Towards Practical Private Processing of Database Queries over Public Data with Homomorphic Encryption

Shiyuan Wang, Divyakant Agrawal, Amr El Abbadi

Department of Computer Science
University of California, Santa Barbara, USA
{sywang, agrawal, amr}@cs.ucsb.edu

***Abstract*—Data privacy is a major concern when users query public online data services. The privacy of millions of people has been jeopardized in numerous user data leakage incidents in many popular online applications. To address the critical problem of personal data leakage through queries, we enable private querying on public data services so that the contents of user queries and any user data are hidden and therefore not revealed to the online service provider. We propose two protocols for processing private database queries, namely BHE and HHE. BHE provides complete query privacy by using Paillier's homomorphic encryption along with the bucketization of public data. In contrast to traditional Private Information Retrieval (PIR) proposals, BHE only incurs one round of client server interaction for processing one query. Built upon BHE, HHE is a hybrid protocol that applies BHE computation and communication on a subset of the data buckets, such that this subset covers the actual requested data but also mimics frequent query patterns of common users, thus achieving practical query performance while providing proper privacy protection. Because of the use of frequent query patterns and data specific privacy protection, HHE is not vulnerable to traditional attacks on k-Anonymity that explore data similarity and skewness. Moreover, HHE consistently protects user query privacy for a sequence of queries in a query session.**

I. Introduction

Public online data services have become important information sources. Billions of people query search engines, social network sites, news portals, and different kinds of specialized information services every day. Research organizations and companies also often need to access public data sources in their work. For example, biomedical researchers who need to verify properties derived from a local gene database, may join a publicly released gene bank with their local gene data table. As another example, a company may search on an online patent database for patents in the same area as its forthcoming inventions.

Privacy, however, has been a big concern when querying these public online data services. User queries can be leaked intentionally to advertisers such as in some of the Google and Facebook applications [2] or even to the general public such as in the AOL log release incident in 2006 [5]. User queries can also be leaked unintentionally, such as to the attackers who hacked the back-end database of Twitter in 2008 [3]. Some of the leaked query traces contain sensitive and even confidential data, such as personal addresses, incomes, medical conditions, and other sensitive information such as credit card numbers and social security numbers [5].

Existing solutions for protecting user query privacy in online data services are not yet satisfactory. Pulling the entire public data to the client side to process user queries on the client is infeasible in many cases. Anonymizing networks such as Tor [8] hide user identities, but require users to trust peers in a third-party infrastructure, and do not protect sensitive contents in the queries from being leaked. *Private Information Retrieval* (PIR) [7], [19] provides users complete privacy and confidentiality by using cryptographic protocols, but traditional single-server PIR protocols [12], [19] are too computationally expensive to be practical [33], while multi-server PIR protocols [7] do not allow communication among all servers. Recent single-server PIR protocol [22] significantly improves server computational performance, but incurs expensive communication costs. Query anonymization, i.e. using k-Anonymity [32], [34], on the other hand, is a low cost solution but does not provide strong privacy, as k-Anonymity and its variants are vulnerable to a number of attacks [10], [20]. Such attacks also apply to hybrid solutions that run PIR on k-anonymous data subset [26], [35].

The aim of this paper is to design techniques towards a practical private database query processing mechanism, such that computation and communication costs are affordable, and proper privacy protection for users' private data and queries are in place. We model a public online data service as a database on a remote server, and consider any user data involved in a query and the plaintext query contents as private data. We focus on the private processing of range queries over public data and joins of private and public data, as range queries and joins are the basis for supporting a lot more database queries.

To conquer the challenge of obfuscated query formulation on the client, i.e. especially for range queries, we propose data bucketization on the server and synchronizing bucket summary with clients. As the basic framework for private database query processing, we fist propose a protocol called BHE based on *homomorphic encryption* [28]. Like PIR, BHE provides users complete privacy and confidentiality. In contrast to PIR, BHE only incurs a *single* round of client server interaction for processing a query.

To make private query processing practical while avoiding the vulnerabilities of k-Anonymity related query anonymization solutions and hybrid approaches, we then propose a new *hybrid* protocol called HHE based on BHE. HHE reduces computation and communication costs by allowing the server to know that a client is interested in a subset of all data buckets, such that the computation and communication are only consumed for that subset instead of on the entire data in BHE. This subset covers a client's actual query buckets, and also includes the relevant frequently co-accessed sets of buckets of other clients, such that this subset is not likely to be linked a specific individual and the client's actual query buckets are not likely to be filtered out. In contrast to k-Anonymity based hybrid approaches [26], [35] which only provide a coarse privacy guarantee of $1/k$ regardless of the data and query semantics, HHE provides a *privacy risk measure that is specific to which buckets and frequent patterns are included in the subset*. Moreover, HHE consistently *protects user query privacy in a sequence of queries of a query session* by generating cover buckets at the beginning of the session and generating decoy buckets based on cover buckets for each subsequent query. Both cover and decoy buckets are generated to minimize privacy risk under a user given constraint.

Our contributions are summarized as follows.

- We propose two protocols using homomorphic encryption, namely BHE and HHE, to solve the private processing of two important database queries, joins and range queries, which can be the basis for private processing of other database queries.
- The basic protocol BHE only incurs a single round of client server interaction for processing a query.
- The hybrid, lower costs protocol HHE is better than previous hybrid solutions in that it protects user query privacy for a sequence of queries in a query session, and it is not subject to traditional attacks on k-Anonymity, because its generated decoy buckets are frequently associated with the private query buckets in the global query history.
- Our privacy measure is defined for a query session, and is specific to the actual data requested and decoy buckets used.

Road map: Section II reviews the related work. Section III reviews homomorphic encryption and presents our model. Section IV and Section V describe the BHE and HHE protocols respectively. Section VI experimentally evaluates the proposed protocols, while Section VII concludes the paper.

II. Related Work

We categorize the previous work for solving the problem of private query processing as follows: Private Information Retrieval, query anonymization using noisy queries, Plausibly Deniable Search, and hybrid approaches.

Private Information Retrieval (PIR) models the private retrieval of public data as a theoretical problem: Given a server which stores a binary string $x = x_1...x_n$ of length n, a client wants to retrieve x_i privately such that the server does *not learn i at all*. Chor et al. [7] introduced the PIR problem and gave solutions on multiple servers. Kushilevitz and Ostrovsky proposed a single server, computational PIR solution [19] which is usually referred to as cPIR. cPIR and its follow-up single-server PIR proposals, however, are criticized as impractical for their expensive computation costs [33], although communication costs have been reduced in the follow-up single-server cPIR proposals [12]. Multi-server PIR solutions have been shown to be more efficient than single-server PIR solutions [25], however, multi-server PIR does not allow communication among all the servers, thus leaving only single-server PIR suitable to use in public online data services. Two approaches were later proposed to make single-server PIR practical. One uses oblivious RAM, but it only applies to a specific setting where a client retrieves its own data outsourced on the server [37]. The other bases the foundation of its PIR protocol on linear algebra [22] instead of the number theory which previous single-server PIR solutions base on. Unfortunately, the latter lattice based PIR scheme cannot guarantee that its security is as strong as previous PIR solutions, and it incurs a lot more communication costs.

Instead of hiding a user's query completely and providing strong privacy and confidentiality in PIR, query anonymization usually uses k-Anonymity [32], [34] and its variants to mix the user's query with other noisy query data. For example in privacy-preserving location based services [23], a user's query point is anonymized with an enclosing region containing $k-1$ points of other users. Such anonymization is computationally efficient compared to PIR, but it does not provide strong privacy, e.g. query anonymization based on k-Anonymity is subject to attacks on k-Anonymity and its variants such as similarity attacks [20] and composition attacks [10]. A similar anonymization technique which generates additional noisy queries is employed in a private web search tool called TrackMeNot [15]. The privacy in TrackMeNot, however, is broken by query classification [30], which suggests that randomly extracted noise alone does not protect a query from identification.

To avoid the vulnerabilities of random k-anonymous noises and to generate meaningful and disguising noises, *Plausibly Deniable Search* (PDS) is proposed in [24], [29] for private text search. PDS employs a topic model or an existing taxonomy to build a static clustering of cover word sets. The words in each cluster are of different topics but with similar specificity to their respective topics, thus are used to cover each other in a query. PDS protects privacy for one query, but not for a sequence of queries. Our HHE protocol is similar to PDS in that we generate meaningful but not similar buckets to cover the buckets containing the requested data. In contrast to PDS, HHE generates these cover / decoy buckets dynamically, and protects privacy for a sequence of queries in a query session.

Recently, hybrid approaches have been proposed to use query anonymization in global and apply PIR protocols in an anonymized data subset [26], [35], thus gaining a good trade-off between processing efficiency and query privacy. However, there are two problems with these hybrid proposals: First, their selection of an anonymized subset just follows the simple k-Anonymity model, and does not have a detailed privacy measure to justify the selection; Second, they are only focused on specific queries such as simple selections [35] and location queries [26]. Our HHE protocol is also a hybrid approach in that it combines cover bucket selection with a homomorphic encryption based protocol, BHE. HHE solves the above mentioned two problems: First, instead of following k-Anonymity or its variants, HHE defines a privacy risk metric based on query semantic and minimizes the privacy risk in selecting cover buckets; Second, HHE supports both joins and range queries, and can be extended to support other database queries.

Other techniques have been proposed for solving private processing of specific queries. For example, private join of a private data column and a public data column is implemented by hashing in [31], but join by hashing is not able to retrieve other relevant data columns. A recent paper proposes processing private remote kNN queries using homomorphic encryption [16]. Theoretical protocols using homomorphic encryption have been proposed to process private document search by keywords in a stream of documents [6], [27]. These protocols are still too expensive to be practical, and they can only perform approximate search. Finally, we are not concerned with private query processing on outsourced encrypted data, although our data bucketization is inspired by the data bucketization idea in a work from that area [13]. Our approaches can also apply to protecting query privacy in outsourced scenarios.

III. Preliminaries

A. Homomorphic Encryption

We rely on *homomorphic encryption* [11], [28] to provide strong privacy protection. Homomorphic encryption allows addition and multiplication to be performed directly on ciphertexts without the need for decryption. Without loss of generality, we use the popular Paillier's homomorphic encryption [28].

Paillier's homomorphic encryption is a public key cryptosystem. Let $n = pq$ where p and q are large primes. Let the public key $K_{pub} = n$, and the private key K_{priv} be the factorization of n, (p, q). In a finite field, $Z_n \times Z_n^* \mapsto Z_{n^2}^*$, in which Z_n and Z_n^* are groups of integers modulo n. In Paillier's cryptosystem, plaintexts $m \in Z_n$ and ciphertexts $c \in Z_{n^2}^*$. The encryption and decryption functions are defined as $E : Z_n \to Z_{n^2}^*$ and $D : Z_{n^2}^* \to Z_n$ respectively. Then the following properties hold:

$$\forall m_1, m_2 \in Z_n$$

$$D(E(m_1)E(m_2) \mod n^2) = m_1 + m_2 \mod n \quad (1)$$

$$D(E(m_1)^{m_2} \mod n^2) = m_1 m_2 \mod n \quad (2)$$

These properties enable one party (e.g. server) to perform blind calculation on the ciphertext provided by another party (e.g. client), while only the party holding the private key (e.g. client) can decrypt and obtain the result. For example, a client sends $E(0)$ to a server, the server performs $E(0)E(100) = E(0 + 100) = E(100)$, and the client decrypts to obtain 100. Similarly, if the client sends $E(1)$ to the server, the server performs $E(1)^{100} = E(1 \cdot 100) = E(100)$, and the client still decrypts to obtain 100. Pailier's cryptosystem has proven to be semantically secure based on the decisional composite residuosity assumption (DCRA) [28].

B. System Model

We refer to private data owners and query initiators as *clients*, and refer to public data service providers as *servers*. We consider interactions between one client and one server. We abstract the private data on the client and the public data on the server as relational databases R_{priv} and R_{pub} respectively. Furthermore, we denote a tuple in R_{priv} as $t(R_{priv})$ and a tuple in R_{pub} as $t(R_{pub})$. let $R_{priv}.A$ and $R_{pub}.B$ be the attributes in R_{priv} and R_{pub} respectively for evaluating query conditions. We assume that these attributes are ordered, and have matching schemas and data domains.

We focus on range queries and joins. A range query specifies a range $[r_1, r_2]$ based on the values of $R_{priv}.A$ and asks for matching R_{pub} tuples whose $R_{pub}.B$ values fall in $[r_1, r_2]$ (half closed range queries are also supported), $t(R_{pub})_{R_{pub}.B \in [r_1, r_2]}$. A join query joins R_{priv} tuples with R_{pub} tuples whose $R_{priv}.A$ and $R_{pub}.B$ values are equal, $t(R_{priv}) \bowtie_{R_{priv}.A = R_{pub}.B} t(R_{pub})$. Similar queries such as a join query with a range constraint can be easily processed once we have the basic foundation to process range and join queries. Without loss of generality, assume that the query results do not just include qualified $R_{pub}.B$ values but include the entire corresponding R_{pub} tuples.

Our goal is to preserve the privacy of a user's private data for a sequence of queries in a session. We assume a public data server has interactions with a large number of such user query sessions. A session usually has a small number of queries. We assume that users do not create profiles on the server nor do they have cookies enabled, and users do not disclose their IP addresses to the server (e.g. via the use of a proxy).

C. Adversary Model

We consider the server and any other attackers who can view the data retrieved from the server and monitor activities on the server as adversaries. We assume that the adversaries are honest but curious: they perform computations correctly and completely as required by the query processing protocol (i.e. they are not malicious), but they are free to infer clients' queries and clients' private data. We assume that data stored on clients are secure, and one client does not collude with adversaries to compromise another client's privacy.

IV. Basic Private Query Processing

One of the main challenges for private query processing is to *privately* represent a given user query, and find and retrieve the qualified values from $R_{pub}.B$ for the query. In our basic framework, we propose to use a novel approach of data bucketization with homomorphic encryption to solve this challenge, and we provide perfect privacy of *query indistinguishability* for *clients*, meaning that the adversaries who may have control of *servers* should not be able to differentiate accesses of different queries on $R_{pub}.B$. One advantage of our framework over any other PIR protocols is that our framework can answer a query in *only one round* of client server interaction, thus saving the bandwidth for the server.

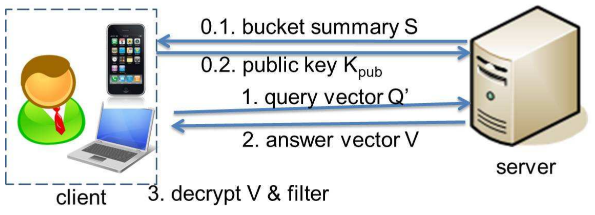


Fig. 1. BHE. In this protocol, before processing any queries, 0.1) Server sends the bucket summary S of its database to the client; 0.2) Client sends her public key K_{pub} to the server. Then to process a query q, 1) Client formulates an encrypted query vector Q' based on S and q, and sends Q' to the server; 2) Server performs blind processing on Q' and public database, sends the answer vector V back to the client; 3) Finally, the client decrypts V and reconstructs the answer to the query q.

A. Public Data Bucketization

We first describe data bucketization. Assume that the ordered values of $R_{pub}.B$ are divided into b buckets, $BK_1, BK_2, ..., BK_b$, in which the values in bucket BK_i are no larger than the values in the bucket BK_{i+1}. The R_{pub} tuples corresponding to the $R_{pub}.B$ values in a bucket BK_i, $t(R_{pub})_{R_{pub}.B\in BK_i}$, can be located in constant time. The boundary values of the buckets, $\{BK_i : [v_{i1}, v_{i2})\}$, are publicly accessible. Let this summary information of the buckets be S. For example, Given $R_{pub}.B \in [0, 100)$, a bucketization summary $S_1 = \{BK_1 : [0, 20), BK_2 : [20, 50), BK_3 : [50, 60), BK_4 : [60, 70), BK_5 : [70, 85), BK_6 : [85, 95), BK_7 : [95, 100)\}$. S should be small enough to be downloadable to the client, since S will be used by the client to formulate obfuscated queries for private query processing on the server. We assume that the server decides the data bucketization. In our technical report [36], we discuss the trade-offs of different bucketizations. In Section VI, we experimentally evaluate the effect of different bucketizations on query performance.

B. Basic Query Processing Protocol

Before a client issues her first query on R_{pub}, the client obtains the bucket summary S about $R_{pub}.B$, generates a privacy, public key pair (K_{priv}, K_{pub}) as defined in Section III-A, and then sends to the server her public key K_{pub}. The server uses K_{pub} to initialize ciphertexts to be calculated for processing all subsequent queries of the client.

After these preliminary steps, processing a query takes the following three steps (also shown in Fig. 1). We call our Basic private query processing protocol based on Homomorphic Encryption as BHE.

Query Formulation. A client uses the bucket summary S to determine the buckets which may have matches to her query, denoted as *potentially matching buckets*. For a range query, the potentially matching buckets are consecutive buckets $BK_i, ..., BK_j$, the concatenation of whose value ranges $[v_{i1}, v_{j2})$ minimally covers the query range $[r_1, r_2]$. For a join, the potentially matching buckets are the buckets whose range the $R_{priv}.A$ values fall in, $[v_{i1}, v_{i2})$. For example, given $S_1 = \{BK_1 : [0, 20), BK_2 : [20, 50), BK_3 : [50, 60), BK_4 : [60, 70), BK_5 : [70, 85), BK_6 : [85, 95), BK_7 : [95, 100)\}$, the potential matching buckets for a range query $[45, 65)$ are BK_2, BK_3 and BK_4; the potentially matching buckets for joining with $R_{priv}.A = \{10, 30, 50, 55, 90\}$ are BK_1, BK_2, BK_3 and BK_6. Note that there is only a slight variation in query formulation and result post filtering between processing a range query and processing a join. The rest of the processing is the same for both types of queries.

After finding the potentially matching buckets, the client creates a binary vector of size b, denoted as Q, sets the entries whose indexes correspond to the positions of potentially matching buckets as 1, and the other entries as 0. Then the client encrypts each entry of Q using her public key K_{pub} with re-randomization by multiplying each time by a new encryption of 0 (re-randomization does not change the entry values according to Equation (1)), resulting in an encrypted query vector Q'. The client sends Q' to the server. In the above example, Q' = ($E(0)$, $E(1)$, $E(1)$, $E(1)$, $E(0)$, $E(0)$, $E(0)$) for the range query, whereas Q' = ($E(1)$, $E(1)$, $E(1)$, $E(0)$, $E(0)$, $E(1)$, $E(0)$) for the join.

Query Processing on Server. Upon receiving Q', the server creates a vector V of the same size as the size of Q', b, and initializes all the entries in V as $E(0)$ using the client's public key K_{pub}. For each entry i in V, the server retrieves $t(R_{pub})_{R_{pub}.B\in BK_i}$, transforms them to byte sequence $ToBytes(t(R_{pub})_{R_{pub}.B\in BK_i})$, and then performs a modular exponentiation $V[i] = Q'[i]^{VBK_i}$, where $VBK_i = ToBytes(t(R_{pub})_{R_{pub}.B\in BK_i})$ is a big integer transformed from the byte sequence. The transformation is a one-to-one mapping so that the client can derive the original $R_{pub}.B$ values in BK_i when she gets VBK_i. In practice when the size of BK_i is too big for all the tuple values of BK_i to fit in a single ciphertext of modulo n[1], we break $t(R_{pub})_{R_{pub}.B\in BK_i}$ into a group of big integers VBK_is, resulting in $V[i]$ being a group of ciphertexts that have the potentially obfuscated matching values. To simplify the following discussion, we can think of $V[i]$ consisting of only one ciphertext value. The server performs the above modular exponentiation on all buckets. After that the server sends vector V to the client.

Query Processing on Client. Upon receiving V, the client only needs to process the entries of $Q[i] = 1$ (the previous potentially matching buckets). For such an entry with index i, the client decrypts $V[i]$ using her private key K_{priv}. According to Equation (2), $D(V[i]) = D(Q'[i]^{VBK_i}) = D(Q'[i]) \cdot VBK_i$. Since BK_i is a potentially matching bucket, $D(Q'[i]) = Q[i] = 1$, the decryption result $D(V[i]) = VBK_i$, from which the client reconstructs the original values of $t(R_{pub})$ in BK_i. Then the client performs post filtering on these potentially matching values to pick the actual matching answers to her query.

The correctness of the BHE protocol can be easily seen based on Equation (2). In the last processing step, $D(V[i]) = D(Q'[i]^{VBK_i}) = D(Q'[i]) \cdot VBK_i = Q[i] \cdot VBK_i$. For the potentially matching buckets BK_i which the client specifies in the query formulation, $Q[i] = 1$, $D(V[i]) = Q[i] \cdot VBK_i = VBK_i = ToBytes(t(R_{pub})_{R_{pub}.B\in BK_i})$, where

[1]The number of bits in n is the number of bits of information that can be stored in one ciphertext.

$t(R_{pub})_{R_{pub}.B \in BK_i}$ is a superset of the query answers. For non-matching buckets BK_j, $Q[j] = 0$, $D(V[j]) = Q[j] \cdot VBK_j = 0$, because these data are not requested by the client.

C. Security Analysis

Given that Paillier's homomorphic encryption is semantically secure, we claim that for all probabilistic, polynomial time (PPT) adversaries, the above basic query processing protocol BHE is semantically secure, i.e., the advantage of the adversaries in breaking the security of BHE (by interpreting the content of a query and its answers) is negligible. Basically the adversaries cannot tell which entries in Q' are $E(1)$ and which are $E(0)$. Hence by examining the modular exponentiation result $V[i] = Q'[i]^{VBK_i}$ for each bucket, the adversaries have no idea as to which bucket has potential answers to the client's query. We provide a formal proof as follows. It follows the similar rationale of the proofs in [6], [27].

Theorem 4.1: Given that Paillier's homomorphic encryption is semantically secure, the proposed BHE protocol is semantically secure.

Proof: Suppose that there is an adversary A that can gain a non-negligible advantage ϵ in playing the game of breaking the security of BHE protocol. Then A can be used to gain an advantage in breaking the semantic security of Paillier's homomorphic encryption.

First we initiate a game with a challenger on behalf of Paillier's homomorphic encryption, simply called as Paillier challenger. The Paillier challenger sends us an integer n. We choose plaintexts $m_0, m_1 \in Z_n$ to be $m_0 = 0$ and $m_1 = 1$. After sending m_0, m_1 to the Paillier challenger, we receive $E(m_{\beta_1})$, in which β_1 is randomly generated by the Paillier challenger and not known to us.

Next we initiate the game of breaking the security of BHE protocol with A, in which we become the challenger. We send modulus n to A. A chooses two queries Q_0 and Q_1 and sends them to us. We flip a coin β_2 and construct the encrypted query vector Q'_{β_2} following the query formulation step in BHE. Then we replace all the entries of $E(1)$ with $E(m_{\beta_1})$ with re-randomization by multiplying each time by a new encryption of 0. With probability $\frac{1}{2}$, $\beta_1 = 0$, Q'_{β_2} is independent of β_2 and does not search for anything. With probability $\frac{1}{2}$, $\beta_1 = 1$ and Q'_{β_2} represents the query Q_{β_2}. After we give Q'_{beta_2} to A, A returns its guess β'_2. If $\beta'_2 = \beta_2$, we let our guess for the Paillier challenge be $\beta'_1 = 1$, otherwise we let $\beta'_1 = 0$.

Since A has ϵ advantage in breaking the security of BHE. If $\beta_1 = 1$, the probability of $\beta'_2 = \beta_2$ is $p(\beta'_2 = \beta_2) = \frac{1}{2} + \epsilon$. If $\beta_1 = 0$, $p(\beta'_2 = \beta_2) = \frac{1}{2}$, since β_2 is randomly chosen and it is completely independent of the choice of β'_2. Then the probability of our advantage in the Paillier challenge is as follows.

$$\begin{aligned} p(\beta'_1 = \beta_1) &= p(\beta'_1|\beta_1) \cdot \frac{1}{2} + p(\beta'_0|\beta_0) \cdot \frac{1}{2} \\ &= (\frac{1}{2} + \epsilon) \cdot \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{2} \\ &= \frac{1}{2} + \frac{\epsilon}{2} \end{aligned}$$

It contradicts the assumption that Paillier's homomorphic encryption is semantically secure. Therefore, the proposed BHE protocol is semantically secure and the advantage of A, ϵ, is negligible. ■

D. Discussion on Bucketization Trade-offs

Similar to the above analysis on costs, we consider equal size bucketization. Note that the data calculation with ciphertexts on the server in the second step of BHE and the data decryption on the client in the third step of BHE are computationally intensive, whereas the encryption of 0 and 1 of size b on the client in the first step of BHE is less computationally intensive. In the case of larger number of buckets b, the size of one bucket s is smaller, the chance for one bucket to be selected as a potentially matching bucket for a query is less, thus the amount of data decryption on the client is less. In contrast, when b is small and s is large, if a bucket is selected as a potentially matching bucket, all the data in the bucket have to be decrypted on the client. In addition, all the public data in the potentially matching buckets can be obtained by the client. However for large size of public data in practice, the server cannot use too fine bucketization, so b cannot be arbitrarily large. The advantage of small b though is to have small query vectors Q'.

V. Hybrid Private Query Processing

Although the basic private query processing protocol BHE achieves perfect privacy, its computation and communication costs are proportional to the size of the entire public data R_{pub}, making it still unsuitable in practice. We hence look for practical yet privacy-preserving alternatives. We note that in most cases, strong privacy rather than perfect privacy would suffice for most users, and trading off partial privacy could potentially reduce costs and improve performance. However, partial privacy must be defined and measured carefully, i.e. the simple model of k-Anonymity has been shown vulnerable to a number of attacks [10], [20].

In this section, we propose a new way of *hybrid* private query processing such that the client selects a subset of the public data buckets that cover the query buckets but also mimic the frequently co-accessed sets of buckets of other users, and then server computation and communication are only consumed on this subset of data buckets. Different from preious work, our selection of the subsets of the buckets *minimizes privacy risk under constraints,* is *dynamic* and can *protect query privacy for a sequence of queries in a session.*

A. Design Overview

Let *private bucket* be a user requested bucket in a query (*private query bucket*), or a bucket containing the user's private data (*private data bucket*); *decoy bucket* be a bucket other than private bucket in the subset of buckets that the client selects to reveal to the server. During our design of a hybrid private query processing solution, we reviewed and rejected three approaches of decoy bucket generation. The first one we rejected is using k random buckets or k continuous buckets [35]. As we discussed before, k-Anonymity is vulnerable, and moreover, these k-anonymous decoy buckets are likely to be inconsistent in a sequence of queries. The second approach we rejected is generating decoy buckets corresponding to the positive Laplace noise generated by differential privacy [9]. This approach incurs a large cost, which is proportional to the number of public data buckets. The third approach we chose not to use is plausibly deniable search (PDS) [24], [29], because we would like our solution to work for various data types instead of only texts, and we target to generate decoy buckets dynamically and to protect query privacy for a sequence of queries in a session.

To protect a user's privacy in a query sequence, we need *consistent* decoy buckets which appear in the user's queries *as likely as private buckets*. Our solution is therefore to *pre-generate super decoy buckets called cover buckets based on the user's private data before a query session starts, and then pick from the cover bucket set to generate decoy buckets for each subsequent query*. The maximum number of cover and decoy buckets *per* private bucket are specified by a user, and let them be η_c and η_d respectively. These two parameters bound the number of buckets on which the server needs to perform calculations as well as the amount of data transmission from the server to the client. η_c gives an upper bound on all queries in a user query session, while η_d gives an upper bound on one query to avoid the cases in which the number of buckets requested in a query is far less than the total number of private buckets but too many decoy buckets from the cover bucket set are still used in the query. Generally $\eta_c \leq \eta_d$ so that every cover bucket in the cover bucket set would be used as a decoy for a query bucket in some queries. To simplify parameter setting, a user can choose a η_d and set $\eta_c = \eta_d$.

Our goal is then to maximize privacy protection in generating cover buckets for private buckets subject to the constraint η_c and generating decoy buckets for each query subject to the constraint η_d. Observing that queries are highly likely to consist of repeatedly co-accessed sets of buckets, we leverage on frequently co-accessed bucket sets mined from historical query logs to define privacy. The intuition is that the bucket sets with higher access frequencies provide better privacy protection, as they are less likely to be linked to specific individuals. Hence, we generate cover buckets and decoy buckets by maximizing the chances that each private bucket is covered among the buckets of at least one frequently co-accessed bucket set pattern, e.g. BK_3 can be covered by buckets BK_2, BK_4 from a co-accessed bucket set pattern (BK_2, BK_3, BK_4). The more frequently co-accessed bucket set patterns contained in the query bucket sets and the more frequent they are, the better privacy protection. We call our solution of Hybrid privacy-preserving query processing outlined above as HHE, and describe it in details below.

B. Privacy Definition for a Query Session

Our privacy metric for protecting the private buckets follows the similar idea of the privacy metric in some privacy-preserving data mining literatures [4], mutual information between a priori entropy and a posteriori entropy. Suppose that adversaries have an *a priori* belief, H_{pri}, and an *a posteriori* belief, H_{pos}, on a user's private data and queries in a query session, before and after they observe the actual query session. Perfect privacy is achieved when $H_{pri} = H_{pos}$, as in BHE and cPIR [19]. In HHE we try to minimize $|H_{pri} - H_{pos}|$.

When defining metrics for H_{pri} and H_{pos} that can capture the characteristics of queries, we explored and rejected several possible metric definitions. We first note that a metric using the number of buckets requested in a query, e.g. k-Anonymity, would be highly biased towards adding a large amount of decoy buckets and does not capture the semantic of the query, i.e. what buckets requested. If considering the actual buckets requested by a query in the metric, for example using the sum of individual buckets' frequencies as the metric, the buckets with the highest access frequencies would probably be chosen as decoy buckets in every query. A simple analysis on multiple queries would be able to single out these decoy buckets and expose private buckets.

We observe that in an online data service with a large number of user sessions, queries are highly likely to have repeatedly co-accessed sets of buckets due to the nature of skewed query accesses. Assume that there exists a historical query log HL with a large number of query sessions of different users, in which a private query is represented by the buckets requested in query formulation step of BHE. For example, the example join query in Section IV which requests BK_1, BK_2, BK_3 and BK_6 is represented as $(1, 2, 3, 6)$. We thus rely on *frequently co-accessed sets of buckets, FBS*, mined from HL, to define H_{pri} and H_{pos}.

However, it is not practical to obtain a historical query log HL from a large number of users in the first place, since historical query buckets are usually private information of users. On the other hand, the frequently co-accessed sets of buckets FBS are aggregate information extracted from HL, thus are much less sensitive, and could be shared among all benign users after we remove low frequency bucket sets. We therefore propose to run privacy-preserving distributed frequent pattern mining [18] on the private query histories of participant users, so that FBS can be mined privately from a virtual global query log HL that consists of private query histories of individual users. A user's private query history can be built when using BHE for processing private queries. To prevent adversaries to masquerade as normal users and to tempt or steal the mining result FBS, we can have a certification server to authenticate identities of participant users. After constructing FBS, the less expensive protocol HHE would be able to run as presented in Sections V-C and V-D. Fig. 2 illustrates the above flow process of FBS.

We call a frequently co-accessed set of buckets as a *query pattern*, call the frequency it appears in HL as *support*, represented as sup as in frequent pattern mining [14]. To account for individual effects of the patterns, we are only interested in *closed* query patterns, which do not have supersets with the same supports as their supports [14]. For example, if $(1, 6)$ only appears in $(1, 2, 3, 6)$, it is not a closed pattern and will not be considered. Figure 3 depicts an example of a query log HL and the corresponding query pattern set FBS.

An adversary's *a priori* belief ideally consists of all the possible query patterns. After the adversary observes a query

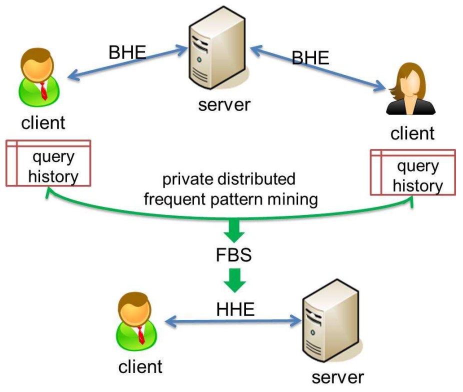
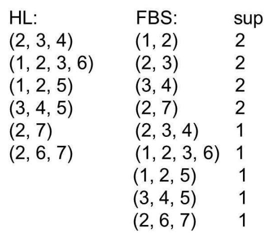


Fig. 2. The Generation and Usage of FBS



HL:	FBS:	sup
(2, 3, 4)	(1, 2)	2
(1, 2, 3, 6)	(2, 3)	2
(1, 2, 5)	(3, 4)	2
(3, 4, 5)	(2, 7)	2
(2, 7)	(2, 3, 4)	1
(2, 6, 7)	(1, 2, 3, 6)	1
	(1, 2, 5)	1
	(3, 4, 5)	1
	(2, 6, 7)	1

Fig. 3. An Example Virtual Historical Query Log HL and Closed Query Patterns FBS

session QS, his *a posteriori* belief only contains the query patterns that appear in QS. Without exact knowledge of the probabilities of all possible query patterns, we approximate the adversaries' *a priori* belief and *a posteriori* belief using the query patterns in FBS, such that the *approximate a priori belief* consists of all query patterns in FBS, and the *approximate a posteriori belief* only contains the patterns of FBS that appear in a query session QS, denoted as $FBS(QS)$. The more user query sessions HL has, the more representative and accurate are the approximate *a priori* and *a posteriori* beliefs. Let the *approximate probability* of a query pattern i be $\widehat{p(i)} = \frac{sup_i}{\sum_{j \in FBS} sup_j}$. Similar to [4], we define the approximate *a priori* belief $\widehat{H_{pri}}$ and *a posteriori* belief $\widehat{H_{pos}}$ based on the entropies of the contained query patterns.

$$\widehat{H_{pri}} = -\sum_{i \in FBS} \widehat{p(i)} \cdot log_2(\widehat{p(i)}) \tag{3}$$

$$\widehat{H_{pos}}(QS) = -\sum_{i \in FBS(QS)} \widehat{p(i)} \cdot log_2(\widehat{p(i)}) \tag{4}$$

For the FBS in Figure 3, $\widehat{H_{pri}} = -4 \cdot \frac{2}{4 \cdot 2 + 5 \cdot 1} \cdot log_2(\frac{2}{4 \cdot 2 + 5 \cdot 1}) - 5 \cdot \frac{1}{4 \cdot 2 + 5 \cdot 1} \cdot log_2(\frac{1}{4 \cdot 2 + 5 \cdot 1}) = 3.085$. Given a query session $QS_1 = \{(2,3,4),(1,2,3,5,6),(1,2,5)\}$, $FBS(QS_1) = \{$(2, 3), (3, 4), (1, 2), (2, 3, 4), (1, 2, 3, 6), (1, 2, 5)$\}$, $\widehat{H_{pos}}(QS_1) = -3 \cdot \frac{2}{4 \cdot 2+5 \cdot 1} \cdot log_2(\frac{2}{4 \cdot 2+5 \cdot 1}) - 3 \cdot \frac{1}{4 \cdot 2+5 \cdot 1} \cdot log_2(\frac{1}{4 \cdot 2+5 \cdot 1}) = 2.1$.

Based on $\widehat{H_{pri}}$ and $\widehat{H_{pos}}$, we define the risk of privacy disclosure in a session QS as

$$risk(QS) = \frac{\widehat{H_{pri}} - \widehat{H_{pos}}(QS)}{\widehat{H_{pri}}} \tag{5}$$

For the above example query session QS_1, $risk(QS_1) = \frac{\widehat{H_{pri}} - \widehat{H_{pos}}(QS_1)}{\widehat{H_{pri}}} = \frac{3.085-2.1}{3.085} = 0.319$. Since a QS has less query patterns than a HL has, $\widehat{H_{pri}} > \widehat{H_{pos}}(QS)$, $0 \leq risk(QS) \leq 1$. If a QS contains more query patterns in FBS and these patterns are more frequent, according to (4), $\widehat{H_{pos}}(QS)$ is larger, and $risk(QS)$ is smaller. The above definition applies to BHE and PIR as well: a query in BHE or PIR requests all public data buckets, thus all query patterns in FBS are covered in QS, $\widehat{H_{pos}}(QS) = \widehat{H_{pri}}$ and $risk(QS) = 0$, which is perfect privacy.

C. Generating Cover and Decoy Buckets

To minimize $risk(QS)$ according to Equation (5), the client needs to select more and frequent patterns from FBS based on which cover and decoy buckets can be generated for private data buckets and private query buckets. We first discuss how to select patterns to generate cover buckets, i.e. to use non-private buckets in these patterns as cover buckets, suject to the constraint η_c. The generation of decoy buckets is similar and is discussed later.

Let *PI pattern* be a pattern that includes at least one private bucket, and *NPI pattern* be a pattern that does not include any private bucket. We say that *only after all PI patterns are considered and the number of cover buckets generated is still less than η_c, are the NPI patterns examined*. This is to avoid the cases that an NPI pattern with an extremely high support could use all η_c and may not leave enough budget to use buckets in PI patterns. Such high support NPI patterns alone are not recommended to be cover buckets, as they could be easily identified by adversaries.

These candidate patterns (PI or NPI) are picked in a way that the union of non-private buckets in these patterns satisfy the constraint η_c, and the total support of all involved patterns (candidate patterns and their sub-patterns) is maximized, thus maximizing $\widehat{H_{pos}}(QS)$ and minimizing $risk(QS)$. To maximize the total support of all involved patterns, *the supports of a pattern and its sub-patterns which appear in FBS should be considered together in pattern selection*. Thus we define the sum of the support of a pattern and the supports of its sub-patterns as *super support*, denoted as $ssup$. For example, as pattern $(1,2,3,6)$ with $sup = 1$ contains sub-patterns $(1,2)$ with $sup = 2$ and $(2,3)$ with $sup = 2$, $ssup(1,2,3,6) = sup(1,2,3,6) + sup(1,2) + sup(2,3) = 1 + 2 + 2 = 5$.

To quickly get sub-patterns and the support support of a pattern, we build a mapping list of sub-patterns for each query pattern. An example of sub-pattern mapping list and super supports for the patterns in FBS of Fig. 3 are shown in Fig. 4(a). To quickly select a PI pattern for a private bucket, we build an inverted bucket-to-patterns list $IBList$. An example $IBList$ for the FBS of Fig. 3 is shown in Fig. 4(b), where a colon separates a data bucket and a list of patterns that cover the bucket. These structures along with FBS are replicated on each server certified benign client so that the client can make its own decision to generate cover and decoy buckets.

In the following, we discuss selecting PI patterns and NPI patterns for generating cover buckets separately, and then

FBS:	sup	sub-patterns	ssup
(1, 2)	2	null	2
(2, 3)	2	null	2
(3, 4)	2	null	2
(2, 7)	2	null	2
(2, 3, 4)	1	(2, 3), (3, 4)	5
(1, 2, 3, 6)	1	(1, 2), (2, 3)	5
(1, 2, 5)	1	(1, 2)	3
(3, 4, 5)	1	(3, 4)	3
(2, 6, 7)	1	(2, 7)	3

(a) Closed Query Patterns FBS

IBList:
1 : (1, 2, 3, 6), (1, 2, 5), (1, 2)
2 : (2, 3, 4), (1, 2, 3, 6), (1, 2, 5), (1, 2), (2, 3), (2, 7)
3 : (2, 3, 4), (1, 2, 3, 6), (3, 4, 5), (2, 3), (3, 4)
4 : (2, 3, 4), (3, 4, 5), (3, 4)
5 : (1, 2, 5), (3, 4, 5)
6 : (1, 2, 3, 6), (2, 6, 7)
7 : (2, 6, 7), (2, 7)

(b) Inverted Bucket-Pattern List $IBList$ for FBS

Fig. 4. Example Closed Query Patterns FBS and Index Structures for Selecting Cover/Decoy Buckets

Algorithm 1 maximize_PI_ssup$(PI_candidates, BK(R_{priv}.A), \eta_c)$

```
Require: $|FB_i - BK(R_{priv.A})| > 0$. If $FB_i \subseteq BK(R_{priv.A})$, set
    $|FB_i - BK(R_{priv.A})| \leftarrow 0.001$.
Require: $PI\_scb$, the set of cover buckets from selected PI patterns
    in the maximization of (6).
Require: $total\_ssup$, the total support of all unique patterns that
    only consist of buckets in $PI\_scb$ and $BK(R_{priv}.A)$.
1: Sort $PI\_candidates$ in non-increasing order of $\frac{ssup(FB_i)}{|FB_i - BK(R_{priv.A})|}$ ($FB_i \in PI\_candidates$).
2: while $|PI\_scb \bigcup (FB_i - BK(R_{priv.A}) - PI\_scb)| \leq |BK(R_{priv}.A)| \cdot \eta_c$ do
3:     Greedily add the buckets in $FB_i - BK(R_{priv.A}) - PI\_scb$
       to $PI\_scb$ and update $total\_ssup$ using $FB_i$ and the sub-
       patterns of $FB_i$ that are not in previously selected patterns.
4: end while
5: if $total\_ssup < ssup(FB_i)$ then
6:     $PI\_scb \leftarrow FB_i - BK(R_{priv.A})$.
7:     $total\_ssup \leftarrow ssup(FB_i)$.
8: end if
9: Return $PI\_scb$.
```

briefly discuss generating decoy buckets, which follows the similar rational as generating cover buckets.

1) Selecting PI patterns for Generating Cover Buckets: A client first considers PI patterns for generating cover buckets. Denote a query pattern from FBS as FB, the private data buckets as $BK(R_{priv}.A)$. Let the set of PI patterns be $IBList(BK(R_{priv}.A))$. For example, the PI patterns for private buckets $BK(R_{priv}.A) = (3,5)$ are $IBList(BK(R_{priv}.A)) = \{$(2, 3, 4), (1, 2, 3, 6), (3, 4, 5), (1, 2, 5), (2, 3), (3, 4)$\}$. Let operator || applied to a pattern be the number of buckets in the pattern. The client first selects PI patterns for which $|FB_i - BK(R_{priv}.A)| \leq |BK(R_{priv}.A)| \cdot \eta_c$ $(FB_i \in IBList(BK(R_{priv}.A)))$. Denote this set of PI patterns as $PI\_candidates$. Given $BK(R_{priv}.A) = (3,5)$ and $\eta_c = 2$, $PI\_candidates = IBList(BK(R_{priv}.A)) = \{$(2, 3, 4), (1, 2, 3, 6), (3, 4, 5), (1, 2, 5), (2, 3), (3, 4)$\}$.

Let the super support operator $ssup$ applied to a bucket set be the total supports of its all contained patterns, and the set operations such as $\bigcup, -$ to be be applicable to bucket sets and bucket sets of patterns. We formalize the problem of selecting patterns for generating cover buckets to maximize the total support of all involved patterns as follows, where $x_i = 1, 0$ denotes if a candidate pattern is selected or not.

$$maximize\ ssup(\bigcup_i FB_i \cdot x_i). \tag{6}$$

$$\begin{aligned} subject\ to \quad & |\bigcup_i (FB_i - BK(R_{priv}.A)) \cdot x_i| \leq |BK(R_{priv}.A)| \cdot \eta_c. \\ \exists j_1, ..., j_h, \quad & (FB_{j_1} \bigcup FB_{j_2} ... \bigcup FB_{j_h}) \bigcap BK(R_{priv}.A) \neq \emptyset \\ where \quad & i, j_1, ... j_h \in |PI\_candidates|.. \end{aligned}$$

In the maximization, we consider the supports of all involved patterns in the union of selected PI patterns. Note that two PI patterns could have overlapping sub-patterns, only the support of a pattern / sub-pattern that is not included in a previously selected PI pattern can be added up. For example, if $x_0 = 1, x_1 = 1$, and assume patterns $(2,3,4)$ and $(1,2,3,6)$ are consecutively selected, since $ssup(2,3,4) = sup(2,3) + sup(3,4) + sup(2,3,4) = 2 + 2 + 1 = 5$, we can only add the supports of patterns $(1,2)$ and $(1,2,3,6)$ but not $(2,3)$ for PI pattern $(1,2,3,6)$, $ssup((2,3,4) \bigcup (1,2,3,6)) = (sup(2,3) + sup(3,4) + sup(2,3,4)) + (sup(1,2) + sup(1,2,3,6)) = 5 + (2 + 1) = 8$. Similarly in the first constraint, when counting the total number of cover buckets in the union of selected PI patterns, only the buckets that are not included by previously selected PI patterns can be counted. For the previous example, $|FB_0 - BK(R_{priv}.A)| = |(2,3,4) - (3,5)| = |(2,4)| = 2$, we can only count $(1,6)$ but not including 2 for the next pattern $(1,2,3,6)$, thus $(FB_0 - BK(R_{priv}.A)) \bigcup (FB_1 - BK(R_{priv}.A))| = |(2,4) \bigcup ((1,2,3,6) - (3,5))| = |(2,4) \bigcup (1,6)| = 2 + 2 = 4$.

The above maximization problem (6) is analogous to a typical NP-complete problem, the 0-1 Knapsack problem [17]. The difference between them is that the operations in (6) are based on the union of sets, while the operations in the 0-1 Knapsack problem are based on accumulation of independent objects. Considering super supports as values of objects and the number of buckets as the total weight of objects, if the PI patterns are disjoint with each other, (6) becomes a standard 0-1 Knapsack problem. Hence, (6) is also an NP-complete problem, and has a fast 2-approximate solution similar to the fast approximation algorithm that gives a 2-approximate solution to the 0-1 Knapsack problem [17]. This solution is described in Algorithm 1.

The time complexity of this algorithm is $O(|PI\_candidates| \cdot log_2|PI\_candidates|)$. The reason that this algorithm is 2-approximate is as follows. Since our solution is suboptimal, we must have some leftover bucket budget Δ at the end. Suppose that Algorithm 1 can take partial buckets from a pattern. Let the optimal total support be OPT. Then by adding $\frac{\Delta}{|FB_j - BK(R_{priv.A})|} ssup(FB_j)$ to $total\_ssup$, we would either match or exceed OPT. Therefore, either $ssup(\bigcup_{j=1}^{i-1} FB_j) \geq \frac{1}{2}OPT$ or $ssup(FB_i) \geq \frac{\Delta}{|FB_j - BK(R_{priv.A})|} ssup(FB_j) \geq \frac{1}{2}OPT$.

As an example of running Algorithm 1, given $BK(R_{priv}.A) = (3,5)$, $\eta_c = 2$, $PI\_candidates = \{$(2, 3, 4), (1, 2, 3, 6), (3, 4, 5), (1, 2, 5), (2, 3), (3, 4)$\}$, $|BK(R_{priv}.A)| \cdot \eta_c$ = 4. $\frac{ssup(FB_i)}{|FB_i - BK(R_{priv.A})|}$ for the sequential $FB_i \in PI\_candidates$ are $\frac{5}{2}, \frac{5}{3}, \frac{3}{2}, \frac{3}{2}, 2, 2$. Re-ordered $PI\_candidates$ are thus $\{$(2, 3, 4), (2, 3), (3, 4), (1, 2, 3, 6), (3, 4, 5), (1, 2, 5)$\}$. Going over re-ordered $PI\_candidates$ and generating cover buckets from these patterns until the end of $PI\_candidates$, the cover bucket budget $|BK(R_{priv}.A)| \cdot \eta_c$ is not exceeded, so $PI\_scb = (1, 2, 4, 6)$, $total\_ssup$ = $ssup(PI\_scb)$ = $sup(2, 3)$ + $sup(3, 4)$ + $sup(2, 3, 4)$ + $sup(1, 2)$ + $sup(1, 2, 3, 6)$ + $sup(3, 4, 5)$ + $sup(1, 2, 5)$ = 10. If $\eta_c = 1$, $|BK(R_{priv}.A)| \cdot \eta_c$ = 2, re-ordered $PI\_candidates = \{$(2, 3, 4), (2, 3), (3, 4), (3, 4, 5), (1, 2, 5)$\}$. At the fifth pattern $(1, 2, 5)$, $PI\_scb = (2, 4)$ and bucket 1 cannot be added, the client thus compares the current $total\_ssup$ = $sup(2, 3)$ + $sup(3, 4)$ + $sup(2, 3, 4)$ + $sup(3, 4, 5)$ = 6 with $ssup(1, 2, 5) = 5$, and keeps the current $PI\_scb = (2, 4)$. The results PI_scb in the above two examples happen to be the same as optimal solutions.

2) Selecting NPI patterns for Generating Cover Buckets: If the maximization goal (6) in PI patterns is already achieved, and yet the budget of cover buckets, $|BK(R_{priv}.A)| \cdot \eta_c$ is not all used, the client selects NPI patterns from the set $FBS - IBList(BK(R_{priv}.A))$ to generate more cover buckets which are different from the cover buckets generated from PI patterns. These NPI patterns are selected such that $|FB_i - PI\_scb| \leq |BK(R_{priv}.A)| \cdot \eta_c - |PI\_scb|$ ($FB_i \in (FBS - IBList(BK(R_{priv}.A)))$). Denote this set of NPI patterns as $NPI\_candidates$.

In usual cases, FBS is representative enough that $IBList(BK(R_{priv}.A))$ is not empty and $PI\_candidates$ exist. In some cases when no PI patterns exist for private data buckets, $IBList(BK(R_{priv}.A)) = \varnothing$ and $PI\_scb = \varnothing$, the client randomly samples a subset of NPI patterns for which $|FB_i| \leq |BK(R_{priv}.A)| \cdot \eta_c$ ($FB_i \in FBS$), and sort them by descending super supports. This random sampling is to avoid the buckets of most frequent patterns to be always selected as cover buckets, which is risky when $PI\_scb = \varnothing$. The maximization goal during selecting NPI patterns, similar to the maximization goal in selecting PI patterns (6), is as follows.

$$maximize\ ssup(\bigcup_i FB_i \cdot x_i). \tag{7}$$

$$\begin{aligned} subject\ to \quad & |PI\_scb \bigcup (\bigcup_i FB_i \cdot x_i)| \leq |BK(R_{priv}.A)| \cdot \eta_c \\ where \quad & i \in |NPI\_candidates|. \end{aligned}$$

Similarly, there exists a 2-approximate algorithm to solve the maximization problem (7), which is described in Algorithm 2.

Consider the previous example, given $BK(R_{priv}.A) = (3, 5)$, if $\eta_c = 3$, the solution $PI\_scb = (1, 2, 4, 6)$ for $PI\_candidates$ would not use all cover bucket budget $|BK(R_{priv}.A)| \cdot \eta_c = 6$. Then the client considers $NPI\_candidates$ = $\{$(2, 6, 7), (1, 2), (2, 7)$\}$ and re-sorts them by $\frac{ssup(FB_i \bigcup PI\_scb) - ssup(PI\_scb)}{|FB_i - PI\_scb|}$ as $\{$(2, 6, 7), (2, 7), (1, 2)$\}$. Going over re-ordered $NPI\_candidates$ and generating cover buckets from these patterns until the end of $NPI\_candidates$, the cover bucket budget $|BK(R_{priv}.A)| \cdot \eta_c$ is not exceeded, so $CBS = (1, 2, 3, 4, 5, 6, 7)$, $total\_ssup$ = $ssup(PI\_scb)$ + $sup(2, 7)$ + $sup(2, 6, 7)$ = 10 + 2 + 1 = 13.

The above process of selecting PI and NPI patterns for generating cover buckets in outlined Algorithm 3. The time complexity of this algorithm is $O(max($ $|FBS|$, $|PI\_candidates| \cdot log_2|PI\_candidates|$, $|NPI\_candidates| \cdot log_2|NPI\_candidates|$)).

Algorithm 2 maximize_NPI_ssup($NPI\_candidates$, PI_scb, $ssup(PI\_scb)$, $BK(R_{priv}.A)$, η_c)

```
Require: $|FB_i - PI\_scb| > 0$. If $FB_i \subseteq PI\_scb$, set $|FB_i - PI\_scb| \leftarrow 0.001$.
Require: $CBS$, the set of buckets from selected PI and NPI patterns in the maximization of (6) and (7).
Require: $total\_ssup$, the total support of all unique patterns that only consist of buckets in $CBS$.
 1: Sort $NPI\_candidates$ in non-increasing order of $\frac{ssup(FB_i \bigcup PI\_scb) - ssup(PI\_scb)}{|FB_i - PI\_scb|}$ ($FB_i \in NPI\_candidates$).
 2: $CBS \leftarrow PI\_scb$.
 3: $total\_ssup \leftarrow ssup(PI\_scb)$.
 4: while $|CBS \bigcup (FB_i - CBS)| \leq |BK(R_{priv}.A)| \cdot \eta_c$ do
 5:     Greedily add the buckets in $FB_i - CBS$ to $CBS$ and update $total\_ssup$ using $FB_i$ and the sub-patterns of $FB_i$ that are not in previously selected patterns.
 6: end while
 7: if $total\_ssup < ssup(PI\_scb \bigcup FB_i)$ then
 8:     $CBS \leftarrow PI\_scb \bigcup FB_i$.
 9:     $total\_ssup \leftarrow ssup(PI\_scb \bigcup FB_i)$.
10: end if
11: Add private buckets $BK(R_{priv}.A)$ to $CBS$ and return $CBS$.
```

Algorithm 3 generate_cover_buckets($BK(R_{priv}.A), \eta_c$)

```
 1: Calculate $IBList(BK(R_{priv}.A))$.
 2: if $IBList(BK(R_{priv}.A)) \neq \varnothing$ then
 3:     Select PI candidate patterns $PI\_candidates$ from $IBList(BK(R_{priv}.A))$, s.t. $|FB_i - BK(R_{priv}.A)| \leq |BK(R_{priv}.A)| \cdot \eta_c$ ($FB_i \in IBList(BK(R_{priv}.A))$).
 4:     $PI\_scb \leftarrow maximize\_PI\_ssup(PI\_candidates, BK(R_{priv}.A), \eta_c$
 5:     if $|PI\_scb| < |BK(R_{priv}.A)| \cdot \eta_c$ then
 6:         Select NPI candidate patterns $NPI\_candidates$ from $FBS - IBList(BK(R_{priv}.A))$, s.t. $|FB_i - PI\_scb| \leq |BK(R_{priv}.A)| \cdot \eta_c - |PI\_scb|$ ($FB_i \in (FBS - IBList(BK(R_{priv}.A)))$).
 7:     else
 8:         $CBS \leftarrow PI\_scb \bigcup BK(R_{priv}.A)$.
 9:     end if
10: else
11:     Randomly sample NPI candidate patterns $NPI\_candidates$ from $FBS$, s.t. $|FB_i| \leq |BK(R_{priv}.A)| \cdot \eta_c$ ($FB_i \in FBS$).
12: end if
13: if $NPI\_candidates \neq \varnothing$ then
14:     $CBS \leftarrow maximize\_NPI\_ssup$($NPI\_candidates$, $PI\_scb$, $ssup(PI\_scb)$, $BK(R_{priv}.A)$, $\eta_c$).
15: end if
```

3) Generating Decoy Buckets: Let the generated cover bucket set including the private buckets $BK(R_{priv}.A)$ be CBS. Generating decoy buckets is analogous to generating cover buckets, except that candidate patterns must consist of only buckets from CBS, the private buckets $BK(R_{priv}.A)$ are replaced by the private query buckets $BK(q)$ for a query q, and η_c is replaced by η_d in the inputs of Algorithms 1 and 2. We show how to generate decoy buckets in Algorithm 4.

Let $FBS(CBS)$ be the set of query patterns that only consist of buckets from CBS. It can be easily obtained by running Algorithm 3 and then materialized to use in Algorithm 4. For example, given $BK(R_{priv}.A) = (3, 5)$, $\eta_c = 2$, then $CBS = (1, 2, 3, 4, 5, 6)$, $FBS(CBS) = \{$(1, 2, 3, 6), (2, 3, 4), (1, 2, 5), (3, 4, 5), (1, 2), (2, 3), (3, 4)$\}$.

```
Algorithm 4 generate_decoy_buckets($BK(q), \eta_d$)
Require: $IBList(BK(q))$, the set of PI patterns for private query $q$.
Require: $FBS(CBS)$, the set of query patterns that only consist of buckets from $CBS$.
Require: $DBS$, the generated decoy bucket set for $BK(q)$ including the private query buckets $BK(q)$.
 1: Calculate $IBList(BK(q)) \leftarrow IBList(BK(q)) \bigcap FBS(CBS)$.
 2: if $IBList(BK(q)) \neq \varnothing$ then
 3:     Select PI candidate patterns $PI\_candidates$ from $IBList(BK(q))$, s.t. $|FB_i - BK(q)| \leq |BK(q)| \cdot \eta_d$ $(FB_i \in IBList(BK(q)))$.
 4:     $PI\_scb \leftarrow maximize\_PI\_ssup(PI\_candidates, BK(q), \eta_d)$
 5:     if $|PI\_scb| < |BK(q)| \cdot \eta_d$ then
 6:         Select NPI candidate patterns $NPI\_candidates$ from $FBS(CBS) - IBList(BK(q))$, s.t. $|FB_i - PI\_scb| \leq |BK(q)| \cdot \eta_d - |PI\_scb|$ $(FB_i \in (FBS(CBS) - IBList(BK(q)))$.
 7:     else
 8:         $DBS \leftarrow PI\_scb \bigcup BK(q)$.
 9:     end if
10: else
11:     Randomly sample NPI candidate patterns $NPI\_candidates$ from $FBS(CBS)$, s.t. $|FB_i| \leq |BK(q)| \cdot \eta_d$ $(FB_i \in FBS(CBS))$.
12: end if
13: if $NPI\_candidates \neq \varnothing$ then
14:     $DBS \leftarrow maximize\_NPI\_ssup(NPI\_candidates, PI\_scb, ssup(PI\_scb), BK(q), \eta_d)$.
15: end if
```

According to Algorithm 1, if $BK(q) = (3)$, $\eta_d = 2$, re-ordered $PI\_candidates = \{(2, 3, 4), (2, 3), (3, 4), (1, 2, 3, 6), (3, 4, 5)\}$. From the first three patterns, $PI\_scb = (2, 4)$ and decoy buckets are BK_2, BK_4. If η_d is changed to 4, decoy buckets would be generated from the first four patterns up to $(1, 2, 3, 6)$, $PI\_scb = (1, 2, 4, 6)$, $DBS = (1, 2, 3, 4, 6)$ and the total support is 8.

D. Hybrid Query Processing Protocol

We build our hybrid private query processing protocol HHE based on the basic protocol BHE presented in Section IV and cover / decoy buckets generation algorithms described above. As BHE, HHE relies on public data bucketization and requires the preliminary steps of bucket summary dissemination and public key exchange. HHE also follows three steps: client query formulation, query processing on the server and finally query processing on the client. In contrast to BHE, HHE requires the client to generate cover buckets before processing any queries, and during the processing of a query, HHE requires the client to generate decoy buckets in the query formulation step, the server to compute only on the data belonging to private and decoy buckets in the query processing step on the server, and the server to only send to the client the data in private and decoy buckets. We illustrate HHE in Fig. 5.

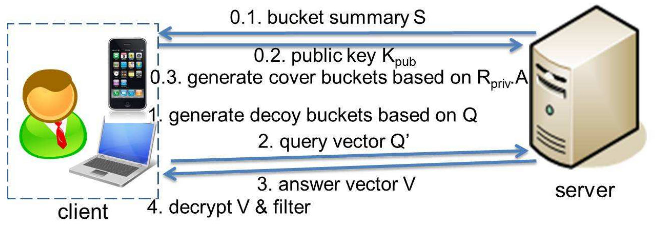


Fig. 5. HHE. In this protocol, before processing any queries, 0.1) Server sends the bucket summary S of its database to the client; 0.2) Client sends her public key K_{pub} to the server; 0.3) Client identifies the private data buckets that potentially include her data $R_{priv}.A$ based on S, and generates a set of cover buckets. Then to process a query q, 1) Client identifies the private query buckets that potentially include her requested data in q based on S, and generates decoy buckets from the cover bucket set. 2) Client formulates a query vector Q' that encrypts private and decoy bucket entries as $E(1)$ and $E(0)$ respectively, and sets other bucket entries as plaintext 0, and sends Q' to the server; 3) Server does blind processing on non-zero entries of Q' and its database, sends the answer vector V back to the client; 4) Finally, the client decrypts V and reconstructs the answer to the query q.

E. Security Analysis

We claim that (1) HHE protects a user's query privacy for a sequence of queries in a user session; (2) The privacy guarantee of HHE for a given query session QS is evaluated by $risk(QS)$ defined in Equation (5), the smaller $risk(QS)$ is, the better privacy protection is.

First, because cover buckets are generated when a session begins, and decoy buckets for each query in QS are generated upon cover buckets, decoy buckets appear consistently like private data buckets in the session. Thus HHE is not subject to attacks that single out inconsistent noises in multiple queries to find private data, such as composition attacks on k-Anonymity [10]. Based on Theorem 4.1, private buckets are semantically secure among decoy buckets. Therefore, we say that HHE protects a user's query privacy for a sequence of queries in a user session, although its privacy gurantee is not perfect, which is discussed below.

Second, privacy leak in HHE comes from the fact that only partial buckets (private and decoy buckets) are requested. Following Algorithms 3 and 4, any private bucket would be covered by at least a PI or an NPI pattern. If a private bucket is covered by a PI pattern, it is part of a frequent query pattern, so the reformulated query is not unique and is not likely to be linked to a specific individual. If a private bucket is covered by an NPI pattern, since we assume that adversaries do not know FBS (this information is secure with clients), although the adversaries may try to perform frequent pattern mining on observed query buckets (private and decoy buckets), they still cannot tell that the private bucket is not part of a real pattern, again the reformulate query looks not unique and is not likely to be linked to a specific individual. The link probability depends on how many patterns are included in the query buckets set and the supports of these patterns, which are quantified by $risk(QS)$.

Finally, we would like to emphasize that HHE is not subject to the common attacks on k-Anonymity such as similarity and skewness attacks [20]. The reason is two folds: 1) HHE protects privacy in a coaser granularity level, buckets instead of data values, and the contents of the reformulated query buckets are completely confidential guaranteed by homomorphic encryption; 2) special characteristics of a query which might be privacy concerns such as unique similarity and skewness would be significantly diminished by using frequent patterns in HHE.

VI. Experimental Evaluation

Our evaluation focuses on the following: (1) the performance, i.e. client query formulation time, server processing time, client post-processing time, query completion time and data communication size, of our proposed solutions BHE, HHE compared to those of PIR; (2) the effects of public data size and query selectivity / query size on performance; (3) the performance trade-offs of different granularities of bucketizations; (4) privacy risk of HHE, and the percentage of private buckets fully covered by frequent query patterns.

A. Implementation and Experiments Setup

Implementation. We used C and relied on an open source Paillier library [1] to implement BHE and HHE. We also implemented the classical single-server cPIR protocol [19] which we denote as cPIR, and adapted the CPU implementation of the currently known fastest single-server PIR solution [21] which we denote as lPIR. To see how much privacy risk HHE can save compared to anonymization approaches that do not consider data and query semantics in generating decoys, we implemented a simple hybrid protocol that selects random buckets as decoys but also obeys the η_c and η_d constraints of HHE, which we refer to as Random. We simulated data transmission time by dividing the communication sizes with 100Mbps and then adding the latency 20ms for simulating queuing delay and round trip time.

Data Set. We used both a synthetic data set and a real data set as the public data R_{pub}. The synthetic data set has between 100K (10^5) to 10M (10^7) tuples (we use 1M data set as default). Each tuple has three integer attributes whose values are uniformly distributed in [0, 10^7]. The first attribute was used as the query matching key $R_{pub}.B$. Data bucketization was done on this attribute by equally distributing ordered tuples in buckets. Three granularities of bucketizations were generated, 10000 buckets, 1000 buckets and 100 buckets respectively (we use 10000 buckets as default). 1000 matching keys of private data $R_{priv}.A$ were generated within a range of $R_{pub}.B$, which is between 5% to 100% of the $R_{pub}.B$ domain (default is 10%). These settings are also shown in Table I. The synthetic data set is used as the default data set for running experiments.

The real data set consists of US medical providers from the National Plan and Provider Enumeration System. We extracted 6 attributes (including both numeric and text attributes) and selected only doctors from this data, resulting in 2,534,461 tuples. We call the result data set as NPI data. The first attribute of this data set, provider ID, was used as the matching key of public data. It was bucketized in 8,449 buckets with each bucket having at most 300 tuples. Assuming that there are 500 cities in US and a user is mainly interested in medical providers in a resident city, we generated 1000 private keys (on provider IDs) of a user session from a range of 0.2% of the entire NPI data.

TABLE I
Experimental Parameters

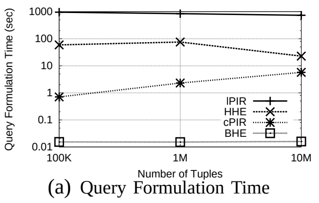
Parameter	Domain	Default
Number of Tuples	$100K, 1M, 10M$	$1M$
Raw Query Size	50, 100, 200, 400	100
Number of Buckets b	100, 1000, 10000	10000
Decoy Buckets Constraint η_c, η_d	5, 10, 15, 20	10
Private Data Distribution	5%, 10%, 50%, 100%	10%

Experiments Setup. The ciphertext key size in cPIR is 1024 bits. The block size in lPIR is 10 buckets. Queries were generated on $R_{priv}.A$ following Zipf distribution of skewness 0.8. A join query selected between 50 to 400 random keys from $R_{priv}.A$, while a range query selected a random range that covered between 50 to 400 consecutive $R_{priv}.A$ keys (default is 100 keys). A user query session has three join and two range queries. The private keys $R_{priv}.A$ of each user session were generated separately. The frequent query bucket patterns set FBS in HHE was mined from 200 user sessions and 1000 queries, such that each pattern appears in at least five queries and two user sessions. The resulting size of FBS, depending on the number of public data tuples, is around 20,000. The number of cover / decoy buckets per private bucket η_c and η_d is between 5 and 20 (default is 10) as shown in Table I. A test run consists of 20 user sessions and 100 queries. The reported results of each run were averaged over these 100 queries. Each of our experiments was run on a Unix server with 2 quad-core Intel Nehalem 2.4 GHz processors and 24 GB memory.

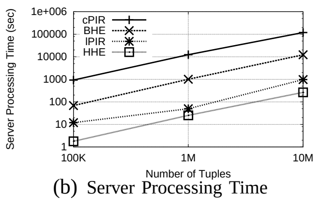
B. Experimental Results

Varying the Number of Public Tuples. We first study the scalability of our proposed solutions BHE and HHE, i.e. the effect of increasing the number of public tuples on query performance. Here the number of public tuples is varied from 100K to 10M, while fixing the other parameters as default. The y-axis in Fig. 6 is in logarithmic scale. As seen from Fig. 6, HHE performs the best overall, although it spends extra time on selecting patterns and generating decoy buckets compared to BHE, as seen in Fig. 6(a); BHE is no better than lPIR in processing time, but has a smaller communication size than lPIR. We also confirm that lPIR is much faster than cPIR, but we want to emphasize that the security of lPIR is not guaranteed as strong as cPIR or BHE, both of which base on number theory. HHE can answer a join or range query involving considerable amount of data within 2 minutes on 1M public tuples (and around 6 minutes on 10M tuples), compared to around 17 minutes of BHE, 19 minutes of lPIR and more than three hours of cPIR (in Fig. 6(c)). The cost reduction of HHE is mainly on the number of buckets that the server needs to process, as seen in Fig. 6(b), and the size of data transmitted from the server to the client, as seen in Fig. 6(d). The client query formulation time in lPIR for 100K and 1M data, as seen in Fig. 6(a), can be reduced by using larger block size such as 100 buckets, but 100 buckets block size does not work on 10M data, so we still use 10 buckets block size for all synthetic data.

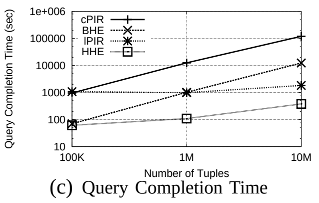
Varying Query Selectivity. We then study the effects of



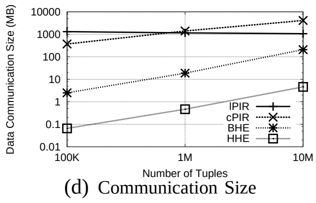
(a) Query Formulation Time



(b) Server Processing Time

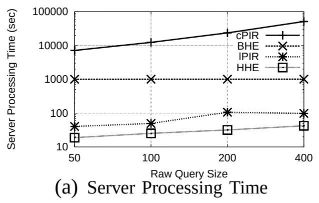


(c) Query Completion Time

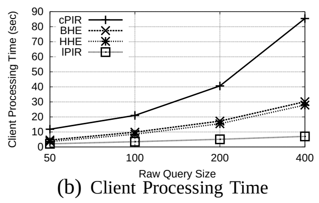


(d) Communication Size

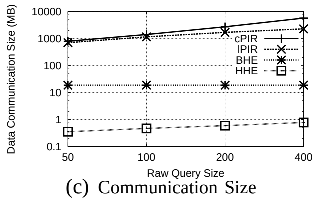
Fig. 6. Effects of Varying Number of Tuples



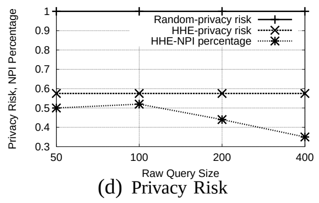
(a) Server Processing Time



(b) Client Processing Time

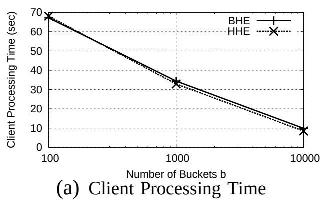


(c) Communication Size

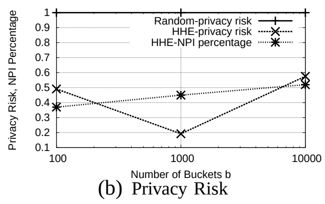


(d) Privacy Risk

Fig. 7. Effects of Varying Raw Query Size

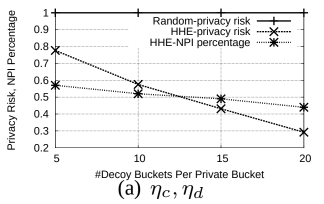


(a) Client Processing Time

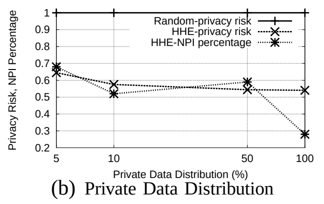


(b) Privacy Risk

Fig. 8. Effects of Varying Number of Buckets



(a) η_c, η_d



(b) Private Data Distribution

Fig. 9. Effects of Other Factors on Privacy Risk

query selectivity / raw query sizes on query performance as well as on the privacy risk of HHE. We varied the raw query size from 50 to 400 while fixing the other parameters as default. As the query size increases, the size of the answers to a query increases. It can be expected that the time that the client spends on decrypting and post-processing results increases, as shown in Fig. 7(b). Since BHE retrieves all the buckets in one batch, its server processing time and data communication size do not increase. In contrast, since cPIR and lPIR run many rounds to retrieve data tuples in different blocks, their server processing time and data communication size increase with the increasing query sizes, as seen in Figs. 7(a) and 7(c). HHE finds more decoy buckets for larger query sizes, so its server processing time increases due to computing on more buckets, which in turn brings down the percentage that NPI patterns are used for generating decoy buckets, thus increasing the percentage that private buckets are covered by PI patterns for the same privacy risk 0.57, as seen in Fig. 7(d). Note that since our privacy risk is defined based on the supports of covering patterns as in Equations (4) and (5), generating more random decoy buckets does not improve privacy, i.e. Random in Fig. 7(d) always incurs above 0.99 privacy risk, and it still does not match any patterns or just matches NPI patterns with increasing numbers of decoy buckets. Since BHE and cPIR have zero privacy risks, they are not shown in Fig. 7(d).

Varying Bucketization. We next specifically study different granularities of bucketizations on query performance as well as on privacy risk in HHE. We varied the number of buckets b from 100 to 10000 on 1M public tuples, while fixing the other parameters as default. As bucketization becomes finer, query matching becomes more accurate, then the amount of ciphertexts that a client needs to decrypt and post-process is smaller, as seen in Fig. 8(a). As bucketization becomes finer, query bucket patterns become more diverse, so it is harder to find PI patterns that can fully cover private buckets, thus increasing NPI percentage in Fig. 8(b).

Other Factors on Privacy Risk. Finally, we study different factors on privacy risk in HHE. We first varied the number of cover / decoy buckets per private bucket η_c and η_d from 5 to 20, while fixing the other parameters as default. As expected, the privacy risk decreases from 0.77 to 0.29 (in Fig. 9(a)). We then varied the distribution of 1000 private keys on public key domain from 5% to 100%, while fixing the other parameters as default. As the private data keys are distributed in broader ranges of the pubic data domain, the generated queries are also distributed in broader ranges of the public data domain, and more PI patterns can be found to cover private buckets, leading to privacy risk reduction and NPI percentage as seen in Fig. 9(b).

Results on NPI Real Data. To mimic real application scenarios such as searching for medical providers, we evaluated our proposals on NPI real data. We fixed raw query size to 100 private data tuples, η_c and η_d as 10. The average query completion time for a join or range query in HHE on NPI data is around 72.6 seconds (compared to 88.2 seconds of lPIR, 2 hours of BHE and 20 hours of cPIR), with around 613KB ciphertext being transmitted (compared to 64MB of lPIR, 160MB of BHE and 6GB of PIR) and privacy risk being 0.53.

VII. Conclusion

In the past, private query processing has not been realized for two reasons: (1) impractical expensive performance, and

(2) unable to support rich functionality queries beyond simple selection queries and retrievals. We have addressed these two problems in the paper by proposing two private query processing protocols, BHE and HHE, based on homomorphic encryption for solving private join queries and range queries. Our experimental evaluation have shown BHE and HHE perform better than PIR, and the performance of HHE is practical on properly finer data bucketization. Although HHE trade-offs partial privacy for performance gain, we measured its privacy using a novel metric based on frequent query bucket patterns and minimized its privacy risk when generating decoy buckets. In addition, our proposed two stages cover and decoy buckets generation ensures the decoy buckets appear as likely as private data buckets in multiple queries of a user session, and thus consistently protects the user's privacy in the session.

REFERENCES

[1] http://acsc.csl.sri.com/libpaillier.

[2] Privacy leaks hit facebook, google, at&t. http://news.cnet.com/2702-1009_3-986.html, 2010.

[3] Twitter finalizes ftc security settlement. http://www.informationweek.com/news/security/attacks/229301037, 2011.

[4] D. Agrawal and C. C. Aggarwal. On the design and quantification of privacy preserving data mining algorithms. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '01, pages 247–255, 2001.

[5] M. Arrington. Aol proudly releases massive amounts of private data. http://www.techcrunch.com/2006/08/06/aol-proudly-releases-massive-amounts-of-user-search-data, August 2006.

[6] J. Bethencourt, D. Song, and B. Waters. New techniques for private stream searching. *ACM Trans. Inf. Syst. Secur.*, 12:16:1–16:32, January 2009.

[7] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, 1998.

[8] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *USENIX Security Symposium*, 2004.

[9] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *TCC*, pages 265–284, 2006.

[10] S. R. Ganta, S. P. Kasiviswanathan, and A. Smith. Composition attacks and auxiliary information in data privacy. In *Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '08, pages 265–273, New York, NY, USA, 2008. ACM.

[11] C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC '09: Proceedings of the 41st annual ACM symposium on Theory of computing*, pages 169–178, 2009.

[12] C. Gentry and Z. Ramzan. Single-database private information retrieval with constant communication rate. In *In Proceedings of the 32nd International Colloquium on Automata, Languages and Programming*, pages 803–815, 2005.

[13] H. Hacigumus, B. R. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database service provider model. In *SIGMOD Conference*, 2002.

[14] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2000.

[15] D. C. Howe and H. Nissenbaum. TrackMeNot: Resisting surveillance in web search. In *Lessons from the Identity Trail: Anonymity, Privacy, and Identity in a Networked Society*, chapter 23, pages 417–436. Oxford University Press, 2009.

[16] H. Hu, J. Xu, C. Ren, and B. Choi. Processing private queries over untrusted data cloud through privacy homomorphism. In *ICDE*, 2011.

[17] O. H. Ibarra and C. E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *J. ACM*, 22:463–468, October 1975.

[18] M. Kantarcioglu and C. Clifton. Privacy-preserving distributed mining of association rules on horizontally partitioned data. *IEEE Trans. on Knowl. and Data Eng.*, 16:1026–1037, September 2004.

[19] E. Kushilevitz and R. Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *FOCS*, pages 364–373, 1997.

[20] N. Li, T. Li, and S. Venkatasubramanian. t-closeness: Privacy beyond k-anonymity and l-diversity. In *ICDE*, pages 106–115, 2007.

[21] C. A. Melchor, B. Crespin, P. Gaborit, V. Jolivet, and P. Rousseau. High-speed private information retrieval computation on gpu. In *SECURWARE*, pages 263–272, 2008.

[22] C. A. Melchor and P. Gaborit. A fast private information retrieval protocol. In *IEEE Internal Symposium on Information Theory*, pages 1848–1852, 2008.

[23] M. F. Mokbel, C.-Y. Chow, and W. G. Aref. The new casper: A privacy-aware location-based database server. In *ICDE*, pages 1499–1500, 2007.

[24] M. Murugesan and C. Clifton. Providing privacy through plausibly deniable search. In *SDM*, pages 768–779, 2009.

[25] F. G. Olumofin and I. Goldberg. Revisiting the computational practicality of private information retrieval. In *Financial Cryptography*, 2011.

[26] F. G. Olumofin, P. K. Tysowski, I. Goldberg, and U. Hengartner. Achieving efficient query privacy for location based services. In *Privacy Enhancing Technologies*, pages 93–110, 2010.

[27] R. Ostrovsky and W. E. Skeith. Private searching on streaming data. *Journal of Cryptology*, 20:397–430, 2007.

[28] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology - EUROCRYPT '99*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer Berlin / Heidelberg, 1999.

[29] H. Pang, X. Ding, and X. Xiao. Embellishing text search queries to protect user privacy. *PVLDB*, 3(1):598–607, 2010.

[30] S. T. Peddinti and N. Saxena. On the privacy of web search based on query obfuscation: A case study of trackmenot. In *Privacy Enhancing Technologies*, pages 19–37, 2010.

[31] R. K. Pon and T. Critchlow. Performance-oriented privacy-preserving data integration. In *DILS*, pages 240–256, 2005.

[32] P. Samarati and L. Sweeney. Protecting privacy when disclosing information: k-anonymity and its enforcement through generalization and suppression. Technical report, 1998.

[33] R. Sion and B. Carbunar. On the computational practicality of private information retrieval. In *Network and Distributed System Security Symposium*, 2007.

[34] L. Sweeney. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(5):557–570, 2002.

[35] S. Wang, D. Agrawal, and A. E. Abbadi. Generalizing pir for practical private retrieval of public data. In *DBSec*, pages 1–16, 2010.

[36] S. Wang, D. Agrawal, and A. E. Abbadi. Towards practical private processing of database queries over public data with homomorphic encryption. Technical Report 2011-06, Department of Computer Science, University of California at Santa Barbara, 2011.

[37] P. Williams and R. Sion. Usable private information retrieval. In *Network and Distributed System Security Symposium*, 2008.