Aggregation Queries in the Database-As-a-Service Model

Einar Mykletun and Gene Tsudik
Computer Science Department
University of California, Irvine
{mykletun,gts}@ics.uci.edu

Abstract

In the Database-As-a-Service (DAS) model, clients store their database contents at servers belonging to potentially untrusted service providers. To maintain data confidentiality, clients need to outsource their data to servers in encrypted form. At the same time, clients must still be able to execute queries over encrypted data. One prominent and fairly effective technique for executing SQL-style range queries over encrypted data involves partitioning (or bucketization) of encrypted attributes.

However, executing aggregation-type queries over encrypted data is a notoriously difficult problem. One well-known cryptographic tool often utilized to support encrypted aggregation is homomorphic encryption; it enables arithmetic operations over encrypted data. One technique based on a specific homomorphic encryption function was recently proposed in the context of the DAS model. Unfortunately, as shown in this paper, this technique is insecure against ciphertext-only attacks. We propose a simple alternative for handling encrypted aggregation queries and describe its implementation. We also consider a different flavor of the DAS model which involves mixed databases, where some attributes are encrypted and some are left in the clear. We show how range queries can be executed in this model.

1 Introduction

The Database-As-a-Service (DAS) model was introduced by Haĉigümus, et al. in [1] and, since then, has received a lot of attention from the research community. DAS involves clients outsourcing their private databases to database service providers (servers) who offer storage facilities and necessary expertise. Clients, in general, do not trust service providers with the contents of their databases and, therefore, store the databases in encrypted format. The central challenge is how to enable an untrusted service provider to run SQL-style queries over encrypted data.

In [1], Haĉigümus, et al. suggested a method for supporting range queries in the DAS model. Since encryption by itself does not facilitate range queries, [2] involves bucketizing (partitioning) attributes upon which range queries will be based. This involves dividing the range of values in the specific domains of the attribute into *buckets* and providing explicit labels for each partition. These bucket labels are then stored along with the encrypted tuples at the server. Based on the same bucketization strategy, the follow-on work in [3] addresses aggregation queries in DAS by proposing the use of a particular homomorphic encryption function. In general, homomorphic encryption is a technique that allows entities who only possess encrypted values (but no decryption keys) to perform certain arithmetic operations directly over these values. For example, given two values $E(A)$ and $E(B)$ encrypted under some homomorphic encryption function $E()$, one can efficiently compute $E(A+B)$. It is easy to see that such functions can easily support SUM operations over a desired range of values.

In this paper we show that the homomorphic encryption scheme in [3] is insecure by demonstrating its susceptability to a ciphertext-only attack. This makes it possible for the server (or any other party with access to the encrypted data) to obtain the corresponding cleartext. We propose a very simple alternative for handling aggregation queries at the server, which does

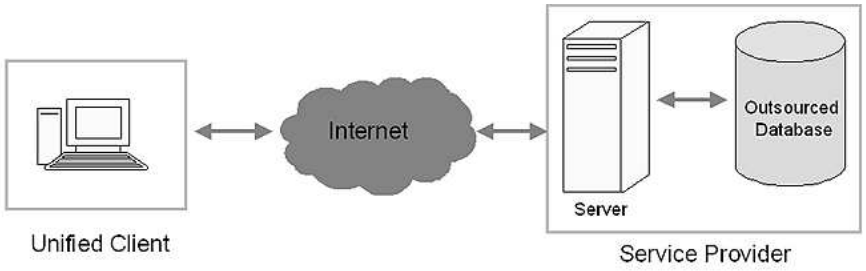

Figure 1. Database-As-a-Service Overview

not involve homomorphic encryption functions. We further describe the protocols for formulating and executing queries as well as updating encrypted tuples. We then focus on a variant of DAS which has not been explored thus far: the so-called mixed DAS model, where some attributes are sensitive (and thus stored encrypted) while others are not (and are thus left in the clear).

Organization: This paper is organized as follows: Section 2 describes the salient features of the DAS model and the bucketization technique. Section 3 introduces homomorphic encryption functions and describes our attack on the scheme in [3]. Section 4 describes our simple solution for supporting aggregation-style queries in the DAS model. and Section 5 addresses query processing in the mixed-DAS model. Section 6 overviews related work and Section 7 concludes the paper.

2 The DAS Model

The Database-As-a-Service (DAS) model is a specific instance of the well-known Application-As-a-Service model. DAS was first introduced by Haĉigümus, et al. [1] in 2002. It involves clients storing (outsourcing) their data at servers administered by potentially untrusted service providers. Although servers are relied upon for the management/administration and availability of clients' data, they are generally not trusted with the actual data contents. In this setting, the main security goal is to limit the amount of information about the data that the server can derive, while still allowing the latter to execute queries over encrypted databases. (A related issue is how to maintain authenticity and integrity of clients' outsourced data; this has been addressed by the related work in [4, 5, 6].)

Before outsourcing, a DAS client is assumed to encrypt its data under a set of secret keys. These keys are, of course, never revealed to the servers. The client also creates, for each queri-able attribute, a bucketization index and accompanying metadata to help in formulating queries. For every encrypted tuple, each attribute index is reflected in a separate label (bucket id) which is given to the server. Table 1 shows an example of partitioning for a *salary* attribute. Clients maintain the metadata describing the partitions.

Although the term "DAS client" generally refers to an organizational entity, the actual client who queries the outsourced data may be a weak device, such as a cell-phone or a PDA. Thus it is important to minimize both bandwidth and computation overhead for such clients.

Table 1. Bucketization

employee.salary	
Partition	**ID**
[0,25K]	41
(25K, 50K]	64
(50K, 75K]	9
(75K, 100K]	22

2.1 Bucketization

There are two basic strategies for selecting bucket boundaries: equi-width and equi-depth. With the former, each bucket has the same range. Table 1 is an example of equi-width bucketization where each partition covers 25K. However, if the attribute is distributed non-uniformly, this bucketization technique essentially reveals (to the server) the accurate bucket-width histogram of the encrypted attribute. In contrast,

equi-depth bucketization attempts to avoid this problem by having each bucket contain the same number of items, thereby hiding the actual distribution of values. The downside of this approach is that, in the presence of frequent database updates, the equi-depth partition needs to be adjusted periodically. This requires additional (and non-trivial) interaction between the server and the client (database owner).

Although useful and practical, bucketization has an unavoidable side-effect of privacy loss since labels (bucket id-s) disclose some information about the cleartext. Unless there are as many buckets as there are distinct values in the domain of an attribute, some statistical information about the underlying data is disclosed through bucket id-s. Some recent results [7, 8] analyze and estimate the loss of privacy due to bucketization. These results show that, although some degree of privacy is invariably lost (since statistical information is revealed), only very limited information can be deduced from encrypted tuples and associated labels [8].

Table 2 shows a subset of a table *employee* with the attributes: *employee id*, *age*, and *salary*. The encrypted version of the table, stored at the server, is shown in Table 3. It contains the fields: *etuple*, bucket identifiers each of the original attributes, and additional ciphertext values denoted by *fieldname*^{h} that will be utilized when the server computes aggregation queries (see Section 3). If the server aggregates data during range queries, it will be unable to include values from encrypted tuples. It should therefore be possible for the service provider to execute certain commands upon the sets selected during range queries, and the next section describes the use of homomorphic encryption which allows arithmetic operations directly over ciphertexts.

Table 2. Plaintext Relation

eid	*age*	*salary*
12	40	58K
18	32	65K
51	25	40K
68	27	76K

Table 3. Relation *employee*

etuple(encrypted tuple)	*eid*^{id}	*age*^{id}	*salary*^{id}	*age*^{h}	*salary*^{h}
%j#9*&JbB@...	72	51	9	52	73
P 5g4*H$j0aO...	72	3	9	29	65
X!f(63¡gl03...	26	33	64	90	43
[f3+Wb5P@r-Cs...	85	33	22	81	38

2.2 Query Processing

A client's SQL query is transformed, based upon metadata, into server-side and client-side queries (Q^s and Q^c). The first is executed by the server over encrypted data. The results are returned to the client where they are decrypted and serve as input to the second query. When Q^c is run at the client, it produces the correct results. As described below, the results from executing Q^s form a superset of those produced by Q^c. In other words, after the decryption of the tuples returned by Q^s, Q^c filters out extraneous tuples.

The use of bucketization limits the granularity of range limits in server-side queries. This is because the server cannot differentiate between tuples within the same bucket (i.e., tuples with identical labels). Therefore, server-side queries are further decomposed into *certain* and *maybe* queries, denoted by Q^s_c and Q^s_m, respectively. The former will select tuples that certainly fall within the range specified in the query and its results can be aggregated at the server. Q^s_m selects *etuples* corresponding to records that may qualify the conditions of the range query, but which cannot be determined without decryption and further selection by the client. This query's result set consists of the *etuples* from the border buckets in the range query. Upon receiving the two result sets the client runs query Q^c to produce the final results.

Figure 2 illustrates the procedure whereby a client query Q is decomposed into Q^c, Q^s_c, Q^s_m. Using Table 3 data as an example, if a query specified the range of salaries between \$30-75K, then Q^s_c would identify bucket 9 and Q^s_m bucket 64. This query-splitting necessitates post-processing by the client – running Q^c against the results returned by the server after running Q^s. We refer to [2] for details about the query-splitting.

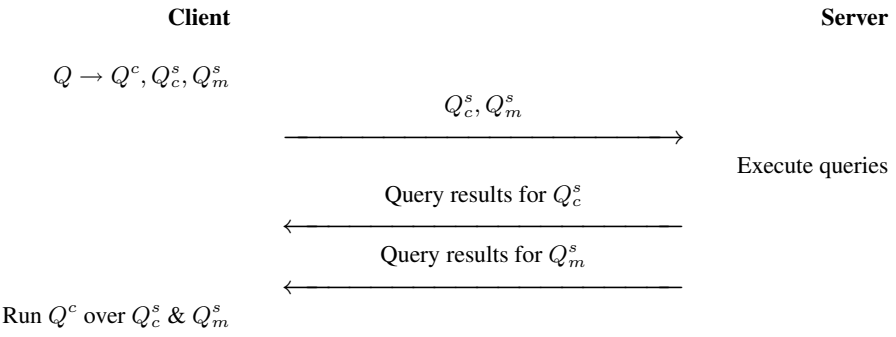

Figure 2. Transformation of Client Query

3 Querying over Encrypted Data

The bucketization technique described above enables a server to run range queries over encrypted tuples. However, we have yet to describe any useful functions that can be computed in conjunction with such range queries. This section focuses on aggregation queries over encrypted data. More specifically, we are interested in mechanisms for computing the most rudimentary (and popular) aggregation function: SUM over a set of tuples selected as a result of a range query.

3.1 Homomorphic Encryption

A homomorphic encryption function allows manipulation of two (or more) ciphertexts to produce a new ciphertext corresponding to some arithmetic function of the two respective plaintexts, without having any information about the plaintext or the encryption/decryption keys. For example, if $E()$ is multiplicatively homomorphic, given two ciphertext $E(A)$ and $E(B)$, it is easy to compute $E(A * B)$. Whereas, if $E()$ is additively homomorphic, then computing $E(A + B)$ is also easy. One well-known example of a multiplicatively homomorhic encryption function is textbook RSA.[1] An example of an additively homomorhic encryption function is Paillier [10].

In more detail (as described in [3]) a homomorphic encryption function can be defined as follows:

> Assume $\mathcal{A}$ is the domain of unencrypted values, $\mathcal{E}_k$ an encryption function using key k, and $\mathcal{D}_k$ the corresponding decryption function, i.e., $\forall a \in \mathcal{A}, \mathcal{D}_k(\mathcal{E}_k(a)) = a$. Let α and β be two (related) functions. The function α is defined on the domain $\mathcal{A}$ and the function β is defined on the domain of encrypted values of $\mathcal{A}$. Then $(\mathcal{E}_k, \mathcal{D}_k, \alpha, \beta)$ is defined as a homomorphic encryption function if $\mathcal{D}_k(\beta(\mathcal{E}_k(a_1), \mathcal{E}_k(a_2), ..., \mathcal{E}_k(a_m))) = \alpha(a_1, a_2, ..., a_m)$. Informally, $(\mathcal{E}_k, \mathcal{D}_k, \alpha, \beta)$ is homomorphic over domain $\mathcal{A}$ if the result of the application of function α on values may be obtained by decrypting the result of β applied to the encrypted form of the same values.

Homomorphic encryption functions were originally proposed as a method for performing arithmetic computations over private databanks [11]. Since then, they have become part of various secure computation schemes and more recently, homomorphic properties have been utilized by numerous digital signature schemes [12, 4]. As mentioned above, some encryption functions are either additively or multiplicatively homomorphic. An open problem in the research community is whether there are any cryptographically secure encryption functions that are both additively and multiplicatively homomorphic. (It is widely believed that none exist.)

3.2 Homomorphic Function in [3]

The homomorphic encryption function proposed in [3] is based upon the so-called Privacy Homomorphism (PH) scheme [11]. PH is a symmetric encryp-

[1] In practice, RSA encryption is not homomorphic since plaintext is usually padded and encryption is made to be *plaintext-aware*, according to the OAEP specifications [9].

tion function with claimed security based on the difficulty of factoring large composite integers (similar to RSA). PH encryption works as follows:

- **Key Setup**:
  $k = (p, q)$, where p and q are large secret primes. Their product: $n = pq$ is made public.

- **Encryption**: Given plaintext (an integer) a,
  $\mathcal{E}_k(a) = C = (c_1, c_2) = (a \pmod p) + R(a) \times p$, $a \pmod q + R(a) \times q)$, where $a \in \mathcal{Z}_n$ and $R(x)$ is a pseudorandom number generator (PRNG) seeded by x.

- **Decryption**: Given ciphertext (c_1, c_2),
  $\mathcal{D}_k(c_1, c_2) = (c_1 \mod p)qq^{-1} + (c_2 \mod q)pp^{-1} \pmod n$

This encryption function exhibits both additive and multiplicative properties (component-wise). The addition of "noise" – through the use of $R(x)$ – is done in multiples of p and q, respectively, which is meant to make encryption non-deterministic and make it more difficult for an attacker to guess the secret key k. However, as we show below, this actually makes it easier to attack this encryption scheme through their extensions to the original homomorphic scheme.

There are several types of textbook-style attacks against encryption functions [13]. At the very least, an encryption function is required to withstand the most rudimentary attack type – *ciphertext-only attack*. Such an attack occurs when the adversary is able to discover the plaintext (or worse, the encryption key) while only having access to ciphertexts (encrypted values). We now show that the above PH-based encryption is subject to a trivial ciphertext-only attack, which results not only in the leakage of plaintext, but also in recovery of the secret keys. The attack is based on the use of a well-known Greatest Common Divisor (GCD) algorithm.

To make the attack work we make one simple assumption: that there are repeated (duplicate) plaintext values. This assumption is clearly realistic since it holds for most typical integer attributes, e.g., salary, age, date-of-birth, height, weight, etc. Of course, PH encryption ensures that identical plaintext values are encrypted into different ciphertexts, owing to the addition of *noise*.

We denote a repeated plaintext value by M and two corresponding encryptions of that value as $C' = (c'_1, c'_2)$ and $C" = (c_1", c_2")$. Let R' and $R"$ represent the respective random noise values for the first half of each ciphertext. Recall that: $c'_1 = M \pmod p + R' \times p$ and $c_1" = M \pmod p + R" \times p$. Then, we have: $c'_1 - c_1" = R' \times p - R" \times p = (R' - R") \times p$.

Since R' and $R"$ are relatively small[2] factoring $(c'_1 - c_1")$ is trivial. Hence, obtaining p (and, likewise, q) is relatively easy. Moreover, we observe that, even if factoring $(c'_1 - c_1")$ were to be hard (which it is not), it is equally trivial to compute the greatest common divisor of $(c'_1 - c_1")$ and n. Note that $p = GCD(n, c'_1 - c_1") = GCD(pq, (R' - R")p)$.

This attack can be performed by the server by simply iterating through pairs of ciphertexts corresponding to a single database attribute, until a pair of duplicate-plaintext ciphertexts are found. In general, given t ciphertexts (for a given attribute), the server would have to perform at most $O(t^2)$ GCD computations before computing p and q. Once p and q are obtained, decrypting all ciphertexts is an easy task.

There are other weaknesses associated with the homomorphic scheme proposed in [3]. An extension is for acommodating encryption of negative numbers stipulates how values should be transformed prior to encryption. However, when such ciphertexts are multiplied, decryption simply fails! A separate issue arises due to the use of noise introduced through the use of $R(x)$. This function produces a pseudo-random number used as a multiplicative coefficient of p and q, both of which are already large integers. Therefore, the resulting ciphertexts increase in size, taking significant storage at the server.

3.3 Other Homomorphic Encryption Functions

Since the encryption function proposed in [3] is insecure, it is worthwhile to investigate whether there are other homomorphic encryption functions that can replace it. Recent cryptographic literature contains several encryption schemes that exhibit the additively homomorphic property. (Note that we are not

[2] If R_i values were large, then the resulting ciphertexts would become even larger than their current size, especially since encryption does not include the noise component in its modular reductions.

as interested in multiplicatively homomorphic property because multiplication is not as frequent as addition in aggregation queries). Candidates include cryptosystems proposed by Paillier [10], Benaloh [14], the elliptic-curve variant of ElGamal [15] and Okamoto/Uchiyama [16]. One common feature of these schemes is that, unlike PH encryption, they are all provably secure public-key cryptosystems based upon solid number-theoretic assumptions. An unfortunate consequence is that ciphertexts tend to get rather large, and the operation of combining ciphertexts can be computationally intensive. This is problematic when dealing with computationally weak clients, such as cellphones or PDAs.

One very different alternative is a symmetric encryption function recently proposed by Castelluccia, et al. [17] in the context of secure aggregation in sensor networks. This function requires no number-theoretic assumptions, is very efficient and incurs only a minor ciphertext expansion [17]. It is based on a variant of a well-known counter (CTR) mode [13] of encryption and can be used in conjunction with any block cipher, such as Triple-DES or AES [18, 19]. (The only notable difference is that it uses an arithmetic addition operation, instead of exclusive-OR to perform the actual encryption. The keystream is generated according to the normal counter mode.)

All of the above homomorphic encryption functions are secure, when used correctly. However, we show – in Section 4 – that there are simpler mechanisms for achieving aggregation over encrypted data.

4 Proposed Approach

With the exception of total summation queries, most aggregation queries are typically predicated upon a range selection over one or more attributes. However, if all tuple attributes are encrypted, aggregation is impossible without some form of bucketization or partitioning. Assumming a bucketization scheme (as described in Section 2.1), we now describe a trivial alternative for supporting aggregation-style queries. This technique does not require any homomorphic encryption and demands negligible exta storage as well as negligible amount of computation.

Our approach involves the data owner precomputing aggregate values, such as SUM and COUNT, for each bucket, and storing them in encrypted form at the server. This allows the server, in response to a bucket-level aggregation query, to directly reply with such encrypted aggregate values, instead of computing them on-the-fly at query processing time. The encrypted bucket-level aggregate values can be stored separately. Table 4 shows sample table with SUM and COUNT values per *salary* attribute bucket, based on the data in Table 1. The number of rows in this table is the same as the number of buckets for the bucketized attribute. During execution of a range query, the server simply looks up the appropriate values from the aggregate table and returns them to the client. This frees the server from expensive computation with homomorphic encryption functions and also obviates any security risks.

Table 4. Aggregate values stored per bucket

employee.salary.aggregates		
Bucket ID	**SUM**	**COUNT**
41	Enc(930)	Enc(15)
64	Enc(1020)	Enc(13)
9	Enc(774)	Enc(9)
22	Enc(568)	Enc(6)

We recognize two drawbacks in the proposed technique: (1) extra storage for encrypted aggregates, and (2) additional computation following database update operations. The first is not an actual concern since extra space is truly negligible in comparison to that stemming from ciphertext expansion in either PH-based or public key homomorphic encryption functions. The second does present a slight complication which we address below. The main benefit is that the server is relieved from adding ciphertexts during query execution, removing this computational overhead.

4.1 Aggregation Query Processing

We now describe the processing of aggregation-style range queries using the proposed technique. As before, each query is partitioned into client- and server-side sub-queries Q^c and Q^s, respectively. Q^c is basically the original query and Q^s is its bucket-level "translation" and split into Q^s_c and Q^s_m (*certain* and *maybe* queries). However, unlike bucket-level range

queries, aggregation queries result in the server returning one or more bucker-level encrypted aggregate values as the query response to Q_c^s. Q_m^s executes as in [1] (described in Section 2.2) and returns the *etuples* belonging to bordering buckets which may be part of the final query response. For example, consider the following query:

SELECT SUM, COUNT from employee WHERE (employee.salary $\geq$ *30K) and (employee.salary* $\leq$ *75K)*

The corresponding server-side query Q^s would be: *SELECT SUM, COUNT from employee.salary.agg WHERE (id=64) or (id=9)*

The corresponding query reply would consist of:

1. Enc(1020) and Enc(13) for bucket id 64

   as well as:

2. *etuples* for all tuples with bucket id 9

As a final step, the client needs (1) decrypt, filter and aggregate the *etuples*, (2) to decrypt and sum up the respective bucket aggregates, and (3) combine results from the two steps to compute correct aggregates.

4.2 Handling Updates

Whenever a data owner updates its outsourced database to modify, delete or insert tuples involving bucketized attributes, the aggregate values need to be updated as well. An update query may therefore require two communication rounds with the server: the stored aggregate values need to be returned by the server in the first round, and then updated and returned by the data owner in the second round. In between the two rounds, the owner modifies the aggregate values accordingly (i.e., computes new SUM and/or new COUNT). This procedure is shown in figure 3, where a client inserts a new tuple and updates the *salary* aggregate table simultaneously.

We use the term *data owner* as opposed to *client* to capture the fact that there may be many clients who are authorized to query the outsourced data (and who have appropriate decryption keys). Whereas, the there might be only one *owner*, i.e., the entity authorized to modify the database. Thus, while an owner is always a client, the opposite is not always true.

We also note that the two-round interaction shown in figure 3 is not necessary if there is only one owner (but many clients). Recall that, for each database, its owner as well all other clients are required to store certain metadata (bucketization scheme) for each bucketized attribute. The size of the bucketization metadata is proportional to the number of buckets. Consequently, it is reasonable to require the (single) owner to store up-to-date bucket-level aggregate values for each bucketized attribute. (In other words, the additional storage is insignificant as it at most doubles the amount of metadata.) Consequently, the first round of communication (as part of update) is unnecessary.

5 Mixed Databases

Up until this point we have discussed a DAS model in which all the client's data is encrypted. We now look at execution of aggregation queries in a novel DAS flavor, where some attributes are encrypted and some are left in the clear. We label this as a *mixed database*. Such databases provide de facto access control since individuals not in possession of decryption keys cannot access sensitive data. Differentiating between confidential and non-confidential attributes also reduces the computational load related to encryption at both the server and client.

An interesting aggregation query in a mixed database specifies a range over a plaintext value while aggregating an encrypted attribute. Table 5 illustrates a mixed database where the *emp id* and age attributes are kept in the clear while *salary* is encrypted. A potential query asks for the total salary of all employees within a certain age group. Such queries cannot be executed with the proposed solution in Section 4, because the attribute over which the range is defined is not bucketized (since it is not encrypted). Instead, this plaintext attribute either has an index built over it or not. In the former case the index is utilized to select the matching tuples, while in the latter, a complete table scan is necessary during query execution. It still remains necessary for the server to aggregate over encrypted data, and we therefore return our focus to homomorphic encryptions functions. Next we compare and analyze the

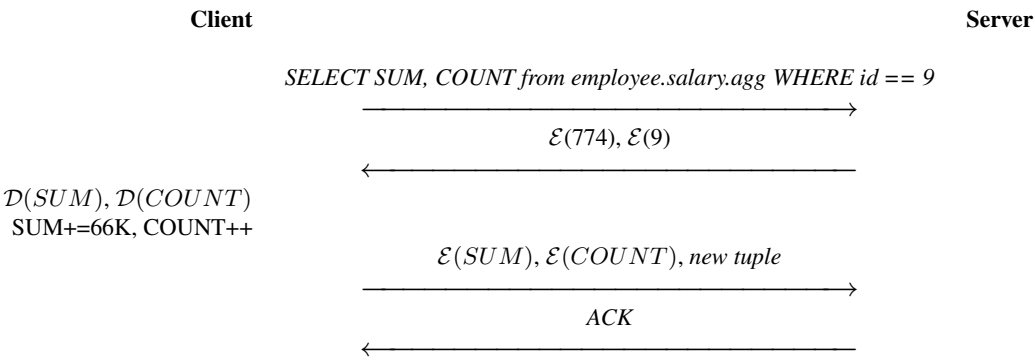

Figure 3. Owner/Client inserts new tuple

homomorphic functions introduced in Section 3.3 to determine the most appropriate candidate function for the mixed DAS model.

Table 5. Mixed Database

faculty.salary		
emp id	*age*	*salary*^{h}
31	52	87
32	45	12
33	38	41

5.1 Additive Homomorphic Encryption Scheme Candidates

We are interested in comparing provably secure additive homomorphic encryption schemes. Criteria used to evaluate schemes included the size of their ciphertexts, the cost of adding ciphertexts, and that of decryption. Cost of encryption is of less importance since it is a one-time offline computation performed by the data owner, and has no effect on query response time.

The four homomorphic encryption schemes that we consider are Paillier [10], Benaloh [14], Okamoto-Uchiyama (OU) [16] and the elliptic-curve variant of ElGamal (EC-EG) [15]. Appendix A describes each of these schemes in greater detail. The privacy homomorphism in [3] does not qualify as a viable candidate because of its weak security, which is pointed out in Section 3.2. Castelluccia et al.'s secret key homomorphic scheme [17] requires that additional data be returned to the client for decryption. This data consists of unique identifiers for each aggregated ciphertext and is proportionate in length to the number of aggregated values. Such bandwidth overhead diminishes the value of data aggregation, and we therefore omit this scheme from our pool of candidates[3].

5.2 Analysis and Comparison of Cryptoschemes

When comparing cryptosystems built upon different mathematical structures (EC-EG operates over elliptic curves while the OU and Benaloh work over multiplicative fields), it is important to devise a common computational unit of measurement for purposes of fair comparison. We choose that unit to be *1024-bit modular multiplications* and follow the same methodology for comparison as in [20]. The fundamental operation in EC-EG is elliptic curve point addition. Appendix B describes how to derive the equivalent number of modular multiplications to that of an elliptic curve point addition. The number of 1024-bit modular multiplications will define the computational cost of summing ciphertexts at the server and decryption of aggregate values at the client.

Table 6 shows the comparison of the three homomorphic cryptosystems. The size of ciphertexts reflects both the overhead of storage at the server and transmission of aggregate values. It is measured in bits. The cost of homomorphic addition (summing two ciphertexts) and decryption is measured by the num-

[3]It is possible to remove the additional bandwidth overhead by storing additional encrypted data at the server, but a description of this technique is outside the scope of this paper

Table 6. Performance Comparison of Additive Homomorphic Cryptosystems

Scheme	Addition	Decryption	Bandwidth
Paillier	4	1536	2048
EC-EG	1	16384	328
OU	1	512	1024
Benaloh	1	131072	1024

ber of 1024-bit modular multiplications required by the operations.

The parameters for each of the four cryptosystems have been selected such as to obtain an equal 1024-bit level of security. For *Paillier*, *Benaloh* and *OU*, primes p and q are selected such that $|n|$ = 1024, while $EC - EG$ uses one of the standard (IEEE) ECC curves over F_{163} defined in [21]. Random nonces are assumed to be 80-bits[4].

The decryption cost for Benaloh and EC-EG depend on the size of the aggregated values to be decrypted. These values in turn are a result of the size of the attribute aggregated and the number of values aggregated. Both cryptosystems employ a baby-giant step algorithm during decryption. These algorithms work by searching for the plaintext in its possible value range, while using tables of pre-computed values (at regular intervals) to speed up the search. The size of these tables directly affect the efficiency of the search in that the larger the tables the faster the search. When deriving the results in Table 6, we assumed aggregation of 10,000 20-bit bit values (e.g. up to million dollar salaries). Let *max* denote the number of bits required to represent the largest possible aggregate value. In our case, *max = 34*. As is common with baby-giant step algorithms, $\sqrt{max}$ pre-computed values are stored in a table, and $\frac{\sqrt{max}}{2}$ computations are required for the search (on average). This means that 2^17 computations will be required during Benaloh and EC-EG decryption, along with pre-computed tables of 2.6MB and 16.7MB, respectively.

[4]Random nonces are used in cryptosystems to make them non-deterministic, in that encryption of identical plaintexts will yield different ciphertexts

5.3 Recommendations

OU and Paillier clearly stand out amongst the four candidate schemes, mainly due to their lower decryption costs. This is of importance since decryption will be performed by clients, which may be computationally limited devices (e.g. cell phone). Between the two, OU is the preferred choice in each of the measured performance categories. This is a result of Paillier's cryptosystem requirement of a larger group structure (2048 versus 1024 bits), resulting in greater storage and bandwidth overhead, as well as more expensive computations. The large cost difference in summation of ciphertexts (4 to 1 ratio) also plays a significant role, since this operation will be executed very frequently by the server. We therefore declare OU to be the algorithm of choice for aggregation queries in mixed-databases.

EC-EG and Benaloh are poor candidate choices because of their extremely high decryption costs and the large storage requirements (at clients) associated with their baby-giant step algorithms. This poor performance reflects the database environment in which they are evaluated, where tables may contain several thousand tuples, creating a large value space to search through (during decryption). The two algorithms are seemingly good choices in alternative settings that only require a few number of small values to be aggregated (e.g. certain sensor networks) [22].

6 Related Work

The Database-As-a-Service (DAS) model was introduced by Haĉigümus, et al. in [1] and, since then, has received a lot of attention from the research community. The specific technique of bucketizing data to support range queries over encrypted tuples was described in [2]. Bucketization involves dividing the range of values in the specific domains of the attribute into *buckets* and providing explicit labels for each partition. Recent work [7, 8] analyze and estimate the loss of privacy due to bucketization. Since statistical information is revealed, some degree of privacy is invariably lost, but these results show that only very limited information can be deduced from the encrypted tuples and their corresponding bucket identifiers [8].

[11] is the first work describing homomorphic en-

cryption functions (referred to as a Privacy Homomorphisms (PHs) by the respective authors). Such functions were originally proposed as a method for performing arithmetic computations over private databanks. [3] suggests a specific homomorphic encryption function to use within a DAS model that utilizes bucketization. The additional functionality provided by this function expands upon the range of queries that can be executed by the DAS server, specifically supporting a set of aggregation operations (SUM, COUNT and AVG).

An alternative DAS flavor involves the use of a Secure Coprocessor (SC) to aid with processing of server-side queries. A SC is a computer that can be trusted with executing its computations correctly and unmolested, even when attackers gain physical access to the device. It also provides tamper resistance, allowing for secure storage of sensitive data such as cryptographic keys. [23] describes a high-level framework for incorporating a SC in a DAS setting, including the query splitting between the client, server and SC, and suggest [24] as a SC candidate.

7 Conclusion

In conclusion, we proposed an alternative technique to homomorphic encryption functions to support aggregation queries over encrypted tuples in the Database-as-a-Server Model. The previously suggested solution in [3] was shown to be insecure. Our technique if simple and reduces the computational overhead associated with aggregation queries on both the server and client. Next we explored mixed databases, where certain attributes are encrypted while others are left in the clear. Additively homomorphic encryption functions are needed to support basic aggregation queries for such databases. We analyzed and compared a set of homomorphic encryption candidates and selected our preferred algorithm.

References

[1] H. Hacigumus, B. Iyer, and S. Mehrotra, "Providing database as a service," in *International Conference on Data Engineering*, March 2002.

[2] H. Hacigumus, B. Iyer, C. Li, and S. Mehrotra, "Executing sql over encrypted data in the database-service-provider model," in *ACM SIGMOD Conference on Management of Data*, pp. 216–227, ACM Press, June 2002.

[3] H. Hacigumus, B. Iyer, and S. Mehrotra, "Efficient execution of aggregation queries over encrypted databases," in *International Conference on Database Systems for Advanced Applications (DASFAA)*, 2004.

[4] E. Mykletun, M. Narasimha, and G. Tsudik, "Authentication and integrity in outsourced databases," in *Symposium on Network and Distributed Systems Security (NDSS'04)*, Feb. 2004.

[5] E. Mykletun, M. Narasimha, and G. Tsudik, "Signature 'bouquets': Immutability of aggregated signatures," in *European Symposium on Research in Computer Security (ESORICS'04)*, Sept. 2004.

[6] P. Devanbu, M. Gertz, C. Martel, and S. G. Stubblebine, "Authentic third-party data publication," in *14th IFIP 11.3 Working Conference in Database Security*, pp. 101–112, 2000.

[7] B. Hore, S. Mehrotra, and G. Tsudik, "A privacy-preserving index for range queries," in *International Conference on Very Large Databases (VLDB)*, 2004.

[8] A. Ceselli, E. Damiani, S. Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati, "Modeling and assessing inference exposure in encrypted databases," in *ACM Transactions on Information and System Security*, vol. 8, pp. 119–152, 2005.

[9] M. Bellare and P. Rogaway, "Optimal asymmetric encryption," in *Advances in Cryptology - Eurocrypt*, pp. 92–111, 2004.

[10] "Public-key cryptosystems based on composite degree residuosity classes," in *99* (P. Paillier, ed.), vol. 1592 of *LNCS*, pp. 206–214, International Association for Cryptologic Research, IEE, 1999.

[11] R. Rivest, L. Adleman, and M. Dertouzous, "On data banks and privacy homomorphisms," in *Foundations of Secure Computation*, Academic Press, pp. 169–179, 1978.

[12] D. Boneh, C. Gentry, B. Lynn, and H. Shacham, "Aggregate and Verifiably Encrypted Signatures from Bilinear Maps,"

[13] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*. CRC Press series on discrete mathematics and its applications, CRC Press, 1997. ISBN 0-8493-8523-7.

[14] J. Benaloh, "Dense Probabilistic Encryption," *Proceedings of the Workshop on Selected Areas of Cryptography*, pp. 120–128, 1994.

[15] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE Transactions on Information Theory*, vol. IT-31, pp. 469–472, July 1985.

[16] T. Okamoto and S. Uchiyama, "A New Public-key Cryptosystem as Secure as Factoring," *EUROCRYPT*, pp. 308–318, 1998.

[17] C. Castelluccia and E. Mykletun and G. Tsudik, "Efficient Aggregation of encrypted data in Wireless Sensor Networks," *Mobile and Ubiquitous Systems: Networking and Services*, 2005.

[18] N. I. of Standards and Technology, "Triple-des algorith," *FIPS 46-3*, 1998.

[19] N. I. of Standards and Technology, "Advanced encryption standard," *NIST FIPS PUB 197*, 2001.

[20] N. Gura, A. Patel, A. Wander, H. Eberle, and S. Shantz, "Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs," *Cryptographic Hardware and Embedded Systems (CHES)*, pp. 119–132, 2004.

[21] IEEE, "Standard P1363: Standard Specifications For Public-Key Cryptography," *http://grouper.ieee.org/groups/1363/*.

[22] E. Mykletun and J. Girao and D. Westhoff, "Public Key Based Cryptoschemes for Data Concealment in Wireless Sensor Networks," *International Conference on Communications*, 2006.

[23] E. Mykletun and G. Tsudik, "Incorporating a Secure Coprocessor in the Database-as-a-Service Model," *International Workshop on Innovative Architecture for Future Generation High Performance Processors and Systems*, 2005.

[24] J. G. Dyer, M. Lindemann, R. S. R. Perez, L. van Doorn, and S. W. Smith, "Building the IBM 4758 Secure Coprocessor," in *EEE Computer*, pp. 57–66, 2001.

[25] T. ElGamal, "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms," *CRYPTO*, vol. IT-31, no. 4, pp. 469–472, 1985.

[26] J.M. Adler and W. Dai and R. L. Green and C.A. Neff, "Computational Details of the VoteHere Homomorphic Election System," *ASIACRYPT*, 2000.

A Cryptographic Schemes

This appendix describes three additive homomorphic encryption schemes.

A.1 Paillier

A new provably secure cryptosystem that supports the additive homomorphic operation was introduced by Pascal Paillier [10]. The cryptosystem is based on the composite residuosity problem. Encryption and decryption funcitons require a very large modulus which affects the size of ciphertexts and the cost of computations. Addition is achieved through the multiplication of ciphertexts. Paillier's cryptographic algorithm is outlined below.

Paillier	
Public Key	$n = pq, g$
Private Key	(p, q)
Encryption	plaintext $m \in \mathbb{Z}_n$, $r \in_R \mathbb{Z}_n$, ciphertext $c = g^m r^n \pmod{n^2}$
Decryption	compute $m = \frac{L(c^\lambda \bmod n^2)}{L(g^\lambda \bmod n^2)} \bmod n$

Let p and q represent 512-bit prime numbers. Their product, $n = pq$, is the resulting 1024-bit modulus, $\lambda(n) = lcm(p-1, q-1)$, $L(u) = \frac{u-1}{n}$, and g is an element such that $gcd(L(g^\lambda \bmod n^2), n) = 1$.

A.2 Okamoto-Uchiyama (OU)

In Eurocrypt 98', Okamoto and Uchiyama proposed a new public-key cryptosystem as secure as factoring and is based on the ability of computing discrete logarithms in a particular sub-group [16]. Specifically, for an odd prime p, the p-Sylow subgroup is defined as $\gamma_p = \{x < p^2 \mid x = 1 \pmod{p}\}$, and $|\gamma_p| = p$. A function L that maps elements from γ_p to $\mathbb{Z}_p$ is defined as $L(x) = (x-1)/p$. Function L has homomorphic properties from multiplication to addition. For elements $a, b \in \gamma_p$, $L(a * b) = L(a) + L(b) \pmod{p}$, and for $c \in \mathbb{Z}_p$, $L(a^c) = c * L(a)$.

Their scheme is characterized by probabilistic encryption, additive homomorphic properties, and relating the computational complexity of the encryption function to the size of the plaintext. We now describe their cryptosystem:

Let p and q be random k-bit primes and set $n = p^2 q$. For an n of approximately 1024 bits, a choice of k could be 341. Next, randomly choose a $g \in_R \mathbb{Z}_n$ such

that element $g_p = g^{p-1} \pmod{p^2}$ has order p. Finally, set $h = g^n \pmod{n}$. The additive homomorphic property is achieved through the multiplication of ciphertexts: $Enc(m_1 + m_2) = Enc(m_1) \times Enc(m_2)$.

Okamoto-Uchiyama (OU)

Public Key	$n = p^2q, g, h$
Private Key	(p, q)
Encryption	plaintext $m \in 2^k$, $r \in_R \mathbb{Z}_n$, ciphertext $c = g^m h^r \pmod{n}$
Decryption	$c' = c^{p-1} \pmod{p^2}$ compute $m = L(c')L(g_p)^{-1} \pmod{p}$
Note that $c^{p-1} \pmod{p^2} = g^{m(p-1)}g^{nr(p-1)} = g_p^m \pmod{p^2}$	

A.3 Benaloh

In [14] Benaloh introduced a probabilistic cryptoscheme whose encryption cost is dependent on the size of the plaintext. The key-setup is as follows: let $n = pq$ for large primes p, q and choose value r such that $r|(p-1)$, $gcd(\frac{p-1}{r}, r) = 1$ and $gcd(q-1, r) = 1$. The public key y is chosen such that $y \in \mathbb{Z}_n^*$ and $y^{(p-1)(q-1)/r} \pmod{n} \neq 1$. The scheme's security is based upon the cryptographically assumption that it is computationally difficult to decide higher residuosity: given z, r and n of unknown factorization, find x such that $z = x^r \pmod{n}$. The additive homomorphic property is achieved through the multiplication of ciphertexts.

Benaloh

Public Key	$n = pq, y, r$
Private Key	(p, q)
Encryption	plaintext $m \in \mathbb{Z}_r$, $u \in_R \mathbb{Z}_n^*$, ciphertext $c = y^m u^r \pmod{n}$
Decryption	compute m such that $(y^{-m'} c \pmod{n}) \in Enc_r(0)$ for $m' = 0, 1, 2,...$ until $r - 1$ or $m = m'$

To understand why decryption works, it is useful to notice that the decryptor needs the ability to decide higher residuosity, which can be done efficiently when the factorization of n is known. Note that $z \in Enc(0)$ iff $z^{(p-1)(q-1)/r} \pmod{n} = 1$. Therefore, one can decrypt a ciphertext c by finding, via brute force, the smallest integer $m' < r$ such that $y^{-m'} c \pmod{n} \in Enc(0)$.

One method to speed up decryption is to store precomputed values $T_i = y^{i(p-1)(q-1)/r} \pmod{n}$, for $i = 0, 1,..., r - 1$ in a lookup table. Then, for $c = Enc(m)$, it is the case that $c^{(p-1)(q-1)/r} \pmod{n} = T_m$ and one can therefore avoid the brute force search by using the lookup table. For large values of r it may be too expensive to store all r T_m values, and one can then resort to a *big-step little-step* method by only pre-computing T_i for $i \approx k\sqrt{r}$ as k ranges from 1 to r. Such an optimization reduces the storage, precomputation time and decryption time to $O(\sqrt{r})$ at the decryptor.

A.4 Elliptic Curve variant of ElGamal (EC-EG)

We now describe the elliptic curve ElGamal encryption scheme (EC-EG). This is equivalent to the original ElGamal scheme [25] but transformed to an additive group. Key set-up consists in choosing an elliptic curve E together with a 163-bit prime p and generator G. Its security is based upon the Elliptic Curve Discrete Log Problem (ECDLP).

ElGamal Encryption Scheme (EC-EG)

Public Key	$E, p, G, Y = xG$, where $G, Y \in F_p$
Private Key	$x \in F_p$
Encryption	plaintext $M = map(m)$, $r \in_R F_p$, ciphertext $C = (R, S)$, where $R = kG, S = M + kY$
Decryption	$M = -xR + S = -xkG + M + xkG$, $m = rmap(M)$

EC-EG is additively homomorphic and ciphertexts are combined through addition. The summation of two EC-EG ciphertexts requires two point additions, namely one for each of the ciphertext components R and S.

map() refers to the mapping function used to map values (e.g. plaintexts) into points on the curve, and vice versa. Such a function is necessary because the operands of elliptic curve operations are elliptic curve points. This mapping needs to be deterministic such that the same plaintext always maps to the same point. Note that the operation used to map a value is independent from the transformation used to

encrypt it: encryption simply transforms a point into another point on the elliptic curve. There exist standard mapping functions but we require one that has the additional property of being homomorphic, i.e. $map(m_1 + m_2) = map(m_1) + map(m_2)$, as suggested in [26].

B Common Computational Unit of Measurement

Section 5.2 describes using a common computational unit of measurement when comparing cryptosystems based upon different underlying fields (elliptic curve and finite fields). In this appendix, we describe how to equate an elliptic curve operation with finite field multiplications.

The computation of our focus is xG over F_p, where x is a $|p|$-bit scalar and G is a point on the curve. Similarly to the square-and-multiply method in finite fields (used during modular exponentiations), we apply the double-and-add algorithm, requiring $|p|$ doublings and $1/2|p|$ additions. Each point doubling/addition operation involves the computation of an inverse which is approximately equivalent to 3 multiplications. With the additional 2 multiplications that take place, we count 5 total multiplications per point addition and doubling, and therefore a total of $5 \times \frac{3}{2} \times |p| = \frac{15}{2} \times |p|$ modular multiplications to compute xG. Next, we need a method for comparing the cost of modular multiplications over different sized moduli. Note that a y-bit modular multiplication has a complexity of $O(y^2)$. One can then conclude that a y-bit modular multiplication is approximately equivalent to $\frac{y^2}{1024^2}$ 1024-bit modular multiplications.

As an example, the computation xG over F_p, where p is a 163-bit prime, requires on average $245 = 163 + 82$ point doublings and additions, i.e. $1225(245 * 5)$ 160-bit modular multiplications. A 1024-bit modular multiplication is approximately 40 times more expensive than that of a 163-bit one, and so, computing xG requires approximately $1225/40 = 31$ 1024-bit modular multiplications.