Toward Private Joins on Outsourced Data

Bogdan Carbunar, Radu Sion

***Abstract*—In an outsourced database framework, clients place data management responsibilities 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 computational privacy and low overhead – 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. The initial client computation overhead for 100000 data items is around 5 minutes and our privacy mechanisms can sustain theoretical throughputs of several million predicate evaluations per second, even for an un-optimized OpenSSL based implementation.**

***Index Terms*—D.4.6 Security and Privacy Protection. E.0.c Data Encryption**

I. Introduction

Outsourcing the "database as a service" [25] 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 manpower. Compared with e.g., a small company, with likely a minimal data management knowledge, such a database service provider intuitively has the advantage of expertize and the ability to offer the service much cheaper, with increased service availability and uptime guarantees.

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 [15], [34], [35].

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 and DNA data (e.g., fuzzy and exact Hamming distances), and health-care 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 [42], [36], [43], [46].

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 II introduces the main system, data and adversary models. Section III overviews, details and analyzes our solution. Section IV proposes predicate instance examples and their handling. Section V introduces our proof-of-concept implementation and provides its experimental analysis. Section VI surveys related work and Section VII concludes.

II. 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], [35]. 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

Bogdan Carbunar is with the School of Computing and Information Sciences at the Florida International University, Miami, FL. E-mail: carbunar@gmail.com

Radu Sion is with the Computer Science Department in Stony Brook University, Stony Brook, NY. E-mail: sion@cs.stonybrook.edu

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 overhead 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 a 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 by a polynomial time algorithm and their size (taken over all encountered values of x) is upper bound. In other words, given a certain value x in the considered data domain, its "matching" values can be determined in polynomial time and their number is upper bound. We call these predicates *finite match* predicates (FMPs). For a relation R matched against a relation S, we define MMS, the *maximum match size*, to be the maximum number of matching values from relation R for any row in relation S. 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 model, the following privacy requirements are of concern.

Initial Confidentiality. The server should not be able to evaluate inter-column join predicates on *initially* stored data without client "unlock" permission. Formally, given a relation A with encoded elements $D[a_1], .., D[a_n]$, a relation B with encoded elements $D[b_1], .., D[b_m]$, any random values $i \in \{1...n\}$ and $j \in \{1..m\}$, for any probabilistic polynomial time server algorithm $\mathcal{S}$, the value $|Pr[\mathcal{S}(D[a_i], D[b_j])] - 1/2|$ is negligible.

Predicate Safety. Following a client join request, the server can only evaluate the stored data for the predicate provided by the client. Specifically, given a relation A with encoded elements $D[a_1], .., D[a_n]$, a relation B with encoded elements $D[b_1], .., D[b_m]$, and a predicate $pred$ for which the client provides opening information $open(pred)$, the server can only learn the value $pred(a_i, b_j) \in \{true, false\}$, $\forall i = 1...n$ and $j = 1...m$. Formally, given a predicate $pred$ and corresponding $open(pred)$ revealed by the client, for any other predicate $pred' \neq pred$ for which the server does not have $open(pred')$ and any random values $i \in \{1...n\}$ and $j \in \{1...m\}$, for any probabilistic polynomial time server algorithm $\mathcal{S}$, the value $|Pr[\mathcal{S}_{pred' \neq pred}(open(pred), D[a_i], D[b_j])] - 1/2|$ is negligible.

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 [44] 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. In fact, in Section V we show that solutions that would employ homomorphisms would be several (2-4) orders of magnitude slower than solutions that we propose in this paper.

We assume that server *storage* is cheap. This assumption is supported by recent findings that show the total cost of storage management is orders of magnitude higher than the storage equipment acquisition costs [18].

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.

A. Tools

1) Encryption, Hashing and Random Numbers.: We consider ideal, collision-free hashes, denoted by H. We consider semantically secure (IND-CPA) encryption mechanisms. We denote by $E_K(v)$ the encryption of value v with 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.

2) 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].

3) 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) [22]:

Definition 1: 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) states that:

Definition 2: Given $g, v \in \mathbb{G}$, it is intractable to find $r \in \mathbb{Z}_q$ such that $v = g^r \ mod \ p$.

III. 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 IV we study two predicate and corresponding G, E and J function instances. In Figure I we summarize the symbols used in our solution.

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 $A \in D$.

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

TABLE I
TABLE OF SYMBOLS USED IN OUR SOLUTIONS.

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 $a_i \in A$, $i = 1...$ 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 $E_K(a_i)$. Compute then 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 ($BF(a_i).insert(e_A(v))$). That is, $BF(a_i)$ encodes all the values v that satisfy the predicate P for a_i. 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 A and B on $pred_{FM}$.

$J(A, B, pred_{FM}, r_{AB})$**.** *:* For each element $b_j \in B$, compute $e_A(b_j) = r_{AB}^{O(b_j)} \ mod \ p$. That is, $e_A(b_j)$ denotes the value b_j encoded in the same fashion as the elements encoded in $BF(a_i)$. 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 make explicit here and in the following only the join attributes.

A. Analysis

We now prove the following results.

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

Proof: During the join function J, for each element $b_j \in B$, the server computes the value $e_A(b_j) = r_{AB}^{O(b_j)} \ mod \ p = (g^{y_A/x_B})^{H(b_j)x_B} = g^{H(b_j)y_A} \ mod \ p$. According to the function E, the Bloom filter $BF(a_i)$ of an element $a_i \in A$ stores values of type $g^{H(v)y_A} \ mod \ p$, for all $v \in P(a_i) = \{v | pred_{FMP}(a_i, v) = true\}$. Thus, if $b_j \in P(a_i)$ then $e_A(b_j)$ is stored in $BF(a_i)$.

■

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

Proof: Let us assume that for a relation A with encoded elements $D[a_1], .., D[a_n]$ and a a relation B with encoded elements $D[b_1], .., D[b_m]$, there exists a PPT algorithm $\mathcal{A}$ and a pair of values $i \in \{1...n\}$ and $j \in \{1..m\}$ such that $|Pr[\mathcal{S}(D[a_i], D[b_j])] - 1/2| > \epsilon$. Let the element $D[a_i]$ = $[E_K(a_i)$, $O(a_i)$, $BF(a_i)]$ and let $D[b_j]$ = $[E_K(b_j)$, $O(b_j)$, $BF(b_j)]$. Then, $\mathcal{A}$ can have advantage ϵ only if (i) $E_K(a_i)$ can be distinguished from $E_K(b_j)$ with advantage larger than ϵ, or if (ii) $O(a_i)$ can be distinguished from $O(b_j)$ with advantage larger than ϵ or if (iii) $O(a_i)$ can be searched for in $BF(b_j)$ (the symmetric case is identical). In case (i), we can also build an algorithm that has advantage larger than ϵ against the IND-CPA game of the semantically secure encryption E. Case (ii) cannot occur in an information theoretic sense, since the values $O(a_i)$ and $O(b_j)$ are obfuscated with different random values. For case (iii), let us consider for simplicity that $BF(b_j)$ stores the set $P(b_j)$ as the set of obfuscated values $e_B(v)$, where $v \in P(b_j)$ – instead of using a Bloom filter to encode the $e_B(v)$ values. Then, if $\mathcal{A}$ can find $g^{O(a_i)}$ in the set of values $e_B(v) = g^{H(v)y_B}$, then we can also build an algorithm that defeats the discrete logarithm assumption (see Section II-A3). ■

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

Proof: We need to prove that given a a relation A with encoded elements $D[a_1], .., D[a_n]$, a relation B with encoded elements $D[b_1], .., D[b_m]$, along with client provided opening information $r_{AB} = g^{y_A/x_B} \ mod \ p$ and any random values $i \in \{1...n\}$ and $j \in \{1..m\}$, for any probabilistic polynomial time server algorithm $\mathcal{S}$, the value $|Pr[\mathcal{S}_{pred' \neq pred}(r_{AB}, D[a_i], D[b_j])] - 1/2|$ is negligible. Let Let the element $D[a_i]$ = $[E_K(a_i)$, $O(a_i)$, $BF(a_i)]$ and let $D[b_j]$ = $[E_K(b_j)$, $O(b_j)$, $BF(b_j)]$. As mentioned in the proof of Theorem 1, no advantage can come from the encrypted values $E_K(a_i)$ and $E_K(b_j)$. Moreover, the opening information r_{AB} does not provide information concerning $BF(b_j)$, thus in the following we ignore this Bloom filter.

Similar to the proof of Theorem 1, we make the simplifying assumption that the structure $BF(a_i) = \pi\{g^{H(v)y_A} | \forall v \in P(a_i)\}$, where π is a random permutation. That is, $BF(a_i)$ stores the encoded matching elements for a_i in a random order, instead of further encoding them in a Bloom filter. Then, any advantage of algorithm $\mathcal{S}$ can be either from (i) $O(a_i)$, $O(b_j)$ and r_{AB} or from (ii) $O(b_j)$, r_{AB} and $BF(a_i)$. In case (i), $\mathcal{S}$ can obtain values $g^{H(b_j)y_A}$ and $g^{H(a_i)x_A}$. However, these values cannot be compared without defeating the discrete logarithm assumption. In case (ii), $\mathcal{S}$ can determine if the value $r_{AB}^{O(b_j)}$ is in $BF(a_i)$. However, further comparisons of $r_{AB}^{O(b_j)}$ with the other elements in $BF(a_i)$ cannot occur due to the use of cryptographic hash functions: the outputs values of the hashes of even similar values a_i and b_j will likely differ in half of their bits. ■

B. Discussion and Extensions

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 overhead.

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. For example, if the 10-bit value 513 (binary 1000000001) is expected to occur multiple times (but no more than $m = 4$), 2 bits will be prefixed to its binary representation, to yield its 4 "variants": 001000000001, 011000000001, 101000000001, 111000000001. For each occurrence, one of these variants will be used instead in computing its obfuscated value in E's definition. Additionally, in any Bloom filter, instead of inserting just $v = 513$ (e.g., $BF(a_i).insert(g^{H(v)y_A})$), all its four variants will be inserted. Care needs to be taken for larger m values, as this solution can lead to space blowups or increases in the rate of false positives due to the additional "variant" information inserted in the Bloom filters.

Bloom Filter Sizes.*:* Bloom filters (see Section II-A2) feature a controllable, arbitrarily small rate of false positives for set inclusion tests. 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. Their percentage is then determined by the equations from Section II-A2. Thus, a tradeoff between storage overhead and rate of false positives (and associated additional network traffic) emerges. Larger Bloom filters reduce this rate but require more storage, whereas smaller ones are cheaper to store but will incur additional network traffic

and client-size pruning of non-matching results. Moreover, associated network traffic costs are heavily dependent on the sizes of values in the data tuples. The optimal sizes for Bloom filters becomes thus an application specific decision. For example, for a predicate MMS $= N = 60$ (e.g., in the simple query in Section II), a desired false positive rate of no more than $p_{fp} = 0.8\%$, the equations from Section II-A2 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 overhead. 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).

IV. 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.

A. 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 III. 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 $l = -\frac{n \ln p_{fp}}{(\ln 2)^2}$.

B. Hamming JOIN

It is often important to be able to evaluate Hamming distance on remote data with privacy in untrusted 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 III. 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 III.

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 $A \in D$ [1], 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

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

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 $e_A^k(v) = g^{H(v)y_A(k)} \ mod \ p$. Encode $e_A^k(v)$ 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 III.

Algorithm 1 The J_H algorithm performing a Hamming join between columns A and B.

```
hammingJOIN(A, B, r_A(k), r_B(k), r_k, k = 1..β)
  forall a_i ∈ A and k = 1..β do
    v(a_ik) = r_A(k)^Z(a_ik) mod p;
  forall b_j ∈ B and k = 1..β do
    v(b_jk) = r_B(k)^Z(b_jk) mod p;
    u(b_jk) = r_k^O(b_jk) mod p;
  forall b_j ∈ B do
    forall a_i ∈ A do
      c ← 0;
      for (k ← 1; k ≤ β; k ← k + 1)
        if v(a_ik) ≠ v(b_jk) then
          if BF_ij(A).contains(u(b_jk)) then
            c ← c + 1;
          else
            c ← −1;   #signaldrop
            break;
      if c = −1 then continue;   #drop(a_i, b_j)
      if c ≤ d then output[E_K(a_i), E_K(b_j)];
```

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). See Algorithm 1 for the pseudo-code of J_H.

$J_H(A, B, r_A(k), r_B(k), r_k), k = 1..\beta$**.** *:* 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.

1) Analysis: We now prove the following result for the Hamming join solution proposed above.

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 $e^{-d/\beta}(1 + \frac{d-1}{\beta})$.

Proof: The operation of splitting the permuted elements into β blocks and then comparing the Hamming distance between blocks can be viewed as a balls and bins process, where blocks represent bins and bit-wise differences represent balls. That is, bit-wise differences between any two elements a_i and b_j are thrown uniformly at random into β blocks. If $d_H(a_i, b_j) \leq d$, for two elements a_i and b_j, then using the balls and bins paradigm, the probability of a pair of blocks (a_{ik}, b_{jk}), $k = 1..\beta$, having Hamming distance exactly l is $P_l = \binom{d}{l}\frac{1}{\beta^l}(1 - \frac{1}{\beta})^{d-l}$. The probability of blocks (a_{ik}, b_{jk}) to have Hamming distance smaller than or equal to 1 is then $P_0 + P_1 = (1 - \frac{1}{\beta})^{d-1}(1 + \frac{d-1}{\beta}) \approx e^{-d/\beta}(1 + \frac{d-1}{\beta})$.

We now have to prove that if any pair of blocks from two values a_i and b_j has at most one bit-wise difference, algorithm J_H indeed returns the pair a_i, b_j. To see why this is the case, consider that

$$r_k^{O(b_{jk})} = (g^{y_A(k)/x_B(k)})^{H(b_{jk})x_B(k)} = g^{y_A(k)H(b_{jk})} \ mod \ p.$$

The values v for which $d_H(a_{ik}, v) = 1$ are encoded in a_{ik}'s Bloom filter in the format $g^{y_A(k)H(v)}$. If $BF(a_{ik}).contains(r_k^{O(b_{jk})})$ then $d_H(a_{ik}, b_{jk}) = 1$ and the counter c associated with the pair $\langle a_i, b_j\rangle$ is incremented. If the above condition is not satisfied, then

$$v(a_{ik}) = r_A(k)^{Z(a_{ik})} \ mod \ p = (g^{R_k/z_A(k)})^{H(a_{ik})z_A(k)} =$$

$$= g^{R_k H(a_{ik})} \ mod \ p.$$

Similarly,

$$v(b_{jk}) = g^{R_k H(b_{jk})} \ mod \ p.$$

If $a_{ik} = b_{jk}$, then $v(a_{ik}) = v(b_{jk})$ and the counter c should not be incremented.

■

As an example, for a value $\beta = 800$ and $d = 100$, the probability of finding a matching pair is 99.39%. Besides using a larger β, this probability can be further increased by trading off data storage and privacy. For instance, the server could store for each block a_{ik} of a data element a_i a Bloom filter containing all possible blocks at Hamming distance 2, a Bloom filter for all possible blocks at Hamming distance 3 and so on, until the desired precision level is reached.

Note on Predicate Safety. *:* The reasoning used in Theorem 2 can be easily used to show that the (d_H,G_H,E_H,J_H) solution satisfies the confidentiality requirement. The predicate safety requirement, however, remains only partially true. The server can in fact determine the actual Hamming distance between matching (but encrypted) (a_i, b_j) pairs (satisfying the $d_H(a_i, b_j) < d$ condition). Moreover, the server can also find the Hamming distance of some encrypted (a_i, b_j) pairs for which $d < d_H(a_i, b_j) \leq \beta$. While out of scope here, a solution can be provided for this case by prefixing original a_i, b_j values with a random number of special symbols with controllable Hamming distances.

2) Complexity Analysis: Let T_{encr} be the time to encrypt an element, T_{exp} the time to perform one modular (p) exponentiation, T_{mul} the time to perform a modular (q) multiplication and T_{hash} the time to perform a crypto-hash operation. h is the number of hash functions used to encode elements in a Bloom filter. Then, if t is the number of attributes in the relation, the following results hold.

Lemma 1: The initial client overhead is $tn(T_{encr} + 2\beta(T_{exp} + T_{hash} + T_{mul}) + b(T_{exp} + (h+1)T_{hash} + T_{mul}))$.

Proof: The per-element initial overhead is the sum of three factors: (i) the cost to encrypt the element, (ii) the cost to generate the obfuscated O and Z values and (iii) the cost to generate the β Bloom filters, each storing b/β elements. The cost of storing one element in a Bloom filter is equal to the cost of generating the obfuscated element (a crypto-hash application and an XOR) plus the cost of another h crypto-hashes for generating the bit-wise positions to be set to 1. ■

Lemma 2: The server-side storage overhead is $O(tnN\beta)$, where N is the bit size of p. The computation overhead for a Hamming join operation over two columns of n elements is $O(n\beta(T_{exp} + hT_{hash}))$.

Proof: The original database has $O(tn)$ elements. Since a Bloom filter encoding s numbers takes $O(s)$ bits (see Section II) and the number of values y that are at Hamming distance 1 from a bit string of length b/β is b/β, the storage required by the β Bloom filters of an element is $O(b)$. The first result follows then from the observation that each of the β blocks of an element, stored as output of the O and Z functions, requires $O(b/\beta)$ bits.

The second result is due to the fact that the Hamming join computation overhead consists of β Bloom filter searches for all the blocks for each of n^2 pairs of elements from the two joined columns. ■

See Section V for a discussion on why for practical purposes a single crypto-hash application may be enough to replace the h Bloom filter hashes.

3) Extensions:

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).

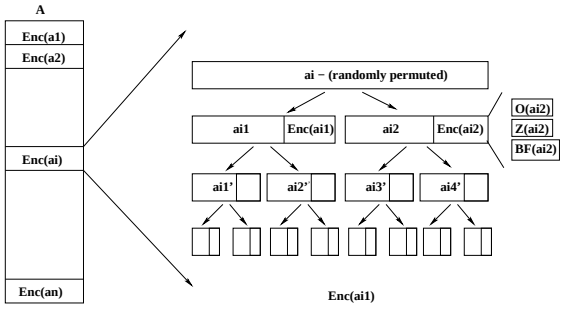

Fig. 1. Data structure for hierarchical private Hamming distance. The hierarchy in the example has three layers. On layer l each element is split into 2^l blocks. The red rectangles denote the O, Z and BF structures associated with each block. Together with $E_K(a_i)$, they are the only values stored on the server.

Arbitrary Distances.*:* One drawback of the previous solution is the fixed nature of the Hamming distance d that can be considered. To accommodate a different distance, additional metadata would need to be generated by the client accordingly. Instead, it would be desirable to provide a single solution for any distances. In the following we show how to extend the above solution for arbitrary distances.

For this purpose, the encoding algorithm E_H is modified to perform a hierarchical generalization of the previous shuffle-and-divide pre-processing step. The new algorithm, E_{FH} calls E_H $\log b$ times, for $\beta = \{1, 2, 2^2, .., b\}$. As a result, each data element has $\log b$ layers of metadata, one for each value of β (the individual block size). Figure 1 illustrates the output of function E_{FH} for two layers.

The extended join algorithm, J_{FH}, is initially executed by the client and takes as an input parameter the distance d of interest. Based on d and the desired miss rate, J_{FH} decides upon the appropriate layer of metadata on which the join should be performed and calls J_H, to be executed on the server, with the corresponding parameters, detailed earlier. For instance, if the join is done on the metadata for the layer corresponding to the value $e = 2^{\lceil \log d \rceil + 1}$, then the miss rate can be upper bound by 8%. The following result is then straightforward.

Theorem 5: (Overheads) The server-side storage overhead for supporting arbitrary distance Hamming joins increases by a factor of $\log b$ over the Hamming join overhead. The computation and traffic overhead remain the same.

Note that the server-side storage overhead for supporting arbitrary distance Hamming joins increases by a factor of $\log b$ over the Hamming join overhead. The computation and traffic overhead remain the same.

Variable Data Sizes.*:* For illustration purposes, the algorithms above have been presented considering elements of the same, known size. We now show how to deploy them also for data columns with values of different representation bit sizes.

In the pre-processing stage, given an alphabet $\mathcal{A} = \{\alpha_0, \alpha_1 ... , \alpha_{u-1}\}$, the client introduces an additional symbol, α_*. It then represents each of the $u+1$ alphabet symbols on $u+1$ bits, with $\alpha_i = 2^i$ and $\alpha_* = 2^u$.

Let l be the expected maximum symbol length of elements stored in the database and $l_A \leq l$ the symbol length of elements in column A. The client then reduces this problem to the previous setting by "padding" each column with α_* symbols up to length l. For example, it appends $(l - l_A)$ symbols of type α_* to each element in A. The padding is done before the random bit-wise permutation of the elements, to prevent the server from differentiating the padding symbols.

Hamming distance predicates will then be rewritten accordingly. For example, to find all pairs of elements from columns A and B whose Hamming distance is less than or equal to d, the client searches for all pairs at distance $d + |l_A - l_B|$. This method has an additional padding-related storage overhead that depends on the distribution of the data column symbol lengths. It functions best if this distribution is very narrow. For flatter distributions, other non-padding mechanisms could be envisioned.

C. Additional Examples

We illustrated above two predicate instance extremes: one very simple and straightforward range predicate and a more complex Hamming distance scenario, requiring custom, predicate - specific mechanisms. In the following we list just a few more (and some of their application domains), straightforward to deploy using the solution above.

$p(x, y) := (f(x, y)\ \mathcal{R}\ r)$**.** Financial, geographic location queries. For example $f(x, y) = x^2 + y^2$ and $\mathcal{R} =$'<'.

$p(x, y) := (\frac{x}{y} \in \mathtt{Z};\quad x, y \in \mathtt{Z})$**.**

$p(x, y) := (x \equiv y^e\ mod\ q)$**.** Cryptography.

$p(x, y) :=$ **(antibiotic** x **matches bacteria** y**).** Health care diagnostics.

$p(x, y) :=$ **(patient** x **has disease** y**).** Census, health care.

V. Experimental Results

Implementation Details.*:* We conducted our experiments using a 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 overhead, 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 at up to `150MB/sec`, shown in Figure 2. 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 [32], 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 our benchmarks, shown in Figure 3, we have concluded that 512-bit modular exponentiations (with 160 bit exponents) take 274usec while 512-bit modular multiplications take only 687nsec.

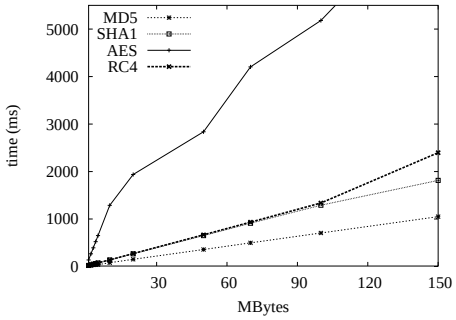

Fig. 2. Comparison of RC4, 3DES, MD5 and SHA1. MD5 can support a throughput of up to 150MB/sec in our setup.

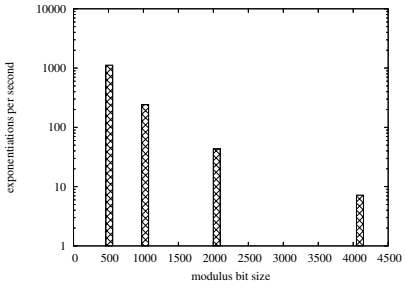

Fig. 3. Modular exponentiation costs when the modulus size ranges from 512 to 4096. The y axis is shown in logarithmic scale.

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. For each value from the data set from [2] there are 25 SNPs, whose values are drawn from the four nucleotides, A, C, G, and T. Thus, each Bloom filter stores 100 values (MMS=100). Note that we have simplified the SNP evaluation for the purposes of illustration, as each SNP is actually composed of two nucleotides (one from the father and one from the mother. This effectively doubles the number of bits needed to represent them. 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 [41] and Caltech 101 [17] databases. A set of images are annotated with scores for lightness, hue or colors of interest [16], [20]. 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.

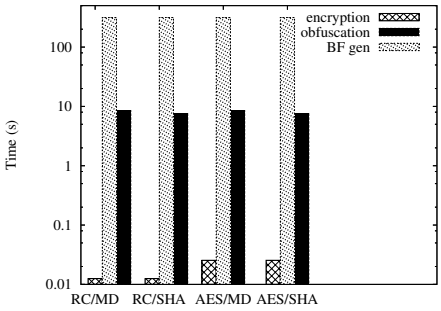

Fig. 4. Client computation overhead: Initial database generation cost for 100000 elements, using various combinations of encryption and hashing algorithms. Total cost is around 5 minutes.

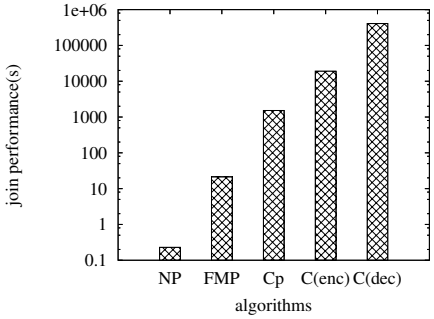

Fig. 6. Server cost: Join costs for columns of 10000 elements. Our solution is 2-4 orders of magnitude faster than other solutions that use 1024-bit modular operations.

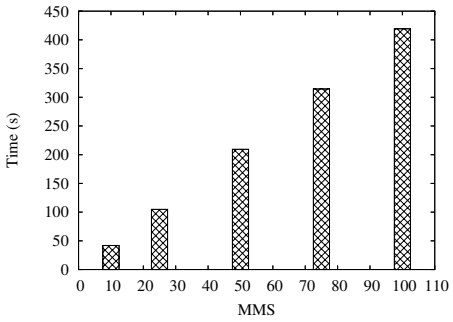

Fig. 5. Client computation overhead: Bloom filter effect: The cost of the RC4/MD5 combination when MMS increases from 10 to 100 – linear dependency between this cost and MMS.

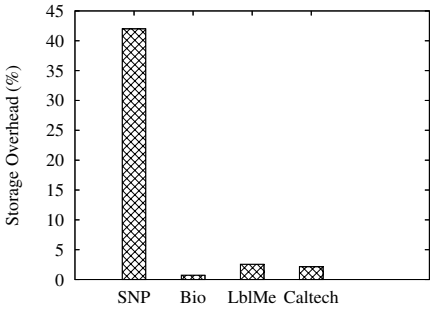

Fig. 7. Server cost: Bloom filter storage overhead, as percentage of the size of the cleartext data. The overhead is 42% for SNP databases, but under 3% for fingerprint or image databases.

Client Computation Overheads. *:* We now describe our investigation of the initial client pre-processing step. Of interest were first the computation overhead 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 100 items each. Figure 4 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 5 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, during the computation of the initial data structures.

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 [41] and Caltech 101 [17] databases. For each image we deployed 1024-bit Bloom filters ($h = 12$ hashes) with MMS=100. 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 [39]. This comparison is motivated by recent related work (e.g., [19]) 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. Finally, we have also compared our solution against a base case with no privacy: the server stores the data in cleartext, performs joins on request from the client and returns the exact results.

Figure 6 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 (iii) $C(dec)$, that uses one 1024-bit RSA decryption operation and (iv) NP, a no-privacy solution where the data is stored in clear at the server. The y axis represents the time in logarithmic scale. The first bar shows the cost of the base case with no privacy (NP). The second 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 (third bar) taking 1525s, three orders of magnitude faster than $C(enc)$ (fourth bar), taking 19168s and four orders of magnitude faster than $C(dec)$ (fifth bar), taking 408163s. One reason for the large overhead of the modular multiplications in the Paillier system (used also in [19]) 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], [32]. Note that as expected our solution is two orders of magnitude (21.3s) less efficient than the trivial solution (0.23s) that stores the data in cleartext at the server.

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 storing the cleartext data. The size of the O value for each element is $N = 512$ bits, which is small and data-independent. Finally, Figure 7 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%. Note that the total space storage overhead for 100000 items is 18.31MB.

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.

Summary. The experimental evaluation of the main overhead show that they are reasonable. The small initial costs of generating metadata and transferring the database are only incurred once. The storage overhead of the metadata is small and independent of the size of the data items. Finally, the cost of executing 100 million private FMPs is 2-4 orders of magnitude faster than that of implementations using the homomorphic properties of certain asymmetric encryption algorithms to provide privacy.

Hamming Join vs. Generic Solution: To understand the advantages of the Hamming Join solution when compared with the generic solution, we consider our fingerprint matching problem. Each fingerprint has 100 features (b=100) and we are interested in matching fingerprints that have up to 4 different features (d=4). In the Hamming Join solution, we consider a value of β=16, that is, we divide the 100 bit feature strings into 16 blocks, of 7 bits each. Let the total space allocated for the Bloom filter associated with a fingerprint be 1024 bits. Then, for each of the 16 blocks of bits of a fingerprint, the associated Bloom filter has 64 bits. Each Bloom filter has to store 7 values, leading to a false positive rate per block Bloom filter $fp \approx 0.62^{64/7} = 0.012$. The overall false positive rate (probability of returning a pair with distance larger than 4) is upper bounded by the false positive rate in *any* of the 16 Bloom filters, $fp_{total} = 16 \times fp = 0.2022$. Let us now use the generic solution to solve the same problem. A single Bloom filter, needs to store $\sum_{i=1}^{4} \binom{100}{i}$ = 4087975 elements. To achieve the same false positive rate as the one achieved by the Hamming Join solution, the generic solution's Bloom filters have to have 13,596,925 bits. Thus, the storage overhead of the generic solution is more than 13000 times larger than the one of the Hamming Join approach.

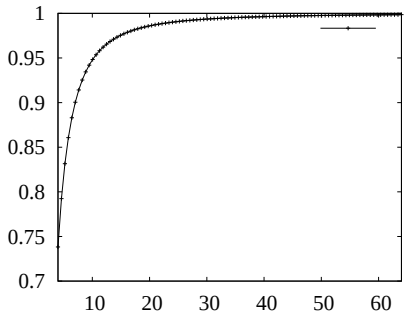

Fig. 8. Hamming Join: Completeness (the probability of returning a matching pair) as a function of the value of β – the number of blocks used for dividing feature strings.

Figure 8 shows for the same problem, the completeness of the result of the Hamming join (the probability of returning a matching pair) as a function of the value of β. Note that for β=16, this probability exceeds 0.975.

VI. Related Work

The paradigm of providing a database as a service recently emerged [25] as a viable alternative, likely due in no small part to the dramatically increasing availability of fast, cheap networks. Given the global, networked, unreliable, possibly hostile nature of the operation environments, providing security and integrity assurances has become essential.

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], [26], [27], [29], [30], [33], [37], [38], [40], [42]. 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], [31].

Hacigumus et al. [24] 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 claimed to be 1-out-of-s where s is the partition size. This balances a tradeoff between client-side 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. We believe this client load requirement to defeat the very purpose of data outsourcing.

Similarly, Hore et al. [28] 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 tradeoff for the index is discussed.

Ge and Zdonik [19] 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). In Section V we show that the overhead of the operations used in [19] is very large, exceeding the overhead of FMP predicate joins by three orders of magnitude.

The problem of searching on encrypted data has also been studied extensively. The setting of this problem consists of clients that need to store encrypted documents on an untrusted server and later wish to privately retrieve the documents containing certain encrypted keywords, without revealing to the server the keywords of interest. Song et al. [45] 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. [23] 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 statistics over searched keywords.

Goh [21] proposed the notion of a "secure index", which is 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. Since this solution requires knowledge of the trapdoor associated with the searched item, secure indexes are not flexible enough to be used for private joins on outsourced data.

Yang et al. [46] study the complementary problem of authenticating the results of joins in outsourced databases, where the server needs to construct a proof of correctness, which can be verified by the client using the data owners signature. The work introduces three join algorithms and demonstrates experimentally that they outperform two benchmark algorithms, by several orders of magnitude, on all performance metrics. We note that this work complements our solutions: we provide the privacy of the outsourced data and of the returned results, while the work of Yang et al. [46] provides proofs of correctness and completeness of the results.

Summary. Previous related work focuses on the problem of performing private search, range and aggregate queries on encrypted data. In this paper we address a different problem, of privately performing join operations on encrypted attributes, using arbitrary FMP predicates. While previous work cannot be used to solve this problem, increased outsourced database functionality can be provided when our solutions are used in conjunction with existing results.

VII. 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.

VIII. Acknowledgments

We would like to thank the anonymous reviewers for their detailed feedback.

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 Workshop on Database Security*, 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. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. Cryptology ePrint Archive, Report 2004/051, 2004. http://eprint.iacr.org/.
[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] Einar Mykletun and Maithili Narasimha and Gene Tsudik. Signature Bouquets: Immutability for Aggregated/Condensed Signatures. In *Proceedings of the European Symposium on Research in Computer Security ESORICS*, pages 160–176, 2004.
[16] R. Fagin. Fuzzy queries in multimedia database systems. In *PODS '98: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 1–10, 1998.

[17] L. Fei-Fei, R. Fergus, and P. Perona. Learning generative visual models from few training examples: an incremental bayesian approach tested on 101 object categories. In *Proceedings of IEEE Workshop on Generative-Model Based Vision*, 2004.

[18] Gartner, Inc. Server Storage and RAID Worldwide. Technical report, Gartner Group/Dataquest, 1999. www.gartner.com.

[19] T. Ge and S. B. Zdonik. Answering aggregation queries in a secure system model. In *Very Large Databases (VLDB)*, pages 519–530, 2007.

[20] 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.

[21] E. Goh. Secure indexes. Cryptology ePrint Archive, Report 2003/216, 2003. http://eprint.iacr.org/2003/216/.

[22] O. Goldreich. *Foundations of Cryptography*. Cambridge University Press, 2001.

[23] P. Golle, J. Staddon, and B. Waters. Secure conjunctive keyword search over encrypted data. In *Proceedings of ACNS*, pages 31–45. Springer-Verlag; Lecture Notes in Computer Science 3089, 2004.

[24] 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.

[25] H. Hacigumus, B. R. Iyer, and S. Mehrotra. Providing database as a service. In *IEEE International Conference on Data Engineering (ICDE)*, 2002.

[26] J. Hale, J. Threet, and S. Shenoi. A framework for high assurance security of distributed objects, 1997.

[27] E. Hildebrandt and G. Saake. User Authentication in Multidatabase Systems. In R. R. Wagner, editor, *Proceedings of the Ninth International Workshop on Database and Expert Systems Applications, August 26–28, 1998, Vienna, Austria*, pages 281–286, Los Alamitos, CA, 1998. IEEE Computer Society Press.

[28] B. Hore, S. Mehrotra, and G. Tsudik. A privacy-preserving index for range queries. In *Proceedings of VLDB*, 2004.

[29] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *IEEE Symposium on Security and Privacy. Oakland, CA*, pages 31–42, 1997.

[30] S. Jajodia, P. Samarati, V. S. Subrahmanian, and E. Bertino. A unified framework for enforcing multiple access control policies. In *SIGMOD*, 1997.

[31] K. LeFevre, R. Agrawal, V. Ercegovac, R. Ramakrishnan, Y. Xu, and D. J. DeWitt. Limiting disclosure in hippocratic databases. In *Proceedings of the International Conference on Very Large Databases VLDB*, pages 108–119, 2004.

[32] A. K. Lenstra and E. R. Verheul. Selecting cryptographic key sizes. *J. Cryptology*, 14(4):255–293, 2001.

[33] Li, Feigenbaum, and Grosof. A logic-based knowledge representation for authorization with delegation. In *PCSFW: Proceedings of the 12th Computer Security Foundations Workshop*, 1999.

[34] Maithili Narasimha and Gene Tsudik. DSAC: integrity for outsourced databases with signature aggregation and chaining. Technical report, 2005.

[35] E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and integrity in outsourced databases. In *ISOC Symposium on Network and Distributed Systems Security NDSS*, 2004.

[36] E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and integrity in outsourced databases. In *Proceedings of Network and Distributed System Security (NDSS)*, 2004.

[37] M. Nyanchama and S. L. Osborn. Access rights administration in role-based security systems. In *Proceedings of the IFIP Workshop on Database Security*, pages 37–56, 1994.

[38] S. L. Osborn. Database security integration using role-based access control. In *Proceedings of the IFIP Workshop on Database Security*, pages 245–258, 2000.

[39] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proceedings of EuroCrypt*, 1999.

[40] D. Rasikan, S. H. Son, and R. Mukkamala. Supporting security requirements in multilevel real-time databases, citeseer.nj.nec.com/david95supporting.html, 1995.

[41] B. Russell and A. T. an William T. Freeman. LabelMe: the open annotation tool. http://labelme.csail.mit.edu/.

[42] R. S. Sandhu. On five definitions of data integrity. In *Proceedings of the IFIP Workshop on Database Security*, pages 257–267, 1993.

[43] R. Sion. Query execution assurance for outsourced databases. In *Proceedings of the Very Large Databases Conference VLDB*, 2005.

[44] R. Sion and B. Carbunar. On the Practicality of Private Information Retrieval. In *Proceedings of the Network and Distributed Systems Security Symposium*, 2007. Stony Brook Network Security and Applied Cryptography Lab Tech Report 2006-06.

[45] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy (S&P 2000)*. IEEE Computer Society, 2000.

[46] Y. Yang, D. Papadias, S. Papadopoulos, and P. Kalnis. Authenticated join processing in outsourced databases. In *SIGMOD Conference*, pages 5–18, 2009.

Bogdan Carbunar is a principal staff researcher in the pervasive platforms and architectures lab of the Applied Research Center at Motorola. His research interests include distributed systems, security and applied cryptography. He has been on the program committee of conferences such as the ISOC Network and Distributed Systems Security Symposium (NDSS), the IEEE International Conference on Multimedia and Expo (ICME) and Financial Cryptography (FC). He holds a Ph.D. in Computer Science from Purdue University.

Radu Sion heads the Stony Brook Network Security and Applied Cryptography (NSAC) Lab. His research interests include Information Assurance and Efficient Computing. He builds systems mainly, but enjoys elegance and foundations, especially if of the very rare practical variety. Sponsors and collaborators include IBM, IBM Research, NOKIA, Xerox, as well as the National Science Foundation which awarded also the CAREER Award. Radu is on the steering board and organizing committees of conferences such as NDSS, Oakland S&P, CCS, USENIX Security, SIGMOD, ICDE, FC and others.