FastGeo: Efficient Geometric Range Queries on Encrypted Spatial Data

Boyang Wang, Ming Li, *Member, IEEE,* and Li Xiong, *Member, IEEE,*

Abstract—Spatial data have wide applications, e.g., location-based services, and *geometric range queries* (i.e., finding points inside geometric areas, e.g., circles or polygons) are one of the fundamental search functions over spatial data. The rising demand of outsourcing data is moving large-scale datasets, including large-scale spatial datasets, to public clouds. Meanwhile, due to the concern of insider attackers and hackers on public clouds, the privacy of spatial datasets should be cautiously preserved while querying them at the server side, especially for location-based and medical usage. In this paper, we formalize the concept of *Geometrically Searchable Encryption*, and propose an efficient scheme, named *FastGeo*, to protect the privacy of clients' spatial datasets stored and queried at a public server. With FastGeo, which is a novel two-level search for encrypted spatial data, an honest-but-curious server can efficiently perform geometric range queries, and correctly return data points that are inside a geometric range to a client without learning sensitive data points or this private query. FastGeo supports arbitrary geometric areas, achieves sublinear search time, and enables dynamic updates over encrypted spatial datasets. Our scheme is provably secure, and our experimental results on real-world spatial datasets in cloud platform demonstrate that FastGeo can boost search time over 100 times.

Keywords—Spatial data, geometric range queries, encrypted data, privacy

1 Introduction

Searchable Encryption (SE) [1] is a promising technique to enable search functionalities over encrypted data at a remote server (e.g., a public cloud) without decryption. Specifically, with SE, a *client* (e.g., a company) can retrieve correct search results from an *honest-but-curious* server without revealing private data or queries. A sequence of SE schemes [1]–[7] have been proposed, where most of them focus on common SQL queries, such as keyword search and range search. Recently, a few SE schemes [8]–[11] have drawn their attentions particularly to geometric range queries over spatial datasets, where a geometric range query retrieves points inside a geometric area, such as a circle or a polygon [12]. However, how to enable arbitrary geometric range queries with sublinear search time while supporting efficient updates over encrypted spatial data remains open.

Spatial data have extensive applications in location-based services, computational geometry, medical imaging, geosciences, etc., and geometric range queries are fundamental search functionalities over spatial datasets. For instance, a client can find friends within a circular area in location-based services (e.g., Facebook); a medical researcher can predict whether there is a dangerous outbreak for a specific virus in a certain geometric area (e.g., Zika in Brazil) by retrieving patients inside this area. Many companies, such as Yelp and Foursquare, are now relying on public clouds (e.g., Amazon Web Services, AWS) to manage their spatial datasets and process queries. However, due to the potential threats of inside attackers and hackers, the privacy of spatial datasets in public clouds should be carefully taken care of, particularly in location-based and medical applications. For instance, a compromise of AWS by an inside attacker or hacker would put millions of Yelp users' sensitive locations under the spotlight.

Different from keyword search relying on equality checking and range search depending on comparisons, a geometric range query over a spatial dataset essentially requires *compute-then-compare operations* [11]. For example, to decide whether a point is inside a circle, we calculate a distance from this point to the center of a circle, and then compare this distance with the radius of this circle; in order to verify whether a point is inside a polygon, we compute the cross product of this point with each vertex of this polygon, and compare each cross product with zero (i.e., positive or negative) [13].

Unfortunately, this requirement of compute-then-compare operations makes the design of a SE scheme supporting geometric range queries more challenging, since current efficient cryptographic primitives are not suitable for the evaluation of compute-then-compare operations in ciphertext. More specifically, Pseudo Random Function (PRF) [14] can only enable equality checking; Order-Preserving Encryption [15] solely supports comparisons; Partially Homomorphic Encryption (e.g., Paillier [16]) can only compute additions (or multiplications). BGN [17] calculates additions and at most one multiplication on encrypted data. On the other hand, Fully

- *Boyang Wang and Ming Li are with the Department of Electrical and Computer Engineering, The University of Arizona, Tucson, AZ 85721. E-mail: {boyangwang, lim}@email.arizona.edu*
- *Li Xiong is with the Department of Mathematics and Computer Science, Emory University, Atlanta, GA 30322. Email: lxiong@emory.edu*

Homomorphic Encryption (FHE) [18] could securely evaluate compute-then-compare operations *in principle*. However, the evaluation with FHE does not reveal search decisions (such as inside or outside) over encrypted data, which limits its usage in search.

In this paper, we formalize the concept of Geometrically Searchable Encryption (GSE), which is evolved from the definitions of SE schemes but focuses on answering geometric queries. We propose a GSE scheme, named FastGeo, which can efficiently retrieve points inside a geometric area without revealing private data points or sensitive geometric range queries to a honest-but-curious server. Instead of directly evaluating compute-then-compare operations, our main idea is to convert spatial data and geometric range queries to a new form, denoted as *equality-vector form*, and leverage a *two-level search* as our key solution to verify whether a point is inside a geometric range, where the first level securely operates equality checking with PRF and the second level privately evaluates inner products with Shen-Shi-Waters encryption (SSW) [19]. The major contributions of this paper are summarized as below:

- With the embedding of a hash table and a set of link lists in our two-level search as a novel structure for spatial data, FastGeo can achieve sublinear search and support arbitrary geometric ranges (e.g., circles and polygons). Compared to recent solutions [8], [11], FastGeo not only provides highly efficient updates over encrypted spatial data, but also improves search performance over 100x.
- We formalize the definition of GSE and its leakage function, and rigorously prove data privacy and query privacy with indistinguishability under selective chosen plaintext attacks (IND-SCPA) [19].
- We implement and evaluate FastGeo in cloud platform (Amazon EC2), and demonstrate that FastGeo is highly efficient over a real-world spatial dataset. For instance, a geometric range query over 49,870 encrypted tuples can be performed within 15 seconds, and an update only requires less than 1 second on average.

2 Related Work

OPE [15] and some SE schemes [20]–[22] that support comparisons, can perform rectangular range queries by applying multiple dimensions. However, those extensions do not work with other geometric range areas, e.g., circles and polygons in general. Wang et. al. [9] proposed a scheme, which particularly retrieves points inside a circle over encrypted data by using a set of concentric circles. Zhu et al. [10] also built a scheme for circular range search over encrypted spatial data. Unfortunately, these two schemes exclusively work for circles, and do not apply to other geometric areas.

Ghinita and Rughinis [8] designed a scheme, which supports geometric range queries by using Hidden Vector Encryption [21]. Instead of encoding a point with a binary vector of T^2 bits, where T is the dimension size, it leverages a hierarchical encoding, which reduces the vector length to $2\log_2 T$ bits. However, its search time is still linear with regard to the number of tuples in a dataset, which not only runs slowly over large-scale datasets but also disables efficient updates.

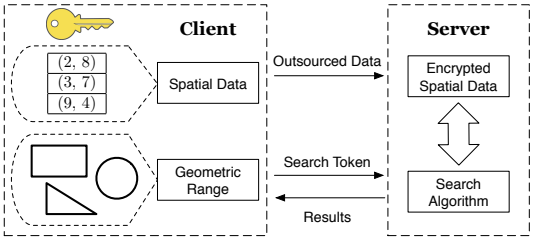


Fig. 1: The system model of a GSE scheme.

Our recent work [11] presents a scheme that can operate arbitrary geometric range queries. It leverages Bloom filters [23] and their properties, where a data point is represented as a Bloom filter, a geometric range query is also formed as a Bloom filter, and the result of an inner product of these two Bloom filters correctly indicates whether a point is inside a geometric area. Its advanced version with R-trees [24] can achieve logarithmic search on average. Although it also utilizes SSW as one of the building blocks, its tree-based index and unique design with Bloom filters are completely different from the novel two-level index introduced in this paper, where these significant differences prevent this previous scheme from supporting efficient updates and practical search time.

Some other works [25]–[28] study secure geometric operations between two parties (e.g., Alice and Bob), where Alice holds a secret point and Bob keeps a private geometric range. With Secure Multi-party Computation (SMC), Alice and Bob can decide whether a point is inside a geometric range without revealing secrets to each other. However, the model of these studies are different from ours (i.e., Alice and Bob both provide individual private inputs, while a client in our model has all the private inputs but the server has no private inputs). Besides, SMC introduces extensive interactions.

3 Problem Statement

System Model. There are two entities, including a client and a server, in our model (in Fig.1). The client is a company or an organization, who stores its spatial datasets on the server. Each tuple in a spatial dataset is essentially a point. In addition, it also wants to perform geometric range queries over its outsourced spatial dataset. The purpose of a geometric range query is to retrieve points inside this geometric range. The server is operated by a cloud service provider, and it offers data storage and query processing services. By leveraging these data services, the client can reduce its local cost.

The server is honest-but-curious, where it provides data services but it is curious and trying to reveal the client's spatial data (i.e., what points are stored) or

geometric range queries (i.e., what queries are searched). As a result, the client encrypts its spatial datasets and geometric range queries before handling them to the server. Only the client itself has the secret key for encryption/decryption. Meanwhile, the server is required to correctly perform geometric range search on encrypted spatial data without decryption, and it should return search results (i.e., the ciphertexts of points that are inside a geometric range query) to the client.

***Definition** 1:* **Definition of GSE.** *A geometrically searchable encryption includes five polynomial-time algorithms* $\Pi = \{\mathsf{GenKey}, \mathsf{BuildIndex}, \mathsf{Enc}, \mathsf{GenToken}, \mathsf{Query}\}$ *such that:*

- $sk \quad \mathsf{GenKey}(1^\lambda)$*: is a probabilistic algorithm that is run by a client. It takes a public security parameter* λ *as input, and outputs a secret key* sk*.*
- $\Gamma \leftarrow \mathsf{BuildIndex}(\mathbf{D}, m)$*: is a deterministic algorithm that is run by a client. It takes a spatial dataset* $\mathbf{D} = \{D_1, ..., D_n\}$ *and a public parameter* m *as input, and outputs an index* Γ*.*
- $\Gamma^* \leftarrow \mathsf{Enc}(\Gamma, sk)$*: is a probabilistic algorithm that is run by a client. It takes an index* Γ *and a secret key* sk *as input, and outputs an encrypted index* Γ^**.*
- $tk_Q \leftarrow \mathsf{GenToken}(Q, sk, m)$*: is a probabilistic algorithm that is run by a client. It takes a geometric range query* Q*, a secret key* sk *and a public parameter* m *as input, and outputs a search token* tk_Q*.*
- $\mathbf{I}_Q \leftarrow \mathsf{Query}(\Gamma^*, tk_Q)$*: is a deterministic algorithm that is run by a server. It takes an encrypted index* Γ^* *and a search token* tk_Q*, and outputs a set of identifiers* $\mathbf{I}_Q$*, where if* $D_i \in Q$*, then* $I_i \in \mathbf{I}_Q$*, for* $i \in [1, n]$*.*

In the above definition, each tuple D_i, for $i \in [1, n]$, in a spatial dataset $\mathbf{D}$ is a point of a data space Δ_T^α, where α denotes the number of dimensions and T is the size of each dimension. We assume each dimension has the same size, and its range is from $[0, T-1]$. Without loss of generality, we also assume $\alpha = 2$, then each point can be described as $D_i = (d_{i,1}, d_{i,2})$, where $d_{i,1}$ and $d_{i,2}$ are the values of this point in x-dimension and y-dimension respectively, and $d_{i,1}, d_{i,2} \in [0, T-1]$. A geometric range query Q is a range within the data space, which can be represented as $Q \subseteq \Delta_T^\alpha$. If a point D_i is inside a query Q, we denote it as $D_i \in Q$; otherwise, we have $D_i \notin Q$.

This definition is a *symmetric-key* GSE, since the encryption of an index and the generation of a search token both use the same secret key. The objective of a GSE scheme is to build an encrypted index of a spatial dataset in order to enable search functionalities in ciphertext, and eventually output a set of identifiers indicating encrypted points that associate to a query. With those identifiers, the server can return corresponding encrypted points, and the client can learn search result in plaintext by decrypting encrypted points locally. The encryption and decryption of each tuple itself can be delivered by another additional layer of standard CPA-secure encryption (e.g., AES-CBC-256), which can be ignored in a searchable encryption. The correctness of a GSE scheme can be formally shown as below:

***Definition** 2:* **Correctness of GSE.** *We say a GSE scheme* Π *is correct if for all* $\lambda \in \mathbb{N}$*, all* sk *output by* $\mathsf{GenKey}(1^\lambda)$*, all* $D_i \in \Delta_T^\alpha$*, all* Γ *output by* $\mathsf{BuildIndex}(\mathbf{D}, \sigma)$*, all* Γ^* *output by* $\mathsf{Enc}(\Gamma, sk)$*, all* $Q \subseteq \Delta_T^\alpha$*, and all* tk_Q *output by* $\mathsf{GenToken}(Q, sk, \sigma)$*, we have*

- *If* $D_i \in Q$*:* $\mathsf{Query}(\Gamma^*, tk_Q) = \mathbf{I}_Q$*, where* $I_i \in \mathbf{I}_Q$*;*
- *If* $D_i \notin Q$*:* $\Pr[\mathsf{Query}(\Gamma^*, tk_Q) = \mathbf{I}_Q$*, where* $I_i \notin \mathbf{I}_Q] \geq 1 - \mathtt{negl}(\lambda)$

where $\mathtt{negl}(\lambda)$ *is a negligible function [14] in terms of* λ*.*

This correctness implies that the scheme will definitely return the identifier of the ciphertext of a tuple D_i, if this point is inside a query Q; and will return the identifier of the ciphertext of a tuple D_i with a negligible probability (i.e., a probability that is significantly small and can be ingored in practice), if this point is outside a query Q.

Security Objectives. The security of GSE is to prevent the server from learning spatial data and geometric queries, while enabling geometric range search. Specifically, the server should not learn the content of data points, or reveal the vertices of a polygon or the center and radius of a circle that the client is searching for.

On the other hand, the server is allowed to obtain certain information about a spatial dataset in order to operate queries efficiently and functionally. For instance, the server learns the total number of tuples in a spatial dataset (i.e., *size pattern*); it reveals which identifiers are touched or retrieved by a search token (i.e., *access pattern*). Those leakage can be formally included in a *leakage function* $\mathcal{L}$ [2]. We will first prove our security under selective chosen plaintext attacks with this leakage function, and then further analyze our privacy by considering an even stronger attacker who can carry statistic attacks. Details will be described in Sec. 6.

4 PRELIMINARIES

Pseudo Random Function (PRF). A PRF [14] generates pseudo random outputs that are computationally indistinguishable from random outputs.

***Definition** 3:* *Let* $F : \{0,1\}^n \times \{0,1\}^n \rightarrow \{0,1\}^n$ *be an efficient keyed function. We say* F *is a pseudo random function if for all probabilistic polynomial-time adversary* $\mathcal{A}$*, there exists a negligible function* $\mathtt{negl}$ *such that:*

$$\left|\Pr[\mathcal{A}^{F_k(\cdot)}(1^n) = 1] - \Pr[\mathcal{A}^{f_n(\cdot)}(1^n) = 1]\right| \leq \mathtt{negl}(n),$$

where key $k \leftarrow \{0,1\}^n$ *is chosen uniformly at random and* f_n *is chosen uniformly at random from the set of functions mapping* n*-bit strings to* n*-bit strings.*

By leveraging PRF, we can perform equality checking on encrypted data. Specifically, given two messages m_0 and m_1, their outputs, $[m_0]$ and $[m_1]$, are the same, if and only if $m_0 = m_1$. We use Init to denote the algorithm to initialize a random key in PRF, and leverage $\mathsf{GetBits}$ to present the algorithm to get pseudo random outputs.

Shen-Shi-Waters Encryption (SSW). SSW [19] can evaluate whether the inner product of two vectors is zero without leaking privacy. Concretely, given two vectors

$\vec{u} = (u_1, ..., u_m)$ and $\vec{v} = (v_1, ..., v_m)$, SSW generates a ciphertext $[\vec{u}]$ with vector $\vec{u}$ and a token $[\vec{v}]$ with vector $\vec{v}$. The evaluation on $[\vec{u}]$ and $[\vec{v}]$ indicates whether the inner product of $\vec{u}$ and $\vec{v}$ is zero as

$$\begin{cases} \text{if } < \vec{u}, \vec{v} >= 0, \quad \mathsf{SSW.Query}([\vec{u}], [\vec{v}]) = 1 \\ \text{otherwise}, \quad \Pr[\mathsf{SSW.Query}([\vec{u}], [\vec{v}]) = 0] \geq 1 - \mathtt{negl}(\lambda) \end{cases}$$

without revealing vector $\vec{u}$ nor $\vec{v}$, where $< \vec{u}, \vec{v} >= \sum_{i=1}^{m} u_i \cdot v_i$ is the inner product of two vectors. Its security can be proved with indistinguishability under Selective Chosen-Plaintext Attacks (IND-SCPA) [19], and the algorithms of SSW are briefly presented as below

- $\mathsf{Setup}(1^\lambda)$: Given a security parameter λ, output a secret key sk.
- $\mathsf{Enc}(sk, \vec{u})$: Given sk and a vector $\vec{u}$, where $\vec{u} = (u_1, ..., u_m)$, output a ciphertext $[\vec{u}]$.
- $\mathsf{GenToken}(sk, \vec{v})$: Given sk and a vector $\vec{v}$, where $\vec{v} = (v_1, ..., v_m)$, output a token $[\vec{v}]$.
- $\mathsf{Query}([\vec{u}], [\vec{v}])$: Given ciphertext $[\vec{u}]$ and token $[\vec{v}]$, output 1 if $< \vec{u}, \vec{v} >= 0$ and output 0 otherwise.

The encryption time and token generation time are both $O(m)$, and the size of a ciphertext and the size of a token are also both $O(m)$, where m is the vector length.

5 FastGeo: An Efficient GSE

Instead of directly running compute-then-compare operations, a straightforward design would be splitting compute-then-compare operations into two steps, where the server computes (e.g., computes a distance) over encrypted data with BGN [17], and then a client locally operates comparisons in plaintext. However, the huge amount of local decryptions at the client side would significantly limit search performance. Besides, the communication overhead is equivalent to downloading the entire dataset, which is obviously not practical.

Overview of FastGeo Design. To overcome those limitations, we transform geometric range queries and data points to a different form, denoted as equality-vector form, such that performing equality checking and evaluating inner products together can correctly and efficiently answer arbitrary geometric range queries. In order to achieve efficient evaluations while still maintaining strong privacy guarantee, we utilize PRF to securely enable equality checking, and we leverage SSW to privately verify whether inner products are zeros. In addition, a dictionary (implemented as a hash table) is combined with the use of PRF to achieve sublinear search. From the high level, our design can be interpreted as a two-level search, where the first level relies on equality checking and the second level depends on evaluating inner products. Moreover, link lists are applied to the second level to support efficient updates.

5.1 Data and Queries in Equality-Vector Form

We use a set of examples to show how to transform data and queries to equality-vector form (in plaintext),

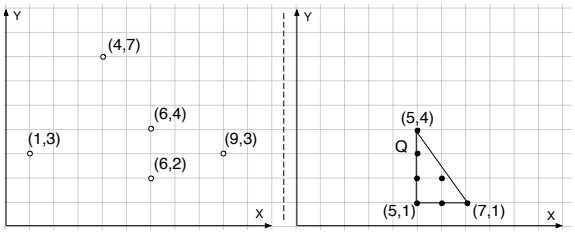


Fig. 2: A spatial dataset and a triangular range query.

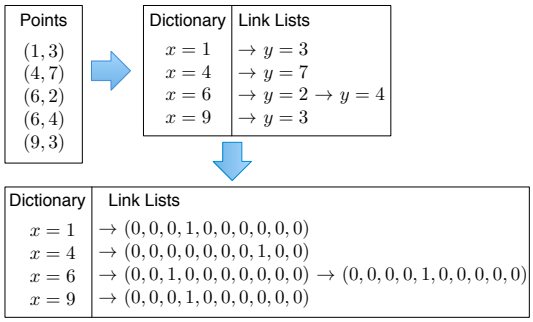


Fig. 3: A dataset before & after converting the form.

such that compute-then-compare operations can be replaced with a combination of equality checking and inner products. Meanwhile, we also demonstrate how to embed a dictionary and a set of link lists to index a spatial dataset. As shown in Fig. 2, we have a spatial dataset with 5 data points and a triangular range query Q. We assume the data space in this example is Δ_{10}^2, i.e., two dimensions and each has a size of $T = 10$, where $x \in [0, 9]$ and $y \in [0, 9]$. Only integers are considered in this example. Without loss of generality, a triangle is leveraged here as an example. Other geometric objects, such as circles and rectangles, are also compatible.

Based on the distinct values of these data points in x-dimension, we first build a dictionary (as illustrated in Fig. 3), where each *element* of this dictionary contains a distinct x-value. For each element, we also create a link list to represent its corresponding y-values, where each *node* in the link list stores a y-value. Obviously, the size of each link list depends on the number of data points for a given x-value. For instance, given $x = 6$, there are two data points $(6, 2)$ and $(6, 4)$, thus, the size of its link list is 2. In addition, a permutation function will also be applied to ensure nodes in each list are in random order. Then, we represent every y-value in each link list as a vector, where the length of each vector is 10 (since $T = 10$). Specifically, if $y = i$, then the component at index i in the vector is set as 1 while others are all assigned 0s. The index of a component starts from 0 and ends at 9. For example, if $y = 3$, its vector is $(0, 0, 0, 1, 0, 0, 0, 0, 0, 0)$, where only the component at index 3 is 1.

Given a geometric range query Q in Fig. 2, we first enumerate a set of all the possible points (of the data space) that are inside this geometric range query, and use this set to represent a geometric range query. Note that this enumeration step is trivial for any geometric

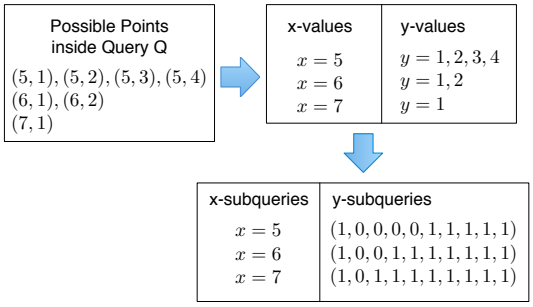


Fig. 4: A query before & after converting the form.

$x = 5$ false
$x = 6$ true
$< (0, 0, 1, 0, 0, 0, 0, 0, 0, 0), (1, 0, 0, 1, 1, 1, 1, 1, 1, 1) >= 0$
$< (0, 0, 0, 0, 1, 0, 0, 0, 0, 0), (1, 0, 0, 1, 1, 1, 1, 1, 1, 1) >= 1 \neq 0$
$x = 7$ false

Fig. 5: An example of search in the equality-form form.

range in plaintext [12]. Next, we generate a set of x-subqueries based on the x-values of these possible points, and compute a y-subquery for each given x-subquery to cover all these possible points (in Fig. 4). Specifically, each x-subquery is still based on a distinct x-value, but we use a vector to represent each y-subquery, where the component at index i in the vector is set as 0 if $y = i$ is covered by this geometric range query for a given x. For instance, when $x = 6$, $y \in [1, 2]$ is covered by Q, therefore, its corresponding y-subquery in the vector format is $(1, 0, 0, 1, 1, 1, 1, 1, 1, 1)$, where components from index 1 to index 2 are set as 0s and others are 1s.

5.2 Search with Equality-Vector Form

With this equality-vector form, we can still operate geometric range queries correctly. Specifically, given query Q, we first search each x-subquery in the dictionary. Once we find a match in terms of x, we continue to evaluate an inner product of its y-subquery with a node in a corresponding link list. If this inner product is 0, it indicates a data point is inside the geometric range query. For instance, as shown in Fig. 5, $x = 5$ and $x = 7$ as x-subqueries are not inside the dictionary via equality checking; once we find a match in the dictionary using $x = 6$, we continue to use its y-subquery to evaluate an inner product with every node in the link list of $x = 6$. Essentially, we can also think our search process as a *sweeping* algorithm, where it sweeps all the associated x-values (i.e., vertical lines) one after another. Sweeping algorithms are commonly used in computational geometry over spatial data for various problems [12].

5.3 Apply Encryption Primitives

With this search in equality-vector form, we can apply PRF to the equality part (i.e., first level) and leverage SSW to the vector part (i.e., second level) of data and queries to protect privacy while enabling geometric

- DataToEnhVector(x, y, m): Given a first-level value x, a second-level value y and vector length m, initialize a vector $\vec{u}_e = (u_0, ..., u_m)$. For $0 \leq i \leq m-1$, if $i = y$, $u_i = H(x)$; otherwise, $u_i = 0$, where $H(\cdot)$ is collision-resistant. Set $u_m = -1$, and return $\vec{u}_e$.
- QueryToEnhVector(x, r, m): Given a first-level value x, a second-level subquery r and vector length m, initialize a vector $\vec{v}_e = (v_0, ..., v_m)$. For $0 \leq i \leq m-1$, if $i \in r$, $v_i = 1$; otherwise, $v_i = 0$. Set $v_m = H(x)$, where $H(\cdot)$ is collision-resistant, and return $\vec{v}_e$.

Fig. 6: Two algorithms generating enhanced vectors

range search. Moreover, a permutation function [14] with fresh randoms should be implemented to each query, such that the sweeping of the encrypted first-level values in each query will follow a random order.

The use of PRF, SSW and a permutation function with equality-vector form can fulfill search functionality and initial security, but another significant issue should still be solved. Specifically, the generation of a x-subquery and its corresponding y-subquery are independent, where a curious server could freely change the search ability of a given query to another one by mismatching different encrypted x-subqueries with different encrypted y-subqueries (i.e., using an encrypted y-subquery $[\vec{v}_i]$ for a different encrypted x-subquery $[x_j]$, where $i \neq j$). For instance, given a search token $tk_Q = \{\{[x_1], [\vec{v}_1]\}, \{[x_2], [\vec{v}_2]\}\}$ of a geometric range query Q, the server could mismatch and query it as

$$tk_{Q'} = \{\{[x_1], [\vec{v}_2]\}, \{[x_2], [\vec{v}_1]\}\},$$

which may leak different access patterns (e.g., different encrypted data points may be retrieved compared to the use of the original search token tk_Q). This type of issues is also referred to as *token collusion* in some previous SE schemes [19], [20], [29].

To prevent token collusion, we propose an enhanced vector form, where an encrypted y-subquery can only be correctly evaluated with its encrypted x-subquery but not others. The idea of this enhancement is to embed a first-level value to the vector form of a second-level value. The algorithms to generate enhanced vectors are described in Fig. 6. For instance, given a data point $(6, 2)$, its y-value in the enhanced vector form is

$$\vec{u}_e = (0, 0, H(6), 0, 0, 0, 0, 0, 0, 0, -1)$$

Correspondingly, given an x-subquery $x = 6$ and its y-subquery $y \in [1, 2]$, its y-subquery is

$$\vec{v}_e = (0, 1, 1, 0, 0, 0, 0, 0, 0, 0, H(6))$$

It is obvious to see that the inner product of the two enhanced vectors $\vec{u}_e$ and $\vec{v}_e$ is zero *if and only if* the inner product of the two original vectors $\vec{u}$ and $\vec{v}$ is zero.

5.4 Details of FastGeo

Finally, with PRF, SSW and the enhanced vector form, we build FastGeo, where the details of each algorithm

$\diamond$ GenKey(1^λ): Given λ, the client computes $sk_p \leftarrow$ PRF.Init(1^λ), $sk_s \leftarrow$ SSW.Setup(1^λ), and outputs $sk = \{sk_p, sk_s\}$.

$\diamond$ BuildIndex($\mathbf{D}, m$): Given $\mathbf{D} = \{D_1, ..., D_n\}$ and m, the client initializes Γ as null. For $D_i = (d_{i,1}, d_{i,2})$, $i \in [1, n]$, it computes

- $\vec{u}_i \leftarrow$ DataToEnhVector($d_{i,1}, d_{i,2}, m$);
- If HashTable.Find($\Gamma, d_{i,1}$) is null, calculates $e = d_{i,1}$, $list \leftarrow$ LinkList.Init(), $list \leftarrow$ LinkList.Append($list, \vec{u}_i$), $\Gamma \leftarrow$ HashTable.Insert($\Gamma, \{e, list\}$);
- Else if HashTable.Find($\Gamma, d_{i,1}$) is not null, evaluates $list \leftarrow$ HashTable.Find($\Gamma, d_{i,1}$), $list \leftarrow$ LinkList.Append($list, \vec{u}_i$);

and finally it permutes nodes in each list, and outputs an index Γ.

$\diamond$ Enc(sk, Γ): Given $sk = \{sk_p, sk_s\}$ and Γ, the client initializes Γ^* as null. For $\{e_j, list_j\} \in \Gamma$, where $1 \le j \le w$ and w is the size of index Γ, the client computes

- $[e_j] \leftarrow$ PRF.GetBits(e_j, sk_p) and $list_j^* \leftarrow$ LinkList.Init(),
- For $\vec{u}_{j,k} \in list_j$, where $1 \le k \le list_j$.size(), calculates $[\vec{u}_{j,k}] \leftarrow$ SSW.Enc($\vec{u}_{j,k}, sk_s$), $list_j^* \leftarrow$ LinkList.Append($list_j^*, [\vec{u}_{j,k}]$)
- $\Gamma^* \leftarrow$ HashTable.Insert($\Gamma^*, \{[e_j], list_j^*\}$)

and finally it outputs an encrypted index Γ^*.

$\diamond$ GenToken(sk, Q, m): Given $sk = \{sk_p, sk_s\}$, Q and m, the client computes a set $S_Q \leftarrow$ Enumerate(Q, Δ_T^2), where $S_Q = \{\{e_i, r_i\} | \textit{ for } 1 \le i \le q_1\}$ and q_1 is the size of Q in x-dimension. For each $\{e_i, r_i\} \in S_Q$, it calculates

$$\vec{v}_i \leftarrow \text{QueryToEnhVector}(e_i, r_i, m), [\vec{v}_i] \leftarrow \text{SSW.GenToken}(\vec{v}_i, sk_s), [e_i] \leftarrow \text{PRF.GetBits}(e_i, sk_p), tk_i = \{tk_{p,i}, tk_{s,i}\} = \{[e_i], [\vec{v}_i]\}$$

It outputs $tk_Q = \{tk_i | \textit{ for } 1 \le i \le q_1\}$, permutes it as $tk_Q \leftarrow$ Permute(tk_Q, γ) with a fresh random γ, and finally returns tk_Q.

$\diamond$ Query(Γ^*, tk_Q): Given Γ^* and tk_Q, where $tk_Q = \{tk_i | \textit{ for } 1 \le i \le q_1\}$, the server initializes $\mathbf{I}_Q$ as null. For each $tk_i = \{tk_{p,i}, tk_{s,i}\}$, it evaluates HashTable.Find($\Gamma^*, tk_{p,i}$),

- If HashTable.Find($\Gamma^*, tk_{p,i}$) is not null, computes $list^* \leftarrow$ HashTable.Find($\Gamma^*, tk_{p,i}$), $\texttt{flag}_k \leftarrow$ SSW.Query($[\vec{u}_k], tk_{s,i}$), where $[\vec{u}_k] \in list^*$, for $1 \le k \le list^*$.size(). For each $\texttt{flag}_k$, if it is true, the server computes $\mathbf{I}_Q = \mathbf{I}_Q \cup I_k$, where I_k is the identifier associated with $[\vec{u}_k]$.

and finally the server outputs a set of identifiers $\mathbf{I}_Q$.

Fig. 7: Details of FastGeo.

are presented in Fig. 7. Some standard hash table functions, including HashTable.Find and HashTable.Insert, and standard link list functions, including LinkList.Init and LinkList.Append, are utilized as sub-algorithms in BuildIndex. We skip the details of those standard functions, since they are commonly known.

In GenToken, given a query Q, the client first enumerates all the possible points inside Q from data space Δ_T^2, and represents them as a set $S_Q = \{\{e_i, r_i\} | \textit{ for } 1 \le i \le q_1\}$, where e_i is a distinct x-value in Q, r_i is the range of Q given e_i, and q_1 is the size of this query in x-dimension. For each $\{e_i, r_i\}$, the client computes a sub-token $tk_i = \{tk_{p,i}, tk_{s,i}\}$, where $tk_{p,i}$ (denoted as the first piece of tk_i) will be leveraged for equality checking and $tk_{s,i}$ (referred to as the second piece of tk_i) will be used for inner product evaluation. Sub-algorithm QueryToEnhVector is implemented to convert each r_i to its enhanced vector form. Search token tk_Q is a set of sub-tokens $\{tk_i | \textit{ for } 1 \le i \le q_1\}$. A permutation function Permute, as we mentioned, is leveraged to randomly permute the order of sub-tokens in tk_Q.

Once the server has a search token tk_Q, it takes each sub-token $tk_i = \{tk_{p,i}, tk_{s,i}\}$ to check if there is a match in an encrypted index Γ^* by evaluating HashTable.Find($\Gamma^*, tk_{p,i}$). If the return of HashTable.Find($\Gamma^*, tk_{p,i}$) is not null (i.e., it returns a link list), the server evaluates each node $[\vec{u}_k]$ in this link list with SSW.Query($[\vec{u}_k], tk_{s,i}$). If the return of SSW.Query($[\vec{u}_k], tk_{s,i}$) is true, the server adds a corresponding identifier I_k to set $\mathbf{I}_Q$. Finally, the server outputs $\mathbf{I}_Q$ after evaluating all the sub-tokens.

Correctness. Due to the correctness of a hash table, PRF and SSW, if a point is inside a geometric range, FastGeo returns its corresponding identifier I_i since

$$\Pr[I_i \in \mathbf{I}_Q | D_i \in Q] = 1$$

where $\mathbf{I}_Q \leftarrow$ Query(tk_Q, Γ^*). On the other hand, due to the *negligible* errors introduced by a hash function and the evaluation of SSW, if a point is not inside a geometric range, our scheme will return identifier I_i with

$$\Pr[I_i \in \mathbf{I}_Q | D_i \notin Q] \le \texttt{negl}(\lambda)$$

The detailed analysis is presented in Appendix. This probability is negligible and can be ignored in practice. For instance, when the hash function is SHA-1 (i.e., 160 bits) and security parameter (i.e., key bit length) $\lambda = 80$, this probability is only $\max\{\frac{1}{2^{80}}, \frac{1}{2^{160}}\} \approx 8.27 \times 10^{-25}$.

Arbitrary Geometric Range. Our design permits geometric range queries with arbitrary shapes, e.g., circles, polygons, etc. The reason is that we represent any geometric range as a set of possible points, and the later encryption on this set and evaluation on its encrypted version are independent of the original geometric shape. This property also prevents the server from learning which type of geometric shapes a client is querying.

Sublinear Search. Since a dictionary can be implemented as a hash table, where finding whether there is a match in the first level requires $O(1)$. Evaluating a node in the second level with one instance of SSW.Query

requires $O(m)$, where m is the length of a vector. Therefore, the overall search time of a geometric range query is $O(m\tau)$, where τ is the number of SSW.Query instances the server needs to evaluate. Compared to linearly querying all the n encrypted data points in the equality-vector form with $O(nm)$, τ is clearly sublinear to n.

In terms of encryption time, encrypting a dictionary with PRF requires $O(w)$, where w is the number of elements in the dictionary. There are n nodes in total in all the w link lists, and each node requires $O(m)$ encryption time with SSE.Enc, the time to encrypt all the nodes needs $O(nm)$. Therefore, the total encryption time spent on our index is $O(nm)$, since $O(nm+w) = O(nm)$, where $w \leq n$. The storage cost of our encrypted index is $O(nm)$. Each instance of SSW.GenToken takes $O(m)$ to generate a second piece of a sub-token and each PRF.GetBits takes $O(1)$ to output a first piece of a sub-token, the overall token generation time per query is $O(mq_1)$, where q_1 is the size of a query in the first level.

Optimized Token Generation. One thing we can observe is that the generation of some second pieces in a search token may be unnecessary. Specifically, a second piece of each sub-token will only be applied to a search process if its first piece outputs a positive match. Therefore, if we separate our two-level search into two phases and allow one round of client-server interaction between the two phases, then a client will generate a second piece if and only if the search result of its first level is positive, which can optimize the overall token generation time. The optimized token generation time is reduced to $O(m\zeta + q_1)$, where ζ is the number of positives in the first level and $\zeta \leq q_1$. Compared to the original $O(mq_1)$, this optimization will be extremely effective, especially when $\zeta \ll q_1$.

Efficient Updates. Efficient updates (including insert, modify and delete) are also available in FastGeo, where the update logic is as the same as those operations in a dictionary and link lists in plaintext. It is because FastGeo only encrypts elements in a dictionary and nodes in link lists, but keeps structures unchanged. In other words, the change of one encrypted data point in the structure does not require re-encrypting any other encrypted data points. A geometric range query with a minimal size of one possible point can be used to find the updating position for one update in our two-level index. The running time of one update operation is $O(m\tau')$ (i.e., $O(m\tau')$ to find the updating node, and $O(1)$ to update pointers and a dictionary), where τ' is the size of the link list this update applies to.

6 Security Analysis

6.1 Leakage Function

As we mentioned, a server is allowed to learn some information to facilitate search functionalities over encrypted data. Concretely, given a spatial dataset and a sequence of geometric range queries, besides public information such as security parameter λ, data space Δ_T^α and public parameter m, the following information are granted to a curious server:

- *Size pattern* (ϕ_1): the number of points in a spatial dataset.
- *Structure pattern* (ϕ_2): the dictionary size (i.e., the number of elements) of a spatial dataset; and for each element in the dictionary, its link list size (i.e., the number of nodes).
- *First-level query-size pattern* (ϕ_3): the size of a geometric range query in the first level.
- *First-level search pattern* (ϕ_4): the number of the same first-level values that a geometric range query has related to a previous query.
- *Access pattern* (ϕ_5): which identifiers are *touched*, and which identifiers are *retrieved* for a given query.

By “touched", it means a data point is matched in the first level, but it does not satisfy in the second level for a given query. By “retrieved", it indicates a data point is not only matched in the first level but also satisfied in the second level. There are also identifiers that are neither touched nor retrieved. With access pattern, the server also discovers the size of results (i.e., how many encrypted data points are retrieved) for a given query.

Those above patterns should be rigorously captured by a leakage function $\mathcal{L}$, which is commonly used in the security analysis of a SE scheme. For size pattern, the input of it to $\mathcal{L}$ is an index Γ of a spatial dataset $\mathbf{D}$, and the output is an integer, $\phi_1 = n \leftarrow \mathcal{L}(\Gamma)$, which is the total number of points. For structure pattern, the input is still Γ, and the output can be formulated as a vector, where the length of this vector is the dictionary size w, and the value of each component in this vector represents the size of a link list. Thus, we have

$$\phi_2 = (s_1, s_2, ..., s_w) \leftarrow \mathcal{L}(\Gamma)$$

where $s_i = list_i$.size(), for $1 \leq i \leq w$, and $\sum_{i=1}^{w} s_i = n$.

In terms of first-level query-size pattern, given a geometric range query Q as an input to $\mathcal{L}$, the output can be described as an integer, $\phi_3 = q_1 \leftarrow \mathcal{L}(Q)$, which is the size of Q in the first level (i.e., the number of sub-tokens generated by Q), The input of first-level search pattern to $\mathcal{L}$ includes a geometric range query Q and a set of previous geometric range queries $\mathbf{Q}' = \{Q'_1, ..., Q'_t\}$, where t is the number of previous geometric range queries. The output of it can be described as

$$\phi_4 = (\eta_1, \eta_2, ..., \eta_t) \leftarrow \mathcal{L}(Q, \mathbf{Q}')$$

where η_i indicates the number of same first-level values that Q and Q'_i have. Particularly, if $\eta_i = 0$, it indicates Q does not have any same first-level values with Q'_i, and if $\eta_i = \phi_3 = q_1$, it implies all the first-level values of Q are included in Q'_i, and it is the maximal value that η_i could achieve.

To formalize access pattern for one geometric range query, the input of it to $\mathcal{L}$ includes an index Γ and a query Q, and its output can be represented as a vector, where the length of this vector is the number of data

points n, and the value of each component in this vector is either 0, 1, or 2 (indicating an identifier is *not touched*, *touched*, or *retrieved* respectively). Then, we have

$$\phi_5 = (\beta_1, \beta_2, ..., \beta_n) \leftarrow \mathcal{L}(\Gamma, Q)$$

where $\beta_i \in \{0, 1, 2\}$, for $1 \leq i \leq n$.

6.2 Formal Security Definitions

We define our security definition with a *game-based approach*, which is widely used in FE-based SE schemes [20]–[22]. Our security can be summarized in two aspects, data privacy and query privacy, where each of these two can be rigorously validated with Selective Chosen-Plaintext Attacks (IND-SCPA) [19].

Data Privacy. Our data privacy indicates, by submitting two spatial datasets $\mathbf{D}_0$ and $\mathbf{D}_1$, a computationally-bounded adversary $\mathcal{A}$ can choose a number of ciphertext requests and token requests confined by leakage function $\mathcal{L}$. However, this adversary is computationally infeasible to distinguish the two spatial datasets.

***Definition** 4:* **IND-SCPA Data Privacy.** *Let* $\Pi = \{\mathsf{GenKey}, \mathsf{BuildIndex}, \mathsf{Enc}, \mathsf{GenToken}, \mathsf{Query}\}$ *be a symmetric-key GSE scheme over security parameter* λ*. We define a security game between a challenger* $\mathcal{C}$ *and an adversary* $\mathcal{A}$ *as below:*

***Init**: $\mathcal{A}$ submits two datasets $\mathbf{D}_0$ and $\mathbf{D}_1$ to $\mathcal{C}$, where $\mathbf{D}_0 = (D_{0,1}, ..., D_{0,n})$, $\mathbf{D}_1 = (D_{1,1}, ..., D_{1,n})$, $D_{0,i}, D_{1,i} \in \Delta_T^\alpha$, for $1 \leq i \leq n$. $\mathbf{D}_0$ and $\mathbf{D}_1$ are subject to $\mathcal{L}(\Gamma_0) = \mathcal{L}(\Gamma_1)$, where $\Gamma_0 \quad \mathsf{BuildIndex}(\mathbf{D}_0)$, $\Gamma_1 \leftarrow \mathsf{BuildIndex}(\mathbf{D}_1)$.*

***Setup**: $\mathcal{C}$ runs $\mathsf{GenKey}(1^\lambda)$ to generate a secret key sk.*

***Phase 1**: $\mathcal{A}$ submits a number of requests, where each request is one of the two following types:*

- *Ciphertext Request: On the jth ciphertext request, $\mathcal{A}$ outputs a dataset $\mathbf{D}'_j$, where $\mathbf{D}'_j = (D'_{j,1}, ..., D'_{j,n})$. $\mathcal{C}$ responses with an encrypted index Γ'^*_j, where $\Gamma'^*_j \quad \mathsf{Enc}(\Gamma'_j, sk)$, $\Gamma'_j \quad \mathsf{BuildIndex}(\mathbf{D}'_j)$ and $\mathbf{D}'_j$ is subject to that $\mathbf{D}'_j$ does not have any same first-level values with $\mathbf{D}_0$ nor $\mathbf{D}_1$.*
- *Token Request: On the jth token request, $\mathcal{A}$ outputs a query Q_j, where $Q_j \subseteq \Delta_T^\alpha$. $\mathcal{C}$ responses with a search token $tk_{Q_j} \leftarrow \mathsf{GenToken}(Q_j, sk)$, where Q_j is subject to $\mathcal{L}(\Gamma_0, Q_j) = \mathcal{L}(\Gamma_1, Q_j)$.*

***Challenge**: With $\mathbf{D}_0, \mathbf{D}_1$ selected in **Init**, $\mathcal{C}$ flips a coin $b \in \{0, 1\}$, and returns Γ_b^* to $\mathcal{A}$, where $\Gamma_b^* \leftarrow \mathsf{Enc}(\Gamma_b, sk)$ and $\Gamma_b \leftarrow \mathsf{BuildIndex}(\mathbf{D}_b)$.*

***Phase 2**: $\mathcal{A}$ continues to submit requests, which are subject to the same restrictions in **Phase 1**.*

***Guess**: $\mathcal{A}$ takes a guess b' of b.*

We say that Π is secure against Selective Chosen-Plaintext Attacks on data privacy if for any polynomial time adversary $\mathcal{A}$ in the above game, it has at most negligible advantage

$$\mathbf{Adv}^{\text{Data}}_{\Pi,\mathcal{A}}(1^\lambda) = \left| \Pr[b' = b] - \frac{1}{2} \right| \leq \mathtt{negl}(\lambda) \qquad (1)$$

Query Privacy. The query privacy of our scheme means, by submitting two geometric range queries Q_0 and Q_1, a computationally-bounded adversary $\mathcal{A}$ is able to choose a number of ciphertext requests and token requests confined by leakage function $\mathcal{L}$. However, adversary $\mathcal{A}$ is not able to distinguish these two geometric range queries. Similarly, we formally define an IND-SCPA Query Privacy, which is presented in Appendix.

6.3 Security Proofs

***Theorem** 1: FastGeo is IND-SCPA data secure if SSW is IND-SCPA data secure and PRF is indistinguishable from a uniformly-random function.*

Sketch: The main idea to prove it is to simulate the data privacy game defined in Def. 4 with an adversary $\mathcal{A}'$. This adversary $\mathcal{A}'$ is able to access the data privacy game of SSW and also access an *oracle*, which is either equal to a PRF or a uniformly-random function. Then, we can demonstrate that compromising the security of our scheme is equivalent to compromising the security of SSW or distinguishing a PRF from a uniformly-random function, which contradicts to the security of SSW or PRF. Details of this proof are presented in Appendix.

With a similar approach, we can also prove our scheme is query secure based on Def. 5. We skip details of the proof of this following theorem due to space limitation.

***Theorem** 2: FastGeo is IND-SCPA query secure if SSW is IND-SCPA query secure and PRF is indistinguishable from a uniformly-random function.*

6.4 Statistic Attacks on Structure Pattern

Attack Analysis. As we discussed, our design leaks structure pattern. This leakage could be leveraged to infer more information about spatial data if an attacker has a stronger ability, where it also has prior knowledge of statistic information on structures.

For instance, in the example described in Fig 2, if an attacker already knows that, when $x = 6$, there are two points while $x = 1$, $x = 3$ or $x = 9$ only has one point. Then, by observing the size of each link list in an encrypted index alone, it can infer which element in this encrypted index is corresponding to $x = 6$ with a probability of 1, because a 2-node link list is unique in this example. On the other hand, to identify which element in this encrypted index is corresponding to $x = 1$, $x = 4$ or $x = 9$, this attacker can only succeed with a probability of $1/3$, since those three elements have the same size of link lists. It is obvious that, for a certain link list size, the number of link lists having this size decides the probability of guessing a correct first-level value. The above type of statistic information of a spatial dataset $\mathbf{D}$ can be formalized as $\chi(\mathbf{D})$, which includes two vectors:

$$\chi(\mathbf{D}) \leftarrow \{\vec{x} = \{x_1, ..., x_w\}, \quad \vec{l} = \{l_1, ..., l_w\}\}$$

where x_i, for $1 \leq i \leq w$, is a distinct first-level value (e.g., x-value) in spatial dataset $\mathbf{D}$, l_i is the number of points having x_i as its first-level value, and w is the total number of distinct first-level values in spatial dataset $\mathbf{D}$. In the above example, we have $\chi(\mathbf{D}) = \{\vec{x} =$

$\{1,3,6,9\}, \vec{l} = \{1,1,2,1\}\}$, where $\vec{l}$ is essentially an unknown permutation of structure pattern $\phi_2 = (s_1, ..., s_w)$.

To formally evaluate the leakage of structure pattern $\phi_2 = (s_1, .., s_w)$ under statistic attacks with prior knowledge $\chi(\mathbf{D})$, we first compute two vectors $\vec{a}$, $\vec{c}$ as

$$\{\vec{a} = (a_1, ..., a_\theta), \quad \vec{c} = (c_1, ..., c_\theta)\} \leftarrow (s_1, ..., s_w)$$

where each a_j, for $1 \leq j \leq \theta$, is a distinct size of link lists in $(s_1, ..., s_w)$, and c_j is the number of link lists that have a size of a_j, and θ is the total number of distinct components in $(s_1, .., s_w)$. Simply, we also have

$$\sum_{j=1}^{\theta} c_j = w, \quad \sum_{j=1}^{\theta} a_j \cdot c_j = n \tag{2}$$

For example, given $\phi_2 = (1,1,2,1)$ and $w = 4$ in Fig. 3, vector $\vec{a}$ and $\vec{c}$ can be calculated as $\vec{a} = (1,2)$ and $\vec{c} = (3,1)$ respectively, where $c_1 = 3$ indicates there are three link lists that have a size of $a_1 = 1$, and $c_2 = 1$ implies there is one link list has a size of $a_2 = 2$.

With $\chi(\mathbf{D})$ and $(s_1, .., s_w)$, the probability for an attacker to correctly guess the first-level value of a data point D_i is

$$p_{i\_1st} = \frac{1}{c_j} \tag{3}$$

where c_j, for $j \in [1, n]$, is the number of link lists having the size of a_j and a_j is the size of the link list containing D_i. Since the probability of an attacker to guess the second-level value of a data point is

$$p_{i\_2nd} = \frac{1}{m} \tag{4}$$

where m is the vector length in the second level. This statistic attacker can further guess the values of a data point D_i at both levels as

$$p_i = p_{i\_1st} \cdot p_{i\_2nd|1st} = p_{i\_1st} \cdot p_{i\_2nd} = \frac{1}{c_j m} \tag{5}$$

Note that, with $\chi(\mathbf{D})$ only, p_{i_1st} and p_{i_2nd} are independent (i.e., $p_{i\_2nd|1st} = p_{i\_2nd}$).

Next, we leverage mean and variance to measure the advantage of this attacker. Specifically, for this attacker, the mean of the probability guessing the values of a data point at both levels can be calculated as

$$\mathsf{E}(p_i) = \frac{1}{n} \cdot \sum_{i=1}^{n} p_i = \frac{1}{n} \cdot \sum_{j=1}^{\theta} \frac{a_j c_j}{c_j m} = \frac{\sum_{j=1}^{\theta} a_j}{nm} \tag{6}$$

Intuitively, a higher mean indicates the advantage of this attacker is higher (i.e., the overall privacy leakage of structure pattern under statistic attacks is higher). Given n, m and structure leakage $(s_1, ..., s_w)$, if $\vec{a} = (a_1, ..., a_w)$ and $\vec{c} = (1, ..., 1)$, where $\theta = w$ and $c_j = 1$ for any $j \in [1, \theta]$, it implies that each link list size is unique (i.e., $p_{i\_1st} = 1$) and the mean is maximized, which is

$$\mathsf{E}_{max} = \frac{\sum_{j=1}^{\theta} a_j \cdot 1}{nm} = \frac{\sum_{j=1}^{\theta} a_j c_j}{nm} = \frac{n}{nm} = \frac{1}{m} \tag{7}$$

On the contrary, if $\vec{a} = (1)$ and $\vec{c} = (n)$, where $\theta = 1$, it indicates that all the link list sizes are the same as 1 (i.e., $p_{i\_1st} = 1/n$) and this privacy leakage on structure is minimized, which is

$$\mathsf{E}_{min} = \frac{\sum_{j=1}^{\theta} a_j}{nm} = \frac{1}{nm} \tag{8}$$

and clearly we have

$$\frac{1}{nm} \leq \mathsf{E}(p_i) \leq \frac{1}{m} \tag{9}$$

Back to our example with $\vec{a} = (1,2)$, $\vec{v} = (3,1)$, where $T = m = 10$ and $n = 5$, $\mathsf{E}(p_i) = \frac{3}{50}$, $\mathsf{E}_{min} = \frac{1}{50}$, and $\mathsf{E}_{max} = \frac{1}{10}$. If for two datasets/structures, where $\mathsf{E}(p_i) = \mathsf{E}(p_i')$, we can further calculate variance $\mathsf{Var}(p_i)$ and $\mathsf{Var}(p_i')$ to measure the leakage to this statistic attacker, where a higher variance indicates a higher leakage in this case. In addition, if we consider all the values of p_i as a random variable P, then we can also leverage its Cumulative Distribution Function (CDF) to further interpret its privacy leakage (i.e., the distribution of p_i, see concrete examples in Sec. 7).

Note that, in reality, an attacker could derive partial prior knowledge of $\chi(\mathbf{D})$ by observing other similar but public spatial datasets, but it is ordinarily hard for this attacker to obtain fully correct and complete prior knowledge of $\chi(\mathbf{D})$ (i.e., some x_i and l_i are unknown or uncertain to this attacker), especially for large-scale datasets. Thus, the leakage we evaluated above is in the worse case, e.g., if $\mathsf{E}(p_i')$ is the attacker's advantage with *partial* prior knowledge, then $\mathsf{E}(p_i') < \mathsf{E}(p_i)$.

Countermeasures. To mitigate the structure leakage of a certain dataset under statistic attacks, there are three approaches can be applied to our design.

The first approach is *padding* [3], where a client pads additional *dummy* nodes to each link list before outsourcing, such that all the link lists have the same size. The second approach is *local counting* [4], where a client keeps a local counter for each link list that has a size greater than 1, such that each link list only has one node. For example, for $x = 6$ in Fig. 3, instead of computing an element as $\mathsf{PRF.GetBits}(6, sk_p)$, and attaching two nodes to this element, a client records a local counter for $x = 6$, and calculates two elements as

$$\mathsf{PRF.GetBits}(6||ctr = 1, sk_p) \quad \mathsf{PRF.GetBits}(6||ctr = 2, sk_p)$$

and attaches one node to each one of these two elements respectively. As a consequence, the structure pattern after applying local counting is $\phi_2' = (1,1,1,1,1)$.

Besides padding and local counting, another approach we can apply is *hiding pointers* [3], where each pointer from a current node to its next node in the list is encrypted with AES and will only be decrypted if this current node is searched. One extra sub-token will be needed for each element in the hash table in order to decrypt the pointer to the first node in each link list during the search process.

As a tradeoff, the search time will slightly increase since revealing each pointer requires an extra AES decryption. This approach can avoid a potentially large number of dummy nodes in padding or a large local storage for counters in local counting. We implement local counting and hiding pointers in our scheme as additional functions to improve security. Tradeoffs of these countermeasures in practice are further investigated in our later experiments in Sec. 7.

Other Statistic Attacks. Besides statistic attacks on structures, an adversary could also carry statistic attacks on access pattern if more information are available. Some recent studies [30]–[33] have shown the impact of this strong attack on keyword queries. We do not consider such strong attacks in this paper, and will study their particular impacts on spatial data in our future work.

6.5 The Tunable Parameter: Vector Length m

For real-world spatial datasets, normally we have a larger data space than the example we mentioned. For example, GPS location is one of the most commonly used spatial data. A longitude and a latitude of a regular location check-in data, e.g., $(\texttt{Long.} = 110.9264, \texttt{Lat.} = 32.2217)$, have seven and six decimal digits respectively. By directly following the examples we have, we can use PRF to encrypt longitude as the first level and leverage SSW to encrypt latitude as the second level. This will certainly guarantee correctness in terms of geometric range search queries. However, a six-digit latitude with SSW will lead to a 1,000,000-length vector in the second level, which will not have efficient search time in practice.

To optimize the performance, we can take some higher digits from the second level and attach them to the first level, such that only shorter vectors are required for the second level. For instance, we can encrypt a location as

$$\mathsf{PRF.GetBits}(110.9264||32.22\#\#, sk_p)$$

in the first level by taking four higher digits from latitude, where $||$ is the concatenation of two strings and $\#$ is a wildcard digit. Let the number of wildcard digit be ρ, then the length vector in the second level can be computed as $m = 10^\rho$. As a result, we only need a 100-length vector for the last two decimal digits of latitude while evaluating SSW in the second level. Essentially, we tune vector length m smaller, and our scheme is now sweeping line segments instead of lines. Similarly, we can also encrypt it as $\mathsf{PRF.GetBits}(110.926\#||32.221\#, sk_p)$, where our scheme essentially sweeps boxes.

A consequence of tuning m smaller is that, the number of elements in the dictionary will increase and the number of first-level subqueries (i.e., q_1) will also increase for a same geometric range query, where more equality checking operations are needed. However, since the unit evaluation time of equality checking on an element is much efficient than the one of an inner product with SSW over *long* vectors, this method will still significantly boost the overall search efficiency.

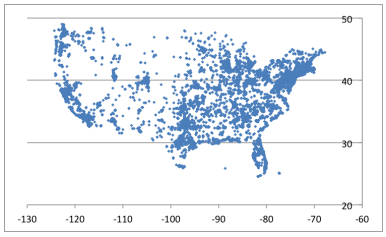


Fig. 8: The distribution of our test dataset.

On the other hand, a smaller m inevitably leads to more privacy leakage. Specifically, tuning m smaller increases the mean of the attacker's probability of guessing the values at both levels (i.e., $\mathsf{E}(p_i)$). Meanwhile, as presented above, first-level query size pattern ($\phi_3 = q_i$) will also increase for a same query. Therefore, vector length m is a tunable parameter in our two-level search to balance privacy leakage and efficiency. In other words, we can also regard vector length m as a part of the input of our leakage function as $\mathcal{L}(\mathbf{D}, Q, m)$.

7 Performance Evaluation

In this section, we leverage a real-world spatial dataset to demonstrate the efficiency of FastGeo. Particularly, we utilize Gowalla location check-in dataset [34]. The raw spatial dataset originally contains 6,442,890 tuples contributed by a number of 196,591 users from all over the world. We pre-process the original dataset before we apply it to our experiments. Specifically, since the number of locations that each user reported in the original dataset varies very differently (e.g., some user reports over 30 times while some user only reports twice), while the locations reported by a same user are normally very close, we only take one tuple from each user in order to avoid bias introduced by different users.

In addition, we focus on locations inside the contiguous U.S. only, where we use Google Maps API (with `python`) to filter out locations outside the contiguous U.S. A small number of tuples with invalid data format are also removed. As a result, we obtain a test dataset (i.e., a subset of Gowalla location check-in dataset), which contains 49,870 tuples (i.e., data points) and will be utilized in following experiments. For each longitude and latitude of a location, we keep four digits after decimal point, e.g., $(\texttt{Long.} = 110.9264, \texttt{Lat.} = 32.2217)$, where the minimal unit is 0.0001. Said differently, for data space Δ_T^α, we have $T = 1 \times 10^7$ and $\alpha = 2$. Note that a 0.0001 difference in longitude or latitude is only around a 10-yard difference in reality. The distribution of this test dataset is shown in Fig. 8.

We evaluate the performance of our scheme with C++, and test experiments on a medium Amazon EC2 instance running Ubuntu 14.04. Functions related to hash tables and link lists are directly from the standard C++ library, PRF is implemented by AES-CBC-256 with OpenSSL, and the running time of SSW is based on the benchmark

results of PBC library and GMP library on our medium Amazon EC2 instance. In the following, for ease of evaluation, when we need to change the value of m, we initially take the longitude and the whole-number part (i.e., digits before the decimal point) of the latitude of a location in the test dataset as the first level, and we use the fractional part of its latitude as the second level in our index, e.g., $\mathsf{PRF.GetBits}(110.9264||32.\#\#\#\#, sk_p)$, where the number of wild digits is $\rho = 4$ and $m = 10^\rho = 10,000$. Then, we decrease the number of wild digits until $\rho = 0$ (i.e., $m = 1$), where both longitude and latitude are included in the first level, e.g., $\mathsf{PRF.GetBits}(110.9264||32.2217, sk_p)$.

7.1 Performance on Encrypted Data

We evaluate the performance of our advanced scheme over encrypted data in several aspects, index generation & encryption time, token generation time, token size, and search time. The two main parameters that impact performance are vector length m and first-level query size q_1. The shape of a geometric range query (either a circle or a polygon) hardly affects the performance.

TABLE 1: Impact of m on index generation time (second).

m	Encrypted Index
1,000	$10,296$
100	981
10	213

The Impact of Vector Length m. As shown in Table 1, when $m = 1,000$ (i.e., $\rho = 3$), it takes around 2.86 hours to build and encrypt our index over the test dataset. For the same dataset, if we choose a smaller vector length, the index generation time can be significantly reduced. For instance, when $m = 100$, it only needs 981 seconds, which is over 10 times faster than the one with $m = 1,000$. Although the generation of an encrypted index takes some time, it is only a one-time cost.

For geometric range queries with a same size (i.e., queries covering a same number of possible points, or said differently, $|Q| = m \times q_1$ is the same), the optimized token generation time is presented in Table 2. We can see that, when we decrease the value of m, the time generating first pieces increases linearly (due to the increase of q_1) while the time spending on second pieces decreases linearly. As a result, at a certain value of m, the overall optimized token generation time is minimized. For instance, when a query contains $|Q| = 1\times10^6$ possible points of the data space, which is approximately the same as the area of Manhattan, NY or the downtown area of San Francisco, CA, if $m = 1,000$, it takes 4.49 seconds on average to generate its search token; while it spends 17.50 seconds if $m = 100$ and 28.48 seconds if $m = 10,000$.

Since the time spent on second pieces also depends on the search results of the first level, the performance of the overall optimized token generation time is *query dependent*. In other words, the actual value of m that minimizes the optimized token generation time depends on different queries. For example, when $|Q| = 1 \times 10^4$, the token time is minimized when $m = 100$.

TABLE 2: Impact of m on optimized token time (s).

$\lvert Q\rvert = m \times q_1$	m	Optimized Token Time		
		First Pieces	Second Pieces	Total
1×10^6	10,000	0.17	28.31	28.48
	1,000	1.66	2.83	4.49
	100	17.21	0.29	17.50
	10	168.02	0.03	168.05
1×10^4	10,000	0.002	7.12	7.122
	1,000	0.02	0.71	0.73
	100	0.18	0.07	0.25
	10	1.76	0.01	1.77

Note that before we apply cryptographic primitives, including PRF and SSW, to generate the search token for a geometric range query, we need to first enumerate all the possible points inside the query and generate the equality-vector form of this range query in plaintext. However, since the overall computation of this sub-step is completely evaluated in plaintext, the running time of this process hardly affects the entire performance of token generation. For instance, given $|Q| = 1 \times 10^6$, if the query is a circle, it takes only 3×10^{-3} seconds to enumerate all the possible points and generate the equality-vector form; and it costs less than 9×10^{-3} seconds if it is a triangle. And when $|Q| = 1 \times 10^4$, the time drop to 3×10^{-5} seconds and 9×10^{-5} seconds, respectively. Obviously, these running time hardly affect the token generation time presented in Table 2.

Besides token generation time, the vector length m also affects token size as shown in Table. 3. As we can observe, token size is also minimized at a certain value of m when token generation time is minimized. For instance, given $|Q| = 1 \times 10^6$, if $m = 1,000$, the token size is 132 KB, which is minimized; given $|Q| = 1 \times 10^4$, the token size is minimized when $m = 100$.

TABLE 3: Impact of m on optimized token size.

$\lvert Q\rvert = m \times q_1$	m	Optimized Token Size		
		First Pieces	Second Pieces	Total Size
1×10^6	10,000	3.2 KB	1 MB	1 MB
	1,000	32 KB	100 KB	132 KB
	100	320 KB	10 KB	330 KB
	10	3.2 MB	1 KB	3.2 MB
1×10^4	10,000	0.03 KB	200 KB	200.03 KB
	1,000	0.32 KB	20 KB	20.32 KB
	100	3.2 KB	2 KB	5.2 KB
	10	32 KB	0.2 KB	32.2 KB

The impact of vector length m on average search time is described in Table 4. For a set of geometric range queries with a same query size, a greater m will end up with a longer average search time. More concretely, when $|Q| = 1 \times 10^6$, if $m = 1,000$, it takes 13.67 seconds on average to operate the search over encrypted data; while if $m = 100$, it only requires 1.50 seconds on average to carry a same query, which is almost ten times faster.

The Impact of First-Level Query Size q_1. Besides

TABLE 4: Impact of m on average search time (s).

$\|Q\| = m \times q_1$	m	Time
1×10^6	1,000	13.67
	100	1.50
	10	0.25

vector length m, first-level query size q_1 also affects the performance in terms of token generation and search. In Fig. 9, with a fixed m, it shows that the token generation time increases with the query size of a geometric range query in the first level.

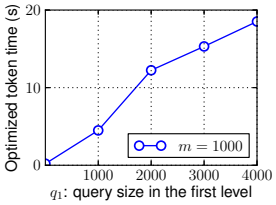


Fig. 9: The impact of q_1 on token generation time.

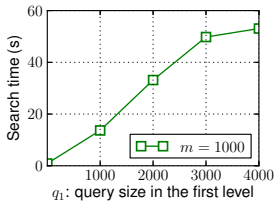


Fig. 10: The impact of q_1 on average search time.

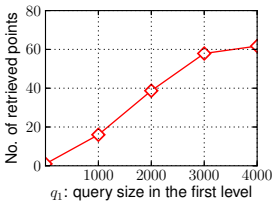


Fig. 11: The impact of q_1 on average result size.

The impact of first-level query size q_1 on the average search time of our scheme is described in Fig. 10. The search time is efficient on average over encrypted data, and normally can be done within one minute. If we compare it with Fig. 11, we can tell that this search time highly depends on the number of retrieved tuples for each given geometric range query, which demonstrates sublinear search. Moreover, this implies that, for geometric range queries with a same size, if a client searches a dense area (e.g., California) with a lot of data points, the search time is slower than searching a sparse area (e.g., Wyoming) as shown in Table 5.

TABLE 5: Average search time (s) in different areas (with $m = 1,000$ and $q_1 = 1,000$).

State	Search Time	Average No. of Retrieved Tuples
California	42.92 seconds	51
Wyoming	0.84 seconds	1

Efficient Updates. Another feature of FastGeo is the ability to carry out efficient updates (i.e., insert, modify, or delete a data point) over encrypted spatial data. Based on our previous discussion, the update time is $O(m\tau')$, where m is the vector length and τ' is the number of nodes in the updated link list. Table 6 shows the impact of m on the average update time over our test dataset. Specifically, when $m = 1,000$, it only takes less than 1 second to update a data point, and a smaller m will further boost the average update time.

TABLE 6: Impact of m on average update time (s).

m	Time
1,000	0.977
100	0.092
10	0.008

7.2 Comparison with Previous Solutions

We compare FastGeo with two previous GSE schemes (denoted as GR [8] and WLW [11] respectively) over our test dataset. The size of geometric range queries tested below is fixed as $|Q| = 1 \times 10^6$, which is about 20 square miles in reality.

GR: Given $T = 1 \times 10^7$, the vector length of GR is $2\lceil \log_2 T \rceil = 48$. GR is *linear* search, and multiple sub-tokens are needed in GR depends on the shape of a query. For ease of comparison, we assume GR outputs one sub-token only for each query, which is the *best case* for its search time and token size.

WLW: We assume the length of Bloom filters used in WLW is $1,000$ and the number of hash functions is 5 in the following comparison. We also utilize the advanced version of WLW, which is built based on R-trees and can achieve *logarithmic* search on average.

FastGeo: We assume vector length m is $1,000$.

TABLE 7: Comparison among schemes.

	Search Time (s)	Complexity	Token Size	Update
GR [8]	1,753	linear	0.96 KB	No
WLW [11]	1,583	logarithmic	20 KB	No
FastGeo	13.67	sublinear	132 KB	Yes

We first compare average search time. As we can see from Table 7, our scheme is extremely efficient, and is at least over 100 times faster compared to others on average. Specifically, our scheme is sublinear search while GR is linear search. In addition, as we described in Table 4, we can also choose a smaller m (e.g., $m = 100$) to further boost the search time of our scheme. More importantly, our scheme can operate highly efficient updates while the other two fail to support. As a necessary tradeoff, our scheme requires more communication cost compared to the other two schemes. For example, our scheme needs 132 KB on token size, while the other two only spend 0.96 KB and 20 KB, respectively. It is worth to mention that the communication cost of our scheme and GR are query dependent (i.e., the token size is different for variant queries), while the one in WLW is constant.

7.3 Leakage under Statistic Attacks

We now evaluate the structure leakage of our scheme over the test dataset under statistic attacks. We can see from Table 8 that, with the increase of m, the privacy

leakage on structure under statistic attacks (i.e., the mean of p_i) is decreasing. Concretely, when $m = 1,000$, the expected probability for an attacker to guess an encrypted point of the values at both levels is only 2.569×10^{-5}, which is very small. Moreover, if we consider all the values of p_i as a random variable P, and we can leverage its Cumulative Distribution Function (CDF) to further demonstrate structure leakage under statistic attacks is small and limited. For example, as shown in Fig. 12 where $m = 1,000$, over 60% of all the p_i are only 3.3×10^{-8}, and over 94% of p_i are less than the mean. Note that about 1.4% of p_i reaches the maximal value 0.001 (i.e., $1/m$) because their corresponding link sizes are unique. In other words, 1.4% of encrypted points are relatively easier to guess than other ones under statistic attacks. However, these points are only a small portion of the entire dataset. For other values of m, the CDF of p_i also have a similar trend as $m = 1,000$, we skip further details due to space limitations.

TABLE 8: Structure leakage under statistic attacks

m	$\mathsf{E}(p_i)$	E_{min}	E_{max}
1,000	2.569×10^{-5}	2×10^{-8}	0.001
100	2.482×10^{-4}	2×10^{-7}	0.01
10	2.434×10^{-3}	2×10^{-6}	0.1
1	2.484×10^{-2}	2×10^{-5}	1

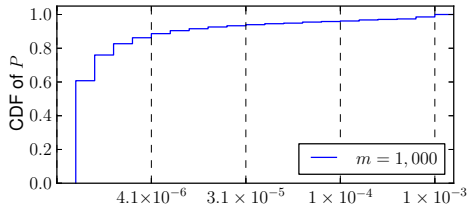


Fig. 12: CDF of P when $m = 1,000$.

In our analyses, we also observe that those unique link list sizes normally happen when a link list size is long in this test dataset. In fact, according to our study, most of the locations stored in those long link lists are major international airports in the U.S. For instance, when $s_k = 117$, which is a unique link list size, it maps to the location of San Francisco International Airport (SFO). In other words, if an attacker has statistic information on structure pattern, then it is easier for this attacker to guess a correct data point if it is at an international airport compared to other places in this test dataset.

Tradeoffs Introduced by Countermeasures. As we mentioned in Sec. 6, three countermeasures, including padding, local counting, and hiding pointers, can be applied to mitigate structure leakage under statistic attacks. Since padding will introduce a huge amount of storage and extra search time, we implement local counting (LC) and hiding pointers (HP) respectively in our scheme as two additional functions to enhance privacy protection. We compare their tradeoffs in Table 9, where we assume query size $|Q| = 1 \times 10^6$ and vector length $m = 1,000$. As we can see, both LC and HP introduce small tradeoffs in search time and token size in order to mitigate structure leakage. In addition, LC requires 80 KB to maintain local counters while HP does not introduce any local storage.

As shown in Table 9 and Fig. 13, we can also see that both LC and HP can significantly mitigate the privacy leakage under statistic attacks. For example, after applying LC or HP, the probability for an attacker (with statistic information on structure pattern) to guess a point is 2.747×10^{-8}. As a result, for points with unique link size before applying countermeasures (e.g., $p_i = 1/m = 0.001$), it is over 3×10^5 times harder for an attacker to reveal those points.

TABLE 9: Tradeoffs Introduced by Countermeasures.

	Search Time	Token Size	Local Storage	$E(p_i)$
With LC	13.68 s	142 KB	80 KB	2.747×10^{-8}
With HP	13.70 s	135 KB	0 KB	2.747×10^{-8}
FastGeo	13.67 s	132 KB	0 KB	2.569×10^{-5}

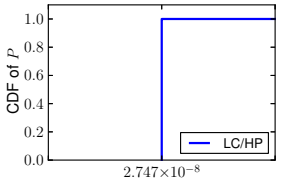


Fig. 13: CDF of P with LC/HP when $m = 1,000$.

8 Conclusion

We propose FastGeo, an efficient two-level search scheme that can operate geometric ranges over encrypted spatial datasets. Our experiment results over a real-world dataset demonstrate its effectiveness in practice. Moreover, our comparison with previous solutions indicates that the general idea of two-level search can be leveraged as an important methodology to boost search time and enable highly efficient updates over encrypted data when complex operations, such as compute-then-compare operations, are involved in search.

References

[1] D. Song, D. Wagner, and A. Perrig, "Practical Techniques for Searches on Encrypted Data," in *Proc. of IEEE S&P'00*, 2000.

[2] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky, "Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions," in *Proc. of ACM CCS'06*, 2006.

[3] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic Searchable Symmetric Encryption," in *Proc. of ACM CCS'12*, 2012.

[4] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner, "Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries ," in *Proc. of CRYPTO'13*, 2013.

[5] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, W. George, A. Keromytis, and S. Bellovin, "Blind Seer: A Searchable Private DBMS," in *Proc. of IEEE S&P'14*, 2014.

[6] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner, "Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation," in *Proc. of NDSS'14*, 2014.

[7] E. Stefanov, C. Papamanthou, and E. Shi, "Practical Dynamic Searchable Encryption with Small Leakage," in *Proc. of NDSS'14*, 2014.

[8] G. Ghinita and R. Rughinis, "An Efficient Privacy-Preserving System for Monitoring Mobile Users: Making Searchable Encryption Practical," in *Proc. of ACM CODASPY'14*, 2014.

[9] B. Wang, M. Li, H. Wang, and H. Li, "Circular Range Search on Encrypted Spatial Data," in *Proc. of IEEE CNS'15*, 2015.

[10] H. Zhu, R. Lu, C. Huang, L. Chen, and H. Li, "An Efficient Privacy-PReserving Location Based Services Query Scheme in Outsourced Cloud," *Ieee Trans. on Vehicular Technology*, 2015.

[11] B. Wang, M. Li, and H. Wang, "Geometric Range Search on Encrypted Spatial Data," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 4, pp. 704–719, 2016.

[12] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2008.

[13] Satyan L. Devadoss and Joseph O'Rourke, *Discrete and Computational Geometry*. Princeton University Press, 2011.

[14] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*, Second, Ed. CRC Press, 2014.

[15] R. A. Popa, F. H. Li, and N. Zeldovich, "An Ideal-Security Protocol for Order-Preserving Encoding," in *Proc. of IEEE S&P'13*, 2013.

[16] P. Paillier, "Public-key Cryptosystems Based on composite Degree Residuosity Classes," in *Proc. of EUROCRYP'99*, 1999.

[17] D. Boneh, E.-J. Goh, and K. Nissim, "Evaluating 2-DNF Formulas on Ciphertexts," in *Proc. of TCC'05*, 2005.

[18] C. Gentry, "Fully Homomorphic Encryption Using Ideal Lattices," in *Proc. of STOC'09*, 2009.

[19] E. Shen, E. Shi, and B. Waters, "Predicate Privacy in Encryption Systems," in *Proc. of TCC'09*, 2009.

[20] E. Shi, J. Bethencourt, T.-H. H. Chan, D. Song, and A. Perrig, "Multi-Dimensional Range Query over Encrypted Data," in *Proc. of IEEE S&P'07*, 2007.

[21] D. Boneh and B. Waters, "Conjunctive, Subset, and Range Queries on Encrypted Data," in *the Proc. of TCC*, 2007.

[22] Y. Lu, "Privacy-Preserving Logarithmic-time Search on Encrypted Data in Cloud," in *Proc. of NDSS'12*, 2012.

[23] 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.

[24] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," in *Proc. of ACM SIGMOD'84*, 1984.

[25] W. Du and M. J. Atallah, "Secure Multi-Party Computation Problems and Their Applications: a Review and Open Problems," in *Proc. of the Workshop on New Security Paradigms*, 2001.

[26] M. J. Atallah and W. Du, "Secure Multiparty Computational Geometry," in *International Workshop on Algorithms and Data Strucutres*, 2001.

[27] A. Narayanan, N. Thiagarajan, M. Lakhani, M. Hamburg, and D. Boneh, "Location Privacy via Private Proximity Testing," in *Proc. of NDSS'11*, 2011.

[28] J. Sedenka and P. Gasti, "Privacy-Preserving Distance Computation and Proxitity Testing on Earth, Done Right," in *Proc. of ACM ASIACCS'14*, 2014.

[29] B. Wang, Y. Hou, M. Li, H. Wang, and H. Li, "Maple: Scalable Multi-Dimensional Range Search over Encrypted Cloud Data with Tree-based Index," in *Proc. of ACM ASIACCS'14*, 2014.

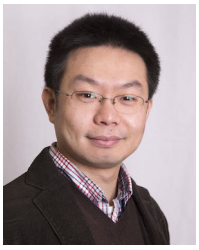
[30] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access Pattern Disclosure on Searchable Encryption: Ramification, Attack and Mitigation," in *Proc. of NDSS'12*, 2012.

[31] M. Kuzu, M. S. Islam, and M. Kantarcioglu, "Efficient Privacy-Aware Search over Encrypted Databases," in *ACM CODASPY'14*, 2014, pp. 249–256.

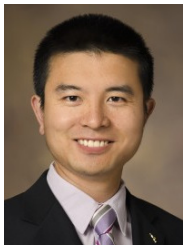
[32] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-Abuse Attacks Against Searchable Encryption," in *Proc. of CCS'15*, 2015.

[33] Y. Zhang, J. Katz, and C. Papamanthou, "All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption," in *USENIX Security*, 2016, pp. 707–720.

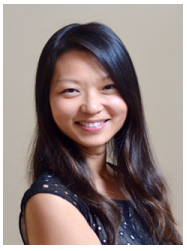
[34] E. Cho, S. A. Myers, and J. Leskovec, "Friendship and Mobility: User Movement in Location-Based Social Networks," in *ACM SIGKDD KDD'11*, 2011.



Boyang Wang is a Ph.D. Candidate in the Department of Electrical and Computer Engineering, the University of Arizona. He received his B.S. degree in Information Security in 2007 and his first Ph.D. degree in Cryptography in 2013 both from School of Telecommunications Engineering, Xidian University, China. His current research interests focus on security and privacy issues in cloud computing, big data, and applied cryptography. He is a student member of IEEE.



Ming Li is an Associate Professor in the Department of Electrical and Computer Engineering of University of Arizona. He was an Assistant Professor in the Computer Science Department at Utah State University from 2011 to 2015. He received his Ph.D. from Worcester Polytechnic Institute in 2011. His main research interests are cyber security and wireless networks, with current emphases on privacy-preserving big data analytics, wireless and spectrum security, and cyber-physical system security. He received the NSF Early Faculty Development (CAREER) Award in 2014. He has won a distinguished paper award from ACM ASIACCS 2013, and CCC blue sky ideas award for best vision papers at ACM SIGSPATIAL 2015. He serves on the TPC of several premier conferences including IEEE INFOCOM, CNS, ACM ASIACCS, and WiSec. He is a member of both IEEE and ACM.



Li Xiong is Professor of Computer Science (and Biomedical Informatics) and holds a Winship Distinguished Research Professorship at Emory University. She has a PhD from Georgia Institute of Technology, an MS from Johns Hopkins University, and a BS from University of Science and Technology of China, all in Computer Science. She and her research group, Assured Information Management and Sharing (AIMS), conduct research that addresses both fundamental and applied questions at the interface of data privacy and security, spatiotemporal data management, and health informatics. She is a recipient of a Google Research Award, IBM Faculty Innovation Award, Cisco Research Award, and Woodrow Wilson Fellowship. Her research is supported by NSF (National Science Foundation), NIH (National Institute of Health), AFOSR (Air Force Office of Scientific Research), and PCORI (Patient-Centered Outcomes Research Institute).

APPENDIX

A. Correctness of Our Scheme

Assume a *perfect* hash function is implemented in the hash table, where its error probability is

$$\begin{aligned} p_{e\_hash} &= \Pr(\mathsf{HashTable.Find}(x,\Gamma)=1,\ \textit{where } x\notin\Gamma) \\ &\leq \mathtt{negl}_1(\lambda) \end{aligned}$$

And according to [19], the error probability of SSW.Query is

$$\begin{aligned} p_{e\_ssw} &= \Pr(\mathsf{SSW.Query}([\vec{u}],[\vec{v}])=1,\ \textit{where } <\vec{u},\vec{v}>\neq 0) \\ &\leq \mathtt{negl}_2(\lambda) \end{aligned}$$

For a point $D_i = (d_{i,1}, d_{i,2})$ and a geometric range query Q, when $D_i \notin Q$, there are three different cases for our scheme to erroneously return identifier I_i: 1) If $d_{i,1} \notin Q_x$ and $d_{i,2} \notin Q_y$, where Q_x is the range projection of Q in x-dimension and Q_y is the range projection of Q in y-dimension. The error probability of returning I_i is

$$p_1 = p_{e\_hash} \cdot p_{e\_ssw} \leq \mathtt{negl}_1(\lambda)\cdot\mathtt{negl}_2(\lambda) = \mathtt{negl}'(\lambda)$$

2) If $d_{i,1} \notin Q_x$ but $d_{i,2} \in Q_y$, the error probability is $p_2 = p_{e\_hash} \leq \mathtt{negl}_1(\lambda)$; 3) If $d_{i,1} \in Q_x$ but $d_{i,2} \notin Q_y$, the error probability is $p_3 = p_{e\_ssw} \leq \mathtt{negl}_2(\lambda)$;

Therefore, if $D_i \notin Q$, the error probability is $\Pr[I_i \in \mathbf{I}_Q | D_i \notin Q] = \mathtt{max}\{p_1, p_2, p_3\} \leq \mathtt{negl}(\lambda)$.

B. IND-SCPA Query Privacy

***Definition** 5:* **IND-SCPA Query Privacy.** *Let* $\Pi = \{\mathsf{GenKey}, \mathsf{BuildIndex}, \mathsf{Enc}, \mathsf{GenToken}, \mathsf{Query}\}$ *be a symmetric-key GSE scheme over security parameter* λ*. We define a security game between a challenger* $\mathcal{C}$ *and an adversary* $\mathcal{A}$ *as below*

***Init**: $\mathcal{A}$ submits two queries Q_0 and Q_1 to $\mathcal{C}$, where $Q_0, Q_1 \subseteq \Delta_T^\alpha$ and they are subject to $\mathcal{L}(Q_0) = \mathcal{L}(Q_1)$.*

***Setup**: $\mathcal{C}$ runs $\mathsf{GenKey}(1^\lambda)$ to generate a secret key sk.*

***Phase 1**: $\mathcal{A}$ adaptively submits a number of requests where each request is one of the two following types:*

- *Ciphertext Request: On the jth ciphertext request, $\mathcal{A}$ outputs a dataset $\mathbf{D}_j = (D_{j,1}, ..., D_{j,n})$, where $D_{j,i} \in \Delta_T^\alpha$, for $1 \leq i \leq n$. $\mathcal{C}$ responses with an encrypted index Γ_j^*, where $\Gamma_j^* \leftarrow \mathsf{Enc}(\Gamma_j)$, $\Gamma_j \leftarrow \mathsf{BuildIndex}(\mathbf{D}_j)$, and $\mathbf{D}_j$ is subject to $L(\Gamma_j, Q_0) = \mathcal{L}(\Gamma_j, Q_1)$.*
- *Token Request: On the jth token request, adversary $\mathcal{A}$ outputs a geometric range query Q'_j, where $Q'_j \in \Delta_T^\alpha$. Challenger $\mathcal{B}$ responses with a search token tk'_j $\mathsf{GenToken}(Q'_j, sk)$, where Q'_j is subject to $\mathcal{L}(Q'_j, Q_0) = \mathcal{L}(Q'_j, Q_1)$.*

***Challenge**: With Q_0, Q_1 selected in **Init**, $\mathcal{C}$ flips a coin $b \in \{0,1\}$, and returns tk_{Q_b} to $\mathcal{A}$, where $tk_{Q_b} \leftarrow \mathsf{GenToken}(Q_b, sk)$.*

***Phase 2**: $\mathcal{A}$ continues to adaptively submit requests, which are subject to the same restrictions in **Phase 1**.*

***Guess**: $\mathcal{A}$ takes a guess b' of b.*

We say that Π *is secure against Selective Chosen-Plaintext Attacks on query privacy if for any polynomial time adversary* $\mathcal{A}$ *in the above game, it has at most negligible advantage*

$$\mathbf{Adv}^{\mathrm{Query}}_{\Pi,\mathcal{A}}(1^\lambda) = \left|\Pr[b'=b] - \frac{1}{2}\right| \leq \mathtt{negl}(\lambda) \qquad (10)$$

C. Detailed Proof of Theorem 1

Proof: **Init**: $\mathcal{A}'$ selects two datasets $\mathbf{D}_0$ and $\mathbf{D}_1$, where $\mathbf{D}_0 = (D_{0,1}, ..., D_{0,n})$, $\mathbf{D}_1 = (D_{1,1}, ..., D_{1,n})$, $D_{0,i}, D_{1,i} \in \Delta_T^\alpha$, for $1 \leq i \leq n$. $\mathcal{A}'$ computes $\Gamma_0 \leftarrow \mathsf{BuildIndex}(\mathbf{D}_0, \sigma)$ and $\Gamma_1 \leftarrow \mathsf{BuildIndex}(\mathbf{D}_1, \sigma)$. Since Γ_0 and Γ_1 are subject to $\mathcal{L}(\Gamma_0) = \mathcal{L}(\Gamma_1)$, we have $\Gamma_0 = \{\{e_{0,1}, list_{0,1}\}, ..., \{e_{0,w}, list_{0,w}\}\}$, $\Gamma_1 = \{\{e_{1,1}, list_{1,1}\}, ..., \{e_{1,w}, list_{1,w}\}\}$, where $list_{0,j}.\mathsf{size}() = list_{1,j}.\mathsf{size}()$, for $1 \leq j \leq w$. $\mathcal{A}'$ submits Γ_0 and Γ_1 to $\mathcal{C}$.

Setup: $\mathcal{C}$ runs $sk_p \leftarrow \mathsf{PRF.Init}(1^\lambda)$ and $sk_s \leftarrow \mathsf{SSW.Setup}(1^\lambda)$ to setup a secret key $sk = \{sk_p, sk_s\}$. $\mathcal{C}$ also initializes a local table $\mathcal{E}$.

Phase 1: $\mathcal{A}'$ issues a number of requests. For a ciphertext request, $\mathcal{A}'$ outputs a dataset $\mathbf{D}'_j$, where $\mathbf{D}'_j = (D'_{j,1}, ..., D'_{j,n})$ and $D'_{j,i} \in \Delta_T^\alpha$. $\mathcal{A}'$ computes $\Gamma'_j \leftarrow \mathsf{BuildIndex}(\mathbf{D}'_j, \sigma)$, where $\Gamma'_j = \{\{e'_{j,k}, list'_{j,k}\} | \textit{ for } 1 \leq k \leq w'_j\}$. Since $\mathbf{D}'_j$ is subject to that it does not have any same first-level values with neither $\mathbf{D}_0$ and $\mathbf{D}_1$, we have $e'_{j,k} \notin \{e_{0,1}, .., e_{0,w}, e_{1,1}, ..., e_{1,w}\}$, for any $k \in [1, w'_j]$. $\mathcal{A}'$ submits Γ'_j to $\mathcal{C}$.

$\mathcal{C}$ initializes an encrypted index Γ_j^* as null. Given a pair of $\{e'_{j,k}, list'_{j,k}\} \in \Gamma'_j$, where $1 \leq k \leq w'_j$, $\mathcal{C}$ first checks its local table $\mathcal{E}$ with $e'_{j,k}$. If $e'_{j,k}$ is not in the table, $\mathcal{C}$ either calculates $[e'_{j,k}] = \mathsf{PRF.Init}(e'_{j,k}, sk_p)$ or randomly pick a $[e'_{j,k}]$, and inserts $(e'_{j,k}, [e'_{j,k}])$ to table $\mathcal{E}$; otherwise, $\mathcal{C}$ directly retrieves $[e'_{j,k}]$ from table $\mathcal{E}$. Then, $\mathcal{C}$ initializes a list $list'^*_{j,k}$ as null. For each $\vec{u}'_{j,k,l} \in list'_{j,k}$, where $1 \leq l \leq list'_{j,k}.\mathsf{size}()$, $\mathcal{C}$ computes $[\vec{u}'_{j,k,l}] = \mathsf{SSW.Enc}(\vec{u}'_{j,k,l}, sk_s)$, and appends $[\vec{u}'_{j,k,l}]$ to list $list'^*_{j,k}$. For a pair of $\{e'_{j,k}, list'_{j,k}\}$, $\mathcal{C}$ outputs a pair of $\{[e'_{j,k}], list'^*_{j,k}\}$, and inserts this pair to Γ_j^*. Finally, $\mathcal{C}$ returns Γ_j^* as an output of a ciphertext request.

For a token request, $\mathcal{A}'$ outputs a query Q_j, where $\mathcal{L}(\Gamma_0, Q_j) = \mathcal{L}(\Gamma_1, Q_j)$. $\mathcal{A}'$ represents Q_j in enhanced equality-vector form as $S_{Q_j} = \{\{e_{j,i}, \vec{v}_{j,i}\} | \textit{ for } 1 \leq i \leq q_{x,j}\}$, and submits S_{Q_j} to $\mathcal{C}$. Given $\mathcal{L}(\Gamma_0, Q_j) = \mathcal{L}(\Gamma_1, Q_j)$, it indicates, for each $e_{j,i}$, where $1 \leq i \leq q_{x,j}$,

$$\begin{aligned} &((\mathsf{HT.Find}(\Gamma_0, e_{j,i}) \neq null) \wedge (\mathsf{HT.Find}(\Gamma_1, e_{j,i}) \neq null)) \\ \textit{or}\ &((\mathsf{HT.Find}(\Gamma_0, e_{j,i}) = null) \wedge (\mathsf{HT.Find}(\Gamma_1, e_{j,i}) = null)) \end{aligned}$$

If $\mathsf{HashTable.Find}(\Gamma_0, e_{j,i})$ is not null, then for $\vec{v}_{j,i}$, the requirement of $\mathcal{L}(\Gamma_0, Q_j) = \mathcal{L}(\Gamma_1, Q_j)$ indicates

$$\begin{aligned} &(<\vec{u}_{0,k}, \vec{v}_{j,i}>= 0 \ \wedge \ <\vec{u}_{1,k}, \vec{v}_{j,i}>= 0) \\ \textit{or}\ &(<\vec{u}_{0,k}, \vec{v}_{j,i}>\neq 0 \ \wedge \ <\vec{u}_{1,k}, \vec{v}_{j,i}>\neq 0) \end{aligned}$$

where $\vec{u}_{0,k} \in list_0$, $\vec{u}_{1,k} \in list_1$, $list_0.\mathsf{size}() = list_1.\mathsf{size}()$, $1 \leq k \leq list_0.\mathsf{size}()$, $list_0 \leftarrow \mathsf{HashTable.Find}(\Gamma_0, e_{j,i})$ and $list_1 \leftarrow \mathsf{HashTable.Find}(\Gamma_1, e_{j,i})$.

Given each $\{e_{j,i}, \vec{v}_{j,i}\} \in S_{Q_j}$, for $1 \leq i \leq q_{x,j}$, $\mathcal{C}$ first checks whether $e_{j,i}$ is in table $\mathcal{E}$. If it is not in the table, $\mathcal{C}$ either computes $[e_{j,i}] = \mathsf{PRF.GetBits}(e_{j,i}, sk_p)$ or randomly pick a $[e_{j,i}]$, and inserts $(e_{j,i}, [e_{j,i}])$ to table $\mathcal{E}$; otherwise, $\mathcal{C}$ directly retrieves $[e_{j,i}]$ from table $\mathcal{E}$. Then, $\mathcal{C}$ computes $[\vec{v}_{j,i}] = \mathsf{SSW.GenToken}(\vec{v}_{j,i}, sk_s)$, and outputs a sub-token $tk_{j,i} = \{tk_{p,j,i}, tk_{s,j,i}\} = \{[e_{j,i}], [\vec{v}_{j,i}]\}$. Finally, $\mathcal{C}$ outputs a search token $tk_{Q_j} = \{tk_{j,i} | \textit{ for } 1 \leq i \leq q_{x,j}\}$.

Challenge: With Γ_0 and Γ_1, $\mathcal{C}$ flips a coin $b \in \{0, 1\}$, initializes Γ_b^* as null, and given each a pair of $\{e_{b,k}, list_{b,k}\} \in \Gamma_b$, where $1 \leq k \leq w$, for each $e_{b,k}$, if $e_{b,k}$ is not in the local table $\mathcal{E}$, $\mathcal{C}$ calculates as below: if $b = 0$, it computes $[e_{b,k}] = \mathsf{PRF.GetBits}(e_{b,k}, sk_p)$; otherwise, it randomly picks $[e_{b,k}]$. And then it inserts $(e_{b,k}, [e_{b,k}])$ to table $\mathcal{E}$; otherwise, $\mathcal{C}$ retrieves $[e_{b,k}]$ directly from table $\mathcal{E}$. Then, $\mathcal{C}$ initializes a list $list_{b,k}^*$ as null. For each $\vec{u}_{b,k,l} \in list_{b,k}$, where $1 \leq l \leq s_k$ and $s_k = list_{b,k}.\mathsf{size}()$, $\mathcal{C}$ computes $[\vec{u}_{b,k,l}] = \mathsf{SSW.Enc}(\vec{u}_{b,k,l}, sk_s)$, and appends $[\vec{u}_{b,k,l}]$ to list $list_{b,k}^*$. For a pair of $\{e_{b,k}, list_{b,k}\}$, $\mathcal{C}$ outputs a pair of $\{[e_{b,k}], list_{b,k}^*\}$, and inserts this pair to Γ_b^*. Finally, $\mathcal{C}$ returns Γ_b^* to $\mathcal{A}'$.

$\mathcal{A}'$ continues to adaptively submit a number of requests in **Phase 2**, and takes a guess b' of b in **Guess**.

In the above simulation, if $\mathcal{A}'$ could distinguish $\vec{u}_{0,k,l}$ and $\vec{v}_{1,k,l}$, for any $l \in [1, s_k]$ and for any $k \in [1, w]$, where $\sum_{k=1}^{w} s_k = n$, or if for any $e_{0,i} \neq e_{1,i}$, where $i \in [1, w]$, $\mathcal{A}'$ could distinguish a PRF from a uniformly-random function, then $\mathcal{A}'$ could distinguish Γ_0 and Γ_1 (i.e., distinguish the two spatial datasets $\mathbf{D}_0$ and $\mathbf{D}_1$). Therefore, the advantage of $\mathcal{A}'$ distinguishing $\mathbf{D}_0$ and $\mathbf{D}_1$ is

$$\mathbf{Adv}_{\Pi,\mathcal{A}'}^{\text{Data}}(1^\lambda) = 1 - (1 - \mathbf{Adv}_{\text{SSW},\mathcal{A}'}^{\text{Data}})^n \cdot (1 - \mathbf{Adv}_{\text{PRF},\mathcal{A}'})^{w'}$$

where w' is the number of $e_{0,i} \neq e_{1,i}$ and $w' \leq w$. Based on the security of PRF and SSW, we learn that $\mathbf{Adv}_{\text{SSW},\mathcal{A}'}^{\text{Data}} \leq \mathtt{negl}_1(\lambda)$ and $\mathbf{Adv}_{\text{PRF},\mathcal{A}'} \leq \mathtt{negl}_2(\lambda)$, then we have

$$\begin{aligned} \mathbf{Adv}_{\Pi,\mathcal{A}'}^{\text{Data}}(1^\lambda) &\leq 1 - (1 - \mathtt{negl}_1(\lambda))^n \cdot (1 - \mathtt{negl}_2(\lambda))^{w'} \\ &\leq 1 - (1 - \mathtt{negl}'(\lambda)) \\ &= \mathtt{negl}'(\lambda) \end{aligned}$$

which demonstrates our scheme is data secure. Properties of negligible functions [14] are applied here. □