NTNU – Trondheim
Norwegian University of
Science and Technology

Exploring CryptDB:
A Practical HE Scheme for SQL Queries

Eirik Klevstad

Submission date:	November 2015
Responsible professor:	Colin Boyd, ITEM
Supervisor:	Christopher Carr, ITEM

Norwegian University of Science and Technology
Department of Telematics

Abstract

This is an awesome abstract.

Preface

This is a silly preface.

Contents

Appendices

List of Figures

List of Tables

List of Acronyms

AES Advanced Encryption Standard.

CBC Cipher-Block-Chaining.

CDF Cummulative Distribution Function.

DET Deterministic.

FHE Fully Homomorphic Encryption.

HE Homomorphic Encryption.

HOM Homomorphic Encryption Onion.

JOIN-ADJ Adjustable Join.

LSAP Linear Sum Assignment Problem.

OPE Order-Preserving Encryption.

PHE Practical Homomorphic Encryption.

PKE Public Key Encryption.

RND Random.

RSA Rivest - Shamir - Adleman.

SQL Structured Query Language.

UDFs User-Defined Functions.

VM Virtual Machine.

VMM Virtual Machine Monitor.

Chapter 1 Introduction

1.1 Motivation

Cloud services are becoming larger and more complex. Users want their content available wherever they are, forcing applications to store content in the cloud. Companies such as Apple [1], Microsoft [3] and other corporations are looking into health information and how your personal information can be integrated into their services. Medical research facilities stores tremendous amounts of personal data, and are currently looking into how to share their research material across facilities and borders. Along with these types of sensitive data, follows great responsibility and security measures. When developing applications and systems, data security and confidentiality are important topics.

As a developer, you are left with two choices. Option one is to build our own server farm or data center on a secure site, which is a rather expensive solution with costs related to both hardware, maintenance, electricity and rent. Option two is to out-source the storage to a cloud provider, where the developer stumbles into another problem. How can we guarantee that the data is stored securely, given that we cannot necessary trust the provider.

The first solution that comes to our mind is that the user could encrypt its data with a strong block cipher, and then decrypt the result at the client side. Sure, but this does not really solve anything. Problems arise when the application needs to perform operations that are too heavy for an ordinary client's machine. What if there existed a database system that could solve these problems for us? A system where the data is safely stored in the cloud without the possibility of having database administrators snooping around, or adversaries able to extract any information in the (un)likely case of a database breach. A system that could perform all kinds of operations on our data and send the encrypted result back, without leaking any sort of information about the query, the result, the data itself or any intermediate values. Homomorphic encryption schemes might be our rescue.

In short terms, homomorphic encryption is a cryptographic property describing the ability to perform certain operations on encrypted data (ciphertexts) without decrypting it first. Fully homomorphic encryption is the enhanced version where the encryption scheme is capable of performing all efficient functions [10]. While still in research mode, fully homomorphic encryption schemes' biggest challenge is efficiency. As for practicality, fully homomorphic encryption schemes still have a long way to go [13].

Something more specific about CryptDB and why it is interesting. Its key ideas.

Maybe we can achieve FHE in another way?

For now: What is practically possible?

1.2 Problem

Describe the problem/objective.
The aim of the project. Short explanation of results

1.3 Scope

Any differences from the project description goes here. Redefine. Explain.

1.4 Related Work

To be continued.

Chapter 2
Background

This chapter will cover the cryptographic background of the necessary components in order to understand CryptDB, along with some of the basic cryptographic principals.

...More. Symmetric/Asymmetric/ has no references. Is this trivial enough to go without?

2.1 Databases and Database Systems

Databases are central building blocks in most computer systems, allowing data to be stored, shared, and read by users. When we want to save our data, we need a place to store it. Formally, databases can be described as a set of related data that us organized in such a way that data can easily be accessed, managed, and updated. As the amount of data stored does not decrease over time, database systems experience an exponential growth and are becoming more and more complex.

Databases are often constructed of one or more *tables* which consists of multiple *fields* or columns as seen in Figure 2.1. These columns are usually defined to store values of some predetermined *data type*, for example integers, dates, strings or decimals.

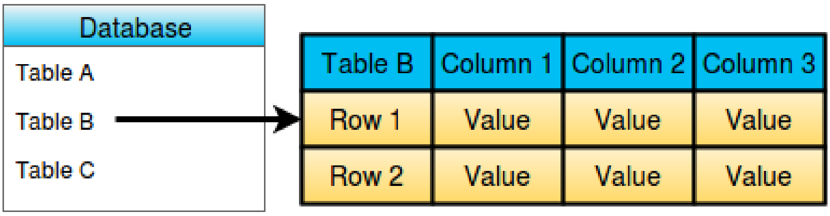


Table B	Column 1	Column 2	Column 3
Row 1	Value	Value	Value
Row 2	Value	Value	Value

Figure 2.1: A table within a database consists of multiple columns, when inserting values into a table, a new row will be created storing the data

Database Management Systems (DBMS) are software used to enable clients and applications to interact with a database. These systems are often classified based on the database model they support, which is a logical structure describing the database as well as determining the permitted ways of storing information. The, possibly, most popular model is the relational database, mostly because of it being the main model for large DBMS such as MySQL, PostgreSQL, Oracle and Microsoft SQL Server.

Now we got a DBMS with a database storing some tables with our data. How do we put data in our database, and how do we extract data from it? In order for databases do anything useful other than just storing the data, they need a query language. A query language is used by DBMS to insert and extract data from databases. Structured Query Language (SQL) is the standard language for performing queries on relational databases, which enables system to insert, update and delete data, as well as extracting

2.1.1 Database Security

Databases are central building blocks in most computer systems, allowing data to be stored, shared, and read by users. As the amount of data stored does not decrease over time, database systems experience an exponential growth. There are companies that stores banking information, health records, and other sensitive information, which makes them a target for criminals and hackers. In order to cope with such attacks, systems that are relying on some sort of database structure, are in need of defence mechanisms.

Most database systems are stored behind different network security measures such as firewalls and intrusion detection systems. A database firewall will, for example, monitor all traffic to and from a database system in order to detect situations that deflects from the predetermined database policy. Such measures does however only provide security for the outside of the system, but a system is in need of security on the inside as well. Most database systems provide different security measures for handling situations that can occur on the inside.

Access control is used for granting different privileges for a user or user groups to a certain object in the database, such as a table, view or procedure. Authentication is used to ensure that only trusted entities are able to reach the database, and to grant the correct privileges based on the identity of the user. For applications where certain user groups are handling sensitive information, a person in charge of the system may have the need to monitor who accessed a given table at a certain time in case of information leaks and similar circumstances. Auditing is a database security technique which does not directly prevent security protocols from being broken, but allows the system administrator to backtrack and discover breaches in the security.

Another vital part of keeping a database secure, is of course the use of database encryption. Both symmetric and asymmetric encryption is possible to perform on database systems, and there exists different ways of layering the encryption based on the application.

Hmm. Not sure about how good this section is. What to add, remove, etc?

2.2 Symmetric Encryption

Symmetric key encryption is, perhaps, the most intuitive type of encryption, where the sender encrypts the data with a secret key that the sender and receiver has agreed upon in advance. When receiving data, the receiver uses the pre-shared secret key in order to decrypt the data. In a more formal matter, symmetric key encryption is usually used either with a block cipher(which encrypts messages in chunks of data) or a stream cipher (which encrypts data bit-by-bit). The scheme usually consist of three algorithms, $KeyGen$, Enc and Dec.

$KeyGen$ is the algorithm that generates the secret key k_s used for encryption and decryption, Enc_k encrypts the data with the secret key, and Dec_k decrypts it using the same secret key. One of the major drawbacks with symmetric key cryptography is that the secret key k_s needs to be shared between the parties that are communicating. If an adversary obtains the secret key, say that one of the parties stored it on a piece of paper that was misplaced and lost, the whole communication channel would be compromised as the adversary could easily decrypt the data.

In database systems, symmetric encryption is often used to store data by encrypting it under some secret key k_s that is kept in the database. Given that the information in the database is stored using the same key, this approach provides little security against unauthorized users that gain access to the secret key. On the plus side, the process of encrypting data using a symmetric encryption scheme is usually fast compared to other approaches that uses multiple keys.

2.3 Asymmetric Encryption

In contrast to symmetric key encryption, asymmetric key encryption does not depend on a pre-shared secret key. Asymmetric key encryption, or Public-Key Encryption, is based upon the fact that some mathematical problems are considered *hard*, such as the integer factorization, discrete logarithms and elliptic curves. By computing a key-pair consisting of a public key and a private key, two users are able to exchange keys over a public channel without worrying about their secret keys being compromised.

The public key is used for encrypting data sent to the user, and the private key is used to decrypt received data that is encrypted with said public key. Two of the most recognized public-key cryptosystems are Rivest - Shamir - Adleman (RSA), which relies of the integer factorization problem, and ElGamal, which relies on the discrete logarithms problem. In addition to secure communication between multiple parties, public key cryptography is also applied to create digital signatures, which provides authentication and data integrity.

For applications using databases that are storing information for different users, asymmetric encryption would be a natural encryption scheme. Information sent from one user to another could be encrypted under the receivers public key and vice-versa, and decrypted by applying their secret keys.

2.4 Homomorphic Encryption

We love to describe encryption as safes where we store our data, then secures it with one or more locks, and hide the secret key. Without the secret key, the data is securely stored inside the safe. Whenever we need our data, we take the hidden key out from its hideout and opens all the locks of the safe, where the data is as intact as we left it. The, perhaps, holiest of all the holy grails in cryptography is called *homomorphic encryption.* This is a special case of encryption where operations on the encrypted data are possible without decrypting it first, or in the perspective of our locked safe: Modify the data on the inside of the safe without ever unlocking it. Figure 2.2 illustrates the main concept of homomorphic encryption used in a cloud service scenario.

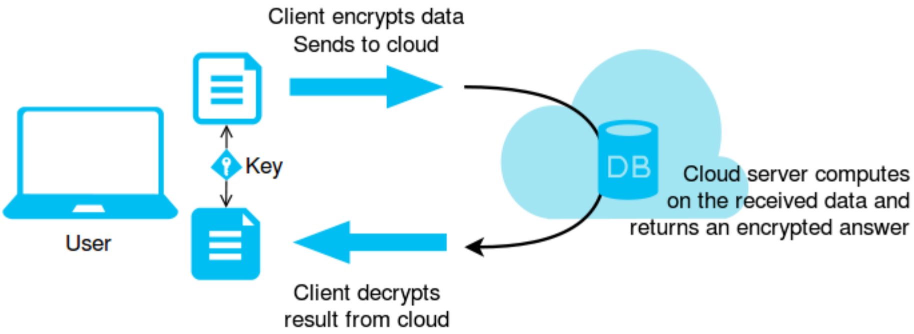

Client encrypts data
Sends to cloud
Key
User
DB
Cloud server computes on the received data and returns an encrypted answer
Client decrypts result from cloud

Figure 2.2: Illustration of a homomorphic system where data is encrypted at the client, sent to the server which performs computation on the encrypted data and returns an encrypted result which is decrypted by the client.

Regular Public Key Encryption (PKE) schemes consists of three algorithms, namely a key generation algorithm (`KeyGen`), an encryption algorithm (`Enc`) and a decryption algorithm (`Dec`). Homomorphic Encryption (HE) schemes adds another algorithm to the toolbox, an evaluation algorithm (`Eval`). Assume a well formed public key pk, a boolean circuit C and a set of ciphertexts $c_1, ..., c_l$ which are encryptions of $m_1, ..., m_l$ respectively. Then `Eval` outputs a ciphertext c which encrypted under pk such that $Dec_{sk}(c) = C(m_1, ..., m_l)$ for some secret key sk [8].

For Practical Homomorphic Encryption (PHE) schemes, *Eval* will be associated to a set of permitted functions f (or circuits). These are functions that the algorithm can handle, and which guarantee a meaningful result when executed. These functions can be expressed as boolean circuits consisting of logical gates such as AND, OR and NOT. Gentry [12] presented a homomorphic encryption scheme consisting of three functions; addition (Add_ϵ), subtraction (Sub_ϵ) and multiplication ($Mult_\epsilon$). When performing homomorphic operations using functions from f, an N-bit noise is generated and added to the encrypted ciphertext, making the relation between the encrypted result and its corresponding plaintext weaker. By performing multiple operations on the ciphertexts, the noise grows larger. Problems arise when the noisy part gets too large, as the decryption algorithm might not be able to decrypt in a reliable way in order to obtain the correct result.

2.5 Fully Homomorphic Encryption

Fully Homomorphic Encryption (FHE), which has no restrictions to what types of operations that can be performed on the encrypted data, was first suggested in 1978 by Rivest, Adleman, and Dertouzos [20]. At this point in time, researchers did not have any secure scheme for using these ideas. More importantly, there were not many use cases driving the need of such schemes. It has therefore been a slightly displaced and forgotten grail until 2009, when Gentry presented the first FHE scheme based on lattices [11], and a year later another scheme using a *bootstrappable* approach [12]. Nearly all modern FHE schemes are based on this bootstrappable concept using Gentry's blueprints.

As mentioned in the section above, a homomorphic operation creates noise when executed. Multiple operations adds more noise which may change the decrypted result beyond the recognizable. But what if we had a scheme that was able to reduce the noise generated by such operations? Boostrapping involves encrypting the secret key sk under its corresponding public key pk and added to a modified encryption algorithm, $Recrypt(pk_{i+1}, D_\epsilon, \overline{sk_i}, c_i)$. This basically enables the algorithm to decrypt the ciphertext c_i taken as input internally while being encrypted under the new public key pk_{i+1}. *Recrypt* returns a *freshly* created ciphertext c_{i+1} which is less noise than the original c_1 ciphertext to continue to perform homomorphic operations on.

Equation 2.1 encrypts the message m under the public key pk_1 as most asymmetric encryption schemes does, in order to obtain the ciphertext c_1. So far, nothing we have not seen before. Now, the secret key sk_1 from the public-key-pair (pk_1, sk_1) is encrypted using a new, fresh public-key pair (pk_2, sk_2) as seen in Equation 2.2. By doing so, the algorithm enables *Eval* to later on decrypt c_1 using sk_1 while under the encryption of pk_2.

$$c_1 = Enc(pk_1, m) \tag{2.1}$$

$$\overline{sk_1} = Enc(pk_2, sk_1) \tag{2.2}$$

Bootstrapping works recursively, so for the algorithm to be able to decrypt the c_1 in the next iteration, it needs to be encrypted under the new public key pk_2 as seen in Equation 2.3. In Equation 2.4, a new and fresh ciphertext is produced by the *Eval* algorithm, which uses pk_2 to encrypt the noisy ciphertext c_1, and D to decrypt it c_1 using its corresponding sk_1 inside *Eval*. By doing this, the noise of the ciphertext is reduced, while still being encrypted - Now under the public key pk_2. Figure 2.3 shows the process of the *Recrypt* algorithm.

$$\overline{c_1} = Enc(pk_2, c_1) \tag{2.3}$$

$$c_1' = Eval(pk_2, D_\epsilon, \overline{sk_1}, \overline{c_1}) \tag{2.4}$$

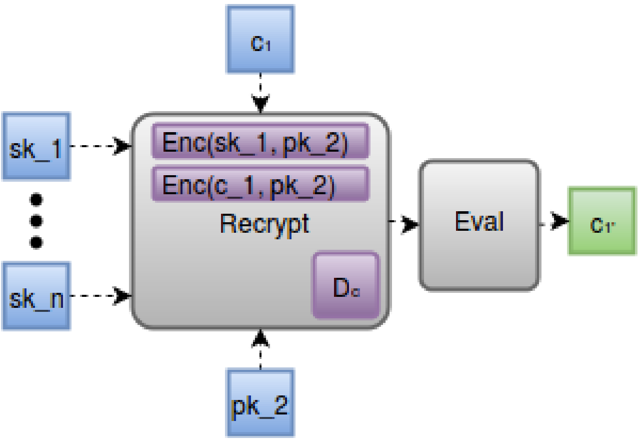


Figure 2.3: Illustration of the Recrypt algorithm, which refreshes the ciphertext c_1 encrypted under the new public key p_2.

When allowing the encryption function to handle its own decryption function at the same time, Gentry showed that the noise added by the homomorphic operations was less than the noise removed by the additional decryption [12], and the breakthrough was made in hunt for a fully homomorphic encryption scheme. While Gentry's system is great in theory, it has been shown that creating FHE schemes is hard in practice. However, some other cryptographic schemes holds different types of homomorphic properties which when combined can provide different homomorphic operations. Addition is, for example, a homomorphic property of the Pailler cryptosystem [15]. Pailler is originally a trapdoor mechanism based on the Composite Residuosity Class Problem which conveniently has the cryptographic property such that

$$ENC_k(x) * ENC_k(y) = ENC_k(x + y)$$

Another practical cryptosystem providing a similar property, is the ElGamal cryptosystem [9]. Along with being an asymmetric cryptosystem , it has the cryptographic property shown below, which enables it to perform homomorphic multiplications.

$$ENC_k(x) * ENC_k(y) = ENC(x * y)$$

Chapter 3

Overview of CryptDB

This chapter covers the overview of CryptDB, starting with the system architecture, followed by its different encryption mechanisms and encryption layer adjustments. Along with the investigation, a small application will be introduced and used as an example to help the reader in understanding the practical and impractical usages of CryptDB.

3.1 System Architecture

CryptDB's architecture (Figure 3.1) is divided into two pieces, a proxy server and a database server. The proxy server is an intermediate server placed between the application server (used by the application to manage key set-up and interacting with clients) and the database server. The proxy server's purpose is to intercept queries going from the application to the database and anonymize the information of the query, as well as decrypting results going from the database server to the user's application. By doing so, it eliminates the possibility of eavesdropping attackers to obtain any sensible information.

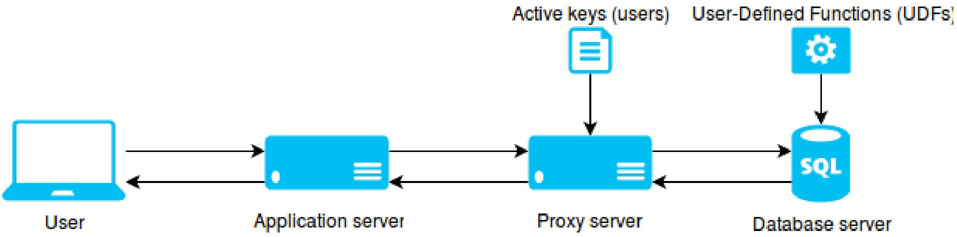


Figure 3.1: System architecture of CryptDB interacting with a client application

Another vital part of the proxy server's domain is to issue queries to the database that adjusts the encryption layer (or onion layer) of the data items(s) that the user has issued a query on, which will be discussed in section 3.3. The last responsibility of the proxy server is to keep track of the users that are currently logged in using a table of active keys. In addition to keeping the list of active keys, the proxy server also keeps an embedded database with the schemes of the encrypted tables in the database. By storing this, the proxy server keeps a continuous picture of the encryption layer for each column, which is used for adjusting the encryption level (or layer) for a column in order to allow certain types of functionality.

The authors [18] have implemented CryptDB both with MySQL and PostgreSQL database management systems. CryptDB requires no modifying of the database system, other than adding a set of User-Defined Functions (UDFs). UDFs in CryptDB allows the database to perform cryptographic operations on the encrypted data, as well as adjust the encryption level of different columns.

CryptDB has two operational modes, or principals. One for applications consisting of only one user, and another for multi-user applications. Application keys consists of both a symmetric key rooted in the users application password and a public-key pair. When logging in, the proxy derives encryption keys with the user's password as the root. These keys are used for accessing the data items that are accessible to that particular user and to encrypt new items. When the user logs out, the keys are deleted from the proxy server.

3.2 SQL-Aware Encryption and the Onion Scheme

CryptDB uses an encryption scheme called *SQL-aware encryption* or *onion encryption*. Basically, this is a collection of different encryption schemes, each providing different levels of security and computations to be executed. Data items stored using CryptDB are encrypted multiple times using these different schemes, or layers, of encryption. The result is a onion-like structure where the outer layers provide maximum security and low functionality, while the inner layers provide less security, but more functionality. Figure 3.2 shows the ordering of the different layers from most secure to least secure. Each layer will be explained throughout this section.

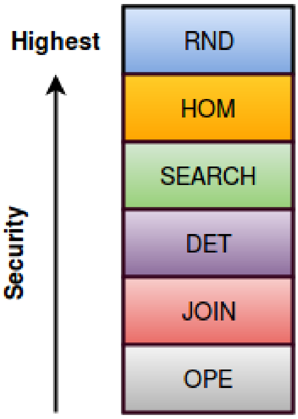


Figure 3.2: Ordering of the different encryption layers based on their security

As a scenario in order to describe the different operations and situations in CryptDB, assume a simple employee application with a table structure and some example records as shown in Table 3.1. Our application is created in order to provide some insight in the salary

distribution within our firm, with respect to age, number
of years within the firm and employment division. EmplNum is the number associated with the employee in other systems and applications, and SSN is the Social Security Number of the employee used for creating pay checks and similar operations.

Id int	EmplNum int	Name varchar(255)	SSN varchar(255)	Age int	Salary int	Division varchar(255)
1	42	Alice	24127312345	42	440000	Marketing
2	1	Bob	17054623456	69	850000	Management
3	1337	Charlie	31129134567	23	390000	Engineering
4	123	Donna	11117945678	36	510000	Management

Table 3.1: Employee table for a simple employee application with example records

3.2.1 Random

Random (RND) is the highest security level in CryptDB and provides the maximum security found in encryption scheme. It uses a strong block cipher such as Blowfish or Advanced Encryption Standard (AES) in Cipher-Block-Chaining (CBC) mode and a random initialization vector (IV) to ensure that the block cipher is probabilistic [18]. RND, being the maximum security level provided, does not allow any computation to be done on the encrypted data. In other terms, this level is a natural choice for sensitive data that are only meant to be read. When a block cipher is probabilistic, it has properties such that when encrypting the same message m_1 multiple times, the resulting ciphertexts are unequal.

For example, given two encryptions of the same plaintext $c_1 = E_k(m_1)$ and $c_2 = E_k(m_1)$, the resulting ciphertexts are c_1 and c_2 such that $c_1 \neq c_2$. Operations supported by this scheme are `SELECT`, `UPDATE`, `DELETE` and `INSERT`. It also supports altering an existing table by removing or adding columns using the `ALTER` statement. In our sample application, the SSN would be the most natural thing to stay encrypted under RND at all times, as there are not many natural operation to perform on such numbers. However, the other columns are not likely to be encrypted under RND as it is likely that the application would want to perform some operations which demands a lower encryption level. As for an example, take the perhaps most natural query at this encryption level, the `INSERT` statement

```
INSERT INTO employee_table
VALUES('', 5, 'Eric', '29026056789', 55, 680000, 'Engineering');
```

which adds a record in the database containing Eric's information. When it comes to SELECT, DELETE and UPDATE, it only supports regular queries without the WHERE clauses and similar statements:

```
SELECT * FROM employee_table;

UPDATE employee_table SET salary = 0;

DELETE FROM employee_table;
```

3.2.2 Homomorphic Encryption

Another vital part of SQL is the ability to perform addition and multiplication. For CryptDB to be able to perform these operations, it utilizes a Homomorphic Encryption Onion (HOM) scheme. As previously described, homomorphic encryption is a technique that enables computation on encrypted data without decrypting it first. CryptDB uses Pailler multiplication [15] for enabling summation operations such as SUM, ADD, INCREMENT and AVERAGE. If our application is in need for finding out the total sum of the salaries within the firm, we do something like this

```
SELECT SUM(Salary)
FROM employee_table;
```

Or extracting the total salary for each division by using the GROUP BY clause

```
SELECT Division, SUM(Salary)
FROM employee_table
GROUP BY Division;
```

Multiplication was not initially implemented, but has been implemented in some versions of CryptDB by an outside team using the El Gamal cryptosystem [19].

3.2.3 Search

In order to perform search for words in the encrypted texts, CryptDB has a scheme called SEARCH. This was an implementation of the cryptographic protocol suggested by Song et al. making it almost as secure as the RND scheme [18]. The main idea is to split a text encrypted with SEARCH into individual keywords on a given delimiter specified by the application developer. Removal of duplicate keywords and randomly permute them are also added to enhance the security. When executing a search, the server would be given an encrypted token of the keyword, and would retrieve encrypted values matching the token. Originally, SEARCH could perform `LIKE` operations as other database systems, but in the most recent version of CryptDB's software it has been deprecated.

3.2.4 Deterministic

As RND allows no computation to be done on the encrypted data and HOM only allows addition, the next layer brings some more functionality to the toolbox. Deterministic (DET) is an encryption scheme enabling the application to perform standard SQL operations such as equality checks, distinct, group by and count. By allowing these sorts of computation, the application leaks information to an adversary. In particular, it leaks which ciphertexts that decrypts to the same plaintext value. Following the previous example; if the scheme encrypted the message m_1 two times, the resulting ciphertexts c_1 and c_2 are such that $c_1 = c_2$. For our sample application, DET would be the scheme used when allowing operations such as `GROUP BY`, `DISTINCT` and `COUNT`. Continuing with our sample application, we now want to find out what types of departments that exists in our table

```
SELECT DISTINCT Division
FROM employee_table;
```

which returns `’Engineering’`, `’Marketing’` and `’Management’`. As described, a deterministic scheme encrypts the same plaintext to the same ciphertext every time. By taking advantage of this, CryptDB is capable of iterating over the encrypted data and return a distinct selection of the different ciphertexts observed. At the DET layer the `WHERE` clause is also supported when running equality checks. An example usage is the query below, where the name, age and salary of all employees from the management division are returned

```
SELECT Name, Age, Salary
FROM employee_table
WHERE Division = ’Management’;
```

For the encryption, DET uses a strong block ciphers, and either with 64-bit or 128-bit block size. If a value is larger than 128-bits, it leaks prefix equality when used with AES [18]. In order to cope with leaking prefix equality, the authors have designed their own version of the CMC mode.

3.2.5 Order-Preserving Encoding

Since SQL also allows the user to compute on order relations between items, CryptDB introduces an Order-Preserving Encryption (OPE) scheme [18]. This scheme is based on a requirement where the sort order of the ciphertext matches the sort-order of the corresponding decrypted plaintexts. It also requires that the scheme reveals no other information about the plaintexts, other than the respective order. For example, if a value $x \geq y$ then the corresponding encryption would be such that $Enc(x) \geq Enc(y)$. In SQL, such an operation is used for order comparison, which are range checks, ranking, sorting, and extracting minimum and maximum values.

CryptDB uses a modified version the scheme proposed by Boldyreva et. al [7] as their OPE scheme, which is a random order-preserving injective function. An injective function is a one-to-one random mapping function that preserves the order of the elements. Since queries related to order consists of retrieving items between two values, or ranges related to one particular value, the encrypted values can be stored as binary trees at the server, ensuring logarithmic run-time [7].

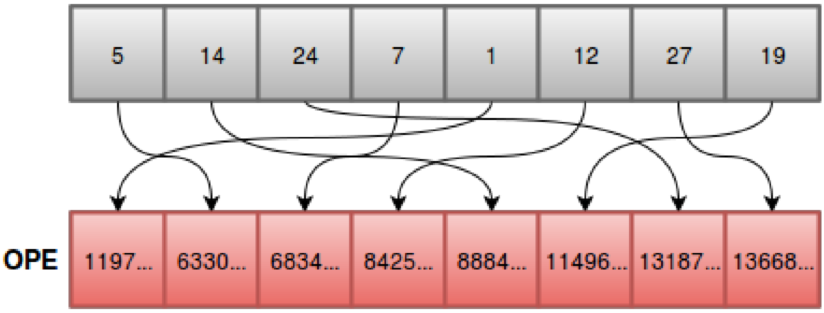


Figure 3.3: Order-preserving encryption of a set of integer values

Figure 3.3 shows the order-preserving encryption of a set of integers, where no information is leaked, except for the order of the ciphertexts. When searching for the persons in our imaginary firm that has a yearly salary over 500 000, the query uses the `WHERE` clause and then performs an order check where it navigates down the binary three for items that has a salary larger than the requested value.

```
SELECT Name, Salary
FROM employee_table
WHERE Salary > 500000;
```

Popa et al. have also proposed their own OPE scheme to go with CryptDB [17], but due to other priorities, this scheme was never actually implemented into

CryptDB's software.

3.2.6 Equality Join and Order-Preserving Join

When it comes to joining columns, two cases are supported in CryptDB. The first is the regular equality join (EQ-JOIN) between two columns, and the other is range joins (OPE-JOIN) which involves order relation checks. Ideally, the proxy server should know in advance which columns that should be allowed to be joined in order to encrypt these column with the same key. Because of the key-chaining approach where each data item is encrypted with a new key, CryptDB is in need of a separate encryption scheme in order to compute joins in a safe manner.

The authors [18] introduces a new cryptographic primitive for equality joins, Adjustable Join (JOIN-ADJ), which is a deterministic function. The idea is to let the proxy server adjust the encryption keys of columns in real-time, based on the observed query. By using this approach, CryptDB avoids the database server to compute joins on its own as attempt to learn about relations between different tables and columns. When a join query is observed at the proxy, it sends an adjusted key to the database server enabling it to adjust the values in one of the two affected columns. When an adjustment has been done, the columns share the key until the proxy server issues a new adjustment key to either one of the columns.

The second case is the order-preserving join, which depends on the OPE scheme previously described. The binary tree structure of the scheme makes in infeasible to use the same approach as EQ-JOIN. Therefore, CryptDB has a requirement that columns where such joins is applicable have to be declared by the application in beforehand and encrypted under the same key. If this measure has not been addressed, CryptDB will simply encrypt all columns with the same key.

Id	EmplNum	University	Graduated	Degree	AdjustedGPA*
int	int	varchar(255)	int	varchar(255)	int
1	42	HiST	1993	B.Sc.	319
2	1	NTNU	1971	M.Sc.	305
3	1337	UiO	2015	M.Sc	287
4	123	NHH	2009	MBA	392

Table 3.2: Educational table for the employee application. **Because of no support for floating values, the GPA is multiplied by 100 and rounded down.*

Assume that along with the table of employees, there is also a table holding the employees educational information as seen in Table 3.2. By performing a join between our two tables on the shared employee number EmplNum, it is possible to extract information from the two combined tables.

```
SELECT Name, Division, University, Graduated
FROM employee_table t1
JOIN employee_education t2
ON t1.EmplNum = t2.EmplNum;
```

3.2.7 Wrapping the Layers

These different layers are wrapped into four different classes or *onions*, namely `EQ`(Equality checks), `ORD`(Order relations), `SEARCH`(Searching) and `ADD`(Addition) as shown in Figure 3.4. Each onion has a special purpose in means of supporting certain operations, and every data item is encrypted each of the different onions using the different layers. However, the application does not necessary maintain all of onions. There is, for example, no need for maintaining the ADD-onion if the data item is a string.

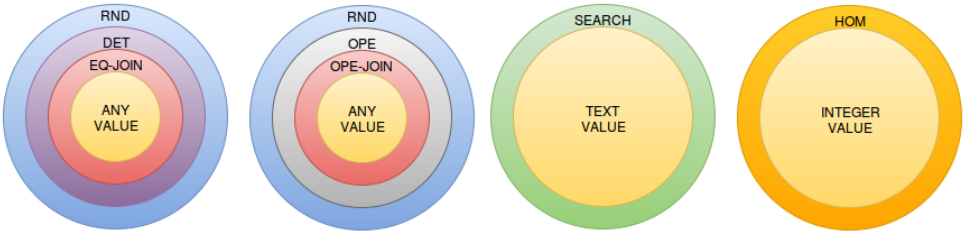


Figure 3.4: Overview of the structure of the SQL-aware Encryption Scheme. From left: The EQ-onion, ORD-onion, SEARCH-onion and ADD-onion

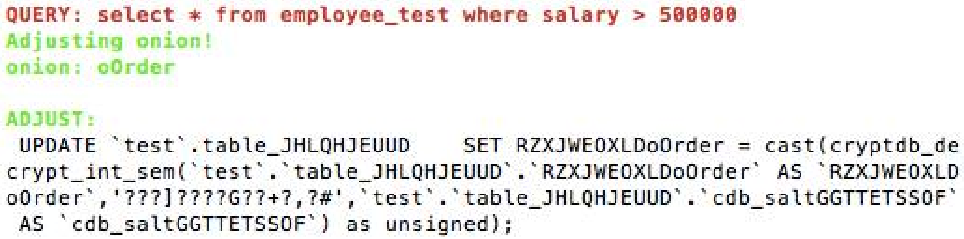
3.3 Adjusting the Encryption Level Based on the Query

Okay, so there is an encapsulation of different encryptions for each data item, and our ideal scenario is that our data is encrypted at the highest, feasible level at all times. But how is the system supposed to perform a range check if the utmost layer is the RND which does not support any functionality at all?

CryptDB solves this particular case gracefully by having the proxy server observe incoming queries in real-time. After observing a query, it checks its embedded table holding the encryption level of the all columns whether or not the current encryption level of the inflicted columns are supporting the query. For example in our sample application, the user has issued the query

```
SELECT *
FROM employee_table
WHERE salary > 5000000;
```

Have in mind that all data items at this point is encrypted with RND as the utmost layer. What the proxy does, is that before sending the rewritten query to the database server, it sends an `UPDATE` query first. This query orders the database server to remove the outer encryption level of the ORD-onion to match the OPE layer, before it sends the rewritten query. Figure 3.5 illustrates how the proxy server observes the query, and issues the `UPDATE`. `SET` is used to change the current encryption level of the ORD-onion, and `cryptdb_decrypt_int_sem` is the name of the UDF that the server will use to adjust the encryption layer to support the query. The curious string `'???]???G??+?,?#'` is the encrypted key which the server will use along with the UDF.



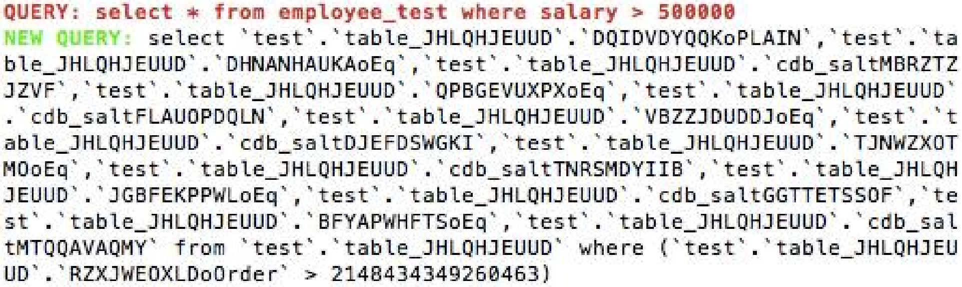
```
QUERY: select * from employee_test where salary > 500000
Adjusting onion!
onion: oOrder

ADJUST:
 UPDATE `test`.table_JHLQHJEUUD    SET RZXJWEOXLDoOrder = cast(cryptdb_de
crypt_int_sem(`test`.`table_JHLQHJEUUD`.`RZXJWEOXLDoOrder` AS `RZXJWEOXLD
oOrder`,'???]????G??+?,?#',`test`.`table_JHLQHJEUUD`.`cdb_saltGGTTETSSOF`
 AS `cdb_saltGGTTETSSOF`) as unsigned);
```

Figure 3.5: Encryption layer adjustment performed by proxy server upon receiving a query. (Is the figure too small/hard to read?. Adjust colors?)

After performing the encryption layer adjustment, the proxy rewrites the query as shown in Figure 3.6. As described, each of the columns are anonymized upon the creation of a table. The proxy server keeps table schemes in order to substitute the different parts of the query so it matches the encrypted columns in the database. For the record, 'test' is the name of the current database.

When a layer has been stripped off, it does not automatically get re-encrypted unless the developer has specified such actions. For example, if an irregular query forces the encryption level lower than necessary. For applications running in single-mode, the encryption layers will over time be in a stable state supporting the relevant queries. But, for multi-mode applications, the developer decides the functionality, hence the types of queries that the application can produce. Therefore, it is possible to start application at the correct layers based on the wanted functionality, and the proxy does not perform any encryption level adjustment at all.



```
QUERY: select * from employee_test where salary > 500000
NEW QUERY: select `test`.`table_JHLQHJEUUD`.`DQIDVDYQQKoPLAIN`,`test`.`ta
ble_JHLQHJEUUD`.`DHNANHAUKAoEq`,`test`.`table_JHLQHJEUUD`.`cdb_saltMBRZTZ
JZVF`,`test`.`table_JHLQHJEUUD`.`QPBGEVUXPXoEq`,`test`.`table_JHLQHJEUUD`
.`cdb_saltFLAUOPDQLN`,`test`.`table_JHLQHJEUUD`.`VBZZJDUDDJoEq`,`test`.`t
able_JHLQHJEUUD`.`cdb_saltDJEFDSWGKI`,`test`.`table_JHLQHJEUUD`.`TJNWZXOT
MOoEq`,`test`.`table_JHLQHJEUUD`.`cdb_saltTNRSMDYIIB`,`test`.`table_JHLQH
JEUUD`.`JGBFEKPPWLoEq`,`test`.`table_JHLQHJEUUD`.`cdb_saltGGTTETSSOF`,`te
st`.`table_JHLQHJEUUD`.`BFYAPWHFTSoEq`,`test`.`table_JHLQHJEUUD`.`cdb_sal
tMTQQAVAQMY` from `test`.`table_JHLQHJEUUD` where (`test`.`table_JHLQHJEU
UD`.`RZXJWEOXLDoOrder` > 2148434349260463)
```

Figure 3.6: Rewritten query to be sent to server after encryption level adjustment. (Is the figure too small/hard to read?. Adjust colors?)

3.4 Security

In section 3.1, two different modes for using CryptDB was mentioned. For applications built for one particular user, the single-user mode of CryptDB is used. Example of applications where such a mode would be useful is when creating a personal diary application or a contact list application. When building applications consisting of users and interaction between users, the developer needs to use the multi-user mode.

3.4.1 Single-user Mode

If an application is created with a single user in mind, then it should be running in the single-user mode where the keys are derived from one secret master key which is rooted in the user's password. By running in this particular mode, the user has access to all data encrypted through the proxy server. When running is this mode, the proxy server and the application are considered to be trusted. The database server however, is considered to be untrusted and subject to the snooping of a curious database administrator owning the server system hosting the database. As of the most recent versions of the CryptDB software, only the single-user mode is supported, which has been used to create a small demo application described in Chapter 5.

3.4.2 Multi-user Mode

Picture a simple online discussion forum, where users can interact by posting messages to discussion boards or sending each other private messages. As described in Section 2.1.1, database applications are in need of access control in order to restrict access to certain tables based on the user's credentials, hence the need for declaring user entities as well as different groups. But how can a user read messages on the discussion board if they are encrypted by another user's key? What about private messages, with suffers from the same case, as they will be encrypted with the sender's key? CryptDB solves these cases by letting developers use annotations in their schemas.

Table 3.3 shows a basic use of such annotations. PRINCTYPE is used to annotate external users, which are authenticated through the access control by providing their password, and internal users, which are entities inside the database. ENC_FOR is used to indicate which columns that consist sensitive data, and in the next column CryptDB needs to store which principals that should have access to the sensitive column. Okay, so we have access to the column, but how can we decrypt something encrypted with someone else's keys?

```
PRINCTYPE physical_user EXTERNAL;
PRINCTYPE user, msg;

CREATE TABLE privmsgs (
        msgid integer,
        subject varchar(255)        ENC_FOR (msgid msg),
        msgtext text                ENC_FOR (msgid msg) );

CREATE TABLE privmsgs_to (
        msgid integer,
        recvid integer,
        sendid integer,
        (sendid user) SPEAKS_FOR (msgid msg),
        (recvid user) SPEAKS_FOR (msgid msg) );
```

Table 3.3: Use of policy annotations when creating multi-user applications

CryptDB chains encryption keys used to encrypt columns or data items to the root key, which is the user's password. The SPEAKS_FOR annotation gives a principal access to all the keys that the principal it speaks for has access to. For example in Table 3.3, both the user sending the private message and the user receiving it, speaks for the principal type msg and thereby has access to all the keys of the msg entity. This basically means that the secret key of the message is encrypted two times, one time with the sender's key and one time with the receiver's key.

Consider this case: Alice wants to send a private message to Bob, which means that the private message needs to be encrypted two times using both Alice and Bob's secret keys. Keys of users who are logged in are kept in a table at the proxy server, but what happens when the proxy server needs to encrypt something with a key which at the time is inaccessible? The keys that are used by a principal to encrypt consists of both a symmetric key and a public-private-key pair. When both parties are logged in, the proxy uses their symmetric keys to encrypt the message. If Bob is currently logged out, the message is encrypted using his public key so that he can decrypt it using the corresponding private key when he logs in.

As previously mentioned, the current version of CryptDB has only the option of developing single-mode applications. One may react to the fact that the developers has stopped maintaining such a useful way of developing applications with multiple user's and interaction between users. However, Mylar [4] is the creators of CryptDB's new project. Mylar addresses the threat of the insecure proxy server being compromised by an attacker, and how to protect the confidentiality of the users' data in such events. During an attack on the proxy server in CryptDB, the only parties that are compromised are those who are currently logged in, where their keys are stored at the proxy server.

3.4.3 Securing Applications When Using CryptDB

Another importance related to annotations is the use of the `SENSITIVE` annotation. By labelling columns as sensitive, CryptDB provides strong security guarantees of the data stored in the affected columns and also semantic security [16]. When a column is marked as sensitive, the only encryption schemes that are allowed are RND, HOM, SEARCH and, if the column has the `UNIQUE` constraint from SQL, DET. The reason for enabling the DET scheme under a constraint of the data being unique, is as there are no equal plaintexts, there will be no equal ciphertexts, and the scheme leaks no information about the encrypted data.

3.5 Limitations

FHE is nowhere near being practical yet, but CryptDB tries it best to be a somewhat practical HE scheme allowing a set of operations that they claim to be enough to support 99.5% of all queries observed in a SQL trace from a production MySQL server [18]. Although, there are multiple fairly trivial data types that CryptDB does not support.

For example, all fixed-point types and floating-point types are not supported, meaning that computing on decimal values are difficult for the application to perform. In Table 3.2, our GPA-column, where the most natural data type would be decimal, has been adjusted to integer by the application. One way to cope with the limitations, is to let the developer be responsible to transform a data type that is not supported into a supported one, and handle the inverse transformation upon receiving an encrypted result. Another option is to let CryptDB exclusively store such values as text with RND or DET encryption, and let client perform operations on such data types. As you probably understand, the last option is not desirable, as it is quite the opposite the goal when using such schemes as CryptDB.

Another data type that is not supported is those related to date and time, namely Timestamp, Date, Time, DateTime, Year and NewDate. In other terms, if your application needs to encrypt dates and at the same time be able to compute on them, you are in bad luck. The authors suggestion is to encrypt each part of a data in separate columns, and perform the computations on those columns instead of a composite date column. While being a rather complicated approach, it works surprisingly well for our demo application. Assume that we add some more columns to Table 3.1 with the precise date of birth in stead of age as seen in Table 3.4.

...	Day	Month	Year	...
...	int	int	int	...
...	24	12	1973	...
...	17	05	1946	...
...	31	12	1991	...
...	11	11	1979	...

Table 3.4: Additional columns to the employee table with date properties stored in separate columns

Consider 17th of May 1980 as our test date, and that we want all employees born after the test date. In a regular database system supporting date related data types, our query would be something like this on a concatenated date of birth column

```
SELECT Name, date_of_birth
FROM employee_table
WHERE date_of_birth > '17/5/1980'
ORDER BY date_of_birth ASC;
```

As for CryptDB where such queries do not return anything meaningful, we need to modify our approach as seen below

```
SELECT Name, Day, Month, Year
FROM employee_table
WHERE (year = 1980 and month = 5 and day > 17)
OR (year = 1980 and month > 5)
OR (year > 1980)
ORDER BY year ASC, month ASC, day ASC;
```

Queries are also subject to some constraints. In SQL, users are also allowed to use *subqueries* or *nested queries*, which is an SQL query embedded inside another query using the WHERE clause. Such subqueries are used by the main query to retrieve data that will be used as a condition in order to further restrict its result. If we now want to retrieve salaries for employees' graduating after 1980, a regular SQL query would be something like this

```
SELECT Name, Salary
FROM employee_table
WHERE EmplNum IN
    (SELECT EmplNum
    FROM employee_education
    WHERE graduated > 1980);
```

In CryptDB, such queries are infeasible, which leaves us with two options. Option one is to perform a join instead using subqueries. While this may solve our problem in this particular situation, it does not necessary work in all situations and may create duplicated rows in the result. Option two is to let the application execute the operation as two separate queries and store the result of the first query as an intermediate value used in the second query.

```
temp_result :=
    SELECT EmplNum
    FROM employee_education
    WHERE graduated > 1980;

SELECT Name, Salary
FROM employee_table
WHERE EmplNum IN (temp_result);
```

Along with complicating tables and adding overhead to the database, such limitations also puts the responsibility of the developer to add software code on the application server or client side. For larger applications and complex operations, such actions are not necessary feasible and hence an limitation to the use of CryptDB as a database system.

Should I mention the fact that multi-user is no longer supported as a limitation?

Chapter 4 Attacking CryptDB

In this chapter, a few known attacks on CryptDB will be presented along with the discussion around...

4.1 Tampering with CryptDB

Akin and Sunar [6] introduces an attack approach where a malicious database administrator targets web applications running in multi-user mode. More specific, software developed using the popular open-source software phpBB [5] where highly customisable forums can be created. By creating a fake user through the web applications, and at the same time be able to observe queries arriving at the database server, a malicious database administrator is able to attack CryptDB.

When creating tables in CryptDB, the order of the columns are preserved. By combining this information with the publicly available source code from phpBB, and attacker is able to understand the encrypted schema of the user table. When creating the fake user and logging in, the administrator can observe in the database the login attempt, and also determine the fake user's row id. By trying to log in with someone else's user-name (and obvious failing), the administrator is able to locate a specific user in the encrypted table. Now the malicious administrator can start to tamper with the database.

By copying any user's encrypted information directly into the fake user's corresponding fields, the attacker may be able to obtain encrypted information without necessary breaking the encryption. Another possibility is for the database manager to overwrite the password of every user with the encrypted version of the fake user's password. The consequence of this scenario is a database manager that can log into the application with every user.

It is clearly stated that CryptDB does not provide any other security guarantees except for data confidentiality [18]. As this attack clearly targets data integrity, it is not necessary CryptDB's a weakness in the software itself. However, it gives a good example on how providing data confidentiality is not enough in order to secure a database system. Data integrity should be provided for both the encrypted data as well as the queries observed at the database.

4.2 Microsoft Research and their Frequency Analysis

In September 2015, a Microsoft Research team released a paper stating that they had successfully broken the OPE and the DET encryption schemes of CryptDB [14]. They did so by using one of the oldest trick in a crypt analyst's book; frequency analysis.

Frequency analysis is to exploit the fact that in most spoken languages, some characters and combinations of characters are more common than others. In the English alphabet, E, T, A and O are the most regular characters, while J, Z, Q and X are considered to be rare. As described in 3.2.4 and 3.2.5, the DET and OPE schemes are deterministic and therefore encrypts the same plaintext to the same ciphertext. OPE also reveals the order between the ciphertexts. Because of this, an attacker with access to the ciphertexts is able to compute a histogram of the observed frequencies in the encrypted data. By comparing the histogram to a similar histogram of frequencies computed from closely related plaintext data, the attacker is able to determine which ciphertext characters corresponds to which plaintext characters with fairly large advantage.

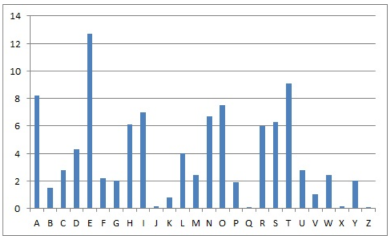


Figure 4.1: Bar chart of the letter frequency observed in the English language

As described in section 3.2, in order for an application to be able to execute equality checks or order comparison, columns need to be encrypted with DET or OPE respectively.

From an encrypted test database containing health records from multiple hospitals in the United States, Naveed, Kamara and Wright claimed to successfully have recovered 80% of OPE-encrypted ciphertexts from 95% of hospitals, and 60% of DET-encrypted ciphertexts from 60% of hospitals [14]. Their concrete attacks was a regular frequency analysis for attacking DET-encrypted columns along with a new attack called *P-Optimization.* For decrypting OPE-encrypted columns, they used a regular sorting attack and a new cumulative sorting attack.

4.2.1 Frequency Analysis on DET-encrypted Columns

For decrypting DET-encrypted columns, frequency analysis was used where the ith most frequent element of an encrypted column c was assigned to the ith most frequent element in an auxiliary dataset built with similar structured data. In addition to performing a plain frequency analysis, the researchs also presented a new attack called *P-Optimization.* Instead of ordering frequencies from most frequent to less frequent, they find an allocation of ciphertext-plaintext pairs that minimizes the overall difference between the histogram computed by the frequency analysis and the auxiliary one. Finding the allocation that minimizes the difference in the histograms is formulated as Linear Sum Assignment Problem (LSAP), and can be solved by using a linear programming solver [14].

For DET-encrypted columns, these attacks recovered the mortality risk and whether a patient lost its life while under hospital care for 100% of the data items in 99% and 100% of the hospitals respectively. They also recovered 100% of the data items in the column storing the severity of a disease for 51% of the hospitals. Additionally to these columns, they also show that it is possible to decrypt some of the information related to race, sex, admission type, primary payer and major diagnostic category [14].

4.2.2 Sorting Attack on OPE-encrypted Columns

When attacking columns encrypted using the OPE scheme, Microsoft Research used two approaches. The first approach was to use a sorting attack given that all of the columns where *dense.* By dense, they mean that both the encrypted column and the equivalent column in the auxiliary dataset contains every element from the plaintext space. By sorting both values from the encrypted column and the auxiliary dataset and map the values based on rank, the attack is able to decrypt 100% of the encrypted column. However, if the column does not contain all the possible plaintext from the plaintext space, the methodology has to be modified.

Cumulative attack for low-density columns is an approach where the attacker does not only leverage the scheme leaking the frequency of the ciphertexts, but also the relative ordering of the ciphertexts. If a ciphertext c is greater than 90% of the ciphertexts in the encrypted column, then it should be matched against values that are greater than 90% of the elements in the plaintext space. By combing a Cummulative Distribution Function (CDF) and frequency analysis, and using an LSAP solver to find the most optimal pairing of ciphertexts and corresponding plaintexts, Microsoft Research was able to recover more than 80% of encrypted data from 95% of the hospitals [14].

4.2.3 CryptDB Developers Answer Microsoft Research

In response to Microsoft Research's paper, the developers of CryptDB has presented a paper containing guidelines [19] on how to safely use CryptDB and remarks on where they claim Microsoft Research has used CryptDB in an unsafe way. One of the alleged flaws in the paper published by Microsoft Research was there inaccurate usage of the DET and OPE schemes. As stated in section 3.4.3, sensitive data should be annotated using `SENSITIVE` which ensures that the column is encrypted under a semantically secure encryption scheme such as RND, HOM, SEARCH or DET if the column has the unique constraint [16].

While Microsoft Research may have used CryptDB in an insecure way, it still addresses an interesting issue. Social security numbers, bank accounts, and such columns may be well suited for being marked as sensitive. But what about sensitive columns where one of the key features of the application is to perform order checks? For example in a medical system used at an emergency room, the doctors want to sort their patients based on the severity of their injuries and treat them accordingly. For this situation to work out in practice, the column has to be encrypted with OPE, and therefore is vulnerable to the attacks presented above. After all, the main idea of such systems as CryptDB is not to perform operations on non-sensitive data.

Chapter 5
CryptDB as a Software

Although CryptDB is mainly a proof of concept rather than a commercial software, the source code is available along with instructions on how to install it. This chapter will cover the installation and setup of CryptDB's proxy server, how to connect the proxy server to a regular database system, and finally a small demo application will be presented.

5.1 Installation of CryptDB's Software

CryptDB comes as a software available through MIT, and can be downloaded using the version control system Git. As the creators has moved on to other projects, the source code has not been maintained since 2014, and is currently only tested on Ubuntu 12.04 and Ubuntu 13.04. During the installation of CryptDB, some issues has been encountered. Since CryptDB is only tested with said Ubuntu versions, the natural approach would be to install the software on one of the two Ubuntu versions. This often involves running some sort of Virtual Machine (VM) in some Virtual Machine Monitor (VMM), which is very inconvenient when developing applications that should be able to run regardless of operating system and environment. When working inside a VM installing and working with software that is at best ready for alpha testing, you better watch your steps.

5.1.1 Docker to the Rescue

Because of the presented reasons, some of the work related to this project has been to detach CryptDB from such requirements and providing a workable environment regardless of operating system which is easy to reboot if one were to "step on the wrong files". Docker [2] is an open-source software allowing programs to be wrapped up in *containers*. Containers are complete file systems containing every piece of code, library and dependency the program need to run properly, which runs directly on the host OS without the need of a VMM and Guest OSs. After a container is built, it can be shipped and run in other environments such as Windows, OS X and Linux without issues or adjustments.

5.1.2 Installing CryptDB using Docker

A guide to installing and setup of CryptDB's database and proxy server can be found in Appendix A.1 along with a small demonstration of the software as presented in Appendix A.2.

5.2 A Small Demo Application

How big should a demo application be? What should be in here?

Single-mode application because of the deprecating of multi-mode in recent releases of the software.

5.3 Ninja hacks

To perform nested SQL queries:

Store results as intermediate values, then new query.

5.4 Discussion of CryptDB as a Software

The authors of CryptDB has taken the idea further with Mylar (https://css.csail.mit.edu/mylar/ which I assume to be CryptDB 2.0, or at least building on the main idea. Have not studied yet.

Chapter 6 Conclusion

6.1 Impact of CryptDB

A lot of start-ups, Google, Microsoft, etc. are using CrytpDB's building blocks and ideas in their commercial software.

6.2 Useful applications

Suggestions

6.3 Conclusion

Some conclusion.

References

[1] Apple Health. http://www.apple.com/researchkit/. Accessed: 2015-10-23.

[2] Docker. https://docker.com/. Accessed: 2015-11-12.

[3] Microsoft Health. https://www.microsoft.com/microsoft-health/en-us. Accessed: 2015-10-23.

[4] Mylar - MIT. https://css.csail.mit.edu/mylar/. Accessed: 2015-11-11.

[5] phpBB. https://phpbb.com/. Accessed: 2015-11-19.

[6] Akin, I. H., and Sunar, B. On the Difficulty of Securing Web Applications using CryptDB. In *Big Data and Cloud Computing (BdCloud), 2014 IEEE Fourth International Conference on* (2014), pp. 745–752.

[7] Boldyreva, A., Chenette, N., Lee, Y., and O'Neill, A. Order-Preserving Symmetric Encryption. In *Advances in Cryptology - EUROCRYPT 2009*. 2009, pp. 224–241.

[8] Damgård, I., Faust, S., and Hazay, C. Secure two-party computation with low communication. In *Theory of Cryptography*. 2012, pp. 54–74.

[9] ElGamal, T. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In *Advances in Cryptology* (1985), pp. 10–18.

[10] Gentry, C. *A Fully Homomorphic Encryption Scheme*. PhD thesis, 2009.

[11] Gentry, C. Fully Homomorphic Encryption Using Ideal Lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing* (2009), pp. 169–178.

[12] Gentry, C. Computing arbitrary functions of encrypted data. *Commun. ACM* (2010), pp. 97–105.

[13] Naehrig, M., Lauter, K., and Vaikuntanathan, V. Can Homomorphic Encryption be Practical? In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop* (2011), pp. 113–124.

[14] Naveed, M., Kamara, S., and Wright, C. V. Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (2015), pp. 644–655.

[15] Paillier, P. Public-key Cryptosystems Based on Composite Degree Residuosity Classes. In *Proceedings of the 17th International Conference on Theory and Application of Cryptographic Techniques* (1999), pp. 223–238.

[16] Popa, R. A. *Building practical systems that compute on encrypted data.* PhD thesis, Massachusetts Institute of Technology, 2014.

[17] Popa, R. A., Li, F. H., and Zeldovich, N. An ideal-security protocol for order-preserving encoding. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (2013), pp. 463–477.

[18] Popa, R. A., Redfield, C. M. S., Zeldovich, N., and Balakrishnan, H. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), pp. 85–100.

[19] Popa, R. A., Zeldovich, N., and Balakrishnan, H. Guidelines for Using the CryptDB System Securely. Cryptology ePrint Archive, Report 2015/979, 2015.

[20] Rivest, R. L., Adleman, L., and Dertouzos, M. L. On Data Banks and Privacy Homomorphisms. *Foundations of Secure Computation* (1978), pp. 169–180.

Appendix A

Appendix

A.1 Setup CryptDB Using Docker

1. Make sure to have Docker installed. Docker can be installed by following the procedure presented at http://docs.docker.com/v1.8/installation/ depending on your operating system. For operating systems other than Linux, Docker is available by installing the Docker Toolbox.

   Note: For readers running OS X or Windows, remove the 'sudo' part of the commands in this setup.

2. Create a folder for your project and open a terminal window while standing in your new folder.

3. Clone the github project by running

   ```
   git clone https://github.com/klevstad/CryptDB_Docker.git
   ```

4. Inside the cloned project, navigate to the folder containing the file called `Dockerfile`.

5. Build a docker image by running the command below in your terminal window. This will download and install the CryptDB software along with MySQL and a small script for running the CryptDB proxy server.

   Command:

   ```
   sudo docker build -t name-of-image:version .
   ```

   Example usage:

   ```
   sudo docker build -t cryptdb:v1 .
   ```

6. After successfully building the docker image, open the Docker Toolbox software if running OS X or Windows. This will open a command line tool that will connect you to your internal Docker machine. All Docker commands must be executed from a terminal windows started through the Docker Toolbox software. If running Linux, you can ignore this step.

7. Run a Docker container based on the previously built image by running the command below. This will create an independent container with the CryptDB software and can easily be started and stopped, or destroyed if necessary. The use of the $-p$ flag indicates what the container's incoming ports should be mapped to on the inside of the container. $-d$ tells Docker to run the container in the background, and $-e$ is used to inject the MYSQL root password into the MYSQL server running inside the container.

   Command:

   ```
   sudo docker run -d --name name-of-container -p port-in:port-out
   -p port-in:port-out -e MYSQL_ROOT_PASSWORD='mypassword'
   name-of-image:version
   ```

   Example usage:

   ```
   sudo docker run -d --name cryptdb -p 3306:3306 -p 3307:3307
   -e MYSQL_ROOT_PASSWORD='letmein' cryptdb:v1
   ```

8. For entering the Docker container, use the command below.

   Command:

   ```
   sudo docker exec -it name-of-container bash
   ```

   Example usage:

   ```
   sudo docker exec -it cryptdb bash
   ```

A.2 Play Around With CryptDB

To play around with CryptDB without any application running, use the following approach using three terminal windows.

A.2.1 Terminal 1: The Proxy Server

This subsection shows how to start the proxy server that will intercept and rewrite queries sent to the database. After entering the container using the command from Step 8 in A.1, type the following command into the terminal window:

```
/opt/cryptdb.sh start
```

This starts the CryptDB proxy server. For stopping the server, abort using `CTRL+C` and run

```
/opt/cryptdb.sh stop
```

A.2.2 Terminal 2: The CryptDB Client

This subsection shows how to start a client window connected to the proxy server. In a new Docker terminal window, enter the container using the command from Step 8 in A.1 and type the following command into the terminal window:

```
mysql -u root -pletmein -h 127.0.0.1 -P 3307
```

Create a database, use it, create tables, etc. Observe that the proxy server intercepts the queries and rewrites them. Also, the data is in plaintext and readable for the logged-in user. `-P 3307` instructs the client to connect to the CryptDB proxy server instead of the mysql database.

A.2.3 Terminal 3: The Snooping Database Administrator

This subsection shows how the data in the database stays encrypted when logging in as a regular database administrator without going through the proxy server. Enter the container using the command from Step 8 in A.1 and type the following command into the terminal window:

```
mysql -u root -pletmein -h 127.0.0.1
```

Snoop around in the database. Observe that all the data is encrypted and impossible for you to decrypt or make any sense out of. The `-P` flag is omitted in this example as MySQL will connect to port 3306 at default.