vol. 387, pp. 103–115, 2017.

[11] P. Hallgren, M. Ochoa, and A. Sabelfeld, "Innercircle: A parallelizable decentralized privacy-preserving location proximity protocol," in *Privacy, Security and Trust (PST), 2015 13th Annual Conference on*, pp. 1–6, IEEE, 2015.

[12] I. Bilogrevic, M. Jadliwala, V. Joneja, K. Kalkan, J.-P. Hubaux, and I. Aad, "Privacy-preserving optimal meeting location determination on mobile devices," *IEEE transactions on information forensics and security*, vol. 9, no. 7, pp. 1141–1156, 2014.

[13] C. Dong, L. Chen, and Z. Wen, "When private set intersection meets big data: an efficient and scalable protocol," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 789–800, ACM, 2013.

[14] Y. Zheng, M. Li, W. Lou, and Y. T. Hou, "Sharp: Private proximity test and secure handshake with cheat-proof location tags," in *European Symposium on Research in Computer Security*, pp. 361–378, Springer, 2012.

[15] F. Kerschbaum, "Outsourced private set intersection using homomorphic encryption," in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, pp. 85–86, ACM, 2012.

[16] T. Sander, A. Young, and M. Yung, "Non-interactive cryptocomputing for nc/sup 1," in *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pp. 554–566, IEEE, 1999.

[17] F. Armknecht, C. Boyd, C. Carr, K. Gjøsteen, A. Jäschke, C. A. Reuter, and M. Strand, "A guide to fully homomorphic encryption.," *IACR Cryptology ePrint Archive*, vol. 2015, p. 1192, 2015.

[18] M. Van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 24–43, Springer, 2010.

[19] Z. Brakerski and V. Vaikuntanathan, "Lattice-based fhe as secure as pke," in *Proceedings of the 5th conference on Innovations in theoretical computer science*, pp. 1–12, ACM, 2014.

[20] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pp. 309–325, ACM, 2012.

[21] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *Advances in Cryptology–CRYPTO 2013*, pp. 75–92, Springer, 2013.

[22] K. Laine and R. Player, "Simple encrypted arithmetic library-seal (v2. 0)," tech. rep., Technical report, September, 2016.