Privacy-Preserving Query Execution using a Decentralized Architecture and Tamper Resistant Hardware

Quoc-Cuong To Benjamin Nguyen Philippe Pucheral
SMIS Project, INRIA Rocquencourt, 78153 Le Chesnay, France
PRiSM Laboratory, 45, Av. des Etats-Unis, 78035 Versailles, France
<Fname.Lname>@inria.fr , <Fname.Lname>@prism.uvsq.fr

ABSTRACT

Current applications, from complex sensor systems (e.g. quantified self) to online e-markets acquire vast quantities of personal information which usually ends-up on central servers. Decentralized architectures, devised to help individuals keep full control of their data, hinder global treatments and queries, impeding the development of services of great interest. This paper promotes the idea of pushing the security to the edges of applications, through the use of secure hardware devices controlling the data at the place of their acquisition. To solve this problem, we propose secure distributed querying protocols based on the use of a tangible physical element of trust, reestablishing the capacity to perform global computations without revealing any sensitive information to central servers. There are two main problems when trying to support SQL in this context: perform joins and perform aggregations. In this paper, we study the subset of SQL queries without joins and show how to secure their execution in the presence of honest-but-curious attackers. Cost models and experiments demonstrate that this approach can scale to nationwide infrastructures.

Keywords

Privacy-preserving, decentralized, SQL query, trusted hardware.

1. INTRODUCTION

With the convergence of mobile communications, sensors and online social networks technologies, we are witnessing an exponential increase in the creation and consumption of personal data. Some data is freely disclosed by users. Some other is transparently acquired by sensor systems through analog processes (e.g., GPS tracking units, smart meters, healthcare sensors) or mechanical interactions (e.g., as simple as opening a door or putting light on). In fine, all this data ends up in servers. It represents an unprecedented potential for applications and business (e.g., car insurance billing, carbon tax charging, traffic decongestion, resource optimization in smart grids, healthcare surveillance, participatory sensing). However, as seen with the PRISM affair, the public opinion is starting to wonder whether these new services are not bringing us closer to the science fiction dystopias, since individuals data is carefully scrutinized by governmental agencies and companies in charge of processing it [28]. It has become clear that centralizing and processing all one's data in a single server is a major problem with regards to privacy concerns. Indeed, privacy violations arise from negligence, abusive use and attacks and no current server-based approach, including cryptography based and server-side secure hardware [1], seems capable of closing the gap.

Many initiatives arise to provide better control to the user over the management of her personal data, as suggested by the World Economic Forum [18]. In this article, we focus on a novel architectural approach to this problem called *Trusted Cells* [14]. This approach capitalizes on emerging practices and hardware advances representing a sea change in the acquisition and protection of personal data. Trusted Cells push the security to the edges of the network, through personal data servers [3] running on secure smart phones, set-top boxes, plug computers[1] or secure portable tokens[2] forming a global decentralized data platform. Indeed, thanks to the emergence of low-cost secure hardware and firmware technologies like ARM TrustZone[3], a full Trusted Execution Environment (TEE) will soon be present in any client device. In this paper, and up to the experiments section, we consider that personal data is acquired and/or hosted by secure devices but make no additional assumption regarding the technical solution they rely on.

In Trusted Cells, security and privacy come from the conjunction of large scale distributed and trusted hardware. However, this must not impede on the possibility of executing queries. Typically, global queries would be helpful to compute aggregates over smart meters without disclosing individual's raw data (e.g., *compute the mean energy consumption per time period and district*). Identifying queries also make sense assuming the identified subjects consent to participate (e.g., *send an alert to people older than 80 and living in* Memphis *if the number of people suffering from flu in* Tennessee *has reached a given threshold*). Hence, individual's privacy on one side and global benefits for the community and business perspectives on the other side are contradictory objectives which need to be reconciled. Computing SQL-like queries on such distributed infrastructure leads to two major and different problems: computing joins between data hosted at different locations and computing aggregates over this same data. This paper addresses the second issue: how to compute global queries over decentralized personal data stores while respecting users' privacy? Indeed, we believe that the computation of aggregates is central to the many novel privacy preserving applications such as smart metering, e-administration, etc.

Our objective is to make as few restrictions on the computation model as possible. We model the information system as a global database formed by the union of a myriad of distributed local data

EDBT'14, March 24–28, 2014, Athens, Greece.

[1] freedomboxfoundation.org
[2] www.gd-sfs.com/portable-security-token
[3] www.arm.com/products/processors/technologies/trustzone.php

stores (e.g., nation-wide context) and we consider regular SQL queries (without external joins involving data from different data stores) and a traditional access control model. Hence the context we are targeting is different and more general than, (1) querying encrypted outsourced data where restrictions are put on the predicates which can be evaluated [2, 6, 17, 22], (2) performing privacy-preserving queries usually restricted to statistical queries matching differential privacy constraints [4, 15] and (3) performing Secure-Multi-Party (SMC) query computations which cannot meet both query generality and scalability objectives [25].

The contributions of this paper are: (1) to propose different secure query execution techniques to evaluate regular SQL "group by" queries over a set of distributed trusted personal data stores, (2) to study the range of applicability of these techniques and (3) to show that our approach is compatible with nation-wide contexts.

The rest of this paper is organized as follows. Section 2 states our problem. Section 3 introduces a framework to execute simple queries and Section 4 concentrates on complex queries involving Group By and Having clauses. Section 5 and 6 analyses the privacy and performance of these solutions through cost models and experiments. Finally, Section 7 discusses related works and section 8 concludes.

2. CONTEXT OF THE STUDY

2.1 Asymmetric Architecture

The architecture we consider is decentralized by nature. It is formed by a large set of low power personal Trusted Data Servers (TDS) embedded in secure devices. Despite the diversity of existing hardware platforms, a secure device can be abstracted by (1) a Trusted Execution Environment and (2) a (potentially untrusted but cryptographically protected) mass storage area (see Fig. 1)[4]. The important assumption is that the TDS code is executed by the secure device hosting it and then cannot be tampered, even by the TDS holder herself.

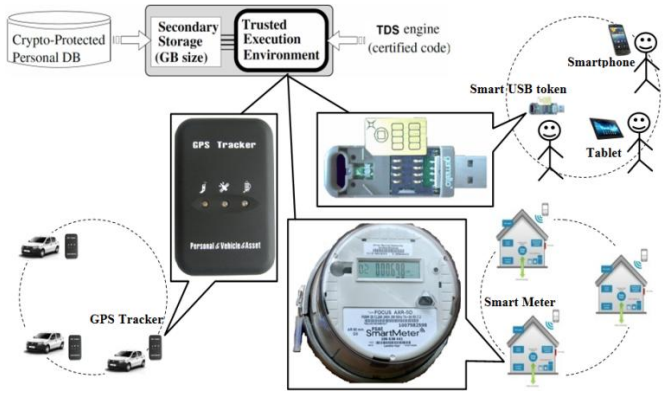


Fig. 1. Trusted Data Servers

We make no assumption about how the data is actually gathered by TDSs, this point being application dependent [3, 28]. We simply consider that *local databases* conform to a common schema which can be queried in SQL. For example, power meter data (resp., GPS traces, healthcare records, etc) can be stored in one (or several) table(s) whose schema is defined by the national distribution company (resp., an insurance company consortium, the Ministry of Health, etc). Since raw data can be highly sensitive, it must be protected by an access control policy defined either by the producer organism, by the legislator or by a consumer association depending on the context. Hence, each TDS is responsible for participating in a distributed query protocol while enforcing the access control rules protecting the local data it hosts.

Since TDSs have limited storage and computing resources and they are not necessarily always connected, an external infrastructure, called hereafter *Supporting Server Infrastructure* (SSI), is required to manage the communications between TDSs, run the distributed query protocol and store the intermediate results produced by this protocol. Because SSI is implemented on regular server(s), e.g., in the Cloud, it exhibits the same low level of trustworthiness.

The computing architecture is said *asymmetric* in the sense that it is composed of a very large number of low power, weakly connected but highly secure TDSs and of a powerful, highly available but untrusted SSI.

2.2 Threat Model

TDSs are the unique elements of trust in the architecture and are considered *honest*. As mentioned earlier, no trust assumption needs to be made on the TDS holder herself because a TDS is tamper-resistant and enforces the access control rules associated to its holder (just like a car driver cannot tamper the GPS tracker installed in her car by its insurance company or a customer cannot gain access to any secret data stored in her banking smartcard).

We consider *honest-but-curious* (also called *semi-honest*) SSI (i.e., which tries to infer any information it can but strictly follows the protocol). Considering *malicious* SSI (i.e., which may tamper the protocol with no limit, including denial-of-service) is of little interest to this study. Indeed, a malicious SSI is likely to be detected with an irreversible political/financial damage and even the risk of a class action.

The objective is then to implement a querying protocol so that (1) the querier can gain access only to the final result of authorized queries (not to the raw data participating in the computation), as in a traditional database systems and (2) intermediate results stored in SSI are obfuscated. Preventing inferential attacks by combining the result of a sequence of authorized queries as in statistical databases and PPDP work (See section 7) is orthogonal to this study.

2.3 Type of queries and scenarios

We assume that the querier can issue the following form of SQL queries[5], borrowing the SIZE clause from the definition of windows in the StreamSQL standard [9]. This clause is used to indicate a maximum number of tuples to be collected, and/or a collection duration.

SELECT <attribute(s) and/or aggregate function(s)>
FROM <Table(s)>
[WHERE <condition(s)>]
[GROUP BY <grouping attribute(s)>]
[HAVING <grouping condition(s)>]
[SIZE <size condition(s)>]

[4] For illustration purpose, the secure device considered in our experiments is made of a tamper-resistant microcontroller connected to a Flash memory chip.

[5] As stated in the introduction, we do not consider *joins* between data stored in different TDSs in this article. However, internal joins which can be executed locally by each TDS are supported.

Let us consider the following scenario. The energy distribution company would like to issue the following query on its customers' smart meters: "*SELECT AVG(Cons) FROM Power P, Consumer C WHERE C.accomodation='detached house' and C.cid = P.cid GROUP BY C.district HAVING Count(distinct C.cid) > 100 SIZE 50000"*. This query computes the mean of energy consumption of consumers living in a detached home grouped by district, for whose districts where more than 100 consumers answered the poll and the poll stops after having globally received at least 50.000 answers. The semantics of the query are the same as those of a stream relational query [13], i.e. the data is pushed from the TDSs (i.e., smart meters) to SSI (e.g., a Cloud provider or the distribution company itself) in the form of *windows*. Only the smart meter of customers who opt-in for this service will participate in the computation. Needless to say that the querier, that is the distribution company, must be prevented to see the raw data of its customers for privacy concerns[6].

In other scenarios where TDSs are seldom connected (e.g., querying mobile PCEHR - Personally Controlled Electronic Health Records - embedded in TDSs), the time to collect the data is probably going to be quite large. Therefore the challenge is not on the overall response time, but rather to show that the query computation on the collected data is tractable in reasonable time, given local resources.

3. BASIC QUERYING PROTOCOL

3.1 Core infrastructure

Our querying protocols share common basic mechanisms to make TDSs aware of the queries to be computed and to organize the dataflow between TDSs and queriers such that SSI cannot infer anything from the queries and their results.

Query and result delivery: queries are executed in pull mode. A querier posts its query to the SSI and TDSs download it at connection time. To this end, SSI can maintain personal *queryboxes* (in reference to mailboxes) where each TDS receives queries directed to it (e.g., *get the monthly energy consumption of consumer C*) and a global *querybox* for queries directed to the crowd (e.g., *get the mean of energy consumption per month for people living in district D*). Result tuples are gathered by the SSI in a temporary storage area. A query remains active until the SIZE clause is evaluated to *true* by the SSI, which then informs the querier that the result is ready.

Dataflow obfuscation: all data (queries and tuples) exchanged between the querier and the TDSs, and between TDSs themselves, can be spied by SSI and must therefore be encrypted. However, an *honest-but-curious* SSI can try to conduct *frequency-based attack* [26], i.e., exploiting prior knowledge about the data distribution to infer the plaintext values of ciphertexts. Depending on the protocols (see later), two kinds of encryption schemes will be used to prevent frequency-based attacks. With non-deterministic (aka probabilistic) encryption, denoted by *nDet_Enc*, several encryptions of the same message yield different ciphertexts while deterministic encryption (*Det_Enc* for short) always produces the same ciphertext for a given plaintext and key [8]. Whatever the encryption scheme, symmetric keys must be shared among TDSs: we note *k1* the symmetric key used by the querier and the TDSs to communicate together and *k2* the key shared by TDSs to exchange temporary results among them Note that these keys may change over time and the way they are delivered to TDSs is context dependent[7].

Access control enforcement: TDSs are assumed to answer only authorized queries, meaning that they are aware of the access control (AC) policy and of the querier credentials. As explained in section 2, this policy can be defined by the application provider, the legislator or even by a consumer association depending on the context. As for the cryptographic material, the policy can be either installed at burn time or be downloaded dynamically depending on the context.

3.2 Select-From-Where statements

Let us first consider simple SQL queries of the form:

SELECT <attribute(s)> FROM <Table(s)> [WHERE <condition(s)>] [SIZE <size condition(s)>]

These queries do not have a GROUP BY or HAVING clause nor involve aggregate functions in the SELECT clause. Hence, the selected attributes may (or may not) contain identifying information about the individuals. Though basic, these queries answer a number of practical use-cases, e.g., a doctor querying the embedded healthcare folders of her patients, an energy provider willing to offer special prices to people matching a specific consumption profile. To compute such queries, the protocol is divided in two phases (see Fig. 2):

Collection phase: (step 1) the querier posts on the SSI a query Q encrypted with k1, its credential C signed by an authority and S the SIZE clause of the query in cleartext so that SSI can evaluate it; (step 2) targeted TDSs download Q when they connect; (step 3) each of these TDSs decrypts Q, checks C, evaluates the AC policy associated to the querier and computes the result of the WHERE clause on the local data; then each TDS either sends its result tuples (step 4), or a dummy tuple[8] whether the result is empty or the querier has not enough privilege to access these local data (step 4'), non-deterministically encrypted with k2. The collection phase stops when the SIZE condition has been reached. The result of the collection phase is actually the result of the query complemented with dummy tuples. We call it Covering Result (CR). For this form of queries, the aggregation phase presented in Fig. 2 is empty.

Filtering phase: (step 9) SSI partitions the Covering Result with the objective to let several TDSs manage next these partitions in parallel. The Covering Result being fully encrypted, SSI sees partitions as uninterpreted chunks of bytes; (step 10) connected TDSs download these partitions. These TDSs may be different from the ones involved in the collection phase; (step 11) each of

[6] At the 1HZ granularity provided by the French Linky power meters, most electrical appliances have a distinctive energy signature. It is thus possible to infer from the power meter data inhabitants activities [23].

[7] In a homogeneous context (i.e., where all TDSs are delivered by a single provider), these keys or a seed allowing to generate a sequence of keys can be installed at burn time. In an open context, a PKI infrastructure could be used so that queriers and TDSs all have a public-private key pair which can be used to exchange symmetric keys. Alternatively, a broadcast encryption scheme can also be used to securely exchange keys between TDSs and querier.

[8] even if the query is encrypted, sending dummy tuples avoids the SSI to learn the query selectivity (and from that guess the query). It is also helpful in the case where SSI and querier are the same entity.

these TDS decrypts the partition and filters out dummy tuples; (step 12) each TDS sends back the true tuples encrypted with key k1 to SSI, which finally concatenates all results and informs the querier that she can download the result (step 13).

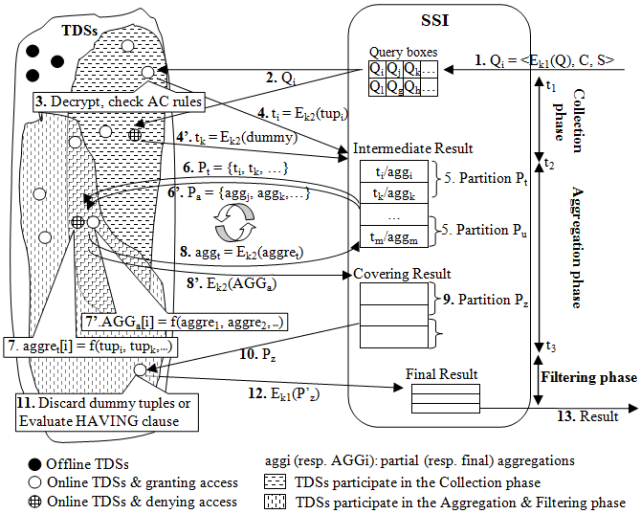


Fig. 2. Querying protocol

Informally speaking, the correctness, security and efficiency properties of the protocol are as follows:

Correctness. Since SSI is honest-but-curious, it will deliver to the querier all tuples returned by the TDSs. Dummy tuples are marked so that they can be recognized and removed after decryption by each TDS. Therefore the final result contains only true tuples. If a TDS goes offline in the middle of processing a partition, SSI resends that partition to another available TDS after a given timeout so that the result is complete.

Security. Since SSI does not know key k1, it cannot decrypt the query nor the result tuples. TDSs use *nDet_Enc* for encrypting the result tuples so that SSI cannot launch any frequency-based attack nor can detect dummy tuples. But what if SSI acquires a TDS with the objective to get the cryptographic material? As stated in Section 2, TDS code cannot be tampered, even by its holder. Whatever the information decrypted internally, the only output that a TDS can deliver is a set of encrypted tuples, which does not represent any benefit for SSI. And what if SSI colludes with the querier? For the same reason, SSI will only get the same information as the querier (i.e., the final result in clear text and no more).

Efficiency. The efficiency of the protocol is linked to the frequency of TDSs connection and to the SIZE clause. Both the collection and filtering phases are run in parallel by all connected TDSs and no time-consuming task is performed by any of them. As the experiment section will clarify, each TDS manages incoming partitions in streaming because the internal time to decrypt the data and perform the filtering is significantly less than the time needed to download the data.

While important in practice, executing Select-From-Where queries in the Trusted Cells context shows no intractable difficulties and the main objective of this section is to present the query framework in an easy to understand way. Executing Group By queries is far more challenging. The next section will present different alternatives to tackle this problem. Rather than trying to get an optimal solution, which is fatally context dependent, the objective is to explore the design space and show that different querying protocols may be devised to tackle a broad range of situations.

4. GROUP BY QUERIES

4.1 Generic query evaluation protocol

Let us now consider general SQL queries of the form[9]:

SELECT <attribute(s) and/or aggregate function(s)> FROM <Table(s)> [WHERE <condition(s)>] [GROUP BY <grouping attribute(s)>] [HAVING <grouping condition(s)>] [SIZE <size condition(s)>]

These queries are more challenging to compute because they require performing set-oriented computations over intermediate results sent by TDSs to SSI. The point is that TDSs usually have limited RAM, limited computing resources and limited connectivity. It is therefore unrealistic to devise a protocol where a single TDS downloads the intermediate results of all participants, decrypts them and computes the aggregation alone. On the other hand, SSI cannot help much in the processing since (1) it is not allowed to decrypt any intermediate results and (2) it cannot gather encrypted data into groups based on the encrypted value of the grouping attributes, denoted by $A_G=\{G_i\}$, without gaining some knowledge about the data distribution. This would indeed violate our security assumption since the knowledge of A_G distribution opens the door to frequency-based attacks by SSI. In the extreme case where A_G contains both quasi-identifiers and sensitive values, attribute linkage would become obvious. Finally, the querier cannot help in the processing either since she is only granted access to the final result, and not to the raw data.

To solve this problem, we suggest a generic aggregation protocol divided into three phases (see Fig. 2):

Collection phase: similar to the basic protocol.

Aggregation phase: (step 5) SSI partitions the result of the collection phase; (step 6) connected TDSs (may be different from the ones involved in the collection phase) download these partitions; (step 7) each of these TDS decrypts the partition, eliminates the dummy tuples and computes partial aggregations (i.e., aggregates data belonging to the same group inside each partition); (step 8) each TDS sends its partial aggregations encrypted with k2 back to SSI; depending on the protocol (see next sections), the aggregation phase is iterative, and continues until all tuples belonging to the same group have been aggregated (steps 6', 7', 8'); The last iteration produces a Covering Result containing a single (encrypted) aggregated tuple for each group.

Filtering phase: this phase is similar to the basic protocol except that the role of step 11 is to eliminate the groups which do not satisfy the HAVING clause instead of eliminating dummy tuples.

The rest of this section presents different variations of this generic protocol, depending on which encryption scheme is used in the collection and aggregation phases, how SSI constructs the partitions, and what information is revealed to SSI. Each solution has its own strengths and weaknesses and therefore is suitable for a specific situation. Three kinds of solutions are proposed: secure aggregation, noise-based, and histogram-based. They are

[9] For the sake of clarity, we concentrate on the management of distributive, algebraic and holistic aggregate functions identified in [27] as the most prominent and useful ones.

subsequently compared in terms of privacy protection (Section 5) and performance (Section 6).

4.2 Secure Aggregation protocol

This protocol, denoted by *S_Agg* and detailed in Fig. 4, instantiates the generic protocol as follows. In the **collection phase**, each participating TDS encrypts its result tuples using *nDet_Enc* to prevent any frequency-based attack by SSI. The consequence is that SSI cannot get any knowledge about the group each tuple belongs to. Thus, during step 5, tuples from the same group are randomly distributed among the partitions. This imposes the **aggregation phase** to be iterative, as illustrated in Fig. 3. At each iteration, TDSs download partitions containing a sequence of (A_G, Aggregate) value pairs ((City, Energy_consumption) in the example) and sends back to SSI a smaller sequence of (A_G, Aggregate) value pairs where values of the same group have been aggregated. SSI gathers these partial aggregations to form new partitions, and so on and so forth until a single partition is produced, which contains the final aggregation.

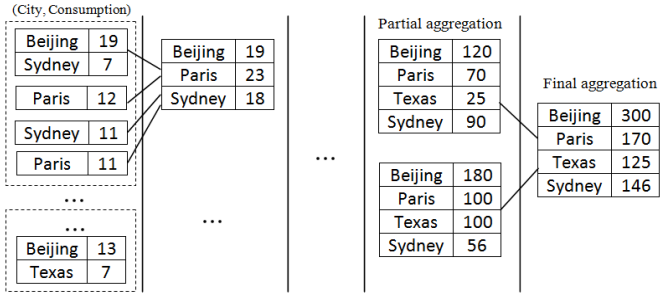


Fig. 3. (iterative partial) aggregation

```
Algorithm S_Agg (K1, K2, Q, α)
Input: (TDS's side): the cryptographic keys (K1,K2),
query Q from Querier
       (SSI's side): reduction factor α (α ≥ 2).
Output:  the final aggregation Ω_final.
1  begin Collection phase
2    Each connected TDS sends a tuple of the form
     tup_e = nE_K2(tup) to SSI
3  end
4  begin Aggregation phase
5    repeat
6      repeat
7        TDSs connect to SSI and SSI chooses tup_e (or
encrypted partial aggregation Ω_e) randomly to parallel
feed these TDSs (in data stream)
8        foreach TDS ∈ TDSs do
9        Receive tup_e (or Ω_e) from SSI
10       Decrypt tup_e (or Ω_e): tup ← nE_K2^-1(tup_e)
                                 Ω ← nE_K2^-1(Ω_e)
11       Add to its partial aggregation: Ω = Ω ⊕ tup
                                 Ω_new = Ω_old ⊕ Ω
12     until all tup_e (or Ω_e) in SSI have been sent to
           TDSs
13     foreach TDS ∈ TDSs do
14       Encrypt its partial aggregation: Ω_e ← nE_K2(Ω)
15       Send Ω_e to SSI
16   until n_Ωe = 1
17 end
18 Filtering phase //evaluate HAVING clause
19 return nE_K1(Ω_final) by SSI to Querier.
```

Fig. 4. Secure Aggregation algorithm

Correctness. The requirement for *S_Agg* to terminate is that TDSs have enough resources to perform partial aggregations. Each TDS needs to maintain in memory a data structure called *partial aggregate* which stores the current value of the aggregate function being computed for each group. Each tuple read from the input partition contributes to the current value of the aggregate function for the group this tuple belongs to. Hence the *partial aggregate* structure must fit in RAM (or be swapped in stable storage at much higher cost). If the number of groups is high (e.g., grouping on a key attribute) and TDSs have a tiny RAM, this may become a limiting factor.

Security. In all phases, the information revealed to SSI is a sequence of tuples or value pairs encrypted non-deterministically (*nDet_Enc*) so that SSI cannot conduct any frequency-based attack.

Efficiency. The aggregation process is such that the parallelism between TDSs decreases at each iteration, up to have a single TDS producing the final aggregation. Note again that incoming partitions are managed in streaming because the cost to download the data significantly dominates the rest.

4.3 Noise-based protocols

In these protocols, called Noise_based and detailed in Fig. 5, *Det_Enc* is used during the **collection phase** on the grouping attributes A_G. This allows SSI to assemble tuples belonging to the same groups in the same partitions. However, using *Det_Enc* reveals the distribution of A_G to SSI. To prevent this disclosure, the fundamental idea is that TDSs add some noise (i.e., fake tuples) to the data in order to hide the real distribution. The added fake tuples must have identified characteristics, as dummy tuples, such that TDSs can filter them out in a later step. The **aggregation** phase is roughly similar to *S_Agg*, except that the content of partitions is no longer random, thereby accelerating convergence and allowing parallelism up to the final iteration. Two solutions are introduced to generate noise: random (white) noise, and noise controlled by complementary domains.

```
Algorithm   R_nf_Noise (K1, K2, Q, nf)
Input: (TDS's side): the cryptographic keys (K1,K2),
query Q from Querier
Output:  the final aggregation Ω_final.
1  begin Collection phase
2     Each connected TDS sends (nf+1) tuples of the
      form tup_e = (E_K2(A_G),nE_K2(Ā_G) to SSI
3  end
4  begin Aggregation phase
5    repeat //(on SSI side)
6       SSI groups tup_e with the same E_K2(A_G)
7       TDSs connect to SSI to download these groups
            (in data stream)
8    until all groups in SSI have been sent to TDSs
9    foreach TDS ∈ TDSs do //(on TDS side)
10     repeat
11       Receive tup_e from SSI
12       Decrypt tup_e: A_G ← E_K2^-1(A_Ge); Ā_G ← nE_K2^-1(Ā_Ge)
13       Filter false tuples (based on the identified
          characteristics)
14       Compute the aggregate values for the group
          A_G: [A_G, AGG]
15     until no more tuples received from SSI
16     Encrypt this aggregate value: [E_K2(A_G),nE_K2(AGG)]
17     Send this encrypted aggregation to SSI
18 end
19 Filtering phase //evaluate HAVING clause
20 return nE_K1(Ω_final) to Querier by SSI
```

Fig. 5. Random noise algorithm

Random (white) noise solutions. In this solution, denoted R_{nf}*_Noise*, n_f fake tuples are generated randomly then added. TDSs apply *Det-Enc* on A_G, and *nDet_Enc* on $\bar{A}_G$ (the attributes not appearing in the GROUP BY clause). However, because the

fake tuples are randomly generated, the distribution of mixed values may not be different enough from that of true values especially if the disparity in frequency among A_G is big. To overcome this difficulty, a large quantity of fake tuples (n_f >>1) must be injected to make the fake distribution dominate the true one.

Noise controlled by complementary domains. This solution, called *C_Noise*, overcomes the limitation of R_{nf}*_Noise* by generating fake tuples based on the prior knowledge of the A_G domain cardinality. Let us assume that A_G domain cardinality is n_d (e.g., for attribute Age, $n_d \approx 130$), a TDS will generate n_d-1 fake tuples, one for each value different from the true one. The resulting distribution is totally flat by construction. However, if the domain cardinality is not readily available, a cardinality discovering algorithm must be launched beforehand (see 4.4).

Correctness. True tuples are grouped in partitions according to the value of their A_G attributes so that the aggregate function can be computed correctly. Fake tuples are eliminated during the aggregation phase by TDSs thanks to their identified characteristics and do not contribute to the computation.

Security. Although TDSs apply *Det-Enc* on A_G, A_G distribution remains hidden to SSI thanks to either white noise such that the fake distribution dominates the true one or controlled noise producing a flat distribution.

Efficiency. TDSs do not need to materialize a large partial aggregate structure as in *S_Agg* because each partition contains tuples belonging to a small set of (ideally one) groups. Additionally, this property guarantees the convergence of the aggregation process and increases the parallelism in all phases of the protocol. However, the price to pay is the production and the elimination afterwards of a potentially very high number of fake tuples (the value is algorithm and data dependent).

4.4 Equi-depth histogram-based protocol

Getting a prior knowledge of the domain extension of A_G allows significant optimizations as illustrated by *C_Noise*. Let us go one step further and exploit the prior knowledge of the real distribution of A_G attributes. The idea is no longer to generate noisy data but rather to produce a uniform distribution of true data sent to SSI by grouping them into equi-depth histograms, in a way similar to [21]. The protocol, named *ED_Hist*, works as follows. Before entering the protocol, the distribution of A_G attributes must be discovered and distributed to all TDSs. This process needs to be done only once and refreshed from time to time instead of being run for each query. The discovery process is similar to computing a Count function Group By A_G and can therefore be performed using one of the protocol introduced above. During the **collection phase**, each TDS uses this knowledge to calculate *nearly equi-depth histograms*, that is a decomposition of the A_G domain into buckets holding *nearly* the same number of true tuples. Each bucket is identified by a hash value giving no information about the position of the bucket elements in the domain. Then the TDS allocates its tuple(s) to the corresponding bucket(s) and sends to SSI couples of the form (*h*(bucketId), *nDet_Enc*(tuple)). During the partitioning step of the **aggregation phase**, SSI assembles tuples belonging to the same buckets in the same partitions. Each partition may contain several groups since a same bucket holds several distinct values. The first aggregation step computes partial aggregations of these partitions and returns to SSI results of the form (*Det_Enc*(group), *nDet_Enc*(partial aggregate)). A second aggregation step is required to combine these partial aggregations and deliver the final aggregation.

Correctness. Only true tuples are delivered by TDSs and they are grouped in partitions according to the bucket they belong to. Buckets are disjoint and partitions contain a small set of grouping values so that partial aggregations can be easily computed by TDSs.

Security. SSI only sees a nearly uniform distribution of *h*(bucketId) values and cannot infer any information about the true distribution of A_G attributes. Note that *h*(bucketId) plays here the same role as *Det_Enc*(bucketId) values but is cheaper to compute for TDSs.

```
Algorithm   ED_Hist (K1, K2, Q)
Input: (TDS's side): the cryptographic keys (K1, K2),
query Q from Querier.
Output: the final aggregation Ω_final.
1  Call distribution discovering algorithm to
discover the distribution
2  begin Collection phase
3    Each connected TDS sends a tuple of the form tup_e
= (h(A_G),nE_K2(Ā_G)) to the SSI. // h(A_G) is the mapping
function applied on the A_G.
4  end
5  begin First Aggregation phase
6    repeat //(on SSI side)
7      SSI groups tup_e with the same h(A_G)
8      TDSs connect to SSI to parallel download
       these groups
9    until all groups in SSI have been sent to TDSs
10   foreach TDS ∈ TDSs do //(on TDS side)
11     repeat
12       Receive tup_e from SSI
13       Decrypt tup_e: A_G ← h^-1(A_Ge); Ā_G ← nE_k2^-1(Ā_Ge)
14       Compute the aggregate values for all groups
           contained in h(A_G): [A_Gi, AGG_i]
15     until no more tuples received from SSI
16     Encrypt these aggregate values: tup_e =
       [E_K2(A_Gi),nE_K2(AGG_i)]
17     Send these encrypted aggregations to SSI
18 end
19 begin Second Aggregation phase
20   repeat //(on SSI side)
21     SSI groups tup_e with the same E_K(A_Gi)
22     TDSs connect to SSI to parallel download
         these groups
23   until all groups in SSI have been sent to TDSs
24   foreach TDS ∈ TDSs do //(on TDS side)
25     repeat
26       Receive tup_e from SSI
27       Decrypt tup_e: A_G ← E_k2^-1(A_Ge); AGG ← nE_k2^-1(AGG_e)
28       Compute the aggregate values for only one
           group A_G: [A_G, AGG]
29     until no more tuples received from SSI
30     Encrypt these aggregate values: [E_K2(A_G),
                                        nE_k2(AGG)]
31     Send these encrypted aggregations to SSI
32 end
33 Filtering phase //evaluate HAVING clause
34 return nE_k1(Ω_final) to Querier by SSI
```

Fig. 6. Histogram-based algorithm

Efficiency. TDSs do not need to materialize a large partial aggregate structure as in *S_Agg* because each partition contains tuples belonging to a small set of groups during the first phase and to a single group during the second phase. As for *C_Noise*, this property guarantees convergence of the aggregation process and maximizes the parallelism in all phases of the protocol. But contrary to *C_Noise*, this benefit does not come at the price of managing fake tuples.

This section shows that the design space for executing complex queries with Group By is large. It presented three rather different alternatives for computing these queries and provided a rough

discussion about their respective correctness, security and efficiency. The next sections compare respectively the security and performance of these alternatives in a deeper way to assess whether one solution dominates the others in all situations or which parameters are the most influential in the selection of the solution best adapted to each context.

5. INFORMATION EXPOSURE ANALYSIS

In this section, in order to quantify the confidentiality of each algorithm, we measure the information exposure of the encrypted data they reveal to SSI by using the approach proposed in [12] which introduces the concept of coefficient to assess the exposure. To illustrate, let us consider the example in Fig. 7 where Fig. 7a is taken from [12] and Fig. 7b is the extension of [12] applied in our context. The plaintext table Accounts is encrypted in different ways corresponding to our proposed protocols. To measure the exposure, we consider the probability that an attacker can reconstruct the plaintext table (or part of the table) by using the encrypted table and his prior knowledge about global distributions of plaintext attributes.

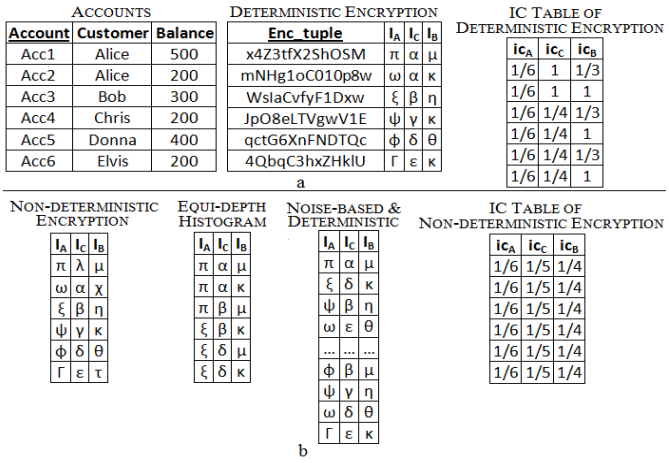


Fig. 7. Encryptions and IC tables

Although the attacker does not know which encrypted column corresponds to which plaintext attribute, he can determine the actual correspondence by comparing their cardinalities. Namely, she can determine that I_A, I_C, and I_B correspond to attributes Account, Customer, and Balance respectively. Then, the IC table (the table of the inverse of the cardinalities of the equivalence classes) is formed by calculating the probability that an encrypted value can be correctly matched to a plaintext value. For example, with *Det_Enc*, $P(\alpha = Alice) = 1$ and $P(\kappa = 200) = 1$ since the attacker knows that the plaintexts *Alice* and *200* have the most frequent occurrences in the Accounts table (or in the global distribution) and observes that the ciphertexts α and κ have highest frequencies in the encrypted table respectively. The attacker can infer with certainty that not only α and κ represent values *Alice* and *200* (*encryption inference*) but also that the plaintext table contains a tuple associating values *Alice* and *200* (*association inference*). The probability of disclosing a specific association (e.g., <*Alice,200*>) is the product of the inverses of the cardinalities (e.g., $P(<\alpha,\kappa> = <Alice,200>) = P(\alpha = Alice) \times P(\kappa = 200) = 1$). The *exposure coefficient* $\mathcal{E}$ of the whole table is estimated as the average exposure of each tuple in it:

$$\varepsilon = \frac{1}{n}\sum_{i=1}^{n}\prod_{j=1}^{k} IC_{i,j}$$

Here, n is the number of tuples, k is the number of attributes, and $IC_{i,j}$ is the value in row i and column j in the IC table. Let's N_j be the number of distinct plaintext values in the global distribution of attribute in column j (i.e., $N_j \leq n$).

Using *nDet_Enc*, because the distribution of ciphertexts is obfuscated uniformly, the probability of guessing the true plaintext of α is $P(\alpha = Alice) = 1/5$. So, $IC_{i,j} = 1/N_j$ for all i, j, and thus the exposure coefficient of *S_Agg* is:

$$\varepsilon_{S\_Agg} = \frac{1}{n}\sum_{i=1}^{n}\prod_{j=1}^{k}\frac{1}{N_j} = 1/\prod_{j=1}^{k} N_j$$

For the nearly equi-depth histogram, each hash value can correspond to multiple plaintext values. Therefore, each hash value in the equivalence class of multiplicity m can represent any m values extracted from the plaintext set, that is, there are $\binom{N_j}{m}$ different possibilities. The identification of the correspondence between hash and plaintext values requires finding all possible partitions of the plaintext values such that the sum of their occurrences is the cardinality of the hash value, equating to solving the NP-Hard *multiple subset sum problem* [11]. We consider two critical values of collision factor h (defined as the ratio G/M between the number of groups G and the number M of distinct hash values) that correspond to two extreme cases (i.e., the least and most exposure) of ε_{ED_Hist}: (1) $h = G$: all plaintext values collide on the same hash value and (2) $h = 1$: distinct plaintext values are mapped to distinct hash values (i.e., in this case, the nearly equi-depth histogram becomes *Det_Enc* since the same plaintext values will be mapped to the same hash value).

In the first case, the optimal coefficient exposure of histogram is:

$$\min(\varepsilon_{ED\_Hist}) = 1/\prod_{j=1}^{k} N_j$$

because $IC_{i,j} = 1/N_j$ for all i, j. For the second case, the experiment in [11] (where they generated a number of random databases whose number of occurrences of each plaintext value followed a Zipf distribution) varies the value of h to see its impact to ε_{ED_Hist}. This experiment shows that the smaller the value of h, the bigger the ε_{ED_Hist} and ε_{ED_Hist} reaches maximum value (i.e., $\max(\varepsilon_{ED\_Hist}) \approx 0.4$) when $h = 1$.

For Noise_based algorithms, when $n_f = 0$ (i.e., no fake tuples), R_{nf_Noise} becomes *Det_Enc* and therefore it has maximum exposure in this case. If n_f is not big enough, since each TDS generates very few fake tuples, the transformed distribution cannot hide some ciphertexts with remarkable (highest or lowest) frequencies, increasing the exposure. The bigger the n_f, the lower the probability that these ciphertexts are revealed. Exceptionally, when the noise is not random (but controlled by domain cardinality of A_G), *C_Noise* has better exposure since all ciphertexts have the same frequency ($IC_{i,j} = 1/N_j$ for all i, j):

$$\varepsilon_{C\_Noise} = \frac{1}{(n_f+1)*n}\sum_{i=1}^{(n_f+1)*n}\prod_{j=1}^{k} IC_{i,j}$$

$$= \frac{1}{n_d*n}\sum_{i=1}^{n_d*n}\prod_{j=1}^{k}\frac{1}{N_j} = 1/\prod_{j=1}^{k} N_j$$

The exposure coefficient gets the highest value when no encryption is used at all and therefore all plaintexts are displayed to attacker. In this case, $IC_{i,j} = 1 \ \forall$ i, j, and thus the exposure coefficient of plaintext table is (trivially):

$$\varepsilon_{P\_Text} = \frac{1}{n}\sum_{i=1}^{n}\prod_{j=1}^{k}1 = 1$$

The information exposures among our proposed solutions are summarized in Fig. 8. In conclusion, *S_Agg* is the most secure protocol. To reach the highest secure level as the *S_Agg*, other protocols must pay some high prices. Specifically, R_{nf}*_Noise* has to generate a very large amount of noise regardless of the value of *G*; *C_Noise* also incurs large noise if *G* is big; and *ED_Hist* must have a significant collision factor.

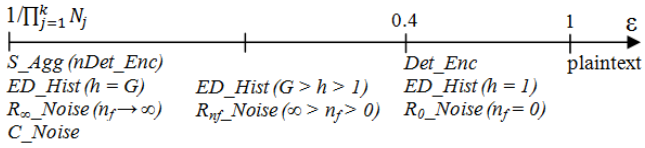


Fig. 8. Information exposure among protocols

6. EXPERIMENTAL EVALUATION

This section evaluates the respective performance of our solutions. We use an analytical cost model for this evaluation and calibrate this model with basic performance measurements performed on a real hardware platform (see 6.2). This choice lies in the difficulty of setting up a very large scale platform of TDSs today while our main objective is to assess whether our protocols can scale up to nation-wide contexts.

6.1 Cost Model

The metrics of interest in this evaluation are the following:

- ***P**_TDS_*: number of TDSs that participate in the computation of a given phase (depending on the protocol, not all connected TDSs may be involved in a computation). This metric reflects the parallelism level of a protocol.
- ***Load**_Q_*: global resource consumption for evaluating a query Q, expressed as the total size of data that TDSs and SSI have to process. This metric reflects the scalability of the solution in terms of capacity of the system to manage a large set of queries in parallel and/or a large set of TDSs to be queried.
- ***T**_Q_*: query response time, reflecting the responsiveness of the protocol. Since the time in the collection phase is application-dependent and is similar for all protocols, and since the time in the filtering phase is also similar for all protocols, T_Q focuses on the time spent on the aggregation phase, which is actually the most complex phase.
- ***T**_local_*: average time that each participating TDS spends to compute the query. This metric reflects the feasibility of the solution because the longer this time, (1) the lower the probability that TDS stays connected during this time and (2) the higher the burden for an individual to accept participating in distributed queries.

The weight associated to each of these metrics is context-dependent, as discussed in Section 6.4. These metrics are computed based on the following main parameters: N_t total number of encrypted tuples sent to SSI by TDSs (without loss of generality, we consider in the model that each TDS produces a single tuple in the collection phase, hence N_t reflects also the number of TDSs participating in the query); *G* number of groups; s_t size of an encrypted tuple; T_t time spent by each TDS to process one tuple (including transfer, cryptographic and aggregation time); N_i^{TDS} number of TDSs that participate in the i^{th} partial aggregation phase (protocol dependent); α, n_{NB}, n_{ED}, reduction factors in the aggregation phase in *S_Agg*, *Noise_based* and *ED_Hist* respectively; n_f number of fake tuples per true tuple in *Noise_based* protocols; *h* the average number of groups corresponding to each hash value in *ED_Hist*.

6.1.1 Secure Aggregation protocol

Because the aggregation phase is iterative, the time spent in this phase is the total time for all iterative steps. In the first step of this phase, the time required to download data from SSI and return temporary result is: $t_1 = \frac{N_t}{N_1^{TDS}} * T_t$; $t_1' = G * T_t$.

Similarly, in step *i* of the aggregation phase, we have:

$t_i = \frac{N_{i-1}^{TDS}}{N_i^{TDS}} * G * T_t$; $t_i' = G * T_t$ ($i = 2 - n$), with *n* is the total number of iterative steps in this phase.

For simplicity, we assume that the reduction factor α in every step is similar:

$$\alpha = \frac{N_t/G}{N_1^{TDS}} = \frac{N_1^{TDS}}{N_2^{TDS}} = \cdots = \frac{N_{n-1}^{TDS}}{N_n^{TDS}}.$$

Since $N_n^{TDS} = 1$, the number of iterative steps is $n = \left\lceil \log_\alpha \frac{N_t}{G} \right\rceil$

The computation time of S_Agg is:

$$T_Q^{S\_Agg} = \sum_{i=1}^{n}(t_i + t_i') = \left\lceil (\alpha+1)\log_\alpha \frac{N_t}{G} \right\rceil * G * T_t$$

To find the optimal time for aggregation phase, let $f(\alpha) = (\alpha + 1)\log_\alpha(N_t/G)$.

We have: $\frac{df}{d\alpha} = \frac{\alpha * ln\alpha - (\alpha+1)}{\alpha * (ln\alpha)^2} * ln\left(\frac{N_t}{G}\right)$

Solving the equation $\frac{df}{d\alpha} = 0$ gives $\alpha \approx 3.6$.

We call α_{op} = 3.6 the optimal reduction factor (i.e., $T_Q^{S\_Agg}$ gets the minimum value when α_{op} = 3.6).

These other metrics are calculated as follows:

$$P_{TDS}^{S\_Agg} = \sum_{i=1}^{n} N_i^{TDS} = \frac{N_t}{G} * \sum_{i=1}^{n} \alpha^{-i}$$

$$Load_Q^{S\_Agg} = \left(N_t + \alpha G \sum_{i=2}^{n} N_i^{TDS} + G\sum_{i=1}^{n} N_i^{TDS}\right) * s_t$$

$$= \left(1 + 2\sum_{i=1}^{n}\alpha^{-i}\right) * N_t * s_t$$

$$T_{local}^{S\_Agg} = \frac{\left(N_t + \alpha G \sum_{i=2}^{n} N_i^{TDS}\right) * T_t}{P_{TDS}^{S\_Agg}}$$

6.1.2 Noise_based protocols

Because all tuples belonging to one group may spread over multiple partitions, the aggregation phase includes two steps.

In the first step, each group contains $(n_f + 1) * N_t / G$ tuples in average, and we assume that there are n_{NB} TDSs handling tuples belonging to one group. The time required to download data from SSI and return temporary result in this step is:

$$t_1 = \frac{(n_f+1) * N_t}{n_{NB} * G} * T_t \; ; \; t_1' = T_t \; ;$$

In the second step, each TDS receives n_{NB} tuples belonging to one group to compute the final aggregation, so the time required is:

$$t_2 = n_{NB} * T_t \; ; \; t_2' = T_t \; ;$$

The computation time of R_{nf}*_Noise* is:

$$T_Q^{R_{nf}\_Noise} = \left(n_{NB} + \frac{(n_f+1) * N_t}{n_{NB} * G} + 2\right) * T_t$$

Apply the Cauchy's inequality, we have:

$$n_{NB} + \frac{(n_f+1)*N_t}{n_{NB}*G} \geq 2 * \sqrt{\frac{(n_f+1)*N_t}{G}}$$

The computation time of R_{nf}_Noise gets optimal value when the optimal reduction factor is: $n_{NB} = \sqrt{\frac{(n_f+1)*N_t}{G}}$.

$$P_{TDS}^{R_{nf}\_Noise} = (n_{NB}+1) * G$$

$$Load_Q^{R_{nf}\_Noise} = \left[(n_f+1) * N_t + 2n_{NB} * G + G\right] * s_t$$

$$T_{local}^{R_{nf}\_Noise} = \sqrt{\frac{(n_f+1)*N_t}{G}} * T_t$$

6.1.3 Histogram-based protocol

Let's h be the average number of groups corresponding to each hash value. By applying the Cauchy's inequality and the same mechanism as in R_{nf}_Noise, the optimal computation time is:

$T_{Q(op)}^{ED\_Hist} = \left(3 * \sqrt[3]{\frac{h*N_t}{G}} + h + 2\right) * T_t$ when the reduction factors in each step are: $n_{ED} = \sqrt[3]{\left(\frac{h*N_t}{G}\right)^2}$; $m_{ED} = \sqrt[3]{\frac{h*N_t}{G}}$

Then, the other metrics are based on these factors as follows:

$$P_{TDS}^{ED\_Hist} = \left(\frac{n_{ED}}{h} + m_{ED} + 1\right) * G$$

$$Load_Q^{ED\_Hist} = (N_t + 2n_{ED} * G + 2m_{ED} * G + G) * s_t$$

$$T_{local}^{ED\_Hist} = \frac{(N_t + n_{ED}*G + m_{ED}*G)*T_t}{(n_{ED}/h + m_{ED} + 1)*G}$$

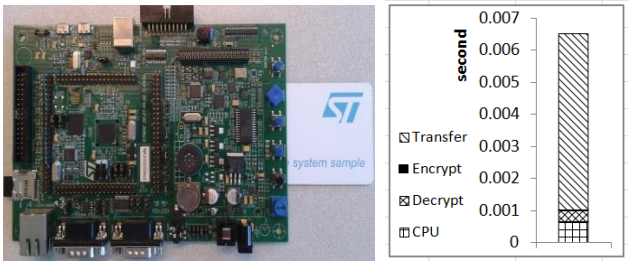
Note that this is just a subset of the complete cost model which can be found in the technical report [20].

6.2 Unit test

To calibrate our model, we performed unit tests on the development board presented in Fig. 9a. This board exhibits hardware characteristics representative of secure tokens-like TDSs, including those provided by Gemalto (the smartcard world leader), our industrial partner. This board has the following characteristics: the microcontroller is equipped with a 32 bit RISC CPU clocked at 120 MHz, a crypto-coprocessor implementing AES and SHA in hardware (encrypting or decrypting a block of 128bits costs 167 cycles), 64 KB of static RAM, 1 MB of NOR-Flash and is connected to a 1 GB external NAND-Flash and to a smartcard chip hosting the cryptographic material. The device can communicate with the external world through USB full speed. The speed in theory is 12 Mbps but the real speed measured with the device is around 7.9 Mbps.

We measured on this device the performance of the main operations influencing the global cost, that is: encryption, decryption, hashing, communication and CPU time, and put these numbers as constants in the formulas. Fig. 9b depicts the internal time consumption of this platform to manage partitions of 4KB. The transfer cost dominates the other costs due to the network latencies. The CPU cost is higher than cryptographic cost because (1) the cryptographic operations are done in hardware by the crypto-coprocessor and (2) TDS spends CPU time to convert the array of raw bytes (resulting from the decryption) to the number format for calculation later. Encryption time is much smaller than decryption time because only the result of the aggregation of each partition needs to be encrypted.

Other TDSs (e.g., smart meters) may be more powerful than smart tokens, although client-based hardware security is always synonym of low power. Anyway, as this section will make clear, the internal time consumption turns out not to be the limiting factor. Hence our choice of considering low-power TDSs in this experiment is expected to broaden our conclusions.



a) b)

Fig. 9. Hardware device & its internal time consumption

6.3 Performance comparisons

In this study, we concentrate on the performance of Group By queries since they are the most challenging to compute. We vary the dataset size (N_t varies from 5 to 65 million), the number of groups (G varies from 1 to 10^6) as well as the number of TDSs participating in the computation as a percentage of all TDSs connected at a given time (varying from 1% to 100%). We fix two parameters and vary the others. When the parameters are fixed, N_t=10^6, G=10^3, s_t=16b, T_t=16µs, h=5 and the percentage of TDS connected is 10% of N_t. We also compute and use the optimal value for all reduction factors as well as forN_t^{TDS}. In the figures, we plot two curves for R_{nf}_Noise protocols, R_2_Noise (n_f = 2) and R_{1000}_Noise (n_f = 1000) to capture the impact of the ratio of fake tuples. We summarize below the main conclusions of the performance evaluation. A more detailed study is provided in a technical report [20].

Level of parallelism (P_{TDS}). Fig. 10a depicts P_{TDS} varying G. For *S_Agg*, when G increases, the iterative merging of partial aggregations has lower convergence and therefore less participating TDSs can be mobilized in parallel to build the aggregations. On the contrary, other protocols can process groups in parallel and independently, so that the level of parallelism increases linearly with G. Fig. 10b depicts P_{TDS} varying N_t. Noise_based protocols seem to benefit most from an increase of N_t in terms of parallelism but the benefit is actually fictitious; it is due to the fact that a higher number of fake tuples are produced and need to be processed (though in parallel).

Resource consumption ($Load_Q$). Fig. 10c and 10d show $Load_Q$ respectively in terms of G and N_t. Not surprisingly, the total load of Noise_based protocols is highest because of the extra processing incurred by fake tuples. However, n_f depends only on N_t, so when G increases, the total load of Noise_based protocols remain constant. Other protocols generate much lower and roughly comparable loads.

Query response time (T_Q). Fig. 10e shows the impact of G over T_Q. In all protocols but *S_Agg*, T_Q depends on the total number of tuples in each group (resp. bucket for *ED_Hist*) because all groups (resp. buckets) are processed in parallel. Hence, when G increases while N_t remains constant, the number of tuples in each group (resp. bucket) decreases and so does T_Q. In *S_Agg*, when G increases, the size of each partial aggregation increases accordingly, and so does the time to process it and in

consequence, so does T_Q. Fig. 10f shows that, for *ED_Hist*, when N_t increases, the number of TDSs which can be mobilized for processing increases accordingly, leading to a minimal impact on execution time. This statement is true also for R_{nf}_Noise protocols with the difference that the greater number of fake tuples generates extra work which is not entirely absorbed by the increase of parallelism. For *S_Agg*, the number of iterative steps increases with N_t and so does T_Q.

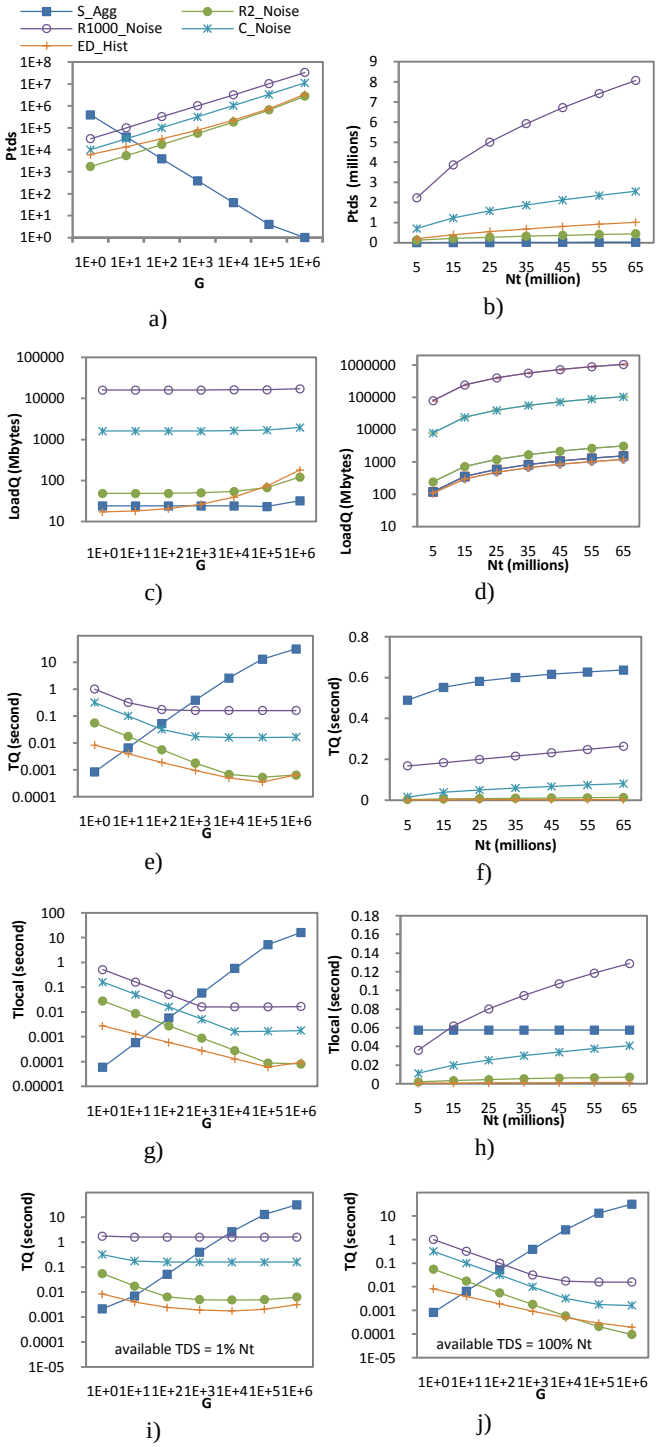


Fig. 10. Performance evaluations

***Local execution time* (T_{local}).** Fig. 10g and 10h plot the average execution time of every participating TDSs varying G and N_t respectively. It shows that all protocols benefit from an increase of G except *S_Agg*. This is due to the fact that, in *S_Agg*, less TDSs can participate in the parallel computation, and therefore each TDS has to process a higher load of bigger partial aggregations. Other protocols benefit from the fact that the computing load is shared evenly between TDSs. Fig. 10h shows that all protocols but Noise_based protocols are insensitive to an increase of N_t again thanks to independent parallelism. The bad behaviour of Noise_based protocols is explained by the fact that the number of fake tuples increases linearly with N_t and this increased load cannot be entirely absorbed by parallelism because the number of TDSs available for the computation is bounded in this setting by 10% of the participating TDSs.

***Elasticity issues*.** A distributed and parallel system is said to be elastic if it can mobilize smoothly a variable part of its computing resources to meet run time requirements. Fig. 10i,e,j measures the elasticity of all protocols by varying the computing resource and assessing its impact on T_Q. The computing resource is materialized here by the number of TDSs which can be mobilized to contribute to a given computation. It is expressed by a percentage of the TDSs contributing to the collection phase. Fig. 10i (resp. Fig. 10j, Fig. 10e) considers scarce (resp. abundant, intermediate) computing resource in the sense that only 1% (resp. 100%, 10%) of the TDSs contributing to the collection phase contribute to the rest of the query computation. Comparing these figures shows that, when the resource is scarce, the parallel computation is not completely deployed, resulting in a longer time to answer the query and vice-versa. Since *S_Agg* does not depend on the number of available TDSs (but on G and on the memory size of TDS), its performance is not impacted by a fluctuation of the resource available. In other words, *S_Agg* has lowest elasticity.

6.4 Conclusion: Trade-off between criteria

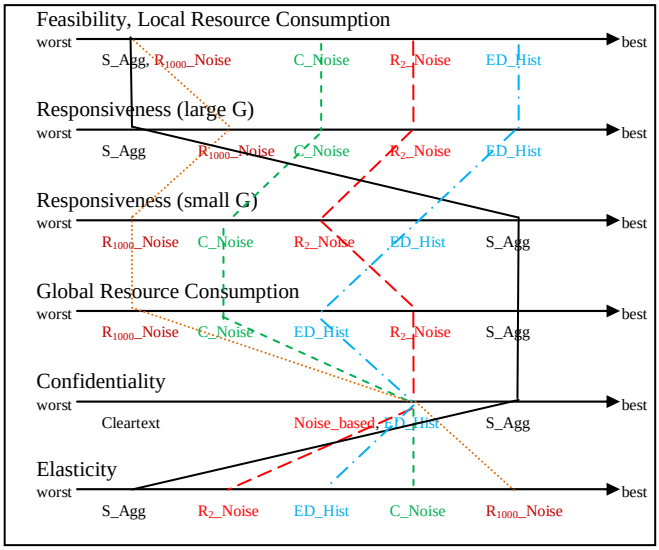


Fig. 11. Comparison among solutions

Fig. 11 summarizes and complements the experimental results described above through a qualitative comparison of our proposed protocols over all criteria of interest to perform a choice.

Each axis can be interpreted as follows. Local resource consumption axis refers to T_{local} metrics and compares the

protocols in terms of feasibility, i.e., is the resource consumed by a single TDS compatible with the actual computing power of the targeted TDSs. This question is particularly relevant for low-end TDSs (e.g., smart tokens) and of lesser interest for high-end TDSs. S_Agg is at the worst extremity of this axis because the final aggregation must be done by a single TDS while ED_Hist occupies the other extremity thanks to its capacity to evenly share the load among all TDSs. Noise_based protocols are in between because they also share the load evenly but at the price of managing a large number of fake tuples. Note that the relative position of S_Agg and ED_Hist is reversed in the Global Resource Consumption axis which refers to $Load_Q$ metrics and compares the scalability of the protocols in terms of number of parallel queries which can be computed. Indeed, the total number of TDSs mobilized by S_Agg for one single query computation is much smaller than that of ED_Hist. Regarding the Responsiveness axis, the relative ordering of S_Agg and ED_Hist actually differs depending on G. According to Fig. 10, S_Agg outperforms ED_Hist for small G (smaller than 10) and is dominated by ED_Hist for larger G. Finally, Elasticity axis is a direct translation of the conclusions drawn in Section 6.3 and Confidentiality axis recalls the conclusion of Section 5.

This figure makes clear that no protocol outperforms the others, though Noise_based protocols are always dominated either by S_Agg or ED_Hist. Let us consider a first scenario where individuals manage their data (e.g., their medical folder) thanks to a secure Personal Data Server embedded in a smart token-like TDS [3]. In such scenario, individuals are likely to connect their TDS seldom, for short periods of time (e.g., when visiting a doctor) and would prefer save resource for executing their own tasks rather than being slow down by the computation of external queries. According to Fig. 11, ED-Hist best matches the above requirements. Conversely, let us consider a smart metering platform composed of power meter-like TDSs, connected all the time and mostly idle. In this case, TDSs' owners do not care how much resources are monopolized to compute queries and the primary concern is for the distribution company to maximize the capacity to perform global computation. S_Agg is more appropriate in this case. In short, ED_Hist and S_Agg are the two best solutions and the final choice depends on the weight associated to each axis for a given application.

7. RELATED WORKS

This work has connections with related studies in different domains, namely protection of outsourced (personal) databases, statistical databases and PPDP, SMC and finally secure aggregation in sensor networks. We review these works below.

Security in outsourced databases. Outsourced database services or DaaS [21] allow users to store sensitive data on a remote, untrusted server and retrieve desired parts of it on request. Many works have addressed the security of DaaS by encrypting the data at rest and pushing part of the processing to the server side. Searchable encryption has been studied in the symmetric-key [6] and public-key [8] settings but these works focus mainly on simple exact-match queries and introduce a high computing cost. Agrawal et al. [2] proposed an order preserving encryption (OPE) scheme, supporting range and aggregate queries, but OPE relies on the strong assumption that all plaintexts in the database are known in advance and order-preserving is usually synonym of weaker security. Bucketization-based techniques [21, 24] use distributional properties of the dataset to partition data and design indexing techniques that allow approximate queries over encrypted data. These techniques often support limited types of queries and lack of a precise analysis of the performance/security tradeoff introduced by the indexes. To overcome this limitation, the work in [12] quantitatively measures the resulting inference exposure. Other works introduce solutions to compute basic arithmetic over encrypted data, but homomorphic encryption [30] supports only range queries, fully homomorphic encryption [19] is unrealistic in term of time, and privacy homomorphism [22] is insecure under ciphertext-only attacks [29]. Hence, optimal performance/security tradeoff for outsourced databases is still regarded as the holy grail.

Statistical Database and PPDP. Statistical databases (SDB) [15] are motivated by the desire to compute statistics without compromising sensitive information about individuals. This requires trusting the server to perform query restriction or data perturbation, to produce the approximate results, and to deliver them to untrusted queriers. So, the SDB model is orthogonal to our context since (1) it assumes a trusted third party (i.e., the SDB server) and (2) it usually produces approximate results to prevent queriers from conducting inferential attack [15]. For its part, Privacy-Preserving Data Publishing (PPDP) [4] provides a non trusted user with some sanitized data produced by an anonymization process such as k-anonymity, l-diversity or differential privacy to cite a few [4]. Similarly, PPDP is orthogonal to our context since it assumes again a trusted third party (i.e., the publisher) and produces sanitized data of lower quality to match the information exposure dictated by a specific privacy model.

Secure Multi-party Computation. Secure multi-party computation (SMC) allows N parties to share a computation in which each party learns only what can be inferred from their own inputs (which can then be kept private) and the output of the computation. This problem is represented as a combinatorial circuit which depends on the size of the input. The resulting cost of a SMC protocol depends on the number of inter-participant interactions, which in turn depends exponentially on the size of the input data, on the complexity of the initial function, and on the number of participants. Despite their unquestionable theoretical interest, generic SMC approaches are impractical where inputs are large and the function to be computed complex. Ad-hoc SMC protocols have been proposed [25] to solve specific problems/functions but they lack of generality and usually make strong assumptions on participants' availability. Hence, SMC is badly adapted to our context.

Secure Data Aggregation. Wireless sensor networks (WSN) [5] consist of sensor nodes with limited power, computation, storage, sensing and communication capabilities. In WSN, an aggregator node can compute the sum, average, minimum or maximum of the data from its children sensors, and send the aggregation results to a higher-level aggregator. WSN has some connection with our context regarding the computation of distributed aggregations. However, contrary to the TDS context, WSN nodes are highly available, can communicate with each other in order to form a network topology to optimize calculations. Other work ([10]) uses additively homomorphic encryption for computing aggregation function on encrypted data in WSN but fails to consider queries with GROUP BY clauses. [26] protects data against frequency-based attacks but considers only point and range queries.

As a conclusion, and to the best of our knowledge, our work is the first proposal achieving a fully distributed and secure solution to compute general SQL queries (without external joins) over a large set of participants.

8. CONCLUSION

An ever increasing amount of personal data is collected and ends-up on servers. Decentralized architectures, devised to help individuals better protect their privacy, hinder global treatments and queries, impeding the development of services of great interest. This paper is a first attempt to fill this gap. It capitalizes on secure hardware advances promising soon the presence of a Trusted Execution Environment at low cost in any client device (trackers, smart meters, sensors, cell phones and other personal devices).

Based on this statement, we have proposed new query execution protocols to compute general SQL queries (without external joins) while maintaining strong privacy guarantees. The objective was not to find the most efficient solution for a specific problem but rather to perform a first exploration of the design space. We proposed three very different protocols and compare them on different axis. The encouraging conclusion is that good performance/security trade-off can be found in many situations and that the proposed protocols can scale up to nation-wide contexts.

We expect that this work will pave the way for the definition of future fully decentralized privacy-preserving querying protocols. The main research directions we foresee are: (1) support external joins (i.e., joins involving data hosted in different TDSs), (2) extend the threat model to (a small number of) compromised TDSs and (3) perform performance study on large scale TDS platforms. The on-going deployment of very large TDS platforms (e.g., the Linky power meters installed by EDF in France or the growing interest for PCEHR hosted in secure tokens) would enable point (3) while providing a strong motivation to investigate issues (1) and (2).

9. ACKNOWLEDGMENTS

The authors wish to thank Philippe Bonnet from University of Copenhagen for fruitful discussions on this paper.

10. REFERENCES

[1] Agrawal, R., Kiernan, J., Srikant, R., Xu, Y.: Hippocratic databases. VLDB, pp 143-154. Hong Kong, (2002).

[2] Agrawal, R., Kiernan, J., Stikant, R., Xu, Y.: Order-preserving encryption for numeric data. ACM SIGMOD, pp. 563-574. Paris (2004)

[3] Allard, T., Anciaux, N., Bouganim, L., Guo, Y., Le Folgoc, L., Nguyen, B., Pucheral, P., Ray, Ij., Ray, Ik., Yin, S.: Secure Personal Data Servers: a Vision Paper. VLDB, pp. 25-35. Singapore (2010)

[4] Fung, B. C. M., Wang, K., Chen, R., Yu, P. S.: Privacy-Preserving Data Publishing: A survey of Recent Developments. ACM Computing Surveys, 42(4), 2010.

[5] Alzaid, H., Foo, E., Nieto, J.G.: Secure Data Aggregation in Wireless Sensor Networks: A Survey. AISC, vol. 81, pp. 93-105, (2008)

[6] Amanatidis, G., Boldyreva, A., O'Neill, A.: Provably-secure schemes for basic query support in outsourced databases. DBSec, pp. 14-30, (2007)

[7] M. Fischlin, B. Pinkas, 1-R. Sadeghi, T. Schneider, I. Visconti : Secure set intersection with untrusted hardware tokens. In CT-RSA, (2011).

[8] Bellare, M., Boldyreva, A., O'Neill, A.: Deterministic and efficiently searchable encryption. CRYPTO, pp. 535–552, (2007)

[9] StreamSQL Guide, available at : http://www.streambase.com/developers/docs/latest/streamsql/

[10] Castelluccia, C., Mykletun, E., Tsudik, G.: Efficient Aggregation of Encrypted Data in Wireless Sensor Networks. IEEE Mobiquitous, (2005)

[11] Ceselli, A., Damiani, E., De Capitani di Vimercati, S., Jajodia, S., Paraboschi, S., Samarati, P.: Modeling and assessing inference exposure in encrypted databases. ACM TISSEC, vol 8(1), pp. 119-152, (2005)

[12] Damiani, E., De Capitani di Vimercati, S., Jajodia, S., Paraboschi, S., Samarati, P.: Balancing confidentiality and efficiency in untrusted relational DBMSs. ACM CCS, pp. 93-102, (2003)

[13] Abadi, D., Carney, D., Cetintemel, U., Cherniack, M., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.: Aurora: a new model and architecture for data stream management, VLDB Journal, 12(2):120-139, (2003).

[14] Anciaux, N., Bonnet, P., Bouganim, L., Nguyen, B., Sandu Popa, I., Pucheral, P.: Trusted Cells: A Sea Change for Personal Data Services. CIDR, Asilomar, USA, (2013)

[15] Fayyoumi, E., John Oommen, B.: A survey on statistical disclosure control and micro-aggregation techniques for secure statistical databases. Softw. Pract. Exper, vol 40(12):1161-1188, (2010)

[16] White, J., Clarke, S., Dougherty, B., Thompson, C., and Schmidt, D.: R&D Challenges and Solutions for Mobile Cyber-Physical Applications and Supporting Internet Services. Journal of Internet Services and Applications, vol. 1(1):45-56, (2010)

[17] Popa, R. A., Redfield, C. M. S., Zeldovich, N., Balakrishnan, H.: CryptDB: protecting confidentiality with encrypted query processing. ACM SOSP, pp85-100. New York, (2011)

[18] The World Economic Forum. Rethinking Personal Data: Strengthening Trust. May 2012.

[19] Gentry, C.: Fully homomorphic encryption using ideal lattices. STOC, pp. 169-178. Maryland, (2009)

[20] To, Q.C., Nguyen, B., Pucheral, P.: Secure Global Protocol in Personal Data Server. Technical report, Versailles (France), 2013. http://www.cse.hcmut.edu.vn/~qcuong/INRIA/TechReport.pdf

[21] Hacigumus, H., Iyer, B., Li, C., Mehrotra, S.: Executing SQL over encrypted data in database service provider model. ACM SIGMOD, pp. 216-227. Wisconsin (2002)

[22] Hacigumus, H., Iyer, B. R., Mehrotra, S.: Efficient execution of aggregation queries over encrypted relational databases. DASFAA, pp. 125-136. Korea (2004)

[23] Lam, H.: A Novel Method to Construct Taxonomy Electrical Appliances Based on Load Signatures,. IEEE Transactions on Consumer Electronics, 2007.

[24] Hore, B., Mehrotra, S., Canim, M., Kantarcioglu, M.: Secure multidimensional range queries over outsourced data. VLDB Journal, vol. 21(3):333-358, (2012)

[25] Kissner, L., Song, D. X.: Privacy-Preserving Set Operations. In *CRYPTO*, pp. 241–257, (2005).

[26] Liu, H., Wang, H., Chen, Y.: Ensuring Data Storage Security against Frequency-based Attacks in Wireless Networks. DCOSS, pp. 201-215. California (2010)

[27] Locher, T.: Foundations of Aggregation and Synchronization in Distributed Systems. ETH Zurich, isbn 978-3-86628-254-4, (2009)

[28] de Montjoye, Y-A., Wang, S. S., Pentland, A.: On the Trusted Use of Large-Scale Personal Data. IEEE Data Eng. Bull. 35(4): 5-8 (2012)

[29] Mykletun, E., Tsudik, G.: Aggregation queries in the database-as-a-service model. DBSec, pp. 89-103. France (2006)

[30] Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. EUROCRYPT, pp. 223-238, (1999)