Secure Range Query Based on Spatial Index

Dingxing Xie*, Yanchao Lu*, Congjin Du*, Jie Li†, Li Li†
Department of Computer Science and Engineering, Shanghai Jiao Tong University
Shanghai, China 200240
*{haber, chzblych, cyberteller}@sjtu.edu.cn, †{lijie, lilijp}@cs.sjtu.edu.cn

***Abstract*—Sensor network has become an increasingly attractive and advantageous subject recently. More and more demands of data storage and data query have been raised in soft-defined sensor network. Bonnet et al. [1] investigated the problem of database in sensor network. In most of such scenes, data is stored in server instead of local. For this reason, data security [2] is very important. While encryption of outsourced data protects against many privacy threats, it could not hide the access patterns of the users. Protecting user information from leakage or attackers while guaranteeing high efficiency of query is becoming an important problem of concern. In this paper, we discuss secure range query based on spatial index. We build the spatial index on the client instead of the server to keep the information away from the potential threat. While keeping a high efficiency of query, we not only encrypt the data, but also hide the access patterns. That will greatly reduce the risk of data leakage. We do simulations and prove our design to be practicable and effective.**

***Index Terms*—Sensor network; Database; Data Security; Range Query; Spatial Data**

I. Introduction

Cloud database [3] has been more and more frequently used in soft-defined sensor networks recently. Outsourcing data to cloud stores has become a popular strategy, as cloud properties like scalability and flexibility allow for significant costs savings.

However, remote data storage causes security risks. The cloud infrastructures cannot always be completely trusted, due to, for example, hacker and insider attacks. Users may not trust that the server will use their data privately while they still need to query from the cloud. While encryption of outsourced data protects against many privacy threats in cloud scenarios, it renders subsequent operations on data (i.e., data analysis) extremely difficult. In some situations, for example in some map applications, we need to retrieve geography data, or some data with even higher dimensions. Conventional encryption schemes, such as block ciphers [6], do not directly support the sorts of comparisons, searches, and other manipulations needed to handle queries without loss of privacy. Though many studies have been done in secure range query [7] [8] [9] [10] [11], they could not handle spatial data perfectly.

In this paper, we study secure range query based on spatial index in order to solve these problems. Our system consists of a server and a client. First we build an R-tree in the client and an address translator to handle spatial data and ensure security; then we build a hierarchical shelter in the server and a levelmap in the client, and design shuffle algorithms to reduce the information leakage.

The rest of this paper is organized as follows. We describe the related work in Section II. System definition is presented in Section III. In Section IV, we present the system architecture and protocol. Evaluations and discussions are reported in Section V and in Section VI we make a conclusion.

II. Related work

Spatial data is one of the most frequently used type of data in sensor network. Spatial data objects often cover areas in multi-dimensional spaces and are not well represented by point locations. For example, map objects like countries, villages etc. occupy regions of non-zero size in two dimensions. A common operation on spatial data is a search for all objects in an area, for example to find all countries that have land within 10 miles of a particular point. This kind of spatial search occurs frequently in computer aided design and geography data applications, and therefore it is important to be able to retrieve objects efficiently according to their spatial location.

The R-tree [15] is a height-balanced tree used for indexing multi-dimensional data. Each R-tree node contains several entries. Each leaf node entry has the form (object-id,R) where object-id is the objects identifier and R is the MBR of the data object. Each internal node entry is of the form (ptr,R) where prt is a pointer to a lower level node and R is its MBR. R-trees support efficient range queries. R-tree variants, such as the R+-tree, the R*-tree and the Rhat-tree [16] based on ASPE [17], incorporate various enhancements.

In the other hand, data security is also very important in soft-defined sensor network. Database is a common way in soft-defined sensor network to store data in server and accesses through the network. Although the individual data records are encrypted, an untrusted server could still infer information about them by observing multiple range and sort operations. For example, the cloud could learn access patterns and correlate them. Consequently, also the analysis operations (queries) need to be privacy protected. Goldreich and Ostrovsky [12] investigated the problem of hiding access patterns in the context of RAM machines. Their motivation was to hide the program executed by a processor from an attacker snooping on the traffic to main memory. Their model consists of a physically shielded CPU that contains a key secret to the outside world; the key is used to store the program encrypted in memory. The CPU progressively fetches instructions from the program and decrypts them using its internal (and also shielded) registers. The execution is said to be oblivious (and the RAM machined called an oblivious RAM ORAM) if the

INISCom 2015, March 02-04, Tokyo, Japan

DOI 10.4108/icst.iniscom.2015.258364

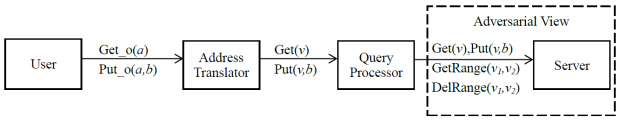


Fig. 1. System Overview.

access patterns for any two inputs causing the same number of accesses to RAM are indistinguishable. An elegant recent work of Pinkas and Reinman [13] improves both the complexity and constants of the Goldreich-Ostrovsky approach by leveraging recent techniques such as Cuckoo Hashing and Randomized Shell Sort. Boneh et al. presented Remote Oblivious Storage which makes proceeding in parallel possible. However, these solutions could not handle spatial data.

However these solutions could not ensure the data security against the attackers who will learn information about the content of the data from the pattern of query. In this paper, we build an R-tree index based on oblivious storage [14] to ensure data security in range queries to spatial data.

III. System Definitions

There are two main parts in our system: a server and a client. The server stores the large amount of data. The client also has some storage capacity, though considerably less than the server. We also define the attacker. Figure 1 provides a high-level overview of the following security definitions.

A. Server and Client

We have a server S and a client C. The data stored in the server is like (v, b). The client includes two parts. One is address translator, and the other is query processor. The client C responses the orders from the users, process queries to the server, and returns the corresponding answer to the users. The server S and the client C responses to the following messages:

B. User

The user U gives the order like Get_o(a) or Put_o(a, b). We assume that the user U only sends a new order to the client after it received an answer to the previous message.

TABLE I
Message in Our System

Server	Get(v)	S returns (v, b) if the id v exists in the server, or returns $NULL$ if there is no such id.
	Put(v, b)	S places entry (v, b) in its storage. If the id v already exists, S overwrites it.
	GetRange($v1, v2$)	S returns all entries (v, b) for some v, where $v1vv2$.
	DelRange($v1, v2$)	S deletes all entries (v, b) for some v, where $v1vv2$.
	Halt	S halts
Client	Get_o(a)	Get all data in range a.
	Put_o(a, b)	Overwrites the data in range a to b.

C. Attaker

We assume that the server is untrusted. The attacker may have access to both the data stored at the server and the messages exchanged between the client and the server.

We use a definition similar to semantic security where the attacker cannot distinguish between the access patterns of any two users. The attacker cannot tamper with the data requested to and returned by the server, but would like to learn information about the content of the data. Allowing the attacker to tamper with the data is a straightforward extension to our protocol similar to the one from [12] and we will thus not focus on this case. (If identifiers are selected from a large field to prevent repeats, adding a simple MAC to the block corresponding to each identifier will allow detection of tampering.)

IV. Secure Range Query Based on Spatial Index

A. Overview

Our system includes a server S and a client C. The client C includes an address translator and a query processor.

Our server memory includes a flat main part and a hierarchical shelter. We chose to represent the storage at the server as a random-access memory with a key-value interface similar to that provided by cloud storage services [3] (e.g., Amazon S3 [4] and Windows Azure [5]). The customer is charged based on the amount of storage used and the number of requests performed. Most cloud services use a hashing algorithm allowing $O(1)$ data retrieval time. To avoid ambiguity of terms, we use the term identifier-block pair (or simply id-block pair) instead of key-value pair. We use the notation v to denote an identifier and b to denote a certain block.

Let c be the size of the identifiers (in bits) and B be the sum of the size of a block and the size of an identifier. Note that, in practice, B is much larger than c. For example, a common value for B is 4 KB and c is 32 bits, resulting in B = 1024c. There are M idblock pairs in the server storage, resulting in a total of MB storage. The shelter is hierarchical and consists of several levels labeled 0, 1n, the size of each level growing exponentially.

The data stored in the server is like (v, b), and the data input by the user is like (a, b). Both b means the data we needed. a means the coordinate in the multidimensional space, which has the structure of $(d1, d2...dn)$, where n means the number of dimension and $d1, d2$ means the range in every dimension respectively. For example, when $n = 2$, $a = (d1, d2)$, where $d1 = (x1, x2)$, $d2 = (y1, y2)$, means the area from $x1$ to $x2$ and $y1$ to $y2$.

The reason we use different data structure is based on two aspects. Firstly, the simpler the data stored in the server is, the better the performance will be. (v,b) is simpler than (a,b), and will achieve higher performance. Secondary, storing a directly in the server will cause risk of data leakage. Although we

Algorithm 1: Get_o(a)

Find data in the data in range a in R, let the result be v where $v = (v1, v2)$.
foreach $vn \in v$ **do**
  send a Get(v) to the query processor
  return the result to the user
end

Algorithm 2: Put_o(a, b)

Find data in the data in range a in R, let the result be v where $v = (v1, v2)$.
foreach $vn \in v$ **do**
  send a Put(v) to the query processor
  return the result to the user
end

could encrypt it, the risk still exists. Storing (v,b) instead of (a,b) will significantly reduce the risk of data leakage.

B. Attack Translator

There is an address translator in the client, which includes an R-tree R. When receiving messages from the users, the address translator translates (a, b) to (v, b), that is translate a to v. Unlike traditional R-tree, the leaf node of the R-tree in the address translator do not stores the data b, but the corresponding id v in the server. In other words, the R-tree in the address translator stores (a, v). For these reasons, we could store the whole R-tree in the client. The algorithm of the address translator is below:

C. Client

The client stores in memory a map, called the LevelMap L, mapping each identifier for which there is an entry in the shelter to the level numbers in which the entry exists. The reverse is also stored in the map: a map from each level number to all identifiers present in that level in the shelter. LevelMap consists of any efficient search data structure. Here in our system we use B+tree. In addition to identifierblock pairs, the client maintains in memory the following information: a secret key SK with $|SK| = k$ for encrypting blocks, the value s and all si mentioned above, as well as a dummy counter d for the main part and dummy counters di for each level that allow the client to fetch the next dummy identifier by incrementing these values (and thus preventing repeated accesses to the same dummy identifier). We use PRPs to denote a length-preserving pseudorandom permutation [18] with domain and range being the identifier space. For the rest of the paper, for simplicity, we say that the client searches, requests, or puts an identifier v from/to the server to mean that the client searches, requests, or puts the permuted identifier v, permuted using a PRP seeded with a client secret. The algorithm of the query processor is below:

D. Shuffle

Figure 2 shows an overview of our shuffling algorithms:

- shuffle(): Shuffles a part of the server's memory in which the identifiers have been permuted with the same PRP.

Algorithm 3: Get(v)/Put(a)

Let l be the highest level in the LevelMap
if vL **then**
  foreach $i \in L$ **do**
    if il **then**
      Send a dummy request $Get(PRPsi(di))$, ignore the answer
    end
    else
      Send a request $Get(PRPsi(di))$, let b be the answer
    end
  end
  Send a dummy request $Get(PRPs(d))$, ignore the answer
  Call shelterInsert(v, b)
  Let decryptSK(b) be the result and send it to the user
end
else
  foreach $i \in L$ **do**
    Send a dummy request $Get(PRPsi(di))$, ignore the answer
  end
  Send a dummy request $Get(PRPs(d))$, let b be the answer
  Call shelterInsert(v, b)
  Let decryptSK(b) be the result and send it to the user
end

Algorithm 4: shelterInsert(v, b)

Let $b = encryptSK(decryptSK(b))$
Send Put(PRPs1(v),b) to the server
while *a level is full* **do**
  if *full level = the last level of the hierarchy* **then**
    Call shuffleMainPart()
  end
  else
    Call shuffleLevel()
  end
end

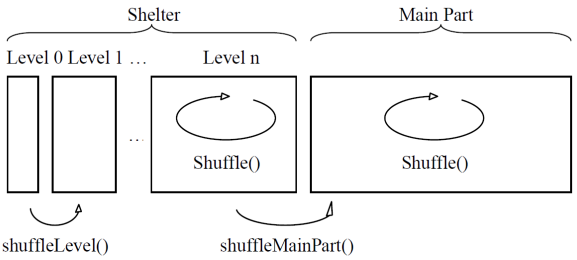


Fig. 2. Shuffle Algorithms Overview.

- shuffleMainPart(): Incorporates a flat shelter into the main part of storage and shuffles the latter.
- shuffleLevel(): Move a level i in the hierarchy of the shelter into level i+1.

While shuffling, any request to an identifier in the shelter will be blocked. Allowing concurrent accesses and shuffling in-place are two goals in tension because it seems natural to make a copy of the data on which to allow accesses to happen while shuffling the original data in parallel. Our solution has three main ideas:

1. During shuffling, the client builds a data structure called the shuffleMap. The shuffleMap summarizes to what identifier each identifier was mapped after a step of the shuffle and stores it at the server. The shuffleMap allows the client to find a requested identifier in a partially-shuffled memory efficiently.

Algorithm 5: shuffle()

```
Logically split the memory in ranges of pairs containing data of size O(m)
Number the chunks with i = 1, 2...C, where C = O(MB/m)
foreach chunk in Server memory do
    Use GetRange() to get the chunk
    Use DelRange() to delete the chunk
    Add to each identifier the prefix "i:"
    Use Put() to put back the range at the server
end
for i = 0toC do
    foreach chunk j in all chunks do
        if chunk j is not already paired with an earlier chunk then
            Pair chunk j with chunk j* = j + 2^i
        end
        Use GetRange(j : 0...0, j : 9...9") and
        GetRange(j* : 0...0, j* : 9...9") to copy both chunks to local
        Use DelRange() to delete the copied chunks
        Use PRP() to each identifier
        Re-encrypt the blocks
        Sort the pairs by the new identifier
        foreach chunk in the first half of the pairs do
            Add a prefix "j:"
        end
        foreach chunk in the second half of the pairs do
            Add a prefix "j*:"
        end
    end
end
foreach chunk in the Server memory do
    Remove the prefix
end
```

Algorithm 6: shuffleMainPart()

```
Create a new hierarchical shelter, the concurrent shelter, containing no pairs.
Make a copy of the current shelter at the server, denoted the copy shelter.
Shuffle the main part using shuffle().
Shuffle the original shelter using shuffle().
foreach item in Shelter do
    if item is not a dummy then
        Update the item in the main part
    end
    else
        Update a dummy in the main part
    end
end
Shuffle the main part using shuffle().
Delete the copy shelter and the original shelter.
Let the concurrent shelter be the new shelter.
```

Algorithm 7: shuffleLevel(i)

```
Shuffle Level i using shuffle()
Shuffle Level i + 1 using shuffle()
foreach pair j in level i do
    if there is a same(permuted) identifier j in level i + 1 then
        Update the value in level i + 1 with the value from level i
    end
    else if pair j is not a dummy then
        Set the value of a dummy variable to contain their value.
    end
end
Delete all contents from level i.
Shuffle level i + 1 using shuffle().
```

Algorithm 8: conc_get(v)/conc_put(v, b)

```
if the request is not concurrent with shuffleMainPart() then
    Simply perform get(v)/put(v) at the server and return any results.
end
else if this request accesses the original shelter then
    if shuffleMainPart() is happening then
        Use the copy shelter
    end
    else
        //shuffleLevel() is happening
        Finish the shuffle
        Proceed with the request
    end
end
else
    Perform the request to the concurrent shelter
    if the request to the main part comes in step 3 of algorithm 6 then
        perform the dummy or permuted-v request directly to the main part as
        in the unchanged algorithm;
    end
    else if starts between step 3-4 then
        Use shuffleMap() to find the permuted identifier at the server and
        request it
    end
    else if starts between step 4-6 then
        Make the request to the main part using the last permutation used by
        the client in the shuffle at Step 3
    end
    else if starts between step 6-7 then
        Use shuffleMap() to find the permuted identifier at the server and
        request it
    end
    else
        Make the request to the main part using the last permutation used by
        the client in the shuffle at Step 6
    end
end
```

2. The client creates a new shelter called the concurrent shelter where any concurrent requests must be stored to prevent repetitions with the data accessed in the preceding epoch, during the shuffle, and in the following epoch.

3. The client either only allows a maximum number of concurrent requests during a shuffle or delays such requests, both in order to ensure that shuffling ends before a new shuffle must begin.

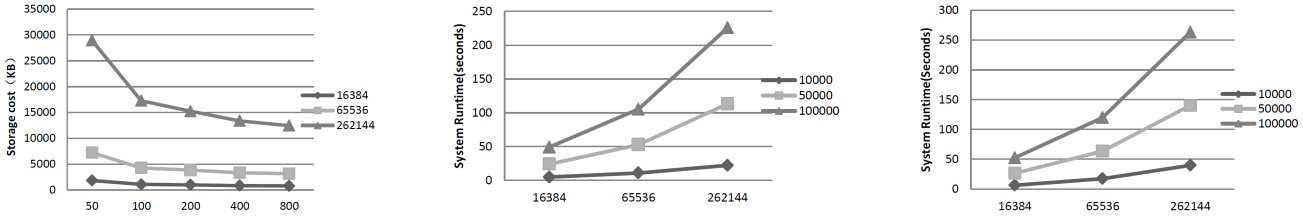
shuffle() splits the memory in portions that are O(m) in size and thus fit in the client's memory, and shuffle pairs of these portions at a time. After shuffle(), the client deletes the shuffleMap(), except for the data corresponding to the last iteration. The client will perform $O((MB/m)log(MB/m))$ requests, and it will Get/Put $O(Mlog(MB/m))$ pairs. Once the last level of the hierarchy gets full, the client uses shuffleMainPart() to shuffle the last level into the main part. The shelter in shuffleMainPart() is the last level in the hierarchy.

E. Concurrent

We explain how requests to the main part are satisfied when the main part is being shuffled considering that we cannot make a copy of the main part because it would introduce too much storage overhead as the following. Note that one shuffle must finish before the next one needs to start. Since a shelter can hold as many as $O(m/c)$ pairs before it needs to be shuffled into the main part, at most $O(m/c)$ requests can happen concurrently for the duration of shuffleMainPart().

V. Evaluation and Discussion

In this section we present out experiments. We generated three sets of data shown in Table II. We use a computer with 3.4GHz i7 CPU and 8GByte RAM. The size of each entry is 4096B. The first 32B contain the geography information and the others contain the data.



(a) R-tree Scale Test

(b) System Runtime

(c) Waiting Time While Shuffling

Fig. 3. Results of Experiments

The numbers of entries of the data sets M are 16384, 65536 and 262144, costing 64MB, 256MB and 1GB of the server memory. So the memory size of the client is $m = \sqrt{MB}$, that is 8KB, 16KB and 32KB respectively

We build an R-tree for each data set. The parameter LC(LeafCapacity) means the maximum number of child that a node can have. The size of the R-tree and LeafCapacity is shown in Table III and Figure 3-a:

The results of the experiments show that the scale of R-tree is small enough to store in the client. As the parameter LeafCapacity increases, the size of R-tree decreases as well. However, the speed of decrease will reduce to 0 as the size decreases. In consideration of performance, we set LeafCapacity to 400.

We make three experiments, proceed 10000, 50000 and 100000 queries respectively. We do them 10 times each, and let the average value be the result. The result is shown in Figure 3-b:

The bottleneck of our system is the communication overhead between the server and the client because the speed of R-tree searching is much faster than communication. The memory of client is $B\sqrt{M}$, the time of get/put is $O(logM)$. In consideration of shuffling, the time will be $O(\sqrt{M}/BlogMB)$, which is different from the real result in the experiments. The reason is that the server only allows at most $O(m/c)$ requests concurrently for the duration of shuffleMainPart().

We do statistics of the I/O time of Get/Put during shuffling. The time statistics is presented in Table IV and the waiting time is shown in Figure 3-c:

The result in Table III fits the theoretical hypothesis in Algorithm 5. It also shows that when $MB/m = 2^n$, the performance will be the best. The result shown in Figure 3-c also shows that as the increase of the number of the items. The waiting time while shuffling increases as well. This is due to two reasons: in one hand, as it is shown in Table III, the waiting time while shuffling increases as the increase of the scale of data set. This makes the speed of shuffling lower, and increases the waiting time. In the other hand, as the increase of the scale, the shelter will be bigger, which will reduce the time of shuffling and increase the concurrent request allowed. This will also reduce the waiting time.

Moreover, we could learn from the experiments that when M is a certain value, the larger the memory of the client m is, the fast the algorithm becomes. How to make a balance between speed and scale is a future direction of our research.

TABLE II
SCALE OF DATA SET

Number of entries	Server Memory	ClientMemory
$2^{14} = 16384$	64MB	8KB
$2^{16} = 65536$	256MB	16KB
$2^{18} = 262144$	1GB	32KB

TABLE III
R-TREE SCALE TEST

Leaf Capacity	16384	65536	262144
50	1875968B	7397376B	29659136B
100	1101824B	4358144B	17689184B
200	1011712B	3940352B	15593472B
400	851968B	3411968B	13688832B
800	57344B	806812B	12759040B

TABLE IV
NUMBER OF GET/PUT WHILE SHUFFLING

Number of items	Number of Get/Put
$2^{10} = 1024$	11264
$2^{14} = 16384$	196608
$2^{18} = 262144$	3407872

VI. CONCLUSION

In this paper we present our secure range query system design based on spatial index. Our main contributions are: First, we study the problem of secure range query of spatial data. Second, we present our system design. We not only encrypts the data, but also hide the query pattern to reduce the information leakage while proceeding range query in spatial data. Third, we implement the design and carry out a performance evaluation of the system.

ACKNOWLEDGEMENT

This work is partially sponsored by the National Basic Research 973 Program of China (No. 2015CB352403), the National Natural Science Foundation of China (NSFC) (No. 61261160502, No. 61272099), the Program for Changjiang Scholars and Innovative Research Team in University (IRT1158, PCSIRT), the Scientific Innovation Act of STCSM

(No. 13511504200), and the EU FP7 CLIMBER project (No. PIRSES-GA-2012-318939).

This work is also sponsored by the National Natural Science Foundation of China (NSFC) (No. 61202025, No. 61428204), Singapore NRF (CREATE E2S2), the State High-Tech Development Plan (2013AA01A601), Microsoft Research Asia (the Urban Informatics Research Grant) and STCSM Grant (No. 12ZR1414900).

REFERENCES

[1] P. Bonnet, J. Gehrke, and P. Seshadri. Towards Sensor Database Systems. In *proc. IEEE MDM*, pp. 3-14, 2001.
[2] H. Chan, A. Perrig, and D. Song. Secure hierarchical in-network aggregation in sensor networks. In *proc. ACM CCS*, pp. 278-287, 2006.
[3] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50-58, 2010.
[4] Amazon. Amazon s3 service level agreement. http://aws.amazon.com/s3-sla/, 2009.
[5] Microsoft Corporation. Windows Azure. http://www.microsoft.com/windowsazure, 2014.
[6] K. Jonathan and Y. Lindell. Introduction to modern cryptography: principles and protocols. CRC Press, 2007.
[7] H. Hacigümüş, B. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *proc. SIGMOD*, 2002.
[8] H. Hacigümüş, B. Iyer, and S. Mehrotra. Efficient execution of aggregation queries over encrypted relational databases. In *proc. DASFAA*, 2004.
[9] B. Hore, S. Mehrotra, M. Canim, and M. Kantarcioglu. Secure multidimensional range queries over outsourced data. *VLDBJ*, 21(3):333-358, 2012.
[10] E. Mykletun and G. Tsudik. Aggregation queries in the database-as-a-service model. *Data and Applications Security XX*, pp. 89-103, 2006.
[11] E. Shi, J. Bethencourt, T. Chan, D. Song, and A. Perrig. Multidimensional range query over encrypted data. In *proc. IEEE SP*, 2007.
[12] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 43(3):431-473, 1996.
[13] B. Pinkas and T. Reinman. Oblivious RAM revisited. In *Proc. CRYPTO*, pp. 502-519, 2010.
[14] D. Boneh, D. Mazieres, and RA. Popa. Remote oblivious storage: Making oblivious RAM practical. 2009.
[15] A. Guttman. R-trees: a dynamic index structure for spatial searching. ACM, 1984.
[16] WK. Wong, DW. Cheung, B. Kao, and N. Mamoulis. Secure kNN computation on encrypted databases. In *proc. SIGMOD*, 2009.
[17] P. Wang and CV. Ravishankar. Secure and efficient range queries on outsourced databases using Rp-trees. In *proc. ICDE*, 2013.
[18] O. Goldreich. Foundations of Cryptography. Cambridge University Press, 2003.
[19] R. Agrawal and R. Srikant. Privacy-preserving data mining. In *proc. SIGMOD*, 2000.
[20] A. Evfimievski, R. Srikant, R. Agrawal, and J. Gehrke. Privacy preserving mining of association rules. In *proc. KDD*, 2002.