Joining Privately on Outsourced Data

Bogdan Carbunar[1] and Radu Sion[2]*

[1] Applied Research Center, Motorola Labs, Schaumburg, IL, USA
[2] Computer Science, Stony Brook University, Stony Brook, NY, USA

Abstract. In an outsourced database framework, clients place data management with specialized service providers. Of essential concern in such frameworks is data privacy. Potential clients are reluctant to outsource sensitive data to a foreign party without strong privacy assurances beyond policy "fine–prints". In this paper we introduce a mechanism for executing general binary JOIN operations (for predicates that satisfy certain properties) in an outsourced relational database framework with full computational privacy and low overheads – a first, to the best of our knowledge. We illustrate via a set of relevant instances of JOIN predicates, including: range and equality (e.g., for geographical data), Hamming distance (e.g., for DNA matching) and semantics (i.e., in health-care scenarios – mapping antibiotics to bacteria). We experimentally evaluate the main overhead components and show they are reasonable. For example, the initial client computation overhead for 100000 data items is around 5 minutes. Moreover, our privacy mechanisms can sustain theoretical throughputs of over 30 million predicate evaluations per second, even for an un-optimized OpenSSL based implementation.

1 Introduction

Outsourcing the "database as a service" [24] emerged as an affordable data management model for parties ("data owners") with limited abilities to host and support large in-house data centers of potentially significant resource footprint. In this model a *client* outsources its data management to a *database service provider* which provides online access mechanisms for querying and managing the hosted data sets.

Because most of the data management and query execution load is incurred by the service provider and not by the client, this is intuitively advantageous and significantly more affordable for parties with less experience, resources or trained man-power. Compared with e.g., a small company, with likely a minimal expertize in data management, such a database service provider intuitively has the advantage of expertize consolidation. More-over it is likely to be able to offer the service much cheaper, with increased service availability and uptime guarantees.

* Sion is supported by the National Science Foundation through awards CCF 0937833, CNS 0845192, IIS 0803197, and by CA Technologies, Xerox, IBM and Microsoft.

Significant security issues are associated with such "outsourced database" frameworks, including communication-layer security and data confidentiality. Confidentiality alone can be achieved by encrypting the outsourced content. Once encrypted however, the data cannot be easily processed by the server. This limits the applicability of outsourcing, as the type of processing primitives available will be reduced dramatically.

Thus, it is important to provide mechanisms for server-side data processing that allow both confidentiality and a sufficient level of query expressibility. This is particularly relevant in relational settings. Recently, protocols for equijoin and range queries have been proposed [34, 35, 33].

Here we go one step further and provide low overhead solutions for *general* binary JOIN predicates that satisfy certain properties: for any value in the considered data domain, the *number* of corresponding "matching" pair values (for which the predicate holds) is upper bound. We call these finite match predicates (FMPs).

Such predicates are extremely common and useful, including any discrete data scenarios, such as ranges, inventory and company asset data sets, forensics, genome and DNA data (e.g., fuzzy and exact Hamming distances), and healthcare databases (e.g., bacteria to antibiotics matches). Moreover, at the expense of additional client-side processing (pruning of false positives) other predicate types (multi-argument, continuous data) can be accommodated.

While on somewhat orthogonal dimensions, it might be worth noting that other important challenges are to be considered in the framework of database outsourcing. Transport layer security is important as eavesdropping of data access primitives is unacceptable. This can be achieved by deploying existing traditional network security protocols such as IPSec/SSL. Moreover, query correctness issues such as authentication and completeness are important and have been previously considered.

The main contributions of this paper include: (i) the proposal and definition of the problem of private joins for generalized query predicates, (ii) a solution for FMPs, (iii) its analysis, (iv) a proof-of-concept implementation and (v) the experimental evaluation thereof.

The paper is structured as follows. Section 2 introduces the main system, data and adversary models. Section 3 overviews, details and analyzes our solution. Section 4 discusses predicate instance examples and the handling thereof. Section 5 introduces our proof-of-concept implementation and provides an experimental analysis thereof. Section 6 surveys related work and Section 7 concludes.

2 Model

We choose to keep the data outsourcing model concise yet representative. Sensitive data is placed by a client on a database server situated at the site and under the control of a *database service provider*. Later, the client can access the outsourced data through an online query interface exposed by the server. Network layer confidentiality is assured by mechanisms such as SSL/IPSec. This

corresponds to a *unified client model* [14, 33]. Clients would like to allow the server to process data queries while maintaining data confidentiality. For this purpose, they will encrypt data before outsourcing. As encrypted data is hard to process without revealing it, to allow for more expressive server-side data processing, clients will also pre-process data according to a set of supported (join) predicates. They will then outsource additional associated metadata to aid the server in processing tasks. This metadata, however, will still be "locked" until such processing tasks are requested by the client.

Later, to allow server – side data processing, the client will provide certain "unlocking" information for the metadata associated with the accessed items. The server will perform *exactly* the considered query (and nothing more) without finding out any additional information.

It is important for the outsourced metadata not to reveal any information about the original data. Additionally, the computation, storage and network transfer overheads should maintain the cost advantages of outsourcing, e.g., execution times should not increase significantly. We consider a relational model, where we consider the outsourced data as a set of t data columns (e.g., relational attributes), D stored on the server. Let n be the average number of values stored in a column and b be the number of bits in the representation of a value. Naturally, we allow relations to contain variable number of tuples. We use this notation for analysis purposes only.

Finite Match Predicates (FMPs). In this paper we consider binary predicates $p : \mathbb{X} \times \mathbb{Y} \rightarrow \mathbb{B} = \{true, false\}$ for which the "match sets" $P(x) := \{y | p(x, y) = true\}$ can be computed efficiently and their size (taken over all encountered values of x) upper bound. In other words, given a certain value x in the considered data domain, its "matching" values are easily determined and their number is upper bound. We call these predicates *finite match* predicates (FMPs). We call the number of matching values *maximum match size* (MMS). For instance, consider the following discrete time – range join query that joins arrivals with departures within the same 30 mins interval (e.g., in a train station):

```
SELECT * FROM arrivals,departures
WHERE ABS(arrivals.time - departures.time) > 30
```

In this example, the FMP has an MMS of 60.

Privacy Requirements. In the considered adversarial setting, the following privacy requirements are of concern.

Initial Confidentiality. The server should not be able to evaluate inter-column (join) predicates on the *initially* received data without ("un-lock") permission from the client.

Predicate Safety. The server should not be able to evaluate predicates on "unlocked" data. This also implies that no additional information should be leaked in the process of predicate evaluation. For instance, allowing the evaluation of predicate $p(x, y) := (|x - y| < 100)$, should not reveal $|x - y|$.

We stress that here we do not provide confidentiality of predicates, but rather just of the underlying target data. We also note that we do not consider here

the ability of the server to use out of band information and general knowledge about the data sets to infer what the underlying data and the query results look like. In fact we envision a more formal definition in which privacy guarantees do not allow any leaks to the server beyond exactly such inferences that the curious server may do on its own based on outside information.

Performance Constraints. The main performance constraint we are interested in is *maintaining the applicability of outsourcing*. In particular, if a considered query load is more efficient (than client processing) in the unsecured data outsourcing model – then it should still be more efficient in the secured version. We believe this constraint is essential, as it is important to identify solutions that validate in real life. There exist a large number of apparently more elegant cryptographic primitives that could be deployed that would fail this constraint. In particular, experimental results indicate that *predicate evaluations on the server should not involve any expensive (large modulus) modular arithmetic such as exponentiation or multiplication*. We resisted the (largely impractical) trend (found in existing research) to use homomorphisms in server side operations, which would have simplified the mechanisms in theory but would have failed in practice due to extremely poor performance, beyond usability.

supported by recent findings that show the total cost of storage management is orders of magnitude higher than the initial storage equipment acquisition costs [17].

Adversary. We consider an *honest but curious* server: given the possibility to get away undetected, it will attempt to compromise data confidentiality (e.g., in the process of query execution). The protocols in this paper are protecting mainly data *confidentiality*. The server can certainly choose to deny service by explicitly not cooperating with its clients, e.g., by not returning results or simply closing connections.

2.1 Tools

Encryption, Hashing and Random numbers. We consider ideal, collision-free hashes and strongly un-forgeable signatures. While, for reference purposes we benchmark RC4 and AES in section 5, we will not be more specific with respect to its nature and strength as it is out of scope here. We note that our solution does not depend on any specific encryption mechanism. We will denote by $E_K(v)$ the encryption of value v with key secret key K. If not specified, the key K will be implicitly secret and known only to the client. In the following, we use the notation $x \hookrightarrow_R S$ to denote x's uniformly random choice from S.

Bloom Filters. Bloom filters [8] offer a compact representation of a set of data items, allowing for fast set inclusion tests. Bloom filters are *one-way*, in that, the "contained" set items cannot be enumerated easily (unless they are drawn from a finite, small space). Succinctly, a Bloom filter can be viewed as a string of l bits, initially all set to 0. To *insert* a certain element x, the filter sets to 1 the bit values at index positions $H_1(x), H_2(x), \ldots, H_h(x)$, where $H_1, H_2, \ldots, H_h$

are a set of h crypto-hashes. Testing set inclusion for a value x is done by checking that the bits for *all* bit positions $H_1(x), H_2(x), \ldots, H_h(x)$ are set. By construction, Bloom filters feature a controllable rate of false positives (p_{fp}) for set inclusion tests. For a certain number N of inserted elements, there exists a relationship that determines the optimal number of hash functions h_o minimizing p_{fp}: $h_o = \frac{l}{N} \ln 2 \approx 0.7\frac{l}{N}$ which yields a false positive probability of $p_{fp} = \left(\frac{1}{2}\right)^{h_o} = \left(\frac{1}{2}\right)^{\frac{l}{N} \ln 2} \approx 0.62^{l/N}$ For a Bloom filter BF, we denote $BF.insert(v)$ the insertion operation and $BF.contains(v)$ the set inclusion test (returning *true* if it contains value v, $false$ otherwise).

For an excellent survey on applications on Bloom filters and their applications in a variety of network problems please see [10].

Computational Intractability Assumptions. Let $\mathbb{G}$ be a finite field of size p prime and order q and let g be a generator for $\mathbb{G}$. The Computational Diffie-Hellman assumption (CDH) [21]:

Definition 1 *(CDH Assumption) given g, g^a mod p and g^b mod p, for $a, b \in \mathbb{Z}_q$, it is computationally intractable to compute the value g^{ab} mod p.*

In the same cyclic group $\mathbb{G}$, the Discrete Logarithm assumption (DL):

Definition 2 *(DL Assumption) given $g, v \in \mathbb{G}$, it is intractable to find $r \in \mathbb{Z}_q$ such that $v = g^r$ mod p.*

3 Outsourced JOINs with Privacy

We define the arbitrary (non hard-coded to a specific application) predicate join solution to be a quadruple $(pred_{FM}, G, E, J)$, where $pred_{FM}$ is the FMP, G is a parameter generation function, E is a data pre-processing function and J denotes a joining function according to predicate $pred_{FM}$. G and E are executed by the client and the output of E is outsourced to the server. J is executed by the server on two attributes of the client's data. In this section we provide a general description of the G, E and J functions and in Section 4 we study two predicate and corresponding G, E and J function instances. In Figure 1 we summarize the symbols used in our solution.

p	prime number
N	bit size of p
$\mathbb{G}$	subgroup of $\mathbb{Z}_p$
q	order of $\mathbb{G}$
g	generator of $\mathbb{G}$
x_A, y_A	secret values for column A

Fig. 1. Symbol Table.

G is a parameter generation operation executed initially by the client. Its input is N, a security parameter and t, the number of columns in the client database D. Let $p = 2p' + 1$ be a N bit long prime, such that p' is also prime. The reason for this choice is to make the CDH assumption harder. Let $\mathbb{G} = \mathbb{Z}_p$ be a group of order q, with a generator g.

$G(N,t)$**.** Generates an encryption key $K \hookrightarrow_R \{0,1\}^*$. For each column $A \in D$, generate two values $x_A, y_A \hookrightarrow_R \mathbb{Z}_q$, $x_A \neq y_A$. Publish p and g and keep secret the key K and the values x_A and y_A, for all columns $A \in D$.

E is executed by the client, after running G. It takes as input a column $A \in D$, the key K and the secret values x_A and y_A corresponding to column A.

$E(A, K, x_A, y_A)$**.** Associate with each element $a_i \in A$, $i = 1..n$ a Bloom filter denoted $BF(a_i)$, with all the bits initially set to 0. Let $P(a_i) = \{v | pred_{FM}(a_i, v) = true\}$ be the set of values that satisfy the predicate $pred_{FM}$ for element a_i. For each $a_i \in A$, encrypt a_i with the key K, producing $E_K(a_i)$ and compute an "obfuscation" of a_i, $O(a_i) = H(a_i)x_A \ mod\ q$. Then, $\forall v \in P(a_i)$, compute $e_A(v) = g^{H(v)y_A} \ mod\ p$ and insert them into a_i's Bloom filter ($BF(a_i).insert(e_A(v))$). Finally, output the values $E_K(a_i)$, $O(a_i)$ and $BF(a_i)$. Let D_T denote the output of E for all the columns in D. The client stores D_T on the server. Hence, element $a_i \in A$ is stored on the server as $D_T(A, i) = [E_K(a_i),\ O(a_i),\ BF(a_i)]$.

We now describe the join operation, J, executed by the server. J takes as input two column names A, B, a desired predicate $pred_{FM}$ and a trapdoor value (computed and sent by the client) $r_{AB} = g^{y_A/x_B} \ mod\ p$ and outputs the result of the join of the two columns on the predicate.

$J(A, B, pred_{FM}, r_{AB})$**.** For each element $b_j \in B$, compute $e_A(b_j) = r_{AB}^{O(b_j)} \ mod\, p$. For each element $a_i \in A$, iff. $BF(a_i).contains(e_A(b_j))$ return the tuple $\langle E_K(a_i), E_K(b_j)\rangle$.

In real life, J will output also any additional attributes specified in the SELECT clause, but for simplicity we explicit here and in the following only the join attributes.

Properties We now list the security properties of this solution, whose proofs will be included in an extended version of the paper.

Theorem 1. *(Correctness) The join algorithm J returns all matching tuples.*

Theorem 2. *The* $(pred_{FM}, G, E, J)$ *solution satisfies the* initial confidentiality *requirement outlined in Section 2.*

Theorem 3. $(pred_{FM}, G, E, J)$ *is* predicate safe*.*

Notes on Transitivity. Under certain circumstances the server may use our solution to perform transitive joins. That is, provided with information to join A with B and later to join B with C, it can join A and C. We make the observation that on certain FMPs any solution will allow the server to perform partial transitive joins, using the outcome of previous joins. That is, when an element $b \in B$ has matched an element $a \in A$ and an element $c \in C$, the server can infer that with a certain probability a also matches c. In conclusion, we believe the transitive join problem to be less stringent than reducing server-side storage and computation overheads.

Same-column Duplicate Leaks. In the case of duplicate values occurring in the same data column, a data distribution leak can be identified. The deterministic nature of the obfuscation step in the definition of E associates the same obfuscated values to duplicates of a value. Upon encountering two entries with the same obfuscated value, the server indeed can infer that the two entries are identical. We first note that if joins are performed on primary keys this leak does not occur. Additionally, it is likely that in many applications this is not of concern. Nevertheless, a solution can be provided, particularly suited for the case when the number of expected duplicates can be upper bound by a small value (e.g., m). The deterministic nature of $O(a_i)$ is required to enable future Bloom filter lookups in the process of predicate evaluation. However, as long as the predicate evaluation is designed with awareness of this, each duplicate can be replaced by a unique value. This can be achieved by (i) populating Bloom filters with multiple different "variants" for each value expected to occur multiple times, and (ii) replacing each duplicate instance with one of these variants instead of the actual value. These variants can be constructed for example by padding each value with different $log_2(m)$ bits.

Bloom Filter Sizes. In the case of a join, the false positive rate of Bloom filters implies that a small percentage of the resulting joined tuples do *not* match the predicate the join has been executed for. These tuples will then be pruned by the client. Thus, a trade-off between storage overhead and rate of false positives (and associated additional network traffic) emerges. For example, for a predicate MMS $= N = 60$ (e.g., in the simple query in Section 2), a desired false positive rate of no more than $p_{fp} = 0.8\%$, the equations from Section 2.1 can be used to determine one optimal setup $l = 600$ and $h = 7$.

Data Updates and Multiple Clients. In data outsourcing scenarios, it is important to handle data updates incrementally, with minimal overheads. In particular, any update should not require the client to re-parse the outsourced data sets in their entirety. The solution handles data updates naturally. For any new incoming data item, the client's pre-processing function E can be executed per-item and its results simply forwarded to the server. Additionally, in the case of a multi-threaded server, multiple clients (sharing secrets and keys) can access the same data store simultaneously.

Complex, Multi-predicate Queries. Multiple predicate evaluations can be accommodated naturally. Confidentiality can be provided for the attributes involved in binary FMPs. For example, in the following database schema, the association between patients and diseases is confidential but any other information is public and can be used in joins. To return a list of Manhattan-located patient names and their antibiotics (but not their disease) the server will access both confidential (disease) and non-confidential (name,zip-code) values.

```
SELECT patients.name,antibiotics.name
FROM patients,antibiotics
```

```
WHERE md(patients.disease,antibiotics.name)
AND patients.zipcode = 10128
```

Only the predicate $md()$ will utilize the private evaluation support. This will be achieved as discussed above, by encrypting the `patients.disease` attribute and generating metadata for the `antibiotics` relation (which contains a list of diseases that each antibiotic is recommended for).

4 Predicate Instances

To illustrate, we choose to detail two predicate instances: a simple, range join and a Hamming distance predicate requiring custom predicate-specific extensions.

4.1 Range JOIN

Consider the binary FMP $p(x, y) := (v_1 \leq (x - y) \leq v_2)$ where $x, y \in \mathbb{Z}$. An instance of this predicate is the following travel agency query, allocating buses to trips, ensuring 5 (but no more than 10) last-minute empty slots per trip:

```
SELECT buses.name,trips.name
FROM buses,trips
WHERE (buses.capacity-trips.participants) >= 5
AND (buses.capacity-trips.participants) <= 10
```

Executing such a query remotely with privacy can be achieved efficiently by deploying the solution presented in Section 3. The parameter generation algorithm, G and the join algorithm J will be the same. As above, the data encoding algorithm encodes in the Bloom filter $BF(a_i)$ of element a_i all integer values in $P(a_i) := \{y | p(a_i, y) = true\}$ namely with values $\in [x - v_2, x - v_1]$. Note that given the size of the range, n and a fixed probability of false positives, p_{fp}, we have that the optimum Bloom filter size is $l = -\frac{n \ln p_{fp}}{(\ln 2)^2}$.

4.2 Hamming JOIN

It is often important to be able to evaluate Hamming distance on remote data with privacy in un-trusted environments. This has applications in forensics, criminal investigation (e.g. fingerprints), biological DNA sequence matching, etc.

Let x and y be b bit long strings and let $0 < d < b$ be an integer value. We use $d_H(x, y)$ to denote the Hamming distance of x and y. We consider the join predicate $pred_{FM}(x, y) := (d_H(x, y) \leq d)$. An example is the following fingerprint matching query that retrieves the names and last dates of entry for all individuals with physical fingerprints (in some binary representation) close enough to the ones of suspects on the current FBI watch list:

```
SELECT watchlist.name,
       immigration.name,
       immigration.date
```

```
FROM watchlist,immigration
WHERE Hamming(watchlist.fingerprint,
              immigration.fingerprint)<5
```

A private execution of this join operation can be deployed using the solution introduced in Section 3. The implementation of the Hamming part of the predicate requires specific adjustments. In particular, in pre-processing, the client pseudo-randomly bit-wise permutes all the data elements consistently. It then splits each data element into β equal sized blocks, where β is an input parameter discussed later. Then, for each such block, it generates three data items: one item will allow later private comparisons with other blocks for equality (Hamming distance 0). The other two (a Bloom filter and a "locked" obfuscated value) will be used by the server to identify (with privacy) blocks at Hamming distance 1. In the following we describe the (d_H, G_H, E_H, J_H) solution, as an extension of the solution presented in Section 3.

The parameter generator, G_H, takes two additional parameters, β and b. b is the bit length of elements from D and β is the number of blocks into which each data element is split. We assume $\beta > d$ is constant, much smaller than the number of elements stored in a database column. Possible values for β are investigated later in this section.

G_H(N,t,β,b). Choose a value $s \hookrightarrow_R \{0,1\}^*$ and generate a secret pseudo-random permutation $\pi : \{0,1\}^b \rightarrow \{0,1\}^b$. For each data column $A \in D$ [3], compute $s_A = H(s, A)$. Use s_A to seed a pseudo-random number generator PRG. Use PRG to generate 3β secret, duplicate-free pseudo-random values $x_A(1), .., x_A(\beta)$, $y_A(1), .., y_A(\beta)$, $z_A(1), .., z_A(\beta) \hookrightarrow_R \mathbb{Z}_q$.

$E_H(A, K, x_A(k), y_A(k), z_A(k)), k = 1..\beta, A \in D$. For each element a_i, $i = 1..n$ of A, compute a_i's *bit-wise* permutation $\pi(a_i)$, then split $\pi(a_i)$ into β blocks of equal bit length, $a_{i1}, .., a_{i\beta}$. For each block a_{ik}, $k = 1..\beta$, generate an obfuscated value $O(a_{ik}) = H(a_{ik})x_A(k)$ *mod* q. Then, create a_{ik}'s Bloom filter by generating all values v for which $d_H(a_{ik}, v) = 1$. That is, generate all values with Hamming distance 1 from block a_{ik}. For each value v, let $e^k_A(v) = g^{H(v)y_A(k)}$ *mod* p. Encode $e^k_A(v)$ into a_{ik}'s Bloom filter, using operation $BF(a_{ik}).insert(e_A)^k(v))$. Compute an additional structure allowing the server to assess (with privacy) equality of the kth block of a_i with the kth blocks of other values, $Z(a_{ik}) = H(a_{ik})z_A(k)$ *mod* q. Finally, output $[E_K(a_i), O(a_{ik}), Z(a_{ik}), BF(a_{ik})]$, for all $k = 1..\beta$. Hence element a_i is stored on the server as a tuple $D_T(A, i) = [E_K(a_i), O(a_{ik}), Z(a_{ik}), BF(a_{ik})]$, similar to the solution in Section 3.

To join two columns A and B on predicate $pred_{FM}$, J_H receives the following 3β trapdoor values from the client (3 for each block) (i) $r_A(k) = g^{R_k/z_A(k)}$ *mod* p, (ii) $r_B(k) = g^{R_k/z_B(k)}$ *mod* p and (iii) $r_k = g^{y_A(k)/x_B(k)}$ *mod* p, for $k = 1..\beta$, where $R_k \hookrightarrow_R \{0,1\}^*$ (generated at the client side).

[3] A here is the column's unique server-side name.

$J_H(A, B, r_A(k), r_B(k), r_k), k = 1..\beta$**.** For each element a_i from A and for each $k = 1..\beta$, compute $v(a_{ik}) = r_A(k)^{Z(a_{ik})}$ *mod p*. For each element b_j from B and for each $k = 1..\beta$, compute $v(b_{jk}) = r_B(k)^{Z(b_{jk})}$ *mod p*. For each element $b_j \in B$ and each element $a_i \in A$, set counter c to 0. For each $k = 1..\beta$, if $BF(a_{ik}).contains(r_k^{O(b_{jk})})$ then do $c = c + 1$ and $k = k + 1$. Else, if $v(a_{ik}) = v(b_{jk})$, do $k = k + 1$. Otherwise, move to the next element, a_{i+1}, from A. If at the end of the k loop, $c < d$, return $\langle E_K(a_i), E_K(b_j)\rangle$. Else, move to the next element from A, a_{i+1}.

Note that for future query purposes the client does not need to remember the values ($x_A(k)$, $y_A(k)$, $z_A(k)$) for each column A. Instead, it generates them by seeding its PRG with s_A. For this, the client only needs to store one value, s.

Theorem 4. *Any given pair of elements from A and B at Hamming distance less than or equal to d is found with probability at least $e^{-d/\beta}(1 + \frac{d}{\beta})$.*

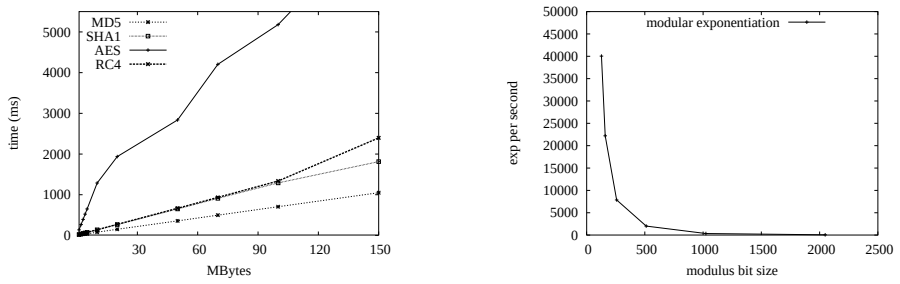
Arbitrary Alphabets. The above solution can also be deployed for an arbitrary alphabet, that is, when the elements stored in the database D are composed of symbols from multi-bit alphabets (e.g., DNA sequences). This can be done by deploying a custom binary coding step. Let $\mathcal{A} = \{\alpha_0, .., \alpha_{u-1}\}$ be an alphabet of u symbols. In the pre-processing phase, the client represents each symbol over u bits ($u/\log u$-fold blowup in storage), such that symbol $\alpha_u = 2^i$. That is, $d_H(\alpha_i, \alpha_j)$ is 1 if $i \neq j$ and 0 otherwise. If each data item has b symbols, each of the item's blocks will have bu/β bits, and, due to the coding, pairs of elements of symbol-wise distance d will have a $2d$ bit-wise Hamming distance. Thus, after the coding phase, the above algorithm can be deployed without change. As an example, for an alphabet of 4 symbols {A,C,G,T}, the following encoding will be used {A=0001,C=0010,G=0100,T=1000}. To compare the strings ACG and ACT (alphabet distance 2), the following two binary strings will be compared instead: 000100100100 and 000100101000 (binary Hamming distance 2).

5 Experimental Results

Implementation Details. We conducted our experiments using a C/C++ implementation of the private predicate join algorithms, on 3.2GHz Intel Pentium 4 processors with 1GB of RAM running Linux. We implemented the cryptographic primitives using OpenSSL 0.9.7a. Our goal was to investigate the feasibility of the algorithms in terms of computation, communication and storage overheads, both on the client and the server side.

To understand the costs of encryption and hashing, we have evaluated several symmetric encryption and crypto-hashing algorithms. In our setup we benchmarked RC4 at just bellow `80MB/sec`, and MD5 to of up to `150MB/sec`, shown in Figure 2(a). We also benchmarked integer hashing throughput at more than 1.1 million MD5 hashes per second, showing the "startup" cost of hashing.

As recommended by the Wassenaar Arrangement [31], we set N, the size of the prime p to be 512 bits and the size of the prime q to be 160 bits. From



(a) **(b)**

Fig. 2. Overheads of cryptographic operations.

our benchmarks, shown in Figure 2(b), we have concluded that 512-bit modular exponentiations (with 160 bit exponents) take 274usec while 512-bit modular multiplications take only 687nsec.

We have considered three types of applications for the private join algorithms. In a first application we used SNPs (single nucleotide polymorphisms) from a human DNA database [2]. An SNP is a variation of a DNA sequence that differs from the original sequence in a single position. The goal of a join is to identify all pairs of sequences from two columns, that differ in a single position. To achieve this, the Bloom filter of a DNA sequence contains all the sequence's SNPs. Since SNPs from [2] have 25 nucleotides, each from the set A, T, C, or G, a Bloom filter stores 75 values (MMS=75). Our second application performs fingerprint matching, that is, identifying similar pairs of fingerprints. We have used fingerprint data from [1] where each fingerprint consists of 100 features. For this application we considered only fingerprints that differ in at most one feature to be a match, thus, Bloom filters store 100 values (MMS=100). The last application identifies picture similarities, using digital images from the LabelMe [40] and Caltech 101 [16] databases. A set of images are annotated with scores for lightness, hue or colors of interest [15, 19]. The Bloom filter associated with an image contains score ranges of interest, which for this application was set to 100 values around the image's score (MMS=100). To compare two images for similarity, the score of one image is searched in the Bloom filter associated with the other image.

Client Computation Overheads. We now describe our investigation of the initial client pre-processing step. Of interest were first the computation overheads involved in generating the encryption, obfuscation and Bloom filter components associated with a database of 100000 elements of 16 bytes each. We experimented with four combinations of encryption algorithms (RC4 and AES) and hashing algorithms (MD5 and SHA1), in a scenario where Bloom filters store 75 items each. Figure 3(a) depicts our results (log scale time axis). For each encryption/hash algorithm combination shown on the x axis, the left hand bar is the encryption cost, the middle bar is the Bloom filter generation cost and

the right hand bar is the obfuscation cost. Our experiments show the dominance of the Bloom filter generation, a factor of 30 over the combined encryption and obfuscation costs. The total computation cost of each implementation is roughly 320 seconds with the minimum being achieved by RC4/MD5. We further investigated the RC4/MD5 combination by increasing the MMS value from 10 to 100. Figure 3(b) shows that the pre-processing overhead increase is linear in the MMS value. The total costs range between 40 seconds (MMS=10) and 7 minutes (MMS=100). We stress that this cost is incurred by the client only once, when computing the initial data structures.

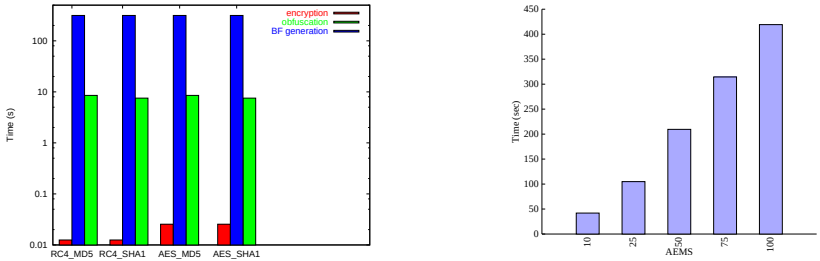


Fig. 3. Client computation overheads.

Server Computation Costs. In order to evaluate the performance of the private join algorithm we used columns of 10000 images each, collected from the LabelMe [40] and Caltech 101 [16] databases. For each image we deployed 1024-bit Bloom filters ($h = 12$ hashes) with MMS=75. The join operation returns all pairs of images that have scores within a given range of each other. In our implementation, for each element from one column we perform a 512-bit modular exponentiation with a 160 bit modulus, followed by a crypto-hash, fragment the result into 12 parts and use each part as a bit position into each of the Bloom filters associated with the elements of the other column.

As, to the best of our knowledge no other solutions exist for arbitrary private joins on encrypted data, we chose to compare our solution against a hypothetical scenario which would use the homomorphic properties of certain encryption schemes such as Paillier [38]. This comparison is motivated by recent related work (e.g., [18]) that deploy this approach to answer SUM and AVG aggregation queries on encrypted data. Moreover, we also considered the cost of solutions that would use RSA encryptions or decryptions to perform private joins.

Figure 4(a) compares our solution against (i) C_P, that performs one modular multiplication within the Paillier cryptosystem with a 1024-bit modulus, for every two elements that need to be compared, (ii) C_{enc}, that uses one 1024-bit RSA encryption for each comparison and (iii) C_{dec}, that uses one 1024-bit RSA decryption operation. The y axis represents the time in logarithmic

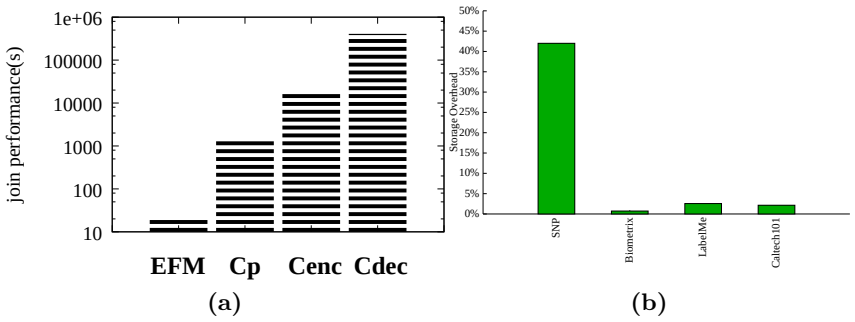


Fig. 4. (a) Join costs for columns of 10000 elements. Our solution is 2-4 orders of magnitude faster than other solutions that use 1024-bit modular operations. (b) Bloom filter storage overhead as percentage of the size of the cleartext data. The overhead is 42% for SNP databases, but under 3% for fingerprint or image databases.

scale. The first bar shows the performance of our FMP join algorithm. The cost is dominated by $10^8 \times 12$ verifications of Bloom filter bit values (the cost of computing 10^4 hashes and exponentiations (modulo a 512-bit prime) is under 3.5s). With a 21.3s computation overhead, the FMP join solution performs two orders of magnitude faster than C_P (second bar) taking 1525s, three orders of magnitude faster than C_{enc} (third bar), taking 19168s and four orders of magnitude faster than C_{dec} (fourth bar), taking 408163s. One reason for the large overhead of the modular multiplications in the Paillier system (used also in [18]) is the fact that while the modulus n has 1024 bits, the multiplications are actually performed in the space $\mathbb{Z}^*_{n^2}$. That is, the active modulus has 2048 bits. Using less than 1024 bits for n is not recommended [3, 31].

Storage Overhead. Since we use symmetric encryption algorithms, the size of the E values stored on the server is roughly the same as the original size of the elements – thus no significant overhead over stioring the cleartext data. The size of the O value for each element is $N = 512$ bits, which is small and data-independent. Finally, Figure 4(b) shows the overhead of the 1024 bit Bloom filters as a percentage of the size of the original data. The largest overhead is 42%, for the SNP database, due to the smaller size of SNPs. However, for image databases, the overhead is under 3% and for fingerprints is under 1%.

Transfer Overhead. We have measured the communication overhead of the initial database transfer between sites located in Chicago and New York, more than a thousand miles apart. With the bottleneck being the uplink capacity of the client, of around 3 Mbps, the overhead of transferring the Bloom filters associated with 100000 items was roughly 32 seconds.

6 Related Work

Extensive research has focused on various aspects of DBMS security and privacy, including access control and general information security issues [5, 4, 6, 7, 12, 13, 25, 26, 28, 29, 32, 36, 37, 39, 41]. Statistical and *Hippocratic* databases aim to address the problem of allowing aggregate queries on confidential data (stored on trusted servers) without leaks [4, 5, 12, 13, 30]. Hacigumus et al.[23] introduced a method for executing SQL queries over partly obfuscated outsourced data. The data is divided into secret partitions and queries over the original data can be rewritten in terms of the resulting partition identifiers; the server can then partly perform queries directly. The information leaked to the server is 1-out-of-s where s is the partition size. This balances a trade-off between client and server-side processing, as a function of the data segment size. At one extreme, privacy is completely compromised (small segment sizes) but client processing is minimal. At the other extreme, a high level of privacy can be attained at the expense of the client processing the queries in their entirety. Similarly, Hore et al. [27] deployed data partitioning to build "almost"-private indexes on attributes considered sensitive. An untrusted server is then able to execute "obfuscated range queries with minimal information leakage". An associated privacy-utility trade-off for the index is discussed.

Ge and Zdonik [18] have proposed the use of a secure modern homomorphic encryption scheme, to perform private SUM and AVG aggregate queries on encrypted data. Since a simple solution of encrypting only one value in an encryption block is highly inefficient, the authors propose a solution for manipulating multiple data values in large encryption blocks. Such manipulation handles complex and realistic scenarios such as predicates in queries, compression of data, overflows, and more complex numeric data types (float), etc. In Section 5 we show that the overhead of the operations used in [18] is very large, exceeding the overhead of FMP joins by three orders of magnitude.

The problem of searching on encrypted data has also been studied extensively. Song et al. [42] introduced an elegant solution that uses only simple cryptographic primitives. Chang and Mitzenmacher [11] proposed a solution where the server stores an obfuscated keyword index which is then used by the client to perform the actual searches. Golle et al. [22] provide a solution with the additional feature of allowing conjunctive keyword searches. In a similar context Boneh et al. [9] proposed the notion of "public key encryption with keyword search". They devised two solutions, one using bilinear maps and one using trapdoor permutations. While ensuring keyword secrecy, these techniques do not prevent servers from building searched keyword statistics and inferring sensitive information.

Goh [20] proposed the notion of "secure index" – a data structure associated with a file. The secure index is stored on a remote server and allows clients to privately query an item into the file. The operation can be performed only if the clients have knowledge of a particular trapdoor value. The construction of a secure index uses pseudo-random functions and Bloom filters. This solution requires knowledge of the trapdoor associated with the searched item. Thus, secure indexes are insufficient to provide private joins on outsourced data.

7 Conclusions

In this paper we introduced mechanisms for executing JOIN operations on outsourced relational data with full computational privacy and low overheads. The solution is not hard-coded for specific JOIN predicates (e.g., equijoin) but rather works for a large set of predicates satisfying certain properties. We evaluated its main overhead components experimentally and showed that we can perform more over 5 million private FMPs per second, which is between two and four orders of magnitude faster than alternatives that would use asymmetric encryption algorithms with homomorphic properties to achieve privacy.

Acknowledgments

We would like to thank our reviewers for their extended comments and suggestions which were extremely helpful in preparing the final version of this paper.

References

1. Biometrix Int. `http://www.biometrix.at/`.
2. International HapMap Project. `http://www.hapmap.org/`.
3. TWIRL and RSA Key Size. Online at `http://www.rsasecurity.com/rsalabs/node.asp?id=2004`.
4. R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic databases. In *Proceedings of the International Conference on Very Large Databases VLDB*, pages 143–154, 2002.
5. R. Agrawal and R. Srikant. Privacy-preserving data mining. In *Proceedings of the ACM SIGMOD*, pages 439–450, 2000.
6. E. Bertino, M. Braun, S. Castano, E. Ferrari, and M. Mesiti. Author-X: A Java-Based System for XML Data Protection. In *IFIP DBSec*, pages 15–26, 2000.
7. E. Bertino, S. Jajodia, and P. Samarati. A flexible authorization mechanism for relational data management systems. *ACM Transactions on Information Systems*, 17(2), 1999.
8. B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.
9. D. Boneh, G. D. Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *Proceedings of Eurocrypt 2004*, pages 506–522. LNCS 3027, 2004.
10. A. Broder, M. Mitzenmacher, and A. B. I. M. Mitzenmacher. Network applications of bloom filters: A survey. In *Internet Mathematics*, pages 636–646, 2002.
11. Y.-C. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *ACNS*, 2005.
12. C. Clifton, M. Kantarcioglu, A. Doan, G. Schadow, J. Vaidya, A. Elmagarmid, and D. Suciu. Privacy-preserving data integration and sharing. In *The 9th ACM SIGMOD workshop on Research issues in data mining and knowledge discovery*, pages 19–26. ACM Press, 2004.
13. C. Clifton and D. Marks. Security and privacy implications of data mining. In *Workshop on Data Mining and Knowledge Discovery*, pages 15–19, Montreal, Canada, 1996. Computer Sciences, University of British Columbia.

14. P. T. Devanbu, M. Gertz, C. Martel, and S. G. Stubblebine. Authentic third-party data publication. In *IFIP Workshop on Database Security*, pages 101–112, 2000.
15. R. Fagin. Fuzzy queries in multimedia database systems. In *Proceedings of the 17th PODS*, pages 1–10, 1998.
16. L. Fei-Fei, R. Fergus, and P. Perona. Learning generative visual models from few training examples. In *Proceedings of IEEE Workshop on Generative-Model Based Vision*, 2004.
17. Gartner, Inc. Server Storage and RAID Worldwide. Technical report, Gartner Group/Dataquest, 1999. `www.gartner.com`.
18. T. Ge and S. B. Zdonik. Answering aggregation queries in a secure system model. In *Very Large Databases (VLDB)*, pages 519–530, 2007.
19. T. Gevers and A. W. M. Smeulders. PicToSeek: Combining Color and Shape Invariant Features for Image Retrieval. *IEEE Trans. on Image Processing*, 9(1):102–119, 2000.
20. E. Goh. Secure indexes. Cryptology ePrint Archive, Report 2003/216, 2003. `http://eprint.iacr.org/2003/216/`.
21. O. Goldreich. *Foundations of Cryptography I*. Cambridge University Press, 2001.
22. P. Golle, J. Staddon, and B. Waters. Secure conjunctive keyword search over encrypted data. In *Proceedings of ACNS*, 2004.
23. H. Hacigumus, B. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *Proceedings of the ACM SIGMOD international conference on Management of data*, pages 216–227. ACM Press, 2002.
24. H. Hacigumus, B. R. Iyer, and S. Mehrotra. Providing database as a service. In *IEEE International Conference on Data Engineering (ICDE)*, 2002.
25. J. Hale, J. Threet, and S. Shenoi. A framework for high assurance security of distributed objects, 1997.
26. E. Hildebrandt and G. Saake. User Authentication in Multidatabase Systems. In *Proceedings of the Ninth International Workshop on Database and Expert Systems Applications, August 26–28, 1998, Vienna, Austria*, 1998.
27. B. Hore, S. Mehrotra, and G. Tsudik. A privacy-preserving index for range queries. In *Proceedings of VLDB*, 2004.
28. S. Jajodia, P. Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *IEEE S&P*, pages 31–42, 1997.
29. S. Jajodia, P. Samarati, V. S. Subrahmanian, and E. Bertino. A unified framework for enforcing multiple access control policies. In *SIGMOD*, 1997.
30. K. LeFevre, R. Agrawal, V. Ercegovac, R. Ramakrishnan, Y. Xu, and D. J. DeWitt. Limiting disclosure in hippocratic databases. In *Proceedings of VLDB*, pages 108–119, 2004.
31. A. K. Lenstra and E. R. Verheul. Selecting cryptographic key sizes. *J. Cryptology*, 14(4):255–293, 2001.
32. Li, Feigenbaum, and Grosof. A logic-based knowledge representation for authorization with delegation. In *PCSFW: Proceedings of the 12th CSFW*, 1999.
33. E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and integrity in outsourced databases. In *ISOC Symposium on Network and Distributed Systems Security NDSS*, 2004.
34. E. Mykletun, M. Narasimha, and G. Tsudik. Signature Bouquets: Immutability for Aggregated/Condensed Signatures. In *Proceedings of the European Symposium on Research in Computer Security ESORICS*, pages 160–176, 2004.
35. M. Narasimha and G. Tsudik. DSAC: integrity for outsourced databases with signature aggregation and chaining. Technical report, 2005.

36. M. Nyanchama and S. L. Osborn. Access rights administration in role-based security systems. In *Proceedings of the IFIP DBSec*, pages 37–56, 1994.
37. S. L. Osborn. Database security integration using role-based access control. In *Proceedings of the IFIP DBSec*, pages 245–258, 2000.
38. P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proceedings of EuroCrypt*, 1999.
39. D. Rasikan, S. H. Son, and R. Mukkamala. Supporting security requirements in multilevel real-time databases, citeseer.nj.nec.com/david95supporting.html, 1995.
40. B. Russell and A. T. an William T. Freeman. LabelMe: the open annotation tool. `http://labelme.csail.mit.edu/`.
41. R. S. Sandhu. On five definitions of data integrity. In *Proceedings of the IFIP DBSec*, pages 257–267, 1993.
42. D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proceedings of the IEEE S&P*, 2000.