Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Technical Report Nr. 782

Systems Group, Department of Computer Science, ETH Zurich

POP: A new Encryption Scheme for Dynamic Databases

by

Tahmineh Sanamrad, ETH Zurich
Lucas Braun, ETH Zurich
Andreas Marfurt, ETH Zurich
Donald Kossmann, ETH Zurich
Ramarathnam Venkatesan, Microsoft Research

March 28, 2013

inf | Informatik
Computer Science

Abstract

This technical report explores a new 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 attacker scenarios. We present security results for four such adversary models. Furthermore, we present the results of performance experiments conducted using the TPC-H benchmark and an off-the-shelf relational database system in order to study the performance overheads of POP.

1 Introduction

One of the big research challenges is to achieve both high performance and good security for database systems. In particular, it is still difficult to protect data against the attacks of *curious and honest* administrators who have root privileges on database server machines. These administrators might have incentives to sniff into the database and sell confidential data to competitors, agencies, or other parties who might be interested in the data. Confronted with such attacks, most organizations face the following dilemma: They can choose between either high security with strong encryption and low performance or poor security with no or weak encryption and high performance.

1.1 State of the Art

Because of its importance, there has been a series of work to make database systems both secure and performant. Full homomorphic encryption is probably the most prominent line of work [9, 17, 18]. The goal of homomorphic encryption is to strongly encrypt the data and process the data without decrypting it. The development of secure hardware is another line of work to protect confidential data against internal attackers. Here, the goal is to develop tamper-proof co-processors that execute expressions on strongly encrypted data. TrustedDB [3] and Cipherbase [2] are two projects that follow this line of work. Unfortunately, neither full homomorphic encryption nor secure hardware are fully practical today and none of these techniques have had any impact on database systems yet.

Most commercial database products support strong encryption using AES; e.g., Oracle [14] and Microsoft SQL Server [12]. Unfortunately, they only support encryption for *data at rest* at the disk level. That is, the data is encrypted on disk so that stolen or lost disks are no threat. However, the database system decrypts the data before processing it so that anybody with direct access to the database server machines can sniff into the data. Furthermore, these products provide no protection against attacks from database administrators because admins still have the right to execute "`SELECT *`" queries, thereby seeing all plaintext results.

To fill the gap between no security/high performance and high security/low performance, there have been a number of recent proposals for *weak* encryption techniques that support query processing without decrypting the data—just as homomorphic encryption. Examples are [1, 8, 10]. Boldyreva et al. [6, 7] have achieved the state-of-the-art results in this regard. They proposed several order-preserving encryption schemes which give different kinds of guarantees against certain kinds of attacks. As shown in [1] and confirmed in our own experiments, good performance can be achieved with such order-preserving encryption techniques, but unfortunately, even their strongest encryption method, is perceived to be not good enough to protect data sufficiently in many applications that involve confidential data.

Independent and orthogonally to work on encryption techniques, new system designs with fine-grained (i.e., column-level) encryption have been explored. These designs exploit the observation that not all data is confidential so that different columns can be encrypted in different ways. For instance, columns with public data (e.g., country names) need not be encrypted at all. Columns with highly confidential data (e.g., account balances or diagnoses in medical records) are encrypted using strong encryption techniques

such as AES. Other data may be encrypted using weaker encryption schemes. Both TrustedDB and Cipherbase (mentioned above) make use of such a fine-grained encryption scheme. CryptDB [15] is another example of a system that exploits such a design. CryptDB, in particular, has the ability to adapt to the query workload and adjust the encryption scheme based on the workload requirements: CryptDB features an onion-scheme that initially encrypts all columns in the strongest possible way. If needed for a specific query, CryptDB peels off a layer of the onion for a column, thereby degrading to a weaker encryption to process the query. While this approach helps to automatically find the strongest possible encryption scheme that can meet the performance requirements, this approach is only as good as the underlying encryption techniques used in each layer. For confidential data that is frequently used in queries, CryptDB will eventually degrade to a weak encryption scheme which might be undesirable from a confidentiality perspective.

1.2 Contributions

The goal of this work is to study a novel technique, called POP. The goal of POP is to improve the level of security of a weak encryption technique without significantly hurting performance. Just like TrustedDB, CryptDB, and Cipherbase, POP assumes a fine-grained, column-level encryption model. The key idea of POP is to partition each column that contains confidential data into so-called runs. As part of this partitioning, each value of a column is randomly assigned to a run and the attacker has no knowledge of which value is encrypted in which run. In a second step, each run is encrypted using a weak encryption technique which supports efficient query processing on that run.

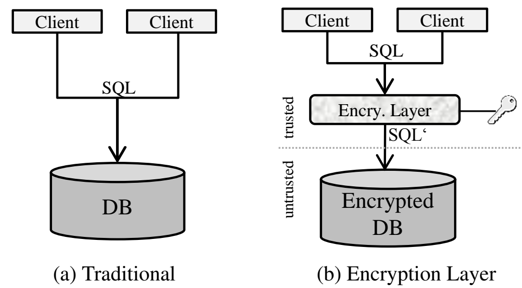
Why does POP achieve good performance in most cases? The answer is simple: Most database algorithms (e.g., merge sorts or hash joins) are based on partitioning the data. So, the additional partitioning imposed by POP does not hurt too much, albeit it is not for free.

The tricky question is how much does POP increase security. As will be shown, confidentiality grows with the number of runs. In the extreme case of one run, POP has the same security (and performance) as its underlying weak encryption scheme. In the other extreme in which each value is stored in a separate partition, POP has the same security (and performance) as a strong encryption scheme. The beauty of POP is that security improves dramatically even for a small number of runs. Specifically, we have defined and proven the security of POP-enhanced encryption for different adversary models. These models capture how well an attacker can narrow down the set of possible plaintext values for a given ciphertext. Furthermore, they capture whether the ciphers of two ordered sequences of plaintext values can be distinguished. Finally, we analyze the advantage of attackers who know several plaintext / ciphertext pairs (i.e., the *known plaintext attack*). Some of these adversary models have been proven before in the so-called *committed database model* [7]; in that model, no updates to the database are allowed once the database has been encrypted. We show with the help of POP that we can make the same (and even stronger) security claims for dynamic models of databases.

Another crucial advantage of POP is that its implementation requires no changes to the database system. That is, POP can be fully implemented as part of a security middleware *on top of* the database system. This makes POP a good candidate for adoption of users of public clouds in which users do not trust the provider or its employees. For instance, POP is an ideal candidate to encrypt address books or the state of other mobile applications, store it in the cloud, process all queries in the cloud, and only decrypt the results in a smart phone, tablet, or client machine.

In summary, this paper presents the following three contributions:

- POP principle and how the security middleware rewrites and post-processes queries to be executed on POP-encrypted databases. This paper describes three POP variants: stateful POP with dictionary encryption (POP-DE), determininistic POP (Det-POP), and probabilistic POP (Prob-POP).
- Study the security properties of POP and the three POP variants for different adversary models.



(a) Traditional (b) Encryption Layer

Figure 1: Extended Client-server Database Architecture

- Results of performance experiments with the TPC-H benchmark. It is shown that the overheads of POP are affordable in most cases. We focus on OLAP queries in this paper. We plan to study OLTP workloads (e.g., TPC-C) as part of future work. In fact, one particular advantage of POP is that it supports updates without sacrificing any security.

1.3 Overview

The remainder of this paper is organized as follows: Section 2 states the most important assumptions that we made in this work. Section 3 describes POP and three POP variants. Section 4 shows how to rewrite queries for POP-encrypted databases. Section 5 presents the most important security results that we were able to prove for POP. Section 6 presents the results of performance experiments. Section 7 contains conclusions and possible avenues for future work.

2 Client-server Architecture

Figure 1a shows the traditional client-server architecture of running applications on top of a database system. The 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 virtually all related work on client-side database security; e.g., [2,8,15,16]. In this architecture, both the applications and the database system are unchanged. That is, the applications (or 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, DB2, MySQL, SQL Server / SQL Azure, or Oracle).

In the architecture of Figure 1b, confidentiality is implemented as part of an *Encryption Layer* in the following way. First, the Encryption Layer encrypts each value of the database using an encryption function and a *key*. For instance, the clear text value *Elton John* for a customer name could be represented by the ciphers 5 and 14 in the encrypted database. Depending on the encryption function, this *key* can take one of several forms: It can be a 128-bit integer or it can be an entire *data dictionary* that keeps a mapping of plaintext / ciphertext values. The *key* is kept securely at the client and is only visible to the Encryption Layer.

The second task of the Encryption Layer is to rewrite SQL queries and update statements. For instance, the query `SELECT * FROM Customer WHERE name = ``Elton John''` could be rewritten into `SELECT * FROM Customer WHERE name IN (5, 14)`. Finally, the Encryption Layer decrypts results returned from the database system. As a result of this

step, an application program (or user) only sees clear text values and the Encryption Layer encapsulates encryption and makes security issues transparent to the application developer.

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. It should be possible to deploy the Encryption Layer on a smart phone or laptop. Furthermore, (virtually) no administration is required for the Encryption Layer: The only requirement is that the key must not be lost. If the key involves a whole dictionary, tools such as SQLite can be used to store such a dictionary at the client.

Trusted means that only the owner of the data has access to the key and the decrypted results returned by the Encryption Layer. For users of public clouds who try to protect their data against external adversaries, the Encryption Layer is deployed on machines inside of the users' organization. In a private cloud with potential internal attackers, the Encryption Layer could be deployed on machines in a separate security domain which is maintained by a few highly trusted administrators.

There are many ways how information can be leaked in the architecture of Figure 1b. The focus of this paper is on attacks of "curious and honest" administrators who have access to the database server. Such an attacker tries to sniff into the database and derive information from the database: We do not provide any protection against attackers who try to disrupt the service of the database system. It is assumed that the attacker can see the whole database and has possibly additional knowledge of the schema and value distributions and correlations of the domains. This paper does not address so-called *query log attacks* [11] 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. To be more concrete, we will formalize the adversary models studied in this paper in Section 5.

3 POP Encryption Scheme

This section presents the POP encryption scheme and three POP variants. The key idea of POP is to take an existing (weak) encryption method as a building block and to enhance its security (i.e., entropy) by applying it separately on different random partitions of the data. As part of this work, we apply POP to improve the security of order-preserving encryption method because these techniques enable the processing of a large variety of queries without decryption and good performance. Hence, the name POP for *partially order-preserving* and calling partitions *runs*. The POP principle, however, is more general and can be applied to any (weak) encryption method that has nice query processing properties. To set the stage and provide context, we start with a presentation of preliminaries and properties an encryption scheme might have.

3.1 Properties of Encryption Methods

The starting point for any POP-enhanced encryption scheme is an encryption method that has some property that can be exploited during query processing. For SQL databases, there are three properties that are useful: equality preserving, order preserving, and homomorphism preserving.

More formally, let a, b be clear text values of a domain that contains confidential information; e.g., *Madonna* and *Elton John* could be names of customers of a secret bank. Let enc be a function that maps a clear text value to a set of ciphers. That is, $enc(a)$ are the codes that are used to represent a in the encrypted database. For each domain that needs to be protected in the database, such an encryption function must be defined. Each enc function encapsulates its key so that we do not model the key as a separate parameter of an enc function here. It is assumed that the attacker does not know the enc function; the attacker may know the kind of function (e.g., AES), but the attacker does not know the *key* which is a (hidden) parameter of the enc function in our model.

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 1: Example enc Functions: Vary Encryption Properties

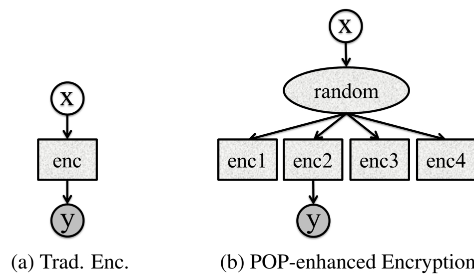
To be useful for SQL query and update processing, an enc function can have none, one, or several of the following properties [13]:

- *Equality-preserving (EP):* Each value corresponds to exactly one cipher; i.e., $a = b \implies \forall y_a \in enc(a), y_b \in enc(b) : y_a = y_b$.
- *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., $a < b \implies \forall y_a \in enc(a), y_b \in enc(b) : y_a < y_b$.
- *Homomorphism-preserving:* For any homomorphism, f: $enc(f(a, b)) = f(enc(a), enc(b))$.

The properties of the encryption function determine which kinds of queries can be executed on the encrypted database without decrypting the data 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. Homomorphism-preserving encryption is needed in order to compute aggregate functions on metrics (e.g., the sum of “price × volume” of orders). While all properties are relevant, this work focuses on equality and order-preserving encryption functions because metrics alone such as *tax rates*, *product prices*, and *order volume* needed as part of aggregations are typically not confidential; what is confidential is the relationship between a metric and a dimension (e.g., the volume of an order of a particular customer). To encrypt dimensions, looking at equality and order-preserving functions is sufficient.

Table 1 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) encryption functions. In the literature, EP is often called *deterministic encryption* whereas NEP is referred to as *probabilistic* or *random 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 in `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 6, both approaches can result in poor performance; with an `IN` predicate, for instance, the set of constants in the `IN` predicate can become large.

Just as an NOP scheme, NEP encryption can impact the performance of queries. The query `SELECT * FROM Customer WHERE name=``Madonna''` is rewritten into the following query using the NEP/NOP scheme of Table 1: `SELECT * FROM Customer WHERE name IN (8, 12)`. Again, it is likely that this is more expensive to evaluate than the original query on a non-encrypted database.



(a) Trad. Enc. (b) POP-enhanced Encryption

Figure 2: POP Principle

clear text	Run 1	Run 2	Code (⟨ run, code-id ⟩)
Beatles		6	$\langle 2, 6\rangle$
Elton John	9		$\langle 1, 9\rangle$
Madonna		8	$\langle 2, 8\rangle$
Metallica		2	$\langle 2, 2\rangle$
Nelly Furtado	1		$\langle 1, 1\rangle$
Tina Turner	5		$\langle 1, 5\rangle$

Table 2: POP Example: $U = 2$, Eq-pres., Part. Order-preserv.

3.2 POP: Basic Idea

Notation: Throughout the paper we will use the following notation (staying aligned with [6, 7]): $\mathcal{D}$ denotes the plaintext domain. M is the number of plaintext values, in other words $M = |\mathcal{D}|$. $\mathcal{R}$ denotes the ciphertext range. N is the number of ciphertext values, thus $N = |\mathcal{R}|$. U is the total number of runs.

Figure 2 illustrates the POP principle. Figure 2a shows the workings of a traditional encryption function. It receives a stream of clear text values and produces a stream of cipher text values. Figure 2b shows how POP composes this traditional scheme to become more secure and have a number of additional operational advantages (e.g., support for updates). Instead of a single enc function per domain, POP makes use of U enc functions per domain, $enc_1, enc_2, \ldots, enc_U$. These U functions possibly all have the same structure (e.g., using MOPE [7]) and just differ in the key that they use. Given a clear text value, x, POP randomly generates a number between 1 and U and encrypts using the corresponding enc function. That is,

$$y = enc_{random(x)}(x)$$

Depending on the *random* function and the structure of the enc functions, the composed encryption scheme of Figure 2b can have different properties. An equality-preserving encryption can be achieved if both the *random* and all enc functions are deterministic. Probabilistic encryption can be achieved on top of deterministic enc functions by providing an non-deterministic *random* function. Order-preserving encryption cannot be achieved with POP (unless for $U = 1$ or a *random* function that carries out range partitioning), but that is not the goal of POP; order-preserving encryption always has limited security and can be difficult to maintain in the presence of updates and we want to do better. However, POP can generate partial orders if all enc functions are order-preserving.

To be more concrete, Table 2 gives an example of a set of customer names encrypted with POP in an equality-preserving way, $U = 2$, and using order-preserving encryption functions with circular shifts. Again, each clear text value is assigned randomly to a specific enc function. We call the set of ciphers that have been generated by a specific enc function *runs*. In this example, Run 1 contains the ciphers for Elton John, Nelly Furtado, and Tina Turner. Run 2 contains the ciphers for Beatles, Madonna, and Metallica.

The attacker does not know the mapping of clear text values to runs. The attacker, however, sees the mapping of ciphers to runs because this mapping needs to be stored in the database because it is needed for query processing. For Table 2, we used order-preserving encryption functions with circular shifts as proposed in [7]; these functions model the domain as a "ring" and randomly select the beginning (i.e., *minimum*) from the whole domain. For instance, Nelly Furtado is the beginning of Run 1 and Metallica is the beginning of Run 2 in Table 2.

As shown in Section 4, POP allows to process a large class of queries efficiently without decrypting the data if the *random* and *enc* functions have certain properties. Section 4 also shows how updates can be processed, how integrity constraints can be maintained, and how POP databases can be created. To give an intuition 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 in the example of Table 2:

```
SELECT * FROM Customer
  WHERE (run = 1 AND
   (code-id < 1 OR code-id > 11)
 OR (run = 2 AND
   (code-id < 3 OR code-id > 7))
```

Actually, there are many ways to rewrite the query using POP and this is just one possible way.

The performance overhead of POP (as compared to no encryption) depends on the number of runs, U; the more runs, the higher the overhead. Likewise, the *security* depends on the number of runs; the more runs, the better security. We will study this performance / confidentiality trade-off in detail in Sections 5 and 6. In the extreme case, of $U = 1$, POP is the same as enc which is typically a weak, yet high performance encryption scheme. In the other extreme, $U = \infty$, POP is the same as *random* which corresponds to a strong yet low performance encryption scheme. The remainder of this section presents three different POP variants that differ in the enc functions used. Section 5 shows that each of these three variants can give different guarantees for our alternative adversary models.

3.3 Stateless POP

By stateless POP (as opposed to stateful POP), we refer to the fact that for these variants, our state is very small, i.e. consists of a set of keys. This set is very small (only some kilobytes) and is independent of the data volume.

3.3.1 Deterministic POP

The first and most basic POP variant that we studied, embeds the MOPE encryption technique suggested by [7] to encrypt the elements within a run (i.e., as enc functions). This technique features deterministic (i.e., equality-preserving) enc functions that are order-preserving and have a modular circular shift as described in the example of Table 2. An enc function is characterized by a 128-bit key and an offset that defines the beginning of the circular shift. As a *random* function, this Det-POP variant uses SHA-2 hashing. As shown in Section 5, this basic POP variant is sufficient to prove Window-one-wayness security bounds.

Construction 3.3.1 *We define the Det-POP scheme* $(\mathcal{K}, \mathcal{O}ffset, \mathcal{R}andom, \mathcal{E}ncrypt, \mathcal{D}ecrypt)$ *as follows:*

- $\mathcal{K}$ *is a pseudo-random function that generates a 128-bit key for each run,* K_r*, and a 128-bit seed for each domain,* $S_{\mathcal{D}}$
- $\mathcal{O}$*ffset randomly selects a sequence of offsets* j_r *for each Run* r *from* M*,* $j_r \stackrel{\$}{\leftarrow} M$.

clear text	Run 1	Run 2	Code (⟨ run, code-id ⟩)
Beatles		6	$\{\langle 2,6\rangle\}$
Elton John	9		$\{\langle 1,9\rangle\}$
Madonna		8	$\{\langle 2,8\rangle\}$
Metallica	13	2	$\{\langle 1,13\rangle, \langle 2,2\rangle\}$
Nelly Furtado	1	3	$\{\langle 1,1\rangle, \langle 2,3\rangle\}$
Tina Turner	5		$\{\langle 1,5\rangle\}$

Table 3: Prob-POP Example

- *$\mathcal{R}andom$ deterministically selects a run r for each plaintext value, x, using a SHA-2 hashing function combined with a pseudo-random number generator (PRNG) to guarantee a uniform and deterministic partitioning of plaintext values across all runs. $r \leftarrow PRNG_{\text{SHA-2}(x,S_D)}$.*
- *$\mathcal{E}ncrypt$ is a modular random order preserving encryption function that takes K_r as the key and x as the plaintext value and computes the ciphertext,*
  $y \leftarrow \mathcal{E}ncrypt(K_r, x - j_r \bmod M)$.
- *$\mathcal{D}ecrypt$ takes the corresponding K_r as the key and ciphertext y and returns the plaintext value,*
  $x \leftarrow \mathcal{D}ecrypt(K_r, y) + j_r \bmod M$.

3.3.2 Probabilistic POP

A big advantage of POP is that it supports probabilistic (non equality-preserving) encryption. As shown in Section 5, this feature is particularly important for databases with skew (e.g., a few customers are responsible for most of the orders). With POP, probabilistic encryption can be achieved in many different ways: by using a probabilistic *random* function and/or by using a probabilistic *enc* function. Table 3 gives an example of such a POP-NEP encryption scheme, thereby using a probabilistic *random* function and a deterministic *enc* function for each run. This set-up is also the approach that we implemented and studied as part of this work; specifically, we used MOPE as *enc* function.

Construction 3.3.2 *We construct the Prob-POP scheme*
$(\mathcal{K}, \mathcal{O}ffset, \mathcal{R}andom, \mathcal{E}ncrypt, \mathcal{D}ecrypt)$ as follows:

- *$\mathcal{K}$, $\mathcal{E}ncrypt$, $\mathcal{D}ecrypt$ are the same as in Construction 3.3.1.*
- *$\mathcal{R}andom$ is a probabilistic function that chooses runs randomly but is aware of the data skew (e.g. knows which customers have many orders and which have not). The function is two-fold:*
  - *first, it chooses a set of ν random runs deterministically (analogously to construction 3.3.1). ν is proportional to the frequency of the item to encrypt (customers with many orders have a high ν).*
  - *second, a run is chosen probabilistically from this set.*

3.4 Stateful POP

The stateful POP with dictionary encryption (POP-DE) scheme is more expensive than the stateless POP variants, but it can be used to make *indistinguishability* guarantees (Section 5). The additional cost is in terms of additional state that needs to be kept at the Encryption Layer. Rather than merely storing a 128-bit key and an offset, the POP-DE scheme maintains an equality and order-preserving dictionary with circular shifts to encrypt values within a run. Table 2 gives an example for such a dictionary. The codes for each run in such a POP-DE dictionary are generated randomly. POP-DE supports dynamic databases with updates: If the random generator does not find a gap in the dictionary of a run for a new value, then

the POP-DE scheme expands and simply establishes a new run (Section 4). As the variants of stateless POP, POP-DE can also be deterministic or probabilistic, but as the security analysis is the same for both variants, we only give the description of the deterministic version here.

Construction 3.4.1 *We construct the POP-DE scheme* $(\mathcal{D}ict, \mathcal{E}ncode, \mathcal{D}ecode, \mathcal{R}andom)$ *as follows:*

- $\mathcal{D}ict$ *is the mapping table between the plaintext messages and the ciphertexts.* $\mathcal{D}ict$ *is dynamically filled up as the database grows, so there is no precomputation phase involved.*
- $\mathcal{R}andom$ *randomly selects a run from* $1 \ldots U$ *where* U *is the currently deployed number of runs.*
- $\mathcal{E}ncode$ *looks up the code* y *for plaintext* x *in the dictionary. If* x *is not in the dictionary,* $r \leftarrow$ *Random and a new code for* x *in run* r *is generated and returned. If there is no gap for* x *in run* r*, another run is chosen. If all the runs are full, a new run is created and* $U := U + 1$.
- $\mathcal{D}ecode$ *takes the run number* r *and code* y *and returns the plaintext message* $x \leftarrow \mathcal{D}ecode(y, r, \mathcal{D}ict)$.

4 POP SQL Processing

This section describes in more detail how the Encryption Layer can rewrite SQL queries and updates and generate POP-encrypted databases. While some of the details vary for the different POP variants, the basic principles are the same for Det-POP, Prob-POP, and POP-DE.

4.1 Query Rewrite

As shown in Figure 1b, the Encryption Layer rewrites incoming SQL queries so that they can be processed by the database system on the encrypted database without decrypting the data. 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 3.2 already showed how to rewrite simple equality and range predicates that involve POP-encrypted columns. 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 POP runs and whether a probabilistic or deterministic POP variant is used. For range predicates, the circular shift must be taken into account as shown in the example of Section 3.2. Overall, simple predicates that involve a column and a constant can be rewritten in a straight-forward way. Below, we show how more complex queries can be processed; i.e., general comparisons, joins, `GROUP BY`, and `ORDER BY` (e.g., Top N).

4.1.1 General Comparisons

Let us assume an *Order* table that involves among others two columns: (a) a *shipDate* that specifies when an order was delivered and (b) a *planDate* that specifies when the order was supposed to be delivered. 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 query works for Det-POP if the same keys and offsets for the enc functions are used to encrypt both the *shipDate* and the *planDate* of an order. Likewise, it works for POP-DE and Prob-POP if the same

dictionaries are used for both columns. In general, it is good practice to use the same keys and dictionaries for the encryption of values of the same domain (e.g., all dates or all names).

One caveat of the rewritten query, however, is that it may return *false positives*; i.e., orders whose *shipDate* was before the *planDate*, but whose *shipDate* was encrypted in a different run than its *planDate*. Such false positives can result in poor performance due to additional client/server communication and post-processing in the Encryption Layer. To avoid such false positives, we suggest to *tweak* the *random* function such that it is not applied to a single value (i.e., each field of a tuple separately) but in the granularity of a whole tuple; this way, it is guaranteed that the *shipDate* and *planDate* of an order are always encrypted in the same run. Fortunately, this restriction in the encryption scheme does not hurt confidentiality.

4.1.2 Joins

Functional joins along foreign key/key relationships can also be processed efficiently in all three POP variants presented in Section 3. To see why, let us look at a query that joins the *Customer* and *Order* tables:

```
SELECT * FROM Customer c, Order o
WHERE c.name = o.cust
```

It turns out that this query need not be rewritten if *Customer.name* and *Order.cust* are encrypted using the same key / dictionary, thereby following the general recommendation to use a single set of keys / dictionary for each domain as proposed in the previous subsection. For Prob-POP, the situation is slightly more complicated: We will revisit this aspect when we discuss the implementation of key/foreign key constraints in a POP encrypted database in Section 4.2.

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 described in the previous subsection. 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. Fortunately, such non-equi join queries are rare in practice.

4.1.3 `GROUP BY` Queries

Like joins, rewriting `GROUP BY` queries is straight-forward assuming that the metrics used in the aggregate functions are not encrypted (Section 3.1). 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; e.g., Det-POP or deterministic POP-DE. With these two variants, 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.

The Prob-POP was well as the probablistic POP-DE schemes require 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 `GROUP BY` clause involves several encrypted columns, then the number of tuples can grow exponentially with the number of columns in the `GROUP BY` clause in the worst case. 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.

Again, an important assumption made in this section is that the metrics used in the aggregate functions are not encrypted. 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 where the grouping and aggregation is carried out in such a case. Furthermore, if the metrics are encrypted using POP variants with MOPE as the underlying weak encryption scheme, and the query asks for the *maximum* and *minimum*, this aggregate cannot be pushed-down because rewriting the query would reveal the *Offset* of the circular shift of each run.

4.1.4 `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 r queries (with r the number of runs) and *merge* the results in the Encryption Layer.

Top N queries 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 involves several encrypted columns, 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.

4.2 Creation and Integrity of POP Databases

POP is a column-level encryption technique and can be combined with any other encryption technique inside of a database. For instance, *dates* and *customer names* can be encrypted using one of the three POP variants and public information such as *country names* or surrogates such as *order-ids* could be stored in plain text in the 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 set of keys or dictionary (Section 4.1). It is typically straight-forward to decide which columns need to be encrypted in which way.

Once the encryption scheme for each column has been selected, an initial number of runs, U, must be selected for each column (or more precisely domain) that needs to be encrypted using a POP-enhanced encryption technique. Values of $U = 5$ or $U = 10$ are common, but as shown in Section 6, the exact choice is not critical to achieve good performance. Tuples can then be encrypted one by one using the constructions described in Section 3 for the three POP variants. For stateful POP, it is guaranteed that an order-preserving cipher can be generated at any point in time; this feature is specific to the MOPE construction [7] that is adopted in these two POP variants. In our POP-DE construction, it is possible that a run has no more available ciphers for a new tuple: In this case, a new run is created and the *random* function is expanded to accommodate the new run.

Referential integrity constraints require special attention if a probabilistic POP variant such as Prob-POP is used. A probabilistic encryption of foreign keys is important if the database is skewed as shown in Section 5.2.3. The attacker might know that certain customers have placed most of the orders so that a probabilistic scheme is needed to distort the frequencies in which ciphers occur in the encrypted database. To implement such probabilistic encryption, we propose to encrypt foreign keys and primary keys in such

Name Dictionary

Value	Code
Tina Turner	$\langle 1,5\rangle$
Nelly Furtado	$\langle 1,1\rangle$
Nelly Furtado	$\langle 2,3\rangle$

Table Customer

name	info
$\langle 1,5\rangle$	...
$\langle 1,1\rangle$	...
$\langle 2,3\rangle$	...

Table Orders

id	cust
1	$\langle 1,5\rangle$
2	$\langle 1,1\rangle$
3	$\langle 1,1\rangle$
4	$\langle 2,3\rangle$

Figure 3: Prob-POP: Keys and Foreign Keys

a way that functional joins can be directly executed without decrypting these keys, as described in Section 4.1. Doing so may involve creating several tuples to represent the same entity. This situation is depicted in Figure 3. "Nelly Furtado" is encrypted using two codes. Correspondingly, two tuples must be stored for "Nelly Furtado" in the *Customer* table and the fields of these two "Nelly Furtado" tuples must be encrypted so that they are indistinguishable. *Orders* can refer to either of these tuples. In Figure 3, for instance, Orders 2 and 3 refer to the first Nelly Furtado Customer tuple (Code $\langle 1,1\rangle$) and Order 4 refers to the second Nelly Furtado Customer tuple (Code $\langle 2,3\rangle$).

4.3 Updates

Once a POP encrypted database has been created, it is straightforward to execute SQL update statements on it. The `WHERE` clause of any update statement (including inserts and deletes) is 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 in the same way as when creating a database and as described in the Construction schemes of Section 3.

For the POP-DE it is possible to garbage collect ciphers that are stored in the dictionary and that are no longer needed as a result of deletes and updates. Such garbage collection can be implemented in many different ways; both reference counting and mark & sweep approaches are applicable. Studying such garbage collection techniques in detail, however, was beyond the scope of this work. For the experiments presented in this paper, no garbage collection was applied.

5 Security Analysis

In this section we will show and briefly prove what security levels can be achieved by POP.[1] We begin with the *Window One-Wayness* and *Window Distance One-Wayness* definitions from [7], and analyse the effect of having POP in those equations. In the next subsection we re-study the indistinguishability results for committed databases in [7], and show that we can reach indistinguishability in a dynamic database set-up. Afterwards, we introduce two additional security notions, namely *Indistinguishability under Dynamic Chosen Plaintext Attack (IND-DCPA)* and *Rank One-Wayness (ROW)*. The two latter security definitions have been driven from concrete scenarios of protecting a database against system administrators.

5.1 One-Wayness

Intuitively, the one-wayness property means that no efficient adversary has any significant advantage of finding the plaintext that corresponds to a ciphertext, y, only by seeing y. The notion of one-wayness for strictly order-preserving (OPE) and modular order-preserving encryption schemes (MOPE) has been

[1] In order to fully understand this section we advise the reader to look at the two consequent Boldyreva papers on OPE namely [6, 7], and additionally [5] to have a short overview on Symmetric Encryption.

thoroughly analysed by [7]. [7] defines two even stronger and more general notions of one-wayness: *Window One-Wayness* and *Window Distance One-Wayness*. An important question to answer is whether our POP enhanced encryption schemes (see section 3.2) scheme can enhance the security of One-wayness in any way or not. In this subsection we restudy the definitions of one-wayness in [7]. Afterwards, we plug in our parameters in those formulas and analyse what level of security can be achieved by using stateful POP instead of pure MOPE.

Window One-Wayness. Given many encryptions $y_1, , , y_z$ of randomly chosen x_i from the domain, find an interval $I \subset [M]$ whose size is r in which one of the inverses lie [7]. For MOPE which is the building block of Det-POP, it has been proven that it is optimally one-way in the sense that an adversary cannot do better than an adversary that outputs a random window independent of the challenge set [7]:

Proposition 5.1.1 *Fix any window size r and challenge set size z. Let $A_{rand}(r)$ be an $r, z - WOW$ adversary that, on any input, outputs a random r-window from [M]. Then for any adversary A,*

$$Adv^{r,z-wow}_{MOPE_{[M],[N]}}(A) \leq Adv^{r,z-wow}_{MOPE_{[M],[N]}}(A_{rand}(r)) \leq r\frac{z}{M} \tag{1}$$

Now we are interested to see how inequality 1 changes using Det-POP. Before we plunge into new derivations, we need to state three remarks. First of all, we are interested in one-wayness instead of window one-wayness so from now on we have $r = 1$.

Remark 5.1.1 *Window one-wayness generalizes the standard notion of one-wayness, when $r = 1$. That is find the exact inverse of some y_i instead of an approximation in some range.*

Second, we show that each run has to be treated independently.

Lemma 5.1.1 *Under random partitioning strategy μ, any executions of runs i and j are indistinguishable.*

This simple lemma has a useful corollary that if we release z ciphertexts, then almost certainly every run will have roughly the same number of ciphertexts revealed, namely $\frac{z}{U}$.

Remark 5.1.2 *The number Z entries revealed in each run satisfies (Hoeffding inequality) $|Z - z/U| \leq (z/U)^{2/3}$, with negligible probability.*

Thus each run is equivalent to an attack on a cipher with a single run with Z cipher texts.

Third adjustment is the number of ciphertexts to be revealed i.e. the parameter z. In order to simulate an adversary that has access to the encrypted database, we need to maximize z in inequality 1. However, the one-wayness definition states that the ciphertexts $y_1, , , y_z$ correspond to cleartexts x_i that are selected uniformly at random from the domain. According to the birthday paradox, for $z = c\sqrt{M}$, two inputs have a high likelihood to collide if we pick x_i randomly and independently with replacement. Thus the maximum number of z to be revealed is bounded by $z \leq \sqrt{M}$. After adjusting the parameters in inequality 1 for Det-POP we will have:

Lemma 5.1.2 *In POP with the random partitioning strategy, the 1,z-window one-wayness advantage with U runs is $\epsilon(n)$ which is $\leq \frac{1}{U\sqrt{M}}$; for $r = 1, z = \lambda\sqrt{M}$ ($Z = \frac{\lambda\sqrt{M}}{U}$) and $\lambda < 1$:*

$$\epsilon(n) \leq \frac{rZ}{M} = \frac{\sqrt{M}}{U \times M} = \frac{1}{U\sqrt{M}} \tag{2}$$

We observe that in the worst case scenario in which an adversary has the most randomly selected ciphertexts possible, the advantage decreases inversely proportional to U and $\sqrt{M}$.

Now let us take a look at the definition and the application of our adjustments to the *Window Distance One-Wayness* notion from [7].

Distance One-wayness. Given many encryptions $y_1,,,,y_z$ of randomly chosen plaintexts, find a range I whose size is r in which the distance between x_i and x_j lies for some i,j. It is shown by [7] that the advantage is bounded by

$$\varepsilon(n) \leq \frac{9z(z-1)}{\sqrt{M-z+1}} \tag{3}$$

Similarly to window one-wayness, if we maximize z, the birthday paradox will limit z to be bounded as $z \leq \sqrt[4]{M} \Rightarrow Z = \frac{\sqrt[4]{M}}{U}$.

Lemma 5.1.3 *In POP, the distance window one-wayness advantage is decreasing quadratically with U. For $Z = \frac{\zeta \sqrt[4]{M}}{U}$ it is*

$$\epsilon(n) \leq \frac{10Z(Z-1)}{\sqrt{M-Z+1}} < 10\frac{\zeta^2\sqrt{M}}{U^2\sqrt{M}\sqrt{1-\zeta+M^{-1}}} = \frac{10\zeta^2}{U^2\sqrt{1-\zeta+M^{-1}}} \tag{4}$$

The factor 10 in place of 9 is to compensate for the approximation of Z from the Hoeffding bound.
So far, by introducing the concept of runs, we have shown that the advantage of the adversary in the Window One-Wayness and Distance Window One-Wayness model is declining by a factor of U and U^2 respectively.

5.2 Indistinguishability

The property of indistinguishability under chosen plaintext attack is considered a basic requirement for most provably secure cryptosystems. An encryption scheme is said to be indistinguishable, if an adversary is not able to distinguish pairs of ciphertexts based on the message they encrypt. [6] shows that indistinguishablity under ordered chosen-plaintext attack is unachievable by a practical order-preserving scheme, unless its ciphertext space is exponential in the size of the plaintext space. In their follow-up paper [7], a new and enhanced order-preserving scheme for static databases called Committed Efficiently-Orderable Encryption(CEOE) is introduced whose indistinguishablity under a committed chosen plaintext attack (IND-CCPA) is proven in [7]. Nevertheless, the adversary in [7] is restricted to only choose plaintext vectors with the *same order and equality pattern* for the experiment. In this section we first revisit how CEOE achieves indistinguishability under committed chosen plaintext attack and then we introduce our dynamic set up and show that we can achieve the same for a dynamic database using POP with its dictionary encoding variant (POP-DE).

5.2.1 Committed Databases

[7] introduces an encryption scheme called CEOE that is based on the fact that the database is committed beforehand i.e. a static database. A ciphertext, y, in CEOE consists of a semantically-secure ciphertext of the message, $\mathcal{E}nc_{SE}(x) = y'$, preceded with a tag, i, which indicates the rank of the message in the ordered and committed message list. In other words, $y = i\|y'$. As a building block, CEOE uses a monotone minimal perfect hash function [4] to efficiently generate the tags. The following adversary experiment is used to prove IND-CCPA

First, One generates the keys, an adversary splits the domain into M_0 and M_1. The perfect hash is picked based on this. Then the adversary picks two vectors X_0 and X_1 that have identical comparison structure, i.e. if $x_i < x_j$ then $x_i^{'} < x_j'$ and similarly for equality. The advantage is defined as follows:

$$Adv_{\text{CEOE}}^{\text{ind-ccpa}}(A) = Pr[Exp_{\text{CEOE}}^{\text{ind-ccpa-1}}(A) = 1] - Pr[Exp_{\text{CEOE}}^{\text{ind-ccpa-0}}(A) = 1]$$

Lemma 5.2.1 *The attacker's advantage in the committed database setting is negligible [7].*

5.2.2 Database with Updates

In this model, the adversary gives us a data sequence that is not fixed ahead of time. In the previous model the entire vector was given all at once. Now we proceed as follows. We will chose U_1 and U_2 with $U_1 + U_2 = U$, the total number of runs. We may keep $U_2 \geq 1$ and small. These are used for update spill overs and are called *update runs*.

1. **Initialization**: The data base is loaded at this time with $X = (x_1, ...x_n)$. We distribute these in U_1 runs. Each one has its own perfect hash that gives an index to each element. We also leave some gaps; the gaps may be deterministic or random. Each run is independently done.
2. **Update phase**: the adversary brings in $x_j, j \leq z$ one at a time. We make updates to the DB in the following way. If x_j can be inserted into some predetermined run $i < U_1$, we can insert it. Else we insert into one of the update runs. The probability that we do not find an insertion point can be decreased by increasing U;

Using POP, we never run out of insertion space for the new coming plaintext messages, because we can always create new runs to accommodate them. In other words, we spare one of the runs of U_2. Here we show that using POP-DE (see section 3.4) we will reach indistinguishability under chosen plaintext attack for a dynamic database. At this stage we use the same restrictions on the adversary model as in [7], namely the plaintext vectors chosen by the adversary should have *same order and equality pattern*. In order to prove indistinguishability under chosen plaintext attack we show that we can efficiently reduce our POP-DE scheme to CEOE and thus claim if CEOE is IND-CCPA then we are IND-DCPA (indistinguishable under dynamic chosen plaintext attack). Please mind that IND-DCPA has only been proven for POP-DE; Whether stateful can be made indistinguishable, remains as a future challenge.

IND-DCPA. Let POP-DE be the scheme from construction 3.4.1. For an Adversary $A = (A_1, A_2)$, we define its IND-DCPA advantage by considering the union of traces of all executions of runs, as:

$$Adv_{\text{POP-DE}}^{\text{ind-dcpa}}(A) = Pr[Exp_{\text{POP-DE}}^{\text{ind-dcpa-1}}(A) = 1] - Pr[Exp_{\text{POP-DE}}^{\text{ind-dcpa-0}}(A) = 1]$$

For $b \in \{0, 1\}$ the experiments $Exp_{POP-DE}^{ind-dcpa-b}(A)$ are defined as follows:

σ denotes a state the adversary can preserve. We say POP-DE is IND-DCPA secure if the IND-DCPA advantage of any adversary is small.

Theorem 5.2.1 *The POP-DE scheme is IND-DCPA secure provided that CEOE [7] is IND-CCPA secure.*

Proof: Let POP-DE be the scheme from construction 3.4.1. Suppose $A = (A_1, A_2)$ is an adversary with non-trivial IND-DCPA advantage against POP-DE. We construct an IND-CCPA adversary B against CEOE. B has access to a left-right encryption oracle. B runs A_1 to receive X_0, X_1, σ. Let $l = |X_0| = |X_1|$. B queries its left-right oracle with matched pair of this messages $i \| y_j \leftarrow O(x_j^0, x_j^1)$ for $j = 1, ..., l$. Then, B throws the y_j part of the ciphertext away and thus has only the i which is an order tag. Finally it runs $A_2(\sigma, i_1, ..., i_l)$. B's communication with A mimics the IND-DCPA experiment if everything would have been encrypted in the same run, thus the IND-DCPA advantage of A in a dynamic set up is at least equal to IND-CCPA advantage of B. In other words, IND-CCPA advantage of adversary B in a committed setup is an upperbound to IND-DCPA advantage of adversary A. Also B is efficient in a constant order since all it has to do is to remove the ciphertext part of the encrypted message.

Experiment 1 : $\mathbf{Exp}^{ind-dcpa-b}_{POP-DE}(A)$

$(X_0, X_1, \sigma) \xleftarrow{\$} A_1$
if $|X_0| \neq |X_1|$ **then** output $\perp$
else $l \leftarrow X_0$
let $x^j_1 < x^j_2 < ... < x^j_l$ be the elements of X_j for $j = 0, 1$
if $\exists i \in \{1 \leq i \leq l\}$ $where$ $|x^0_i| \neq |x^1_i|$ **then** output $\perp$
for $j = 1 \rightarrow l$ **do**
  $r \xleftarrow{\$} U_1$
  $flag \leftarrow 0$
  for $u = 1 \rightarrow U_1$ **do** ▷ randomly select run r without replacement until gap found
    if space available for x^b_j in r **then**
      $y_j \leftarrow Encode(x^b_j, r, Dict)$
      $flag \leftarrow 1$
    else $r \xleftarrow{\$} U_1$
  if $flag = 0$ **then** $r \leftarrow createNewRun$
    $U_1 \leftarrow U_1 + 1$
    $y_j \leftarrow Encode(x^b_j, r, Dict)$
$d \xleftarrow{\$} A_2(\sigma, y_1, ..., y_l)$;**return** d

5.2.3 Statistical Indistinguishability

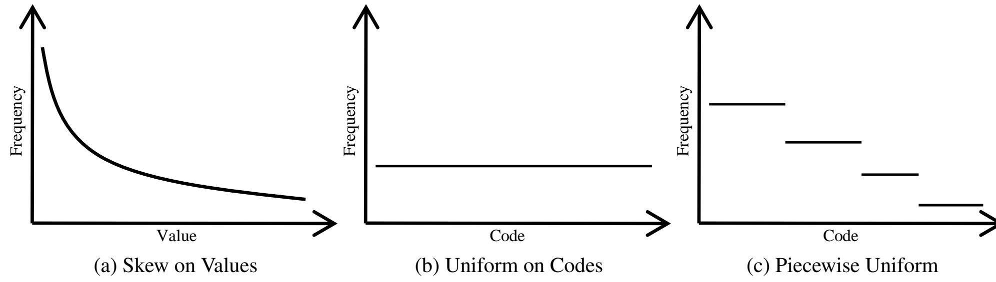
Databases usually contain a lot of statistical information on the data. A problem with equality-preserving encryption schemes is that they leak the frequencies of the ciphertexts. This information can be used by attackers with background knowledge (on the frequencies of plaintexts) to perform statistical attacks. In this section, we define a new adversary model which is based on the previous definition, i.e. indistinguishability under chosen plaintext attack *but this time* the adversary is allowed to choose plaintext vectors with different equality patterns. Still, the plaintext vectors should be ordered, therefore we call it *indistinguishability under ordered and dynamic chosen plaintext attack* (IND-ODCPA). Let us illustrate this with an example:

Assume adversary $\mathcal{A}$ once selects $X_0 = (a, a, b, c)$ and the second time it selects $X_1 = (a, b, c, d)$ (note that X_0 and X_1 have the same ordering, but not the same equality pattern). A not IND-ODCPA secure encryption scheme (basically all deterministic encryption schemes) would return something like the following:
$\mathcal{E}nc(X_0) = (12, 12, 25, 36)$
$\mathcal{E}nc(X_1) = (12, 25, 36, 44)$
For adversary $\mathcal{A}$ it is easily distinguishable which one is X_0 and which one is X_1. However, an IND-ODCPA secure scheme is able to return something like the following:
$\mathcal{E}nc(X_0) = (12, 25, 36, 44)$
$\mathcal{E}nc(X_1) = (12, 24, 37, 45)$
Clearly, these two vectors are indistinguishable for adversary $\mathcal{A}$.

In section 3, we have defined a probabilistic encryption scheme called Prob-POP that flattens out the frequent peaks in the data distribution. We will now use the new adversary model to show whether Prob-POP achieves IND-ODCPA or not. The adversary is given a left-right oracle on our Prob-POP scheme. His IND-ODCPA advantage is defined as:

$$Adv^{\text{ind-odcpa}}_{\text{ProbPOP}}(\mathcal{A}) = Pr[Exp^{\text{ind-odcpa-0}}_{\text{Prob-POP}}(\mathcal{A}) = 1] - Pr[Exp^{\text{ind-odcpa-1}}_{\text{Prob-POP}}(\mathcal{A}) = 1]$$

where for $b \in \{0, 1\}$ the experiments $Exp^{\text{ind-ODCPA}}_{\text{Prob-POP}}(\mathcal{A})$ are analog to experiment 1. We say that Prob-POP



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

Figure 4: Distorting a Frequency Distribution

is IND-ODCPA secure if the IND-ODCPA advantage of any adversary against Prob-POP is small.

Proof: Prob-POP can be IND-ODCPA if an idealized smoothening function is used, to flatten out every peak in the frequency distribution. To illustrate the notion of an idealized smoothening function, assume we have a skewed distribution of the values in the database as shown in figure 4a. Applying the idealized smoothening function will give us a flat distribution shown in figure 4b.
However, according to the definition of Prob-POP in section 3.3.2 (our implemented version of Prob-POP), we will end up having a step frequency distribution shown in figure 4c. Generally speaking, using an idealized smoothing function is costly in practice because it will lead to an explosion in the number of codes inserted into Database.
The question of whether our practical implementation of Prob-POP is still IND-ODCPA secure, remains open as a future challenge. Note that we do not claim that we are semantically secure, because we are still leaking order. All the chosen plaintext attacks are under the condition that adversary chooses two ordered messages.

5.3 Rank One-Wayness

Rank One-Wayness. As noted earlier, knowing the Domain, $\mathcal{D}$, and all their ciphertexts allows one to break OPE using sorting. Introducing modular partial orders by Det-POP amends the OPE scheme in this regard. For example, if the actual database entries are chosen according to some statistical model and constitute only a fraction of the whole domain, the attack becomes harder. If we view the OPE in the runs as a coding scheme, then the problem at hand is that of finding the most likely subset of $\mathcal{D}$ given the encryptions, and this may lead to a huge number of solutions. Whether the data model is stochastic (or otherwise) belongs to the realm of random partial orders and combinatorial optimization. We do not pursue this here.
In this regard we suggest a notion of **Rank One-Wayness**. In this model, the adversary is given the rank of a ciphertext y in run r, $rank_{y,r}$ and is asked to find the actual rank of its underlying plaintext, $rank_x$ among the values of $\mathcal{D}$. Rank One-Wayness (ROW) advantage of an adversary $\mathcal{A}$ against Det-POP $_{[M],[N],[U]}$ with domain size M, and U number of runs is: [2]

$$\begin{aligned} Adv^{\text{ROW}}_{\text{[M],[N],[U]}}((A)) &= Pr[Exp^{\text{ROW}}_{\text{Det-POP}_{\text{[M],[N],[U]}}}(A) = 1] \\ &= \frac{\binom{M-1}{\frac{M}{U}-1}}{\binom{M}{\frac{M}{U}}\frac{M}{U}} \end{aligned} \tag{5}$$

[2]also applicable to POP-DE

Experiment 2 : $\mathbf{Exp}^{\text{ROW}}_{\text{Det-POP}_{[M],[N],[U]}}(A)$

1: $r \xleftarrow{\$} U; x \xleftarrow{\$} M; rank_x \leftarrow Rank(x)$
2: $y_x \leftarrow \mathcal{E}ncrypt(K_r, x) - j_r \ mod \ M$
3: $rank_y \leftarrow Rank(y_x, r)$
4: $rank'_x \xleftarrow{\$} A(rank_y)$
5: **if** $rank_x = rank'_x$ **then return** 1
6: **else return** 0

The function Rank(x), takes a plaintext message x and returns its ordinal rank in $\mathcal{D}$. Its other variant Rank(y, r) returns the ordinal rank of the ciphertext (or code) within a run r. The advantage of a ROW-adversary on Det-POP is uniform as shown in equation 5. Hereby we can say that Det-POP is optimally rank one-way in the sense that an adversary cannot do better than an adversary that outputs a random rank independent of the challenge set.
Here we give a short description on the derivation of the formula in equation 5. The denominator holds all possible encoding combinations in a Det-POP scheme. According to combinatorics theory, the number of ways you can select a subdomain of size M/U out of a domain of size M is:

$$\prod_{i=0}^{U-1} \binom{M - \frac{M}{U} i}{\frac{M}{U}}$$

This product now has to be multiplied by the number of cyclic combinations of the ciphertexts we can have within a run. Using again the Hoeffding inequality we can say in each run we will have about $\frac{M}{U}$ ciphertexts; this number corresponds exactly to the number of combinations we will have within a run yiedling the following denominator:

$$\frac{M}{U} \prod_{i=0}^{U-1} \binom{M - \frac{M}{U} i}{\frac{M}{U}}$$

Now that we have seen how the denominator was derived, we proceed with the numerator. In the numerator we are interested in one specific combination, so the number of combinations we can have for the rest of the elements in a run is:

$$\binom{M-1}{\frac{M}{U} - 1} \prod_{i=1}^{U-1} \binom{M - \frac{M}{U} i}{\frac{M}{U}}$$

The two products in the numerator and denominator cancel out quite well yielding equation 5.

5.4 Known Plaintext Attack

We now consider an adversary who receives several plaintext-ciphertext pairs. In our modular order-preserving scheme let us assume the adversary has two pairs of plaintext and ciphertext, namely x_1, y_1 and x_2, y_2. These two pairs being revealed, enables the adversary to break the domain space, $\mathcal{D}$, and ciphertext space, $\mathcal{R}$ into subspaces. According to the extended one-wayness analysis in subsection 5.1, the advantage of the adversary is declining more rapidly with the number of runs, and is thus less affected by changes in the domain size. This shows that having runs makes an underlying order preserving scheme more resilient to the break-down of the range and domain spaces in a known plaintext attack.
However, revealing two pairs of plaintext and ciphertext, would break the effectiveness of the modular order-preserving scheme and change it to a regular order-preserving scheme (such as ROPE [6]). This matter will affect the rank one-wayness advantage of the adversary in two ways: first the advantage of

the adversary will be higher and second the probability distribution will no longer be uniform but will fall onto a negative hypergeometric probability distribution as follows:

$$Adv^{\text{ROW}}_{\text{[M],[N],[U]}}((A)) = Pr[Exp^{\text{ROW}}_{\text{POPROP}_{\text{[M],[N],[U]}}}(A) = 1] = \frac{\binom{rank_x - 1}{rank_{(y,r)} - 1}\binom{M - rank_x}{M/U - rank_{(y,r)}}}{\binom{M}{M/U}} \tag{6}$$

Figure 5 shows the probability distribution of $\mathcal{A}^D$ advantage in case the modular order preservig encryption scheme is broken due to a known plaintext attack. In this case using *Runs* is essential to keep the cryptosystem from breaking completely, i.e. without POP we would have: $Adv^{\text{ROW}}_{\text{OPE}}(\mathcal{A}^D) = 1$ However, as shown in Figure 5, even with small number of *Runs* the extreme values of the domain (starting with 'A' or 'Z') are exposed to a higher probability of discovery. The values in the middle of the domain have almost a uniform probability distribution and thereby safer.

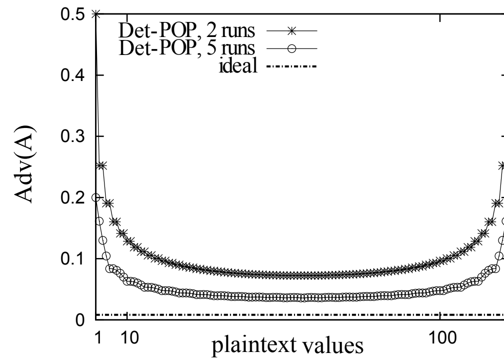


Figure 5: Rank One-Wayness Probability Distribution for POP under Known Plaintext Attack

Here we give a short description on the derivation of the formula in equation 6. The denominator holds all possible encoding combinations in a Det-POP scheme using a strictly order preserving encryption scheme such as ROPE [6] as the underlying weak encryption scheme. According to the combinatorics theory the number of ways you can select a subdomain of size M/U out of a domain of size M is:

$$\prod_{i=0}^{U-1} \binom{M - \frac{M}{U} i}{\frac{M}{U}}$$

This product is basically the number of total orders you can have construct given U partial orders of size M/U. Now that we have seen how the denominator was derived, we proceed with the numerator. In the numerator we are interested in one specific combination, so the number of combinations we can have for the rest of the elements in a run is:

$$\binom{rank_x - 1}{rank_{(y,r)} - 1}\binom{M - rank_x}{M/U - rank_{(y,r)}} \prod_{i=1}^{U-1} \binom{M - \frac{M}{U} i}{\frac{M}{U}}$$

The two products in the numerator and denominator cancel out quite well yielding equation 6.

6 Performance Experiments

This section assesses the performance overhead of POP using the queries of the TPC-H benchmark. We chose TCP-H as a benchmark because it has a OLAP workload and therefore models well the behaviour

of a typical cloud database application. We present the results for deterministic and probabilistic POP. The performance of these two POP variants is compared to two baselines that represent the current state of the art. First, a database that is not encrypted, referred to as *plain*. Obviously, all POP variants are expected to perform worse than such a plain database. Second, a database that is encrypted using AES in ECB mode (Det-AES) and 128 bit keys. We expect Det-POP to perform better than AES-ECB if the query involves range or `LIKE` predicates or sorting. Comparing Det-AES and Prob-POP, the trade-offs are less clear: Prop-POP is probabilistic whereas AES-ECB is deterministic. We did not try AES in CBC mode. AES in CBC mode is probabilistic and in such a scheme (almost) no operator can be processed in the database so that all query processing needs to be carried out as part of post-processing in the Encryption Layer.

6.1 Benchmark Environment

The experiments were carried out using a client-server environment as shown in Figure 1b: the Encryption Layer rewrote the queries as explained in Section 4.1 and the rewritten queries were then processed by the database server process on the encrypted database. Client and server were hosted on different machines in different locations. The communication went through one router (2 hops) and the RTT was 0.7ms on average. The client was written in Java, ran on a machine with 24 GB of memory and 8 dual-core CPUs (with 2.27 GHz clock speed) and communicated to the database server using the JDBC driver built on top of TCP socket connections. The server machine had 132 GB of memory and 8 CPUs with 2.40 GHz clock speed. MySQL Version 5.6 was used as a database system. Both machines ran a Debian-based Linux distribution. The results returned by the database system were decrypted and post-processed by the Encryption Layer as described in Section 4.1.

We measured query rewriting time (compilation time), the time decryption and post-processing took (client time), the time spent for query processing in MySQL (server time) and bytes transmitted between client and server (communication cost). We canceled queries after a timeout of 30 minutes. Furthermore, we repeated all queries ten times and report on the mean and standard deviation of the last six iterations.

In all experiments reported in this section, the 22 queries and 2 refresh functions of the TPC-H benchmark were used. We used a database with a scaling factor of 10 (i.e., 10 GB).

6.2 Database Encryption

Metrics used in aggregate functions (e.g., volume of orders) were not encrypted. Furthermore, surrogates (e.g., order numbers) were not encrypted.

All other attributes such as names, dates, etc. were encrypted using Det-POP, Prob-POP, or Det-AES. A specifically challenging operator for encrypted values is *extract-year*, appearing in Q7, Q8 and Q9. This is why in order to prepare for such dates we encrypted the year of *o_orderyear* in a separated field. We also made sure that the values of *l_shipdate*, *l_commitdate* and *l_receiptdate* for a specific record were encrypted in the same run. For Prob-POP, we distorted the correlation between the *customer* and the *orders* table by adding customer clones (as described in Section 4.2). In order to prevent an adversary from linking clones to original tuples, we had to encrypt the *customer* table completely (even insensitive attributes like *c_comment*). We marked clones as such, which is not a privacy threat as long as an adversary cannot link the clones to their original customer record. In order to achieve this, all unique customer attributes (e.g. *c_phone*, *c_name* and *c_address*) had to be encrypted probabilistically. This was especially challenging for the attribute *c_acctbal* which on one hand uniquely defines a customer and on the other hand is a metric and should therefore not be encrypted. We decided to set *c_acctbal* to 0 for all customer clones and to the correct value for the orginінal customers. The encrypted attribute *c_name* is used as link between originals and clones. As we have to link originals to clones anyway during post-processing (using the decrypted attribute), this is also how we retrieve the correct value of *c_acctbal*.

Only for *plain* (one of the baselines), nothing was encrypted. We varied the number of runs, U for Det-POP and show the results in Section 6.6. By default, we used $U = 10$ runs in all other experiments presented in this section.

Plain	Det-AES	Det-POP	Prob-POP
24 GB	63 GB	44 GB	50 GB

Table 4: Database Size (GB)

6.3 Database Size

Table 4 shows the size of the databases for the two POP variants and the two baselines. These numbers include indexes. In all approaches, the same kind of indexes were created, however, for POP-encrypted index attributes, we used a composed index <run, key>. The *plain* database is smallest because any kind of encryption blows up the database because the ciphers are larger than plain text values. The Prob-POP database was significantly bigger than the Det-POP database for creating additional tuples to maintain referential integrity constraints with probabilistic encryption as described in Section 4.2 (Figure 3). The Det-AES database was even bigger because it pads ciphertexts to multiples of 16 bits (and a minimal length of 32 bits).

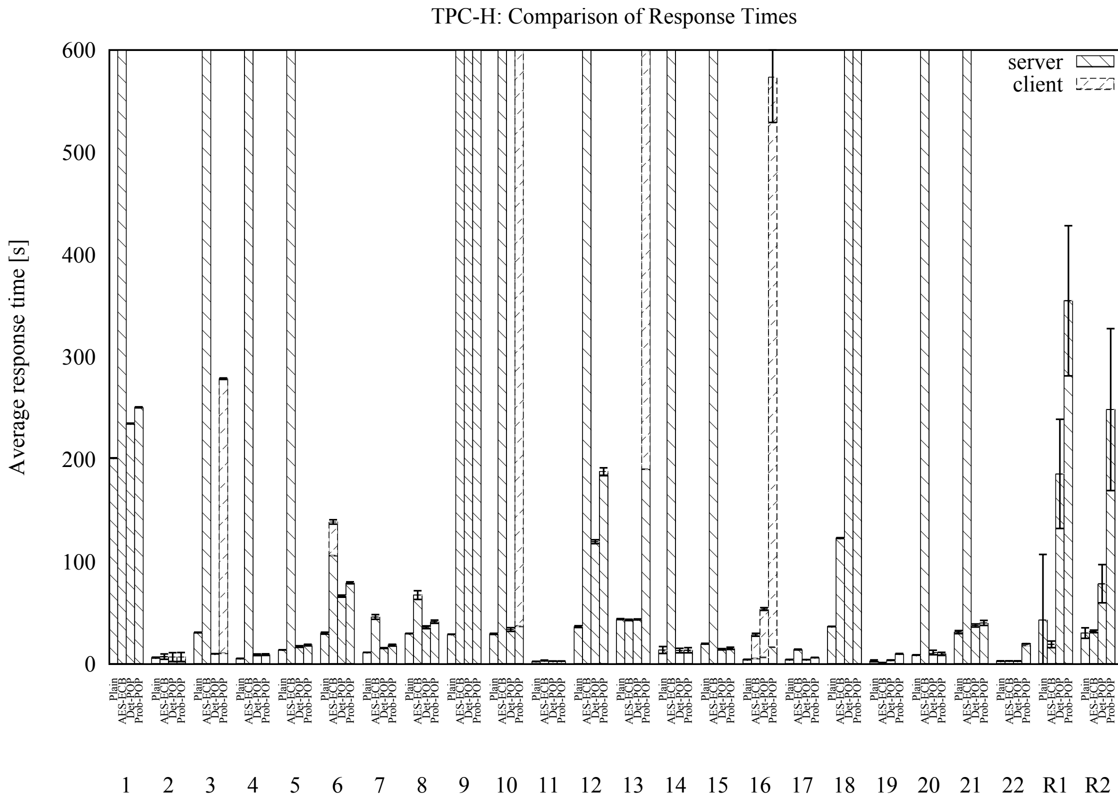


Figure 6: Running Time (secs): TPC-H Queries, $U = 10$

6.4 TPC-H Queries

Figure 6 shows the TPC-H benchmark for Det-POP, Prob-POP, and the two baselines as a breakdown of client and server time. We cut the y-axis at 10 minutes for readability reasons. The exact numbers (including standard deviation) can be found in table 5. As expected, *plain* has the lowest response time in (almost) all cases. The good news is that both POP variants are typically not far behind and have little overhead for most queries. Again, as expected, Det-POP beats Prob-POP across the board as deterministic encryption is much cheaper than probabilistic encryption. Det-AES has acceptable performance in all cases in which the query can (almost) fully be processed by MySQL without decrypting the data; these are simple join queries and queries with equality predicates only. It turns out that not many TPC-H queries fit this bill so that Det-AES timed out for half of the TPC-H queries. In all these cases, Det-AES had to ship a significant amount of data from the database to the Encryption Layer and a great deal of post-processing was required at the Encryption Layer. In contrast, the POP variants only timed out for query 9 (where also Det-AES times out) and query 18 (only Prob-POP).

While the big picture is clear and quite encouraging for POP, there are a number of more detailed observations that can be made by looking at the results of certain queries:

- *Range Predicates (Q1, Q5, Q6, Q14, Q15, Q20):* Being able to answer range queries efficiently was the most important motivation for POP. We see that both POP variants do well in that regard as they prevent irrelevant data (false positives) from being shipped to the client. Det-AES on the other hand, has no means to encrypt range queries and therefore the entire data has to be shipped, which often results in unacceptable performance.
- *Group-By (Q3, Q10, Q16, Q18):* Group by clauses on encrypted attributes are no problem for deterministic encryption, but are a serious challenge for probabilistic encryption as explained in Section 4.1.3. The reason why Det-AES has high client time for Q16 is that this query additionally employs a range predicate.
- *Top-N (Q2, Q3, Q10, Q18):* Top-N queries can be processed as sub-queries as described in Section 4.1.4. While this works well for Q2, Q3 and Q10, it seems that it does not work well for Q18. This is because the working set is large and therefore the different MySQL threads (initiated by the different parallel sub-queries) pollute each other's caches.
- *Infix Predicates (Q9):* Unlike prefix predicates that can be rewritten as range predicates, infix and postfix predicates cannot. Therefore all encryption methods strugle with predicates as *"p_name LIKE %BLUE%"*
- *Non-equi Joins (Q2, Q9, Q21):* While equi joins work well in POP, non-equi joins are difficult for all encryption techniques. The cost of a non-equi join depends on its selectivity. If the selectivity is small (as in Q9), lots of irrelevant data records have to be shipped to the client as they can only be discarded after decryption. For Q2, the top-N parallelism compensates for the non-equi-join penalty. For Q21, the POP variants perform well because *l_shipdate*, *l_commitdate* and *l_receiptdate* are encrypted in the same run.
- *Large Result Sets (e.g. Q16):* If the result sets grow large (several thousands), lots of tuples have to be shipped and decrypted and therefore client time dominates the server response time (compare to Figure 6 and Table 7).

6.5 Compilation & Communication Costs

Aditionally to client and sever time, we measured query compilation time, which was negligible in all cases except for RF1 for which we show the results in Table 6. This is because insertion tuples have to be encrypted before sending them to the database. As communication happens asynchronously and interleaved with post-processing, it is hard to measure. Instead, we measured bytes transmitted between client and server and present them in Table 7. No measurements are available for queries that timed-out.

Query	**Plain**	**AES-ECB**	**Det-POP**	**Prob-POP**
Q01	201 (+/- 0.239)	timeout	235 (+/- 0.354)	251 (+/- 0.482)
Q02	6.38 (+/- 0.331)	7.33 (+/- 2.55)	6.91 (+/- 4.32)	6.94 (+/- 4.37)
Q03	30.9 (+/- 0.304)	timeout	10.0 (+/- 0.136)	279 (+/- 0.736)
Q04	5.46 (+/- 0.0924)	timeout	9.08 (+/- 0.776)	9.31 (+/- 0.792)
Q05	14.0 (+/- 0.0367)	timeout	17.2 (+/- 0.771)	18.6 (+/- 0.796)
Q06	30.2 (+/- 0.963)	139 (+/- 2.12)	66.4 (+/- 1.09)	79.4 (+/- 0.961)
Q07	11.5 (+/- 0.0438)	46.1 (+/- 2.29)	15.7 (+/- 0.410)	18.6 (+/- 0.853)
Q08	29.8 (+/- 0.0707)	67.5 (+/- 4.27)	36.0 (+/- 1.40)	41.4 (+/- 1.57)
Q09	timeout	timeout	timeout	timeout
Q10	29.6 (+/- 0.505)	timeout	33.8 (+/- 1.81)	964 (+/- 10.2)
Q11	2.69 (+/- 0.0229)	3.53 (+/- 0.456)	2.9 (+/- 0.0612)	2.87 (+/- 0.0658)
Q12	36.6 (+/- 0.912)	timeout	120 (+/- 1.88)	188 (+/- 3.57)
Q13	44.0 (+/- 0.466)	43.2 (+/- 0.621)	43.6 (+/- 0.445)	1190 (+/- 1.35)
Q14	13.8 (+/- 3.44)	timeout	13.3 (+/- 2.18)	13.5 (+/- 2.55)
Q15	19.9 (+/- 0.392)	timeout	14.6 (+/- 0.629)	15.4 (+/- 1.13)
Q16	4.48 (+/- 0.28)	28.6 (+/- 1.39)	53.8 (+/- 1.29)	574 (+/- 44.3)
Q17	4.38 (+/- 0.164)	14.3 (+/- 0.359)	4.39 (+/- 0.0678)	6.44 (+/- 0.0743)
Q18	36.7 (+/- 0.0128)	123 (+/- 0.337)	1510 (+/- 1.75)	timeout
Q19	3.2 (+/- 0.829)	1.35 (+/- 0.0018)	3.9 (+/- 0.213)	10.1 (+/- 0.345)
Q20	8.84 (+/- 0.238)	timeout	11.4 (+/- 2.22)	9.98 (+/- 1.54)
Q21	31.3 (+/- 1.51)	timeout	37.7 (+/- 1.5)	40.3 (+/- 2.4)
Q22	2.97 (+/- 0.00327)	3.15 (+/- 0.00837)	3.01 (+/- 0.00368)	19.9 (+/- 0.136)
RF1	43.2 (+/- 64.0)	19.4 (+/- 3.24)	186 (+/- 53.3)	355 (+/- 73.4)
RF2	30.3 (+/- 5.05)	32.0 (+/- 1.25)	78.5 (+/- 18.7)	249 (+/- 79.2)

Table 5: Running Time (secs): TPC-H Queries, $U = 10$

Plain	Det-AES	Det-POP	Prob-POP
0.321	0.873	61.8	62.2

Table 6: Query Compilation Time (secs): RF1, $U = 10$

6.6 Vary Number of Runs

The number of runs is an important parameter of all POP schemes. As shown in Figure 7, however, the performance degradation is not dramatic if the number of runs does not grow beyond 50. Figure 7 shows how the running time of Det-POP increases with the number of runs for the first six TPCH-queries and relative to the *Plain* baseline. We chose these queries because they cover a wide spectrum of different operators (range predicates, aggregates, Top N, sub-selects, etc.). Overall, it can be seen that the curves are fairly flat. Only for Q6, there is a significant (but still linear) increase in the response time with a growing number of runs. Q2 and Q3 are Top N queries, are executed as parallel subqueries and become therefore with increasing number of runs (until we hit the limit of parallelism). Such queries even beat *Plain* as long as the database is not busy with other queries. For Q2, the parallelism compensates for the non-equi join (compare to Section 6.4), but cannot beat *Plain*. Q1, Q4 and Q5 are typical representatives of most TPCH-queries: the optimal number of runs is somewhere between 5 and 20 (and not 1 as one might expect). This indicates, that even if we do not employ parallelism explicitly, modern query processors partition the data anyway and execute certain operations (like hash joins and merge sorts) in parallel.

Query	Plain	Det-AES	Det-POP	Prob-POP
Q01	0.704	N/A	1.69	1.73
Q02	12.6	7020	3710	4040
Q03	0.624	N/A	15.7	3220
Q04	0.397	N/A	1.38	1.42
Q05	0.585	N/A	1.66	3.25
Q06	0.231	209000	1.18	1.22
Q07	0.882	1.52	3.38	9.96
Q08	0.664	0.961	1.95	6.60
Q10	5.11	N/A	6.23	477000
Q11	121	126	121	124
Q12	0.538	N/A	1.71	1500
Q13	0.998	0.998	0.998	108000
Q14	0.302	N/A	2.79	2.91
Q15	0.465	N/A	1.54	1.58
Q16	1260	128000	1360	24500
Q17	0.238	0.351	0.321	2.43
Q18	6.06	11.4	11.0	N/A
Q19	0.914	4.43	3.12	54.3
Q20	116	N/A	124	129
Q21	133	N/A	4.04	5.68
Q22	0.627	0.513	0.627	1210
RF1	11600	36600	17900	18800
RF2	205	205	205	205

Table 7: Communication Cost (Kbytes transmitted): TPC-H Queries, $U = 10$, std negligible

This shows that the partitioning induced by POP does not hurt performance, but even becomes beneficial, especially in distributed systems (resp. the cloud).

7 Conclusion

This paper presented a new encryption scheme for databases called POP. The key idea of POP is to randomly partition the values of a domain and then use a conventional encryption technique to encrypt the values of each partition separately. This composed approach allows the use of weaker encryption techniques for each partition, thereby providing much better query performance than a strong encryption scheme and much better security guarantees than a weak encryption scheme.

We studied two different POP variants. The first variant (Det-POP) was based on a modular order-preserving encryption technique, called MOPE [7]. The second variant (Prob-POP) was based on MOPE and a probabilistic partitioning of plaintext values as part of the POP partitioning. Furthermore, we studied different adversary scenarios in which administrators try to sniff into the (encrypted) database and infer confidential information. We were able to show that the strongest POP variant (i.e., Prob-POP) is able to provide security against frequency attack, even if the attacker knows the plaintext domain and its frequency distribution.

The performance experiments showed that deterministic POP (i.e., Det-POP) has affordable overhead as compared to an unencrypted, plain database that provides no overhead for most TPC-H queries. Only for queries with certain access patterns (e.g., non-equi join queries), the performance of Det-POP was

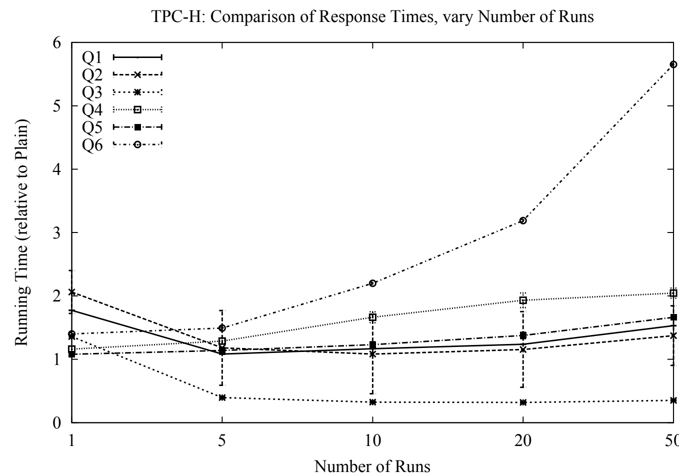


Figure 7: Det-POP/Plain, Vary # of Runs, Div. Queries

significantly worse because these operators could not be performed on the encrypted database so that large volumes of data had to be shipped from the database server to the middleware for post-processing. In contrast, running TPC-H queries on a database that was encrypted using Det-AES had significant overhead for most queries: Half of the queries timed out and took longer than 30 minutes even on a fairly small database with scaling factor 10 (i.e., 10 GB of raw data). The performance of the probabilistic POP variant that we studied, was somewhere in between Det-POP and Det-AES, but it only timed out for a single query for which *plain* and Det-POP also timed out. Furthermore, the experiments showed that the performance of POP is not sensitive to its parameter, U, the number of partitions (Runs). This result is encouraging because large U values provide better security.

References

[1] R. Agrawal and R. Srikant. Privacy-preserving data mining. *SIGMOD*, 2000.

[2] A. Arasu and S. Blanas. Orthogonal security with cipherbase. In *CIDR*, 2013.

[3] S. Bajaj and R. Sion. TrustedDB: a trusted hardware based database with privacy and data confidentiality. In *SIGMOD*, 2011.

[4] D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. Monotone minimal perfect hashing: searching a sorted table with o(1) accesses. SODA, 2009.

[5] M. Bellare and P. Rogaway. Introduction to modern cryptography, 2005.

[6] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. Order-preserving symmetric encryption. In *EUROCRYPT*, 2009.

[7] A. Boldyreva, N. Chenette, and A. O'Neill. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *CRYPTO*, 2011.

[8] E. Damiani, S. De Capitani Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati. Balancing confidentiality and efficiency in untrusted relational DBMSs. In *CCS*, 2003.

[9] C. Gentry. *A fully Homomorphic Encryption Scheme*. PhD thesis, Stanford University, 2009.

[10] H. Hacigümüş, B. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *SIGMOD*, 2002.

[11] Y. Hong, X. He, J. Vaidya, N. Adam, and V. Atluri. Effective anonymization of query logs. In *CIKM*, 2009.

[12] S. Hsueh. Database encryption in SQL Server 2008 Enterprise Edition. *Microsoft White Paper*, 2008.

[13] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. 2007.

[14] A. Nanda. Transparent data encryption. *Oracle Magazine*, 2005.

[15] R. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *SOSP*, 2011.

[16] R. Sion. Secure data outsourcing. In *VLDB*, 2007.

[17] N. Smart and V. F. Fully homomorphic encryption with relatively small key and ciphertext sizes. Cryptology ePrint Archive, Report 2009/571, 2009.

[18] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. Cryptology ePrint Archive, Report 2009/616, 2009.