Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Technical Report Nr. 735

Systems Group, Department of Computer Science, ETH Zurich

Query Processing on Encrypted Data in the Cloud

by

Stefan Hildenbrand, Donald Kossmann, Tahmineh Sanamrad,

Carsten Binnig, Franz Faerber and Johannes Woehler

Date:
12.092011

inf | Informatik
Computer Science

Query Processing on Encrypted Data in the Cloud

Stefan Hildenbrand*, Donald Kossmann*, Tahmineh Sanamrad*,
Carsten Binnig†, Franz Faerber‡ and Johannes Woehler‡

*Systems Group, ETHZ
Zurich, Switzerland
{firstname.lastname}@inf.ethz.ch

†DHBW Mannheim
Mannheim, Germany
{firstname.lastname}@dhbw-mannheim.de

‡SAP AG
Walldorf, Germany
{firstname.lastname}@sap.com

***Abstract*—This paper explores a new encryption technique called POP. POP addresses the need to encrypt databases in the cloud and to execute complex SQL queries on the encrypted data efficiently. POP can be configured to meet different privacy requirements and attacker scenarios. Two such scenarios, referred to as *domain attack* and *frequency attack*, are studied in detail in this paper. Privacy and performance experiments conducted using the TPC-H benchmark show that POP makes it indeed possible to achieve good privacy with affordable performance overheads in many cases.**

I. Introduction

Believing the trade press, cloud computing is the next big thing. Cloud computing promises reduced cost, flexibility, higher availability, and more focus on the core business of an organization. Virtually all players of the IT industry are jumping on the cloud computing band wagon.

Believing the trade press again, the only concern that seems to be able to stop cloud computing is security [1]. The goal of this paper is to address one of these concerns and present a novel encryption scheme, called POP. POP allows to keep the data encrypted in the cloud and process a wide range of SQL queries in the cloud without decrypting the data.

Obviously, there are many ways to encrypt data in a database system. Most database products encrypt data in a coarse-grained way; e.g., Oracle encrypts in the granularity of a file [2]. This approach protects against the loss of a disk, but it does not protect against most attacks in the cloud. In the cloud, such an approach mandates that either the encrypted data is decrypted in the cloud, thereby failing to protect against malicious database administrators or man-in-the-middle attacks. Alternatively, the data must be shipped to a trusted domain in an encrypted form, thereby defeating the purpose of cloud computing and the *database-as-a-service* paradigm. The alternative is to encrypt in a fine-grained way such as an individual field of a tuple or a small set of tuples. Again, there are several encryption techniques for this purpose and an overview is given in Section VII.

For fine-grained encryption, there is a fundamental *performance / security* trade-off. POP is a novel, fine-grained encryption scheme that can be configured to cover the whole spectrum. Concretely, POP has an integer parameter that can be set in the range of 1 to ∞. If set to ∞, POP behaves like a strong encryption scheme (e.g., AES). If set to 1, then POP behaves like an order-preserving encryption scheme. Typically, this parameter will be set to something in the middle (e.g., 10). Surprisingly, it turns out that with such a parameter setting, POP has good performance for a wide range of SQL queries and high entropy, resulting in good privacy.

The design of POP was motivated by a use case of a Swiss bank. In the last few years, database administrators of several Swiss and Liechtenstein banks sold customer data to German and French tax authorities and published such data on Wikileaks. POP is a crucial building block to fix this use case and we have worked extensively with our collaboration partner to apply POP to their specific scenario. In addition to its nice performance and privacy properties, POP has a number of additional advantages. First, POP can be implemented *on top of* an off-the-shelf relational database system or database-as-a-service offering (e.g., SQL Azure or Amazon RDS) without changing the database system. Second, POP can be composed nicely with other fine-grained techniques such as homomorphic encryption [3] or strong encryption (e.g., [4]) in order to cover a wider range of use cases: As will become clear, POP was specifically designed to encrypt certain information such as customer names and dates whereas other schemes are more appropriate for other kinds of information.

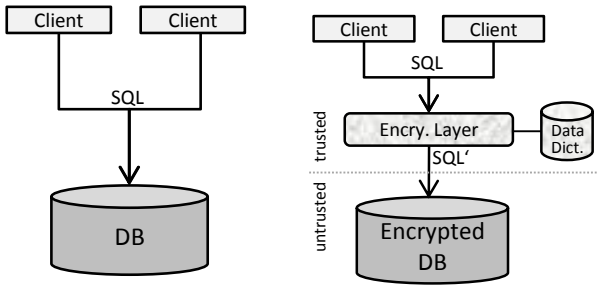
In summary, the main contributions of this paper are a presentation of POP, a detailed description of the attack scenarios that POP addresses, and a comprehensive evaluation of the privacy (i.e., entropy) and performance characteristics of POP using the TPC-H benchmark. The remainder of this document is organized as follows: Section II gives the problem statement. Section III describes the POP technique. Section IV shows how queries can be rewritten in the context of POP and gives implementation details. Section V presents the results of privacy experiments. Section VI gives the results of performance experiments. Section VII discusses related work. Section VIII contains conclusions and avenues for future work.

II. Problem Statement

This section describes the client-server architecture and attack scenarios that POP is trying to address.

A. Encryption Layer

Figure 1a shows the traditional client-server architecture of running applications on top of a database system. The



(a) Traditional (b) Encryption Layer

Fig. 1. Extended Client-server Database Architecture

application or end user issues SQL statements to the database server. The database server executes these SQL statements and returns the results.

Figure 1b shows the extended architecture studied in this paper. This architecture has also been assumed by [5], [6]. In this architecture, the applications and the database system are unchanged. That is, applications and end users issue the same SQL statements as in the traditional system of Figure 1a. Furthermore, the same off-the-shelf database systems can be used in the backend (e.g., Amazon RDS or Oracle).

In the architecture of Figure 1b, privacy is implemented as part of an *Encryption Layer* in the following way. First, the database is encrypted using a *data dictionary*. This dictionary is kept in a separate database that is only accessible by the Encryption Layer. The dictionary keeps for every clear text value one (or several) codes which are used in the encrypted database. For instance, the clear text value *Elton John* for a customer name could be represented by the codes 5 and 14 in the encrypted database. Second, the Encryption Layer rewrites SQL queries and update statements. For instance, the query `SELECT * FROM Customer WHERE name = ''Elton John''` could be rewritten into the query `SELECT * FROM Customer WHERE name IN (5,14)`. Again and obviously, query rewrite will be based on the dictionary. Finally, the Encryption Layer decodes results returned from the database system in the cloud. As a result of this step, an application program (or user) only sees clear text values and encryption is completely encapsulated in the Encryption Layer.

In the architecture of Figure 1b, the Encryption Layer is assumed to be *thin* and *trusted*. Thin means that not much computational power is needed to implement the Encryption Layer; the heavy weight-lifting of executing joins, aggregates, etc. is expected to be carried out in the database. Furthermore, (virtually) no administration is required at the Encryption Layer and simple database systems such as SQLite or MySQL are sufficient. Indexing of the data dictionary is automatic and all that is needed is a back-up service for the data dictionary.

Trusted means that only the owner of the data has access to the data dictionary. For users of public clouds who try to protect their data against external adversaries, the Encryption Layer and the data dictionary are deployed on machines inside of the users' organization. At the Swiss bank, the data dictionary is deployed on machines in a separate security domain which is maintained by a few highly trusted engineers.

TABLE I
ATTACKER MODEL ASSUMPTIONS

Attack Scenario	Attack Type	Attacker has access to	no access to
Domain Attack	Passive	POP-encrypted database Domain of the data	Encryption layer Original queries Rewritten queries Statistics about data
Frequency Attack	Passive	POP-encrypted database Domain of the data Statistics about data	Encryption layer Original queries Rewritten queries

B. Attack Scenarios and Running Example

There are many ways how information can be leaked in the architecture of Figure 1b. For the Swiss bank, we studied two specific scenarios called *domain* and *frequency attack*. In these scenarios, the adversary has full (root privilege) access to the encrypted database and the machines that run the encrypted database. Furthermore, the adversary has a-priori knowledge of the values and the value distribution of the domains stored in the database. Other scenarios in which the adversary has, e.g., partial access to the Encryption Layer are beyond the scope of this paper. Furthermore, this paper does not address the so-called *query log attack* [7] in which the adversary can infer information from the sequence of queries submitted and the state changes of the databases. Looking at such scenarios is left for future work.

Specifically, the two attacker models addressed in this work are depicted in Table I. They can be best described using an example.

Customer(<u>name</u>, city, info)
Order(<u>id</u>, *cust*, info)

name is a string and a key of *Customer*; *id* is an integer and a key of *Order*. *cust* in *Order* is a foreign key that references a *Customer*. The *info* fields of both tables contain other information that may be relevant to a *Customer* or *Order* (e.g., rating, balance, price, product, shipDate, etc.); for brevity, we do not detail these attributes here and explain them if needed as we go along.

The first scenario is called *domain attack*. In this scenario, it is assumed that the attacker has complete knowledge of the domain of values that are encrypted. In the running example, the attacker could know the names of all customers from a telephone book and try to infer the *info* for one or several customers. As will become clear in Section III, the domain attack imposes specific requirements on the encryption of the *name* attribute in the *Customer* table.

The second scenario is called *frequency attack*. In this scenario, the attacker has precise knowledge of all the values of the domain (as in the domain attack) and in addition the frequencies of each value in the database. For instance, the attacker could know the names of all cities in the database and how many customers exist in each city – again, using

the local telephone book of that city. The frequency attack imposes even higher requirements on the encryption scheme. Again, the required properties of the encryption scheme are discussed in detail in the next section.

C. Problem Statement

In summary, the problem addressed in this paper is to protect organizations against domain and frequency attacks from internal and external adversaries by encrypting the data. Good performance is achieved by processing the data *inside* the database system *without* decrypting the data. In general, it is not possible to achieve strong encryption (i.e., maximum entropy) and high performance at the same time [8]. Fortunately, most SQL queries exhibit only specific, pre-defined access patterns to the data. POP exploits exactly this fact in order to achieve both goals (privacy and performance) for a large class of SQL queries; e.g., functional joins, most subselects, group-bys, ranking and Top N, range predicates, LIKE predicates, and general comparisons.

III. POP Techniques

This section introduces POP and shows how it can be applied to protect against domain and frequency attacks and achieve good performance at the same time. Before describing POP, we define different properties of encryption schemes and explain how these properties impact privacy and performance.

A. Properties of Encryption

1) Definitions and Example: Let x, y be clear text values of a domain that needs to be protected. For example, *Madonna* and *Elton John* could be names of customers of a Swiss bank. Let enc be a function that maps a clear text value to a set of codes. That is, $enc(x)$ are the codes that are used to represent x in the encrypted database. For each domain that needs to be protected in the database, such an encryption function must be defined. enc and all other encryption functions are stored in a dictionary as shown in Figure 1b. It is assumed that an attacker does not have access to that dictionary. The enc function can have none, one, or several of the following properties:

- *Equality-preserving (EP):* Each value corresponds to exactly one code; i.e.,
  $x = y \implies \forall c_x \in enc(x), c_y \in enc(y) : c_x = c_y$.
- *Order-preserving (OP):* If a value is smaller than another value, then all its codes are smaller than the codes of the other value; i.e.,
  $x < y \implies \forall c_x \in enc(x), c_y \in enc(y) : c_x < c_y$.
- *Homomorphism-preserving:* For any homomorphism, f:
  $enc(f(x, y)) = f(enc(x), enc(y))$.

The properties of the encryption function determine which kinds of queries can be executed on the encrypted database in the cloud and, consequently, how queries must be rewritten by the Encryption Layer of Figure 1b. Equality preservation, for instance, is important to implement equality predicates. Accordingly, order preservation is important to implement range or `LIKE` predicates such as searches for customer names with wildcards; this observation has been exploited by Agrawal et al. [9]. In general, equality preservation and order preservation are important for the encryption of *dimensions*; e.g., customer names and dates. Homomorphism-preserving encryption is needed in order to compute aggregate functions on *metrics*; e.g., the sum of "price × volume" of orders. This work focuses on the encryption of dimensions (e.g., names and dates) and therefore studies the privacy / performance tradeoffs of encrypting such columns. There have been recent break-throughs in homomorphic encryption for metrics [10]; studying such techniques is beyond the scope of this paper.

TABLE II
EXAMPLE DICTIONARIES: VARY ENCRYPTION PROPERTIES

clear text	EP/OP	EP/NOP	NEP/OP	NEP/NOP
Beatles	1	5	{ 1,2,4 }	{ 2, 11, 13 }
Elton John	2	3	{ 5,6 }	{ 4, 12 }
Madonna	7	2	{ 8 }	{ 8, 12 }
Metallica	9	8	{ 11,12,13 }	{ 1, 5 }

Table II shows examples (using customer names) of different combinations of equality-preserving (denoted as EP), non equality-preserving (NEP), order-preserving (OP), and non order-preserving (NOP) dictionaries. In the literature, EP is often also called *deterministic encryption* whereas NEP is equivalent to *probabilistic encryption*. These examples demonstrate how queries can be processed on an encrypted database. For instance, the query `SELECT * FROM Customer WHERE name = ''Madonna''` can easily be rewritten to `SELECT * FROM Customer WHERE name = 2` in the EP/NOP scheme. On the negative side, processing a query like `SELECT * FROM Customer WHERE name LIKE ''M%''` with the EP/NOP scheme involves either reading the whole *Customer* table and post-filtering the results in the Encryption Layer or enumerating all relevant codes in an `IN` predicate; i.e., `SELECT * FROM Customer WHERE name IN (2, 8)`. As shown in Section VI, both approaches can result in poor performance.

Just as an NOP scheme, probabilistic encryption (i.e., NEP) can impact the performance of queries. The query `SELECT * FROM Customer WHERE name = ''Madonna''` is rewritten into the following query using the NEP/NOP dictionary: `SELECT * FROM Customer WHERE name IN (8, 12)`. Again, this rewritten query is likely to be more expensive to evaluate than the original query on a non-encrypted database.

In summary, these examples illustrate that encryption may impact query performance. The remainder of this subsection discusses how the properties of an encryption scheme impact the privacy with regard to domain and frequency attacks.

2) Domain Attack: In order to protect data against domain attacks, the encryption scheme must be **non order-preserving** (i.e., NOP). To see why, assume that an attacker knows that the database contains information about the four customers Beatles, Elton John, Madonna, and Metallica. Furthermore, the attacker sees the codes 1, 2, 7, and 9 and knows that the encryption scheme is EP/OP. Obviously, it is trivial for the

TABLE III
EP/POP DICTIONARY (TWO RUNS)

clear text	Run 1	Run 2	Code ($\langle$ run, code-id $\rangle$)
Beatles	4		$\langle 1,4\rangle$
Elton John		1	$\langle 2,1\rangle$
Madonna		7	$\langle 2,7\rangle$
Metallica	5		$\langle 1,5\rangle$

TABLE IV
NEP/POP DICTIONARY (THREE RUNS)

clear text	Run 1	Run 2	Run 3	Codes
Beatles	1		2	$\{\langle 1,1\rangle\langle 3,2\rangle\}$
Elton John		1,4		$\{\langle 2,1\rangle\langle 2,4\rangle\}$
Madonna	3	7		$\{\langle 1,3\rangle\}\langle 2,7\rangle$
Metallica	5	9	3	$\{\langle 1,5\rangle\langle 2,9\rangle\langle 3,3\rangle\}$

attacker to infer that Code 1 corresponds to the Beatles, Code 2 to Elton John, etc.

A domain attack against an NEP/OP encryption scheme is more difficult because the attacker may not know how many codes are used to encrypt the Beatles. Nevertheless, an attacker can easily infer that Code 1 belongs to the Beatles and Code 13 belongs to Metallica in the NEP/OP scheme of Table II.

3) Frequency Attack: Since the frequency attack is a generalization of the domain attack, an NOP encryption scheme is also required to protect against the frequency attack. In addition, an encryption scheme must be **non equality-preserving** (or probabilistic) to ensure privacy against frequency attacks. To see why, assume that the attacker knows that the Beatles have placed four orders, Elton John has placed ten orders, etc. If the Beatles are represented by a single code in an EP/NOP encryption scheme, then the attacker can easily infer that code by counting the number of occurrences of each *cust* code in the *Order* table. Of course, the situation is better if many customers have placed the same number of orders or the attacker only knows the probability distribution of frequencies. However, to be safe in the general case, an NEP encryption scheme is required to protect confidential data against a frequency attack.

In summary, NOP is required to protect against domain attacks; in this case, EP/NOP is sufficient. To protect against frequency attacks NEP/NOP is required. The remainder of this section presents two specific schemes for domain and frequency attacks: EP/POP and NEP/POP. EP/POP is a special instantiation of an EP/NOP scheme for domain attacks. Accordingly, NEP/POP is a special NEP/NOP scheme for frequency attacks. Both techniques, EP/POP and NEP/POP, are derived from a scheme that we call partially order-preserving encryption or POP, for short.

B. POP: Basic Idea

Table III illustrates the basic idea of POP: Rather than creating a total order or a completely random order, POP creates a *partial order* of the codes. Technically, the codes are partitioned into so-called *runs*. Within each run, all codes are order-preserving. Across runs, no order can be implied between two codes.

Table III shows an equality-preserving POP dictionary with two runs. Such an EP/POP dictionary can be used against domain attacks. As shown in the fourth column of Table III, codes have two fields: (a) the run and (b) an (order-preserving) code-id value within that run.

Conceptually, POP covers the spectrum between the two extreme cases. OP is the same as POP with one run and strong encryption (i.e., NOP) is the same as POP with an infinite number of runs (i.e., every value has its own run). Surprisingly, as shown in Section V, POP has similar entropy as strong encryption starting already with a small number of runs (i.e.,10). To ensure the highest possible entropy with POP, codes should be distributed over the runs using a Uniform distribution. (Obviously, POP does not help, if all codes are allocated in the same run.)

Just like OP, POP allows to process a large class of queries efficiently without decrypting the data. Details of how this is done for a large variety of queries are given in Section IV. Section IV also shows how updates can be processed, how integrity constraints can be maintained, and how POP databases can be created. To give an impression on how queries are rewritten using POP, consider a query that asks for all customers with `name LIKE ''M%''`. This query can be rewritten into the following query using the dictionary of Table III:

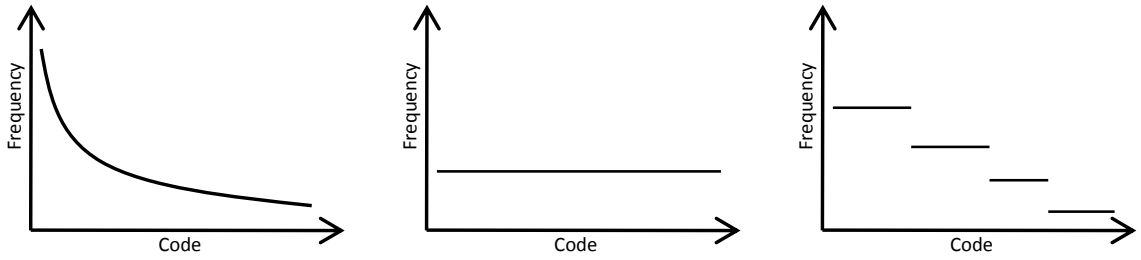
```
SELECT * FROM Customer
WHERE (run = 1 AND code-id > 4)
   OR (run = 2 AND code-id > 4)
```

Obviously, the performance overhead of POP (as compared to no encryption) depends on the number of runs; the more runs, the higher the overhead. Likewise, the *degree of privacy* depends on the number of runs; the more runs, the more privacy. We will study this performance / privacy trade-off in detail in Sections V and VI. It will become clear that POP allows to achieve both, good privacy and good performance for a wide range of queries. In particular, POP achieves good performance even with a fairly large number of runs.

In the example above, there are actually many possible code ranges that can be used to implement a `LIKE` predicate; any one can be used. For instance, *code-id* > 6 could have also been used for the second run. This way, POP can help to obfuscate the query log, thereby using different rewritten queries for the same clear-text query. Fortunately, all rewritten queries have the same performance if a range is chosen that does not include any *false positives*.

C. NEP/POP: Frequency Attacks and Correlations

Table IV shows a non equality-preserving POP dictionary (i.e., NEP/POP). Such NEP/POP dictionaries can be used in order to protect data against frequency attacks. With NEP/POP each value can be associated to several codes. As shown in Table IV, it is even possible that the same value has multiple codes within a single run. For instance, Elton John has two



(a) Skew on Values (b) Uniform on Codes (c) Piecewise Uniform

Fig. 2. Distorting a Frequency Distribution

codes in Run 2 in Table IV. For efficient query processing, the only condition that must hold is: If $c_1 < c_2$ and c_1, c_2 are two codes of the same run representing values v_1, v_2, then $v_1 \leq v_2$. ($v_1 < v_2$ is not required.) Other than that, the basic principles of NEP/POP are the same as for EP/POP. In particular, queries and updates are rewritten in (almost) the same way as described in Section IV.

To protect data against a frequency attack, the frequencies must be *distorted*. This idea is not new and has been explored independently from POP; e.g., [11], [12]. Fortunately, this idea can be combined nicely with the POP principle and that is the basis of the NEP/POP technique (or probabilistic POP). Therefore, the remainder of this section describes this frequency distortion idea in more detail.

Figure 2a shows the frequency distribution of *values* in the original (unencrypted) database. For instance, Figure 2a could show for each customer how many orders that customer has placed. Figure 2b shows a *secure* frequency distribution of *codes* in an encrypted database. In Figure 2b, the same number of orders are associated to each *Customer code*. As a result, it is impossible for an attacker to deduce which customer corresponds to which code, even if the attacker knows the distribution of Figure 2a.

The frequency distribution of Figure 2b is achieved by assigning a different number of codes to each customer. More concretely, five times as many codes would be assigned to "Madonna" than to, say, "Elton John" if Madonna has placed five times as many orders. Accordingly, there would be five times as many "Madonna" tuples in the *Customer* table. In general, it can take a large number of codes in order to achieve such perfect uniform distributions of codes as shown in Figure 2b. Effectively, the *greatest common denominator*, g, of all frequencies must be taken and for each value f_v/g codes must be generated if f_v is the frequency of value v in the original frequency distribution. In many cases, $g = 1$ so that the size of the *Customer* table would be as big as the size of the *Order* table in the encrypted database. Obviously, that would result in a significant loss of performance.

The effects of a more practical approach are depicted in Figure 2c. Rather than taking the greatest common denominator of the frequencies of *all* values of the domain, a *base frequency* is defined for a set of values. If the first, say, 100 customers have each placed more than 50 orders, we could define 50 as a base frequency and allocate codes accordingly. That is, if the top customer has 120 orders, then two codes would be allocated for that customer and 50 (random) orders would be associated to each of these two codes. The remaining 20 orders would be associated to codes that are generated in the next iteration(s) of this approach. As a result, the frequency distribution is a stepwise function (as shown in Figure 2c); the number of steps are the number of iterations and the height of each step is the base frequency selected in each iteration.

Figure 2 shows how the frequency distribution of a single attribute can be distorted. An attacker, however, could also know about correlations. For instance, an attacker could know that the name "Peter" is more common in the city of New York than in Delhi. Correlations can be distorted in the same way as frequencies by flattening the *multi-attribute* frequency distribution. In this example, all combinations of *name, city* would be considered and the same flattening algorithm of Figure 2 would be applied to this *composite domain*.

IV. POP Query Processing

This section describes how POP works in detail. First, it is shown how queries can be rewritten in the context of a POP-encoded database (both deterministic and probabilistic POP). Second, the creation of an POP-encoded database is explained. Third, update processing is discussed. Fourth, indexing in POP encrypted databases is presented.

A. POP: Query Rewrite

As shown in Figure 1b, the Encryption Layer rewrites SQL queries so that they can be processed by the database system on the encrypted database in the cloud. Furthermore, the Encryption Layer post-processes query results returned by the database system. This post-processing involves decrypting the (POP) codes and it may involve post-filtering the results.

The discussion of Section III-B showed how to rewrite simple equality and range predicates in both EP/POP and NEP/POP schemes. In general, clear-text values in the queries are replaced by their corresponding codes. Furthermore, simple predicates may result in disjunctions depending on the number of runs affected by the predicates. There are a number of corner cases (e.g., dealing with values in queries that are not represented in the dictionary), but all these cases can be handled in a straight-forward way. Below we show how more complex queries can be processed. In particular, details are given for general comparisons, joins, `GROUP BY`, and `ORDER BY` (e.g., Top N).

1) General Comparisons: Let us assume that the *Order* table of Section II-B has two additional fields: (a) a *shipDate* that specifies when an order was delivered and (b) a *planDate* that specifies when the order was supposed to be delivered. Furthermore, assume that both of these dates are encrypted using POP and the same dictionary. Now, assume that a query asks for all orders that have a *shipDate* later than the *planDate*. Such a query can be processed using the following query on the encrypted database:

```
SELECT * FROM ORDER
WHERE shipDate.run != planDate.run
  OR shipDate.code-id > planDate.code-id;
```

This rewritten query works for both EP/POP and NEP/POP encryption and returns a *superset* of the matching tuples. In this superset, false positives can arise for order tuples for which the *shipDate* and *planDate* are encoded in different runs. Obviously, such false positives result in suboptimal performance. However, if the shipDate and planDate of each order is encoded in the same run, then no false positives arise. In general, we recommend to encrypt values in the same runs if the values are compared often to another; e.g., two dates of the same order. Fortunately, this restriction in the encryption scheme does not hurt privacy.

2) Joins: Functional joins along foreign key/key relationships are a particularly simple case because the encrypted database preserves integrity constraints, as shown in Section IV-B. As an example, a query that joins the *Customer* and *Order* tables can be rewritten as follows, independent of whether deterministic (EP/POP) or probabilistic (NEP/POP) is used:

```
SELECT * FROM Customer, Order
WHERE name.run = cust.run
  AND name.code-id = cust.code-id
```

Again, the database system does not care whether the join keys are encrypted or not. Also, the encrypted query result can be decoded with simple look-ups from the dictionary and without any more sophisticated post-processing.

Theta joins with arbitrary join predicates are not as simple. The join predicates of theta joins must be rewritten in the same way as any other general comparison. As shown in the previous subsection, general comparisons between two attributes of the same tuple can be carried out efficiently by making sure that both attributes are encrypted using codes of the same run. Unfortunately, this property cannot be ensured for joins because (logically) each tuple of the first table must be compared to all tuples of the second table. This is exactly the reason why POP shows poor performance for several TPC-H queries (Section VI). Fortunately, such non-equi join queries are rare in practice.

3) `GROUP BY` *Queries:* Like joins, rewriting `GROUP BY` queries is trivial. The `SELECT`, `FROM`, and `GROUP BY` clauses of the original and rewritten query are the same. Only the (non-join) predicates of the `WHERE` clause need to be rewritten as described at the beginning of this section. The `HAVING` clause of a `GROUP BY` query is also unchanged if an equality-preserving scheme is used; i.e., EP/POP. With EP/POP, furthermore, no post-processing of results by the Encryption Layer is needed; the Encryption Layer simply needs to decode the encrypted values of the query results.

An NEP/POP scheme requires post-processing by the Encryption Layer. The same `GROUP BY` key may be represented by several codes in the encrypted database so that post-processing involves additional grouping and aggregation; in the worst case, n times as many tuples are shipped from the DBMS to the Encryption Layer, with n the number of codes for each value. If the query involves a `HAVING` clause, then this `HAVING` clause must be executed in the Encryption Layer after decoding and post-grouping and post-aggregation. As a result, the extra communication cost to ship tuples that do not meet the `HAVING` clause can become unboundedly high, resulting in potentially poor performance.

An important assumption made throughout this section is that the metrics used in the aggregation functions of the `GROUP BY` query are not encrypted or encrypted using a homomorphism-preserving encryption scheme. If metrics are encrypted using POP only *count*, *min*, and *max* are supported as aggregate functions. If the metrics are encrypted and a complex aggregate function is used (e.g., *sum*), then grouping and aggregation cannot be pushed into the cloud. In this case, all the tuples that qualify the `WHERE` clause need to be shipped from the cloud to the Encryption Layer so that the client needs to carry out the grouping and aggregation as part of the query result post-processing.

4) `ORDER BY` *Queries:* If the `ORDER BY` clause of a query involves only one attribute and this attribute is encrypted using POP, then the best way to implement the `ORDER BY` query is as follows:

- For each run, issue a separate `ORDER BY` query that restricts to values of that run in its `WHERE` clause.
- Open a cursor for each of these queries and *merge* the results in the Encryption Layer.

Top N queries [13] can be processed in a similar way, thereby *stopping* the merge process once a sufficient number of results have been produced. In the worst case, $r + N$ tuples must be shipped from the cloud as opposed to N tuples in traditional (unencrypted) Top N query processing.

If the `ORDER BY` clause has several attributes, then we suggest to pre-order the results by the first attribute in the cloud and determine the final order with regard to the other attributes as part of post-processing in the Encryption Layer.

B. Creating POP Databases, Integrity Constraints

POP can be combined with any other encryption technique within a database. The only assumption made is that, if applied, POP is used to encrypt an *entire domain*. That is, the whole column of a table and columns of other tables that correspond to the same domain and may be part of comparison predicates of queries are encrypted using the same dictionary. If a key of a table is encrypted using POP, for instance, then all foreign keys to that table must be encrypted using POP

Name Dictionary

Value	Code
Elton John	$\langle 2,1\rangle$
Madonna	$\langle 2,7\rangle$
Madonna	$\langle 1,4\rangle$

Table Customer

name	info
$\langle 2,1\rangle$	...
$\langle 2,7\rangle$	...
$\langle 1,4\rangle$	...

Table Orders

id	cust
1	$\langle 2,1\rangle$
2	$\langle 2,7\rangle$
3	$\langle 2,7\rangle$
4	$\langle 1,4\rangle$

Fig. 3. NEP/POP: Keys and Foreign Keys

and the same dictionary so that joins can be computed in the encrypted database and so that the encrypted database can check for referential integrity constraints. In the running example, we would recommend to encrypt *Customer.name*, *Customer.city*, and *Order.cust* using POP. Other *info* fields which could potentially identify a customer and are subject to range predicates such as *age* should also be encrypted using POP. Other identifying fields such as *SSN* (i.e., social security numbers) or surrogates (e.g., *order-id*) for which range predicates are not reasonable can be encrypted using any traditional (strong) encryption technique. As mentioned in the previous subsection, metrics such as *price* or *volume* which are aggregated as part of `GROUP BY` queries, should not be encrypted at all or using a homomorphic encryption technique. In our experience, it is typically obvious which domains to encrypt and for which domains POP (either EP/POP or NEP/POP) is advantageous. The meta-data that records which attributes are encrypted in which way must be maintained by the Encryption Layer as part of the database that holds the data dictionary.

The dictionary in the Encryption Layer can be implemented using tables in a standard, off-the-shelf relational database system (e.g., PostgreSQL). For each domain that is encrypted using POP (EP/POP or NEP/POP), one table is created and stored in the database of the Encryption Layer. For example, the values of *Customer.name* and *Order.cust* would be encrypted using the same dictionary table. The values of *Customer.city* would be encrypted using a separate dictionary table. The schema of all dictionary tables is:

$\mathcal{D}$(value, run, code-id)

In order to rewrite queries and updates (i.e., encode cleartext values), such dictionary tables should have an index on *value*. In addition, a composite index on *run, code-id* should be maintained to decode query results returned by the cloud.

If an NEP/POP scheme is used for primary and foreign keys, then several tuples must be created in order to represent the same entity. This situation is depicted in Figure 3. "Madonna" is encrypted using two codes. Correspondingly, two tuples must be stored for "Madonna" in the *Customer* table. *Orders* can refer to either of these tuples. In Figure 3, for instance, Orders 2 and 3 refer to the first Madonna Customer tuple (Code $\langle 2,7\rangle$) and Order 4 refers to the second Madonna Customer tuple (Code $\langle 1,4\rangle$).

A dictionary for a particular domain is created using the "initial number of runs" as a parameter. Our privacy experiments (Section V) indicate that five is a good number for EP/POP encryption which protects against domain attacks and ten is a good number for NEP/POP encryption for frequency attacks. With EP/POP, a new value that has not been recorded in the dictionary yet is encrypted in the following way: First, a run is selected using a Uniform distribution. Second, an order-preserving code-id is created for that value within that run. If no such code-id exists, then a new run is created and a random code-id within that new run is created for the new value. This way, the number of runs can grow as new values are encrypted.

If an NEP/POP encryption scheme is used in order to protect against a frequency attack, then runs need to be selected more carefully. How to protect a domain against frequency attacks with an NEP/POP scheme is described in the next subsection.

C. Updates

Once a POP encrypted database has been created, it is straightforward to execute SQL update statements on it; i.e., inserts, deletes, and updates. The `WHERE` clauses of such update statements are rewritten in the same way as the `WHERE` clauses of `SELECT` statements. Values in the `SET` or `VALUES` clauses of `UPDATE` and `INSERT` statements must be encoded using the dictionary. Encoding such values is also straightforward if an EP/POP dictionary is used (domain attack): If the value already exists in the dictionary, then the code found in the dictionary is used. If not, then a new code must be generated. To this end, a run is selected randomly using a uniform distribution and a new code for the new value is created in that run as described in Section IV-B. Again, if it is not possible to generate a new code in that run (there is no gap for the new code at the right place), then a different run is selected or simply a new (empty) run is created.

For NEP/POP dictionaries (frequency attack), more care needs to be taken in order to make sure that no skew arises in the frequency distribution of the *codes* after executing a sequence of `UPDATE` statements. The exact details depend on the details of the frequency attack scenario and the UPDATE workload. In any case, maintaining a *distorted* frequency distribution in a POP encryption scheme is straightforward.

In the presence of deletes and updates, it may be possible to garbage collect codes that are stored in the dictionary and that are no longer used in the encrypted database. Such garbage collection can be implemented in many different ways; both reference counting and mark & sweep approaches are applicable. Studying such garbage collection techniques was beyond the scope of this work. For the experiments presented in this paper, no garbage collection was applied.

D. Indexes

One of the big advantages of a fine-grained encryption technique such as POP is that it can be implemented *on top of* an off-the-shelf database system without changing the database system in any way. In particular, POP can make use of indexes implemented in the RDBMS. In order to index the *Customer.name* column, for instance, a composite (B-tree) index on the *Customer.name.run, Customer.name.code-id* columns would be used in a POP encrypted database

and this index would help optimize rewritten queries in the encrypted database in the same way as the *Customer.name* index helps optimize queries in the non-encrypted database. As a result, database administrators can tune the physical design of POP encrypted databases in the same way as of non encrypted databases; another important advantage of a fine-grained encryption technique like POP.

V. Privacy Experiments and Results

The goal of POP is to achieve good privacy with affordable performance overhead. Obviously, POP will not have as good performance as a non-encrypted database. Furthermore, POP will not be as secure as a strongly encrypted database. The next section presents the results of performance experiments that assess the performance overheads of POP as compared to non-encrypted databases. This section presents the results of experiments that study the privacy achieved with POP for both the domain attacks and frequency attacks as compared to a theoretically *ideal* encryption scheme.

A. Measuring Privacy

There have been many proposals to quantify privacy as part of work on the anonymization of databases; e.g., K-Anonymity [14], L-Diversity [15], or T-Closeness [16]. However, these metrics are not applicable to transactional databases that must return exact results. In order to quantify privacy for transactional databases, we use a metric that has been proposed and used in [17], [18]. The idea is to define the *privacy of a value* by the probability that that value is encoded by a specific code. More formally, we define the privacy of a value (e.g., the name of a specific customer, say, Madonna) by the maximum probability that this value belongs to a specific code. That is, the privacy function is defined as follows:

$$priv(v) = max_{c \in DB} P(v = c|\mathcal{K}) \tag{1}$$

Here, v denotes the value to be protected (e.g., "Madonna"), DB is the encrypted database in the cloud, and c denotes a code in DB (e.g., $\langle 1, 2 \rangle$). $\mathcal{K}$ represents the knowledge of the attacker; i.e., the encrypted database, the domain of values and their frequencies for the frequency attack. $P(v = c|\mathcal{K})$ denotes the probability that Code c belongs to Value v under the condition that the attacker has Knowledge $\mathcal{K}$.

This *priv(v)* metric is best explained by example. If $|dom(v)|$ is the size of the domain of values of v and a deterministic encryption scheme is used (i.e., every value is represented by exactly one code in DB), then ideally (with the strongest possible encryption) the following condition holds:

$$priv(v) = \frac{1}{|dom(v)|} \tag{2}$$

That is, each code corresponds to each value with the same probability. Ideal means that the codes have the highest possible entropy [19]. In such an ideal equality-preserving encryption scheme, the *priv(v)* function does not depend on $\mathcal{K}$ which means that the attacker's knowledge is useless; i.e., $P(v = c) = P(v = c|\mathcal{K})$.

For a frequency attack, an ideal encryption scheme can be characterized as follows:

$$priv(v) = \frac{freq(v)}{\sum_{c \in DB} freq(c)} \tag{3}$$

Here, $freq(v)$ corresponds to the frequency of Value v in the original database. Again, this equation is independent of $\mathcal{K}$. In the frequency attack scenario, $priv(v)$ denotes the probability to get *one* code right; the probability to get *all* codes of a value right is of course much smaller. Correspondingly, more frequent values have higher $priv(v)$ values (i.e., less privacy) than less frequent values.

Obviously, POP is not ideal because it maintains partial order in the codes and, therefore, reduces the entropy of the codes. To compute the $priv(v)$ function for POP involves some combinatorics; the combinatorics we used for our experiments are described in [20]. Essentially, $P(v = c|\mathcal{K})$ is computed by computing the number of possible worlds that the dictionary can have under the condition that $v = c$ divided by the total number of possible worlds that the (hidden) dictionary can have given the knowledge of the attacker, $\mathcal{K}$. Similar computations and analyses are carried out in the context of probabilistic databases [21]. All experiments reported in this section were carried out using MatLab. We will publish the exact formulae and MatLab code on PubZone.

To give a simple and extreme example, the code for Beatles can be either $\langle 1, 4 \rangle$ or $\langle 2, 1 \rangle$ in the EP/POP dictionary of Table III because the Beatles are encoded either in the first or in the second run and in each case, the Beatles have the lowest code-id in that run. As a result, $priv(Beatles) = 0.5$ in this simple example. In general, the $priv(v)$ function depends on the size of the domain (i.e., $|dom(v)|$) and the number of runs for domain attacks. For frequency attacks, the two main parameters are the number of runs and the *width* of each step in the distorted frequency distribution depicted in Figure 2c.

B. Domain Attack(EP/POP)

Figure 4 plots the $priv(v)$ function for 2 and 5 runs for a domain of 120 different values and an EP/POP approach (domain attack). As a baseline, Figure 4 also shows the *ideal priv(v)* function that is constant, $\frac{1}{120}$, for all 120 values of the domain (Equation 2 of Section V-A).

Two observations can be made from Figure 4. First, the EP/POP curve for five runs is fairly close to *ideal*. Even with two runs, EP/POP provides good privacy for most values. (Obviously, how much privacy is good enough depends on the application.)

The second observation is that privacy is worst for the *extreme* values of the domain; e.g., customers whose name start with an "A" or "Z". Fortunately, the curves are quite steep for those extreme values so that only a few extreme values are affected: four to five values at both ends and this number does not depend on the domain size. Furthermore, it is fortunate that there is a fix to improve the privacy of those values, too: simply insert a few dummy customers with names "AAA1", "AAA2", etc. and "ZZZ1", "ZZZ2", etc. A

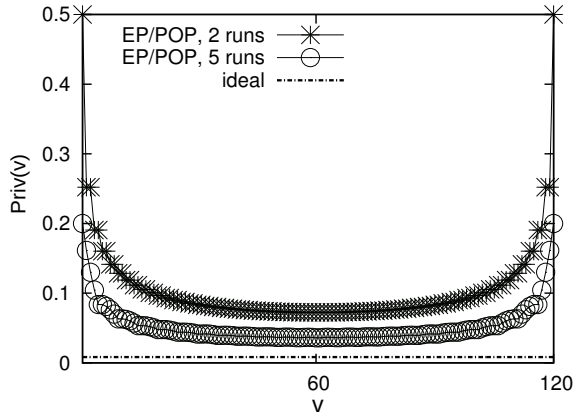


Fig. 4. EP/POP Priv(v): Vary #Runs, Domain Size=120

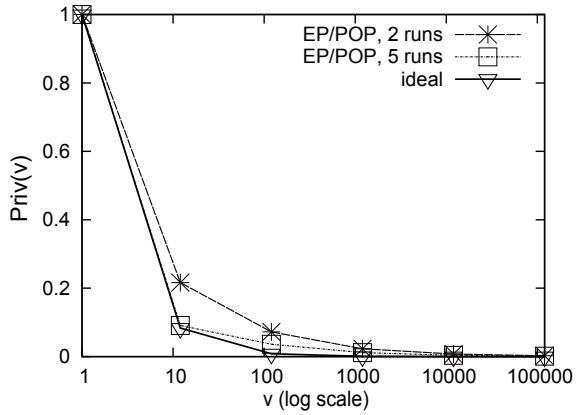


Fig. 5. EP/POP Priv(v): Vary Domain Size, #Runs, Single v

similar approach has been studied in more detail in [22]. With this *padding approach*, the real customers with "A" and "Z" names are protected in the same way as other customers whose names are in the middle of the domain. Even if the attacker knows that, say, four customers are fake at both ends of the name spectrum, that information does not help the attacker to decrypt real "A" customers with high probability. Of course, it is important to make sure that the attacker cannot identify the exact codes of the fake customers; as a result, it may be necessary to generate fake orders for those customers, too. Furthermore, those fake customers need to be factored out in aggregate queries; e.g., counting the number of customers or summing up the balance of the accounts of all customers.

Figure 5 shows how privacy improves with the domain size. Obviously, if the domain is small (only a few values) and the attacker has precise knowledge of the domain, then the attacker has a high chance of guessing the right value for each code. In the special case of a domain size of 1, the attacker can certainly assign the right value to the only code in the database, independent of the number of runs. As shown in Figure 5, privacy improves sharply with a growing domain size. With a domain size of 10, the probability of guessing the right value for a code is already in the order of 0.1; for large domains of 100,000 values, the probability is below 2×10^{-3}.

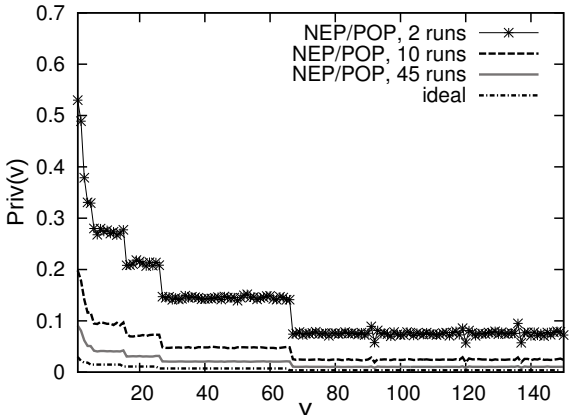


Fig. 6. NEP/POP Priv(v): Vary #Runs, Top 150 Values

C. Frequency Attack

Figure 6 shows the results for the more general frequency attack. Specifically, Figure 6 plots the $priv(v)$ function for the Top 150 most frequent values of a domain; these are the most vulnerable values. In this experiment, a Zipf distribution was used for the original frequency distribution of the values (same as in Figure 2a). The ordering of the values was random. That is, we repeated the experiment 1000 times with different databases; in some databases, the most frequent values were *small* or *large* (i.e., extreme with regard to the ordering of the domain); in others, they were in the middle. As a baseline, Figure 6 shows the *priv(v)* curve for an ideal encryption scheme as defined in Equation (3) of Section V-A.

The results of Figure 6 are approximations. The exact formulae are complex and MatLab crashes even for small parameter settings if we try to compute the $priv(v)$ function exactly. The results shown in Figure 6 represent an *upper bound* of the $priv(v)$ function for NEP/POP; that is, the results are conservative and the real privacy of NEP/POP is better than what is shown in Figure 6. Furthermore, we had to limit the experiments to a domain of 150 values because of computational constraints. As shown in the previous section, privacy improves as the domain gets larger. For smaller domains, the privacy degrades, but not dramatically. For example, with a domain of 50 values, the maximum *priv(v)* was 0.3 with 10 runs as opposed to 0.2 with a domain of 150 values.

Figure 6 shows that privacy improves with the number of runs. With only two runs, privacy is poor if the most frequent value is an extreme value of the domain (e.g., the lowest or highest value of the domain). With 10 runs, however, NEP/POP ensures good privacy in all scenarios. With 45 runs, NEP/POP is close to the *ideal* privacy that can be achieved in a frequency attack scenario.

For readability, Figure 6 does not show error bars. The experiments, however, demonstrate that the standard deviation in this experiment is large for two runs and drops radically with a growing number of runs. Again, the large variance comes from situations in which frequent values are extreme values in the domain vs. situations in which the extreme

values of the domain are infrequent. The variance could be decreased dramatically by the *padding approach* described in the previous subsection. Doing so would also significantly improve the average privacy because it improves the bad cases.

VI. Performance Experiments

This section assesses the performance overhead of POP using the TPC-H benchmark. The performance of POP is compared to two baselines: First, a database that is not encrypted, referred to as *plain*. Obviously, POP performs worse than such a plain database. Second, a database that is strongly encrypted. This approach is referred to as *NOP*. As compared to NOP, POP should perform better if the query involves range, `LIKE` predicates, sorting, or ranking (e.g., TOP N). For brevity, only the results for the more general general frequency attack with probabilistic POP and NOP encryption are given. Obviously, the performance of deterministic encryption is better, but overall the same effects can be observed. First, we present the response times of all the queries and update functions of the TPC-H benchmark. Then, we show the sensitivity of the performance of POP to the number of runs.

A. Benchmark Environment

The experiments were carried out using a client-server environment as shown in Figure 1b. That is, a client process issued queries and ran the Encryption Layer; these queries were processed by the database server process on the encrypted database. All experiments were executed on a machine with two quad-core AMD 2.4 GHz CPUs and 16 GB of main memory. This machine ran Ubuntu Linux 10.04. The machine ran both the client and the server. In other words, the client was thick and client/server communication was fast. This set-up favored the strong encryption scheme (NOP) because NOP was not penalized as much for its inability to process certain queries at the server on the encrypted data. PostgreSQL Version 9 was used as a database system for both the client (i.e., the data dictionary) and the server (i.e., the encrypted database). Two separate PostgreSQL database instances were used for the client and the server. Transfer of query results between the server and the client was effected by storing the results in a file in the file system and then loading that result file into a temporary table of the client's PostgreSQL database instance. Aggregate queries with NEP/POP encryption, for instance, were processed in the following way (Section IV-A): First, the rewritten query was executed by the server on the encrypted database. Second, the (encrypted) results of the rewritten query were stored in the file system and loaded into the client's database. Third, the final (clear text and post-aggregated) query results were computed using the client's database system and a remainder query. In this post-processing step the results were also decrypted by joining the encrypted query results with the dictionary tables.

In all experiments reported in this section, the queries and update functions of the TPC-H benchmark were used. The benchmark database was generated with the *dbgen* tool provided by the TPC organization. We used a database with a scaling factor of 10 (i.e., 10 GB). POP was applied in a straight-forward way as described in Section IV-B. Metrics used in aggregate functions (e.g., tax or volume in orders) were not encrypted. Furthermore, surrogates (e.g., order numbers) were not encrypted. All other attributes such as names, dates, etc. were encrypted using POP. In the performance experiments presented in this paper, we used POP with five runs for domain attacks and POP with ten runs for frequency attacks.

In all experiments, we measured the time to process the (rewritten) query on the encrypted database on the server (i.e., cloud), the time to ship the results from the server to the client and load them into the client's database, and the time to decrypt and post-process the results at the client. As will become clear, the time to process the queries in the cloud dominated the total cost of query processing in almost all cases of the TPC-H benchmark.

B. TPC-H Benchmark

Figure 7 shows the running times of the queries and update functions of the TPC-H benchmark for the frequency attack scenario that requires probabilistic encryption. The plain database is the original TPC-H database (as generated by the *dbgen* tool) and offers no protection against attackers. The NOP and POP databases, in contrast, applied the frequency and correlation distortion techniques described in Section III-C. Implementing this scheme involved a significant increase in the size of the dimension tables of the TPC-H benchmark. For instance, the *Customer* table had seven times as many entries for the NOP and POP database as in the plain database in this experiment. Consequently, NOP and POP had higher response times in almost all cases.

Comparing POP to plain, Figure 7 shows that the overheads of POP are moderate for most queries (i.e., Q1, Q3, Q5, Q6, Q11, Q12, Q13, Q14, Q16, Q17, and Q19). These queries could be executed inside the database system using POP and only moderate post-processing (i.e., decrypting the results) was required in the Encryption Layer. For some queries (i.e., Q2, Q4, Q8, Q15, and Q18), POP was even faster than plain. This phenomenon can be explained as POP operates on integers (i.e., codes) which can be processed faster than varchars which are used in the plain database to represent names, cities, etc. Figure 7, however, also shows that for certain kinds of queries, plain dramatically outperforms POP; i.e., Q7, Q9, Q10, Q20, and Q21. These TPC-H queries involve SQL access patterns that cannot be processed with POP unless the data is decrypted; most notably, these queries involve non-equi joins or sub-selects with correlated non-equi predicates. As a result, POP requires to send large portions of the database to the Encryption Layer, decrypting the data there, and processing the queries at the client. Processing such queries would not be feasible if the client were, say, an iPhone.

Comparing POP to NOP, it can be seen that POP outperforms NOP significantly for most queries. In the extreme case (Q3), POP outperforms NOP by an order of magnitude. The reason is simple: NOP requires to rewrite range and `LIKE` predicates into `IN` predicates that contain all possible

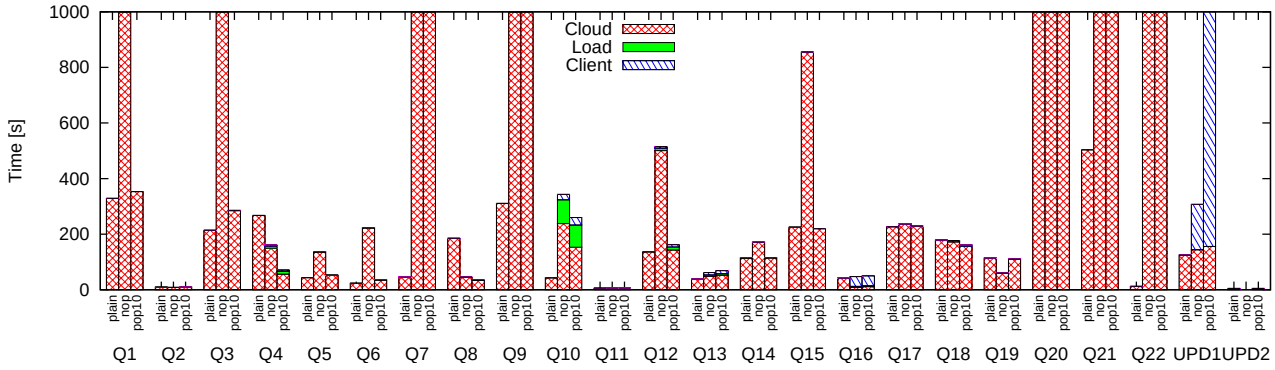


Fig. 7. TPC-H Running Time (s): Plain vs. Trad. Encryption (NEP/NOP) vs. NEP/POP (10 runs)

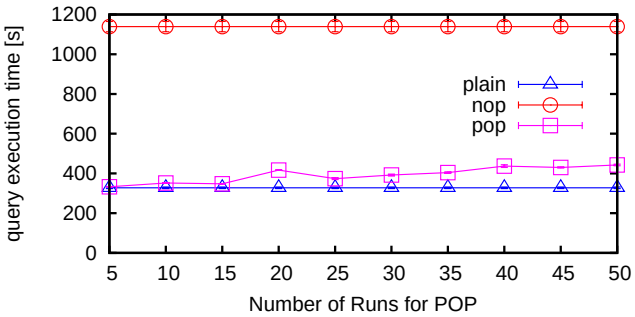


Fig. 8. Time (s): Vary #Runs for EP/POP, Q1

codes (Section III-A). To protect against frequency attacks, a large number of codes need to be generated and, thus, the performance of these `IN` predicates deteriorates rapidly. We also tried an NOP variant that post-filters all results in the Encryption Layer (no `IN` predicate), but that variant showed even worse performance so we do not present the results. Furthermore, NOP cannot handle non-equi join queries or any query that POP cannot handle well either.

Figure 7 also shows the performance of the two update functions of the TPC-H benchmark. UPD1 inserts new orders, thereby also inserting new customers. UPD2 deletes orders. So far, our implementation has not been optimized for updates. Correspondingly, POP (and NOP) show poor performance for UPD1. We plan to improve our implementation of updates as part of a future release; we believe that with some engineering effort, the performance of POP can be made comparable to that of plain for updates.

C. Vary Number of Runs

Obviously, the number of runs is one of the critical parameters that impacts both the security (i.e., entropy) and performance of POP. Section V showed that POP has good entropy starting with about 10 runs. Figure 8 shows the performance of EP/POP with a varying number of runs for Query 1. It can be seen that the overhead grows roughly linearly with the number of runs. Query 1 of the TPC-H benchmark has a range predicate on shipping dates of orders and POP involves rewriting that predicate into a disjunction. For Query 1, the range is fairly large so that it involves codes from all runs; as a result, there are as many predicates in the disjunction as there are runs (up to 50 runs in Figure 8). Even for 50 runs, however, the overhead of POP was affordable and the performance of POP did not deteriorate with the number of runs. We ran the whole benchmark with a varying number of runs (not shown for brevity). In general, the overheads of POP were moderate in all our experiments, independent of the number of runs. As shown in Figure 7, POP only showed poor performance if the query involved non-equi joins.

VII. Related Work

Due to its importance, there has been a great deal of work on encryption and preserving privacy. In particular, privacy-preserving techniques have been applied in a variety of different contexts; e.g., information retrieval [23], data anonymization [14], and privacy-preserving data mining [24].

The specific task of ensuring privacy for database services in the cloud has also been studied extensively in the past [1]. In general, there are four approaches to address this problem. The first approach involves additional data structures in order to facilitate query processing on encrypted data in the cloud. Sion, for instance, proposes to provide query-specific bloom filters that allow the cloud database system to pre-filter the data; a final post-filtering step is then carried out in the Encryption Layer [5], [25]. An alternative, related approach was proposed by Boldyreva et al. [26], [27].

The second approach to effect privacy in the cloud is to partition the data and encrypt data in the granularity of partitions (rather than individual values) [28]. Data is shipped from the cloud to the Encryption Layer in the granularity of partitions and these partitions are post-filtered in the Encryption Layer. [29] show how such partitions can be indexed.

The third approach is based on order-preserving encryption. [9] and [30] fall into this category. None of these techniques are applicable in the presence of a domain or frequency attack, as detailed in Section III-A. Nevertheless, POP falls into this class of techniques.

The fourth approach is to find homomorphic encryption techniques. The goal is to guarantee algebraic properties

that make query processing on the encrypted data efficient. Homomorphic encryption has been first proposed in the Late Seventies [3] and has recently received revived attention [10], [31]. Unfortunately, homomorphic encryption is still too slow for large-scale data processing as studied in this work (e.g., TPC-H queries on GBytes of data). Even if it were practical, the ring property supported by homomorphic encryption is not sufficient to process range predicates and sorting efficiently. Likewise, any other *strong* encryption technique that has been proposed by the security community so far (e.g., [4]) does not support the special SQL access patterns exploited by POP and does, therefore, not meet the performance requirements of most DaaS applications.

POP could be one of the "Onion layers of encryption" as suggested in [32] but at the same time POP provides its own configuration parameter to tune the level of security. Recently, a number of products have been launched that support DB encryption; examples are Safenet, RSA BSAFE, Vormetrics, or Voltage. No entropy or performance results of these systems have been published. Typically, these systems are based on RSA, AES, or DES encryption, but the exact properties of the used encryption scheme have not been published. Furthermore, it seems that most of these systems were not designed for frequency attacks so that their encryption scheme is too weak. Built-in encryption as supported in commercial database systems such as Oracle 10g [33] is also not a viable solution because it does not protect against attacks from system administrators.

VIII. Conclusion

This paper presented a new encryption scheme for databases called POP. POP enables the processing of a wide range of SQL queries without decrypting the data. POP is an important building block for DaaS (databases as a service) in public and private clouds because it protects dimension data (e.g., names and dates) against malicious system administrators and men in the middle. POP can be combined well with other techniques; in particular, with techniques to distort frequencies and correlations. Security and performance experiments showed that POP ensures good privacy (i.e., high entropy) and good performance results for a wide range of queries.

There are a number of research challenges that we would like to address as part of future work. First, we would like to devise bulkloading techniques in order to efficiently encrypt an existing database. Second, we would like to improve the performance of SQL `INSERT` statements and study update-intensive workloads (i.e., TPC-C). Finally, we would like to study techniques to protect privacy against query log attacks. Such techniques involve encrypting (or fuzzifying) *queries* and we believe that POP can help to support techniques to do this kind of encryption of queries (rather than data).

Acknowledgements: We would like to thank Daniela Brauckhoff, Roger Hüsser, Nico Voutsis, and Stephan Murer from CreditSuisse for helping to implement POP inside Credit Suisse. Furthermore, we would like to thank Srdjan Capkun and Karame Ghassan from the ETH Security Group for many helpful comments. This work was funded by the ETH Enterprise Computing Center.

References

[1] R. Chow, P. Golle, M. Jakobsson, E. Shi, J. Staddon, R. Masuoka, and J. Molina, "Controlling data in the cloud: outsourcing computation without outsourcing control," in *CCSW*, 2009.
[2] P. Wahl and P. Needham, "Oracle Advanced Security with Oracle Database 11g Release 2," 2010.
[3] R. Rivest, L. Adleman, and M. Dertouzos, "On data banks and privacy homomorphisms," 1978.
[4] T. El Gamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," in *CRYPTO*, 1985.
[5] R. Sion, "Secure data outsourcing," in *VLDB*, 2007.
[6] E. Damiani, S. D. C. Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati, "Balancing confidentiality and efficiency in untrusted relational DBMSs," in *CCS*, 2003.
[7] Y. Hong, X. He, J. Vaidya, N. Adam, and V. Atluri, "Effective anonymization of query logs," in *CIKM*, 2009.
[8] M. Kantarcioglu and C. Clifton, "Security issues in querying encrypted data," in *Data and Applications Security XIX*, 2005.
[9] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu, "Order preserving encryption for numeric data," in *SIGMOD*, 2004.
[10] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Stanford University, 2009.
[11] S. Chawla, C. Dwork, F. Mcsherry, A. Smith, and L. J. Stockmeyer, "Toward privacy in public databases," in *In TCC*, 2005.
[12] R. Gavison, "Privacy and the limits of the law," *Computers, Ethics, and Social Values*, 1995.
[13] M. J. Carey and D. Kossmann, "On saying "Enough already!" in SQL," in *SIGMOD*, 1997.
[14] L. Sweeney, "k-anonymity: a model for protecting privacy," *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 2002.
[15] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam, "l-diversity: Privacy beyond k-anonymity," in *ICDE*, 2006.
[16] N. Li, T. Li, and S. Venkatasubramanian, "t-closeness: Privacy beyond k-anonymity and l-diversity," in *ICDE*, 2007.
[17] A. Evfimievski, J. Gehrke, and R. Srikant, "Limiting privacy breaches in privacy preserving data mining," in *PODS*, 2003.
[18] G. Miklau and D. Suciu, "A formal analysis of information disclosure in data exchange," in *SIGMOD*, 2004.
[19] E. T. Jaynes, *Probability Theory: The Logic of Science (Vol 1)*. Cambridge University Press, 2003.
[20] M. Michael and U. Eli, *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.
[21] D. Suciu and N. Dalvi, "Foundations of probabilistic answers to queries," in *SIGMOD*, 2005.
[22] P. Tendick and N. Matloff, "A modified random perturbation method for database security," *ACM Trans. Database Syst.*, 1994.
[23] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan, "Private information retrieval," *J. ACM*, 1998.
[24] R. Agrawal and R. Srikant, "Privacy-preserving data mining," *SIGMOD Rec.*, 2000.
[25] R. Sion, "Towards secure data outsourcing," in *Handbook of Database Security*. Springer US, 2008.
[26] M. Bellare, A. Boldyreva, and A. O'Neill, "Deterministic and efficiently searchable encryption," in *CRYPTO*, 2007.
[27] G. Amanatidis, A. Boldyreva, and A. O'Neill, "Provably-secure schemes for basic query support in outsourced databases," in *IFIP WG 11.3*, 2007.
[28] H. Hacigümüş, B. Iyer, C. Li, and S. Mehrotra, "Executing SQL over encrypted data in the database-service-provider model," in *SIGMOD*, 2002.
[29] B. Hore, S. Mehrotra, and G. Tsudik, "A privacy-preserving index for range queries," in *VLDB*, 2004.
[30] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill, "Order-preserving symmetric encryption," in *EUROCRYPT*, 2009.
[31] N. Smart and F. Vercauteren, "Fully homomorphic encryption with relatively small key and ciphertext sizes," Cryptology ePrint Archive, Report 2009/571, 2009.
[32] C. Curino, E. P. C. Jones, R. A. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich, "RelationalCloud: a Database Service for the cloud," in *CIDR*, 2011.
[33] A. Nanda, "Transparent Data Encryption," 2005.