DISS. ETH NO. 22450

Encrypting Databases in the Cloud Threats and Solutions

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH

(Dr. sc. ETH Zurich)

presented by

TAHMINEH SANAMRAD

M.Sc. in Computer Science, ETH Zurich

born on 14.01.1983

citizen of
Iran

accepted on the recommendation of

Prof. Dr. Donald Kossmann (ETH Zürich, Switzerland), examiner
Prof. Dr. Gustavo Alonso (ETH Zürich, Switzerland), co-examiner
Prof. Dr. Srdjan Capkun (ETH Zürich, Switzerland), co-examiner
Prof. Dr. Johannes Gehrke (Cornell University, NY, USA), co-examiner

2014

Typeset with LaTeX
Printer: ETH Zurich
Printed in Switzerland

Abstract

With the growing importance of cloud computing, database encryption has become a critical technology to protect data against honest-but-curious attackers. Our goal is to encrypt the data in such a way that it remains protected against powerful attackers and at the same time achieve good performance by processing queries in the cloud without decrypting the data. *Order-Preserving Encryption (OPE)* is one of the most attractive techniques for database encryption since it allows the execution of range and rank queries on encrypted data. On the other hand, people are reluctant to use OPE-based techniques in practice because of their vulnerability against attackers with knowledge of the domain and its frequency distribution.

This dissertation makes three important contributions. First, it formalizes a set of real-world attacker scenarios on encrypted databases, namely *domain attack*, *frequency attack* and *query log attack*. Query log attack refers to the inference of secrets by observing the (encrypted) queries submitted to the encrypted database. To this end, a number of encryption techniques have been developed and studied in literature. Unfortunately, most of these schemes have ignored an important threat called query log attack. Second, based on this formalization, it shows how these attacks impact the security of an important class of encryption techniques, namely OPE. Third, it explores new encryption techniques called *Probabilistic Order-Preserving Encryption (Prob-OPE)* and *Randomly Partitioned Encryption (RPE)* which are proven to be resilient against the attacker scenarios mentioned previously. These encryption techniques address the need to encrypt databases in the cloud and at the same time execute complex SQL queries efficiently. Prob-OPE and RPE can be configured to meet different privacy and performance requirements. Privacy and performance experiments conducted using the TPC-H queries show that Prob-OPE and RPE make it indeed possible to achieve a higher level of privacy compared to the state of the art with low performance overheads.

Kurzfassung

Die Datenverarbeitung durch Internetdienste (engl. cloud computing) ist heute weit verbreitet da sie zuverlässig, skalierbar und überall verfügbar ist. In diesem Kontext stellen sich neue Herausforderung an die Datensicherheit mittels Verschlüsselung. Daten sollen verschlüsselt werden, so dass sie auch vor starken Angriffen geschützt sind. Gleichzeitig muss es möglich sein die Daten ohne allzu grosse Effizienzverluste zu verarbeiten. Ortnungserhaltende Verschlüsselungsverfahren (engl. Order-Preserving Encryption, kurz OPE) gehören zu den attraktiveren Ansätzen, da mit ihnen Bereichs- und Ortnungsanfragen effizient abgearbeitet werden können. Leider sind sie unteranderem anfällig für Angriffe wenn der Bereich der Datenwerte, z.B. Städtenamen, oder deren relative Häufigkeit bekannt ist.

In dieser Doktorarbeit formalisieren wir als erstes die für das OPE-Verfahren praxisrelevanten Angriffsarten wie z.B. den vorherig erwähntenn Angriff über den Datenbereich (engl. domain attack), die Häufigkeitverteilung der Werte (engl. frequency attack) sowie den Logbuchangriff, welcher auf der Analyse der (verschlüsselten) Datenbankanfragen basiert (engl. query log attack). Gerade der Logbuchangriff wurde bisher in der Literatur selten behandelt, er stellt aber ein in der Praxis durchaus denkbares Szenario dar.

Darauf aufbauend wird dann die Sicherheit des OPE-Verfahrens untersucht. Die daraus gewonnenen Einsichten erlauben es uns dann verbesserte Verfahren zu entwickeln mit signifikant erhöhter Widerstandskraft. Insbesondere präsentieren wir zum erstenmal eine probabilistische Erweiterung namens Prob-OPE (engl. Probabilistic Order-Preserving Encryption) und eine Erweiterung basierend auf einer zufälligen Unterteilung des Datenbereichs mit dem Kürzel RPE (engl. Randomly Partitioned Encryption). Beide Verfahren sind so desgined, dass sie erheblich sicherer sind ohne viel Effizienz zu verlieren. Durch eine geeignete Wahl der Parameter, zum Beispiel die Anzahl der Partitionierung beim RPE-Verfahren, kann in der Praxis graduell Sicherheit gegenüber Effizienz balanciert werden. Wir eruieren die Performance unserer neuen Verfahren basierend auf dem TPC-H Benchmark und zeigen mittels mathemtatischer Methoden wie stark sich die Datensicherheit erhöht. Es zeigt sich, dass Prob-OPE und RPE es ermöglichen die Datensicherheit zu erhöhen ohne einen grossen Performanceverlust in Kauf nehmen zu müssen.

Acknowledgments

Foremost, I would like to express my sincere gratitude to my advisor, Prof. Kossmann, for the continuous support of my Ph.D study and research, for his enthusiasm and faith in me. His guidance helped me throughout these years that I was working as a researcher here at ETH Zurich.

Besides my advisor, I would like to thank the rest of my thesis committee: Prof. Gustavo Alonso, Prof. Srdjan Capkun, and Prof. Johannes Gehrke, for their encouragement and helpful comments. I am grateful to Prof. Peter Fischer, Prof. Nesime Tatbul, Dr. Ramarthanam Venkatesan and Dr. Kyumars Sheykh Esmaili whom I had the honor of collaboration during my Ph.D. My sincere thanks also goes to Mr. Nipun Agarwal for offering me the summer internship opportunity at Oracle Research Labs and Dr. Kevin Moore for leading me on diverse exciting projects there.

In particular I want to thank my colleagues: Lucas Braun for the sleepless nights we were working together before deadlines, Christian Badertscher and Besmira Nushi for their insightful comments on my work. I thank all my friends and fellow labmates, who have made the past four and half years the most fun and diverse years of my life. Special thanks also goes to our helpful administrative staff: Jena Bakula, Simonetta Zysset, Eva Ruiz Muller and Karel Kudlacek.

I would like to thank my loving mother for supporting me spiritually and emotionally throughout my life, my late grandparents, may they rest in peace, who made it possible for me to get this far in my career, and last but not least, my amazing husband, Martin, who supported me both scientifically and emotionally through my ups and downs, who believed in me and encouraged me to reach beyond what I have expected of myself. I am also grateful to his family who treated me like one of their own, making me feel at home here while I was away from my own family.

Contents

1 Introduction

According to the trade press, cloud computing is the next big thing. Cloud computing promises reduced cost, flexibility, improved time to market, 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 issue that seems to be able to stop cloud computing are security concerns [20]. One of the most prominent issues for cloud computing is *privacy*; that is, the protection of sensitive data from unauthorized users. Privacy is defined as having *knowledge* about what is happening to one's data and being able to fully *control* it. However, by using cloud data services, the control is outsourced to the cloud provider.

The events that motivated this work were privacy violations in a *private* cloud. Between 2008 and 2010, database administrators of several Swiss and Liechtenstein banks sold CDs with customer information to French and German tax authorities. These events triggered a number of political discussions between the four countries and, maybe more importantly, these events were a major embarrassment for the Liechtenstein and Swiss financial marketplaces.

Topping all the news in the year 2013 is the U.S. National Security Agency (NSA) scandal. The vast scale of online surveillance carried out by NSA, which was revealed by Edward Snowden, is changing how businesses store commercially sensitive data, with potentially dramatic consequences for the future of the Internet, according to [64]. The survey of 1,000 information and communications technology decision-makers from France, Germany, Hong Kong, the UK and the US was carried out by NTT Communications. It found that, following the Snowden revelations, almost 90% of the survey

participants had changed the way they use the cloud. The study also found that almost *a third* of those questioned were moving their company's data to locations where they "know it will be safe", and 16% said they had delayed or canceled their contracts with cloud service providers [64].

In this thesis the fundamental assumption is that the client (the cloud user) is trusted, whereas the cloud service provider is untrusted, as shown in Figure 1.1. Consequently, the user's data needs to be protected from whoever has access to the persistent or non-persistent storage on the cloud, e.g., a system administrator or a hacker that has found his way into the cloud. Another important assumption is that the untrusted party is passively malicious, i.e., an *honest-but-curious* adversary model that runs the programs and algorithms correctly, but might look at the information passed between entities.

In general, there are two ways to protect the data from the untrusted cloud service provider: First approach is to use software-based techniques such as *Encryption*. Second approach is to use *Tamper-Proof Hardware* for storing and processing user's sensitive data in the cloud. However, the latter approach requires the cloud service provider to accommodate the data owner's trusted hardware in the cloud. In this thesis we study the former approach, which is encryption, and can be accomplished independent of the cloud service provider.

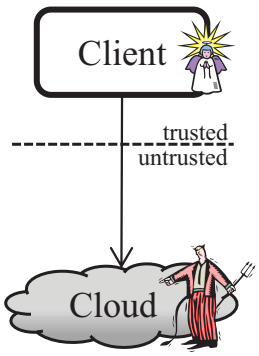


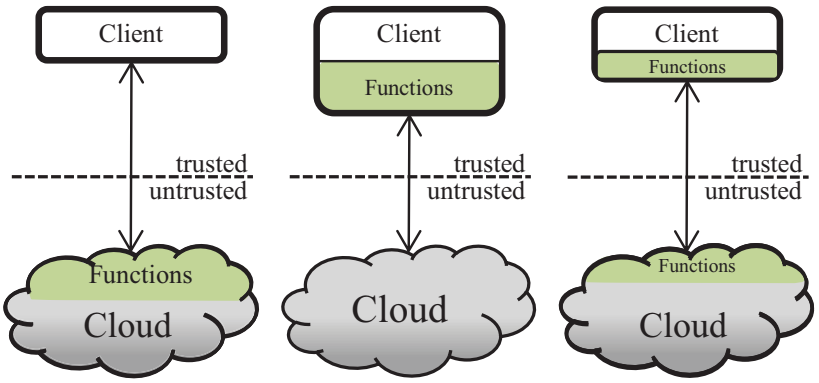
Figure 1.1: Basic Adversary Model

As mentioned earlier, encryption is a possible way to protect data against honest-but-curious adversaries. However, there is no free lunch when using encryption. Encrypting data increases the data processing overhead and thereby affects the latency requirements of a system. The spectrum of available encryption schemes ranges from semantically secure but highly inefficient encryption schemes to weak but efficient encryption schemes. Once the data is encrypted with a semantically secure encryption scheme, only a limited set of functions can be applied on the encrypted data. As a result, all the processing capabilities of the cloud are of no use. On the other hand, a weak encryption scheme, which preserves certain functionalities, leaks information about its underlying plaintext and is vulnerable to certain attacker models. There are two approaches to reach a desirable security/performance trade-off between the two ends of the encryption spectrum. The first approach is to start from a strong encryption scheme and return certain functions to it; for example Homomorphic encryption schemes are semantically secure and support addition and/or multiplication on encrypted data [27]. The second approach is to start from a weak encryption scheme that supports certain functionalities and make it stronger. In this thesis we take the latter

approach, i.e. we take a weak encryption scheme and reinforce its security with minimal loss in functionality and performance.

1.1 Problem Statement

The goal of this thesis is to exploit the *strong and flexible processing capabilities* of the cloud, and at the same time *preserve the privacy* of the client. Nevertheless, privacy comes at the cost of performance and functionality. The state of the art among cloud service providers is shown in Figure 1.2a where no security is used. As a result, all the application functionality is handled by the cloud. At the other end, as shown in Figure 1.2b, user's can upload encrypted data to the cloud, e.g., when using Dropbox [1, 4]. Consequently, all the functions have to be implemented on the client side. The main problem this thesis tries to solve is how to split the workload between the cloud and the client, as shown in Figure 1.2c, such that both the security and performance requirements of the client are met.



(a) No Security (b) Full Security (c) Balanced Security

Figure 1.2: Possible Functionality Distributions

Here is a detailed list of problems encountered while solving the security, functionality, and performance trade offs for the cloud data services:

- **Security/Performance Trade off.** With the current advances on mobile devices and network technologies, clients access their desired cloud data services from anywhere. Client's devices are limited by size, power and memory. Hence, the first challenge is how to offload the heavy data processing tasks from the client to the cloud while preserving the privacy of the client.
- **Application Specific Adversary Models.** Only a limited set of applications have been treated formally by the cryptography community so far. With the emergence

of cloud computing and big data, it is time to define new adversary models that can address the security requirements of applications in these new realms. Specifically, in this thesis we are looking at relational databases residing in a private or public cloud such as Microsoft Azure [3] and Google Cloud SQL [2].

- **Security Measures and Notions.** There are several metrics to measure the performance of a system, e.g., response time, throughput, and so on. When adding the security dimension to a system, it is important to define metrics to measure the security based on the possible adversarial models. Measuring security alongside other known performance measures gives the user a clear picture of the performance/security trade off.

- **Query Processing Support.** Database queries have different properties. The challenge is to find an encryption scheme that can support query processing for different classes of queries. In general, our goal is to find a middle ground between the functions we can support and the functions we need to give up in order to achieve our high performance/high security goal for cloud databases.

- **Performance Requirements.** Users have different performance requirements, therefore an encryption scheme needs to be tunable, so that it can adjust to the users performance requirements.

1.2 Contributions

Cloud computing providers offer their services according to several fundamental models: *infrastructure as a service (IaaS)*, *platform as a service (PaaS)*, and *software as a service (SaaS)* where IaaS is the most basic and each higher model abstracts from the details of the lower models [67].

This thesis investigates the security of the databases in the cloud from the platform as a service (PaaS) layer. Our concrete contributions are as follows:

- **Formalizing New Adversary Models [31,51,53,54,56].** These models are driven from real-world use cases from the financial industry. The basic assumption, governing all these attacker models, is that the *cloud is untrusted*. We particularly look at the following three attacker models:

  - *Domain Attack.* In this attacker scenario the adversary not only has access to the encrypted data, but also knows about the plaintext domain of the values in the encrypted database. In our initial *bank usecase*, the database administrator of the Swiss and Liechtenstein banks had a list of all customers and only needed to retrieve other relevant account information for selected customers.

  - *Frequency Attack.* In this attacker scenario the adversary not only has access to the encrypted data, but also knows about the plaintext domain of the values and its frequency distribution. Although frequency attacks are a well-known cryptanalysis method, in this thesis we define the frequency attack specifically for the data stored in a database.
  - *Query Log Attack.* In this attacker scenario the adversary not only has access to the encrypted data, but also knows about the plaintext domain of the values, its frequency distribution and additionally has access to the database *query logs*.

- **New Security Measures and Definitions [51,53,54,56].** Defining new adversary models, requires new security notions and measures to be defined to capture the probability distribution of an adversary attempting a certain attack:

  - *Rank One-Wayness.* A measure that captures the probability of success of an adversary in correlating the ordering rank of a ciphertext to its underlying plaintext.
  - *Frequency One-Wayness.* A measure that captures the probability of success of an adversary in correlating the frequency of a ciphertext to its underlying plaintext.
  - *Query Indistinguishability.* A security notion that detects queries which reveal secrets about the underlying encryption scheme not revealed by the encrypted data.

- **Security Analysis of the State of the Art [51, 53].** We revisit the state of the art in database encryption and analyze their security towards the newly defined adversary models.

- **Probabilistic Order Preserving Encryption (Prob-OPE) [51].** As opposed to OPE, this scheme creates different ciphertexts for the same plaintext while still preserving the order. We introduce Prob-OPE as the probabilistic variant of OPE to reinforce its against attackers who exploit the frequency distribution of the encrypted data to infer useful information out of it.

- **Randomly Partitioned Encryption (RPE) [51].** After thoroughly analyzing the state of the art in the realm of database encryption, we introduce a new encryption scheme, called *Randomly Partitioned Encryption*. The basic idea is to randomly partition the domain into so called *Runs* in a fine-grained manner, i.e. each plaintext value is randomly sent to a different run. Each run is then encrypted using a weak encryption scheme, e.g. an order preserving encryption scheme [6,14,15,41,48,69,71]. We present two variants of *Randomly Partitioned Encryption*:

  - **Deterministic Randomly Partitioned Encryption (Det-RPE).** This scheme deterministically assigns the plaintext values to *Runs*. Additionally, a deterministic encryption scheme is used to encrypt the values within a *Run*. As a result, the attacker has no knowledge of which plaintext value is encrypted in which *Run*. Det-RPE distorts the total order in a one-way fashion making it impossible for an adversary to reconstruct the total order out of multiple partial orders even if having domain knowledge. However, a deterministic scheme is not a safe option for skewed frequencies.
  - **Probabilistic Randomly Partitioned Encryption (Prob-RPE).** This encryption scheme builds on Det-RPE, and additionally achieves probabilistic encryption in two ways: (a) by assigning the same plaintext value to different ciphertexts within a *Run*, i.e. composing RPE with *Probabilistic OPE*, and (b) by assigning the same plaintext value to different *Runs*. In a second step, each *Run* is encrypted using an order-reserving encryption technique which supports efficient query processing on that *Run*.

- **RPE Security [51].** Even though a weak encryption scheme is used to encrypt each *Run*, RPE can provably achieve high levels of security. Intuitively, the reason is that there are exponentially many ways to re-assemble the original column from a set of random partitions. Furthermore, RPE can be configured to be probabilistic even if the underlying weak encryption scheme is deterministic. Specifically, we show that RPE is able to address our newly introduced and common attack scenarios (domain, frequency, and query log attack) that weak encryption schemes such as order-preserving encryption (e.g., [6, 14]) and deterministic encryption schemes (e.g., AES in ECB mode) are not able to address.

- **RPE Performance [52].** Modern database systems process queries in a divide and conquer way, thereby partitioning the data and processing it partition by partition. As a result, the additional layer of partitioning provided by the RPE *Runs* does not hurt performance as long as each *Run* can be processed efficiently. RPE achieves this by using a weak encryption scheme such as order-preserving encryption for each *Run*.

- **RPE Functionality [52].** In terms of query functionality, RPE is as good as its underlying weak encryption scheme. If an order-preserving encryption scheme is used, then RPE can effectively process many classes of queries including queries with range predicates, wild cards (i.e., *LIKE* predicates), equi-joins, and group-by/aggregation. One important advantage of RPE is that it supports updates even in situations in which a traditional weak encryption scheme would not; for instance, a probabilistic order-preserving encryption scheme might exhaust the available codes whereas RPE never hits such a dead end because it can add new

Runs dynamically. Furthermore, some of the optimizations used in systems like Monomi [66] preclude updates whereas any kind of update can be applied to data that was encrypted using RPE.

- **Fixed Range Query Rewrite [51, 53].** The encryption schemes that have been used for database encryption so far are oblivious to the fact that if a database is compromised, its query logs are compromised as well. Query log attack shows that a lot of encryption schemes do leak information about their secret parameters once the query logs are hijacked. To fix this problem for Prob-OPE and RPE, the queries need to be rewritten differently. In the *fixed range query rewrite* method the overlapping ranges are avoided, and range queries will always return ranges with a fixed size. This way the probability of success for a query log adversary can be tuned according to the negligibility requirements of the data owner.

1.3 Overview

This dissertation is structured as follows:

Chapter 2 describes the abstract adversary model that is assumed throughout this thesis. Moreover, it introduces the crucial components of a secure database system and discusses the possible architectures that can realize security for cloud databases. Each of the architectures are discussed and compared in detail. Nevertheless, the contribution of this thesis is totally independent of the underlying architecture.

Chapter 3 introduces the state of the art encryption schemes that are favored for database applications. Reading this chapter gives the reader a common ground on database encryption. This chapter explains what encryption properties can be exploited by the SQL operators, how SQL queries can be rewritten, and what security guarantees are given by each of these encryption schemes in a nutshell.

Chapter 4 presents three adversary models, namely domain attack, frequency attack, and query log attack which are driven from our use cases from the financial industry. We use a game-based approach to precisely define each adversary. For each scenario, we define a security notion to capture the probability distribution of an adversary succeeding. Later in this thesis we evaluate each presented encryption scheme towards the attacker models in this chapter and derive their probability distributions. The adversary models introduced in this chapter are independent of the underlying encryption mechanism.

Chapter 5 elaborates on query log attack. In this thesis for the first time we define a security notion based on indistinguishability properties of the queries. So far all the encryption schemes that have been presented and used in a database application ignored the fact that queries can leak a great deal about the underlying encryption scheme. In this chapter we introduce a framework to evaluate the queries and measure how much

they reveal about the underlying encryption scheme. This framework is independent of the underlying encryption scheme, and can be used for any kind of queries (not only SQL).

Chapter 6 revisits the constructions of order-preserving and modular oder-preserving encryption from [14, 15]. This chapter shows why they do not fulfill the security requirements defined in Chapter 4 and need to be reinforced or replaced. The encryption schemes introduced in this thesis are composed of the constructions given in this chapter.

Chapter 7 describes the probabilistic order-preserving encryption (Prob-OPE) scheme. This encryption scheme extends the order-preserving encryption (OPE) scheme from Chapter 6 to create a probabilistic variant of OPE that can give stronger security guarantees. In this chapter we discuss the database functionality, analyze the security, and show performance results of Prob-OPE compared to OPE.

Chapter 8 describes the randomly partitioned encryption (RPE) scheme. This encryption scheme distorts the order relationship that exists in order-preserving encryption schemes, revisited in Chapter 6, to create a partially order-preserving encryption that can give stronger security guarantees. In this chapter we discuss the database functionality, analyze the security, and show performance results of RPE compared to other encryption variants introduced earlier.

Chapter 9 describes the probabilistic randomly partitioned encryption (Prob-RPE) scheme. This encryption scheme not only distorts the order relationship which exists in OPE schemes, but also flattens the frequency distribution yielding a probabilistic and partially order-preserving encryption that can give stronger security guarantees. In this chapter we discuss the database functionality, analyze the security, and show performance results of the Prob-RPE compared to other encryption variants presented earlier.

Chapter 10 concludes this thesis by asserting the main results and discusses the venues for future work.

2 Preliminaries

This chapter first describes the adversary model and the respective assumptions about the adversary. An adversary is a malicious entity whose aim is to prevent the users of a system from achieving their goal (primarily privacy, integrity, and availability of data). Throughout this thesis we assume to have a passive adversary whose goal is to discover secret data stored on the cloud. The client on the other hand, is considered to be a trusted entity who remotely connects to a relational database in the cloud and submits SQL queries. In addition to the trusted client and untrusted cloud, a third component is required to enable query processing on encrypted data. This component needs to be trusted and is responsible for executing various cryptographic and database functionalities depending on the encryption scheme that was used to encrypt the data in the cloud. In this chapter we discuss various architectures in which these three components, namely the client, the trusted component and the cloud database interact. We argue about the advantages and disadvantages of each approach in detail.

2.1 Adversary Model

Throughout this thesis we assume the adversary is honest-but-curious, i.e. it passively observes the data but does not actively corrupt it. Additionally, we assume the adversary has full access to the cloud, i.e. either a system/database administrator, a government employee or a hacker that has managed to intrude into the system. As a result not only the persistent storage in the cloud cannot be trusted but also the main memory and caches can be read and exploited.

With this definition we exclude the cases where the adversary colludes with the client or has compromised the trusted component. We also assume that all the communication channels are encrypted and well-secured.

2.2 Architecture

As already mentioned, one possible way to enforce security on an outsourced database is to encrypt its data. In order to enable query processing on encrypted data, there are a few possible architectures. These architectures share the same principle, i.e. utilizing a *Trusted Component* to execute the confidential tasks. The trusted component is assumed to be *thin* and *trusted*. *Thin* means that not much computational power is needed to implement the trusted component, because the heavy weight-lifting of executing joins, aggregates, etc. is expected to be carried out in the cloud. The trusted component encapsulates security operations and makes them transparent to the *Application Developer* or the *Client*. The trusted component has three significant responsibilities:

1. **Key Management:** The trusted component stores and manages the keys required to decrypt the data.

2. **Query Rewrite:** The queries written by the client contain plain text. Therefore, the plain text in the query has to be encrypted by the trusted component. Moreover, depending to the encryption scheme used to encrypt the data in the cloud, the queries written by the client might need to be restructured.

3. **Decryption and Post-Processing:** Depending on the encryption scheme, the trusted component needs to decrypt and possibly post-process the encrypted query results.

There are different architectures that support the encryption of data in the cloud. In the remainder of this section we will describe each architecture and list its advantages and disadvantages. Before continuing, we introduce the notation used to explain the various architectures to implement a secure database in the cloud.

Notation. Let us denote the plain database as DB and the encrypted database as DB'. The trusted component is abbreviated as TC. A plaintext query sent by the client is denoted as q whereas the rewritten query by the trusted component is denoted as q'. Symmetrically, the encrypted result retrieved from the encrypted database is denoted as r', and the decrypted and final result that is given back to the client in answer to q is denoted as r.

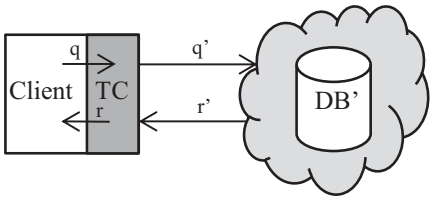


Figure 2.1: Client-Side Security

2.2.1 Client-Side Security

The client is assumed to be a trusted entity in this thesis, thus its machine can be used to implement the security functions. As shown in Figure 2.1, in this approach the security functions are implemented on the client-side.

Here is a list of advantages, marked with a *checkmark*, and disadvantages, marked with a *cross* for implementing the security on the client-side.

✓ *No Additional Trust Component:* Adding an extra component to the system not only causes an overhead in the performance but also adds the attacker surface. In client-side security only the client needs to be trusted which complies with our initial assumption.

✗ *No Portability:* In order to access the encrypted data stored in the database in the cloud, the client needs to install and maintain a trusted component on all its machines.

✗ *Decentralized Access Control and Key Management:* Since the security policies and keys are locally stored on client's machine, updating those needs to be applied to each trusted component on each device separately.

✗ *Decentralized Maintenance:* The trusted component's code on the client machine needs fixes and updates. It is the client's task to launch updates and fixes on every single device.

2.2.2 Security Middleware

The next approach, as shown in Figure 2.2, is to have a security middleware between the client and the cloud database server [22,29,31,47,51,55]. This way the middleware serves as a proxy between the client and the cloud. The security middleware offloads key management, query rewrite and encryption/decryption tasks from the client. At the same time the security middleware stays thin, because the final goal is to perform the heavy processing tasks in the cloud.

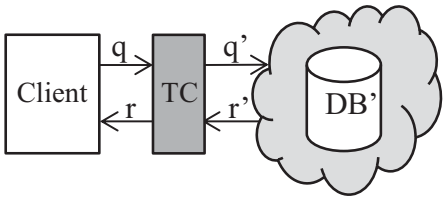


Figure 2.2: Security Middleware Architecture

✗ *Another Trust Component:* Adding a middleware means that there is another component that needs to be trusted by the client. Also an additional component means an additional point of attack.

✓ *Portability:* In case of having a security middleware, each client's device can simply connect through the middleware's trusted component to the cloud, therefore solving the portability problem.

✓ *Centralized Access Control and Key Management:* The security middleware is in charge of the keys and the security policies, thus every change will be applied once on middleware, whereas in the client-side security every update had to propagated to the affected clients.

✓ *Centralized Maintenance:* The code on the security middleware is also prone to fixes and updates, but there is only one device that has to undergo such troubles.

2.2.3 Tamper-proof Embedded Hardware

The third architecture, as shown in Figure 2.3, is to embed a tamper-proof hardware in the cloud as in [9, 10]. The query is then broken into a confidential part that is executed in the tamper-proof hardware and non-confidential part that is executed openly in the cloud. The task of key management, query rewrite and partially encryption/decryption is performed in the tamper-proof hardware. However, the client also needs to perform a few security related tasks such as query rewrite and encryption/decryption. That is why in Figure 2.3 there are two trusted components namely *TC1* which resides on the client's machine and *TC2* which resides in the cloud.

✗ *Agreement with Cloud Service Provider:* So far we have assumed that cloud is untrusted. In this architecture we need to convince the cloud service provider to put our approved tamper-proof hardware in their cloud. Reaching such an agreement is not always feasible.

✗ *Result Leakage:* The queries submitted to the database do not correspond to the final result that is returned to the client. This is because the tamper-proof hardware has obtained the final result after performing several iterations of rewriting

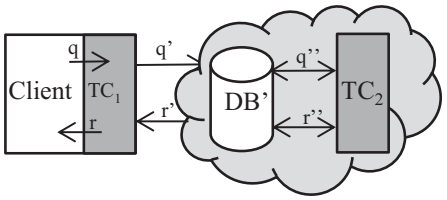


Figure 2.3: Extended Client-server Database Architecture

the queries and post-processing the intermediate query results. This mismatch between the query submitted and the returned result can leak information about the data and needs to be further explored.

✗ *Complex Task of Breaking Queries:* The queries received by the cloud database are broken into smaller parts in several iterations which can be considered a difficult and intelligent task.

✓ *Maximize Processing on the Cloud:* Since the goal is to maximize the amount of query processing carried out in the cloud, such a tamper-proof hardware can execute insensitive tasks on the cloud divided in a fine-grained manner, thereby minimizing the post-processing time.

✓ *Minimize Network Cost:* Post-processing on encrypted data is handled by the embedded hardware in the cloud. The encrypted result is then shipped to the client. Therefore, this approach saves a great deal of network bandwidth by minimizing the amount of data that needs to be shipped to the client.

The contribution of this thesis is not limited to any specific architecture discussed in this chapter. The performance results generated in the experiment sections however can be reproduced by employing the client-side security architecture or in a security middleware set up. We did not test the performance of our encryption schemes using embedded hardware.

3
Database Encryption

Encryption is a possible way to fulfill the confidentiality requirement of data that is being stored in an untrusted database. SQL queries are used to retrieve data from a relational database. Each SQL-operator implements a different functionality. In this chapter we revisit the properties of the state of the art encryption schemes that can be exploited to process SQL queries on encrypted data. To clarify how queries can be processed on encrypted data, we show several examples and introduce a set of rules to rewrite queries in order to be processed on an encrypted database given a specific encryption scheme.

3.1 Properties of Encryption Methods

The properties of an encryption function determine which kinds of queries can be executed on the encrypted data without decrypting them and, consequently, how queries must be rewritten by the trusted component. For SQL databases, there are three encryption properties that are useful: *determinism*, *order preservation*, and *homomorphism*. In this section we define each property and clarify with examples why these properties play a crucial role in enabling query processing on encrypted data. As a running example, assume we have a `customer` and an `order` table in our database. Table 3.1 shows the `name` column of joining customer and order table, $customer \bowtie order$. Customer name is a sensitive field and needs to be kept confidential using encryption. There are several ways to encrypt the `name` column. In the remainder of this section we will see what consequences does an encryption scheme have on query processing on the encrypted

Name	Name(DET)	Name(OPE)	Name (PROB)
Benzema	Pxf34	1354	Cx5vf
Messi	Xv430	2564	H4op1
Messi	Xv430	2564	A43fx
Neymar	Io987	3129	G09pl
Ronaldo	LP21f	4980	klo12

Table 3.1: Different encryptions of the `name` column.

Relational Algebra Operator	Symbol
Selection	σ
Join	$\bowtie$
Aggregation	Γ
Distinct	$\mathcal{D}$
Order By	$\mathcal{O}$

Table 3.2: Extended Relational Algebra Operators.

`name` column.

Notation. Let $m, n, l \in \mathbb{N}\ s.t.\ m < n$. Let x_i be an element of a plaintext space, $\mathcal{X} = \{0,1\}^m$, that contains confidential information. Let y_i be an element in the ciphertext space, $\mathcal{Y} = \{0,1\}^n$. Let K be a key that is selected uniformly at random from the key space, $\mathcal{K} = \{0,1\}^l$. Let $\mathcal{E}nc_K^{\text{ES}}(x_i)$ be the encryption function from encryption scheme, $\mathcal{ES}$, that maps the plaintext value, x_i to a set of ciphertexts, given a secret key, K. The function which rewrites plaintext queries to be processable on encrypted data is denoted as $\mathcal{QRF}$.

Extended Relational Algebra. The query rewrite is one of the main tasks of a trusted component and an essential part of this thesis. To formalize the rules that have been used to rewrite queries for different encryption scheme, we use an extended relational algebra syntax as shown in table 3.2.

Naming Convention. In the query rewrite examples in this chapter, the tables in the encrypted database are denoted as *table_enc*, and the encrypted columns are denoted as *column_enc*.

3.1.1 Deterministic Encryption Schemes

A deterministic encryption scheme (as opposed to a probabilistic encryption scheme) is a cryptosystem which always produces the same ciphertext for a given plaintext and key.

Construction 1. *A Deterministic Encryption scheme,* $DET = (\mathcal{K}, \mathcal{E}nc, \mathcal{D}ec)$ *is defined as follows:*

- *Algorithm* $\mathcal{K}$ *returns a random key,* $K \xleftarrow{\$} Keys$.
- $\mathcal{E}nc$ *takes* K *and* x *as input and returns* $y = \mathcal{E}nc_K(x)$ $s.t.$ $x_i = x_j \iff \mathcal{E}nc_K(x_i) = \mathcal{E}nc_K(x_j) \wedge \|\mathcal{E}nc(x_i)\| = 1$.
- $\mathcal{D}ec$ *takes* K *and* y *as input and returns* $x = \mathcal{D}ec_K(y)$.

Table 3.1 gives an example of a set of customer names encrypted with DET.

Query Rewrite

A deterministic encryption scheme is important to implement equality predicates. For example given Table 3.1, the client submits a plaintext query to retrieve "Messi" from the `customer` table:

```
SELECT name FROM customer WHERE name = 'Messi'
```

If the name column is encrypted using a deterministic encryption scheme (DET), the query can be rewritten such that it retrieves the encryption of "Messi" which is $\mathcal{E}nc_K^{\text{DET}}('Messi') = Xv430$:

```
SELECT name_enc FROM customer_enc WHERE name_enc = 'Xv430'
```

In general the following SQL operators are supported by any deterministic encryption scheme, and the plaintext queries are rewritten according to the following rules:

- Equality Predicate:

$$\mathcal{QRF}_{\text{DET}}(\sigma_{column=x}(table)) = \sigma_{column_{enc}=\mathcal{E}nc_K^{\text{DET}}(x)}(table_{enc}) \tag{3.1}$$

- Inequality Predicate:

$$\mathcal{QRF}_{\text{DET}}(\sigma_{name!=x}(table)) = \sigma_{column_{enc}!=\mathcal{E}nc_K^{\text{DET}}(x)}(table_{enc}) \tag{3.2}$$

- IN Predicate:

$$\mathcal{QRF}_{\text{DET}}(\sigma_{column\ IN\ (w,x)}(table)) = \sigma_{column_{enc}\ IN\ (\mathcal{E}nc_K^{\text{DET}}(w),\mathcal{E}nc_K^{\text{DET}}(x))}(table_{enc}) \tag{3.3}$$

- EQUI-JOIN:

$$\mathcal{QRF}_{\text{DET}}(table1 \bowtie_{table1.column=table2.column} table2) = (table1_{enc} \bowtie_{table1_{enc}.column_{enc}=table2_{enc}.column_{enc}} table2_{enc}) \tag{3.4}$$

- DISTINCT:

$$\mathcal{QRF}_{\text{DET}}(\mathcal{D}_{column}(table)) = \mathcal{D}_{column_{enc}}(table_{enc}) \tag{3.5}$$

Summary of Security Analysis

The property of indistinguishability under chosen plaintext attack is considered a basic requirement for most provably secure *public key cryptosystems*. If a cryptosystem possesses the property of indistinguishability, then an adversary will be unable to distinguish pairs of ciphertexts based on the message they encrypt. Deterministic encryption schemes do not satisfy the indistinguishability property, therefore they are not semantically secure. Examples of deterministic encryption algorithms include RSA cryptosystem (without encryption padding), and many block ciphers (e.g AES or DES) when used in ECB mode.

A deterministic encryption scheme also leaks the frequency distribution of its underlying data. This weakness allows the adversary to perform statistical analysis on the encrypted database. For instance, in Table 3.1 by looking at *DET* column, it is obvious that "Messi" has placed more orders than the others. Of course, if the underlying data has uniform frequency distribution the frequency leakage of a deterministic encryption scheme will not impose any threat. In the next chapter there will be more discussion on *frequency attacks* on encrypted databases.

3.1.2 Order-Preserving Encryption Schemes

An order preserving encryption (OPE) scheme is a deterministic symmetric encryption scheme whose encryption algorithm produces ciphertexts that preserve numerical ordering of the plaintexts. This property makes OPE very attractive for database applications, since it allows efficient range and rank query processing on encrypted data. OPE was first proposed in the database community by Agrawal et al. [6]. However, the first formal cryptographic treatment of OPE did not appear until recently by Boldyreva et al. in [14].

Construction 2. *An Order-Preserving Encryption scheme,* $OPE = (\mathcal{K}, \mathcal{E}nc, \mathcal{D}ec)$ *is defined as follows:*

- *Algorithm* $\mathcal{K}$ *returns a random key,* $K \xleftarrow{\$} Keys$.
- $\mathcal{E}nc$ *takes* K *and* x *as input and returns* $y = \mathcal{E}nc_K(x)$ $s.t.$ $x_i < x_j \iff \mathcal{E}nc_K(x_i) < \mathcal{E}nc_K(x_j)$.
- $\mathcal{D}ec$ *takes* K *and* y *as input and returns* $x = \mathcal{D}ec_K(y)$.

Table 3.1 gives an example of a set of customer names encrypted with OPE.

Query Rewrite

An order preserving encryption scheme is important to implement range and rank predicates. For example given Table 3.1, the client submits a plaintext query to retrieve all the names which are lexicographically greater than "Messi" from the `customer` table:

```
SELECT name FROM customer WHERE name > 'Messi'
```

If the name column is encrypted using an order preserving encryption scheme (OPE), the query can be rewritten such that it retrieves all the ciphertexts whose encryption is bigger than the encryption of "Messi":

```
SELECT name_enc FROM customer_enc WHERE name_enc > '2564'
```

In general the following SQL operators are supported by any order preserving encryption scheme, and the plaintext queries are rewritten according to the following rules:

- Equality Predicate:

$$\mathcal{QRF}_{\text{OPE}}(\sigma_{column=x}(table)) = \sigma_{column_{enc}=\mathcal{E}nc_K^{\text{OPE}}(x)}(table_{enc}) \tag{3.6}$$

- Inequality Predicate:

$$\mathcal{QRF}_{\text{OPE}}(\sigma_{name!=x}(table)) = \sigma_{column_{enc}!=\mathcal{E}nc_K^{\text{OPE}}(x)}(table_{enc}) \tag{3.7}$$

- IN Predicate:

$$\mathcal{QRF}_{\text{OPE}}(\sigma_{column\ IN\ (w,x)}(table)) = \sigma_{column_{enc}\ IN\ (\mathcal{E}nc_K^{\text{OPE}}(w),\mathcal{E}nc_K^{\text{OPE}}(x))}(table_{enc}) \tag{3.8}$$

- Range Predicate:

$$\mathcal{QRF}_{\text{OPE}}(\sigma_{column>x}(table)) = \sigma_{column_{enc}>\mathcal{E}nc_K^{\text{OPE}}(x)} \tag{3.9}$$

- Like Predicate:

$$\mathcal{QRF}_{\text{OPE}}(\sigma_{column\ LIKE\ 'M\%'}(table)) = \sigma_{column_{enc}\geq\mathcal{E}nc_K^{\text{OPE}}('M')\ AND\ column_{enc}<\mathcal{E}nc_K^{\text{OPE}}('N')} \tag{3.10}$$

- EQUI-JOIN:

$$\mathcal{QRF}_{\text{OPE}}(table1\ \bowtie_{table1.column=table2.column}\ table2) = (table1_{enc}\ \bowtie_{table1_{enc}.column_{enc}=table2_{enc}.column_{enc}}\ table2_{enc}) \tag{3.11}$$

- NON EQUI-JOIN:

$$\mathcal{QRF}_{\text{OPE}}(table1 \ \bowtie_{table1.column>table2.column} \ table2) = \\ (table1_{enc} \ \bowtie_{table1_{enc}.column_{enc}>table2_{enc}.column_{enc}} \ table2_{enc}) \tag{3.12}$$

- ORDER BY:

$$\mathcal{QRF}_{\text{OPE}}(\mathcal{O}_{column}(table)) = \mathcal{O}_{column_{enc}}(table_{enc}) \tag{3.13}$$

- MIN/MAX:

$$\mathcal{QRF}_{\text{OPE}}(\Gamma_{MIN(name)}(table)) = \Gamma_{MIN(name_{enc})}(table_{enc}) \tag{3.14}$$

Summary of Security Analysis

[14] introduces an *ideal object* to formalize the security of OPE schemes. Additionally new security notions based on one-wayness and indistinguishability are introduced and analysed. There are a few other works from the security community that build up on [14] to find an OPE encryption scheme that meets the *ideal* security requirements, such as in [15, 41, 48, 69]. Whereas these solutions focus on finding an encryption scheme that satisfies typical adversaries in the world of cryptography, the problem of dealing with adversaries having domain and frequency knowledge remains.

Order preserving encryption schemes have two domain related weaknesses: First problem is that being deterministic it leaks the frequency distribution of its underlying data. As any deterministic encryption scheme, this weakness allows the adversary to perform statistical analysis on the encrypted database. For instance, in Table 3.1 by looking at *OPE* column, it is obvious that "Messi" has placed more orders than the others. Of course, if the underlying data has uniform frequency distribution the frequency leakage of a order preserving encryption scheme will not impose any threat. The second problem is that order preserving encryption schemes leak total order. This fact imposes an obvious threat against adversaries with Domain knowledge. In the next section we will explain the domain and frequency attacks in more detail.

3.1.3 Homomorphic Encryption Schemes

Homomorphic encryption is an encryption method which allows specific types of computations to be carried out on ciphertext and generate an encrypted result which, when decrypted, matches the result of operations performed on the plaintext. In other words, the encryption function is a group homomorphism that given two groups $(\mathcal{X}, \bullet)$ and $(\mathcal{Y}, \oplus)$, $\mathcal{E}nc : \mathcal{X} \longrightarrow \mathcal{Y}$ s.t. $\forall\, x_i$ and $x_j \in \mathcal{X} \implies \mathcal{E}nc(x_i \bullet x_j) = \mathcal{E}nc(x_i) \oplus \mathcal{E}nc(x_j)$

Currently, the homomorphic schemes that are widely used allow only homomorphic computation of one operation (either addition or multiplication) on plaintexts, such

schemes are called *partially homomorphic encryption (PHE)* schemes such as Paillier [44] and ElGamal [23]. A cryptosystem which supports both addition and multiplication is called a *fully homomorphic encryption (FHE)* [27] and is far more powerful. The existence of an efficient and fully homomorphic cryptosystem would have great practical implications in the outsourcing of private computations.

Query Rewrite

A homomorphic encryption scheme is important to implement the aggregate functions such as SUM and AVG on encrypted data. The queries can be simply rewritten using the homomorphic properties of the encryption function:

- SUM/AVG:

$$\mathcal{QRF}_{\text{HES}}(\Gamma_{SUM(column)}(table)) = \Gamma_{SUM(column_{enc})}(table_{enc}) \tag{3.15}$$

[5] has made a systematic study to use fully homomorphic encryption for solving database queries beyond simple aggregations and numeric calculations.

Summary of Security Analysis

Homomorphic encryption schemes fulfill the indistinguishability requirement against chosen plaintext attacks and therefore are considered one of the strongest cryptosystems. However, their practicality for database applications needs a lot of improvement [5].

3.1.4 Probabilistic Encryption Schemes

A probabilistic encryption scheme uses randomness in its encryption algorithm, so that when encrypting the same plaintext several times it produces different ciphertexts. Various symmetric encryption algorithms achieve this property by adding a random element to the encryption process each time a plaintext is being encrypted (e.g., block ciphers when used in a chaining mode such as CBC).

Construction 3. *An Probabilistic Encryption scheme,* $PROB = (\mathcal{K}, \mathcal{E}nc, \mathcal{D}ec)$ *is defined as follows:*

- *Algorithm* $\mathcal{K}$ *returns a random key,* $K \xleftarrow{\$} \mathcal{K}$.
- $\mathcal{E}nc$ *takes* K, x *and a random initialization vector,* $V \xleftarrow{\$} \mathcal{V}$, *as input and returns* $y = \mathcal{E}nc_K(x, V)$.
- $\mathcal{D}ec$ *takes* K *and* y *as input and returns* $x = \mathcal{D}ec_K(y, V)$.

Table 3.1 gives an example of a set of customer names encrypted with $PROB$. In this section we only consider non-homomorphic probabilistic encryption schemes.

Query Rewrite

A probabilistic encryption scheme is semantically secure but has no property that can be used for query processing. In order to do query processing on probabilistically encrypted data, all the tables in the query need to be shipped to a trusted component, decrypted and processed there. This is a very inefficient procedure because it costs both a lot of network bandwidth and processing power on the client's machine. For example given Table 3.1, the client submits a plaintext query to retrieve "Messi" from the `customer` table:

```
SELECT name FROM customer WHERE name = 'Messi'
```

If the name column is encrypted using a probabilistic encryption scheme (PROB), the equality predicate needs to be discarded, since to encrypt "Messi", the query rewrite function needs to know the random element that has been used to encrypt "Messi" in the first place. This random element is stored alongside the encrypted "Messi" in the database. Hence, the above query will be rewritten as:

```
SELECT name_enc FROM customer_enc
```

This indicates that in order to retrieve only one row, we had to ship the whole table to the trusted component and undergo a vast amount of heavy decryption jobs to filter all the rows that correspond to "Messi". To summarize, no SQL operator is supported by non-homomorphic probabilistic encryption schemes.

Summary of Security Analysis

To be semantically secure, that is, to hide even partial information about the plaintext, an encryption algorithm must be probabilistic. Probabilistic encryption is particularly important when using public key cryptography. When a deterministic encryption algorithm is used, the adversary can simply try encrypting each of his guesses under the recipient's public key, and compare each result to the target ciphertext. In our adversary model the adversary has no access to the keys to reproduce the encryption of different words.

3.2 Related Work

Due to its importance, there has been a large number 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 [18], data anonymization [62], and privacy-preserving data mining [7]. The specific task of ensuring privacy for database services has also been studied extensively in the past [20]. A recent line of work includes

systems like CryptDB [47] and derivates like Monomi [66]. To achieve good performance, these systems make use of weak encryption techniques if necessary. That means, they support strong encryption only for data that is rarely queried or not on the critical path of query processing, thereby favoring practicality and performance over security. As mentioned in the introduction, the work on RPE is orthogonal to these systems. Ultimately, we believe that the best results can be achieved by using RPE schemes as a considerably performant and secure encryption scheme into systems like TrustedDB [10], Cipherbase [9], CryptDB [47], and Monomi [66]. For cases in which the underlying weak encryption scheme performs poorly, RPE naturally performs poorly too. Examples are non-equi joins and certain kinds of sub-selects. To the best of our knowledge, only the use of RPE enables a database system to process updates and complex decision support queries. TrustedDB and CryptDB can only handle simple queries. Monomi can handle TPC-H queries, but does not support updates. While Cipherbase was designed to be general-purpose, it has so far only been applied to transactional workloads without complex queries.

In general, there are four approaches to address the database privacy problem: The first approach involves providing 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 security middleware [58, 59]. Alternative, related approaches were proposed in [8, 12, 26].

The second approach to ensure privacy in the cloud is to partition the data and then encrypt in the granularity of partitions (rather than individual values) [22, 29, 30]. Data is shipped from the cloud to the security middleware in the granularity of partitions and these partitions are post-filtered in the middleware. [32] show how such partitions can be indexed.

The third approach is based on order-preserving encryption. [6, 14, 15, 48, 71] fall into this category. Order-preserving encryption is an attractive encryption scheme for database applications since it allows to process range and rank queries on encrypted data without decryption. Therefore, OPE is considered very performant (in terms of response time) supporting a large class of queries. Exploiting order relationships even allows for further query processing optimizations [63]. OPE was first proposed by [6] and recently won the SIGMOD 2014 Test of Time Award because of its impact on achieving high performance and at the same time supporting confidentiality for outsourced databases. However, none of the proposed OPE techniques are applicable in the presence of a domain or frequency attack. There have been several improvements on order-preserving techniques as in [36, 57, 72], however their performance in terms of response time deviates largely from order-preserving schemes.

The fourth approach is to find homomorphic encryption techniques. The goal is to guarantee algebraic properties that allow for efficient query processing on the encrypted

data. Homomorphic encryption has been first proposed in the late 70's [50] and has recently regained attention [27, 60]. Unfortunately, there has not been a breakthrough that would make homomorphic encryption practical 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.

Finally, we would like to iterate that transparent encryption as supported in commercial database systems such as Oracle 10g [43] is not sufficient against honest-but-curious attackers. It violates the principle of separation, i.e. the keys are kept in the cloud and data is decrypted in the memory and caches of the cloud machines. This fact makes it possible for system administrators to flush the memory and cache content or even launch side channel attacks to find the keys [73].

3.3 Conclusion

Table 3.3 demonstrates a summary of this chapter. It shows which encryption schemes support which SQL operators. We use this table in our quest to find an efficient but secure encryption scheme for cloud databases. According to Table 3.3, non-homomorphic probabilistic encryption schemes (PROB), e.g., AES-CBC, do not support any SQL operators and are thus unacceptable for today's cloud database performance requirements. On the other hand, we see that order-preserving encryption schemes support most SQL operators. Order-preserving encryption is crucial for cloud databases because it enables them to achieve high performance. To emphasize the role that OPE has played in the advancement of database encryption schemes in the last 10 years, the initial OPE paper from Agrawal [6] was awarded the prestigious SIGMOD Test of Time Award 2014. On the other hand, in terms of security, OPE has certain weaknesses against adversaries with domain knowledge. In this table we additionally include AES-ECB as a wide-spread representative of deterministic encryption, and Paillier cryptosystem [44] as a wide-spread representative (used in [5, 47, 66]) of partially homomorphic encryption to enable a complete overview of the state of the art database encryption schemes. In the next chapter we formulate those weaknesses by formal attacker models. We then suggest various encryption schemes that are based on OPE but improve its security in a tunable manner according to the client's performance and security requirements.

SQL-Operator	AES-ECB	OPE	Paillier [44]	AES-CBC
DISTINCT	✓	✓	✗	✗
WHERE (=, !=)	✓	✓	✗	✗
WHERE ($<$, $>$)	✗	✓	✗	✗
LIKE(Prefix%)	✗	✓	✗	✗
LIKE(%Suffix)	✗	✗	✗	✗
IN	✓	✓	✗	✗
EQUI-JOIN	✓	✓	✗	✗
NON EQUI-JOIN	✗	✓	✗	✗
TOP N	✗	✓	✗	✗
ORDER BY	✗	✓	✗	✗
SUM/AVG	✗	✗	✓	✗
MIN/MAX	✗	✓	✗	✗
GROUP BY	✓	✓	✗	✗

Table 3.3: SQL-Operator Support by State of the Art Encryption Schemes

4

Database Adversary Models

Initially, the events that motivated this thesis were privacy violations in a *private* cloud. Between 2008 and 2010, database administrators of several Swiss and Liechtenstein banks sold CDs with customer information to French and German tax authorities. These events were a major embarrassment for the Liechtenstein and Swiss financial marketplaces. What makes privacy in a private cloud particularly challenging is that the attacker often has precise knowledge of the value domains. The database administrator of the Swiss and Liechtenstein banks, for instance, had a list of all customers and only needed to retrieve other relevant account information for selected customers.

There have been many proposals to quantify privacy as part of work on the anonymization of databases; e.g., K-Anonymity [62], L-Diversity [40], or T-Closeness [39]. However, these metrics are not applicable to transactional databases that must return exact results. In order to quantify privacy for transactional databases, we define new security notions. In order to understand the privacy requirements, it is crucial to describe the attacker scenarios and measure its probability distribution. In general, there are many ways how information can be leaked in the architecture of Chapter 2. For instance, clients who see plaintext values of sensitive information could pass this information along. Fortunately, most applications that maintain sensitive data have sophisticated authorization that limits the capabilities of users to create significant damage. (At least, this observation holds for the financial industry whose use cases motivated this thesis.) The real weak point is the encrypted database. This chapter studies two attacker scenarios against the database. Both of these scenarios are crucial to meet the specific "tax CD" use case mentioned earlier.

4.1 Running Example

The two scenarios can be best described using an example. As a running example, we use a database with the following two relations.

Customer(name, city, info)

Order(id, *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. We explicitly model the *city* attribute of *Customer* in order to illustrate the frequency attack (below).

The first scenario is called *domain attack*. In this scenario, it is assumed that the attacker (e.g., database administrator or cloud provider) has complete knowledge of the domain of values that are encrypted. In the running example, the attacker could know the names of all customers and try to infer the *info* for one or several customers. As will become clear later in this chapter, 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 (or the distribution) 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. If the *city* is supposed to be a secret, then the frequency attack imposes even higher requirements on the encryption scheme. Again, the required properties of the encryption scheme will be discussed in detail in this chapter.

The frequency attack covers a wide spectrum of scenarios; such scenarios include attacks based on inference and attribute correlations. As another example, an attacker could know about the frequency distribution of the *cust* attribute in the *Order* table; i.e., the attacker knows how many orders each customer has placed. If *Customer.name* and *Order.cust* are not encrypted properly, the attacker could easily infer the customer information with this frequency information. Attacks that are based on correlations are not discussed in detail in this thesis. Nevertheless, it should become intuitively clear that encryption techniques that ensure privacy against frequency attacks can also be used against such correlation attacks. For instance, the name Peter occurs more frequently for customers from New York than for customers from Bangalore. To protect domains with such correlations, the frequency distribution of the *composite* attribute values must be distorted.

Let x_1, x_2 be plaintext values of a domain that needs to be protected. For example, *Neymar* and *Messi* could be names of customers of a Liechtenstein bank. Let $\mathcal{E}nc$ be

a function that maps a plaintext value to a single ciphertext or to a set of ciphertexts. That is, $\mathcal{E}nc(x)$ is/are the ciphertext(s) 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. $\mathcal{E}nc$ and all other encryption functions are stored on the trusted component as discussed in Chapter 2. It is assumed that an attacker does not have access to the trusted component.

As thoroughly discussed in Chapter 3, an encryption function $\mathcal{E}nc$ can have none, one, or several of the following properties:

- *Deterministic (DET):* Each plaintext corresponds to exactly one ciphertext; i.e., $x_1 = x_2 \implies \forall y_1 \in \mathcal{E}nc(x_1), y_2 \in \mathcal{E}nc(x_2) : y_1 = y_2$.

- *Order-Preserving (OP):* If a plaintext is smaller than another plaintext, then all its ciphertexts are smaller than the ciphertexts of the other plaintext; i.e., $x_1 < x_2 \implies \forall y_1 \in \mathcal{E}nc(x_1), y_2 \in \mathcal{E}nc(x_2) : x_1 < x_2$.

- *Homomorphic:* For any homomorphism, f: $\mathcal{E}nc(f(x_1, x_2)) = f(\mathcal{E}nc(x_1), \mathcal{E}nc(x_2))$.

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 trusted component. Deterministic encryption, for instance, is important to implement equality predicates. Accordingly, order preservation is important to implement range and rank predicates such as searches for customer names with wildcards; this observation has been exploited by Agrawal et al. [6]. Homomorphic encryption is needed in order to compute aggregate functions on metrics (e.g., the sum of "price $\times$ volume" of orders). While all properties are relevant, this thesis focuses on deterministic and order-preserving encryption.

Table 4.1 shows examples (using customer names) of different combinations of deterministic (denoted as DET), probabilistic (denoted as PROB), order-preserving (OP), and non order-preserving (NOP) encryption. These examples demonstrate how queries can be processed on an encrypted database. For instance, the query:

```
SELECT * FROM customer WHERE name = 'Neymar'
```

can easily be rewritten to:

```
SELECT * FROM customer_enc WHERE name_enc = 2
```

in the DET/NOP scheme. On the negative side, processing a query like:

```
SELECT * FROM customer WHERE name LIKE 'M\%'
```

clear text	DET/OP	DET/NOP	PROB/OP	PROB/NOP
Benzema	1	5	{ 1,2,4 }	{ 2, 11, 13 }
Messi	2	3	{ 5,6 }	{ 4, 12 }
Neymar	7	2	{ 8 }	{ 8, 12 }
Ronaldo	9	8	{ 11,12,13 }	{ 1, 5 }

Table 4.1: Example: Various Encryption Properties

with the DET/NOP scheme involves either reading the whole *Customer* table and post-filtering the results in the trusted component or enumerating all relevant codes in an `IN` predicate; i.e.:

```
SELECT * FROM customer_enc WHERE name_enc IN (2, 14)
```

Both approaches can result in poor performance.

Just as a NOP scheme, PROB encryption can impact the performance of queries. The query:

```
SELECT * FROM customer WHERE name = 'Neymar'
```

is rewritten into the following query using the PROB/NOP encryption:

```
SELECT * FROM customer_enc
```

This rewritten query is far more expensive to evaluate than the original query on a non-encrypted database.

In summary, these examples illustrate that privacy does not come for free. As a result, we carefully evaluated the performance overheads of our proposed techniques and present the results in this thesis. The remainder of this subsection discusses how the properties of an encryption scheme impact the privacy with regard to domain and frequency attacks.

Notation. Let $\mathcal{X}$ be a set of plaintext values in a finite domain, and $\mathcal{Y}$ be the set of ciphertext values. The size of $\mathcal{X}$ is denoted as $X = |\mathcal{X}|$; the same applies for the size of $\mathcal{Y}$, $Y = |\mathcal{Y}|$. Plaintext elements are denoted as x and ciphertext elements as y. Additionally, we denote the Key set to be $\mathcal{K}eys$ and $K \xleftarrow{\$} \mathcal{K}eys$ denotes that a key, K, is selected uniformly at random from $\mathcal{K}eys$. The $\$$ sign on top of the $\leftarrow$ depicts that the selection was uniformly at random. $\mathcal{K}$ is a randomized algorithm that creates a random key from a finite set. Let $\mathcal{E}nc$ be the encryption function having a key, K, and a plaintext value, x, as its input parameters; thus, we have: $y = \mathcal{E}nc(K, x)$. Symmetrically, $\mathcal{D}ec$ will be the decryption function, taking y and K as input, yielding: $x = \mathcal{D}ec(K, y)$. Let $Rank_x$ be the ordering rank of x in $\mathcal{X}$. Symmetrically, we have $Rank_y$ which denotes the ordering rank of y in $\mathcal{Y}$. Let $rank$ be the function that returns the rank of an element in its corresponding space. The frequency distribution of $\mathcal{X}$ and $\mathcal{Y}$ is denoted as $\mathcal{F}_{\mathcal{X}}$ and

$\mathcal{F}_{\mathcal{Y}}$ respectively. Let $freq$ be the function that returns the frequency of an element in its corresponding space. Let $\mathcal{A}$ denote an adversary.

A couple of concrete usecases from the financial industry have given birth to new adversary models for cloud databases. These models have not been cryptographically treated so far. In this section we formally introduce the new adversary models and security metrics.

4.2 Domain Attack

Domain attack is launched by an adversary that has a-priori knowledge of the plaintext domain. In our motivating usecase from the financial industry, the database administrator had a list of all customers and needed to retrieve other relevant account information for selected customers. Domain attack is particularly harmful to order preserving encryption schemes. In order to protect data against domain attacks, the encryption scheme must be **non order-preserving**. To see why, assume that an attacker knows that the database contains information about the four customers Benzema, Messi, Neymar, and Ronaldo. 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 attacker to infer that Code 1 corresponds to the Benzema, Code 2 to Messi, etc.

A domain attack against a PROB/OP encryption scheme is more difficult because the attacker may not know how many codes are used to encrypt the Benzema. Nevertheless, an attacker can easily infer that Code 1 belongs to the Benzema and Code 13 belongs to Messi in the PROB/OP scheme of Table 4.1.

In the remainder of this section, for the first time we introduce the domain attack, and define a new security metric called rank-one wayness to measure the probability distribution of an adversary launching a domain attack on an order preserving encryption variant.

4.2.1 Rank One-Wayness

As shown in an example earlier, knowing the Domain of the plaintext values, $\mathcal{X}$, and having all the ciphertexts in the encrypted database allows one to break an order-preserving encryption scheme using sorting. On the other hand, 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. Thus, it is crucial to define a measure that captures the probability of an attacker's success launching a domain attack.

In any given encryption scheme, $\mathcal{ES}$, in order to measure the success probability of $\mathcal{A}$, in correlating the rank of a given ciphertext to the rank of its underlying plaintext, we introduce a new measure called Rank One-Wayness (ROW). The ROW advantage is defined to be the probability of Experiment 1 returning *1*.

$$Adv^{\text{ROW}}_{\mathcal{ES}}(\mathcal{A}) = Pr[Exp^{\text{ROW}}_{\mathcal{ES}}(\mathcal{A}) = 1)] \tag{4.1}$$

Experiment 1 : $\text{Exp}^{\text{ROW}}_{\mathcal{ES}}(\mathcal{A})$

1: $K \xleftarrow{\$} \mathcal{K}eys$
2: $x \xleftarrow{\$} \mathcal{X}$
3: $y \leftarrow \mathcal{E}nc(K, x)$
4: $Rank_y \leftarrow rank(y)$
5: $x' \xleftarrow{\$} \mathcal{A}(\mathcal{X}, Rank_y)$
6: **if** $x = x'$ **then return** 1
7: **else return** 0

Experiment 1 first runs the key generator to get a random key, K. Then it randomly chooses an element, x from the plaintext domain, X. Then it encrypts, x to get $y = \mathcal{E}nc(K, x)$. The experiment then computes the rank of y in the ciphertext space, $\mathcal{Y}$, using the $rank()$ function. The adversary receives the plaintext domain, and the rank of the ciphertext, $Rank_y$, and needs to find the corresponding plaintext, x, in $\mathcal{X}$.

To further clarify Experiment 1, consider an example where $\mathcal{ES} = OPE$ and $\mathcal{X} = \{'Benzema', 'Messi', 'Totti'\}$. Assume the ciphertext space to be $\mathcal{Y} = \{143, 465, 706\}$. We start the experiment by choosing a random element from $\mathcal{X}$, for instance $'Messi'$ and encrypt it. We provide $\mathcal{A}$ with $\mathcal{X}$ and $Rank_{465} = 2$. $\mathcal{A}$ has to return an element from $\mathcal{X}$ which he thinks corresponds to the second element of $\mathcal{Y}$. The probability that $\mathcal{A}$ guesses $'Messi'$ correctly is called the *ROW advantage*.

4.3 Frequency Attack

A frequency attack is an attack that is launched by an adversary that has a-priori knowledge of the plaintext values as well as their frequency distribution.

Since the frequency attack is a generalization of the domain attack, a NOP encryption scheme is also required to protect against the frequency attack. In addition, an encryption scheme must be **probabilistic** to ensure privacy against frequency attacks. To see why, assume that the attacker knows that the Benzema has placed four orders, Messi has placed ten orders, etc. If the Benzema are represented by a single code in a DET/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 frequency skew (without exact values). However, to be safe in the general case, a PROB encryption scheme is required to protect confidential data against a frequency attack.

Another prominent weakness of not only current OPE schemes but in general all deterministic encryption schemes is their inability to hide the original frequency distribution of the plaintext domain, most specifically if the plaintext domain has skewed frquency distribution as shown in Figure 7.3a. Deterministic encryption schemes allow $\mathcal{A}$ to simply find a mapping between the plaintext and the ciphertext space just by looking at their frequency distributions and thereby discovering the plaintext-ciphertext correspondence. Similar to *Domain Attack* to model an $\mathcal{A}$ on RPE, we define a new security metric called *Frequency One-Wayness (FOW)* advantage.

Assuming that the plaintext values have non-uniform frequency distributions, all deterministic encryption schemes that are widely used in well-known database engines such as Oracle, MSSQL and MySQL including all the variants of OPE schemes proposed in [6, 14, 15] and used in [47, 66] allow an adversary with frequency knowledge to efficiently break the encryption by sorting the ciphertexts according to their frequencies.

4.3.1 Frequency One-Wayness

In any given encryption scheme, $\mathcal{ES}$, in order to measure the success probability of $\mathcal{A}$, in correlating the frequency of a given ciphertext to the frequency of its underlying plaintext, we introduce a new measure called Frequency One-Wayness (FOW). The FOW advantage is defined to be the probability of Experiment 2 returning *1*.

$$Adv^{\text{FOW}}_{\mathcal{ES}}(\mathcal{A}) = Pr[Exp^{\text{FOW}}_{\mathcal{ES}}(\mathcal{A}) = 1] \tag{4.2}$$

Experiment 2 : $\mathbf{Exp}^{\text{FOW}}_{\mathcal{ES}}(\mathcal{A})$

1: $y \xleftarrow{\$} \mathcal{Y}$
2: $x \leftarrow \mathcal{D}ec(y)$
3: $Freq_y \leftarrow freq(y)$
4: $x' \xleftarrow{\$} \mathcal{A}(\mathcal{X}, \mathcal{F}_x, Freq_y)$
5: **if** $x = x'$ **then return** 1
6: **else return** 0

Experiment 2 chooses a random element, y from the ciphertext space, Y. Then it decrypts, y to get its underlying plaintext, x. Th experiment then computes the frequency of y in the ciphertext space, $\mathcal{Y}$, using the $freq()$ function. The adversary receives the plaintext domain, the frequency distribution of the plaintext values in $\mathcal{X}$, $\mathcal{F}_x$, and the

frequency of the ciphertext, $Freq_y$, and needs to find the corresponding plaintext, x, in $\mathcal{X}$.

Again we illustrate this with an example. Let $\mathcal{X} = \{'Benzema', 'Benzema', 'Totti'\}$ and $\mathcal{ES} = OPE$. The ciphertext space will be $\mathcal{Y} = \{143, 143, 706\}$. We start the experiment by choosing a random element from $\mathcal{Y}$, for instance 143 and decrypt it. We give $\mathcal{A}$: $\mathcal{X}$, $\mathcal{F}_{\mathcal{X}} = \{2, 1\}$ and $freq_{143} = 2$. $\mathcal{A}$ has to return an element from $\mathcal{X}$ which he thinks corresponds to 143. The probability that $\mathcal{A}$ guesses correctly is called the *FOW advantage*.

4.4 Query Log Attack

A *Query Log Attack (QLA)* is launched by an adversary that in addition to domain knowledge, has access to the database query logs, Q'. For instance, in our case a database administrator has access to the rewritten query logs for maintenance or troubleshooting reasons, and preventing this kind of access is not an option. Depending on the encryption scheme, the query logs may reveal more information about the underlying data than what the encryption scheme was initially intended to leak. Thus, it is crucial to add query log analysis to the list of possible cryptanalysis on database encryption schemes. In general, the query logs leak (1) queries, (2) origin (e.g. an IP address of the client submitting the query), (3) frequencies, and (4) time stamp of the query. In this thesis we will focus on the *queries*. How to perform query analysis and what an adversary can obtain from it, depends on the underlying encryption scheme.

We define the success probability of an adversary, $\mathcal{A}$, as his advantage to break the Rank One-Wayness or the Frequency One-Wayness of the underlying encryption scheme, $\mathcal{ES}$. The adversary, $\mathcal{A}$, is given the plaintext domain, $\mathcal{X}$, the frequency distribution of the plaintext domain, $\mathcal{F}_x$, the ciphertext frequency, $Freq_y$, the ciphertext rank, $Rank_y$, and additionally the rewritten query logs, Q'. In the end, $\mathcal{A}$ is asked to return the underlying plaintext, x. The advantage of $\mathcal{A}$ is formulated as:

$$Adv^{\text{QLA}}_{\mathcal{ES}}(\mathcal{A}) = Pr[Exp^{\text{QLA}}_{\mathcal{ES}}(\mathcal{A}) = 1] \tag{4.3}$$

Experiment 3 chooses a random element, y from the ciphertext space, Y. Then it decrypts, y to get its underlying plaintext, x. Th experiment then computes the frequency and rank of y in the ciphertext space, $\mathcal{Y}$, using the $freq()$ and $rank()$ function. The adversary receives the plaintext domain, $\mathcal{X}$, the frequency distribution of the plaintext values in $\mathcal{X}$, $\mathcal{F}_x$, the frequency and rank of the ciphertext, $Freq_y$ and $Rank_y$, and the rewritten query logs, Q', and needs to find the corresponding plaintext, x, in $\mathcal{X}$.

Corollary 1. *For an adversary, $\mathcal{A}$ to win the query log attack, it can play the domain attack, frequency attack, or perform query log analysis.*

Experiment 3 : $\mathbf{Exp}^{\text{QLA}}_{\mathcal{ES}}(\mathcal{A})$

1: $y \xleftarrow{\$} \mathcal{Y}$
2: $x \leftarrow \mathcal{D}ec(y)$
3: $Rank_y \leftarrow rank(y)$
4: $Freq_y \leftarrow freq(y)$
5: $x' \xleftarrow{\$} \mathcal{A}(\mathcal{X}, \mathcal{F}_x, Freq_y, Rank_y, Q')$ ▷ note the additional argument Q'
6: **if** $x = x'$ **then return** 1
7: **else return** 0

Lemma 1. *For an encryption scheme, $\mathcal{ES}$ to be safe against the query log attack, it is necessary for $\mathcal{ES}$ to be safe against domain and frequency attack.*

Proof. Here we need to show that we can win the QLA experiment, if we can win either Experiment 1 or Experiment 2 for an arbitrary encryption scheme, $\mathcal{ES}$. To prove this, we need to solve the QLA experiment with the solver of the Experiment 1 or Experiment 2.

Suppose $\mathcal{A}$ is an adversary with non-trivial *ROW* advantage against $\mathcal{ES}$. We construct a QLA adversary, $\mathcal{A}_{\text{QLA}}$, against $\mathcal{ES}$. As per definition in Experiment 3, $\mathcal{A}_{\text{QLA}}$ is given the plaintext domain, $\mathcal{X}$, the frequency distribution of the plaintext values in $\mathcal{X}$, $\mathcal{F}_x$, the frequency and rank of the ciphertext, $Freq_y$ and $Rank_y$, and the rewritten query logs, Q'. $\mathcal{A}_{\text{QLA}}$ simply runs $\mathcal{A}(\mathcal{X}, Rank_y)$. Trivially, we have found an efficient way to solve $\mathcal{A}_{\text{QLA}}$ with $\mathcal{A}$. $\mathcal{A}_{\text{QLA}}$ is efficient since it has simply used $\mathcal{A}(\mathcal{X}, Rank_y)$ on a subset of the parameters it has received from the QLA experiment making its complexity is constant.

For the second part, suppose $\mathcal{A}$ is an adversary with non-trivial *FOW* advantage against $\mathcal{ES}$. We construct a QLA adversary, $\mathcal{A}_{\text{QLA}}$, against $\mathcal{ES}$. As per definition in Experiment 3, $\mathcal{A}_{\text{QLA}}$ is given the plaintext domain, $\mathcal{X}$, the frequency distribution of the plaintext values in $\mathcal{X}$, $\mathcal{F}_x$, the frequency and rank of the ciphertext, $Freq_y$ and $Rank_y$, and the rewritten query logs, Q'. $\mathcal{A}_{\text{QLA}}$ simply runs $\mathcal{A}(\mathcal{X}, \mathcal{F}_x, Freq_y)$. Trivially, we have found an efficient way to solve $\mathcal{A}_{\text{QLA}}$ with $\mathcal{A}$. $\mathcal{A}_{\text{QLA}}$ is efficient since it has simply used $\mathcal{A}(\mathcal{X}, \mathcal{F}_x, Freq_y)$ on a subset of the parameters it has received from the QLA experiment making its complexity is constant.

□

Next chapter describes query log attacks in a general context, and introduces a framework to evaluate the query logs for any given encryption scheme without the domain and frequency assumptions that we did in this chapter.

5
Query Log Attack

We live in the era of cloud computing. There are many economical reasons to outsource data: Cloud computing offers lower cost by making use of an economy of scale, it reduces capital expenses, and it allows organizations to focus on their core business. Unfortunately, there is also a big road block for cloud adoptions: Security. Organizations are worried that their confidential data might be stolen.

To draw a balance between economic interests and confidentiality, a number of systems and techniques have been developed over the last decade [9,10,29,47]. All of these systems rely to some extent on the use of encryption schemes discussed in Chapter 3. These encryption schemes allow to process certain queries in the cloud without decrypting the data. This way, if applicable, these schemes show good performance and fulfill the economic promise of cloud computing. Unfortunately, these encryption schemes are often not secure and can easily be attacked. As a result, a great deal of research has been invested into improving the security of these schemes with no or a graceful degradation of performance. Prominent examples are [15, 22, 29, 31, 35, 36, 48, 51, 57, 70–72].

While there have been great advances in improving the security of such encryption schemes for various kinds of attacks, almost all recent work has ignored an important kind of attack to which cloud computing environments are particularly vulnerable: Query Log Attacks. In this kind of attack, the attacker has access to the encrypted database, possibly additional background knowledge (e.g., value distributions in the domain), *and* the stream of queries and updates that are submitted by clients and applications. Imagine a malicious database or system administrator at the cloud provider and it is easy to see why this attack is real. Traditional work on encryption assumes a model in which the attacker only has access to the encrypted database and background

Name (plaintext)	Name (OPE)	Name (MOPE [15])	Name (Prob-OPE [51])
Benzema	1354	3178	1354
Messi	2564	4567	2568
Messi	2564	4567	2796
Neymar	3129	1890	3871
Ronaldo	4980	2601	4980

Table 5.1: Example: Encryption Techniques

knowledge, but does not observe the query log (i.e., stream of updates and queries).

To see the importance of the query log on two recently developed order-preserving encryption techniques, consider Table 5.1. Table 5.1 shows a *Name* column of, say, a *Customer* table in a database. It shows the *Name* column in plaintext, the *Name* column using a traditional, very weak order-preserving encryption technique, encryption using the MOPE technique [15], and a probabilistic order-preserving encryption technique [51]. The idea of MOPE is to *shift* the beginning of the ciphertexts. This way, the attacker cannot decipher the encrypted database even if the attacker has precise knowledge of the domain (e.g., an address book). [15] provides the proofs for a number of fairly strong security properties for MOPE that hold if the attacker does not have access to the query log. The following example shows how MOPE can lose its security advantages over traditional, weak OPE when the attacker has access to the query log:

Example-1: MOPE. The client or application submits a query that asks for the first customer in the database in alphabetical order:

```
SELECT MIN(name) FROM customer
```

As will become clear in this thesis, there are many ways to rewrite this query so that it can be processed on an MOPE-encrypted *Name* column in the cloud without decrypting the *Name* column; each with different performance and security properties. The most natural approach is to rewrite this query in the following way:

```
SELECT MIN(name_enc) FROM customer_enc WHERE name_enc > 3000
```

This rewritten query can be processed entirely in the cloud (without any decryption). The result is the (encrypted) tuple with ciphertext 3178 which is returned from the cloud to the client. The client decrypts this tuple and receives the correct result: “Benzema”. This approach has the best possible performance: The query can be processed in the cloud on the encrypted database in (almost) the same way as the original query on a plaintext database. Furthermore, only a single tuple is shipped from the cloud to the client, thereby incurring no additional communication cost. Unfortunately, however, this rewritten query leaks the *offset* used by MOPE in this example. As a result, the attacker knows that the cipher 3178 corresponds to the alphabetically first customer

and, as a result, MOPE degrades to OPE which is generally perceived to be a very weak encryption scheme.

Example-2: Prob-OPE. The second example shows how a probabilistic order-preserving encryption which (like MOPE) has certain strong encryption guarantees for static databases without query logs [51] can degrade to a weak OPE encryption in the presence of query logs. In this example, the client is interested in all the information of customer "Messi":

```
SELECT name FROM customer WHERE name = 'Messi'
```

The most natural way to rewrite this query for processing on the encrypted database in the cloud is as follows:

```
SELECT name_enc FROM customer_enc
WHERE name_enc < 3000 AND name_enc > 2500
```

Again, this is the most performant way to process the query on the encrypted database in the cloud with only little overhead as compared to executing the original query on a plaintext database in the cloud. Unfortunately, however, this rewrite defeats the purpose of probabilistic encryption and Prob-OPE degrades to OPE as a result. An attacker who sees the rewritten query (or a stream of rewritten queries) may observe that `{2568,2796}` are never queried separately; hence, these values most likely correspond to the same plaintext value. This inference destroys the probabilistic nature of Prob-OPE which is needed to protect the data if the attacker knows that "Messi" has placed more orders than the other patients.

These examples show that it is critical to consider query log attacks when analyzing the security of an encryption scheme. This chapter introduces a framework that enables us for the first time, to reason about the security of an encryption scheme under various query log attacks.

5.1 Attacks on Ciphers

To formalize query log attacks, we use a similar approach that has been used to formalize attacks on ciphers. In this section we revisit the attacker scenarios on ciphers. Later in this chapter we present query log attacker models and use reduction proofs to reduce them to known attacker models on ciphers revisited in this section.

To argue about security, it is crucial that the assumptions, goal and scenario of the adversary are clearly defined. In order to formalize an adversary model, we define an *Experiment*[1] that the *Adversary* needs to win. An experiment, as shown in Figure 5.1 is denoted as $\mathcal{E}$ and is a probabilistic system that has two interfaces. On the left interface,

[1] An Experiment is also called a Game in some security literatures.

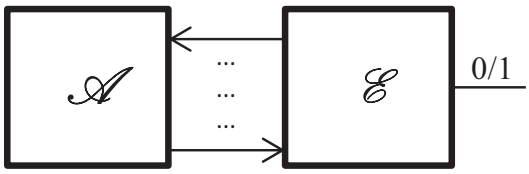


Figure 5.1: An Experiment interacts with an Adversary.

$\mathcal{E}$ is connected to an adversary (attacker), denoted as $\mathcal{A}$, and on its right interface it outputs 1 if $\mathcal{A}$ fulfills the *Goal* of $\mathcal{E}$, otherwise 0. $\mathcal{A}$ is (or has access to) an algorithm that interacts with $\mathcal{E}$ and its goal is to win the experiment. The probability that $\mathcal{A}$ succeeds is called $\mathcal{A}$'s *Advantage*, denoted as $Adv(\mathcal{A}) = Pr[\mathcal{E}(\mathcal{A}) = 1]$.

In this section we go over the well-known attacker scenarios on the encryption schemes. These scenarios will be used in the reduction proofs in Section 4.4 where we define and prove the security of different query log attacks. Before we revisit the attacker models on ciphers, we introduce the notation that will be used in the coming sections.

Notation. Let x denote a plaintext value from the plaintext space, $\mathcal{X}$. Respectively, let y denote a ciphertext value from the ciphertext space, $\mathcal{Y}$. Let $\mathcal{E}nc(\tau, \mathbf{x})$ be the encryption function of an arbitrary encryption scheme, $\mathcal{ES}$ and τ be a random element (e.g. a key or a combination of a key and an initialization vector). $\mathbf{x} \xleftarrow{\$} \mathcal{X}$ simply means that a vector of plaintext elements have been chosen uniformly at random. The $\$$ implies a uniformly random selection from a set.

5.1.1 Ciphertext-only Attack (CoA)

In a *Ciphertext-only Attack* the attacker is given a series of ciphertexts for some plaintext unknown to the attacker [38]. The goal of the attacker is to distinguish pairs of ciphertexts based on the plaintext they encrypt. Indistinguishability under *Ciphertext-only Attack* is captured through the following experiment:

Experiment 4 chooses randomly two plaintext values, $(x_0, x_1) \in \mathcal{X}$, flips a coin and encrypts randomly one of them, $y = \mathcal{E}nc(\tau, x_b)$. The attacker is then given both (x_0, x_1) and y. Based on y, the attacker tries to find out whether x_0 or x_1 was encrypted. The probability that an attacker succeeds in this experiment is denoted as the IND-CoA advantage, $Adv_{\text{ES}}^{\text{IND-CoA}}(\mathcal{A})$ and is optimal if an attacker cannot do better than to randomly guess b, i.e. $Adv_{\text{ES}}^{\text{IND-CoA}}(\mathcal{A}) \leq \frac{1}{2}$. The probability is less than $\frac{1}{2}$ because this experiment can be repeated multiple times and the attacker does not hold a state of the previous iterations of the experiment.

Experiment 4 : $\mathbf{Exp}_{\text{ES}}^{\text{IND-KPA}}(\mathcal{A})$

$\tau \xleftarrow{\$} \mathcal{T}$
$(x_0, x_1) \xleftarrow{\$} \mathcal{X}$
$b \xleftarrow{\$} \{0,1\}$
$y \leftarrow \mathcal{E}nc(\tau, x_b)$
$b' \xleftarrow{\$} \mathcal{A}((x_0, x_1), y)$
if $b' = b$ **then return** 1
else return 0

5.1.2 Known-Plaintext Attack (KPA)

In a *Known-Plaintext Attack* the attacker is given a couple of plaintext-ciphertext pairs [38]. The goal of the attacker is to distinguish pairs of ciphertexts based on the plaintext they encrypt which were not initially given. Indistinguishability under *Known-Plaintext Attack* is captured through the following experiment:

Experiment 5 : $\mathbf{Exp}_{\text{ES}}^{\text{IND-KPA}}(\mathcal{A})$

$\tau \xleftarrow{\$} \mathcal{T}$
$\mathbf{x} \xleftarrow{\$} \mathcal{X}$
$\mathbf{y} \leftarrow \mathcal{E}nc(\tau, \mathbf{x})$
$(x_0, x_1) \xleftarrow{\$} \mathcal{X}\ s.t.\ x_0, x_1 \notin \mathbf{x}$
$b \xleftarrow{\$} \{0,1\}$
$y \leftarrow \mathcal{E}nc(\tau, x_b)$
$b' \xleftarrow{\$} \mathcal{A}(\mathbf{x}, \mathbf{y}, (x_0, x_1), y)$
if $b' = b$ **then return** 1
else return 0

Experiment 5 chooses uniformly at random a vector of plaintext values, **x**. It then encrypts these values using the $\mathcal{E}nc(\tau, \mathbf{x})$ function to obtain a vector of the corresponding ciphertext values, **y**. Then the experiment chooses randomly two plaintext values, $(x_0, x_1)\ s.t.\ x_0, x_1 \notin \mathbf{x}$, flips a coin and encrypts randomly one of them, $y = \mathcal{E}nc(\tau, x_b)$. The attacker is then given both **x** and **y**, (x_0, x_1) and y. Based on y and the information he may extract from the known plaintext-ciphertext pairs,**x** and **y**, the attacker tries to find out whether x_0 or x_1 was encrypted. The probability that an attacker can succeed in this experiment is denoted as the IND-KPA advantage, $Adv_{\text{ES}}^{\text{IND-KPA}}(\mathcal{A})$ and is optimal if an attacker cannot do better than to randomly guess b, i.e. $Adv_{\text{ES}}^{\text{IND-KPA}}(\mathcal{A}) \leq \frac{1}{2}$. The probability is less than $\frac{1}{2}$ because this experiment can be repeated multiple times.

5.1.3 Chosen-Plaintext Attack (CPA)

In a *Chosen-Plaintext Attack* the attacker is given plaintext-ciphertext pairs for the plaintext vector chosen by the attacker [38]. In other words the attacker has access to an encryption oracle, $\mathcal{O}$, and can encrypt any plaintext he wishes to see encrypted. The goal of the attacker is to distinguish pairs of ciphertexts based on the plaintext they encrypt. Therefore, the attacker, $\mathcal{A}$, consists of two functions $\mathcal{A} = (A_1, A_2)$. A_1 chooses a vector of plaintext values and A_2 tries to distinguish which plaintext was encrypted. Indistinguishability under *Chosen-Plaintext Attack* is captured through Experiment 6.

Experiment 6 : $\mathbf{Exp}_{\mathrm{ES}}^{\text{IND-CPA}}(A)$

$\tau \xleftarrow{\$} \mathcal{T}$
$(x_0, x_1) \xleftarrow{\$} A_1(\mathcal{X})$
$b \xleftarrow{\$} \{0, 1\}$
$y \leftarrow \mathcal{E}nc(\tau, x_b)$
$b' \xleftarrow{\$} A_2(\mathcal{O}, (x_0, x_1), y)$ ▷ $\mathcal{O}$ is the encryption oracle.
if $b' = b$ **then return** 1
else return 0

The adversary using A_1, chooses two plaintext values, (x_0, x_1) and gives it to the experiment. The experiment flips a coin and randomly chooses to encrypt one of them, $y = \mathcal{E}nc(\tau, x_b)$. The attacker is given an encryption oracle, $\mathcal{O}$, (x_0, x_1) and y. Based on y and the information he may extract by using the encryption oracle, the attacker tries to find out whether x_0 or x_1 was encrypted. The probability that an attacker can succeed in this experiment is denoted as the IND-CPA advantage, $Adv_{\mathrm{ES}}^{\text{IND-CPA}}(\mathcal{A})$ and is optimal if an attacker cannot do better than to randomly guess b, i.e. $Adv_{\mathrm{ES}}^{\text{IND-CPA}}(\mathcal{A}) \leq \frac{1}{2}$. Optimally the probability will get less than $\frac{1}{2}$ once the experiment is repeated multiple times.

5.2 Query Log Attack (QLA)

The main goal of an encryption scheme is to protect the data. In the previous section we have revisited the adversary models that capture the indistinguishability property of an encryption scheme. In a database application, other than the ciphertext itself, there are other scenarios that can lead to breaking an encryption scheme. As already motivated in the introduction of this chapter, database queries can weaken their underlying encryption scheme by totally or partially revealing their secrets. So far, there has been a lot of work dedicated in improving the security of an encryption scheme, however none of those schemes have been analyzed against query log attacks. In this section we define

a standard framework that enables query log security analysis for any given encryption scheme that is to be used in database applications.

In order to define a new security notion, it is crucial to formalize the assumptions, goals and attacker scenarios. Thus, in this section we define a few new paradigms that will be used in the query log attack framework. This thesis explicitly looks at database applications, and thus whenever the term "query" is used, an "SQL Query" is meant. Nevertheless, the query log attack can be used also for other type of queries (e.g. search queries in information retrieval). Consider Table 5.1 as a running example. The following query is sent by the client to the trusted component:

```
SELECT name FROM customer WHERE name = 'Messi'
```

Since the database is encrypted, the client's query is intercepted by the trusted component and rewritten in a way that it can be processed on encrypted data. For example if the `name` column is encrypted using OPE, the above query is rewritten by the trusted component as follows:

```
SELECT name_enc FROM customer_enc WHERE name_enc = 2564
```

Definition 1. ***Query Rewrite Function*** *($\mathcal{QRF}$). The Query Rewrite Function is a function that takes a plaintext query that is submitted by the client, and an encryption scheme $\mathcal{ES}$ as input and outputs a rewritten query.*

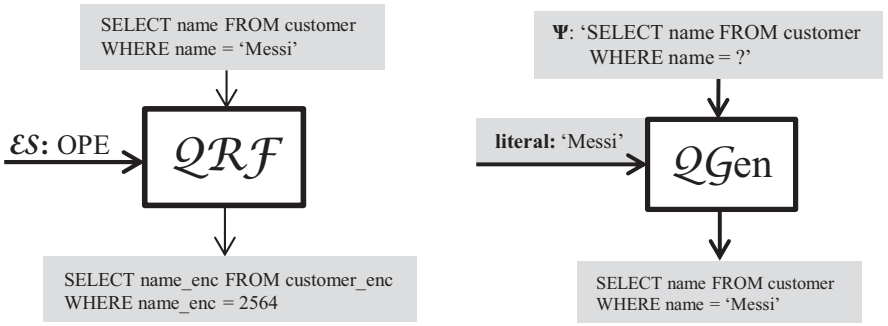
Depending on the encryption scheme and the query rewrite mechanism, the query submitted by the client will be rewritten differently. For example, having data encrypted in the database with OPE, the query of our running example is usually rewritten as follows and shown in Figure 5.2a:

```
SELECT name FROM customer_enc
  WHERE name_enc = Enc_OPE('Messi')
```

Definition 2. ***Query Generator*** *($\mathcal{QG}en$). The Query Generator is a function that takes a vector of query literals, either plaintext or ciphertext, and an SQL statement (analog to Prepared Statements), Ψ, as input and generates an SQL query.*

Please note than unlike $\mathcal{QRF}$, $\mathcal{QG}en$ is independent from the encryption scheme. $\mathcal{QG}en$ can also be inversed, i.e. $\mathcal{QG}en^{-1}$ takes a query and extracts its literals using a parser. Figure 5.2a shows how $\mathcal{QG}en$ works for our running example.

Reduction Proof. We use reduction to prove the lemmas in this section. Therefore, here we give a brief explanation on how it works. To prove that Problem B (with unknown complexity) is at least as hard as Problem A (with known complexity), one



(a) $\mathcal{QRF}$ rewrites plaintext queries. (b) $\mathcal{QG}en$ generates queries

Figure 5.2: Query Functions

solves Problem A using the solver of Problem B. It suffices to find an efficient[2] transformation, ϕ between the Solver of Problem B, $\mathcal{T}$ into the Solver of Problem A, $\mathcal{S}$, i.e. $\mathcal{S} = \phi(\mathcal{T})$.

Prior to formalizing the query log attack through experiments, it is important to state our assumptions about the adversary's knowledge, state the goal of the experiment, and describe the scenario through which the adversary is interacting with the experiment.

- Assumptions: The adversary, $\mathcal{A}$, has compromised the database server, i.e. $\mathcal{A}$ has access to all encrypted data, *rewritten query logs* and every medium on the server where data can be stored and processed (e.g. disk, main memory and cache).

- Goal: Distinguishing between the queries submitted through the real system by the client and queries generated by a simulated system that can only interact with the encrypted database.

- Scenario: In the remainder of this section we define three concrete query log adversary models namely, the *Simple Query Attack, Known-Query Attack* and *Chosen-Query Attack*. These attacks are based on attacks on ciphers revisited in Section 5.1, but they are extended for query log attacks and database applications.

Notation. For the sake of simplicity, let Ψ denote the structure of an SQL query, i.e. SQL operators and the schema that should be used to generate a query. Let $\mathcal{R}$ be the real system that generates queries using a query rewrite function. Let $\mathcal{S}$ be the simulated system that generates queries using the encrypted database. $\mathcal{Q}_y$ denotes the rewritten query logs that have been accumulated on the untrusted database server. In general, let q_x denote a plain query submitted by the client and q_y denote a rewritten query submitted by the trusted component to the encrypted database.

[2]Polynomial-time in the size of the input

5.2.1 Query Rewrite Properties

A $\mathcal{QRF}$ can be implemented in several ways for a given encryption scheme. Each way may fulfill a functional or non-functional requirement. For example, a query rewrite function can be implemented to maximize security, or to minimize the network cost and query processing overhead. Here we introduce three classes and formally define each class:

Definition 3. ***Encryption-Equivalent*** $\mathcal{QRF}$**.** *A $\mathcal{QRF}$ is encryption-equivalent with respect to an encryption scheme $\mathcal{E}nc^{ES}$, denoted as $\mathcal{QRF} \equiv \mathcal{E}nc^{ES}$, if it does nothing but encrypt the literals in the query. Formally, for an arbitrary query q_x:*

$$\mathcal{QG}en^{-1}(\mathcal{QRF}(q_x)) = \mathcal{E}nc^{ES}(\mathcal{QG}en^{-1}(q_x))$$

For instance, given Table 5.1, the following `MIN` query:

```
SELECT MIN(name) FROM customer
```

would be rewritten on an MOPE-encrypted database as:

```
SELECT name_enc FROM customer_enc
```

Since $\mathcal{QG}en^{-1}(\mathcal{QRF}(q_x)) = \mathcal{E}nc^{ES}(\mathcal{QG}en^{-1}(q_x)) = \emptyset$, such a rewrite function would be encryption-equivalent. However, using an encryption-equivalent rewrite function in this case degrades the performance drastically since the whole `customer_enc` table needs to be shipped and post-processed on the client to find the minimum `name`.

Definition 4. ***Leaky*** $\mathcal{QRF}$**.** *A query rewrite function is leaky if it reveals more than calling $\mathcal{E}nc^{ES}$ on the query literals. In other words:*

$$\mathcal{QG}en^{-1}(\mathcal{QRF}(q_x)) \supset \mathcal{E}nc^{ES}(\mathcal{QG}en^{-1}(q_x))$$

The most performant way to rewrite the `MIN` query on MOPE is:

```
SELECT MIN(name_enc) FROM customer_enc
WHERE name_enc > 3000
```

Since $\mathcal{QG}en^{-1}(\mathcal{QRF}(q_x)) = \{3000\}$ whereas $\mathcal{E}nc^{ES}(\mathcal{QG}en^{-1}(q_x)) = \emptyset$, based on Definition 4, this way of rewriting the MIN query is leaky.

Definition 5. ***Absorbing*** $\mathcal{QRF}$**.** *A query rewrite function is absorbing if it reveals less than calling $\mathcal{E}nc^{ES}$ on the query literals. In other words:*

$$\mathcal{QG}en^{-1}(\mathcal{QRF}(q_x)) \subset \mathcal{E}nc^{ES}(\mathcal{QG}en^{-1}(q_x))$$

For instance, given Table 5.1, the following query:

```
SELECT name FROM customer
WHERE name = 'Messi'
```

can be rewritten on an OPE-encrypted database as:

```
SELECT name_enc FROM customer_enc
```

In this case $\mathcal{E}nc^{ES}(\mathcal{QG}en^{-1}(q_x)) = 2564$ whereas $\mathcal{QG}en^{-1}(\mathcal{QRF}(q_x)) = \emptyset$, since the former set is indeed a superset of the latter, this means that the QRF used to rewrite q_x is absorbing.

In the next section, we define the concept of indistinguishability under query log attack. We will show how an adversary wins a query log attack with the help of the leaky queries in the rewritten query logs.

5.2.2 Indistinguishability under QLA

The notion of *Indistinguishability* is a problem of distinguishing between the output of two systems, $\mathcal{R}$ and $\mathcal{S}$, using a distinguisher, $\mathcal{D}$. One can prove that two systems $\mathcal{R}$ and $\mathcal{S}$ are equivalent, denoted as: $\mathcal{R} \equiv \mathcal{S}$ if they have the same behavior. In our definition of Query Log Attack, the goal of the adversary is to distinguish between two systems $\mathcal{R}$ and $\mathcal{S}$ as shown in Figure 5.3. System $\mathcal{R}$ is the *Real System* where the actual clients submit plaintext queries, q_x, then the *Trusted Component* rewrites these queries into q_y and submits them to the encrypted database, DB'. System $\mathcal{S}$ is the *Simulated System* where a query simulator interacts with the encrypted database, DB', and generates queries, q_y^s.

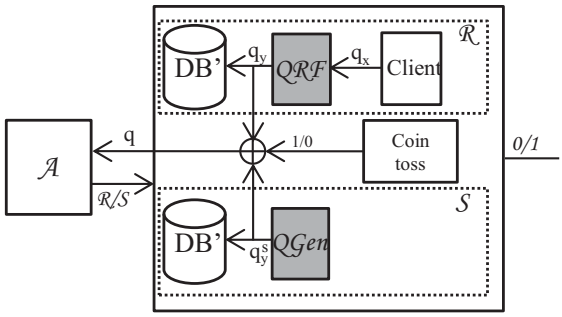


Figure 5.3: $\mathcal{A}$ needs to distinguish between system $\mathcal{R}$ and system $\mathcal{S}$

The notion of *Indistinguishability under Query Log Attack* denotes the case where $\mathcal{A}$ is unable to distinguish the *Real System* from the *Simulated System* when randomly given a query, q from either, $\mathcal{R}$ or $\mathcal{S}$. In other words, $\mathcal{A}$'s advantage (probability) in correctly returning the origin of q should be no better than a random guess:

$$Adv^{\text{IND}}_{\mathcal{RS}}[D(q)] \leq \frac{1}{2}$$

We assume that the queries do not contain any *client information*, and both systems receive queries with the same timestamp, frequency and operator distribution. For example, if *System R* only has *join queries* then also *System S* only generates *join queries*. After defining the goal that the adversary has to fulfill, we proceed to the next section where we define concrete attacker scenarios involving query logs.

5.2.3 Simple Query Attack (SQA)

In this scenario we assume that in addition to the encrypted data in the cloud, the adversary has access to the rewritten query logs, $\mathcal{Q}'$ on the database server. This scenario matches our initial honest-but-curious adversary assumption who has full access on the machine where the data resides like a database or system administrator. The idea of *Simple Query Attack* is similar to the *Ciphertext-only Attack* in which the adversary has access to a ciphertext and needs to guess its underlying plaintext. We have revisited the ciphertext-only attack in Section 5.1.1 in this chapter.

To give an example for such an attack consider the `name` column from the `customer` table has been encrypted using MOPE [15] as shown in Table 5.1. The idea of MOPE is to hide the total order by hiding the beginning of the domain. This way, the attacker cannot decipher the encrypted database even if the attacker has precise knowledge of the domain. The following example shows how MOPE can lose its security advantages over traditional, weak OPE when the attacker has access to the query log. The client submits a query that asks for the first customer in the database in alphabetical order:

```
SELECT MIN(name) FROM customer
```

There are many ways to rewrite this query so that it can be processed on an MOPE-encrypted *Name* column in the cloud without decrypting the *Name* column. The most natural approach is to rewrite this query in the following way:

```
SELECT MIN(name_enc) FROM customer_enc WHERE name_enc > 3000
```

This rewritten query can be processed entirely in the cloud. The result is the (encrypted) tuple with ciphertext 3178 which is returned from the cloud to the client. The client decrypts this tuple and receives the correct result: “Benzema”. This approach has the best possible performance: The query can be processed in the cloud on the encrypted database in (almost) the same way as the original query on a plaintext database. Furthermore, only a single tuple is shipped from the cloud to the client, thereby incurring no additional communication cost. Unfortunately, however, this rewritten query leaks the *offset* used by MOPE in this example. As a result, the attacker knows that the cipher 3178 corresponds to the alphabetically first customer and, as a result, MOPE degrades to OPE which is generally perceived to be a very weak encryption scheme.

In order to detect such threats in a rewritten query, we define an experiment to capture the winning probability of an adversary launching a *Simple Query Attack*. We define

the Indistinguishability under *Simple Query Attack* as $Exp^{\text{SQA}}_{\text{IND}}(\mathcal{A})$ shown in Experiment 7. The advantage of $\mathcal{A}$ is defined as the probability to succeed in Experiment 7.

$$Adv^{\text{SQA}}_{\text{IND}}(\mathcal{A}) = Pr[Exp^{\text{SQA}}_{\text{IND}}(\mathcal{A}) = 1]$$

Experiment 7 : $\mathbf{Exp}^{\text{SQA}}_{\text{IND}}(\mathcal{A})$

$q_0 \leftarrow \mathcal{QG}en(x \xleftarrow{\$} \mathcal{X}, \Psi)$
$q_1 \leftarrow \mathcal{QG}en(x' \xleftarrow{\$} \mathcal{X}, \Psi)$ ▷ $\|x\| = \|x'\|$
$q'_0 \leftarrow \mathcal{QRF}(q_0)$ ▷ Query from $\mathcal{R}$
$q'_1 \leftarrow \mathcal{QG}en(\mathcal{E}nc(\mathcal{QG}en^{-1}(q_1)))$ ▷ Query from $\mathcal{S}$
$b \xleftarrow{\$} 0, 1$ ▷ Coin-toss
$q' \leftarrow q'_b$
$b' \xleftarrow{\$} \mathcal{A}(q, (q_0, q_1), \mathcal{Q}', DB')$ ▷ $\mathcal{A}$ needs to distinguish
if $b == b'$ **then return** 1
else return 0

In the beginning, the experiment generates two plaintext queries, q_0 and q_1 using $\mathcal{QG}en(\mathcal{X})$. Please note that both queries are generated using the same SQL operators. q_0 is then given to $\mathcal{QRF}(q_0)$ to be rewritten using the real system, $\mathcal{R}$. q_1 is given to $\mathcal{QG}en^{-1}(q_1)$, so that the plaintext literals of q_1 can be extracted. The extracted plaintext literals are encrypted and given to $\mathcal{QG}en(\mathcal{Y})$ to generate a query from the encrypted data using the same SQL operators as in q_1. From these two queries, q'_0 and q'_1, one is selected at random and given to the adversary. The adversary needs to distinguish whether the given query, q, belongs to the *Real System*, $\mathcal{R}$, or the *Simulated System*, $\mathcal{S}$ and output its decision. The output of the experiment is 1 if the adversary is able to decide correctly, otherwise 0. The probability that an attacker can succeed in this experiment is denoted as the IND-QoA advantage, $Adv^{\text{IND-SQA}}_{\text{ES}}(A)$ and is optimal if an attacker cannot do better than to randomly guess b, i.e. $Adv^{\text{IND-SQA}}_{\text{ES}}(A) \leq \frac{1}{2}$. The probability is less than $\frac{1}{2}$ because this experiment can be repeated multiple times.

Lemma 2. *If a database encryption scheme, $\mathcal{ES}$, is IND-SQA, then it is also IND-CoA.*

Proof. We need to show that we can win the IND-SQA experiment, if we can win IND-CoA experiment for an arbitrary encryption scheme, $\mathcal{ES}$. To prove this, we need to solve the IND-SQA experiment with the solver of the IND-CoA experiment. Suppose $\mathcal{A}$ is an adversary with non-trivial *IND-CoA* advantage against $\mathcal{ES}$. We construct an IND-SQA adversary, $\mathcal{A}_{\text{SQA}}$, against $\mathcal{ES}$. As per definition in the IND-SQA experiment 7, $\mathcal{A}_{\text{SQA}}$ is given a rewritten query, q', two plaintext queries, (q_0, q_1), the encrypted database, DB', and the rewritten query logs, $\mathcal{Q}'$. $\mathcal{A}_{\text{SQA}}$ first runs $QGen^{-1}(q_0)$ to extract x_0, then $QGen^{-1}(q_1)$ to extract x_1 and eventually $QGen^{-1}(q')$ to extract y. Eventually, $\mathcal{A}_{\text{SQA}}$

runs $\mathcal{A}((x_0, x_1), y)$. Thus we have found an efficient way to solve $\mathcal{A}_{\text{SQA}}$ with $\mathcal{A}$. $\mathcal{A}_{\text{SQA}}$ is efficient since it has used $QGen^{-1}(q)$ which its complexity is linear to its input size. □

Corollary 2. *In order to win the IND-SQA experiment, an adversary needs to either distinguish between the ciphertext literals in the query (see Lemma 2), or distinguish the queries generated by the simulated system from the ones that are generated by the real system after analyzing the query logs.*

In Experiment 7, the attacker needs to distinguish the queries generated by a $\mathcal{QRF}$ instance from the queries that are randomly generated from the encrypted data. Upon receiving the query q to distinguish, the adversary computes and compares $Prob(b = 0|\mathcal{Q}_y)$ versus $Prob(b = 1|\mathcal{Q}_y)$. If the $\mathcal{QRF}$ is leaky then the query logs contain additional information that make $Prob(b = 0|\mathcal{Q}_y) \neq Prob(b = 1|\mathcal{Q}_y)$. This inequality helps the attacker to distinguish between $\mathcal{S}$ and $\mathcal{R}$ with a greater advantage than a random adversary.

5.2.4 Known-Query Attack (KQA)

In this scenario we assume the adversary has access to the rewritten query logs, $\mathcal{Q}'$, and a certain number of plain and rewritten query pairs, in addition to the encrypted data on the database server. The idea of *Known-Query Attack* is similar to the *Known-Plaintext Attack* in which the adversary has access to a certain number of plaintext-ciphertext pairs. We have revisited the known plaintext attack in Section 5.1.2 in this chapter. An adversary can guess an original query from its rewritten version by collecting additional information from the query logs. For example, knowing the client who has originated the query, or the time-stamp of a query, or even the frequency distribution of various queries can lead to a known-query attack. In general, statistical and linkage attacks[3] on query logs can be classified as *Known-Query Attacks*.

To give an example for such an attack consider the `name` column from the `customer` table has been encrypted using MOPE as shown in Table 5.1. The idea of MOPE is to hide the total order by hiding the beginning of the domain. This way, the attacker cannot decipher the encrypted database even if the attacker has precise knowledge of the domain. The following example shows how MOPE can lose its security advantages over traditional, weak OPE when the attacker has access to even one pair of plain and rewritten query. For instance, the client knows that the following rewritten query:

```
SELECT name_enc FROM customer_enc WHERE name_enc = 4567
```

corresponds to the following plaintext query:

[3] Attacks that are based on the adversary's background knowledge.

```
SELECT name FROM customer WHERE name = 'Messi'
```

Before analyzing the query logs, we will see that MOPE turns into the weak OPE, because based on this known pair of plain-rewritten query, the adversary could extract a plaintext-ciphertext pair. It is proven in [15] that MOPE is not safe against known-plaintext attack and consequently degrades to OPE.

In this model, we need to take care of two threats. The first and obvious threat is to deal with a simple query attack, introduced in the previous section, since attacker has access to the query logs. The second threat is the eventual known-plaintext attack that will be possible after leaking a few plain and rewritten query pairs. We define the Indistinguishability under *Known-Query Attack* as $Exp^{\text{KQA}}_{\text{IND}}(\mathcal{A})$ shown in Experiment 8. The advantage of $\mathcal{A}$ is defined as the probability to succeed in Experiment 8.

$$Adv^{\text{KQA}}_{\text{IND}}(\mathcal{A}) = Pr[Exp^{\text{KQA}}_{\text{IND}}(\mathcal{A}) = 1]$$

Experiment 8 : $\mathbf{Exp}^{\text{KQA}}_{\text{IND}}(\mathcal{A})$

$\mathbf{q_p} \leftarrow \mathcal{QG}en(\mathcal{X}, \Psi')$ ▷ Vector of plain queries
$\mathbf{q_c} \leftarrow \mathcal{QRF}(\mathbf{q_p})$ ▷ Vector of rewritten queries from previous step
$q_0 \leftarrow \mathcal{QG}en(x \xleftarrow{\$} \mathcal{X}, \Psi)$
$q_1 \leftarrow \mathcal{QG}en(x' \xleftarrow{\$} \mathcal{X}, \Psi)$ ▷ $|x| = |x'|$
$q'_0 \leftarrow \mathcal{QRF}(q_0)$ ▷ Query from $\mathcal{R}$
$q'_1 \leftarrow \mathcal{QG}en(\mathcal{E}nc(\mathcal{QG}en^{-1}(q_1)))$ ▷ Query from $\mathcal{S}$
$b \xleftarrow{\$} 0, 1$ ▷ Coin-toss
$q' \leftarrow q'_b$
$b' \xleftarrow{\$} \mathcal{A}(q', (q_0, q_1), \mathbf{q_p}, \mathbf{q_c}, \mathcal{Q}', DB')$
if $b == b'$ **then return** 1
else return 0

First $\mathcal{QG}en$ is called to generate a vector of plaintext queries, $\mathbf{q_p}$. $\mathcal{QRF}$ is then called to rewrite the previously generated plaintext queries, $\mathbf{q_p}$, generating their corresponding encrypted queries $\mathbf{q_c}$. At this stage the experiment has generated the set of plain-rewritten query pairs that will be given to the adversary. Afterwards, the experiment generates two plaintext queries, q_0 and q_1 using $\mathcal{QG}en(\mathcal{X})$. q_0 is given to $\mathcal{QRF}(q_0)$ to be rewritten using the real system, $\mathcal{R}$. On the other hand, the plaintext literals of q_1 are being extracted by $\mathcal{QG}en^{-1}(q_1)$, encrypted and then the ciphertext is given to $\mathcal{QG}en(\mathcal{Y})$ to generate a query from the encrypted data, q'_1. From these two rewritten queries, q'_0 and q'_1, one is selected at random and given to the adversary. Moreover, $\mathbf{q_p}$ and $\mathbf{q_c}$ are also given to the adversary this time. Adversary needs to distinguish whether the given query, q, belongs to the *Real System*, $\mathcal{R}$, or the *Simulated System*,

$\mathcal{S}$ and output its decision. The output of the experiment is 1 if the adversary is able to decide correctly, otherwise 0. The probability that an attacker can succeed in this experiment is denoted as the IND-KQA advantage, $Adv_{\text{ES}}^{\text{IND-KQA}}(\mathcal{A})$ and is optimal if an attacker cannot do better than to randomly guess b, i.e. $Adv_{\text{ES}}^{\text{IND-KQA}}(\mathcal{A}) \leq \frac{1}{2}$. The probability is less than $\frac{1}{2}$ because this experiment can be repeated multiple times.

Lemma 3. *A database encryption scheme, $\mathcal{ES}$, is IND-KQA, if $\mathcal{ES}$ is IND-KPA.*

Proof. We need to show that we can win the IND-KQA experiment, if we can win IND-KPA experiment for an arbitrary encryption scheme, $\mathcal{ES}$. To prove this, we need to solve the IND-KQA experiment with the solver of the IND-KPA experiment. Suppose $\mathcal{A}$ is an adversary with non-trivial *IND-KPA* advantage against $\mathcal{ES}$. We construct an IND-KQA adversary, $\mathcal{A}_{\text{KQA}}$, against $\mathcal{ES}$. As per definition in the IND-KQA experiment 8, $\mathcal{A}_{\text{KQA}}$ is given a vector of plain and rewritten queryies, a rewritten query, q', two plaintext queries, (q_0, q_1), the encrypted database, DB', and the rewritten query logs, $\mathcal{Q}'$. $\mathcal{A}_{\text{KQA}}$ first runs $QGen^{-1}(q_0)$ to extract x_0, then $QGen^{-1}(q_1)$ to extract x_1 and eventually $QGen^{-1}(q')$ to extract y. It additionally runs $QGen^{-1}(\mathbf{q_p})$ and $QGen^{-1}(\mathbf{q_c})$ to extract a vector or plaintext-ciphertext pairs, **x** and **y**. Eventually, $\mathcal{A}_{\text{KQA}}$ runs $\mathcal{A}(\mathbf{x}, \mathbf{y}, (x_0, x_1), y)$. Thus we have found an efficient way to solve $\mathcal{A}_{\text{KQA}}$ with $\mathcal{A}$. $\mathcal{A}_{\text{KQA}}$ is efficient since it has used $QGen^{-1}(q)$ which its complexity is linear to its input size.

□

Corollary 3. *In order to win the IND-KQA experiment, an adversary needs to either distinguish between the ciphertext literals in the query (see Lemma 3) by using a number of plain and rewritten queries, or distinguish the queries generated by the simulated system from the ones that are generated by the real system by analyzing the query logs.*

5.2.5 Chosen-Query Attack(CQA)

In this scenario we assume the adversary has access to the rewritten query logs, $\mathcal{Q}'$, and a query rewrite function, $\mathcal{QRF}$, in addition to the encrypted data on the database server. The idea of *Chosen-Query Attack* is similar to the *Chosen-Plaintext Attack* in which the adversary can adaptively choose the plaintexts he wants to see encrypted. This scenario can happen if for example the client and the database administrator collude, which is not the primary assumption of this thesis but still it is important to cover it in this section. We define the Indistinguishability under *Chosen-Query Attack* as $Exp_{\text{IND}}^{\text{CQA}}(\mathcal{A})$ shown in Experiment 9. The advantage of $\mathcal{A}$ is defined as the probability to succeed in Experiment 9.

$$Adv_{\text{IND}}^{\text{CQA}}(\mathcal{A}) = Pr[Exp_{\text{IND}}^{\text{CQA}}(\mathcal{A}) = 1]$$

In the beginning, the experiment generates two plaintext queries, q_0 and q_1 using $\mathcal{QG}en(x \xleftarrow{\$} \mathcal{X}, \Psi)$ and $\mathcal{QG}en(x' \xleftarrow{\$} \mathcal{X}, \Psi)$. Please note that both queries are generated

Experiment 9 : $\mathbf{Exp}^{\text{CQA}}_{\text{IND}}(\mathcal{A})$

$q_0 \leftarrow \mathcal{QG}en(x \xleftarrow{\$} \mathcal{X}, \Psi)$
$q_1 \leftarrow \mathcal{QG}en(x' \xleftarrow{\$} \mathcal{X}, \Psi)$ ▷ $|x| = |x'|$
$q'_0 \leftarrow \mathcal{QRF}(q_0)$ ▷ Query from $\mathcal{R}$
$q'_1 \leftarrow \mathcal{QG}en(\mathcal{E}nc(\mathcal{QG}en^{-1}(q_1)), \Psi)$ ▷ Query from $\mathcal{S}$
$b \xleftarrow{\$} 0, 1$ ▷ Coin-toss
$q' \leftarrow q'_b$
$b' \xleftarrow{\$} \mathcal{A}(q', (q_0, q_1), \mathcal{QRF}, \mathcal{Q}', DB')$
if $b == b'$ **then return** 1
else return 0

using the same SQL operators and schema. q_0 is then given to $\mathcal{QRF}(q_0)$ to be rewritten using the real system, $\mathcal{R}$. q_1 is given to $\mathcal{QG}en^{-1}(q_1)$, so that the plaintext literals of q_1 can be extracted. The extracted plaintext literals are encrypted and given to $\mathcal{QG}en(\mathcal{Y}, \Psi)$ to generate a query from the encrypted data using the same SQL operators as in q_1. From these two queries, q'_0 and q'_1, one is selected at random and given to the adversary. The adversary needs to distinguish whether the given query, q, belongs to the *Real System*, $\mathcal{R}$, or the *Simulated System*, $\mathcal{S}$ and output its decision. This time the adversary additionally has access to a $\mathcal{QRF}$ that can help him distinguish q'. The output of the experiment is 1 if the adversary is able to decide correctly, otherwise 0. The probability that an attacker can succeed in this experiment is denoted as the IND-CQA advantage, $Adv^{\text{IND-CQA}}_{\text{ES}}(\mathcal{A})$ and is optimal if an attacker cannot do better than to randomly guess b, i.e. $Adv^{\text{IND-CQA}}_{\text{ES}}(\mathcal{A}) \leq \frac{1}{2}$. The probability is less than $\frac{1}{2}$ because this experiment can be repeated multiple times.

Lemma 4. *A database encryption scheme, $\mathcal{ES}$, is IND-CQA, if $\mathcal{ES}$ is IND-CPA and the attacker's advantage in winning Experiment 9 is no better than a random guess between the real and simulated system.*
$Adv^{\textit{IND-CQA}}_{\textit{ES}}(\mathcal{A}) \leq \frac{1}{2}$

Proof. We need to show that we can win the IND-CQA experiment, if we can win IND-CPA experiment for an arbitrary encryption scheme, $\mathcal{ES}$. To prove this, we need to solve the IND-CQA experiment with the solver of the IND-CPA experiment. Suppose $\mathcal{A}$ is an adversary with non-trivial *IND-CPA* advantage against $\mathcal{ES}$. We construct an IND-CQA adversary, $\mathcal{A}_{\text{CQA}}$, against $\mathcal{ES}$. As per definition in the IND-CQA experiment 9, $\mathcal{A}_{\text{CQA}}$ is given a $\mathcal{QRF}$, a rewritten query, q', two plaintext queries, (q_0, q_1), the encrypted database, DB', and the rewritten query logs, $\mathcal{Q}'$. $\mathcal{A}_{\text{CQA}}$ first runs $QGen^{-1}(q_0)$ to extract x_0, then $QGen^{-1}(q_1)$ to extract x_1 and eventually $QGen^{-1}(q')$ to extract y. It additionally creates an encryption oracle by using $\mathcal{O}(\mathcal{X}) = \mathcal{QG}en^{-1}(\mathcal{QRF}(\mathcal{QG}en(\mathcal{X})))$. Eventually, $\mathcal{A}_{\text{CQA}}$ runs $\mathcal{A}(\mathcal{O}, (x_0, x_1), y)$. Thus, we have found an efficient way to solve

$\mathcal{A}_{\text{CQA}}$ with $\mathcal{A}$. $\mathcal{A}_{\text{CQA}}$ is efficient since it is using $\mathcal{QG}en^{-1}$, $\mathcal{QRF}$ and $\mathcal{QG}en$ which their complexity is linear to their input size.

□

Corollary 4. *In order to win the IND-CQA experiment, an adversary needs to either distinguish between the ciphertext literals in the query (see Lemma 4) by using the query rewrite function, or distinguish the queries generated by the simulated system from the ones that are generated by the real system by analyzing the query logs.*

5.3 Simulatable Queries

In the introduction we have shown that MOPE and Prob-OPE are not robust under query log attacks because their queries were leaky, i.e., required additional knowledge to be revealed. This makes them distinguishable from the simulated system. Nevertheless, some queries on MOPE- or Prob-OPE-encrypted databases are indistinguishable from the simulated queries. This is owe to the fact that, same instance of a $\mathcal{QRF}$ may produce leaky, encryption-equivalent, or aborbing queries when given different SQL operators (in our case the Ψ parameter). In general, non-leaky queries are called simulatable queries.

Definition 6. ***Simulatable Query.*** *A query q_x is simulatable if:*

$$\mathcal{QG}en^{-1}(\mathcal{QRF}(q_x)) \subseteq \mathcal{E}nc^{ES}(\mathcal{QG}en^{-1}(q_x))$$

For example, assume the client submits the following query to a Prob-OPE-encrypted database:

```
SELECT name FROM customer
ORDER BY name
```

In this case, the query rewrite function just maps the schema names to the encrypted schema names and rewrites the query as follows:

```
SELECT name_enc FROM customer_enc
ORDER BY name_enc
```

5.4 Related Work

Anonymization. In the are of data anonymization, query log attack has been referred to as the ability of the adversary to being able to spot a specific data contributor by analyzing the query logs, this area has been addressed by [33, 49]. Nevertheless, our

concern is to identify the queries that can influence the strength of a given encryption scheme and to fix them. Another line of work from [34, 46] on the query log privacy is dedicated to anonymizing the search query logs so that these query logs are publishable and do not leak any information about an individual. The data to protect in this scenario is the query log.

Private Information Retrieval. This mechanism has been first introduced in [19] and gained a lot of attention. Private Information Retrieval allows users to retrieve an item from a server without revealing which item is retrieved. Our goal is not to hide what information the user is interested in from the database sever but to protect the encryption scheme against query log analysis.

Database Intrusion Detection. In [21, 37, 42] the goal is to detect whether a client querying a database has turned malicious or not. This is achieved by looking for access pattern anomalies through the queries submitted to a database. These mechanisms are orthogonal to our approach in identifying malicious queries, they are not necessarily tackling an encrypted database and the danger query logs impose on the encryption scheme. Nevertheless, their methods can be used in conjunction with our framework to detect a *Chosen-Query Attack* introduced in Section 5.2.5. Our framework prevents dangerous queries to be submitted to the database, whereas these methods detect whether a dangerous query has been submitted to the database or not.

Query Authentication [11, 45]. In query authentication the outsourced database needs to guarantee the integrity (correctness and completeness), and authenticity of the query results. This scenario is about checking the results rather than controlling the queries and evaluating their information leakage.

5.5 Conclusion

In this chapter we showed why it is important to consider query logs when analyzing the security of an encrypted database. For the first time we have introduced a framework to analyze the security of query logs for any given encryption scheme. Along the way, we have introduced a few notions and tools such as a Query Rewrite Function, a Query Generator, a Real System, and a Simulated System to be used in our query log attacker models. In the end, we have concretely proved:

- In order to win the IND-SQA experiment, an adversary needs to either distinguish between the ciphertext literals in the query (see Lemma 2), or distinguish the queries generated by the simulated system from the ones that are generated by the real system after analyzing the query logs.

- In order to win the IND-KQA experiment, an adversary needs to either distinguish between the ciphertext literals in the query (see Lemma 3) by using a number of plain and rewritten queries, or distinguish the queries generated by the simulated

system from the ones that are generated by the real system by analyzing the query logs.

- In order to win the IND-CQA experiment, an adversary needs to either distinguish between the ciphertext literals in the query (see Lemma 4) by using the query rewrite function, or distinguish the queries generated by the simulated system from the ones that are generated by the real system by analyzing the query logs.

As already mentioned in the introduction, there are a dozen of database encryption systems and schemes proposed in different communities. Nevertheless, adversary models that capture the query log security have never been defined or proposed before. The conclusions of this chapter can be used to analyze the query log security for any existing or upcoming database encryption system or scheme.

6

Order Preserving Encryption (OPE)

An order-preserving symmetric encryption (OPE) scheme is a deterministic symmetric encryption scheme whose encryption algorithm produces ciphertexts that preserve numerical ordering of the plaintexts. This property makes OPE very attractive for database applications, since it allows efficient range and rank query processing on encrypted data. However, the order relationship between plaintext and ciphertext remains intact after encryption, making it an easy target for a *Domain Attack*. Moreover, being deterministic, makes OPE particularly vulnerable against *Frequency Attacks*. OPE was first proposed in the database community by Agrawal et al. [6], and treated cryptographically for the first time by Boldyreva et al. in [14], followed by [15, 41, 48, 69] in search for an "ideal object". These solutions however focus on typical adversaries in the world of cryptography. However, the problem of dealing with *Domain* and *Frequency Attacks* remains. In this chapter we will revisit two interesting order-preserving encryption scheme variants introduced in [14,15]. Randomized OPE is a stateless randomized order preserving encryption scheme introduced in [14], and Modular OPE is a modular order preserving encryption scheme introduced in [15] where the ciphertexts form a ring and therefore the total order is not leaking. We additionally show how the query rewrite would work for them, and analyze their security against the database adversary models introduced in Chapter 4. In the coming chapters we show how these OPE variants can be composed with new concepts and ideas to build stronger encryption schemes.

Notation. Let $\mathcal{X}$ be the set of plaintext values from a finite domain, and $\mathcal{Y}$ be the set of ciphertext values. The size of $\mathcal{X}$ is denoted as $X = |\mathcal{X}|$; the same applies for the size of $\mathcal{Y}$, $Y = |\mathcal{Y}|$. Plaintext elements are denoted as x and ciphertext elements as y.

Name (plaintext)	Name (OPE)	Name (MOPE [15])
Benzema	1354	3178
Messi	2564	4567
Messi	2564	4567
Neymar	3129	1890
Ronaldo	4980	2601

Table 6.1: Example: ROPE vs. MOPE

Additionally, we denote the Key set to be $\mathcal{K}eys$ and $K \xleftarrow{\$} \mathcal{K}eys$ denotes that a key, K, is selected uniformly at random from $\mathcal{K}eys$. The $\$$ sign on top of the $\leftarrow$ depicts that the selection was uniformly at random. $\mathcal{K}$ is a randomized algorithm that creates a random key from a finite set. Let $\mathcal{E}nc$ be the encryption function having a key, K, and a plaintext value, x, as its input parameters; thus, we have: $y = \mathcal{E}nc(K, x)$. Symmetrically, $\mathcal{D}ec$ will be the decryption function, taking y and K as input, yielding: $x = \mathcal{D}ec(K, y)$.

6.1 Randomized OPE (ROPE)

In [14], a stateless and pseudo-randomized order preserving encryption scheme has been proposed based on the hyper geometric distribution. In this encryption scheme, a pseudo-random lazy sampling algorithm has been used to map range gaps to domain gaps in a recursive, binary search manner to determine the image of an input x. The efficiency and security of this encryption algorithm has been analyzed and the correctness has been proven in [14, 15]. Table 6.1 gives an example of a set of customer names encrypted with *ROPE*.

Construction 4. *A Random Order-Preserving Encryption scheme,* $ROPE = (\mathcal{K}, \mathcal{E}nc, \mathcal{D}ec)$ *is defined as follows:*

- *Algorithm* $\mathcal{K}$ *returns a random key,* $K \xleftarrow{\$} Keys$.
- $\mathcal{E}nc$ *takes* K *and* x *as input and returns* $y = \mathcal{E}nc(K, x)$ $s.t.$ $x_i < x_j \iff \mathcal{E}nc(K, x_i) < \mathcal{E}nc(K, x_j)$.
- $\mathcal{D}ec$ *takes* K *and* y *as input and returns* $x = \mathcal{D}ec(K, y)$.

6.1.1 Query Rewrite

ROPE is a realization of OPE as described in Chapter 3. Like any order-preserving encryption scheme, ROPE enables to process range and rank queries on encrypted

databases. For example given Table 3.1, the client submits a plaintext query to retrieve all the names which are lexicographically greater than "Messi" from the `customer` table:

```
SELECT name FROM customer WHERE name > 'Messi'
```

If the name column is encrypted using an order preserving encryption scheme (OPE), the query can be rewritten such that it retrieves all the ciphertexts whose encryption is bigger than the encryption of "Messi":

```
SELECT name_enc FROM customer_enc WHERE name_enc > '2564'
```

In general the following SQL operators are supported by ROPE, and the plaintext queries are rewritten according to the following rules:

- Equality Predicate:

$$\mathcal{QRF}_{\text{ROPE}}(\sigma_{column=x}(table)) = \sigma_{column_{enc}=\mathcal{E}nc_K^{\text{ROPE}}(x)}(table_{enc}) \tag{6.1}$$

- Inequality Predicate:

$$\mathcal{QRF}_{\text{ROPE}}(\sigma_{column!=x}(table)) = \sigma_{column_{enc}!=\mathcal{E}nc_K^{\text{ROPE}}(x)}(table_{enc}) \tag{6.2}$$

- IN Predicate:

$$\mathcal{QRF}_{\text{ROPE}}(\sigma_{column\ IN\ (w,x)}(table)) = \sigma_{column_{enc}\ IN\ (\mathcal{E}nc_K^{\text{ROPE}}(w),\mathcal{E}nc_K^{\text{ROPE}}(x))}(table_{enc}) \tag{6.3}$$

- Range Predicate:

$$\mathcal{QRF}_{\text{ROPE}}(\sigma_{column>x}(table)) = \sigma_{column_{enc}>\mathcal{E}nc_K^{\text{ROPE}}(x)}(table_{enc}) \tag{6.4}$$

- Like Predicate:

$$\mathcal{QRF}_{\text{ROPE}}(\sigma_{column\ LIKE\ 'M\%'}(table)) = \sigma_{column_{enc}\geq\mathcal{E}nc_K^{\text{ROPE}}('M')\ AND\ column_{enc}<\mathcal{E}nc_K^{\text{ROPE}}('N')} \tag{6.5}$$

- EQUI-JOIN:

$$\mathcal{QRF}_{\text{ROPE}}(table1\ \bowtie_{table1.column=table2.column}\ table2) = (table1_{enc}\ \bowtie_{table1_{enc}.column_{enc}=table2_{enc}.column_{enc}}\ table2_{enc}) \tag{6.6}$$

- NON EQUI-JOIN:

$$\mathcal{QRF}_{\text{ROPE}}(table1 \ \bowtie_{table1.column>table2.column} \ table2) = \\ (table1_{enc} \ \bowtie_{table1_{enc}.column_{enc}>table2_{enc}.column_{enc}} \ table2_{enc}) \tag{6.7}$$

- ORDER BY:

$$\mathcal{QRF}_{\text{ROPE}}(\mathcal{O}_{column}(table)) = \mathcal{O}_{column_{enc}}(table_{enc}) \tag{6.8}$$

- MIN/MAX:

$$\mathcal{QRF}_{\text{ROPE}}(\Gamma_{MIN(column)}(table)) = \Gamma_{MIN(column_{enc})}(table_{enc}) \tag{6.9}$$

6.1.2 Security Analysis

In this section we analyze the security of ROPE against the attacker scenarios introduced in Chapter 4.

Security against Domain Attack

As defined in Section 4.2, a domain attack is an adversary model where the attacker has knowledge about the plaintext domain. In our security analysis we consider that in a domain attack the attacker has precise knowledge of the domain to compute an upperbound probability distribution.

In case of an order-preserving encryption scheme such as ROPE, an attacker who has knowledge about the domain and has compromised the encrypted database, can simply sort the domain and the ciphertexts in the encrypted database and figure out the mapping between those two. For example, in Table 6.1, the adversary knows that the plaintext values correspond to football players. Thus, he can guess that the first ciphertext values corresponds to a plaintext value at the beginning of the domain which is "Benzema". Therefore, we conclude that a strictly order-preserving scheme such as ROPE allows the adversary $\mathcal{A}$ in Experiment 1 to efficiently break the encryption using sorting. Thus we will have:

Lemma 5. *ROW-advantage of $\mathcal{A}$ on ROPE is defined as his winning probability in Experiment 1:*

$$Adv^{ROW}_{ROPE}(\mathcal{A}) = Pr[Exp^{ROW}_{ROPE}(A) = 1] = 1 \tag{6.10}$$

Despite a lot of effort in finding an ideal object to improve the security of OPE schemes as in [14, 15, 41, 48, 71], Lemma 5 applies to all OPE schemes in case of a domain attack.

Definition 7. *A random adversary, $\mathcal{A}^R$, in case of a domain attack, is an adversary that cannot do better than to randomly select a value from the domain. The ROW-advantage of a random adversary is therefore:*

$$Adv^{ROW}_{Prob\text{-}OPE}(\mathcal{A}^R) = Pr[Exp^{ROW}_{Prob\text{-}OPE}(\mathcal{A}^R) = 1] = \frac{1}{|\mathcal{X}|} \tag{6.11}$$

That is, each ciphertext corresponds to each plaintext with the same probability. This case is ideal because an adversary cannot do better than a random guess. In such an ideal encryption scheme, the attacker has no advantage of knowing the domain.

Security against Frequency Attack

As defined in Section 4.3, a frequency attack is an adversary model where the attacker has knowledge about the domain and its underlying frequency distribution. In our security analysis we consider that in a frequency attack the attacker has precise knowledge of the domain and its underlying frequency distribution to compute an upperbound probability distribution.

Another prominent weakness of not only ROPE or current OPE schemes but in general all deterministic encryption schemes is their inability to hide the original frequency distribution of the plaintext domain, most specifically if the plaintext domain has skewed frequency distribution. Deterministic encryption schemes allow an adversary, $\mathcal{A}$, who plays the frequency one-wayness experiment 2 to simply find a mapping between the plaintext and the ciphertext space just by looking at their frequency distributions and thereby discovering the plaintext-ciphertext correspondence.

Since ROPE is a deterministic encryption scheme the FOW-advantage of $\mathcal{A}$ on MOPE depends on the plaintext frequency distribution, namely $F_{\mathcal{X}}$.

Lemma 6. *Let $G = \{x | Freq_{\mathcal{X}}(x) = Freq_{\mathcal{Y}}(y_x)\}$ be the set of distinct plaintext values having the same frequency as y_x in Experiment 2. Then, the FOW-advantage of the $\mathcal{A}^{DF}$ adversary on MOPE is defined as his winning probability in Experiment 2:*

$$\begin{aligned} Adv^{FOW}_{ROPE}(\mathcal{A}) &= Pr[Exp^{FOW}_{ROPE}(A) = 1] \\ &= \frac{1}{|G|} \end{aligned} \tag{6.12}$$

Proof. In order to win Experiment 2 on ROPE, the adversary needs to choose a plaintext value having the same frequency as the given ciphertext in the experiment. The cardinality of set G determines the number of possibilities of $\mathcal{A}$ on ROPE. Hence, the game is won with a probability of $\frac{1}{|G|}$. □

For instance, assuming that the adversary knows the frequency distribution in Table 6.1, and has to reverse a ciphertext based on its frequency, the adversary's advantage

to figure out whose ciphertext has a cardinality of one is $\frac{Pr[Exp^{\text{FOW}}_{\text{ROPE}}(A)=1]}{3}$ because there are three customers that have a frequency of one, i.e. $|G| = 3$. On the other hand, the adversary's advantage in reversing a ciphertext of cardinality two is $Pr[Exp^{\text{FOW}}_{\text{ROPE}}(A)] = 1$ because there is only one customer that has placed two orders and this is going to be "Messi".

Deterministic encryption schemes can be safe against frequency attacks if the underlying plaintext domain has uniform frequency distribution. For example, if all the customers in Table 6.1 have placed one order then guessing which customer has placed one order based on such a uniform frequency distribution will be a random process, i.e. the adversary needs to choose randomly an element from the domain and the knowledge about the underlying frequency will be useless to him.

Corollary 5. *A deterministic encryption scheme is optimally safe against a Frequency Attack if and only if $F_{\mathcal{X}}$ is uniform.*

Security against Query Log Attack

Query log attack has been defined in Section 4.4. In this attack the adversary needs to win either the domain attack or the frequency attack with the help of the query logs. In Chapter 5, a framework has been introduced to identify query logs that weaken their underlying encryption scheme. In this section, in order to analyze the security of ROPE against query log attack two questions have to be taken care of:

1. Is ROPE indistinguishable under Simple Query Attack? This property assures that rewritten ROPE queries, as shown in Section 6.1.1, are not leaky, i.e., they do not weaken the ROPE-encrypted data.

2. How safe is ROPE against domain and frequency attack if the query logs are revealed? This property evaluates the resilience of ROPE against the query log attack introduced in Section 4.4.

In Chapter 5, three query log attacks have been introduced, namely Simple Query Attack, Known-Query Attack and Chosen-Query Attack. However, for the honest-but-curious adversary model that has motivated this dissertation, only Simple Query Attack is relevant. This is because we have assumed that the adversary has only compromised the database server, and does not collude with the client in any ways. Therefore, for each encryption scheme ,introduced or revisited in this thesis, we only analyze the security against Simple Query Attacks.

In order to analyze the security of ROPE under Simple Query Attack, we take the query rewrite mechanism in Section 6.1.1, the query processing overhead, as a reference. For every SQL operator that ROPE supports we analyze whether it is simulatable or not using Definition 6.

Query Operators without Literal Parameters. These operators include JOIN, ORDER BY, TOP N, DISTINCT, GROUP BY and MIN/MAX. These operators are rewritten such that their query logs fulfill the criteria of simulatable queries:

$$\mathcal{QG}en^{-1}(\mathcal{QRF}(q_x)) \subseteq \mathcal{E}nc^{ES}(\mathcal{QG}en^{-1}(q_x))$$

Query Operators requiring Literal Parameters. These operators include equality, range and LIKE predicates. Since ROPE is deterministic every plaintext literal is rewritten to its corresponding ciphertext literal, again fulfilling the simulatable query definition.

Thus, we conclude that ROPE is safe against Simple Query Attacks, even when the queries are rewritten to maximize performance, as shown in Section 6.1.1. However, ROPE is a weak encryption scheme that leaks total order and the frequency distribution of its underlying plaintext. Therefore, the adversary can win the query log experiment, introduced in Section 4.4, since he has access to both the underlying domain and frequency.

Corollary 6. *The advantage of an adversary playing Experiment 3 on ROPE, is the maximum probability of an adversary winning either the domain or the frequency attack on ROPE. Since ROPE is not safe against domain attack. The maximum probability of an adversary to win Experiment 3 is 1.*

$$Adv^{QLA}_{ROPE}(\mathcal{A}) = Pr[Exp^{QLA}_{ROPE}(\mathcal{A}) = 1] = 1 \tag{6.13}$$

6.2 Modular Order Preserving Encryption

Modular Order Preserving Encryption has been proposed in [15] to improve the security of OPE. This scheme models the domain as a "ring" and randomly selects a beginning (i.e., *a new minimum*) from the whole domain. This characteristic hides the total order, still allowing range queries to be processed on encrypted data. Table 6.1 gives an example of a set of customer names encrypted with *MOPE*.

Construction 5. *Let $\mathcal{OPE} = (\mathcal{K}, \mathcal{E}nc, \mathcal{D}ec)$ be an order-preserving encryption scheme. A modular order-preserving encryption scheme $\mathcal{MOPE} = (\mathcal{K}_m, \mathcal{E}nc_m, \mathcal{D}ec_m)$ is defined as follows:*

- *$\mathcal{K}_m$ runs $\mathcal{K}$, picks $j \xleftarrow{\$} \mathcal{X}$ and returns (K, j).*
- *$\mathcal{E}nc_m$ takes (K, j) and x as input and returns $\mathcal{E}nc(K, x - j \bmod X)$.*
- *$\mathcal{D}ec_m$ takes (K, j) and y as input and returns $\mathcal{D}ec(K, y) + j \bmod X$.*

6.2.1 Query Rewrite

MOPE is a modular order preserving encryption. Similar to order-preserving encryption scheme, MOPE enables to process range and rank queries on encrypted databases. The modular property of MOPE makes the query rewrite special, i.e. the range queries have to be rewritten as *wrap-around queries*. For example given Table 6.1, the client submits a plaintext query to retrieve all the names which are lexicographically greater than "Messi" from the `customer` table:

```
SELECT name FROM customer WHERE name > 'Messi'
```

If the name column is encrypted using an order preserving encryption scheme (OPE), the query can be rewritten such that it retrieves all the ciphertexts whose encryption is greater than the encryption of "Messi":

```
SELECT name_enc FROM customer_enc WHERE name_enc > '4567'
AND name_enc <= '2601'
```

Since MOPE is modular there is no total order, however it is possible to rewrite rank queries on MOPE if the modular offset is revealed in the query. For instance, the client submits a query that asks for the first customer in the database in alphabetical order:

```
SELECT MIN(name) FROM customer
```

there are many ways to rewrite this query so that it can be processed on an MOPE-encrypted *Name* column in the cloud without decrypting the *Name* column; each with different performance and security properties. The most natural approach is to rewrite this query in the following way:

```
SELECT MIN(name) FROM customer WHERE name > 3000
```

This rewritten query can be processed entirely in the cloud (without any decryption). The result is the (encrypted) tuple with ciphertext 3178 which is returned from the cloud to the client. The client decrypts this tuple and receives the correct result: "Benzema". This approach has the best possible performance: The query can be processed in the cloud on the encrypted database in (almost) the same way as the original query on a plaintext database. Furthermore, only a single tuple is shipped from the cloud to the client, thereby incurring no additional communication cost. Unfortunately, however, this rewritten query leaks the *offset* used by MOPE in this example. As a result, the attacker knows that the cipher 3178 corresponds to the alphabetically first customer and, as a result, MOPE degrades to OPE which is generally perceived to be a very weak encryption scheme. More detailed security analysis will be given in the remainder of this section.

In general the following SQL operators are supported by MOPE, and the plaintext queries are rewritten according to the following rules:

- Equality Predicate:

$$\mathcal{QRF}_{\text{MOPE}}(\sigma_{column=x}(table)) = \sigma_{column_{enc}=\mathcal{E}nc_K^{\text{MOPE}}(x)}(table_{enc}) \tag{6.14}$$

- Inequality Predicate:

$$\mathcal{QRF}_{\text{MOPE}}(\sigma_{name!=x}(table)) = \sigma_{column_{enc}!=\mathcal{E}nc_K^{\text{MOPE}}(x)}(table_{enc}) \tag{6.15}$$

- IN Predicate:

$$\mathcal{QRF}_{\text{MOPE}}(\sigma_{column\ IN\ (w,x)}(table)) = \sigma_{column_{enc}\ IN\ (\mathcal{E}nc_K^{\text{MOPE}}(w),\mathcal{E}nc_K^{\text{MOPE}}(x))}(table_{enc}) \tag{6.16}$$

- Range Predicate:

$$\mathcal{QRF}_{\text{MOPE}}(\sigma_{column>x}(table)) = \sigma_{column_{enc}>\mathcal{E}nc_K^{\text{MOPE}}(x+offset)\wedge column_{enc}<\mathcal{E}nc_K^{\text{MOPE}}(offset)}(table_{enc}) \tag{6.17}$$

- Like Predicate:

$$\mathcal{QRF}_{\text{MOPE}}(\sigma_{column\ LIKE\ 'M\%'}(table)) = \sigma_{name_{ct}\geq\mathcal{E}nc_K^{\text{MOPE}}('M'+offset)\wedge column_{enc}<\mathcal{E}nc_K^{\text{MOPE}}('N'+offset)}(table_{enc}) \tag{6.18}$$

- EQUI-JOIN:

$$\mathcal{QRF}_{\text{MOPE}}(table1\ \bowtie_{table1.column=table2.column}\ table2) = (table1_{enc}\ \bowtie_{table1_{enc}.column_{enc}=table2_{enc}.column_{enc}}\ table2_{enc}) \tag{6.19}$$

- ORDER BY:

$$\mathcal{QRF}_{\text{MOPE}}(\mathcal{O}_{column}(table)) = \mathcal{O}_{column_{enc}}(\sigma_{column_{enc}>\mathcal{E}nc(offset)}(table_{enc})) \cup \mathcal{O}_{column_{enc}}(\sigma_{column_{enc}<=\mathcal{E}nc(offset)}(table_{enc})) \tag{6.20}$$

- MIN/MAX:

$$\mathcal{QRF}_{\text{MOPE}}(\Gamma_{MIN(column)}(table)) = \Gamma_{MIN(column_{enc})}(\sigma_{column_{enc}>\mathcal{E}nc(offset)}(table_{enc})) \tag{6.21}$$

6.2.2 Security Analysis

In this section we analyze the security of MOPE against the attacker scenarios introduced in Chapter 4.

Security against Domain Attack

As defined in Section 4.2, a domain attack is an adversary model where the attacker has knowledge about the plaintext domain. In our security analysis we consider that in a domain attack the attacker has precise knowledge of the domain to compute an upperbound probability distribution.

In case of a modular order-preserving encryption scheme such as MOPE, an attacker who has knowledge about the domain and has compromised the encrypted database, cannot draw any conclusion about the ciphertexts in the encrypted database because the total order is hidden. For example, in Table 6.1, the adversary knows that the plaintext values correspond to football players. However, guessing that the first ciphertext which is 1890 corresponds to "Benzema" which is the beginning of the domain would mean failing in Experiment 1.

Lemma 7. *ROW-advantage of $\mathcal{A}$ on MOPE is defined as his winning probability in Experiment 1:*

$$Adv^{ROW}_{MOPE}(\mathcal{A}) = Pr[Exp^{ROW}_{MOPE}(A) = 1] = \frac{1}{X} \tag{6.22}$$

Proof. In order to win Experiment 1 on MOPE, the adversary needs to know the modular offset. Since offset is chosen randomly from $\mathcal{X}$, the adversary will win the game with a probability of $\frac{1}{X}$. □

Nevertheless, MOPE is vulnerable to the Known Plaintext Attacks, where the adversary additionally has one or more plaintext-ciphertext pairs. MOPE will change into ROPE, even if the adversary has only one plaintext-ciphertext pair which is not very comforting. For example in Table 6.1 once the adversary knows that "Ronaldo" corresponds to 2601 in the encrypted database. He can unwrap the ring, and find out the total order, i.e. the ciphertext following 2601 will correspond to the beginning of the domain and so on and so forth.

Security against Frequency Attack

As defined in Section 4.3, a frequency attack is an adversary model where the attacker has knowledge about the domain and its underlying frequency distribution. In our security analysis we consider that in a frequency attack the attacker has precise knowledge of the domain and its underlying frequency distribution to compute an upperbound probability distribution.

Similarly to ROPE, MOPE is a deterministic encryption scheme thus it is unable to hide the frequency distribution of the plaintext domain, most specifically if the plaintext domain has skewed frequency distribution. Deterministic encryption schemes allow an adversary, $\mathcal{A}$, who plays the frequency one-wayness experiment 2 to simply find a mapping between the plaintext space and the ciphertext space just by looking at their frequency distributions and thereby discovering the plaintext-ciphertext correspondence.

Since MOPE is a deterministic encryption scheme the FOW-advantage of $\mathcal{A}$ on MOPE depends on the plaintext frequency distribution, namely $F_{\mathcal{X}}$.

Lemma 8. *Let $G = \{x | Freq_{\mathcal{X}}(x) = Freq_{\mathcal{Y}}(y_x)\}$ be the set of distinct plaintext values having the same frequency as y_x in Experiment 2. Then, the FOW-advantage of the $\mathcal{A}^{DF}$ adversary on MOPE is defined as his winning probability in Experiment 2:*

$$\begin{aligned} Adv^{FOW}_{MOPE}(\mathcal{A}) &= Pr[Exp^{FOW}_{MOPE}(A) = 1] \\ &= \frac{1}{|G|} \end{aligned} \tag{6.23}$$

The proof of Lemma 8 follows the same reasoning as for Lemma 6 which can be found in Subsection 6.1.2.

As also stated in Subsection 6.1.2, a deterministic encryption like MOPE can be safe against frequency attacks if the underlying plaintext domain has uniform frequency distribution.

Security against Query Log Attack

Query log attack has been defined in Section 4.4. In this attack the adversary needs to win either the domain attack or the frequency attack with the help of the query logs. In Chapter 5, a framework has been introduced to identify query logs that weaken their underlying encryption scheme. In this section, in order to analyze the security of MOPE against query log attack two questions need to be taken care of:

1. Is MOPE indistinguishable under Simple Query Attack? This property assures that rewritten MOPE queries, as shown in Section 6.2.1, are not leaky, i.e., they do not weaken the MOPE-encrypted data.

2. How safe is MOPE against domain and frequency attack if the query logs are revealed? This property evaluates the resilience of MOPE against the query log attack introduced in Section 4.4.

Having a circular oder, MOPE employs a different rewrite mechanism to support efficient query processing, as shown in Section 6.2.1. In order to analyze the performance-optimized query rewrite method in Section 6.2.1, we divide the supported SQL operators in three groups:

Equality-based Operators. These operators include equality, non-equality and IN predicates, JOIN, DISTINCT and GROUP BY. Since MOPE is a deterministic scheme, a plaintext literal in a query is rewritten to its corresponding ciphertext. This fulfills the simulatability requirement.

Range and LIKE Predicates. MOPE has been introduced by Boldyreva et al. [15] as a stronger alternative of ROPE that can handle range queries. To rewrite a range query on MOPE, the query rewrite function has to write a wrap-around range query. Here is an example using Table 3.1:

```
SELECT name FROM customer
WHERE name > 'Neymar'
```

The wrap-around range query on MOPE-encrypted column will be:

```
SELECT name_enc FROM customer_enc
WHERE name_enc > 1890 AND name_enc < 3178
```

However, a wrap-around range query rewrite method is no more simulatable, since $\{1890, 3178\} \nsubseteq \{1890\}$. More formally, the query simulatability test fails:

$$\mathcal{QG}en^{-1}(\mathcal{QRF}(q_x)) \nsubseteq \mathcal{E}nc^{ES}(\mathcal{QG}en^{-1}(q_x))$$

Rank queries. If MOPE wants to support TOP N, ORDER BY and MIN/MAX queries efficiently, it has to reveal the beginning of the circle. We use Example-1 from the introduction here: Assume the client submits the following plaintext query to retrieve lexicographically the smallest name in the customer table.

```
SELECT MIN(name) FROM customer
```

The most efficient way to rewrite the above query is as follows:

```
SELECT MIN(name_enc) FROM customer_enc
WHERE name_enc > 2601
```

This rewritten query incurs minimum communication and post-processing costs. However, since $\{2601\} \nsubseteq \emptyset$, it fails the simulatability test.

Corollary 7. *The adversary exploits the additional information from the leaky queries to discover the beginning of the domain. Consequently, by discovering the beginning of the domain, MOPE is degraded to ROPE. This degradation will happen, in case the query rewrite method presented in Section 6.2.1 has been used.*

Corollary 8. *The advantage of an adversary to win Experiment 3 on MOPE, is the maximum probability of an adversary winning either the domain or the frequency attack on MOPE. Since MOPE is not safe against Simple Query Attack and degrades to ROPE as a result. The maximum probability of an adversary to win Experiment 3 on MOPE is 1.*

$$Adv^{QLA}_{MOPE}(\mathcal{A}) = Pr[Exp^{QLA}_{MOPE}(\mathcal{A}) = 1] = 1 \tag{6.24}$$

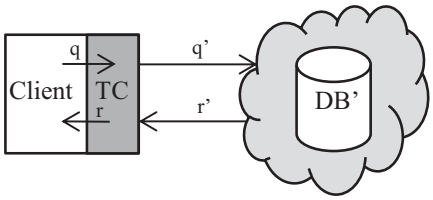


Figure 6.1: Client-Side Security

6.3 Experiments

This section assesses the performance overhead of ROPE in the context of the 22 queries and 2 refresh functions of the TPC-H benchmark. We present results for Plain (not encrypted database), ROPE [14] which was revisited in this chapter and is the state of the art for range and rank queries as used in systems like CryptDB [47] and Monomi [66], and AES ECB (resilient against domain attack). From now on in the experiments we refer to ROPE as OPE for simplicity.

6.3.1 Benchmark Environment

All experiments were conducted on two separate machines for client and server which corresponds to the architecture shown in Figure 6.1. The client and the trusted component were written in Java, ran on a machine with 24 GB of memory and communicated to the database server using JDBC. The server machine had 132 GB of memory available and hosted a MySQL 5.6 database. Both machines had 8 cores and ran a Debian-based Linux distribution.

We used a 10 GB data set (scaling factor 10) and measured end-to-end response time for all queries in separation. Queries that did not finish within 30 minutes where canceled and reported as a time-out. This is why we do not show results for Q9, which even times out for *Plain*. Metrics used in aggregate functions (e.g., volume of orders) and surrogates (e.g., order numbers) were left unencrypted while all other (sensitive) attributes, such as names, dates, etc. were encrypted. Whenever SQL operators on encrypted data were not supported, the entire data was shipped and then post-processed at the trusted component.

6.3.2 Response Time

Figure 6.2 shows the relative response time of OPE compared to AES ECB (a deterministic implementation of AES) and relative to the plain database. Figure 6.3, on the other hand, shows the average response time values including the standard deviation of running each experiment 1000 times. As illustrated we consider a time-out to be for

a query that takes more than 600 seconds. Additionally, Figure 6.3 shows the break down of the time a query spends on the server and on the client. The following concrete observations are to be made from these figures:

- **Range Queries (Q1,Q3,Q4,Q5,Q10,Q12,Q14,q15,Q20,Q21).** Many queries in TPC-H have range predicates which reduce the amount of data retrieved from the disk to a huge extend. On the contrary, AES is not able to perform range predicates on encrypted data, thereby it needs to read full tables from the disk. As illustrated in Figure 6.3, these queries time-out already on the server side, before even reaching the post-processing phase on the client.

- **Equality Predicates (Q2,Q16).** When there are only equality predicates involved in a query, AES is much faster than OPE. As shown in Figure 6.3 both Q2 and Q16 spend almost the same amount of time at the server. The shipped data has to be decrypted at the client side where AES outperforms OPE because of its faster decryption operation.

- **Highly Selective Equality and Range Predicates (Q7,Q8,Q13,Q18,Q22).** According to the first observation although range queries are a killer for a deterministic but non-order preserving scheme, it seems that a few TPC-H queries are exempt from this rule. This is because those queries have highly selective equality predicates that already filter a lot of false-positives from the final result that needs to be shipped. On the client-side even though AES has to handle more data, its fast decryption process compensates for the higher amount of data it has to process compared to OPE that only needs to decrypt and has no false positives.

- **Updates.** Not only AES has faster decryption algorithms, but also faster encryption algorithms. This is why updates are faster for AES compared to OPE.

6.3.3 Network Costs

Figure 7.8 shows the relative network cost of OPE and AES ECB (A deterministic implementation of AES) to the unencrypted plain database. Please note that Q9 cause a time-out even for the plain database, that is why we are not interested in its performance for our comparisons.

Since AES ECB only supports equality predicates and joins, we can see that for most of the queries, the network cost is very high. This indicates that in case of AES ECB, a lot of data must be shipped to the trusted component and post-processed after decryption whereas OPE does a lot of filtering already in the cloud and therefore reduces the network cost drastically.

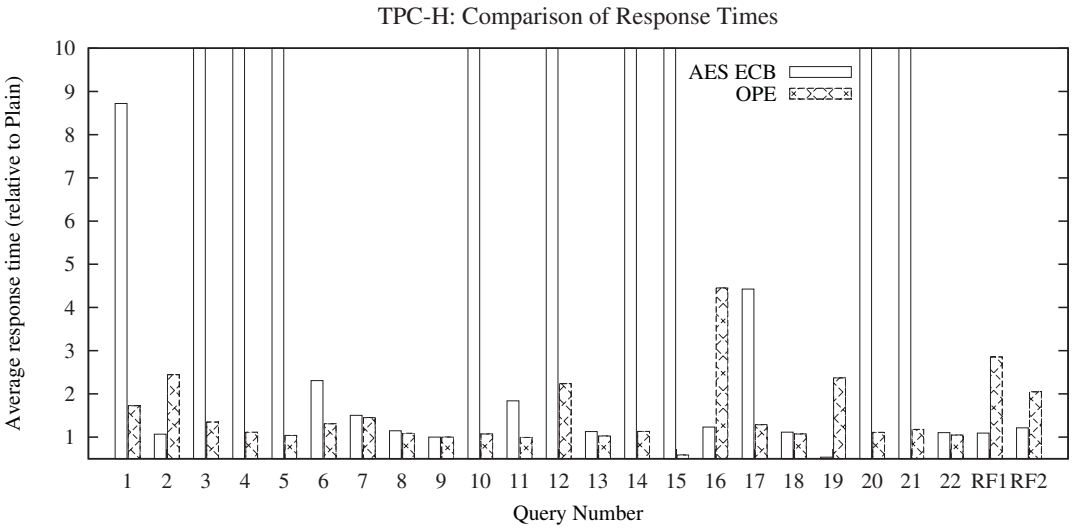


Figure 6.2: OPE Response Time: Relative to Plain

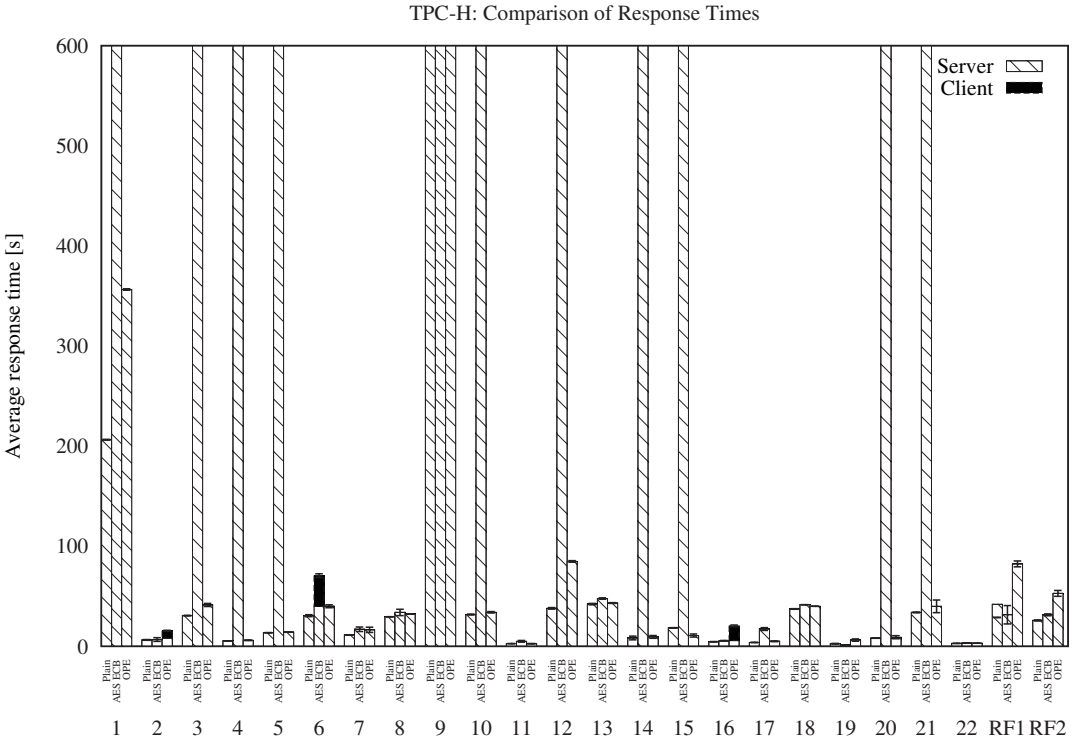


Figure 6.3: OPE Response Time: Client-Server Breakdown

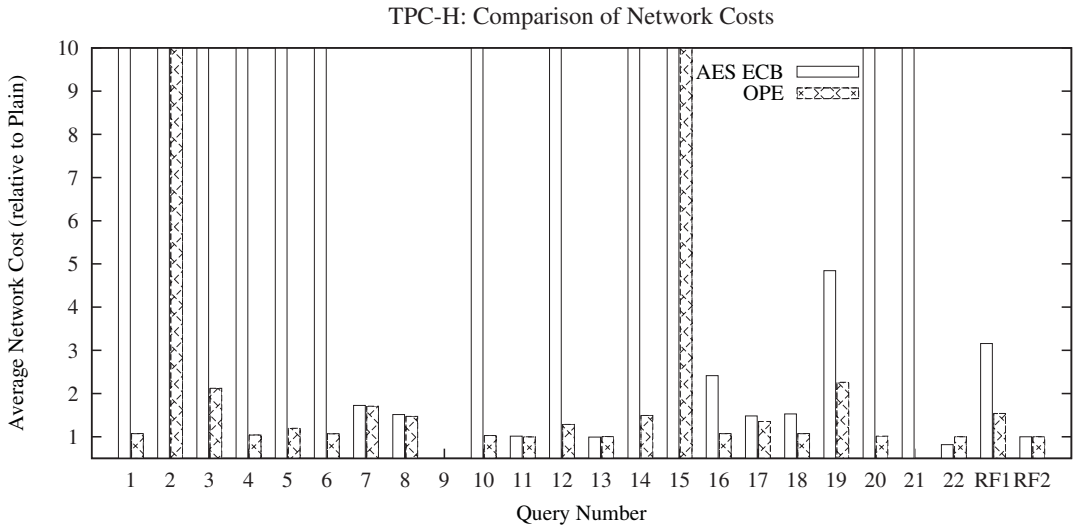


Figure 6.4: OPE Network Cost: Relative to Plain

6.4 Related Work

OPE was first proposed in the database community by Agrawal et al. [6] to support efficient range and rank queries on encrypted data. OPE encryption schemes preserve the order of the plaintext after encryption. Therefore, comparisons can be directly executed on ciphertext values without decrypting them. With advances in cloud computing OPE has again gained a lot of attention. However, its weaknesses against adversaries with domain and frequency knowledge is considered to be a barrier on its way to be deployed in the real world database. The first formal cryptographic treatment of OPE did not appear until recently in [14, 15].

Index Tagging Schemes. In [15], an Index Tagging Scheme is proposed for the static and committed databases called *Committed Efficiently-Orderable Encryption*. In CEOE a combination of traditional encryption and an index tagging scheme is used. This scheme uses a key and the domain as input, and constructs a monotone minimal perfect hash function that maps the i-th largest plaintext of the domain to the tag value i. A different tagging scheme is *Mutable Order Preserving Encryption (mOPE)* [48]. This scheme is based on a mutable search tree storing the index information. While the scheme supports dynamic databases and variable domains, security relevant concerns - like the domain being unknown to the attacker - remain.

Indistinguishability based OPE. In [41], a new indistinguishability-based security notion for OPE, which can ensure *secrecy of lower bits of plaintext* under essentially a random ciphertext probing setting is proposed. The authors then propose a new scheme

satisfying this security notion while the earlier schemes do not satisfy it. Note that the known security notions tell us nothing about the above partial-plaintext indistinguishability because they are limited to being one-way-based.

General OPE. In [71], the authors propose a general approach called Generalized Order Preserving Encryption (GOPE). Unlike OPE, the ciphertexts of GOPE may not be numbers, but GOPE still enables the comparisons on the encrypted data without the need to decrypt them. They present two GOPEs in polynomial-sized and superpolynomial-sized domains that satisfy stronger notions of security than that of the ideal OPE object. In [69], the authors investigate possible alternative for the ideal OPE object.

These schemes focus on cryptographic analysis of OPE schemes, still ignoring the real world usecases, such as adversaries with domain, frequency and query log knowledge.

6.5 Conclusion

Table 6.2 summarizes the SQL-operators that are supported by the state of the art encryption schemes and the encryption schemes revisited in this chapter, namely ROPE and MOPE. From this point on we use OPE instead of ROPE for the sake of simplicity. HES is an abbreviation for Homomorphic Encryption Schemes, and AES-CBC is a probabilistic and semantically secure mode of AES as opposed to AES-ECB which is deterministic and not semantically secure. As shown in Table 6.2, semantically secure encryption schemes (e.g. AES CBC) do not support any query processing and thus not attractive for database applications. On the other end, weak encryption schemes such as OPE support a large number of SQL operators but shatter against domain and frequency attacks since they are deterministic and leak the total order as shown in Table 6.5.

SQL Operator	AES-ECB	OPE	MOPE	Paillier [44]	AES-CBC
DISTINCT	✓	✓	✓	✗	✗
WHERE (=, !=)	✓	✓	✓	✗	✗
WHERE (<, >)	✗	✓	✓	✗	✗
LIKE(Prefix%)	✗	✓	✓	✗	✗
LIKE(%Suffix)	✗	✗	✗	✗	✗
IN	✓	✓	✓	✗	✗
Equi-Join	✓	✓	✓	✗	✗
Non Equi-Join	✗	✓	✓	✗	✗
TOP N	✗	✓	✓	✗	✗
ORDER BY	✗	✓	✓	✗	✗
SUM	✗	✗	✗	✓	✗
MIN/MAX	✗	✓	✓	✗	✗
GROUP BY	✓	✓	✓	✗	✗

Table 6.2: OPE: Supported SQL Operators

Table 6.5 summarizes the result of the security analysis done in this chapter on OPE and MOPE. Each row represents an OPE variant, and each column presents an

OPE variant	Domain Attack	Frequency Attack (uniform)	Frequency Attack (skewed)	Query Log Attack	Known-Plaintext Attack
OPE	Plain	OPE	Plain	Plain	OPE
MOPE	MOPE	MOPE	Plain	Plain	OPE

Table 6.3: OPE: Security Downgrade

attacker model introduced in Chapter 4. Each cell shows how low an encryption scheme downgrades under a given attack. By plain we mean an unencrypted database.

Since MOPE turns into OPE in case of a query log attack which is an inevitable threat in a database domain, we will no more consider it for further performance and security analysis. Instead we propose more robust encryption schemes with stronger security guarantees.

7

Probabilistic OPE (Prob-OPE)

Order-preserving encryption schemes such as ROPE and MOPE that were analyzed in the previous chapter are deterministic. Deterministic encryption schemes have the weakness of revealing the frequency distribution of their underlying plaintext values. In this chapter, we propose a *probabilistic order preserving encryption* scheme that can be composed with either ROPE or MOPE from the previous chapter or any state of the art order preserving encryption scheme. In Probabilistic OPE (Prob-OPE), as shown in Table 7.1, the idea is that the same plaintext is encrypted differently each time it goes through the encryption algorithm while the order is still preserved, allowing range and rank queries to be processed on encrypted data efficiently. A probabilistic order-preserving encryption scheme is a probabilistic symmetric encryption scheme whose algorithm not only produces ciphertexts that preserve numerical ordering of the plaintexts, but also generates different ciphertexts for the same plaintext. This property flattens out the original frequency distribution of the plaintext values, therefore resisting any statistical analysis. However, probabilistic schemes still leak total order and are exposed to domain attacks as it becomes clear in the remainder of this chapter.

There are different methods to realize a probabilistic order preserving encryption as in [24, 35, 68, 72]. In this thesis we introduce a new method that can be easily composed with any state of the art order preserving encryption scheme, and imposes no change to the encryption function. In our method, as shown in Figure 7.1, we introduce an additional step before using an order-preserving encryption. In this step, the plaintext, x gets expanded by concatenating k random bits to its least significant bit side, thereby preserving the order. In order to guarantee the uniformity of the frequency distributions in the ciphertext space, k needs to be lower-bounded by the logarithm of the frequency

Customer Name	ROPE	Prob-ROPE
Benzema	2	2
Benzema	2	3
Benzema	2	5
Messi	3	8
Neymar	5	10
Neymar	5	11
Ronaldo	8	14
Xavi	10	17

Table 7.1: ROPE vs. Prob-ROPE

of the most frequent element in $\mathcal{X}$ i.e. $k > \log_2(Freq_{max}(x))$. In Table 7.1 it is shown how a skewed frequency distribution in the customer name column is flattened out in the Prob-OPE variant.

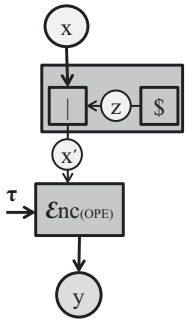


Figure 7.1: Probabilistic OPE

Notation. Let $\mathcal{X}$ be the set of plaintext values from a finite domain, and $\mathcal{Y}$ be the set of ciphertext values. The size of $\mathcal{X}$ is denoted as $X = |\mathcal{X}|$; the same applies for the size of $\mathcal{Y}$, $Y = |\mathcal{Y}|$. Plaintext elements are denoted as x and ciphertext elements as y. Additionally, we denote the Key set to be $\mathcal{K}eys$ and $K \xleftarrow{\$} \mathcal{K}eys$ denotes that a key, K, is selected uniformly at random from $\mathcal{K}eys$. The $\$$ sign on top of the $\leftarrow$ depicts that the selection was uniformly at random. $\mathcal{K}$ is a randomized algorithm that creates a random key from a finite set. Let $\mathcal{E}nc$ be the encryption function having a key, K, and a plaintext value, x, as its input parameters; thus, we have: $y = \mathcal{E}nc(K, x)$. Symmetrically, $\mathcal{D}ec$ will be the decryption function, taking y and K as input, yielding: $x = \mathcal{D}ec(K, y)$. Let $\mathcal{Z}$ be the set of binary strings of length k, $\mathcal{Z} = \{0,1\}^k$ and z be an element of $\mathcal{Z}$. After concatenating z to x, the expanded plaintext is denoted as $x' = x \| z$ where $x' \in \mathcal{X} \times \mathcal{Z}$.

Construction 6. *Let $\mathcal{OPE} = (\mathcal{K}, \mathcal{E}nc, \mathcal{D}ec)$ be a deterministic order-preserving encryption scheme from the previous section. We define the probabilistic order preserving scheme, $\mathcal{P}rob - \mathcal{OPE}(\mathcal{K}_p, \mathcal{E}nc_p, \mathcal{D}ec_p)$ as follows:*

- *$\mathcal{K}_P$ runs $\mathcal{K}$ and returns K.*
- *$\mathcal{E}nc_p$ takes K and $x' = x\|z$ where $z \xleftarrow{\$} \mathcal{Z}$ as input and returns $\mathcal{E}nc(K, x')$.*
- *$\mathcal{D}ec_p$ takes K and y as input, runs $x' = \mathcal{D}ec(K, y) = x\|z$ and returns x.*

7.1 Creating Prob-OPE Databases

Prob-OPE can be composed with any state of the art order-preserving encryption technique within a database. The only assumption made is that, if applied, Prob-OPE 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 encryption function. If a key of a table is encrypted using Prob-OPE, for instance, then all foreign keys to that table must be encrypted using Prob-OPE 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 Prob-OPE. Other *info* fields which could potentially identify a customer and are subject to range predicates such as *age* should also be encrypted using Prob-OPE. 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 (non order-preserving) encryption technique. Metrics such as *price* or *volume* which are aggregated as part of `GROUP BY` queries, should not be encrypted at all until a practical homomorphic encryption technique with a secure key size has been found. In our experience, it is typically obvious which domains to encrypt and for which domains Prob-OPE is advantageous. The meta-data that records which attributes are encrypted in which way must be maintained by the trusted component.

If a Prob-OPE 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 7.2. “Messi” is encrypted using two ciphertexts. Correspondingly, two tuples must be stored for “Messi” in the *Customer* table. *Orders* can refer to either of these tuples. In Figure 7.2, for instance, Orders 2 and 3 refer to the first Messi Customer tuple (Ciphertext 7) and Order 4 refers to the second Messi Customer tuple (Ciphertext 10).

Name Encryption

Value	Code
Benzema	1
Messi	7
Messi	10

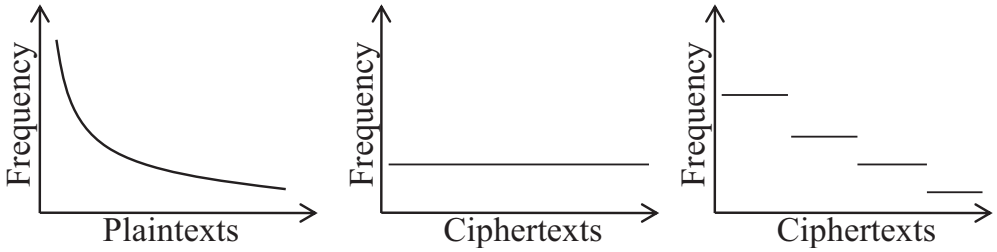
Table Customer

name	info
1	...
7	...
10	...

Table Orders

id	cust
1	1
2	7
3	7
4	10

Figure 7.2: Prob-OPE: Keys and Foreign Keys



(a) Value Skew (b) Uniform Codes (c) Piecewise Unif.

Figure 7.3: Distorting a Frequency Distribution

7.2 Frequency Distortion

In order to protect a database against a frequency attack, a probabilistic scheme must be used. With such a scheme, the frequency distribution of the values can be *distorted* to blend the outstanding ones in the crowd and thus protect them [17, 25]. This idea is illustrated in Figure 7.3. Figure 7.3a shows the frequency distribution of *values* in the original (unencrypted) database. Specifically, Figure 7.3a shows for each customer how many orders that customer has placed. Figure 7.3b shows a *secure* frequency distribution of *codes* in an encrypted databases. In Figure 7.3, 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 7.3a.

The frequency distribution of Figure 7.3b is achieved by assigning a different number of ciphertexts to each customer. Using Construction 6, a uniform frequency distribution can be achieved with a high probability if the number of random bits concatenated to the least significant bit of the plaintext is bigger than the logarithm of the plaintext values with the maximum frequency, i.e. $k > \log_2(Freq_{max}(x))$. This ensures minimum collision of the generated ciphertexts guaranteeing a uniform frequency distribution independent of the minimum frequency. In our example, more ciphertexts are assigned to "Messi" than to, say, "Benzema" because Messi has placed more orders. Accordingly,

there will be more "Messi" tuples in the *Customer* table as observed in the example shown in Figure 7.2. Unfortunately, it can take a large number of ciphertexts in order to achieve such perfect uniform distributions of ciphertexts as shown in Figure 7.3b. Effectively, the *greatest common denominator*, g, of all frequencies must be taken and for each plaintext value f_x/g ciphertexts must be generated if f_x is the frequency of value x in the original frequency distribution. In many examples, $g = 1$ 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 performance loss.

The effects of a more practical approach are depicted in Figure 7.3c. 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 7.3c); the number of steps are the number of iterations and the height of each step is the base frequency selected in each iteration. Using Construction 6 a step-wise frequency distribution can be achieved when ciphertexts generated for a plaintext value would collide. This can be achieved if the number of bits concatenated to the original plaintext is smaller than the maximum frequency, i.e. $k < \log_2(Freq_{max}(x))$. The smaller k gets, the more collision, and on the flip side the ciphertext frequency distribution will resemble the original frequency distribution even more, i.e. less security against a frequency attack. Nevertheless, k is a parameter that can be adjusted according to the privacy and performance requirements.

Fortunately, all these techniques can naturally be combined with OPE in order to achieve efficient processing of complex queries even in frequency attack scenarios. All that needs to be done is to organize the (probabilistic) ciphertexts as shown in Table 7.2.

Figure 7.3 shows how the frequency distribution of a single attribute can be distorted. The same technique can be applied in order to distort correlations. In this case, the multi-attribute distribution is flattened. For example, a number of codes could be allocated for customers named Peter in New York. The principle is the same, but breaking correlations can result in the generation of a larger number of codes for each value, thereby impacting performance.

7.3 Query Rewrite

The query rewrite for Prob-OPE is slightly different than OPE that has been shown in the previous chapter. Although, the range queries are rewritten the same way, the equality predicates have to be rewritten as ranges. For example, given Table 7.1, a user

submits a query:

```
SELECT name FROM customer WHERE name = 'Benzema'
```

For a deterministic order preserving scheme such as ROPE the query is rewritten as:

```
SELECT name_enc FROM customer_enc WHERE name_enc = 2
```

However, for Prob-OPE the equality predicate should be replaced by a range predicate:

```
SELECT name_enc FROM customer_enc WHERE 1 < name_enc < 8
```

Although ORDER BY can be kept the same way, the rest of the rank queries have to be rewritten in such a way that duplicates are taken into account. For example, given Table 7.1, a user submits a query:

```
SELECT TOP 3 name FROM customer ORDER BY name
```

In such a query client expects to get back "Benzema", "Messi" and "Neymar" as the result. In fact if a deterministic OPE is used, such as the one proposed in the previous chapter, the above query only needs a few schema adjustments as follows:

```
SELECT TOP 3 name_enc FROM customer_enc ORDER BY name_enc
```

However, in a probabilistic OPE as proposed in this chapter, using the above method to rewrite the above query will end up returning "Benzema" three times. That is why in the rank queries the max frequency needs to be known to make sure that the result contains no true negatives. One way to rewrite the above query is therefore as follows:

```
SELECT TOP 9 name_enc FROM customer_enc ORDER BY name_enc
```

The key idea is that we need to multiply the N in the `TOP N` by the max frequency. This assures us that we have the real TOP 3 in our results and only need to filter out the duplicates. In a database with millions of rows retrieving `TOP 10` instead of `TOP 3` has a negligible effect on performance, assuming that the frequency of any given domain element does not exceed three.

7.3.1 Formal Rewrite Rules

In general the following SQL operators are supported by Prob-OPE, and the plaintext queries are rewritten according to the following rules:

- Equality Predicate:

$$\mathcal{QRF}_{\text{Prob-OPE}}(\sigma_{column=x}(table)) = \tag{7.1}$$
$$\sigma_{column_{enc} \geq\ \mathcal{E}nc_K^{\text{Prob-OPE}}(x|z_l) \wedge column_{enc} \leq\ \mathcal{E}nc_K^{\text{Prob-OPE}}(x|z_u))}(table_{enc})$$

- Inequality Predicate:

$$\mathcal{QRF}_{\text{Prob-OPE}}(\sigma_{column!=x}(table)) = \sigma_{column_{enc}<\mathcal{E}nc_K^{\text{Prob-OPE}}(x|z_l)\wedge column_{enc}>\mathcal{E}nc_K^{\text{Prob-OPE}}(x|z_u)}(table_{enc}) \tag{7.2}$$

- IN Predicate:

$$\mathcal{QRF}_{\text{Prob-OPE}}(\sigma_{column\ IN\ (w,x)}(table)) = \sigma_{column_{enc}\geq \mathcal{E}nc_K^{\text{Prob-OPE}}(x|z_l)\wedge column_{enc}\leq \mathcal{E}nc_K^{\text{Prob-OPE}}(x|z_u)}(table_{enc}) \cup \sigma_{column_{enc}\geq \mathcal{E}nc_K^{\text{Prob-OPE}}(w|z_l)\wedge column_{enc}\leq \mathcal{E}nc_K^{\text{Prob-OPE}}(w|z_u)}(table_{enc}) \tag{7.3}$$

- Range Predicate:

$$\mathcal{QRF}_{\text{Prob-OPE}}(\sigma_{column>x}(table)) = \sigma_{column_{enc}>\mathcal{E}nc_K^{\text{Prob-OPE}}(x|z_u)}(table_{enc}) \tag{7.4}$$

- Like Predicate:

$$\mathcal{QRF}_{\text{Prob-OPE}}(\sigma_{column\ LIKE\ 'M\%'}(table)) = \sigma_{column_{enc}\geq \mathcal{E}nc_K^{\text{Prob-OPE}}('M'|z_l)\ AND\ column_{enc}<\mathcal{E}nc_K^{\text{Prob-OPE}}('N'|z_l)} \tag{7.5}$$

- EQUI-JOIN:

$$\mathcal{QRF}_{\text{Prob-OPE}}(table1 \bowtie_{table1.column=table2.column} table2) = (table1_{enc} \bowtie_{table1_{enc}.column_{enc}=table2_{enc}.column_{enc}} table2_{enc}) \tag{7.6}$$

- NON EQUI-JOIN:

$$\mathcal{QRF}_{\text{Prob-OPE}}(table1 \bowtie_{table1.column>table2.column} table2) = (table1_{enc} \bowtie_{table1_{enc}.column_{enc}>table2_{enc}.column_{enc}} table2_{enc}) \tag{7.7}$$

- ORDER BY:

$$\mathcal{QRF}_{\text{Prob-OPE}}(\mathcal{O}_{column}(table)) = \mathcal{O}_{column_{enc}}(table_{enc}) \tag{7.8}$$

- MIN/MAX:

$$\mathcal{QRF}_{\text{Prob-OPE}}(\Gamma_{MIN(column)}(table)) = \Gamma_{MIN(column_{enc})}(table_{enc}) \tag{7.9}$$

7.4 Security Analysis

In this section we will analyze the security of Prob-OPE against the database attacker models presented in Chapter 4.

7.4.1 Security Analysis of the Composed Construction

We have used ROPE from [14] to build Prob-OPE. [15] has introduced a security notion on one-wayness properties of ROPE. In this section we find a mapping between Prob-OPE and ROPE to show the relationship between these two encryption schemes in terms of the already introduced security notions in [15].

The one-wayness (OW) property in general 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 ROPE been thoroughly analyzed in [15] as follows: Given z encryptions $y_1, , , y_z$ of randomly chosen x_i from the domain, the adversary $\mathcal{A}$ succeeds if he finds any inverse from $y_1, , , y_z$ [15].

Proposition 1. *Fix a challenge set of size z. Let $\mathcal{A}$ be an OW adversary that outputs a plaintext from $\mathcal{X}$. Then for any adversary $\mathcal{A}$,*

$$Adv^{OW}_{ROPE}(\mathcal{A}) = \kappa \tag{7.10}$$

The question is, what is the one-wayness advantage of Prob-OPE applying ROPE as its underlying encryption scheme. Thus, we need to find a reduction from the One-Wayness adversary on Prob-RPE to the One-Wayness adversary on ROPE as follows:

Lemma 9. *If the adversary on Prob-OPE wins the One-Wayness experiment with probability ϵ, then the probability that he inverts a ciphertext is less than or equal the advantage of the adversary playing the one-wayness experiment on the underlying encryption scheme which is ROPE in our case.*

$Adv^{OW}(\mathcal{A}^R) = \frac{1}{X} \geq \ Adv^{OW}_{Prob\text{-}OPE[X]}(\mathcal{A}) = Adv^{OW}_{ROPE[X]}(A)$

Proof. Given an adversary against Prob-OPE with a one-wayness advantage of ϵ, we want to construct an adversary against its underlying encryption scheme. As defined earlier, the one-wayness experiment for the underlying encryption scheme offers the adversary z ciphertexts from a single run and the adversary can invert at least one of them. In order to relate this adversary to the Prob-OPE adversary, we need to find a reduction. In this case the reduction is trivial, the challenge set is inverted using the Prob-OPE adversary with an advantage of ϵ and what remains to be done is to deterministically remove the additional random bits from the least significant bit of the result returned by Prob-OPE adversary. Therefore we can say that succeeding the one-wayness experiment by an adversary against Prob-OPE is upperbounded by the advantage of an adversary against ROPE and lowerbounded by a random adversary, $\mathcal{A}^R$ that randomly returns a value from the domain of size X. □

7.4.2 Security against Domain Attack

As defined in Section 4.2, a domain attack is an adversary model where the attacker has knowledge about the plaintext domain. In our security analysis we consider that in a domain attack the attacker has precise knowledge of the domain to compute an upperbound probability distribution.

In case of an order-preserving encryption scheme such as ROPE, revisited in Section 6.1, an attacker who has knowledge about the domain and has compromised the encrypted database, can simply sort the domain and the ciphertexts in the encrypted database and figure out the mapping between those two. However, once an OPE scheme becomes probabilistic using Construction 6, then breaking such a scheme by a domain attack is no more trivial.

Lemma 10. *ROW-advantage of $\mathcal{A}$ on Prob-OPE is defined as his winning probability in Experiment 1:*

$$Adv^{ROW}_{Prob\text{-}OPE}(\mathcal{A}) = Pr[Exp^{ROW}_{Prob\text{-}OPE}(\mathcal{A}) = 1] = \frac{\binom{rank_y-1}{rank_x-1}\binom{Y-rank_y}{X-rank_x}}{\binom{Y-1}{X-1}} \tag{7.11}$$

Proof. Here we give a description on the derivation of the formula in equation 7.11. The denominator holds all possible encoding combinations in a Prob-OPE scheme after taking out the item in the experiment from the set. According to combinatorics theory, the number of ways you can assign a domain of size X to the ciphertext domain of size Y after excluding the item in the experiment is:

$$\binom{Y-1}{X-1}$$

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 is the number of combinations we can have for the values above our selected element times the number of combinations for the values below our selected element:

$$\binom{rank_y-1}{rank_x-1}\binom{Y-rank_y}{X-rank_x}$$

□

An important observation to make from Equations 7.11 and 7.11 is that the ROW advantage not only depends on the domain size but also on the rank of the plaintext in the domain. Consequently, extreme values of the domain (e.g. ”AAA” or ”ZZZ”) are breaking the uniformity of the probability distribution. In order to fix this problem, a modular OPE can be used to form a ring structure and hide these extreme values or dummy values can be inserted at the extremes.

Definition 8. *A random adversary, $\mathcal{A}^R$, in case of a frequency attack, is an adversary that cannot do better than to randomly select a value from the domain. The FOW-advantage of a random adversary is therefore:*

$$Adv^{FOW}_{Prob\text{-}OPE}(\mathcal{A}^R) = Pr[Exp^{FOW}_{Prob\text{-}OPE}(\mathcal{A}^R) = 1] = \frac{freq(x)}{\sum_{v \in \mathcal{X}} freq(v)} \tag{7.12}$$

Here, $freq(x)$ corresponds to the frequency of Value x in the plain database. Again, this equation is independent of DB. Note that in the frequency attack scenario, the adversary's advantage denotes the probability to get one *ciphertext right; the probability to get* all *ciphertexts of a value right is of course much smaller. Correspondingly, more frequent values have higher probability of being discovered.*

As a result the more frequent the plaintext, a given ciphertext has a higher probability to map onto it. A simple clarification example would be a bag with 2 white marbles and 8 black marbles. Of course, the probability of picking a black marble is higher. In general the following lemma captures the advantage of an adversary playing the Frequency One-Wayness Experiment.

In Figure 7.4 the ROW advantage is plotted for each encryption scheme discussed so far. As a baseline we plot the advantage of an adversary that outputs a random x regardless of the domain. This is considered to be "ideal" and is exactly what MOPE achieves. In previous chapter we have shown why MOPE breaks against domain attack in presence of a single known plaintext-ciphertext pair (Known-Plaintext Attack or KPA). We conclude that the MOPE and ROPE are both vulnerable and easily breakable against a Domain Attack. On the other hand, the ROW-advantage of Prob-OPE is getting closer to the ideal, however still far away. Specially there is no security provided for the values at extremes. This can be fixed to some extent by inserting dummy values at the extremes, however this is not an elegant solution.

As shown in equation 6.22, the advantage of a ROW-adversary on MOPE is the same as a random adversary which makes the ROW-advantage probability distribution ideal. Hereby, we can say that MOPE is optimally Rank One-Way in the sense that an adversary cannot do better than a random adversary that outputs a random plaintext after observing its ciphertext rank, independent of the domain. In other words:

$$Adv^{\text{ROW}}_{\text{MOPE}}(\mathcal{A}) \leq Adv^{\text{ROW}}(\mathcal{A}^R)$$

7.4.3 Security against Frequency Attack

A known weakness of a deterministic encryption scheme is its inability to hide the original frequency distribution of the plaintext domain, specially if the plaintext domain has a skewed frequency distribution. An encryption scheme must be **probabilistic** to ensure privacy against frequency attacks. To see why, assume that the attacker knows that

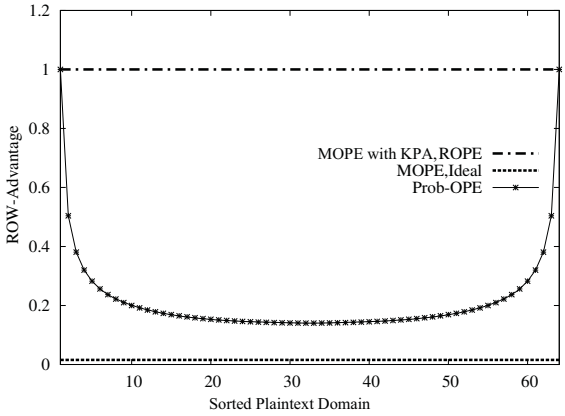


Figure 7.4: ROW advantage under Domain Attack

"Benzema" has placed three orders, "Neymar" has placed two orders, etc. If "Benzema" is represented by a single ciphertext in an encryption scheme, then the attacker can easily infer that ciphertext by counting the number of occurrences of each *cust* ciphertext 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 frequency skew (without exact values). However, to be safe in the general case, a probabilistic encryption scheme is required to protect confidential data against a frequency attack. Such a probabilistic encryption scheme allows to distort the frequency distribution.

Prob-OPE as introduced by Construction 6 has a tunable parameter to distort the plaintext frequencies. If $k > \log_2(Freq_{max}(x))$ then a skewed frequency distribution, as shown in Figure 7.3a, will be turned into a uniform frequency distribution, as shown in Figure 7.3b. A uniform frequency distribution is ideal because there is no frequency distribution available to be used.

Lemma 11. *Let $y = \mathcal{E}nc_K^{\text{Prob-OPE}}(x)$, where $\mathbf{y}$ is the set of ciphertexts corresponding to x. $freq_{max}(\mathbf{y})$ returns the frequency of the most repeated ciphertext in $\mathbf{y}$. More formally, $y_j \in \mathbf{y}$ s.t. $\forall\ y_i \in \mathbf{y},\ freq(y_j) \geq freq(y_i)$. The FOW-advantage of $\mathcal{A}$ on Prob-OPE encryption scheme is defined as his winning probability in Experiment 2:*

$$Adv^{\text{FOW}}_{\text{Prob-OPE}}(\mathcal{A}) = Pr[Exp^{\text{FOW}}_{\text{Prob-OPE}}(A) = 1] = \frac{freq(x)}{\sum_{v \in \mathcal{X}\ \wedge\ freq(v) \geq freq_{max}(y)} freq(v)} \tag{7.13}$$

Proof. In order to win Experiment 2 on Prob-OPE, the adversary needs to choose a plaintext value having the same frequency as the given ciphertext in the experiment. Prob-OPE flattens out a frequency distribution using a tunable parameter namely k, which is the number of additional bits attached to the original plaintext. Depending on k the final ciphertext frequency can be anything between uniform and a step-wise function

or logarithmic function. As already introduced in Definition 8, once the ciphertexts form a uniform frequency distribution, it is easier to guess values with higher frequencies. However, due to performance reasons, a uniform ciphertext frequency distribution is not always practical. As shown in Figure 7.3c, the frequency can be broken into pieces. As a result some ciphertexts can have higher frequencies than others. Of course if a plaintext value has lower frequency than a ciphertext it means that by no means that ciphertext can correspond to that plaintext. For example in Table 7.2, the adversary knows that "Benzema" has placed only one order, so by no means 7 can correspond to "Benzema". Thus, considering this simple fact, Equation 7.12 for a random adversary needs to be rewritten into Equation 7.13 so that only the plaintext values in the domain are considered that actually can have that many ciphertexts. Thus, it is shown that adversary's advantage increases once the ciphertext frequency distribution changes from Figure 7.3b to Figure 7.3c. □

7.4.4 Security against Query Log Attack

To win the query log attack, defined in Section 4.4, the adversary needs to either win the domain attack or the frequency attack with the help of the query logs. In Chapter 5 a framework has been introduced to identify leaky queries. In this section, in order to analyze the security of Prob-OPE against query log attack, two questions need to be taken care of:

1. Is Prob-OPE indistinguishable under Simple Query Attack? This property assures that rewritten queries, as discussed in Section 7.3, are not leaky, i.e., they do not weaken the Prob-OPE-encrypted data.

2. How safe is Prob-OPE against domain and frequency attack if the query logs are revealed? This property evaluates the resilience of Prob-OPE against the query log attack introduced in Section 4.4.

To break the underlying frequency distribution, Prob-OPE assigns a new ciphertext to the same plaintext. Similar to OPE, we divide the *supported* query operators into two classes:

Query Operators without Literal Parameters. These operators include JOIN, ORDER BY, TOP N and MIN/MAX. Such operators are rewritten such that $\mathcal{QG}en^{-1}(\mathcal{QRF}(q_x)) = \emptyset$. This property fulfills the simulatable query criteria:

$$\mathcal{QG}en^{-1}(\mathcal{QRF}(q_x)) \subseteq \mathcal{E}nc^{ES}(\mathcal{QG}en^{-1}(q_x))$$

Query Operators requiring Literal Parameters. These operators include equality, range and LIKE predicates. Since Prob-OPE is probabilistic, every plaintext literal is mapped to a range, obviously missing the simulatability criteria. As an example consider the query of Example-2 in the introduction using Table 3.1:

```
SELECT name FROM customer
WHERE name = 'Messi'
```

To process the above query efficiently, it will be rewritten into:

```
SELECT name_enc FROM customer_enc
WHERE name_enc <= 2568 AND name_enc >= 2796
```

This rewritten query incurs minimum communication and post-processing costs. However, since $\{2568, 2796\} \nsubseteq \{2568\}$, it fails the simulatability test.

Corollary 9. *The adversary exploits the additional information from the leaky queries to discover the correspondence of ciphertexts to a plaintext. Consequently, this possibility to bin ciphertexts degrades Prob-OPE to ROPE. This degradation will of course happen, if the query rewrite method presented in Section 7.3 is used.*

Corollary 10. *The advantage of an adversary to win Experiment 3 on Prob-OPE, is the maximum probability of an adversary winning either the domain or the frequency attack on Prob-OPE. Since Prob-OPE is not safe against Simple Query Attack and degrades to ROPE as a result. The maximum probability of an adversary to win Experiment 3 on Prob-OPE is 1.*

$$Adv^{QLA}_{Prob\text{-}OPE}(\mathcal{A}) = Pr[Exp^{QLA}_{Prob\text{-}OPE}(\mathcal{A}) = 1] = 1 \tag{7.14}$$

Simulatable Queries

Table 7.2 summarizes which SQL operators are simulatable under a Simple Query Attack. NA means that the SQL operator is not supported in that encryption scheme. As shown in Section 6.2, MOPE is not safe against Simple Query Attacks. Table 7.2 shows that MOPE is concretely not safe against the range queries because the query rewrite function needs to wrap-around ranges by revealing the modular offset. Also MOPE is not safe against rank queries because again the query rewrite function has to reveal the modular offset in order to perform rank queries.

In this chapter, we have introduced Prob-OPE, which is a probabilistic order preserving encryption scheme. This encryption scheme is not safe against a Simple Query Attack, if the queries contain values from the plaintext domain. Each value in the plaintext domain is translated into a range in the Prob-OPE scheme. This fact will be revealed in all the point and range queries, making them non-simulatable as shown in Table 7.2. As already shown in Section 6.1 we have already shown that OPE is safe against simple query attacks, thus all the SQL-operators are simulatable on OPE.

SQL-Operator	OPE	MOPE	Prob-OPE
DISTINCT	✓	✓	NA
WHERE (=, !=)	✓	✓	✗
WHERE ($<, >$)	✓	✗	✗
LIKE(Prefix%)	✓	✗	✗
LIKE(%Suffix)	NA	NA	NA
IN	✓	✓	✗
Equi-Join	✓	✓	✓
Non Equi-Join	✓	✗	✓
TOP N	✓	✗	✗
ORDER BY	✓	✗	✓
SUM	NA	NA	NA
MIN/MAX	✓	✗	✓
GROUP BY	✓	✓	NA

Table 7.2: Prob-OPE: SQL-Operator Simulatability

7.5 Fixed Range Query Rewrite

There are many ways to rewrite a query. Usually a query is rewritten to get the best response time, i.e. most of the processing has been pushed down to the cloud. In the previous section, we have introduced an attack that is based on the information that an adversary can extract from the query logs. We have shown that a Prob-OPE is not IND-SQA because it leaks information about the valid ranges, and consequently about the ciphertexts that map to the same plaintext. In subsection 7.4.4, we have shown that the leakage is caused due to range and point queries. In this section we introduce a new rewrite called *Fixed Range Query Rewrite* that fixes the problem with the leaky ranges for Prob-OPE.

The idea of Fixed Range Query Rewrite is to divide the plaintext domain, $\mathcal{X}$ into j disjoint fixed-sized sub-ranges of size r, fr_i where $i = 1, ..., j$ and $|fr_i| = r$. Whenever a range or point query is asked, the smallest units that are returned are those *fixed ranges* that contain the user's results.

As an example, consider Table 7.3 where Prob-OPE has been used for encryption. The client submits the query:

```
SELECT name FROM customer WHERE name = 'Neymar'
```

Using Prob-OPE the query should be rewritten as:

```
SELECT name_enc FROM customer_enc
WHERE name_enc > 8 AND name_enc < 15
```

Fixed Range	Name (Plain)	Name (Prob-OPE)
fr_1	Benzema	2
	Benzema	3
	Messi	8
fr_2	Neymar	10
	Neymar	13
	Ronaldo	15

Table 7.3: Two fixed ranges of size 3 (i.e., $r = 3$)

This query can be fully executed in the cloud such that the result only needs to be decrypted in the trusted component. However, since this query is revealing some information about the borders of plaintext values, we will use Fixed Range Query Rewrite mechanisms to fuzzify the result. Thus, the original query will be rewritten as follows to cover the whole fixed range, namely fr_2 that contains the result:

```
SELECT name_enc FROM customer_enc WHERE name_enc > 8
```

The result includes additional false positives, namely, "Ronaldo" that will be filtered out during the post-processing. Obviously, using *Fixed Ranged Query Rewrite* returns a super-set of the result that needs to be post-filtered and therefore leads to performance loss.

7.5.1 Security against Query Log Attack

In Section 7.4.4 we have shown that Prob-OPE is safe if the rewritten queries do not contain any literal parameter. However, Prob-OPE downgrades to OPE if the queries contain literals, e.g., equality or range predicates. There are two ways to fix this:

1. Use absorbing query rewrite function
2. Use fixed range query rewrite function

While the first approach guarantees security, it undermines performance. Essentially, the query will incur maximum shipping cost and post-processing effort which is not interesting for database applications. Here we show why using the second approach, namely fixed range query rewrite is a better compromise between query log security and performance.

The idea of fixed range query rewrite is to partition the domain into fixed ranges. The ranges can be stored in the database. In case of Prob-OPE if an equality or range query is submitted, the plaintext literal will be mapped to a range which the ciphertext of that literal belongs to. Consider the query of Example-2 in the introduction using Table 3.1:

```
SELECT name FROM customer
WHERE name = 'Messi'
```

To process the above query efficiently, it will be rewritten into:

```
SELECT name_enc FROM customer_enc
WHERE name_enc <= 2568 AND name_enc >= 2796
```

This rewritten query incurs minimum communication and post-processing costs, however it fails the simulatability test, as shown in the previous section. On the other hand, if an absorbing query rewrite function will turn the above query into:

```
SELECT name_enc FROM customer_enc
```

An absorbing query rewrite maximizes query log security but undermines performance entirely. Instead by using a fixed range query rewrite function, this query changes into:

```
SELECT name_enc FROM customer_enc
WHERE name_fr = 1
```

This rewritten query incurs far less communication and post-processing costs, than using an absorbing query rewrite and it is also passes the simulatability. A similar approach to fixed range query rewrite was taken in [29] where they bucketize data. However, the fixed range query rewrite does not change the encryption scheme, but it just rewrites queries differently. This method is only applied on equality and range predicate in case of Prob-OPE. Other SQL operators such as ORDER BY, TOP N, and JOIN actually benefit from the individual data encryption in Prob-OPE versus data bucketization appraoch proposed in [29].

7.6 Experiments

This section assesses the performance overhead of Prob-OPE in the context of the 22 queries and 2 refresh functions of the TPC-H benchmark. We present results for Plain (not encrypted database), OPE (state of the art for range and rank queries), Prob-OPE (resilient against frequency attack), Prob-OPE with Fixed Range Query Rewrite (resilient against frequency and query log attack).

The fixed ranges for the date data types are selected to be a quarter, which is three months (90 days). For varchar the fixed ranges are bucketize to alphabetically, i.e. all words beginning with "A" are in fr_1, are words beginning with "B" are in fr_2 and so on. For the numeric values the fixed ranges are selected so that each fixed range accommodates between 10 or 20 percent of the values of the domain.

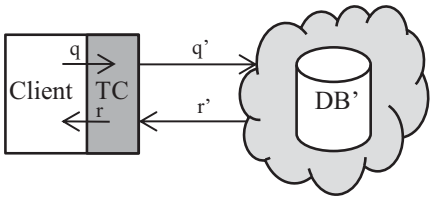


Figure 7.5: Client-Side Security

7.6.1 Benchmark Environment

All experiments were conducted on two separate machines for client and server which corresponds to the architecture shown in Figure 7.5. The client and the trusted component were written in Java, ran on a machine with 24 GB of memory and communicated to the database server using JDBC. The server machine had 132 GB of memory available and hosted a MySQL 5.6 database. Both machines had 8 cores and ran a Debian-based Linux distribution.

We used a 10 GB data set (scaling factor 10) and measured end-to-end response time for all queries in separation. Queries that did not finish within 30 minutes where canceled and reported as a time-out. This is why we do not show results for Q9, which even times out for *Plain*. Metrics used in aggregate functions (e.g., volume of orders) and surrogates (e.g., order numbers) were left unencrypted while all other (sensitive) attributes, such as names, dates, etc. were encrypted. Whenever SQL operators on encrypted data were not supported, the entire data was shipped and then post-processed at the trusted component.

7.6.2 Response Time

Figure 7.6 shows the relative response time of Prob-OPE and Prob-OPE-FR compared to OPE and relative to the plain database. Figure 7.7, on the other hand, shows the average response time values including the standard deviation of running each experiment 1000 times. As illustrated we consider a time-out to be for a query that takes more than 600 seconds. Additionally, Figure 7.7 shows the break down of the time a query spends on the server and on the client. The following concrete observations are to be made from these figures:

- **Fixed Range Query Rewrite.** According to Figure 7.6, in a few cases such as Q1, Fixed Range Query Rewrite has a better response time. The reason is that Fixed Range Query Rewrite mechanism divides the ranges into independent partitions that can be queried in separately thereby exploiting intra-request parallelism on the database server. In most of the cases Prob-OPE-FR performs the same as Prob-OPE because the ranges used in the query spans exactly over one or more

fixed ranges, omitting the shipment of false positives. In Q14 however the Fixed Range Query Rewrite method times out. This is because unlike Prob-OPE, Prob-OPE-FR has to do costly OPE decryption on the client side to be able to compute the aggregate value.

- **Updates.** RF1 and RF2 represent the two update functions of the TPC-H benchmark. RF1 inserts a value in the `order` and `lineitem` table. Prob-OPE and Prob-OPE-FR have identical behavior so no comparison is required. However, compared to OPE, Prob-OPE performs two to three times worse. This is because of the integrity constraints. In Section 7.1 we have discussed how a new entry in the child table, might cause creating and inserting a clone in the parent table to assure that JOIN will still work. This overhead both for insertion and deletion is caused by this phenomena that is inevitable when a probabilistic encryption scheme is used.

- **Heavy JOIN (Q5,Q13).** In Q5 we see four times additional overhead because of its heavy join predicate. A join in a probabilistic scheme has to ship a lot of data to the client. As it is also shown in Figure 7.7 the additional overhead is on the server side for Q5.

- **Heavy Aggregation (Q10,Q16).** Although Prob-OPE is not able to perform aggregates on the server side, its response time for most the queries is in the same order of OPE. However, for Q10 it seems no more to be the case. Q10 performs aggregation on a large group of attributes. Using a probabilistic scheme means that all these attributes need to be shipped to the client, decrypted (which is very costly for OPE), post-grouped and aggregated. This is why in Figure 7.7 we see that the time-out is caused during the post-processing part.

7.6.3 Network Costs

Figure 7.8 shows the relative network cost of OPE, Prob-OPE and Prob-OPE-FR (Prob-OPE with Fixed Range Query Rewrite method) to the unencrypted plain database. Please note that Q9 causes a time-out even for the plain database, that is why we are not interested in its performance for our comparisons.

The very important and foremost observation is that the probabilistic variants of OPE incur far higher shipment costs than OPE. This is because for every aggregation all the data needs to be shipped to the trusted component and the duplicates need to be removed after decryption. Luckily as shown in Subsection 7.6.2 the response time is not heavily affected by the amount of data that needs to be shipped in most of the queries.

Another important observation is that the Fixed Range Query Rewrite method, as expected, incurs much higher network cost since it is shipping fixed ranges with false

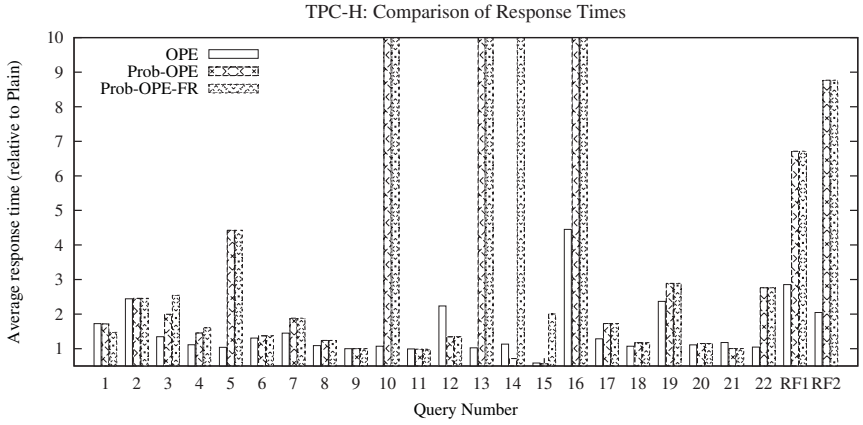


Figure 7.6: Prob-OPE Response Time: Relative to Plain

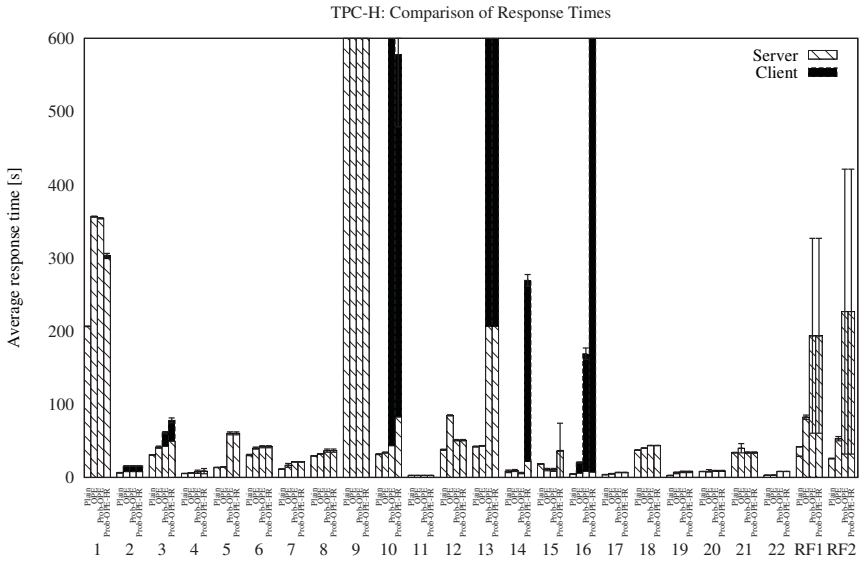


Figure 7.7: Prob-OPE Response Time: Client-Server Breakdown

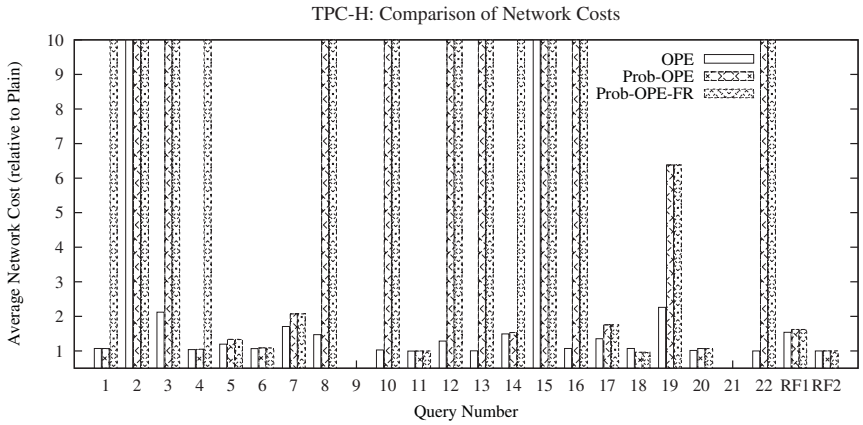


Figure 7.8: Prob-OPE Network Cost: Relative to Plain

positives to the client. Nevertheless, probabilistic OPE variants incur far less network cost compared to semantically secure encryption schemes such as AES CBC.

7.7 Related Work

Probabilistic OPE. Statistical attacks were one of the earliest attacks used by cryptanalysts to break an encryption scheme. In regard to order preserving encryption, there were a few attempt done to distort the original frequency distribution. In [72], the authors propose an encryption algorithm that appends a random string to the plaintext before encryption, so multiple occurrences of a plaintext lead to different ciphertexts. Although their method protects the encrypted database against statistical attacks, it does not prevent an attacker with the domain knowledge to infer information, since the total order is still leaking.

Order Preserving Encryption with Splitting and Scaling (OPESS) is proposed by [68]. OPESS is used to index the encrypted values in the outsourced XML databases. The idea is to map the same plaintext values to differenct ciphertext values to protect the data against statistical attacks. OPESS consists of two stages, splitting and scaling. In splitting, each plaintext value is encrypted into one or more ciphertext values by using different keys. The number of keys used to encrypt a plaintext value is based on the number of occurrences of a plaintext value. In scaling, the number of occurrences of encrypted values is multiplied by a scaling factor.

Structure Preserving Database Encryption Scheme. Another probabilistic encryption scheme is proposed by [24]. The idea is to break the correlation between ciphertext and plaintext values by encrypting each value with its unique cell coordinates. However this scheme can be used only on a trusted server where a database admin-

istrator can manage the new index structure in the encrypted database. Also the join operation is no more possible, and the only way to perform join is to decrypt the tables. Moreover, a database reorganization process changes cell coordinates and therefore all affected cells need to be re-encrypted with their new coordinates.

Multivalued-Order Preserving Encryption (MV-OPES) is proposed by [35]. The idea is to encrypt a value to different multiple values to prevent statistical attacks and at the same time preserve order. When encrypting plaintext values in a column, MV-OPES generates boundaries for all integers in the domain using the increasing/decreasing function. An initial value is randomly chosen and encrypted using a traditional encrypting schemes (e.g. AES), then the boundaries are derived around that initial value using the decreasing or increasing function. In other words, each plaintext value is assigned to a range in the ciphertext space. The weakness of their scheme is first of all against an attacker with domain knowledge they are not better off than any other encryption scheme introduced so far. Additionally, the query rewrite process reveals the mapping between plaintext values and ranges, even enabling a statistical attack by using query logs.

Superset Retrieval. In [22, 29] encryption schemes have been proposed to promote the *Database as a Service* model by increasing the performance of encrypted databases. These encryption mechanisms supported range and rank query processing on encrypted data by bucketing the data into certain ranges. At first glance this might appear similar to our *Fixed Range Query Processing* approach presented in Section 7.5. However, in our approach the data can be encrypted also in a fine-grained manner, to support more query processing on encrypted data and therefore resulting in far better performance than what has been shown in [22, 29].

7.8 Conclusion

Table 7.4 summarizes the SQL-operators that are supported by the state of the art encryption schemes, OPE and Prob-OPE. HES is an abbreviation for Homomorphic Encryption Schemes, and AES-CBC is a probabilistic and semantically secure mode of AES as opposed to AES-ECB which is deterministic and not semantically secure. As shown in Table 7.4, Prob-OPE supports almost all the SQL operators that OPE supports. However, because of being probabilistic it cannot support GROUP BY and DISTINCT.

Table 7.8 summarizes the result of the security analysis done in this chapter on Prob-OPE. Each row represents an OPE variant, and each column presents an attacker model introduced in Chapter 4. Each cell shows how low an encryption scheme downgrades under a given attack. By plain we mean an unencrypted database. The FR postfix indicates the case when Fixed Range Query Rewrite mechanism is used to rewrite queries on the encrypted database.

Please note that each variant in Table 7.8 has a tunable probability distribution under each attacker scenario. In general, the encryption schemes introduced in this thesis have

SQL Operator	AES-ECB	OPE	Prob-OPE	Paillier [44]	AES-CBC
DISTINCT	✓	✓	✗	✗	✗
WHERE (=, !=)	✓	✓	✓	✗	✗
WHERE (<, >)	✗	✓	✓	✗	✗
LIKE(Prefix%)	✗	✓	✓	✗	✗
LIKE(%Suffix)	✗	✗	✗	✗	✗
IN	✓	✓	✓	✗	✗
Equi-Join	✓	✓	✓	✗	✗
Non Equi-Join	✗	✓	✓	✗	✗
TOP N	✗	✓	✓	✗	✗
ORDER BY	✗	✓	✓	✗	✗
SUM	✗	✗	✗	✓	✗
MIN/MAX	✗	✓	✓	✗	✗
GROUP BY	✓	✓	✗	✗	✗

Table 7.4: Prob-OPE: Supported SQL Operators

OPE variant	Domain Attack	Frequency Attack (uniform)	Frequency Attack (skewed)	Query Log Attack
OPE	Plain	OPE	Plain	Plain
Prob-OPE	Prob-OPE	Prob-OPE	Prob-OPE	Plain
Prob-OPE-FR	Prob-OPE	Prob-OPE	Prob-OPE	Prob-OPE

Table 7.5: Prob-OPE: Security Downgrade

a performance/security knob that can be adjusted according to the performance/security requirements of the system.

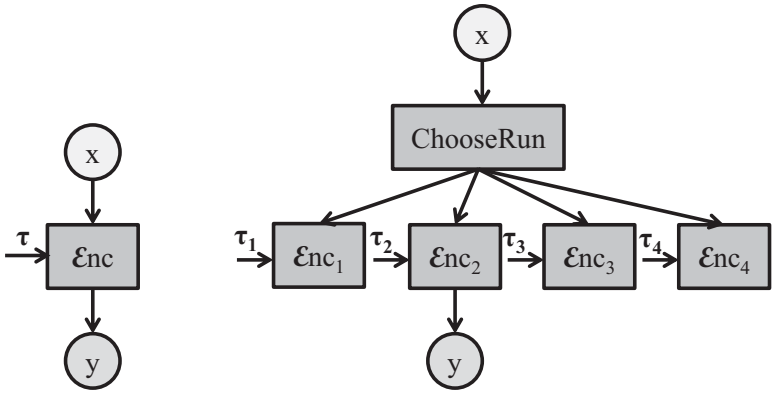
8

Randomly Partitioned Encryption (RPE)

Although querying on OPE encrypted databases is efficient, OPE is vulnerable against adversaries with knowledge on the domain and frequency distribution [6]. This fact makes OPE a weak and undesirable encryption scheme for encrypting sensitive data in a database. Going back to the usecase from the financial industry that motivated this work, the database administrator had a list of all customers and only needed to retrieve other relevant account information for selected customers. An order-preserving encryption scheme is immediately broken in this case, simply by performing a one-to-one mapping from the sorted customer list to the sorted ciphertexts.

This chapter presents the deterministic *Randomly Partitioned Encryption* scheme and explains how it is used to create efficient encrypted databases, how the queries are rewritten and how does it compare to the state of the art encryption both in terms of privacy and performance. The security of RPE has been fully defined, analyzed and proven in this chapter. We show how RPE extends the performance/privacy envelope by providing strong security guarantees as a traditional strong encryption technique and providing a similar performance as a traditional weak encryption technique.

The key idea of RPE is to take an existing weak encryption method such as OPE as in [6, 14, 15, 41, 71] as a building block and to enhance its security by applying it separately on different random partitions of the data called *Runs*. As part of this partitioning, each plaintext value of a column is randomly assigned to a *Run* and the attacker has no knowledge of which plaintext 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*. In contrast to other partially order preserving encryption schemes, such as [36, 57] where they bin the plaintext domain in random-length (or fixed ranged)



(a) OPE (b) RPE

Figure 8.1: RPE Principle

partitions, RPE creates partially ordered partitions called *Runs* by randomly assigning each domain value to a *Run*; thereby creating more uncertainty as will be discussed in the Section 8.3.

Figure 8.1 illustrates the RPE principle. Figure 8.1a shows the workings of a traditional encryption function. It receives a plaintext, x, and produces a ciphertext, y, and τ is the set of input parameters that $\mathcal{E}nc$ takes (e.g. a secret key). Figure 8.1b shows how RPE 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 $\mathcal{E}nc$ function per domain, RPE makes use of $\mathcal{U}$ encrypt functions per domain, $\mathcal{E}nc_1, \mathcal{E}nc_2, \ldots, \mathcal{E}nc_U$. These $\mathcal{U}$ functions possibly all have the same structure (e.g. order-preserving [14]) and just differ in the secret key they use. Given a plaintext, x, *ChooseRun()* randomly generates a number between 1 and U and encrypts x using the corresponding $\mathcal{E}nc$ function.

Depending on the *ChooseRun* function and the structure of the $\mathcal{E}nc$ function, the composed encryption scheme of Figure 8.1b can have different properties. The performance overhead of RPE (as compared to no encryption) depends on the number of runs, U. In the extreme case of $U = 1$, RPE is the same as $\mathcal{E}nc$ which is typically a weak, yet high performance encryption scheme. In the other extreme, $U = \infty$, RPE is the same as *random* which corresponds to a strong yet low performance encryption scheme.

The methods introduced in this chapter can be best described using the running example of a database with the following two relations.

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

Name (Plaintext)	Run 1	Run 2	RPE (run, ciphertext)
Benzema	4		(1,4)
Messi		1	(2,1)
Neymar		7	(2,7)
Ronaldo	5		(1,5)

Table 8.1: Det-RPE with Two Runs

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. As a naming convention we use the _enc prefix to mark columns and tables in the encrypted database, e.g. customer_enc is the encrypted customer table in the encrypted database.

Table 8.1 shows the basic idea of RPE on top an order-preserving scheme: Rather than creating a total order or a completely random order, RPE creates *partial orders* of the ciphertexts by randomly partitioning the domain into ordered *runs*. Within each run, all codes are order-preserving. Across runs, no order can be implied between two codes.

Table 8.1 presents an example *Deterministic RPE (DET-RPE)* with two runs. Such an encryption scheme can be used against domain attacks because it has distorted the total order of the values. As shown in the fourth column of Table 8.1, codes have two fields: (a) the run and (b) the (order-preserving) code within that run.

RPE allows to process range and rank queries with a reasonable overhead. A query that asks for all customers with `name LIKE ''M%''`, for instance, can be rewritten into the following query for Table 8.1:

```
SELECT * FROM customer_enc
WHERE (name_run = 1 AND name_enc > 4)
  OR (name_run = 2 AND name_enc > 4)
```

Obviously, the performance overhead of RPE (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 8.3 and 8.5. It will become clear that RPE allows one to achieve both, good privacy and good performance for a wide range of types of queries.

More formally, deterministic RPE (abbreviated as Det-RPE) is a deterministic encryption scheme because a deterministic *ChooseRun* and a deterministic $\mathcal{E}nc$ function are used. Determinism of the *ChooseRun* function can be achieved by making its output depend on the plaintext, x, i.e. by using a *pseudo-random* function. Our Det-RPE construction has to generate two sets of keys, one set for the runs and the other for the *ChooseRun* function. The encryption function of Det-RPE then composes the

ChooseRun function with the encryption function of a typical OPE scheme. Det-RPE can be composed with any OPE scheme from [6, 14, 15, 41, 69, 71].

Construction 7. *Let $\mathcal{OPE} = (\mathcal{K}, \mathcal{E}nc, \mathcal{D}ec)$ be a deterministic order-preserving encryption scheme. $\mathcal{D}et\text{-}\mathcal{RPE}(\mathcal{K}_{Det-RPE}, \mathcal{E}nc_{Det-RPE}, \mathcal{D}ec_{Det-RPE})$ is defined to be a determinstic randomly partitioned encryption scheme, as follows:*

- *$\mathcal{K}_{Det-RPE}$ runs $\mathcal{K}$ independently for each run to get $(K_1, ..., K_U)$. Also, it runs $\mathcal{K}$ independently to generate K_{map} for the $\mathcal{C}hooseRun$ function.*
- *$\mathcal{E}nc_{Det-RPE}$ takes K_u and x, as input where $u = \mathcal{C}hooseRun(K_{map}, x)$. Then it returns u and $y = \mathcal{E}nc(K_u, x)$, i.e. (u, y).*
- *$\mathcal{D}ec_{Det-RPE}$ takes (u, y) as input and returns $x = \mathcal{D}ec(K_u, y)$.*

8.1 Creating RPE Databases

RPE can be combined with any other encryption technique within a database. The only assumption made is that, if applied, RPE 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 encryption function. If a key of a table is encrypted using RPE, for instance, then all foreign keys to that table must be encrypted using RPE 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 RPE. Other *info* fields which could potentially identify a customer and are subject to range predicates such as *age* should also be encrypted using RPE. 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 (non order-preserving) encryption technique. As mentioned earlier, metrics such as *price* or *volume* which are aggregated as part of `GROUP BY` queries, should not be encrypted at all until a practical homomorphic encryption technique with a secure key size has been found. In our experience, it is typically obvious which domains to encrypt and for which domains RPE is advantageous. The meta-data that records which attributes are encrypted in which way must be maintained by the trusted component.

If Det-RPE scheme is used to encrypt a column, then for each column two columns need to be created in the encrypted database. As shown in Figure 8.2, one column holds the run number and one column holds the ciphertext generated by the encryption function of that run. A combined index is defined on each $(run, ciphertext)$ pair. Revealing runs enables query processing on encrypted data.

Name Encryption

Plaintext	RPE(run,ciphertext)
Messi	$\langle 1, 1 \rangle$
Neymar	$\langle 2, 2 \rangle$

Table Customer

name_run	name_enc	info
1	1	...
2	2	...

Table Orders

id	cust_run	cust_enc
1	1	1
2	2	2
3	2	2

Figure 8.2: Det-RPE Tables

8.2 Query Rewrite

As described in Chapter 2, the trusted component rewrites SQL queries so that they can be processed by the database system that hosts the encrypted data in the cloud. Furthermore, the trusted component post-processes query results returned by the cloud. This post-processing involves decrypting the RPE ciphertexts and it may involve post-filtering and post-aggregating the results.

In the introduction we have already shown how to rewrite simple equality and range predicates in a Det-RPE scheme. In general, plaintext values in the queries are replaced by their corresponding ciphertexts. Furthermore, simple predicates may result in disjunctions depending on the number of runs affected by the predicates. This section focuses on more complex queries. In particular, this section details how general comparisons, joins, `GROUP BY`, and `ORDER BY` (e.g., Top N) queries can be rewritten. Eventually, using an extended version of relational algebra, we show how query rewrite for each supported SQL operator would work.

8.2.1 General Comparisons

Let us assume that the *Order* table of our running example 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 RPE. 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_enc
WHERE shipDate_run != planDate_run
  OR shipDate_enc > planDate_enc;
```

This rewritten query works for Det-RPE 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 encrypted in different runs. Such false positives result in suboptimal

performance because these false positives are shipped from the cloud to the trusted component and must be post-filtered by it after they have been decrypted. However, if the shipDate and planDate of each order is encrypted 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 each other; e.g., two dates of the same order. Fortunately, this restriction in the encryption scheme does not hurt privacy. Later in this chapter we will study the performance of queries (including such comparison queries) on RPE encrypted databases.

8.2.2 Joins

Functional joins along foreign key/key relationships are a particularly simple case because they need not be rewritten at all. To see why, we will look at a query that joins the *Customer* and *Order* tables. This query can be processed on both an RPE encrypted and non encrypted database using the following SQL query. The exact implementation of key/foreign key constraints in an RPE encrypted database is discussed in Section 8.1:

```
SELECT * FROM customer_enc, order_enc
WHERE name_enc = cust_enc AND name_run = cust_run
```

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. 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. As a result, a large number of false positives must be shipped from the cloud to the trusted component; i.e., pairs of tuples whose join keys are encoded in different runs and for which the join condition does not hold.

8.2.3 `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 in the previous subsections. The `HAVING` clause of a `GROUP BY` query is also unchanged if a deterministic scheme is used; i.e., DET-RPE. With DET-RPE, furthermore, no post-processing of results by the trusted component is needed; the trusted component simply needs to decrypt the query results; i.e., the `GROUP BY` keys.

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 homomorphic encryption scheme once these techniques become practical (Chapter 3). If metrics are encrypted using RPE, only *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 trusted component and grouping and aggregation need to be executed in the trusted component.

8.2.4 `ORDER BY` Queries

If the `ORDER BY` clause of a query involves only one attribute and this attribute is encrypted using RPE, 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 U queries (with U the number of runs) and *merge* the results in the trusted component.

Top N queries [16] 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, $U \times N$ 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 trusted component. One downside is that the number of queries whose results must be merged grows exponentially with the number of attributes in the `ORDER BY` clause. As each database cursor incurs an overhead, it might be beneficial to ask a *single* `ORDER BY` query that retrieves all *runs* that must be merged. In this approach, the `ORDER BY` clause of the rewritten query is composed as follows:

```
ORDER BY attr1-run, attr1-code-id,
  attr2-run, attr2-code-id, ...
```

8.2.5 Updates

Once an RPE 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 encrypted using RPE. 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 8.1. 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.

8.2.6 Formal Rewrite Rules

Here we formalize the query rewrite rules for RPE using an extended version of relational algebra, that expresses aggregations as Γ and ORDER BY as $\mathcal{O}$. Let $\mathcal{QRF}$ denote the *Query Rewrite Function* that is called by the trusted component to rewrite a plaintext queries,q , submitted by the *Client* as input, into a rewritten query, q' to be processed on the encrypted database, DB', i.e. $\mathcal{QRF}(q) = q'$. In general the following SQL operators are supported by deterministic RPE, and the plaintext queries are rewritten according to the following rules:

- Equality Predicate:

$$\mathcal{QRF}_{\text{RPE}}(\sigma_{column=x}(table)) = \tag{8.1}$$
$$\sigma_{column_{enc}=\mathcal{E}nc^{\text{RPE}}_{K_{chooseRun(x)}}(x)\ \wedge\ column_{run}=chooseRun(x)}(table_{enc})$$

- Inequality Predicate:

$$\mathcal{QRF}_{\text{RPE}}(\sigma_{column!=x}(table)) = \tag{8.2}$$
$$\sigma_{column_{enc}!=\mathcal{E}nc^{\text{RPE}}_{K_{chooseRun(x)}}(x)\ \wedge\ column_{run}!=chooseRun(x)}(table_{enc})$$

- IN Predicate:

$$\mathcal{QRF}_{\text{RPE}}(\sigma_{column\ IN\ (w,x)}(table)) = \tag{8.3}$$
$$\sigma_{(column_{enc}=\mathcal{E}nc^{\text{RPE}}_{K_{chooseRun(x)}}(x)\ \wedge\ column_{run}=chooseRun(x))\ \vee\ ...}(table_{enc})$$

- Range Predicate:

$$\mathcal{QRF}_{\text{RPE}}(\sigma_{column>x}(table)) = \tag{8.4}$$
$$\sigma_{(column_{enc}>\mathcal{E}nc_{K_1^{\text{RPE}}(x)}\ \wedge\ column_{run}=1)\ \vee\ ...}(table_{enc})$$

- Like Predicate:

$$\mathcal{QRF}_{\text{RPE}}(\sigma_{column\ LIKE\ 'M\%'}(table)) = \tag{8.5}$$
$$\sigma_{(column_{enc}\geq\ \mathcal{E}nc_{K_1^{\text{RPE}}('M')}\ \wedge\ column_{run}=1)\wedge(column_{enc}<\ \mathcal{E}nc_{K_1^{\text{RPE}}('N')}\ \wedge\ column_{run}=1)\ \vee\ ...}$$

- EQUI-JOIN: For RPE the JOIN has to make sure that values being joined are in the same partition. The rewrite rule will be as follows:

$$\mathcal{QRF}_{\text{RPE}}(table1 \ \bowtie_{table1.column=table2.column} \ table2) = \tag{8.6}$$
$$table1_{enc} \ \bowtie_{t1_{enc}.column_{enc}=t2_{enc}.column_{enc} \wedge t1_{enc}.column_{run}=t2_{enc}.column_{run}} \ table2_{enc}$$

- NON EQUI-JOIN: For RPE the rewrite has to make sure to only compare values within same partitions to each other. The final interleaving will be done in the post-processing step at the client-side.

$$\mathcal{QRF}_{\text{RPE}}(table1 \ \bowtie_{table1.column>table2.column} \ table2) = \tag{8.7}$$
$$\sigma_{column_{run}=1}(table1_{enc} \ \bowtie_{table1_{enc}.column_{enc}>table2_{enc}.column_{enc}} \ table2_{enc}) \cup$$
$$\sigma_{column_{run}=2}(table1_{enc} \ \bowtie_{table1_{enc}.column_{enc}>table2_{enc}.column_{enc}} \ table2_{enc}) \ ...$$

- ORDER BY: In RPE each partition needs to be sorted separately. The result is then merged after decryption in the post-processing phase.

$$\mathcal{QRF}_{\text{RPE}}(\mathcal{O}_{column}(table)) = \tag{8.8}$$
$$\Omega_{column_{enc}}(\sigma_{column_{run}=1}(table_{enc})) \cup$$
$$\Omega_{column_{enc}}(\sigma_{column_{run}=2}(table_{enc}))$$

- MIN/MAX: In RPE since we have multiple partitions, each of them could hold the MIN/MAX element and we do not know which one has it. Thus, we return the MIN/MAX element of each run and during post-processing we pick the overall MIN/MAX.

$$\mathcal{QRF}_{\text{RPE}}(\Gamma_{MIN(column)}(table)) = \tag{8.9}$$
$$\Gamma_{MIN(column_{enc})}(\sigma_{column_{run}=1}(table_{enc})) \cup$$
$$\Gamma_{MIN(column_{enc})}(\sigma_{column_{run}=2}(table_{enc}))$$

8.3 Security Analysis

In this section we will analyze the security of RPE against the database attacker models presented in Chapter 4.

8.3.1 Security Analysis of the Composed Construction

We have used ROPE from [14] to build RPE. [15] has introduced two security notions on one-wayness properties of ROPE. In this section we find a mapping between RPE

and ROPE to show the relationship between these two encryption schemes in terms of the already introduced security notions in [15].

The one-wayness (OW) property in general 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 ROPE been thoroughly analyzed in [15] as follows: Given z encryptions $y_1, , , y_z$ of randomly chosen x_i from the domain, the adversary $\mathcal{A}$ succeeds if he finds any inverse from $y_1, , , y_z$ [15].

Proposition 2. *Fix a challenge set of size z. Let $\mathcal{A}$ be an OW adversary that outputs a plaintext from $\mathcal{X}$. Then for any adversary $\mathcal{A}$,*

$$Adv^{OW}_{ROPE}(\mathcal{A}) = \kappa \tag{8.10}$$

The question is, what is the one-wayness advantage of RPE applying ROPE as its underlying encryption scheme. To answer this question, we first need to prove that the runs can be treated independently. According to Construction 7, every plaintext to be encrypted goes through two independent phases. First, a *run* is chosen at random, and second an encryption scheme is applied on the plaintext using an independent key for each *Run*. Thus, each run has to be treated independently.

Lemma 12. *Under random partitioning strategy μ, any executions of runs r_i and r_j are indistinguishable.*

Proof. Key Generator $\mathcal{K}$ in the Construction 7 generates statistically independent keys for each run so each run contains statistically independent data from the other runs. □

After showing that the *runs* are independent, we need to find a reduction from the One-Wayness adversary on RPE to the One-Wayness adversary on ROPE as follows:

Lemma 13. *If the adversary on RPE wins the One-Wayness experiment with probability ϵ, then the probability that he inverts something from the i-th run is $\frac{1}{U} \times \epsilon = \kappa$ where κ is the advantage of the adversary playing the one-wayness experiment on the underlying encryption scheme.*

$Adv^{OW}(\mathcal{A}^R) = \frac{1}{X} \geq \ Adv^{OW}_{RPE[X][Y][U]}(\mathcal{A}) \times \frac{1}{U} \leq Adv^{OW}_{ROPE[X][Y]}(A)$

Proof. Given an adversary against RPE with a one-wayness advantage of ϵ, we want to construct an adversary against its underlying encryption scheme. As defined earlier, the one-wayness experiment for the underlying encryption scheme offers the adversary z ciphertexts from a single run and the adversary can invert at least one of them. In order to relate this adversary to the RPE adversary, we need to find a reduction. According to Lemma 12, in RPE we have U independent runs. One of these U runs, lets call it the i-th run, is the run that corresponds to the original one-wayness experiment. Then, we give our RPE adversary random ciphertexts from different runs, of course. Whenever we give

the RPE adversary something from the i-th run, we just choose a ciphertext that is given by the original one-wayness experiment to the underlying adversary. In the end, the RPE adversary wins the newly constructed experiment if and only if the underlying adversary can invert a ciphertext that correspond to the i-th run because only then the RPE adversary can win the one-wayness experiment of the underlying encryption scheme. If the RPE adversary wins with probability ϵ then the probability that he inverts something from the i-th run is $\frac{1}{U} \times \epsilon$ (assuming that approximately equal amounts of ciphertexts per run are given to the adversary). The reason is that an adversary against RPE has the freedom to choose in which run he *wants to break* the one-wayness. And with probability $1/U$ he hits the run that the one-wayness experiment of the underlying encryption scheme can be won at. Therefore we can say that succeeding the one-wayness experiment by an adversary against RPE is upperbounded by the number of runs times the advantage of an adversary against ROPE and lowerbounded by a random adversary, $\mathcal{A}^R$ that randomly returns a value from the domain of size X. □

8.3.2 Security against Domain Attack

As defined in Section 4.2, a domain attack is an adversary model where the attacker has knowledge about the plaintext domain and has access to the encrypted database. In our security analysis we consider that in a domain attack the attacker has precise knowledge of the domain to compute an upperbound probability distribution.

An ordinary OPE scheme such as in [6, 14, 15, 41, 48, 69, 71] allows the domain adversary, $\mathcal{A}$, to efficiently break the encryption by solely using sorting. In other words: $Adv^{\text{ROW}}_{\text{OPE}}(\mathcal{A}) = 1$. This is because OPE leaks the total order of the plaintext values. In order to break the total order among the values we have introduced RPE in this chapter. RPE amends all variants of OPE schemes to a great extent by randomly partitioning the domain into *Runs*, thereby breaking the total order into U partial orders. According to [13,28], the problem of finding the right total order out of U partial orders is classified as inapproximable.

Lemma 14. *ROW-advantage of $\mathcal{A}$ on Det-RPE is defined as:*

$$Adv^{ROW}_{\text{Det-RPE [X][Y][U]}}(\mathcal{A}) = Pr[Exp^{ROW}_{\text{Det-RPE}}(\mathcal{A}) = 1] \quad = \frac{\binom{Rank_x - 1}{Rank_{(y,u)} - 1}\binom{X - Rank_x}{\frac{X}{U} - Rank_{(y,u)}}}{\binom{X}{\frac{X}{U}}} \tag{8.11}$$

As described in Chapter 4, the $rank(x)$ function returns the ordinal rank of plaintext $x \in \mathcal{X}$, i.e. $Rank_x$. In case of RPE, every ciphertext, y, has a rank in its run, yet there is no ranking of the ciphertexts possible across runs. Thus, in Equation 8.11 we use

slightly a different rank function that returns the rank of a ciphertext within its run, u i.e. $rank(y, u) = Rank_{(y,u)}$.

Essentially, this probability is computed by computing the number of possible worlds that the randomized partitioning can have under the condition that plaintext $Rank_x$ corresponds to $Rank_{(y,u)}$ divided by the total number of *possible worlds* that the randomized partitioning can create. Similar computations and analyses are carried out in the context of probabilistic databases [61].

To give a simple and extreme example, the code for "Benzema" can be either $\langle 1, 4\rangle$ or $\langle 2, 1\rangle$ in the Det-RPE Table 8.1 because "Benzema" is encoded either in the first or in the second run and in each case, "Benzema" will have the lowest ciphertext in that run. As a result, the ROW-advantage of an adversary in reversing "Benzema" is 0.5 in this simple example. Here we give a more detailed proof on how the probability distribution in Lemma 14 has been computed.

Proof. Here we give a description on the derivation of the formula in equation 8.11. The denominator holds all possible ways to partition the domain having U runs. We assume that the plaintext values from domain of size X are uniformly distributed among U runs. This means each run holds $\frac{X}{U}$ values. According to combinatorics theory, the number of ways you can select a subdomain of size $\frac{X}{U}$ out of a domain of size X is :

$$\binom{X}{\frac{X}{U}}$$

For RPE the story does not end here. After selecting randomly the values for the first run, there remains $X - \frac{X}{U}$ values from the domain that has to be randomly selected from and put in the second run. This continues until all X values from the domain are distributed among U runs, yielding the following number of combinations to randomly distribute X values in U runs:

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

Now that we have seen how the denominator was derived, we proceed with the numerator. In the numerator we are interested in all possible worlds that can be built having $y = \mathcal{E}nc^{K_u}(x)$ where $rank(x) = Rank_x$ and $rank(y, u) = Rank_{(y,u)}$. Hence, fixing this outcome, the number of combinations we can have for the rest of the elements in a run is, all the possible combinations of ciphertexts that can appear above y in run u selected from all the plaintexts that can appear above x in the domain, i.e. :

$$\binom{Rank_x - 1}{Rank_{(y,u)} - 1}$$

times all the possible combinations combinations of ciphertexts that can appear below y in run u selected from all the plaintexts that can appear below x in the domain, i.e. :

$$\binom{X - Rank_x}{\frac{X}{U} - Rank_{(y,u)}}$$

Our combinatorics only fixes one element which is in run u, therefore we have to also consider all the combinations of distributing the rest of the values in the rest of the runs. All in all we will end up having the following combinatorics in the numerator:

$$\binom{Rank_x - 1}{Rank_{(y,u)} - 1}\binom{X - Rank_x}{\frac{X}{U} - Rank_{(y,u)}}\prod_{i=1}^{U-1}\binom{X - \frac{X}{U}i}{\frac{X}{U}}$$

Luckily, the two products in the numerator and denominator cancel out quite well yielding:

$$\frac{\binom{Rank_x - 1}{Rank_{(y,u)} - 1}\binom{X - Rank_x}{\frac{X}{U} - Rank_{(y,u)}}}{\binom{X}{\frac{X}{U}}}$$

□

In general, the ROW-advantage depends on the size of the domain (i.e., X) and, of course, on the number of runs (i.e., U). So, we will study the sensitivity of the ROW-advantage towards these two parameters in detail here.

Figure 8.3 plots the ROW-advantage for OPE [6, 14, 41, 48], Prob-OPE [51], and RPE with 8 and 16 runs for a domain of 64 sorted values. As a baseline, Figure 8.3 also shows the *ideal* ROW-advantage for a random adversary according to Definition 7 that is constant, $\frac{1}{64}$, for all 64 values of the domain. The closer the ROW-advantage gets to the ideal distribution, the more secure that encryption scheme is against domain attack. As we can see, the state of the art OPE schemes do not provide any security against domain attacks.

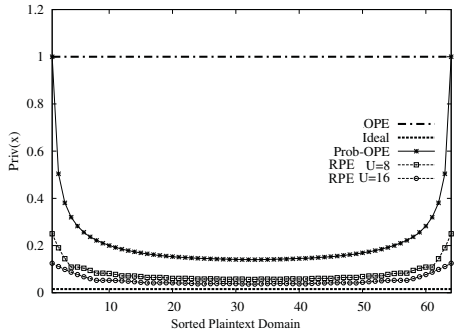


Figure 8.3: ROW-Advantage: Vary Plaintext x, Domain Size=64

The Det-RPE curve for 16 runs is fairly close to *ideal*. In RPE the number of runs is a tunable parameter and should be adjusted according to the privacy / performance requirements of the application. Another observation with RPE 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 probability does not depend on the domain size. Furthermore, it is fortunate that there is a fix to improve the privacy of those values, too: either by inserting dummy values at the extremes [65], or by using a modular order preserving scheme proposed by [15] where the ciphertexts form a ring and the beginning of the domain is hidden.

However, MOPE is immediately broken when query logs are revealed as proven in Subsection 6.2.2. With the *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.

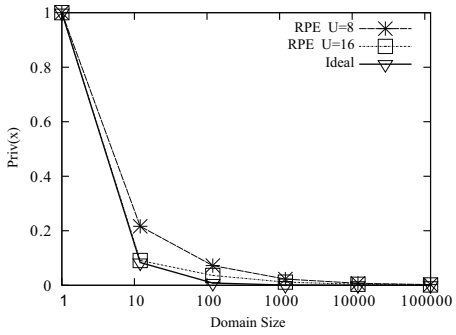


Figure 8.4: ROW-Advantage of Domain Midpoint x, vary Domain Size

As shown in Figure 8.4, 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}. 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.

The equations presented in this section can be tuned to meet the *user's negligibility requirement*. For example, in Equation 8.11, depending on the domain size, the number

of runs can be tuned to meet the security/performance requirements of the system. The higher the number of runs, the closer one gets to the "ideal" threshold, but also the bigger is the performance overhead.

8.3.3 Security against Frequency Attack

As defined in Section 4.3, a frequency attack is an adversary model where the attacker has knowledge about the domain and its underlying frequency distribution. In our security analysis we consider that in a frequency attack the attacker has precise knowledge of the domain and its underlying frequency distribution to compute an upperbound probability distribution.

Det-RPE like MOPE and ROPE is a deterministic encryption schemes thus it is unable to hide the frequency distribution of the underlying plaintext. This is particularly insecure when the plaintext domain has skewed frequency distribution. Deterministic encryption schemes allow an adversary, $\mathcal{A}$, who plays the frequency one-wayness experiment 2 to simply find a mapping between the plaintext space and the ciphertext space just by matching their frequency distributions and thereby discovering the plaintext-ciphertext correspondence.

Since Det-RPE is a deterministic encryption scheme the FOW-advantage of $\mathcal{A}$ on Det-RPE depends on the plaintext frequency distribution, namely $F_{\mathcal{X}}$.

Lemma 15. *Let $G = \{x | Freq_{\mathcal{X}}(x) = Freq_{\mathcal{Y}}(y_x)\}$ be the set of distinct plaintext values having the same frequency as y_x in Experiment 2. Then, the FOW-advantage of the $\mathcal{A}$ adversary on Det-RPE is defined as his winning probability in Experiment 2:*

$$\begin{aligned} Adv^{FOW}_{\text{Det-RPE}}(\mathcal{A}) &= Pr[Exp^{FOW}_{\text{Det-RPE}}(A) = 1] \\ &= \frac{1}{|G|} \end{aligned} \tag{8.12}$$

Lemma 15 is same as Lemma 6, thus it has identical proof which can be found in Subsection 6.1.2.

As also stated in Subsection 6.1.2, a deterministic encryption like Det-RPE can be safe against frequency attacks if the underlying plaintext domain has uniform frequency distribution.

8.3.4 Security against Query Log Attack

To win the query log attack, defined in Section 4.4, the adversary needs to either win the domain attack or the frequency attack with the help of the query logs. In Chapter 5 a framework has been introduced to identify leaky queries. In this section, in order to analyze the security of Det-RPE against query log attack, two questions need to be taken care of:

1. Is Det-RPE indistinguishable under Simple Query Attack? This property assures that rewritten queries, as discussed in Section 8.2, are not leaky, i.e., they do not weaken the RPE-encrypted data.

2. How safe is Det-RPE against domain and frequency attack if the query logs are revealed? This property evaluates the resilience of Det-RPE against the query log attack introduced in Section 4.4.

RPE breaks the total order of OPE schemes by randomly partitioning the domain into partially ordered runs. RPE has two variants according to [51], a deterministic and a probabilistic variant. Here we discuss the deterministic variant of RPE.

Rank Queries and Equality-based Operators. These operators include equality, non-equality and IN predicates, JOIN, DISTINCT, GROUP BY, TOP N, ORDER BY and MIN/MAX. Since RPE is a deterministic scheme, a plaintext literal in a query is rewritten to its corresponding ciphertext. This fulfills the simulatability requirement.

Range and LIKE Predicates. As described in [51], a range query has to be rewritten on each run separately. [51] claims that the runs are independent, however query logs actually reveal the correspondence between the "independent runs". Assume the client submits the following plaintext query:

```
SELECT name FROM customer
WHERE name > 'Messi'
```

This query will be rewritten as follows:

```
SELECT name_run, name_enc FROM customer_enc
WHERE (name_run=1 AND name_enc >= 1123)
      (name_run=2 AND name_enc > 2593)
```

This rewritten query incurs minimum communication and post-processing costs. However, since $\{1123, 2593\} \nsubseteq \{1123\}$, it fails the simulatability test.

Corollary 11. *The adversary exploits the additional information from the leaky queries to discover the correspondence between the "independently" ordered runs. Consequently, the total order can be discovered from the partially sorted runs and RPE degrades to ROPE. This degradation will of course happen, if the query rewrite method presented in Section 8.2 is used.*

Corollary 12. *The advantage of an adversary to win Experiment 3 on Det-RPE, is the maximum probability of an adversary winning either the domain or the frequency attack on Det-RPE. Since Det-RPE is not safe against Simple Query Attack and degrades to ROPE as a result. The maximum probability of an adversary to win Experiment 3 on Det-RPE is 1.*

$$Adv^{QLA}_{\textit{Det-RPE}}(\mathcal{A}) = Pr[Exp^{QLA}_{\textit{Det-RPE}}(\mathcal{A}) = 1] = 1 \tag{8.13}$$

Simulatable Queries

Like other encryption schemes discussed so far, there are certain SQL operators that cannot be simulated on Det-RPE-encrypted database. These SQL operators are called non-simulatable and were discussed using example in this section. Table 8.2 summarizes which SQL operators are simulatable under a Simple Query Attack for the encryption schemes introduced so far in this thesis. NA means that the SQL operator is Not Applicable on the data if that encryption scheme is used.

SQL-Operator	OPE	MOPE	Prob-OPE	Det-RPE
DISTINCT	✓	✓	NA	✓
WHERE (=, !=)	✓	✓	✗	✓
WHERE ($<, >$)	✓	✗	✗	✗
LIKE(Prefix%)	✓	✗	✗	✗
LIKE(%Suffix)	NA	NA	NA	NA
IN	✓	✓	✗	✓
Equi-Join	✓	✓	✓	✓
Non Equi-Join	✓	✗	✓	✓
TOP N	✓	✗	✗	✓
ORDER BY	✓	✗	✓	✓
SUM	NA	NA	NA	NA
MIN/MAX	✓	✗	✓	✓
GROUP BY	✓	✓	NA	✓

Table 8.2: Det-RPE: SQL-Operator Simulatability

As shown in Table 8.2, only range queries are not simulatable for Det-RPE-encrypted databases. This is because each range predicate, is rewritten to separate range predicates for each run. Thus, these separate ranges reveal an approximate order relationship across runs. If enough queries are revealed then the total order will be discovered and Det-RPE with stronger security guarantees against attackers with domain knowledge downgrades to the weak OPE with no security guarantees against attacker with domain knowledge. In order to fix the problem with the non-simulatable queries, we take the same approach taken for Prob-OPE scheme, which is to use a new query rewrite method, namely the Fixed Range Query Rewrite.

8.4 Fixed Range Query Rewrite

There are many ways to rewrite a query. Usually a query is rewritten to get the best response time, i.e. most of the processing has been pushed down to the cloud. In this

thesis, we have introduced an attack that is based on the information that an adversary can extract from the query logs. We have shown that a Det-RPE is not IND-SQA because it leaks information about the valid ranges, and consequently about the correspondence between the random partitions, making them no more random. In subsection 8.3.4 we have shown that the leakage is caused due to range queries. In this section we introduce a new rewrite called Fixed Range Query Rewrite that fixes the problem with the leaky ranges.

The idea of *Fixed Range Query Rewrite* is to divide the plaintext domain, $\mathcal{X}$ into j disjoint fixed-sized sub-ranges of size r, fr_i where $i = 1, ..., j$ and $|fr_i| = r$. Whenever a range query is asked, the smallest units that are returned are those *fixed ranges* that contain the user's results.

As an example, consider Table 8.3 where deterministic RPE has been used for encryption. The client submits the query:

```
SELECT name FROM customer WHERE name > 'Messi'
```

Using Det-RPE one possible way to rewrite this query is:

```
SELECT name_run,name_enc FROM customer_enc
       WHERE (name_run = 1 AND 3 < name_enc < 10)
          OR (name_run = 2 AND 3 < name_enc < 10)
```

This query can be fully executed in the cloud such that the result only needs to be decrypted in the trusted component. However, since this query is revealing some information about the correspondence of ciphertexts across runs, we will use Fixed Range Query Rewrite mechanisms to fuzzify the result. Thus, the original query will be rewritten as follows to return all the fixed ranges, namely fr_1 and fr_2, that contain the result:

```
SELECT name_run,name_enc FROM customer_enc
       WHERE (name_run = 1 AND 0 < name_enc < 10)
          OR (name_run = 2 AND 0 < name_enc < 10)
```

In this example the whole table is returned. The result includes two additional false positives, namely, "Benzema" and "Messi" that need to be filtered out during the post-processing.

Since *Fixed Ranged Query Rewrite* returns a super-set of the result that needs to be post-filtered, this method will lead to performance loss. On the security side, the guarantees and analysis will deviate from what we have discussed earlier in Section 8.3.2.

Table 8.3: Two fixed ranges of size 3, $r = 3$

Fixed Range	Customer Names	Run 1	Run 2	($\langle$ run, y $\rangle$)
fr_1	Benzema		3	$\langle 2, 3\rangle$
	Messi	1		$\langle 1, 1\rangle$
	Neymar		5	$\langle 2, 5\rangle$
fr_2	Ronaldo		9	$\langle 2, 9\rangle$
	Sneijder	6		$\langle 1, 6\rangle$
	Xavi	7		$\langle 1, 7\rangle$

8.4.1 Security against Query Log Attack

In case of RPE, range queries are problematic because they help to reconstruct the total order, which is exactly what RPE is trying to hide. Hence, *Fixed Ranged Query Rewrite* helps again to limit the ROW-advantage under query log attack. Assuming we have a uniform code distribution within a range in all runs, Equation 8.11 will change if Fixed Range Query Rewrite has been used to rewrite the queries:

$$Adv^{\text{ROW}}_{\text{Det-RPE-FR}}(\mathcal{A}^Q) = Pr[Exp^{\text{ROW}}_{\text{Det-RPE-FR}}(\mathcal{A}^Q) = 1] \quad = \frac{\binom{Rank_x^{fr}-1}{Rank_{(y,r)}^{fr}-1}\binom{r-Rank_x^{fr}}{\frac{r}{u}-Rank_{(y,r)}^{fr}}}{\binom{r}{\frac{r}{U}}} \tag{8.14}$$

Informally speaking, by using the Fixed Range Query Rewrite the randomness is no more distributed throughout the domain, but is only available within the fixed range. From Equation 8.14 we reach the following conclusions:

- If $U = r$ then we have a uniform probability distribution. This means although no range query is possible within a fixed range (like having AES in each range), among different fixed ranges, range queries are possible.
- Getting better security is achievable by both increasing the range size and the number of runs. However they both come at the cost of performance.
- Along the number of runs, range size is the other tunable parameter to adjust the ROW-advantage of Det-RPE whereas the domain size does not play a role anymore.

Figure 8.5 shows the QLA-advantage in case of a query log attack when Fixed Range Query Rewrite is used assuming that the plaintext values have a uniform frequency distribution . QLA-advantage is based on Equation 8.11. However this time another tunable parameter is added to the equation namely the range size r. The bigger the range size the better the privacy gets as shown in Figure 8.5 where $r = 64$ which is the

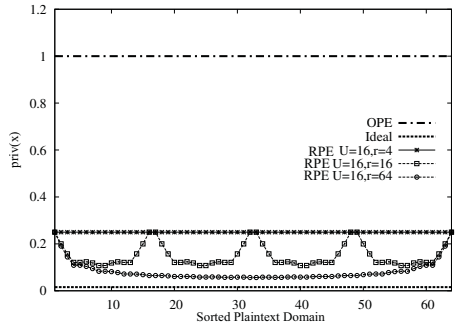


Figure 8.5: QLA-Advantage: Vary Plaintext x, different values for U and r

size of the domain. On the other hand, since the fixed ranges are queried atomically, this means that the whole domain needs to be shipped. In case of $r = 16$, it is shown that each fixed range has a similar pattern to what has been shown in Figure 8.3 for domain attack. Thus, the extreme values in the range are more exposed. This will quickly flatten out once bigger domains with wider ranges have been chosen.

In Section 8.3.4 we have shown that RPE is safe against Simple Query Attack if the rewritten queries do not contain any range predicates. However, RPE downgrades to OPE if the queries contain ranges. Here we show how Fixed Range Query Rewrite fixes this problem.

Assume the client submits the following plaintext query:

```
SELECT name FROM customer
WHERE name > 'Messi'
```

This query will be rewritten as follows to incur the minimum communication and post-processing cost:

```
SELECT name_run, name_enc FROM customer_enc
WHERE (name_run=1 AND name_enc >= 1123)
      (name_run=2 AND name_enc > 2593)
```

However, such a rewrite is not simulatable by a random system. If we use the Fixed Range Query Rewrite however, we can make the above query simulatable as follows:

```
SELECT name_run, name_enc FROM customer_enc
WHERE name_fr > 1
```

This rewritten query is simulatable. In general, although Fixed Range Query Rewrite suggests leaking the correspondence between individual runs at the fixed range boundaries, but it keeps the security tunable and measurable by varying the fixed range size. Allowing arbitrary range queries on RPE can possibly turn RPE to OPE, whereas Fixed Range Query Rewrite makes RPE robust against Fixed Range Query Rewrite since it reduces the correspondence leakage under control.

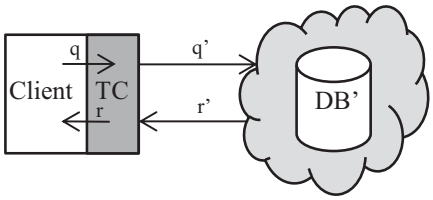


Figure 8.6: Client-Side Security

8.5 Experiments

This section assesses the performance overhead of Det-RPE in the context of the 22 queries and 2 refresh functions of the TPC-H benchmark. We present results for Plain (not encrypted database), OPE (state of the art for range and rank queries), Prob-OPE (resilient against frequency attack), Prob-OPE with Fixed Range Query Rewrite (resilient against frequency and query log attack).

The fixed ranges for the date data types are selected to be a quarter, which is three months (90 days). For varchar the fixed ranges are bucketized to alphabetically, i.e. all words beginning with "A" are in fr_1, are words beginning with "B" are in fr_2 and so on. For the numeric values the fixed ranges are selected so that each fixed range accommodates between 10 or 20 percent of the values of the domain.

8.5.1 Benchmark Environment

All experiments were conducted on two separate machines for client and server which corresponds to the architecture shown in Figure 8.6. The client and the trusted component were written in Java, ran on a machine with 24 GB of memory and communicated to the database server using JDBC. The server machine had 132 GB of memory available and hosted a MySQL 5.6 database. Both machines had 8 cores and ran a Debian-based Linux distribution.

We used a 10 GB data set (scaling factor 10) and measured end-to-end response time for all queries in separation. Queries that did not finish within 30 minutes where canceled and reported as a time-out. This is why we do not show results for Q9, which even times out for *Plain*. Metrics used in aggregate functions (e.g., volume of orders) and surrogates (e.g., order numbers) were left unencrypted while all other (sensitive) attributes, such as names, dates, etc. were encrypted. Whenever SQL operators on encrypted data were not supported, the entire data was shipped and then post-processed at the trusted component.

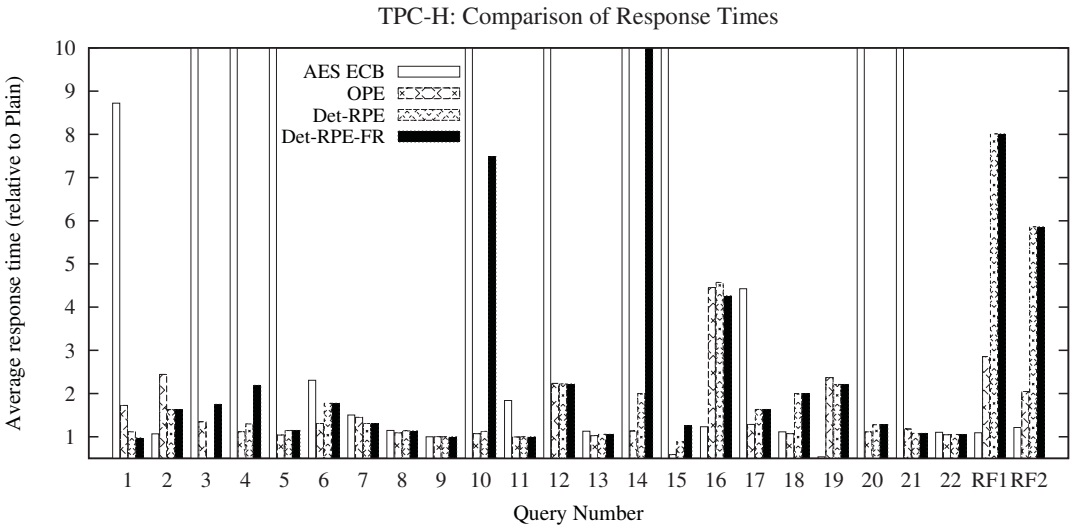


Figure 8.7: RPE Response Time: Relative to Plain

8.5.2 Response Time

Figure 8.7 shows the relative response time of Det-RPE and Det-RPE-FR compared to OPE as proposed in [14] and used in [47,66] and AES in ECB (deterministic implementation of AES) and relative to the plain database. Figure 8.8, on the other hand, shows the average response time values including the standard deviation of running each experiment 1000 times. As illustrated we consider a time-out to be for a query that takes more than 600 seconds. Additionally, Figure 7.7 shows the break down of the time a query spends on the server and on the client. The following concrete observations are to be made from these figures:

- *Range Predicates (Q1, Q5, Q6, Q14, Q15, Q20):* Being able to answer range queries efficiently was the most important motivation for RPE. We see that RPE and OPE do well in that regard as they prevent irrelevant data (false positives) from being shipped to the client. AES-ECB on the other hand, cannot process range queries on encrypted data and therefore the entire data has to be shipped, which often results in a big performance loss. As shown in Figure 8.7, AES-ECB hits the time-out limit for Q3, Q4, Q5, Q10, Q12, Q14, Q15, Q20 and Q21.

- *Group-By (Q3, Q10, Q16, Q18):* Group by clauses on encrypted attributes are fully supported for deterministic encryption.

- *Top-N (Q2, Q3, Q10, Q18):* Top-N queries can be processed as sub-queries as described in Section 8.2.4. 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%"*. In case of Q9 even the plain database times out, that is why we do not consider Q9 measurements in our benchmarks.

- *Non-equi Joins (Q2, Q21):* While equi-joins work well in RPE, non-equi joins are difficult for all encryption techniques. For Q2, the top-N parallelism compensates for the non-equi-join penalty. For Q21, the RPE variant perform well because *l_shipdate*, *l_commitdate* and *l_receiptdate* are encrypted in the same run.

- *Fixed Range Query Rewrite.* According to Figure 8.7, in a few cases such as Q1, Fixed Range Query Rewrite has a better response time. The reason is using parallel queries, since each fixed range should be queried independently, thereby automatically partitioning the handling and aggregation of data. In most of the cases Det-RPE-FR performs the same as Det-RPE because the ranges used in the query spans exactly over one or more fixed ranges, omitting the shipment of false positives. In Q14 however the Fixed Range Query Rewrite method times out. This is because unlike Det-RPE, Det-RPE-FR has to do costly OPE decryption on the client side as shown in Figure 8.8 to be able to compute the aggregate values. Det-RPE-FR as shown in Figure 8.8 has a big standard deviation. This is because the range predicate in Q10 sometimes covers exactly one fixed range, and sometimes spans over two fixed ranges. In the latter case the query cannot be aggregated at the server side and has to be shipped to the client-side for post-filtering and aggregation.

The number of runs is an important parameter of all RPE schemes. Figure 8.9 shows how the response time of Det-RPE varies with the number of runs for some selected TPCH queries. Note that RPE with one run is equivalent to an OPE scheme. In general, we see that for most query operators partitioning not only increases security but also either decreases response time or does not have any effect on it. More concretely, we chose these queries because they cover a wide spectrum of different operators (range predicates, aggregates, Top N, sub-selects, etc.). 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 and are therefore executed with a degree of parallelism equal to the number of runs, which result in a decreasing response time. Q1, Q4 and Q5 are typical representatives for most TPCH-queries: their curves stay fairly flat. This shows that partitioning does generally not hurt RPE. as hypothesized in section 8.2, *equi-joins*

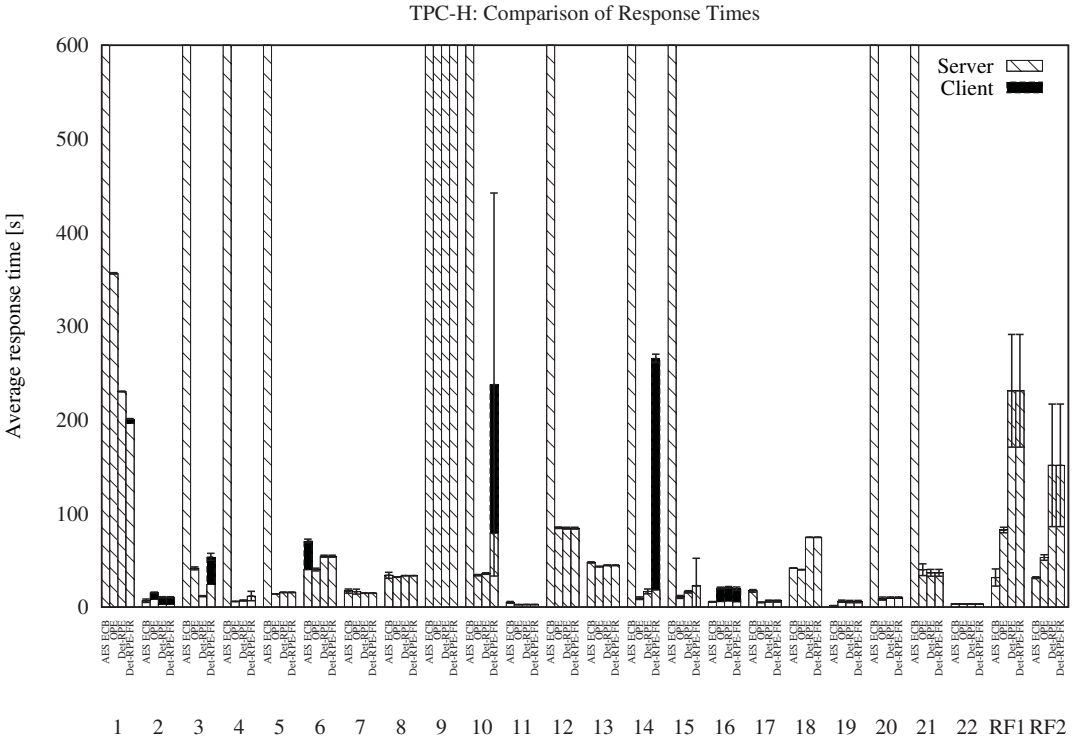


Figure 8.8: RPE Response Time: Client-Server Breakdown

work the same as for plaintext, therefore the curve stays fairly flat. The same is true for queries that involve a low number of inequality predicates, like Q20 (which involves 3 inequality predicates, one of them being a prefix predicate). Inequality predicates become more complex the more runs there are because the OR clauses of the rewritten queries get longer (see section 8.2.6). This is why queries that involve more such predicates (like Q6) increase linearly with the number of runs. A very interesting trend can be observed for queries that involve many *equality predicates*: as the predicates are mostly on indexed columns and we use composite B-tree indexes of the form $\{a.run, a.code\}$, the index traversal becomes faster the more runs we have (probably because the most selective nodes containing the runs are in the uppermost level of the tree) and response time decreases linearly up to the point where it equals the performance of *Plain*. *Top N* queries are executed with a degree of parallelism equal to the number of runs, which results in a decreasing response time up to the point where the number of partitions equals the number of CPU cores (which was 8 in our case).

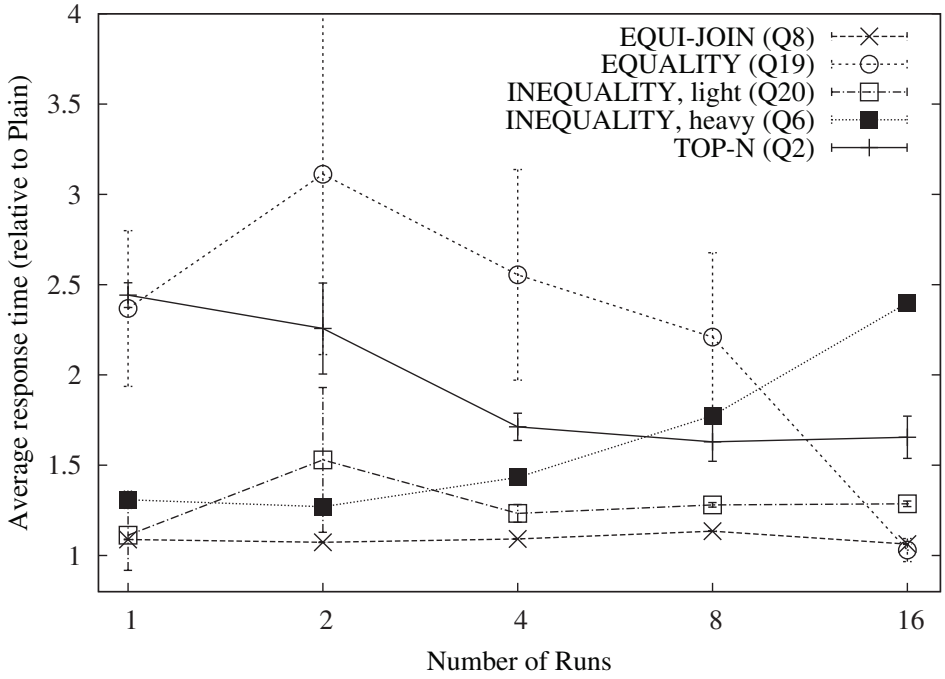


Figure 8.9: RPE Response Time: Vary number of Runs

8.5.3 Network Costs

Figure 8.10 shows the network cost of AES ECB, OPE, Det-RPE and Det-RPE-FR (Det-RPE with Fixed Range Query Rewrite method) to the unencrypted relative to the

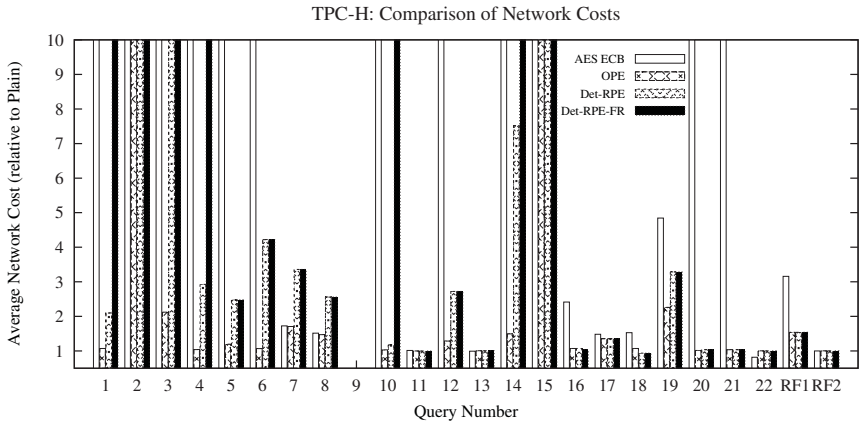


Figure 8.10: RPE Network Cost: Relative to Plain

plain database. Please note that Q9 caused a time-out even for the plain database, that is why we are not interested in its performance for our comparisons.

The very important and foremost observation is that AES incurs far higher shipment costs than the OPE variants. This is because to evaluate every range and rank query all the data has to be shipped to the trusted component, decrypted, post-filtered and then aggregated. As shown in Subsection 8.5.2 the response time is also most of the time heavily affected by the amount of data that needs to be shipped in most of the queries.

In general, Det-RPE has higher network cost than OPE as shown in Figure 8.10 for Q1,Q3,Q5,Q6,Q7,Q8,Q12 and Q14. This is because we add the size of the query which is sent to the encrypted database to our network cost. Since RPE has to send disjunctive queries and the size of the query increases with increasing number of runs, it causes an additional network cost. Nevertheless, in terms of query results both RPE and OPE ship the same number of bytes back to the trusted component. Another less common reason is when an ORDER BY and LIMIT clause exist in a query, the LIMIT has to be replicated for each partition separately since there is no total order as described in Subsection 8.2.4.

Another important observation is that the Fixed Range Query Rewrite method, as expected, incurs much higher network cost since it is shipping fixed ranges with false positives to the client. Nevertheless, both the network cost and reponse time for Det-RPE-FR is most of the time far better and within the acceptable range compared to AES ECB.

8.6 Related Work

The concept of having an adversary with domain knowledge has always been mentioned in [6], but never been formally treated by any of the previous work. There has been several attempts to amend the OPE schemes against domain adversaries.

Chaotic OPE (COPE) is proposed by [57]. COPE divides the plaintext domain into random sized buckets. Then each bucket is randomly distributed, and the values within a bucket will be either in an ascending or descending order. After this randomization phase, the values are being encrypted. COPE hides the order of the encrypted values by changing the order of buckets in the plaintext domain.

Multivalued-partial OPE (MV-POPES). In [36], the authors create partial orders by first binning the plaintext data in fixed sized bins and then randomly distribute the bins and then encrypt the values. Both [36, 57] do not formalize the adversary model they are trying to protect the data from.

8.7 Conclusion

Table 8.4 summarizes the SQL-operators that are supported by the state of the art encryption schemes, OPE, Prob-OPE and RPE. HES is an abbreviation for Homomorphic Encryption Schemes, and AES-CBC is a probabilistic and semantically secure mode of AES as opposed to AES-ECB which is deterministic and not semantically secure. As shown in Table 8.4, RPE supports all the SQL operators that OPE supports.

SQL Operator	AES-ECB	OPE	Prob-OPE	RPE	Paillier [44]	AES-CBC
DISTINCT	✓	✓	✗	✓	✗	✗
WHERE (=, !=)	✓	✓	✓	✓	✗	✗
WHERE (<, >)	✗	✓	✓	✓	✗	✗
LIKE(Prefix%)	✗	✓	✓	✓	✗	✗
LIKE(%Suffix)	✗	✗	✗	✗	✗	✗
IN	✓	✓	✓	✓	✗	✗
Equi-Join	✓	✓	✓	✓	✗	✗
Non Equi-Join	✗	✓	✓	✓	✗	✗
TOP N	✗	✓	✓	✓	✗	✗
ORDER BY	✗	✓	✓	✓	✗	✗
SUM	✗	✗	✗	✗	✓	✗
MIN/MAX	✗	✓	✓	✓	✗	✗
GROUP BY	✓	✓	✗	✓	✗	✗

Table 8.4: RPE: Supported SQL Operators

Table 8.7 summarizes the result of the security analysis done in this chapter on RPE. Each row represents an OPE variant, and each column presents an attacker model introduced in Chapter 4. Each cell shows how low an encryption scheme downgrades under a given attack. By plain we mean an unencrypted database. The FR postfix indicates the case when *Fixed Range Query Rewrite* mechanism is used to rewrite queries on the encrypted database.

OPE variant	Domain Attack	Frequency Attack (uniform)	Frequency Attack (skewed)	Query Log Attack
OPE	Plain	OPE	Plain	Plain
Prob-OPE	Prob-OPE	Prob-OPE	Prob-OPE	Plain
Prob-OPE-FR	Prob-OPE	Prob-OPE	Prob-OPE	Prob-OPE
RPE	RPE	RPE	Plain	Plain
RPE-FR	RPE	RPE	Plain	RPE

Table 8.5: RPE: Security Downgrade

Please note that each encryption variant in Table 8.7 has a tunable probability distribution under each attacker model. In general, the encryption schemes introduced in this thesis have a performance/security knob that can be adjusted according to the performance/security requirements of the system.

In summary, RPE has provably higher security and functionality than any traditional weak encryption scheme at the same performance. On the other hand, RPE has much better performance and functionality than any traditional strong encryption scheme. As discussed in Section 8.3, the exact level of security depends on the implementation details of RPE: RPE can be configured to have strong encryption properties, but in this case performance and functionality deteriorate in the same way as they do for other strong encryption schemes. In fact, weak and strong encryption can be considered as special cases of RPE: weak encryption corresponds to RPE with one *Run*; strong encryption corresponds to RPE with an infinite number of *Runs*. RPE covers the entire space in between, thereby providing a *tunable security/performance parameter* for many practical applications.

9

Probabilistic RPE (Prob-RPE)

In the previous chapter we have introduced RPE as a non-order-preserving encryption scheme that can handle range and rank queries as efficiently as an order-preserving encryption scheme and offer improved security. Although a deterministic RPE fulfills the equality preserving property and thereby facilitates the execution of a large class of queries, it has a trivial disadvantage i.e. its inability to hide the frequency distribution of its underlying plaintext values. Therefore, a **probabilistic** encryption scheme must be used to ensure privacy against frequency attacks. To see why, assume that the attacker knows that Neymar has placed two orders, Messi has placed one order, etc. If Neymar is represented by a single ciphertext, then the attacker can easily infer that ciphertext by counting the number of occurrences of each *cust* 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 frequency skew (without exact values). However, to be safe in the general case, a probabilistic encryption scheme is required to protect confidential data against a frequency attack.

In this chapter we present the probabilistic variant of Randomly Partitioned Encryption (Prob-RPE). RPE is made probabilistic in two ways: (a) by having the same plaintext encrypted differently within each run, and (b) by assigning the same plaintext value to different runs to guarantee database dynamism and improved security. (a) is achieved by using Prob-OPE as introduced in Chapter 7 to encrypt the plaintext values within each run, thereby still preserving the total order within a single run. (b) is achieved by using a probabilistic $ChooseRun()$ function.

In Det-RPE the $ChooseRun()$ function has chosen a *Run* deterministically, i.e. same plaintext always will be encrypted in the same *Run*. Also, in each *Run* we have

Plaintext	Run 1	Run 2	Run 3	Codes
Benzema	1		2	$\{\langle 1,1\rangle\langle 3,2\rangle\}$
Messi		1,4		$\{\langle 2,1\rangle\langle 2,4\rangle\}$
Neymar	3	7		$\{\langle 1,3\rangle\}\langle 2,7\rangle$
Ronaldo	5	9	3	$\{\langle 1,5\rangle\langle 2,9\rangle\langle 3,3\rangle\}$

Table 9.1: Prob-RPE with Three Runs

used the deterministic $\mathcal{E}nc_{\text{OPE}}$ function from [14]. In the contrary, for Prob-RPE we will use a randomized $ChooseRun()$ function to select the run in which the value has to be encrypted in, and a probabilistic order-preserving encryption scheme to encrypt the values within a run.

Table 9.1 shows an example for Prob-RPE. Such an encryption scheme can be used to protect data against frequency attacks. The same techniques (e.g., query rewrite) can be applied to Det-RPE and Prob-RPE. The remainder of this section describes these techniques in more detail.

With Prob-RPE, it is possible that the same value has two different ciphertexts within the same run using the Prob-OPE from Chapter 7 instead of OPE as the underlying weak encryption scheme. For instance, "Messi" has two ciphertexts in Run 2 in Table 9.1.

As shown in Figure 9.1, first the $ChooseRun()$ function randomly selects a run, $u \in \mathcal{U}$ where plaintext x has to be encrypted in. Afterwards, the plaintext x is send through the Prob-OPE scheme where a random bit string is concatenated to the least significant bit of x resulting in x'. x' is then sent to the encryption function of run u, namely $\mathcal{E}nc_u$.

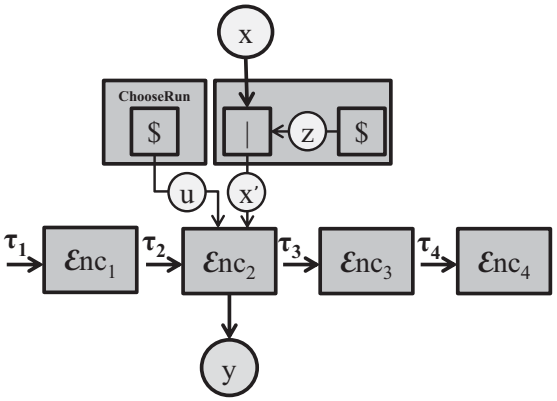


Figure 9.1: Probabilistic RPE

Prob-RPE construction is defined as follows:

Construction 8. *Let $\mathcal{P}rob\text{-}\mathcal{OPE}(\mathcal{K}_p, \mathcal{E}nc_p, \mathcal{D}ec_p)$ be a probabilistic order-preserving encryption scheme from Chapter 7. We define a probabilistic RPE scheme, $\mathcal{P}rob\text{-}\mathcal{RPE}(\mathcal{K}_{Prob-RPE}, \mathcal{E}nc_{Prob-RPE}, \mathcal{D}ec_{Prob-RPE})$, as follows:*

- *$\mathcal{K}_{Prob-RPE}$ runs $\mathcal{K}_p$ independently for each run and returns U keys, namely $(K_1, ..., K_U)$.*
- *$\mathcal{E}nc_{Prob-RPE}$ takes K_u and x, as input where $u = \mathcal{C}hooseRun()$. Then it returns u and $y = \mathcal{E}nc_p(K_u, x)$, i.e. (u, y).*
- *$\mathcal{D}ec_{Prob-RPE}$ takes (u, y) as input and returns $x = \mathcal{D}ec_p(K_u, y)$.*

9.1 Creating Prob-RPE Databases

Prob-RPE is similar to RPE and it can be combined with any other encryption technique within a database. The only assumption made is that, if applied, Prob-RPE 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 that are encrypted using the same encryption function. If a key of a table is encrypted using Prob-RPE, for instance, then all foreign keys to that table must be encrypted using RPE so that joins can be computed in the encrypted database and the encrypted database can check for referential integrity constraints. In the running example of customer-order relationship from the previous chapter, we would recommend to encrypt *Customer.name*, *Customer.city*, and *Order.cust* using RPE. Other *info* fields which could potentially identify a customer and are subject to range predicates such as *age* should also be encrypted using RPE. 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 (non order-preserving) 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 until a practical homomorphic encryption technique with a secure key size has been found. In our experience, it is typically obvious which domains to encrypt and for which domains RPE (either DET-RPE or PROB-RPE) is advantageous. The meta-data that records which attributes are encrypted in which way must be maintained by the trusted component.

If an RPE scheme (either probabilistic or deterministic) is used to encrypt a column, then for each column two columns need to be created in the encrypted database. As shown in Figure 9.2, one column holds the run number and one column holds the ciphertext generated by the encryption function of that run. A combined index is defined on each $(run, ciphertext)$ pair. Revealing runs enables query processing on encrypted data.

Name Encryption

Plaintext	RPE(run,ciphertext)
Messi	$\langle 2, 1 \rangle$
Neymar	$\langle 2, 7 \rangle$
Neymar	$\langle 1, 4 \rangle$

Table Customer

name_run	name_enc	info
2	1	...
2	7	...
1	4	...

Table Orders

id	cust_run	cust_enc
1	2	1
2	2	7
3	2	7
4	1	4

Figure 9.2: Prob-RPE: Keys and Foreign Keys

In case Prob-RPE 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 9.2. "Neymar" is encrypted using two ciphertexts. Correspondingly, two tuples must be stored for "Neymar" in the *Customer* table. *Orders* can refer to either of these tuples. In Figure 9.2, for instance, Orders 2 and 3 refer to the first "Neymar" Customer tuple ($\langle 2, 7 \rangle$) and Order 4 refers to the second "Neymar" Customer tuple ($\langle 1, 4 \rangle$).

9.2 Query Rewrite

As described in Chapter 2, the trusted component rewrites SQL queries so that they can be processed by the database system that hosts the encrypted data in the cloud. Furthermore, the trusted component post-processes query results returned by the cloud. This post-processing involves decrypting the RPE ciphertexts and it may involve post-filtering and post-aggregating the results.

In the previous chapter we have already shown how SQL queries are rewritten for Det-RPE-encrypted databases. Since Prob-RPE is a probabilistic encryption scheme, some SQL operators such as DISTINCT and GROUP BY are no more supported and therefore their query results need to undergo post-processing and post-aggregation on the trusted component. In general, plaintext values in the queries are replaced by their corresponding ciphertexts. Furthermore, simple predicates may result in disjunctions depending on the number of runs affected by the predicates. This section focuses on more complex queries. In particular, this section details how `GROUP BY`, and `ORDER BY` (e.g., Top N) queries can be rewritten. Eventually, using an extended version of relational algebra, we show how query rewrite for each supported SQL operator would work.

9.2.1 `GROUP BY` Queries

While it was trivial to perform GROUP BY queries on Det-RPE (as discussed in Section 8.2.3), a Prob-RPE scheme requires post-processing by the trusted component. 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 cloud to the trusted component, with n being the number of ciphertexts for each plaintext. If the query involves a HAVING clause, then this HAVING clause must be executed in the trusted component after decryption, 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. Fortunately, queries with highly selective HAVING clauses are rare in practice.

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 homomorphic encryption scheme once these techniques become practical (Section 3.1). If metrics are encrypted using RPE, only *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 trusted component and grouping and aggregation need to be executed in the trusted component.

9.2.2 ORDER BY Queries

ORDER BY queries can be carried out in the same fashion as for Det-RPE. However, it gets trickier once the ORDER BY clause is combined with a TOP N clause. A TOP N query on Prob-RPE-encrypted database results in duplicates that are encrypted differently. Nevertheless, TOP N clause can be adjusted TOP N$\times$ n where n is the maximum number of codes a plaintext can have which is 2^k where k is the number of random bits concatenated to the original plaintext during the Prob-OPE process.

9.2.3 Updates

Once a Prob-RPE-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 encrypted using RPE. A run is selected randomly using a uniform distribution and a new ciphertext for the new value is created in that run as described in Section 9.1. Again, if it is not possible to generate a new ciphertext in that run (there is no gap for the new ciphertext at the right place), then a different run is selected or simply a new (empty) run is created.

For Prob-RPE more care needs to be taken to preserve the referential integrities. This is particularly relevant if Prob-RPE is used to encrypt key/foreign-key relations. Once an insert is submitted to the child table, using Prob-RPE it could be the case that a new ciphertext is generated for the foreign key attribute. This fact implies that the

parent table might not have the new ciphertext and therefore result in a foreign key constraint violation. To avoid this problem, once a new value is inserted in a child table, an insert operation is triggered on the parent table with the new ciphertext to preserve the referential integrity constraints. In the worst case this insert might cascade to multiple tables, and that is why this operation is called *cascading inserts*. Cascading inserts make sure that referential integrity, and consequently JOINS are supported by Prob-RPE-encrypted databases. Section 9.5 shows how the performance is affected using Prob-RPE.

9.2.4 Formal Rewrite Rules

Here we formalize the query rewrite rules for Prob-RPE using an extended version of relational algebra, that expresses aggregations as Γ and ORDER BY as $\mathcal{O}$. Let $\mathcal{QRF}$ denote the *Query Rewrite Function* that is called by the trusted component to rewrite a plaintext queries, q, submitted by the *client* as input, into a rewritten query, q' to be processed on the encrypted database, DB', i.e. $\mathcal{QRF}(q) = q'$. In general the rewrite rules of Prob-RPE result from the intersection between Prob-OPE (introduced in Chapter 7) and Det-RPE (introduced in Chapter 8). More concretely, the queries in Prob-RPE are rewritten using the following rules:

- Equality Predicate: In Prob-OPE the equality predicate changes into a range predicate, and in Prob-RPE a certain plaintext value can be in any run, consequently the query is rewritten in as a range query that hits all the runs.

$$\mathcal{QRF}_{\text{Prob-RPE}}(\sigma_{column=x}(table)) = \tag{9.1}$$
$$\sigma_{column_{run}=1 \,\wedge\, column_{enc} \geq\, \mathcal{E}nc^{\text{Prob-OPE}}_{K_1}(x|z_l) \wedge column_{enc} \leq\, \mathcal{E}nc^{\text{Prob-OPE}}_{K}(x|z_u))}(table_{enc})$$
$$\sigma_{column_{run}=2 \,\wedge\, column_{enc} \geq\, \mathcal{E}nc^{\text{Prob-OPE}}_{K_2}(x|z_l) \wedge column_{enc} \leq\, \mathcal{E}nc^{\text{Prob-OPE}}_{K}(x|z_u))}(table_{enc})$$
$$\ldots$$

- Inequality Predicate: It follows the same principle as in the equality predicate. The only difference is the query should exclude the range where the plaintext can be encrypted in.

$$\mathcal{QRF}_{\text{Prob-RPE}}(\sigma_{column!=x}(table)) = \tag{9.2}$$
$$\sigma_{column_{run}=1 \,\wedge\, column_{enc} <\, \mathcal{E}nc^{\text{Prob-OPE}}_{K_1}(x|z_l) \wedge column_{enc} >\, \mathcal{E}nc^{\text{Prob-OPE}}_{K}(x|z_u))}(table_{enc})$$
$$\sigma_{column_{run}=1 \,\wedge\, column_{enc} <\, \mathcal{E}nc^{\text{Prob-OPE}}_{K_1}(x|z_l) \wedge column_{enc} >\, \mathcal{E}nc^{\text{Prob-OPE}}_{K}(x|z_u))}(table_{enc})$$
$$\ldots$$

- IN Predicate: Each plaintext in the IN predicate will turn into a range predicate on all runs.

- Range Predicate:

$$\mathcal{QRF}_{\text{Prob-RPE}}(\sigma_{column>x}(table)) = \tag{9.3}$$
$$\sigma_{column_{run}=1 \wedge column_{enc}> \mathcal{E}nc_{K_1}^{\text{Prob-OPE}}(x|z_u)}(table_{enc})$$
$$\sigma_{column_{run}=2 \wedge column_{enc}> \mathcal{E}nc_{K_2}^{\text{Prob-OPE}}(x|z_u)}(table_{enc})$$
$$...$$

- Like Predicate:

$$\mathcal{QRF}_{\text{Prob-RPE}}(\sigma_{column\ LIKE\ 'M\%'}(table)) = \tag{9.4}$$
$$\sigma_{column_{run}=1 \wedge column_{enc}\geq \mathcal{E}nc_{K_1}^{\text{Prob-OPE}}('M'|z_l) \wedge column_{enc}< \mathcal{E}nc_{K_1}^{\text{Prob-OPE}}('N'|z_l))}(table_{enc})$$
$$\sigma_{column_{run}=2 \wedge column_{enc}\geq \mathcal{E}nc_{K_2}^{\text{Prob-OPE}}('M'|z_l) \wedge column_{enc}< \mathcal{E}nc_{K_2}^{\text{Prob-OPE}}('N'|z_l))}(table_{enc})$$
$$...$$

- EQUI-JOIN: For RPE the JOIN has to make sure that values being joined are in the same run. The rewrite rule will be as follows:

$$\mathcal{QRF}_{\text{Prob-RPE}}(table1 \ \bowtie_{table1.column=table2.column} \ table2) = \tag{9.5}$$
$$table1_{enc} \ \bowtie_{t1_{enc}.column_{enc}=t2_{enc}.column_{enc} \wedge t1_{enc}.column_{run}=t2_{enc}.column_{run}} \ table2_{enc}$$

- NON EQUI-JOIN: For RPE the rewrite has to make sure to only compare values within same run to each other. The final interleaving will be done in the post-processing step at the client-side.

$$\mathcal{QRF}_{\text{Prob-RPE}}(table1 \ \bowtie_{table1.column>table2.column} \ table2) = \tag{9.6}$$
$$\sigma_{column_{run}=1}(table1_{enc} \ \bowtie_{table1_{enc}.column_{enc}>table2_{enc}.column_{enc}} \ table2_{enc}) \cup$$
$$\sigma_{column_{run}=2}(table1_{enc} \ \bowtie_{table1_{enc}.column_{enc}>table2_{enc}.column_{enc}} \ table2_{enc}) \ ...$$

- ORDER BY: In RPE each run needs to be sorted separately. The result is then merged after decryption in the post-processing phase.

$$\mathcal{QRF}_{\text{Prob-RPE}}(\mathcal{O}_{column}(table)) = \tag{9.7}$$
$$\Omega_{column_{enc}}(\sigma_{column_{run}=1}(table_{enc})) \cup$$
$$\Omega_{column_{enc}}(\sigma_{column_{run}=2}(table_{enc}))$$

- MIN/MAX: In RPE since we have multiple runs, and each of them could hold the MIN/MAX element but we do not know which one has it. Thus, we return the

MIN/MAX element of each run and during post-processing we pick the correct MIN/MAX.

$$\mathcal{QRF}_{\text{Prob-RPE}}(\Gamma_{MIN(column)}(table)) = \\ \Gamma_{MIN(column_{enc})}(\sigma_{column_{run}=1}(table_{enc})) \cup \\ \Gamma_{MIN(column_{enc})}(\sigma_{column_{run}=2}(table_{enc})) \tag{9.8}$$

9.3 Security Analysis

In this section we will analyze the security of RPE against the database attacker models presented in Chapter 4.

Notation. Let $\mathcal{X}$ be the set of plaintext values from a finite domain, and $\mathcal{Y}$ be the set of ciphertext values. The size of $\mathcal{X}$ is denoted as $X = |\mathcal{X}|$; the same applies for the size of $\mathcal{Y}$, $Y = |\mathcal{Y}|$. Plaintext elements are denoted as x and ciphertext elements as y. Additionally, we denote the Key set to be $\mathcal{K}eys$ and $K \xleftarrow{\$} \mathcal{K}eys$ denotes that a key, K, is selected uniformly at random from $\mathcal{K}eys$. The $\$$ sign on top of the $\leftarrow$ depicts that the selection was uniformly at random. $\mathcal{K}$ is a randomized algorithm that creates a random key from a finite set. Let $\mathcal{E}nc$ be the encryption function having a key, K, and a plaintext value, x, as its input parameters; thus, we have: $y = \mathcal{E}nc(K, x)$. Symmetrically, $\mathcal{D}ec$ will be the decryption function, taking y and K as input, yielding: $x = \mathcal{D}ec(K, y)$. Let $\mathcal{Z}$ be the set of binary strings of length k, $\mathcal{Z} = \{0,1\}^k$ and z be an element of $\mathcal{Z}$. After concatenating z to x, the expanded plaintext is denoted as $x' = x \| z$ where $x' \in \mathcal{X} \times \mathcal{Z}$.

9.3.1 Security against Domain Attack

As defined in Section 4.2, a domain attack is an adversary model where the attacker has knowledge about the plaintext domain and has access to the encrypted database. In our security analysis we consider that in a domain attack the attacker has precise knowledge of the domain to compute an upperbound probability distribution.

As shown earlier an OPE scheme such as in [6, 14, 15, 41, 48, 69, 71] breaks under domain attack by solely sorting the ciphertexts in the encrypted database and the plaintext values in the domain. In other words: $Adv^{\text{ROW}}_{\text{OPE}}(\mathcal{A}) = 1$. This is because OPE leaks the total order of the plaintext values. RPE was introduced in the previous chapter to ensure security against domain attack by breaking the total order.

The ROW-advantage of a domain adversary on an Prob-RPE-encrypted database is slightly different than from the Det-RPE-encrypted database. In Prob-RPE the ciphertext space is expanded to accommodate more codes for each of the values in the domain of $\mathcal{X}$. Consequently, the ROW-advantage of $\mathcal{A}$ against Prob-RPE with an expanded

Plaintext	Run 2
Benzema	
Messi_1	1
Messi_2	4
Neymar	7
Ronaldo	9

Table 9.2: Expansion Example for Table 9.1

plaintext space where its size is equal to the ciphertext space , $|\mathcal{X}'| = Y$, is shown in Lemma 16.

Lemma 16. *ROW-advantage of $\mathcal{A}$ on Prob-RPE is defined as:*

$$\begin{aligned} Adv^{ROW}_{Prob\text{-}RPE}(\mathcal{A}) &= Pr[Exp^{ROW}_{Prob\text{-}RPE}(\mathcal{A}) = 1] \\ &= \sum_{\forall\ x_i = x} Adv^{ROW}_{Det\text{-}RPE[X'][Y][U]}(\mathcal{A}) \end{aligned} \tag{9.9}$$

Proof. The ROW-advantage of a domain adversary on Det-RPE was computed by exploiting the bijective relationship between $\mathcal{Y}$ and $\mathcal{X}$. In Prob-RPE however, since there can exist more than one the ciphertext that map to the same plaintext, the bijective relationship is therefore destroyed. To prove Formula 9.9, we create a bijection for Prob-RPE by expanding the plaintext space and make it as big as the ciphertext space. We define the expanded plaintext space as $\mathcal{X}'$. Please note that $|\mathcal{X}'| = Y$. We use Table 9.2 as a clarifying example that shows how an expansion may look like for *Run 2* of Table 9.1. Let `A` be the event where `y=''Messi`$_1$`''` and `B` the event where `y=''Messi`$_2$`''`. The probability of `y=''Messi''` is the probability of $(A \vee B)$. Trivially, y cannot be both at the same time, since the mapping is bijective through the expansion assumption. This fact implies that $A \cap B = \emptyset$; therefore we can use, $pr(A \cup B) = pr(A) + pr(B)$. Thus, for every value $x \in \mathcal{X}$ we sum up the probabilities of its expanded values $x_i \in \mathcal{X}'$ using the ROW-advantage of $\mathcal{A}$ on Det-RPE. Assuming $Rank_{x_i} = i + Rank_x$ in $\mathcal{X}'$, the probability of one of the expanded plaintext to be equal to a given ciphertext to be:

$$Pr(x_i = y) = \frac{\binom{Rank_{x_i}-1}{Rank_{(y,u)}-1}\binom{X'-Rank_{x_i}}{\frac{Y}{U}-Rank_{(y,u)}}}{\binom{Y}{\frac{Y}{U}}} \tag{9.10}$$

This is the same negative hypergeometric probability distribution that has captured the ROW-advantage of A on Det-RPE except for using the expanded plaintext space, $\mathcal{X}'$ instead of $\mathcal{X}$. Please note that $|\mathcal{X}'| = |\mathcal{Y}|$. □

In Det-RPE we have shown that the ROW-advantage depends on the size of the domain (i.e., X), the number of runs (i.e., U), and the rank location of a value in the domain. This was however for the case that we do not use probabilistic encryption. In case we are using probabilistic encryption a lot of ciphertexts might correspond to the same plaintext which we have to take into account. Thus, the ROW-advantage for probabilistic encryption schemes also depends on the frequency distribution of the values. Thus we will study the sensitivity of the ROW-advantage towards these parameters in detail in this section.

Moreover, Definition 8 shows how a perfect encryption scheme can be characterized for a domain attack, namely a random adversary can do as good as:

$$Adv^{\text{FOW}}_{\text{Prob-OPE}}(\mathcal{A}^R) = Pr[Exp^{\text{FOW}}_{\text{Prob-OPE}}(\mathcal{A}^R) = 1] = \frac{freq(x)}{\sum_{v \in \mathcal{X}} freq(v)} \tag{9.11}$$

Here, $freq(x)$ corresponds to the frequency of Value x in the original database. Again, this equation is independent of DB. Note that in the frequency attack scenario, ROW-advantage denotes the probability to get *one* ciphertext right; the probability to get *all* codes of a value right is of course much smaller. Correspondingly, more frequent values have higher probability of being discovered.

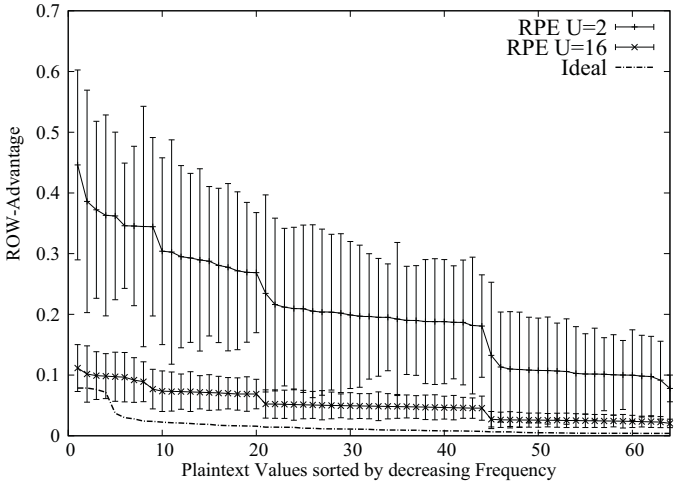


Figure 9.3: ROW-Advantage: Vary Frequency and Order of x

Figure 9.3 plots the ROW-advantage for the 64 values of a domain if these values were encrypted using an PROB-RPE scheme against a domain attack. The frequency is distorted as described in Section 7.2. In this experiment, a Zipf distribution was used for the original frequency distribution of the values (as in Figure 7.3a). The ordering of the values was random. That is, we repeated the experiment 1000 times using Formula 9.11 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. The curve shows the ROW-advantage mean from all these experiments for PROB-RPE with 2 and 16 runs. As a baseline, the ROW-advantage curve for an ideal encryption scheme has been used. We had to limit the experiments to a domain of 64 values because of floating point underflow in the computations, since the probabilities get too small for a computer to handle.

The results of Figure 9.3 are approximations. The exact formulae are complex and MatLab crashes even for small parameter settings if we try to compute the ROW-advantage exactly. The results shown in Figure 9.3 represent an *upper bound* of the ROW-advantage for Prob-RPE; that is, the results are conservative and the real privacy of Prob-RPE is better than what is shown in Figure 9.3. As shown already in RPE, privacy improves as the domain gets larger. For smaller domains, the privacy degrades, but not dramatically.

Figure 9.3 shows that ROW-advantage improves with the number of runs. With only two runs, there are scenarios in which one ciphertext for the most frequent value can be found with a high probability. This situation arises if the most frequent value is an extreme value of the domain (e.g., the lowest or highest value of the domain). With 16 runs, PROB-RPE is already close to the *ideal* privacy that can be achieved in a domain attack on probabilistic encryption.

Figure 9.3 also shows that the standard deviations in this experiment (i.e., error bars) are large. The error bars are particularly high for two runs and decrease with the 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 adding *fake* extreme values or using a modular OPE scheme in the database, as described in the previous chapter.

In summary, this section showed that Prob-RPE achieves a close to ideal (random adversary with no extra knowledge) probability distribution against attackers with domain.

9.3.2 Security against Frequency Attack

As defined in Section 4.3, a frequency attack is an adversary model where the attacker has knowledge about the domain and its underlying frequency distribution. In our security analysis we consider that in a frequency attack the attacker has precise knowledge of the domain and its underlying frequency distribution to compute an upperbound probability distribution.

In order to protect a database against a frequency attack, a probabilistic scheme such as Prob-RPE must be used. With such a scheme, the frequency distribution of the values can be *distorted* to blend the outstanding ones in the crowd and thus protect them [17, 25]. In Section 7.2, we have shown how the plaintext frequency distribution

can be distorted using a tunable probabilistic encryption scheme, such as Prob-OPE or Prob-RPE.

In general Prob-OPE follows the same pattern as Prob-OPE when it comes to security against frequency attack.

Lemma 17. *Let $y = \mathcal{E}nc_K^{\text{Prob-RPE}}(x)$, where y is a set of ciphertexts in case of a probabilistic encryption. $freq_{max}(y)$ is the ciphertext $y_1 \in y \; s.t. \; \forall \; y_i \in y, \; freq(y_1) \geq freq(y_i)$. The FOW-advantage of $\mathcal{A}$ on Prob-OPE encryption scheme is defined as his winning probability in Experiment 2:*

$$Adv^{\text{FOW}}_{\text{Prob-RPE}}(\mathcal{A}) = Pr[Exp^{\text{FOW}}_{\text{Prob-RPE}}(A) = 1] = \frac{freq(x)}{\sum_{v \in \mathcal{X} \; \wedge \; freq(v) \geq freq_{max}(y)} freq(v)} \tag{9.12}$$

9.3.3 Security against Query Log Attack

To win the query log attack, defined in Section 4.4, the adversary needs to either win the domain attack or the frequency attack with the help of the query logs. In Chapter 5 a framework has been introduced to identify leaky queries. In this section, in order to analyze the security of Prob-RPE against query log attack, two questions need to be taken care of:

1. Is Prob-RPE indistinguishable under Simple Query Attack? This property assures that rewritten queries, as discussed in Section 9.2, are not leaky, i.e., they do not weaken the Prob-RPE-encrypted data.

2. How safe is Prob-RPE against domain and frequency attack if the query logs are revealed? This property evaluates the resilience of Prob-RPE against the query log attack introduced in Section 4.4.

Since Prob-RPE is a combination of Det-RPE and Prob-OPE, trivially it is not safe against the Simple Query Attack. All the examples for query log attack on Prob-OPE (see Section 7.4.4) and Det-RPE (see Section 8.3.4) apply to Prob-RPE. Thus, we do not repeat them here. Nevertheless, in this section we elaborate on the main important results.

Corollary 13. *The adversary exploits the additional information from the leaky queries to discover the correspondence between the "independently" ordered runs to figure out the total order which was distorted thanks to RPE and assigns ciphertexts to their underlying plaintext bin to break Prob-OPE. Consequently, Prob-RPE degrades to ROPE. This degradation will of course happen, if the query rewrite method presented in Section 9.2 is used.*

Corollary 14. *The advantage of an adversary to win Experiment 3 on Prob-RPE, is the maximum probability of an adversary winning either the domain or the frequency attack on Prob-RPE. Since Prob-RPE is not safe against Simple Query Attack and degrades to ROPE as a result. The maximum probability of an adversary to win Experiment 3 on Prob-RPE is 1.*

$$Adv^{QLA}_{Prob\text{-}RPE}(\mathcal{A}) = Pr[Exp^{QLA}_{Prob\text{-}RPE}(\mathcal{A}) = 1] = 1 \tag{9.13}$$

Simulatable Queries

Like other encryption schemes discussed so far, there are queries with certain SQL-operators that cannot be simulated on Prob-RPE-encrypted database. These SQL-operators are called non-simulatable because the simulated system cannot generate them. Table 9.3 shows which SQL operators are simulatable under a Simple Query Attack for the encryption schemes presented in this thesis. NA means that the SQL operator is Not Applicable on the data if that encryption scheme is used.

In general the operators simulatable by Prob-RPE are the intersection of operators simulatable by both Prob-OPE and Det-RPE.

SQL-Operator	OPE	MOPE	Prob-OPE	Det-RPE	Prob-RPE
DISTINCT	✓	✓	NA	✓	NA
WHERE (=, !=)	✓	✓	✗	✓	✗
WHERE ($<, >$)	✓	✗	✗	✗	✗
LIKE(Prefix%)	✓	✗	✗	✗	✗
LIKE(%Suffix)	NA	NA	NA	NA	NA
IN	✓	✓	✗	✓	✗
Equi-Join	✓	✓	✓	✓	✓
Non Equi-Join	✓	✗	✓	✓	✓
TOP N	✓	✗	✗	✓	✓
ORDER BY	✓	✗	✓	✓	✓
SUM	NA	NA	NA	NA	NA
MIN/MAX	✓	✗	✓	✓	✓
GROUP BY	✓	✓	NA	✓	NA

Table 9.3: Prob-RPE: SQL-Operator Simulatability

As shown in Table 9.3, both equality predicates and range queries are not simulatable for Prob-RPE-encrypted databases. This is because each equality predicate is rewritten into a range predicate for each run. This fact implies leaking the boundaries that a plaintext value can map to in the ciphertext space and also how the correspondence of ranges across runs looks like. If enough queries are revealed then not only the total order

but also the underlying frequency distribution will be discovered and Prob-RPE with stronger security guarantees against attackers with domain and frequency knowledge downgrades to the weak OPE with no security guarantees against any of such attackers. In order to fix the problem with the non-simulatable queries, we take the same approach taken for Prob-OPE and Det-RPE scheme, which is to use the *fixed range query rewrite*.

9.4 Fixed Range Query Rewrite

There are many ways to rewrite a query. Usually a query is rewritten to get the best response time, i.e. most of the processing has been pushed down to the cloud. In this thesis, we have introduced an attack that is based on the information that an adversary can extract from the query logs. In the previous section we have shown that a Prob-RPE is not IND-SQA because it leaks information about the plaintext boundaries, and the range correspondence between the random partitions a.k.a. runs. In subsection 9.3.3 we have shown that the leakage is caused due to queries with range and equality predicates. In this section we show how we use *fixed range query rewrite* to fix the query leakage problems.

The idea of *Fixed Range Query Rewrite* is to divide the plaintext domain, $\mathcal{X}$ into j disjoint fixed-sized sub-ranges of size r, fr_i where $i = 1, ..., j$ and $|fr_i| = r$. Whenever a range or point query is asked, the smallest units that are returned are those *fixed ranges* that contain the user's results. For a probabilistic encryption scheme such as Prob-OPE or Prob-RPE it is important to consider the frequency distribution of the plaintext values when determining the fixed ranges.

As an example, consider Table 9.4 where Prob-RPE has been used for encryption. The client submits the query:

```
SELECT name FROM customer WHERE name >= 'Messi'
```

Using Prob-RPE one possible way to rewrite this query is:

```
SELECT name_run,name_enc FROM customer_enc
WHERE (name_run = 1 AND name_enc>0) OR (name_run = 2 AND name_enc > 3)
```

Since this query is revealing some information about the correspondence of ciphertexts across runs, we will use fixed range query rewrite mechanisms to fuzzify the result. Thus, the original query will be rewritten as follows to return all the fixed ranges, namely fr_1 and fr_2, that contain the result:

```
SELECT name_run,name_enc FROM customer_enc
WHERE (name_run = 1 AND name_enc > 0) OR (name_run = 2 AND name_enc > 0)
```

Table 9.4: Two fixed ranges of size 3, $r = 3$

Fixed Range	Customer Names	Run 1	Run 2	($\langle$ run, y $\rangle$)
fr_1	Benzema		3	$\langle 2, 3\rangle$
	Messi	1		$\langle 1, 1\rangle$
	Messi		5	$\langle 2, 5\rangle$
fr_2	Neymar		9	$\langle 2, 9\rangle$
	Neymar	6		$\langle 1, 6\rangle$
	Ronaldo	7		$\langle 1, 7\rangle$

In this example the whole table is returned. The result includes an additional false positive namely "Benzema" that need to be filtered out during the post-processing.

Since *Fixed Ranged Query Rewrite* returns a super-set of the result that needs to be post-filtered, this method will lead to performance loss. On the security side, the guarantees and analysis will deviate from what we have discussed earlier in Section 9.3.1.

9.4.1 Security against Query Log Attack

In case of Prob-RPE, both queries with range and equality predicates are problematic because they help to reconstruct the total order and the original frequency distribution, which is exactly what Prob-RPE is trying to hide. Hence, Fixed Ranged Query Rewrite helps again to limit the ROW-advantage under query log attack. Assuming we have a uniform code distribution within a range in all runs, Equation 9.9 will change if fixed range query rewrite has been used to rewrite the queries:

$$\begin{aligned} Adv^{\text{ROW}}_{\text{Prob-RPE-FR}}(\mathcal{A}) &= Pr[Exp^{\text{ROW}}_{\text{Prob-RPE-FR}}(\mathcal{A}) = 1] \\ &= \sum_{\forall\ x_i = x} Adv^{\text{ROW}}_{\text{Det-RPE-FR}[X'][Y][U]}(\mathcal{A}) \end{aligned} \tag{9.14}$$

The row advantage for Prob-RPE using fixed ranges is derived from Formula 9.9, but instead of using the ROW-advantage on Det-RPE, the ROW-advantage on Det-RPE-FR as discussed and shown in Section 8.4.1 is used. Informally speaking, by using the fixed range query rewrite the randomness is no more distributed throughout the domain, but is only available within the fixed range.

9.5 Experiments

This section assesses the performance overhead of Prob-RPE in the context of the 22 queries and 2 refresh functions of the TPC-H benchmark. We first compare the probabilistic encryption schemes relative to the plain database: Prob-OPE (resilient against

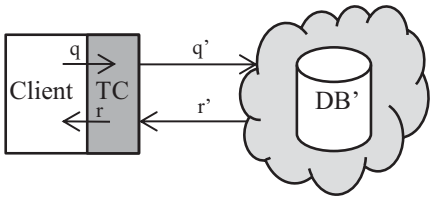


Figure 9.4: Client-Side Security

frequency attack), Prob-RPE and Prob-RPE-FR. Then we compare Det-RPE with Prob-RPE. We have performed also a benchmark with AES in CBC mode, but it lead to time-out for all the queries, this is why we omit presenting its results in this section. All RPE methods used to compare in this section have 8 runs if not determined otherwise.

The fixed ranges for the date data types are selected to be a quarter, which is three months (90 days). For varchar the fixed ranges are bucketized to alphabetically, i.e. all words beginning with "A" are in fr_1, are words beginning with "B" are in fr_2 and so on. For the numeric values the fixed ranges are selected so that each fixed range accommodates between 10 or 20 percent of the values of the domain.

9.5.1 Benchmark Environment

All experiments were conducted on two separate machines for client and server which corresponds to the architecture shown in Figure 9.4. The client and the trusted component were written in Java, ran on a machine with 24 GB of memory and communicated to the database server using JDBC. The server machine had 132 GB of memory available and hosted a MySQL 5.6 database. Both machines had 8 cores and ran a Debian-based Linux distribution.

We used a 10 GB data set (scaling factor 10) and measured end-to-end response time for all queries in separation. Queries that did not finish within 30 minutes where canceled and reported as a time-out. This is why we do not show results for Q9, which even times out for *Plain*. Metrics used in aggregate functions (e.g., volume of orders) and surrogates (e.g., order numbers) were left unencrypted while all other (sensitive) attributes, such as names, dates, etc. were encrypted. Whenever SQL operators on encrypted data were not supported, the entire data was shipped and then post-processed at the trusted component.

9.5.2 Response Time

Figure 9.5 shows the comparison of Prob-RPE with Prob-OPE as introduced in Chapter 7. Figure 9.6 shows the response time break down of the time spend on the server vs. on the client. All these encryption schemes have significantly higher response times than

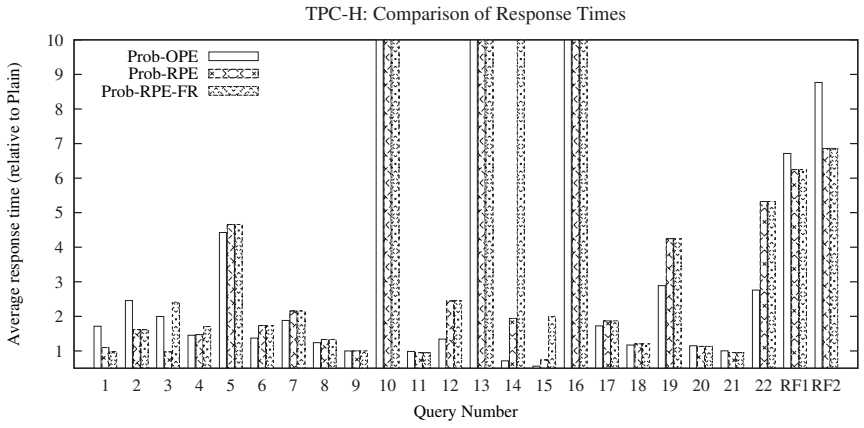


Figure 9.5: TPC-H Relative Response Times to Plain: Prob-OPE vs. Prob-RPE vs. PROB-RPE-FR

their deterministic counter-parts. This is mainly due to the frequency distortion (mentioned in section 7.2) which causes a tremendous expansion of some TPC-H dimension tables, e.g., the *Customer* and *Product* tables. We can draw the following top-level conclusions:

- *AES-CBC:* The probabilistic variant of AES times out in all the queries. AES-CBC is semantically secure, but offers no processing on encrypted data. Consequently, all the tables involved in a query need to be shipped to the trusted component and post-processed incurring a big network cost and latency.

- *Range Predicates (Q1, Q4, Q5, Q6, Q7, Q8, Q11, Q15, Q17):* Both Prob-RPE and Prob-OPE have acceptable performance when dealing with range queries, because they prevent irrelevant data (false positives) from being shipped to the client. On the other hand, Prob-RPE has far better security guarantees which makes it the more powerful option among them.

- *Group-By:* Group by clauses cannot be evaluated on probabilistically encrypted data. Nevertheless, the data that needs to be aggregated can be selected and shipped as range predicates can be executed normally if Prob-RPE or Prob-OPE is used.

- *Order By and Top-N (Q2, Q3, Q18, Q20, Q21):* For Prob-RPE, ORDER BY queries can be processed as sub-queries in parallel as described in Section 8.2.4. Consequently, Prob-RPE outperforms Prob-OPE.

- *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%"*

- *Range and Equality Predicates (Q12, Q14, Q19, Q22):* Since both Prob-RPE and Prob-OPE submit a single query, the more runs there are the more post-processing work RPE has to do, which is the reason why it is slightly outperformed by Prob-OPE. Given than PROB-RPE not only protects a database against frequency attack, but also against domain attack, the performance penalty is affordable.

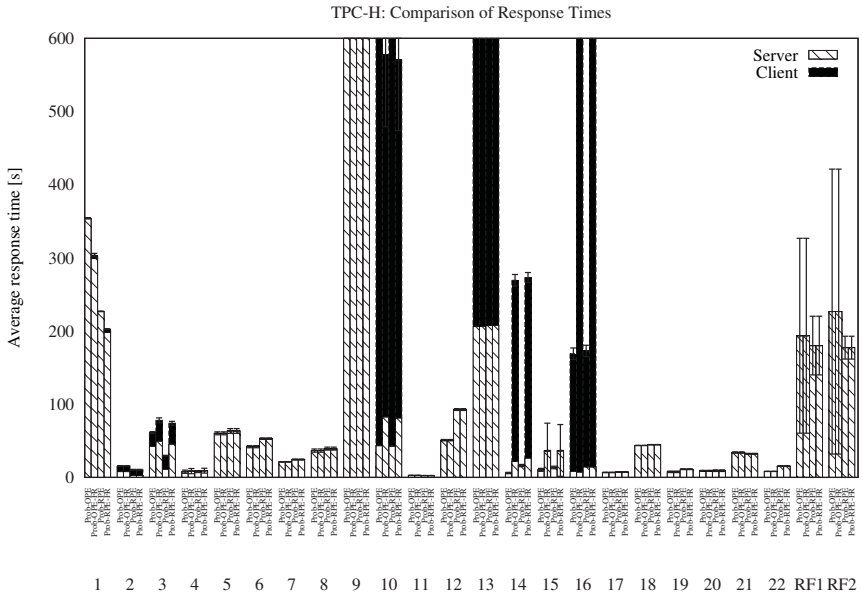


Figure 9.6: Prob-RPE Response Time: Relative to Plain

Figure 9.7 compares Det-RPE with and without fixed range query rewrite to Prob-RPE with and without fixed range query rewrite. From the results in Figure 9.5 we draw the following top-level conclusions:

- *Deterministic vs. Probabilistic RPE:* There are two queries that have far lower response time in deterministic RPE namely Q10, Q13, and Q16. As clearly shown in Figure 9.7 this difference is occurred on the client-side, i.e. a lot of time is spent to post-filter and post-aggregate the result. Also for Q13 we can see that a lot of time has been spent on the server which shows that a lot of false positives had to be transferred to the client.

- *Inclusive Ranges (Q2, Q5, Q6, Q7, Q8, Q11, Q12, Q13, Q17, Q18, Q19, Q20, Q21, Q22):* These queries contain fixed ranges. Being a superset to the fixed ranges means that no false positives need to be shipped to the trusted component and no changes are imposed to the query rewrite function and the result. Selecting the fixed ranges smartly can thus influence the performance of fixed range query

processing. As analyzed in Section 9.3.3, fixed ranges need not to be large to provide acceptable security against query log attack. The smaller the fixed ranges, the better the performance.

- *Query Parallelism (Q1,Q10):* The fixed range query rewrite breaks these queries into subqueries, providing intra-query parallelism thus achieving even better response time than the traditional query rewrite method.
- *Pre-Filtering (Q1, Q4, Q15):* These queries have false positives, but they can be filtered on the trusted component without decryption, which saves a lot of time.
- *Refresh Functions (RF1, RF2):* As fixed range query processing does not change refresh functions, the performance stays the same. However, Det-RPE is slower in RF1 which corresponds to inserts. This is because encrypting values in Prob-RPE is faster than in Det-RPE.

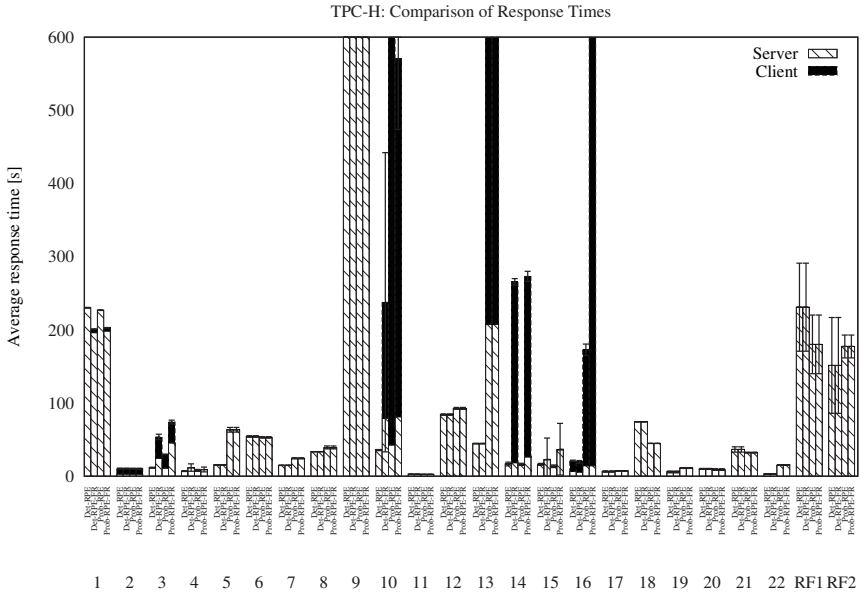


Figure 9.7: Prob-RPE Response Time: Client-Server Breakdown

9.5.3 Network Cost

Figure 9.8 shows the network cost of Prob-OPE, Prob-RPE and Prob-RPE-FR (Prob-RPE with fixed range query rewrite method) relative to the plain database. Please note that Q9 caused a time-out even for the plain database, that is why we are not interested in its performance for our comparisons.

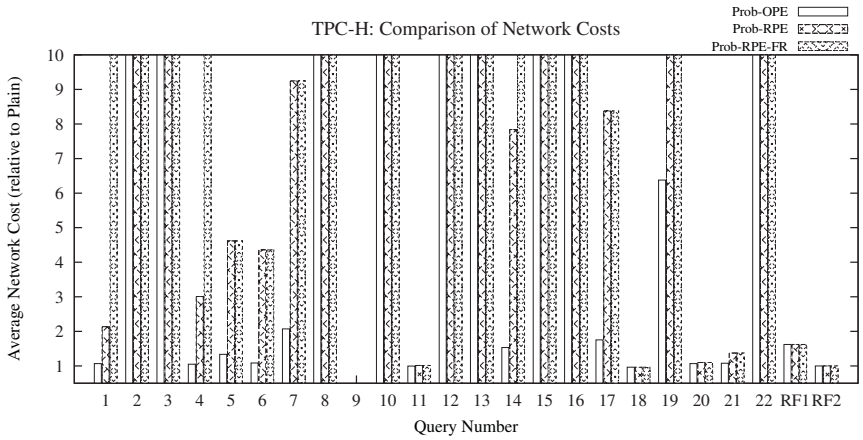


Figure 9.8: Prob-RPE Network Cost: Relative to Plain

The very important and foremost remark is that AES CBC incurs far higher shipment costs than the OPE variants. This is because to evaluate any kind of query all the data had to be shipped to the trusted component, decrypted, post-filtered and then aggregated. We omit showing the bars here since it always exceeded the y-axis boundary on the graph.

In general, Prob-RPE has higher network cost than Prob-OPE as shown in Figure 9.8 for Q1,Q5,Q6,Q7,Q14,Q17 and Q19. This is because we add the size of the query which is sent to the encrypted database to our network cost. Since RPE has to send disjunctive queries and the size of the query increases with increasing number of runs, it causes an additional network cost. Nevertheless, in terms of query results both RPE and OPE ship the same number of bytes back to the trusted component. Another less common reason is when an ORDER BY and LIMIT clause exist in a query, the LIMIT has to be replicated for each partition separately since there is no total order as described in Subsection 8.2.4.

Another important observation is that the fixed range query rewrite method, as expected, incurs much higher network cost since it is shipping fixed ranges with false positives to the client. Nevertheless, both the network cost and reponse time for Prob-RPE-FR is most of the time far better and within the acceptable range compared to AES CBC.

9.6 Conclusion

Table 9.5 summarizes the SQL-operators that are supported by the state of the art encryption schemes and the encryption schemes introduced in this thesis, namely Prob-OPE, Det-RPE, and Prob-RPE. As shown in Table 9.5 semantically secure encryption

schemes (e.g. AES CBC) do not support any query processing and thus not attractive for database applications. On the other end, weak encryption schemes such as OPE support a large number of SQL operators but shatter against domain and frequency attacks since they are deterministic and leak the total order. RPE is proven to be resilient against attackers with domain and frequency knowledge, and at the same time provides same functionality and performance as OPE.

SQL-Operator	AES-ECB	OPE	Prob-OPE	Det-RPE	Prob-RPE	Paillier [44]	AES-CBC
DISTINCT	✓	✓	✗	✓	✗	✗	✗
WHERE (=, !=)	✓	✓	✓	✓	✓	✗	✗
WHERE (<, >)	✗	✓	✓	✓	✓	✗	✗
LIKE(Prefix%)	✗	✓	✓	✓	✓	✗	✗
LIKE(%Suffix)	✗	✗	✗	✗	✗	✗	✗
IN	✓	✓	✓	✓	✓	✗	✗
Equi-Join	✓	✓	✓	✓	✓	✗	✗
Non Equi-Join	✗	✓	✓	✓	✓	✗	✗
TOP N	✗	✓	✓	✓	✓	✗	✗
ORDER BY	✗	✓	✓	✓	✓	✗	✗
SUM	✗	✗	✗	✗	✗	✓	✗
MIN/MAX	✗	✓	✓	✓	✓	✗	✗
GROUP BY	✓	✓	✗	✓	✗	✗	✗

Table 9.5: Prob-RPE: Supported SQL Operators

Table 9.6 summarizes the result of the security analysis done in this chapter on Prob-RPE. Each row represents an OPE variant, and each column presents an attacker model introduced in Chapter 4. Each cell shows how low an encryption scheme downgrades under a given attack. By plain we mean an unencrypted database. The FR postfix indicates the case when *fixed range query rewrite* mechanism is used to rewrite queries on the encrypted database.

Please note that each encryption variant in Table 9.6 has a tunable probability distribution under each attacker model. In general, the encryption schemes introduced in this thesis have a performance/security knob that can be adjusted according to the performance/security requirements of the system.

OPE variant	Domain Attack	Frequency Attack (uniform)	Frequency Attack (skewed)	Query Log Attack
OPE	Plain	OPE	Plain	Plain
Prob-OPE	Prob-OPE	Prob-OPE	Prob-OPE	Plain
Prob-OPE-FR	Prob-OPE	Prob-OPE	Prob-OPE	Prob-OPE
Det-RPE	Det-RPE	Det-RPE	Plain	Plain
Det-RPE-FR	Det-RPE	Det-RPE	Plain	Det-RPE
Prob-RPE	Prob-RPE	Prob-RPE	Prob-RPE	Plain
Prob-RPE-FR	Prob-RPE	Prob-RPE	Prob-RPE	Prob-RPE

Table 9.6: Prob-RPE: Security Downgrade

10 Conclusion

This thesis tackled the problem of encrypting databases in the cloud towards achieving the high performance/high security goal. Clearly, encrypting data introduces query processing overhead which makes classical encryption schemes (e.g., AES or RSA) non-performant when it comes to database applications. In this work we revisited state of the art encryption schemes and discussed their performance/security trade offs for database applications. Among these schemes, order-preserving encryption has received a lot of attention from the database community, since it allows the database to process a large classes of queries without the need to decrypt the data, thereby fulfilling the (high) performance requirements of a system (in terms of low response time and low network cost). Unfortunately, the security of order-preserving encryption is questionable. OPE schemes have been treated cryptographically to some extent, but there are adversary models that are specific to database applications and have not been considered before.

In this dissertation, we introduced three use-case driven adversary models from database applications, namely *Domain Attack*, *Frequency Attack* and *Query Log Attack*. In all these models the fundamental assumption about the adversary is that he is honest-but-curious and has full access to the device (or devices) where the database server resides. The attacker scenarios are modeled as Experiments (or Games) which an adversary needs to win. In domain attack the attacker has access to the encrypted data and the plaintext domain. We introduced *Rank One-Wayness* as a measure to compute the success rate of an adversary attempting a domain attack. In frequency attack, the attacker has access to encrypted data and the frequency of the plaintext domain. We introduced *Frequency One-Wayness* as a measure to compute the success rate of an adversary attempting a frequency attack. To this end, a number of encryption techniques have

been developed and studied for database encryption. Unfortunately, they have ignored an important threat in the database application realm, called *query log attack*, i.e., the inference of secrets by observing the (rewritten) queries submitted to the database. This thesis showed that queries can impose danger on their underlying encryption schemes if they are rewritten in a certain way.

For the first time, this dissertation formally models the information leakage of the database queries. A new security notion is defined to identify the leaky queries, called *Indistinguishability under Query Log Attack*. We introduced various attacker scenarios that make use of this security notion, namely *Simple Query Attack*, *Known-Query Attack* and *Chosen-Query Attack*. In all these scenarios the adversary is required to distinguish between the queries originated from a real system and the queries originated from a simulated system. We provided a new framework to analyze the safety of queries on any given encryption scheme. As a case study we have chosen the order preserving encryption variants because of their importance and efficiency in query processing on encrypted data. We then presented a new query rewrite mechanism called the *Fixed Range Query Rewrite* to make the stronger OPE variants, presented in this thesis, robust against query log attacks. In the end, we showed that the new query rewrite mechanism preserves response time but adds the network overhead on the read-intensive TPC-H queries.

After introducing new adversary scenarios, we analyzed the security of order preserving encryption schemes under such attacks. A known weakness of order-preserving encryption is its vulnerability against attackers with domain knowledge since it leaks the total order. To amend this weakness, we introduced the concept of *Randomly Partitioned Encryption (RPE)* which distorts the total order by randomly partitioning the domain into sorted runs. Another weakness of order-preserving encryption is being deterministic, thus in this work we introduced *Probabilistic Order-Preserving Encryption (Prob-OPE)* to break the original frequency distribution of the plaintext values. Additionally, we showed that by composing RPE and Prob-OPE, a stronger encryption scheme can be achieved called *Probabilistic RPE*. We analyzed the resilience of these encryption schemes against the introduced attacker models. In the end, we showed that these new encryption schemes could reinforce the security of OPE schemes against adversaries with various domain knowledge while keeping up with the OPE schemes in terms of database performance. We have conducted extensive performance benchmarks using the TPC-H queries and could show that RPE makes it indeed possible to achieve a higher level of privacy compared to the state of the art while preserving the low performance overhead of OPE schemes.

While the key idea of RPE is simple, this approach had a number of surprising consequences:

- *Security:* Even though a weak encryption scheme is used to encrypt each *Run*, RPE could provably achieve high levels of security. Intuitively, the reason is that

there are exponentially many ways to re-assemble the original column from a set of random partitions [28]. Furthermore, RPE can be configured to be probabilistic even if the underlying weak encryption scheme is deterministic. Specifically, we show that RPE is able to address two common attack scenarios, called *domain* and *frequency attack* that weak encryption schemes such as order-preserving encryption (e.g., [6,14,15,48]) and deterministic encryption schemes (e.g., deterministic RPE) are not able to address.

- *Performance:* Modern database systems process queries in a divide & conquer way, thereby partitioning the data and processing it partition by partition. As a result, the additional layer of partitioning provided by the RPE *Runs* did not hurt performance as long as each *Run* can be processed efficiently. RPE achieved this by using a weak encryption scheme such as order-preserving encryption for each *Run*.

- *Functionality:* In terms of query functionality, RPE was as good as its underlying weak encryption scheme. If an order-preserving encryption scheme was used, then RPE could effectively process many classes of queries including queries with range predicates, wild cards (i.e., *LIKE* predicates), equi-joins, group-by / aggregation and rank queries.

- *Dynamism:* One important advantage of RPE was that it supported updates even in situations in which a traditional weak encryption scheme would not; for instance, a probabilistic order-preserving encryption scheme might exhaust the available ciphertexts whereas RPE never hit such a dead-end because it could add new *Runs* dynamically as it runs out of space. Furthermore, some of the optimizations used in systems like Monomi preclude updates since it uses the state of the art encryption schemes whereas any kind of update could be applied to data that was encrypted using RPE.

In summary, RPE had provably higher security and functionality than any traditional weak encryption scheme at the same performance. On the other hand, RPE had much better performance and functionality than any traditional strong encryption scheme. RPE could be configured to have strong encryption properties, but in this case performance and functionality deteriorated in the same way as they do for other strong encryption schemes. In fact, weak and strong encryption could be considered as special cases of RPE: weak encryption corresponded to RPE with one *Run*; strong encryption corresponded to RPE with an infinite number of *Runs*. RPE covered the space in between, thereby providing a *tunable security/performance parameter* for many practical applications.

Another crucial advantage of RPE was that its implementation required no changes to the database system. That is, RPE could be fully implemented as part of a trusted

component *on top of* the database system. This made RPE a good candidate for adoption of users of public clouds in which users do not trust the provider or its employees. Furthermore, RPE could be easily integrated into systems like TrustedDB, CryptDB, Cipherbase, and Monomi thereby replacing the weak encryption schemes that these systems are using with RPE. As a result, our work on RPE was orthogonal to the techniques developed as part of these projects and complements nicely the recent advancements on processing encrypted data in the cloud.

10.1 Future Work

Given the presented results and conclusions, this thesis opens new possible research areas to follow and challenges to address. Some potential topics and questions are as follows:

- **Reducing Encryption/Decryption Cost of OPE Algorithms:** RPE and Prob-OPE use ROPE [14] as a buidling block. ROPE uses a pseudo-random lazy sampling algorithm to map range gaps to domain gaps in a recursive, binary search manner to determine the image of an input. While most of the time OPE and its variants outperform AES because of less data that needs to be shipped and post-processed, still when it comes to decryption speed, AES outperforms OPE. AES algorithms are widely available and heavily optimized whereas OPE algorithms are still very immature and need a lot of improvement.

- **Different Underlying OPE Encryption Algorithms:** RPE and Prob-OPE use ROPE [14] as a buidling block. There are several improved versions of ROPE available that claim to have better indistinguishability guarantees such as [15, 41, 48, 69]. It is interesting to see which one of them can deliver better security and performance at the same time by implementing them and running benchmarks on them as well.

- **Modern Hardware:** RPE supports parallelism by creating randomized and independent sorted runs. As discussed in this thesis, these sorted runs not only provide better security by breaking the total order relationship among ciphertexts but also for some queries reduce the response time. To query these independent runs, the queries are divided into disjunctive subqueries that exploit a lot of features on modern hardware. As shown in Section 8.5.2 some SQL operators become more performant with increasing number of runs. Therefore, it would be interesting to observe how modern hareware, such as FPGA devices that additionally can be tamper-proof, influence the performance requirements of the system when instead of traditional OPE, RPE is used.

- **New Access Patterns:** So far we have considered only relational databases and SQL operators, therefore we have developed encryption schemes that are optimized for such workloads. In the cloud computing and big data era a new wave of databases have been introduced to support big data storage and processing. It would be interesting to see how the findings of this thesis would apply to these new models.

- **Extending the Encryption Toolkit:** Coming from an applied background, there is a great potential in applied cryptography and extending the encryption toolbox, especially in this new information era where data is generated in a huge scale and yet the current available tools to ensure privacy and security do not keep up with the growth of data and variety of applications. The applied community is always looking forward to new and innovative encryption schemes that can provide acceptable privacy and preserve the performance requirements of their applications.

- **Order-preserving Encryption on Infinite Domains:** Supporting dynamic Databases, RPE is a possible method to realize order-preserving encryption on infinite domains. This hypothesis however needs more theoretical treatment which could be a challenging topic for a future work.

List of Figures

List of Tables

Bibliography

[1] Dropbox. web-based file hosting service. www.dropbox.com.

[2] Google cloud sql. `https://developers.google.com/cloud-sql/`.

[3] Microsoft azure. `http://azure.microsoft.com/`.

[4] Secretsync. http://getsecretsync.com/ss/.

[5] Is homomorphic encryption the holy grail for database queries on encrypted data? Technical Report 2012-01, Department of Computer Science, University of California, Santa Barbara, 2012.

[6] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Order preserving encryption for numeric data. In *SIGMOD*, 2004.

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

[8] G. Amanatidis, A. Boldyreva, and A. O'Neill. Provably-secure schemes for basic query support in outsourced databases. In *IFIP WG 11.3*, 2007.

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

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

[11] S. Bajaj and R. Sion. Correctdb: Sql engine with practical query authentication. pages 529–540, 2013.

[12] M. Bellare, A. Boldyreva, and A. O'Neill. Deterministic and efficiently searchable encryption. In *CRYPTO*, 2007.

[13] B. Berger and P. W. Shor. Approximation alogorithms for the maximum acyclic subgraph problem. SODA, 1990.

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

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

[16] M. Carey and D. Kossmann. On saying "enough already!" in SQL. In *SIGMOD*, 1997.

[17] S. Chawla, C. Dwork, F. Mcsherry, A. Smith, and L. Stockmeyer. Toward privacy in public databases. In *TCC*, 2005.

[18] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. *ACM J.*, 1998.

[19] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. *J. ACM*, pages 965–981, 1998.

[20] 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.

[21] E. Costante, J. den Hartog, M. Petkovi, S. Etalle, and M. Pechenizkiy. Hunting the unknown. In *Data and Applications Security and Privacy XXVIII*, pages 243–259. Springer, 2014.

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

[23] T. El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *CRYPTO*, 1985.

[24] Y. Elovici, R. Waisenberg, E. Shmueli, and E. Gudes. A structure preserving database encryption scheme. In *Secure Data Management*, pages 28–40, 2004.

[25] R. Gavison. Privacy and the limits of the law. *Computers, Ethics, and Social Values*, 1995.

[26] T. Ge and S. B. Zdonik. Fast, secure encryption for indexing in a column-oriented DBMS. In *ICDE*, 2007.

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

[28] V. Guruswami, R. Manokaran, and P. Raghavendra. Beating the random ordering is hard: Inapproximability of maximum acyclic subgraph. In *FOCS*, 2008.

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

[30] H. Hacigümüs, S. Mehrotra, and B. R. Iyer. Providing database as a service. In *ICDE*, 2002.

[31] S. Hildenbrand, D. Kossmann, T. Sanamrad, C. Binnig, F. Faerber, and J. Woehler. Query processing on encrypted data in the cloud. Technical Report 735, Department of Computer Science, Swiss federal Institute of Technology Zurich, 2011.

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

[33] H. Hu, J. Xu, C. Ren, and B. Choi. Processing private queries over untrusted data cloud through privacy homomorphism. In *ICDE*, pages 601–612, 2011.

[34] R. Jones, R. Kumar, B. Pang, and A. Tomkins. "i know what you did last summer": Query logs and user privacy. In *Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management*, CIKM, 2007.

[35] H. Kadhem, T. Amagasa, and H. Kitagawa. Mv-opes: Multivalued-order preserving encryption scheme: A novel scheme for encrypting integer value to many different values. *IEICE Transactions*, pages 2520–2533, 2010.

[36] H. Kadhem, T. Amagasa, and H. Kitagawa. A secure and efficient order preserving encryption scheme for relational databases. In *KMIS*, pages 25–35, 2010.

[37] A. Kamra, E. Terzi, and E. Bertino. Detecting anomalous access patterns in relational databases. *The VLDB Journal*, pages 1063–1077, 2008.

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

[39] N. Li, T. Li, and S. Venkatasubramanian. t-closeness: Privacy beyond k-anonymity and l-diversity. In *ICDE*, 2007.

[40] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam. l-diversity: Privacy beyond k-anonymity. In *ICDE*, 2006.

[41] T. Malkin, I. Teranishi, and M. Yung. Order-preserving encryption secure beyond one-wayness. *IACR Cryptology ePrint Archive*, page 409, 2013.

[42] S. Mathew, M. Petropoulos, H. Ngo, and S. Upadhyaya. A data-centric approach to insider attack detection in database systems. In *Recent Advances in Intrusion Detection*, pages 382–401. Springer, 2010.

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

[44] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, 1999.

[45] H. Pang, A. Jain, K. Ramamritham, and K.-L. Tan. Verifying completeness of relational query results in data publishing. SIGMOD, pages 407–418, 2005.

[46] B. Poblete, M. Spiliopoulou, and R. Baeza-Yates. Website privacy preservation for query log publishing. In *Privacy, Security, and Trust in KDD*, pages 80–96. Springer, 2008.

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

[48] R. A. Popa, F. H. Li, and N. Zeldovich. An ideal-security protocol for order-preserving encoding. In *IEEE Symposium on Security and Privacy*, pages 463–477, 2013.

[49] R. Riley, C. Clifton, and Q. Malluhi. Maintaining database anonymity in the presence of queries. In *Security and Trust Management*, pages 33–48. Springer, 2013.

[50] R. Rivest, L. Adleman, and M. Dertouzos. On data banks and privacy homomorphisms. 1978.

[51] T. Sanamrad, L. Braun, K. Donald, and R. Venkatesan. Randomly partitioned encryption for cloud databases. In *DBSec*, 2014.

[52] T. Sanamrad, L. Braun, and D. Kossmann. An efficient and randomized encryption scheme for dynamic databases in the cloud. Submitted to VLDB 2015.

[53] T. Sanamrad, L. Braun, and D. Kossmann. How to identify the leaky queries on encrypted databases. Submitted to SIGMOD 2015.

[54] T. Sanamrad, L. Braun, D. Kossmann, and V. Ramarathnam. POP: A new encryption scheme for dynamic databases. Technical Report 782, Department of Computer Science, Swiss federal Institute of Technology Zurich, 2013.

[55] T. Sanamrad, L. Braun, D. Widmer, D. Kossmann, and P. Nick. My private google calendar and gmail. *IEEE Data Eng. Bull.*, 2012.

[56] T. Sanamrad and D. Kossmann. Query log attack on encrypted databases. In *Secure Data Management*, 2013.

[57] L. Seungmin, P. Tae-Jun, L. Donghyeok, N. Taekyong, and K. Sehun. Chaotic order preserving encryption for efficient and secure queries on databases. *IEICE transactions on information and systems*, 92(11):2207–2217, 2009.

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

[59] R. Sion. Towards secure data outsourcing. In *Handbook of Database Security*. 2008.

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

[61] D. Suciu and N. Dalvi. Foundations of probabilistic answers to queries. In *SIGMOD*, 2005.

[62] L. Sweeney. k-anonymity: a model for protecting privacy. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 2002.

[63] J. Szlichta, P. Godfrey, and J. Gryz. Fundamentals of order dependencies. *PVLDB*, 2012.

[64] M. Taylor. Nsa revelations 'changing how businesses store sensitive data', March 2014.

[65] P. Tendick and N. Matloff. A modified random perturbation method for database security. *ACM Trans. Database Syst.*, 1994.

[66] S. Tu, M. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. 2013.

[67] W. Voorsluys, J. Broberg, and R. Buyya. *Introduction to Cloud Computing*. Wiley Press, 2011.

[68] W. H. Wang and L. V. S. Lakshmanan. Efficient secure query evaluation over encrypted xml databases. In *VLDB*, pages 127–138, 2006.

[69] S. Wozniak, M. Rossberg, S. Grau, A. Alshawish, and G. Schaefer. Beyond the ideal object: towards disclosure-resilient order-preserving encryption schemes. In *CCSW*, pages 89–100, 2013.

[70] L. Xiao and I.-L. Yen. A note for the ideal order-preserving encryption object and generalized order-preserving encryption. *IACR Cryptology ePrint Archive*, 2012.

[71] L. Xiao, I.-L. Yen, and D. Huynh. A note for the ideal order-preserving encryption object and generalized order-preserving encryption. *IACR Cryptology ePrint Archive*, 2012:350, 2012.

[72] Z. Yang, S. Zhong, and R. N. Wright. Privacy-preserving queries on encrypted data. In *Computer Security–ESORICS 2006*, pages 479–495. Springer, 2006.

[73] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-vm side channels and their use to extract private keys. In *CCS*, 2012.