NTNU – Trondheim
Norwegian University of
Science and Technology

Exploring CryptDB:
A Practical HE Scheme for SQL Queries

Eirik Klevstad

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

Norwegian University of Science and Technology
Department of Telematics

Abstract

CryptDB is a practical homomorphic encryption scheme for SQL queries, enabling storage and processing of encrypted data. The system consists of two entities. One being a custom proxy server, which responsibility is to intercept and rewrite queries that are sent between the user and the database server, which is the second entity. The database server is a regular relational database supporting SQL queries, where a small set of CryptDB specific functions have been implemented.

CryptDB utilizes an SQL-aware encryption scheme, meaning that data items that are sent to the database are encrypted multiple times with different encryption schemes. The encryption schemes are encapsulated in each other from most secure to least secure. By doing so, CryptDB provides confidentiality to the data, while still being able to execute most queries seen in regular SQL traces.

This report assesses the different encryption schemes used in CryptDB along with its security and limitations. The report also indicates that some of the proposed building blocks suggested in CryptDB for homomorphic encryption schemes may not be that well suited for real world applications. It also describes how to install and set up CryptDB, along with a demonstration application to assess and present some of its use-cases.

Keywords: CryptDB, Databases, Homomorphic Encryption, SQL

Acknowledgements

I would like to thank professor Colin Boyd for valuable insight and help in order to complete this project. I would also like to thank my supervisor, Ph.D. Candidate Christopher Carr, for his excellent supervising, feedback and help during the project. Finally, a shout-out to Andreas Mosti, System Engineer as DIPS ASA, for his excellent assistance and tips when working with Docker and virtual environments.

Contents

List of Figures

List of Tables

List of Acronyms

AES Advanced Encryption Standard.

API Application Programming Interface.

CBC Cipher-Block-Chaining.

CDF Cummulative Distribution Function.

DBMS Database Management System.

DET Deterministic.

EQ-JOIN Equality Join.

FHE Fully Homomorphic Encryption.

GCD Greatest Common Divisor.

HE Homomorphic Encryption.

HOM Homomorphic Encryption Onion.

IV Initialization Vector.

JOIN-ADJ Adjustable Join.

LSAP Linear Sum Assignment Problem.

LWE Learning With Errors.

MIT Massachusetts Institute of Technology.

OPE Order-Preserving Encryption.

OPE-JOIN Order-Preserving Encryption Join.

OS Operating System.

PHE Practical Homomorphic Encryption.

PKE Public Key Encryption.

RND Random.

RSA Rivest - Shamir - Adleman.

SQL Structured Query Language.

SSN Social Security Number.

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 [5] and other corporations are increasing their focus on 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. Competitions, such as iDASH [4] have been held in order to create the best suited methodology for how to securely process and store sensitive information. But in 2015, encrypting and storing sensitive information is not enough. Sensitive information is valuable, and even more valuable if we are able to utilize it in a secure manner.

As developers, we are left with two choices when creating applications utilizing some sort of data storage. 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 (HE) is a cryptographic property describing the ability to perform certain operations on encrypted data (ciphertexts) without decrypting it first. Fully Homomorphic Encryption (FHE) is the enhanced version where the encryption scheme is capable of performing all functions efficiently [16]. While still in research mode, today's fully homomorphic encryption schemes' biggest challenge is the fact that they are highly inefficient. While being a hot research topic, we may assume that FHE schemes still have a long way to go before being deployed in commercial systems [19]. However, if the practical systems ever reaches the theoretic potentiality of FHE, such schemes and systems could be a possible game changer for cloud based services in the future.

In 2011, a research team at the Massachusetts Institute of Technology (MIT) presented CryptDB, which is an encrypted database system providing practical and provable confidentiality by utilizing a restricted form of homomorphic encryption [24]. CryptDB allows its users to interact with the system by issuing Structured Query Language (SQL) queries over the encrypted data in an efficient manner. While being a proof of concept, its pioneering design is being used by large companies such as Google and Microsoft [2].

1.2 Problem

The objective of this project is to critically assess CryptDB in terms of functionality and security, along with developing a small demonstration application to better understand its possible use cases. This project has resulted in an overview and a security assessment of the CryptDB system, a guide on how to install and run CryptDB regardless of operating system, and finally a small sample application for demonstration purposes.

1.3 Overview of the Report

In Chapter 2, a general background on databases and homomorphic encryption will be presented. Chapter 3 consists of an analysis of CryptDB in terms of functionality and security, while Chapter 4 covers some of the different attacks on CryptDB that have been proposed in the past years. The demonstration application and its results are presented in Chapter 5. Finally, Chapter 6 contains a comparison of CryptDB to a FHE scheme, as well as a discussion on the future of CryptDB as a possible solution for HE schemes.

Chapter 2
Background

This chapter covers the cryptographic background of the necessary components of a database in order to understand CryptDB, along with some of the basic cryptographic principals and homomorphic encryption.

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. More formally, databases can be described as a set of related data that is 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.

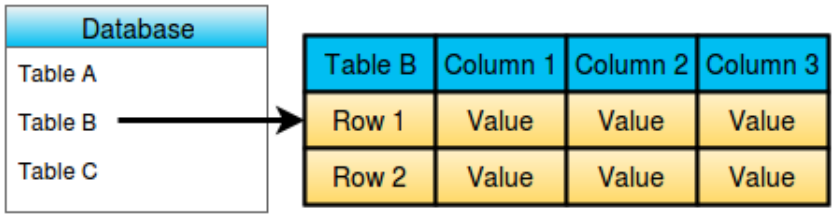


Figure 2.1: A table within a database consists of multiple columns. When inserting values into a table, a new row will be created storing the values.

A Database Management System (DBMS) is a piece of 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 DBMSes such as MySQL, PostgreSQL, Oracle and Microsoft SQL Server.

Let us assume a DBMS with a database storing some tables with our data. How do we put data in it, 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 a DBMS to insert and extract data from databases, as well as compute on the data stored within it. SQL is the standard language for performing queries on relational databases, which enables systems to insert, update and delete data, as well as performing functional operations such as summation, average, and finding maximum and minimum values.

2.2 Database Security

In today's world, companies store banking information, health records, and other sensitive information in their databases, which makes them a target for criminals and hackers. In order to cope with attacks from such groups, systems that are relying on some sort of database structure are in need of defence mechanisms and protection.

Network Security As database systems are usually connected to a network, they may be accessible from outside networks such as the internet. Attackers may break into the database by weaknesses in the network and steal sensitive information.

Usually, 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 deflections may indicate that the system is under attack.

Access Control When storing data in a database, it is not always desirable for all users to be able to access all parts of the database. If the database is left open for anyone to connect to, one of the risks is that strangers gain access to the database and possibly sensitive information. But granting access in a database is more than just keeping strangers out.

In order to prevent such incidents most databases utilize access control, which is used for granting different privileges to a user or group of users. In a physical setting, this would be to hand out id cards for users which should be able to access the data. For database systems, the id cards are replaced by login credentials, where

the user identifies itself with a user name and password. This can also be leveraged into granting different types of users or groups of users access to different types of data. A normal user should for example only be able to access relevant data for its role, while an administrator should be able to access all data.

Auditing With a high security clearance and access to sensitive data, great responsibility follows. For applications where trusted users have access to very sensitive information, problems may arise. Sometimes sensitive information in a special table is leaked from a firm or a criminal investigation, and the people in charge are left with a possible mole in their ranks.

Auditing is used by a system administrator to log which users access which data, as well as the time the data was accessed. Such security measures do not directly prevent the security from being breached, but allows the system administrators to locate the possible mole in such incidents.

Data Encryption People with physical access to a database may be able to bypass the security measurements that are supposed to protect the database and make a hard copy of it. If the data is stored in plaintext on the database, access control and auditing do not offer much protection against an on-site attacker.

Encryption is used to preserve confidentiality of the data stored in the database. By encrypting sensitive data under some secret key, it increases the difficulty for an attacker to obtain sensitive information, given that the secret key is securely hidden. This means that a curious database administrator is able to view all of the data, given that the secret keys are stored somewhere of the database and the administrator has root privileges.

2.3 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 piece-by-piece). The scheme usually consist of three algorithms, $KeyGen$, Enc and Dec.

$KeyGen$ is the algorithm that generates the secret key sk used for encryption and decryption, Enc_{sk} encrypts the data with the secret key, and Dec_{sk} decrypts it using the same secret key. One of the major drawbacks with symmetric key cryptography is that the secret key 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 encrypting data under some secret key sk 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 and straight-forward.

2.4 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*. This assumption is called the computational hardness assumption, where we assume that cryptosystems built on these problems are considered to be difficult for an attacker to break, given that the attacker utilizes the current state-of-the-art computational power. Examples on such problems are integer factorization, finding discrete logarithms and finding points on 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 on a problem similar to the integer factorization problem [27], and ElGamal, which relies on the discrete logarithms problem [15]. 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.

2.5 Homomorphic Encryption

We love to describe encryption as safes where we store our data, then secure 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 open 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 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.

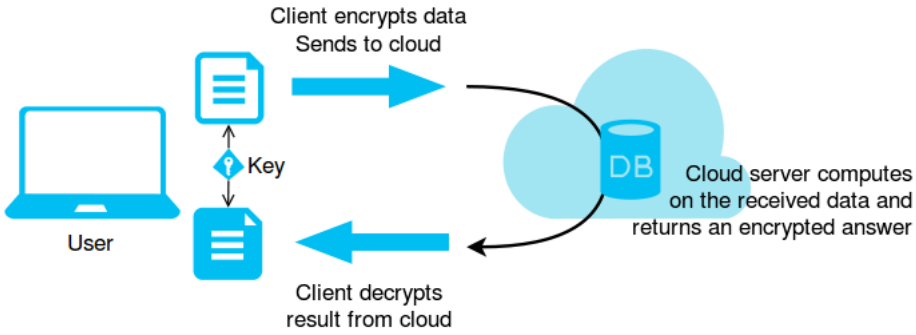


Figure 2.2: Illustration of a homomorphic system where data is encrypted at the client side, sent to the server, which performs computation on the encrypted data, and returns an encrypted result which is decrypted by the client.

Figure 2.2 illustrates the main concept of homomorphic encryption, where the client transmits encrypted data for the server to compute on, and decrypting the resulting answer without the server ever to know what it has computed on. 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). 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 encrypted under pk such that $Dec_{sk}(c) = C(m_1, ..., m_l)$ for the corresponding secret key sk in the public-key pair [14].

For Practical Homomorphic Encryption (PHE) schemes, $Eval$ will be associated with a set of permitted functions f (or circuits). These are functions that the algorithm can handle, and which guarantee a meaningful result when executed. Such functions can be expressed as boolean circuits consisting of logical gates such as AND, OR and NOT. Gentry presented a homomorphic encryption scheme consisting of three functions; addition (Add_ϵ), subtraction (Sub_ϵ) and multiplication ($Mult_\epsilon$) [18]. 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 the ciphertext in a reliable way in order to obtain the correct result.

2.6 Fully Homomorphic Encryption

FHE, which has no restrictions on what types of operations can be performed on the encrypted data, was first suggested in 1978 by Rivest, Adleman, and Dertouzos [27]. 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 [17], and a year later another scheme using a *bootstrappable* approach [18]. Nearly all modern FHE schemes are based on this bootstrappable concept using Gentry's blueprints [29].

As mentioned in the previous section, a homomorphic operation creates noise when executed. Multiple operations add 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 adding it 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 noisy than the original c_1 ciphertext, to continue performing homomorphic operations on.

Equation 2.1 encrypts the message m under the public key pk_1 as most asymmetric encryption schemes do, 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, which is a decryption function from our set of permitted functions, to decrypt 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.

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

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

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 [18], and a breakthrough was made in the hunt for a fully homomorphic encryption scheme. While Gentry's system is great in theory, it has been shown that creating FHE schemes is difficult in practice. However, some other cryptographic schemes hold different types of properties which when combined provide different homomorphic operations. Addition is, for example, a homomorphic property of the Pailler cryptosystem [21]. Pailler is originally a trapdoor mechanism based on the Composite Residuosity Class Problem which conveniently has a 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 [15]. 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 provides an overview of CryptDB, starting with the system architecture, followed by its different encryption mechanisms and encryption layer adjustments. Along with the assessment, 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 is divided into two pieces, a proxy server and a database server as seen in Figure 3.1. 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. Its 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 on the server side.

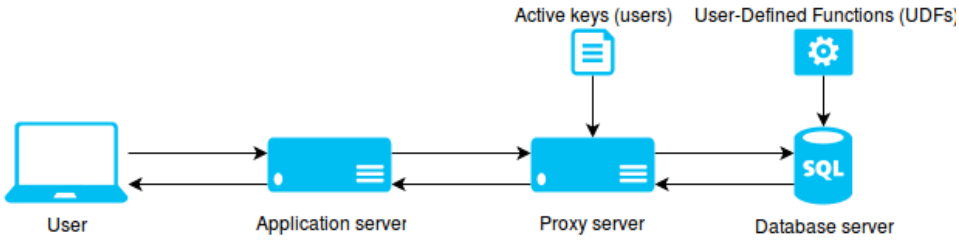


Figure 3.1: System architecture of CryptDB interacting with a client application.

Another vital part of the proxy server's domain is to issue queries to the database that adjusts the encryption layer of the columns 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 layer for a column in order to allow certain types of functionality.

The authors [24] 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*. 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.

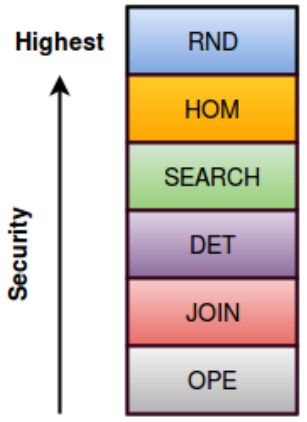


Figure 3.2: Ordering of the different encryption layers based on the security they provide.

In order to describe the different operations in CryptDB assume, as a scenario, 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 Social Security Number (SSN) used for creating paychecks for employees 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 the 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 [24]. 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 is only meant to be read. When encrypting the same message m_1 multiple times with a block cipher that is probabilistic, the resulting ciphertexts c_1, ..., c_n are unequal.

For example, given two encryptions of the same plaintext m_1; $c_1 = Enc_{sk}(m_1)$ and $c_2 = Enc_{sk}(m_1)$ under the same secret key sk, the resulting ciphertexts are c_1 and c_2 such that $c_1 \neq c_2$. SQL 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 employee table 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 for enabling additive operations such as `SUM` and `AVERAGE` [21]. If our application is in need of finding out the total sum of the salaries within the firm, we would do something like this

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

Multiplication was not initially implemented, but has been implemented in some versions of CryptDB by an outside team using the El Gamal cryptosystem [25].

3.2.3 Search

In order to perform a 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 [24]. The main idea is to split a text encrypted with SEARCH into individual keywords on a given delimiter specified by the application developer. To enhance the security, duplicate keywords are removed before they are randomly permuted. 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. In the most recent version of CryptDB's software it has been deprecated, as the developers did not have the time to integrate it into the latest version.

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. Deter-

ministic (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 computations, 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 returning 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';
```

We could also extract the total salary for each division by using the `GROUP BY` clause along with the `SUM` operation introduced previously:

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

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 [24].

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 [24]. 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 of the scheme proposed by Boldyreva et. al [13] 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 [13].

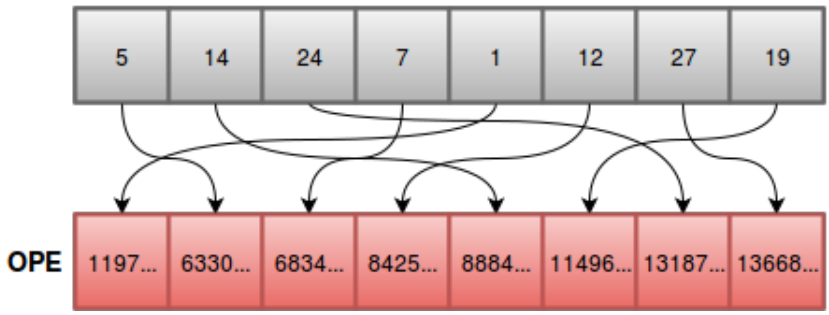


Figure 3.3: Order-preserving encryption of a set of integer values.

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 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 [23] which the claim to be an enhancement of the scheme proposed by Boldyreva, but due to other priorities, this scheme was never actually implemented into CryptDB's software.

3.2.6 Equality Join and Order-Preserving Join

A good database system should be able to join columns. A join is to combine information in two (or more) tables into one by joining the tables on some shared column. CryptDB supports two cases. The first is the regular Equality Join (EQ-JOIN) between two columns, and the other is Order-Preserving Encryption Join (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.

Popa et al. introduce a new cryptographic primitive for equality joins, Adjustable Join (JOIN-ADJ), which is a deterministic function [24]. 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 protects against the administrator to learn any relations between tables and columns by trying to perform joins on his own. 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 are applicable have to be declared by the application 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 int	EmplNum int	University varchar(255)	Graduated int	Degree varchar(255)	AdjustedGPA* 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 as seen below.

```
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 the onions. There is, for example, no need for maintaining the ADD-onion if the data item is a string type.

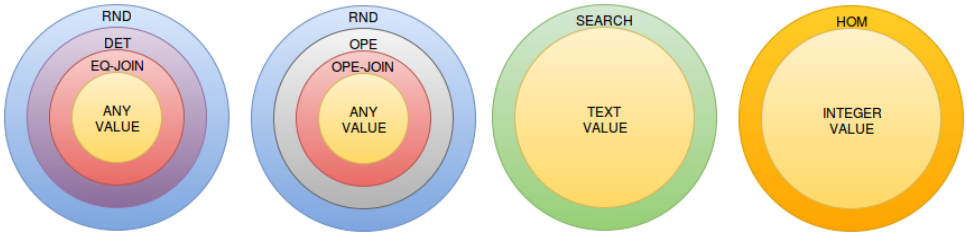


Figure 3.4: Overview of the structure of the SQL-aware encryption scheme. From left: The EQ-onion, ORD-onion, SEARCH-onion and ADD-onion.

3.3 Adjusting the Encryption Level Based on the Query

We have seen that 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. As previously mentioned, the CryptDB proxy server has an embedded table that stores the current encryption layer of all columns. After observing a query, it checks this table 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;
```

Keep in mind that all data items at this point are 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` query. `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.

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 with a mapping between the encrypted column in the database server, and the actual decrypted name of the column. By doing so, it is able 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

```
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.

to start application at the correct layers based on the desired functionality, and the proxy does not perform any encryption level adjustments at all.

3.4 Security

In section 3.1, two different modes for using CryptDB were mentioned. For applications built for one particular user, the single-user mode of CryptDB is used. Examples 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 multiple users with their own usernames and passwords, 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 encryption keys are derived from one secret master key rooted in the application's password (the password to the database). By running in this particular mode, the user has access to all data encrypted through the proxy server. In this mode, both the proxy server and the application server 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 that is 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 presented 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

2.2, 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, which suffer 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.

Annotations are in most languages and frameworks used for describing code and attributes, as well as inducing different behaviours, and are usually parsed at compile time. In CryptDB, the annotations is used when creating database schemas, and then used by the software to perform different types on encryptions based on the annotated attributes. 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 contain sensitive data, and in the next column CryptDB needs to store which principals that should have access to the sensitive column. We see that 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 in database schemas when creating multi-user applications. Annotations are highlighted in red.

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 means that the secret key of the message is encrypted two times, first with the sender's key and then 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 data consists of both a symmetric key and an asymmetric 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 his 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 [6] 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. In the event of an attack on the proxy server in CryptDB, the only parties that are compromised are those who are currently logged in, as their keys are stored at the proxy server and accessible to the attacker.

3.4.3 Securing Applications When Using CryptDB

Another importance related to the use of annotations is 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 [22]. 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 Functional Limitations

FHE is nowhere near being practical yet, but CryptDB tries its 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 an SQL trace from a production MySQL server [24]. However, 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 an integer by the application. One way to cope with the limitations is to let the developer be responsible for transforming 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 the client perform operations on such data types. Obviously neither options are desirable, as it is quite the opposite of the goal when using schemes such as CryptDB.

Floating points are not the only unsupported data types. Data types related to date and time, namely Timestamp, Date, Time, DateTime, Year and NewDate are neither supported, giving us a bit of a headache. 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 date (day, month, year, second, minute, hour and so on) in separate columns, and perform the computations on those columns instead of a composite date column.

Could we possibly used Unix timestamps, which are integers representing the number of seconds since 1st of Januar 1970 (given that the application does not need any dates before the startpoint of Unix time)? By representing dates as integer values, we lose a lot of functionality, as we are only able to compute the order of the dates. As soon as we want to retrieve dates within a specific year, we need to retrieve all data items and perform the operation on the client side, or do a range query for dates between two values. When performing the range query, we need to compute the upper and lower Unix timestamps of our interval. The downside with this approach is that it forces the encryption layer down to OPE, which is the most insecure encryption scheme in CryptDB. By storing the different parts of the date in separate columns, we are able to keep the encryption layer to DET for most date related queries.

While the three-column approach is rather complicated, 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 this point in time. 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 using the `WHERE` clause to bind them together. 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 of 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 necessarily feasible and hence an limitation on the use of CryptDB as a database system.

Chapter 4
Attacking CryptDB

CryptDB is not flawless, and while relying on well-tested cryptographic building blocks, when combining them it does not necessarily guarantee absolute security. In this chapter, a few known attacks on CryptDB will be presented. The first violates the some of the assumptions of CryptDB, but provides nevertheless valuable insight on some of the problems related to such solutions, while the second attack is able to retrieve encrypted information.

4.1 Tampering with CryptDB

Akin and Sunar [12] introduce an attack where a malicious database administrator targets web applications running in multi-user mode. More specifically, software developed using the popular open-source software phpBB [10] where highly customisable forums can be created. By creating a fake user through the web application, and at the same time being 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, the 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 necessarily 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 [24], meaning that the attacks above clearly violates the assumptions of CryptDB with respect to database administrators being passive. As this attack clearly targets data integrity, it is not necessary a weakness in the software itself. Also, as CryptDB does not support the multi-user mode in the recent versions, it is not that relevant, but may help to provide some insight in why the developers have removed this essential part of the software. It also 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 [20]. They did so by using one of the oldest tricks in a crypt analyst's book; frequency analysis.

Frequency analysis is to exploit the fact that in most written 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. Figure 4.1 shows the frequency of the letters in the English language. 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.

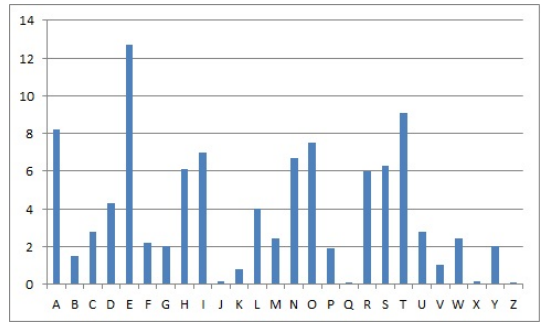


Figure 4.1: Bar chart of the letter frequency observed in the English language.

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 [20]. 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.

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 [20]. 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.

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 [20]. 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.

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 [25] 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.

While Microsoft Research may have used CryptDB in an insecure way, they still address an interesting issue with CryptDB. 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. This clearly breaks one of CryptDB's requirements for safe usage, but this type of situation may force us to break these security requirements in order to produce a somewhat useful system. 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 presentation of a small demo application.

5.1 Installation and setup 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. Since CryptDB is only tested with said Ubuntu versions, the natural approach would be to install the software on one of the two versions. This often involves running some sort of Virtual Machine (VM) in some Virtual Machine Monitor (VMM) such as VirtualBox or VMware, which is very inconvenient when developing applications that should be able to run regardless of operating system and environment. When working inside a VM and installing and using software that is, at best, ready for alpha testing, you better watch your steps. During the installation of CryptDB, some issues have been encountered where the installation is aborted or failed for unknown reasons, before running ever so smoothly on the next attempt.

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 (OS) which is easy to restore if one were to corrupt the state if the system. Docker [3] 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, and runs

directly on the host OS without the need of a VMM and Guest OSes. A *Dockerfile* specifies the environment of the container, where the developer can add instructions on software to be installed, commands to run, as well as external volumes that should be accessible to the container. After the container is built, it can be shipped and run in other environments such as Windows, OS X and Linux without issues or adjustments related to the contents of the container.

In order for Docker to be able to install CryptDB seamlessly, a small change was added to CryptDB's installation script. The `-y` flag was added to the `sudo apt-get install` command which installs dependencies and other necessary libraries for compiling CryptDB. A guide on how to install and set up CryptDB with both database and proxy server can be found in Appendix A.1 along with a small introduction on how to play with the software as presented in Appendix A.2.

5.1.2 Problems Related to Connecting the Proxy Server to the MySQL Server

The CryptDB proxy server listens on port `3307`, intercepts and modifies traffic it receives, and relays it to port `3306` at the proxy backend. One of the issues experienced when setting up CryptDB was related to the hardness of getting the proxy server and the MySQL server connected. For some reason, CryptDB and the build of Ubuntu 12.04 that was used in the beginning of the project, were not that excited in talking to each other using `localhost`.

`Localhost` was, however, suggested by the developers to use in their guide on how to run CryptDB on a singular machine. As the proxy server and the MySQL server were running as independent servers (but not connected), it was quite hard to debug because of the lack of error messages. The system seemed to run properly, but no queries were intercepted by the proxy server. After a lot of fiddling with the configuration settings for both the proxy and the MySQL server, a breakthrough was made when substituting `localhost` with `127.0.0.1`.

Apparently, it seems that specifying `localhost` as host name when using MySQL on Unix systems tells the client to connect to the server using sockets [7]. However, sockets do not seem to work that well with CryptDB, without being able to verify whether or not this is tied to CryptDB itself or the most recent versions of MySQL. By switching to a specific host address such as `127.0.0.1`, the MySQL client is instructed to connect to the port using a TCP/IP connection. When using this approach, the proxy server and the MySQL server finally started to communicate.

5.2 A Small Demo Application

Along with the exploring of CryptDB, a tiny sample application has been developed in order to understand the sytem. The application allows a single user to maintain employee information for a fictive firm. As the software itself only supports applications running in the single-user mode, there has not been made any attempts in creating an application consisting of multiple users. The application is written in Python, which is a high-level programming language with a lot of functionality [11]. In order to add database functionality to the application, MySQLdb is used as an interface to provide the MySQL Application Programming Interface (API) for Python [9]. The application consists of roughly 500 lines of code, and can be found at https://github.com/klevstad/TTM4501-Demo-Application.

5.2.1 Database Server and Proxy Server

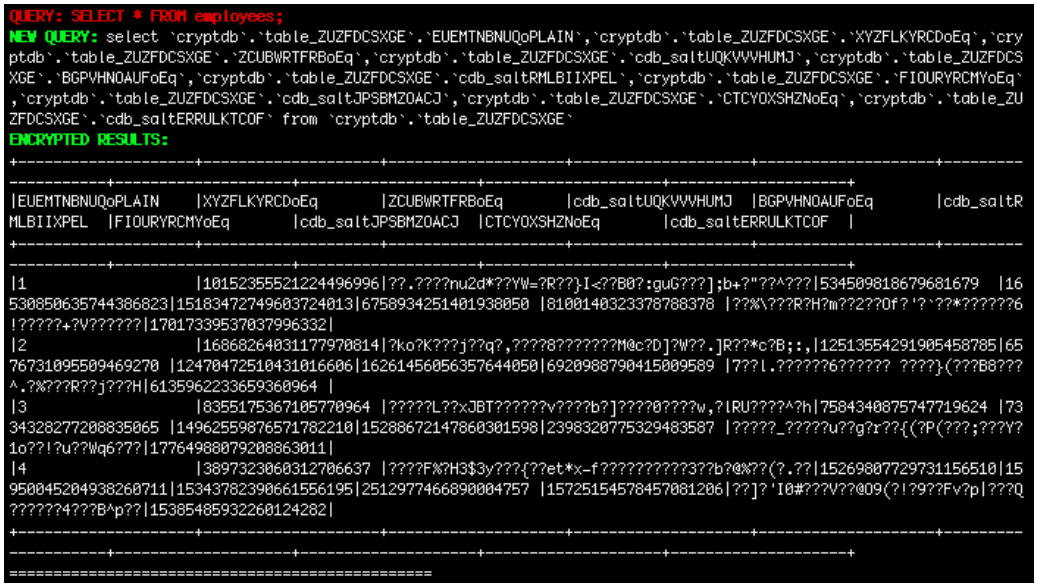
CryptDB needs to have a database running in the background for the proxy server to connect to. As described in the previous section, Docker is used to set up a container hosting both the database and the CryptDB proxy. When connecting to a database, the user needs to specify a few parameters such as the address of the machine hosting the database server, user name, password, port number and the name of the database it wants to connect to. A regular MySQL database server runs on port `3306`, while CryptDB's proxy server is programmed to listen for connections on port `3307` and relay queries to the database server listening on port `3306`. All this is taken care of by following the guide presented in Appendix A.1.

5.2.2 Application for Storing Employee Records

Figure 5.1 shows the main menu when logging into the application. There are not many features implemented in the application, as the whole process of just getting CryptDB's software up and running was somewhat time consuming. The application allows the user to display the list of employees along with their personal records, adding new employees, as well as updating them. It is mainly developed to test out functionality, therefore it also supports "free queries" where the user can perform queries directly to the proxy (or database). For navigating in the application, the user specifies commands by typing in their corresponding number.

When the user selects option 1 in order to show information regarding the employees of the firm, the application sends a `SELECT * FROM employees;` query to the proxy server. As previously explained, the proxy server intercepts the query and anonymizes it. This can be seen in Figure 5.2, which also shows the encrypted result that is returned from the database server. The result is then decrypted and sent to the application, where it is displayed to the user as seen in Figure 5.3.

Figure 5.1: Main menu for users logged in as CryptDB users, i.e. connecting through the proxy server.



EUEMTNBNUQoPLAIN	XYZFLKYRCDoEq	ZCUBWRTFRBoEq	cdb_saltUQKVVVHUMJ	BGPVHNOAUFoEq	cdb_saltRMLBIIXPEL	FIOURYRCMYoEq	cdb_saltJPSBMZOACJ	CTCYOXSHZNoEq	cdb_saltERRULKTCOF
1	101523555212244969960	??.????nu2d*??YW=?R??}I<??B0?:guG???];b+?"??^???|534509818679681679	1653085063574438682	3|151834727496037240	13|675893425140193805	0|810014032337878837	8 |??%\???R?H?m??2??Of?'?`??*??????6!?????+?V??????|17017339537037996332|
2	168682640311779708	14|?ko?K???j??q?,????8???????M@c?D]?W??.]R??*c?B;:,|125135542919054587	85|6576731095509469	270 |12470472510431016	606|162614560563576	44050|692098879041500	9589 |???l.??????6?????? ????}(???B8???^.?%???R??j???H|6135962233659360964 |
3	835517536710577096	4 |?????L??xJBT??????v????b?]????0????w,?lRU????^?h|758434087574771962	4 |73343282772088350	65 |14962559876571782	210|152886721478603	01598|239832077532948	3587 |?????_?????u??g?r??{(?P(???;???Y?1o??!?u??Wq6???|17764988079208863011|
4	389732306031270663	7 |????F%?H3$3y???{??et*x-f???????????3??b?@%??(?.??|152698072973115	6510|159500452049382	60711 |153437823906615	56195|251297746689000	4757 |157251545784570	81206|??]?'I0#???V??@09(?!?9??Fv?p|???Q??????4???B^p??|153854859322601	24282|

Figure 5.2: The observed processing at the proxy server when querying information about all employees.

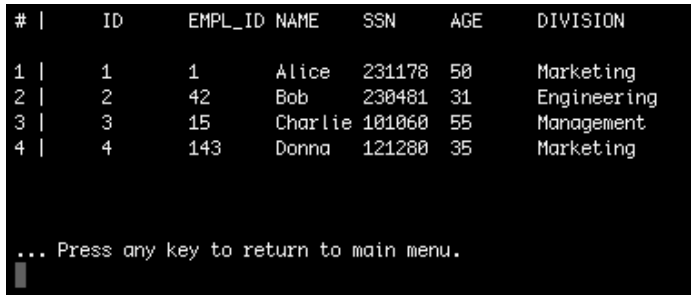


Figure 5.3: The result displayed at the CryptDB user when querying for information regarding all employees.

Figure 5.4: Main menu for administrators, i.e. connecting directly to the database server.

5.2.3 Confirming the Encrypted Administrator View

To illustrate the case of the curious database administrator, which is the most easy-to-demonstrate threat, the application has two views. These are the CryptDB user view, and the database administrator view, which can easily be switched between in the application. Depending on whether the application is connected through the proxy or not, the menu items also changes as shown in Figure 5.1 and Figure 5.4. Such a switch is performed by simply disconnecting from the application and perform a new log-in on either port 3306 or 3307 based on the previous view. The available menu items depends on the current user's role.

For curious database administrators, in addition to performing "free queries", they are also able to view the tables stored in the database, as well as displaying the contents of a table. Remember that one of the key features of CryptDB is that it protects against such curious administrators by encrypting all data inserted in the database.

Figure 5.5: Encrypted tables in CryptDB from a database administrator's perspective.

Figure 5.5 displays the result when querying the database with the `SHOW TABLES;` statement. All table names are encrypted making it difficult for the administrator to achieve any knowledge about the database. One of the features in the administrator's view is to view the contents of a table, which is done by specifying the name of the table. While the data is encrypted, this does not prevent a curious to perform queries on the encrypted data. For example, there is no obstacles for him to perform a query such as

```
SELECT *
FROM table_ZUZFDCSXGE;
```

But, the resulting data is of course encrypted, and nothing more than obscure symbols are returned. Figure 5.6 shows the resulting data to the query above. Note that the only property not to be encrypted is the row id, hence the database administrator is able to find out the number of rows in a table.

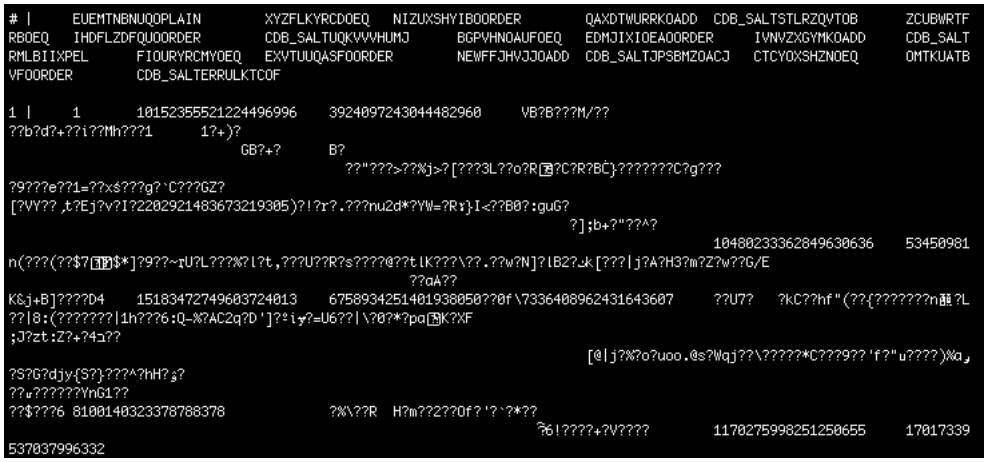


Figure 5.6: Encrypted data observed by the database administrator.

5.2.4 Inspecting CryptDB Using DBMS Tools

Python and MySQLdb does not provide any graphical way to investigate the database server. However, by installing a DBMS environment such as MySQL Workbench [8], inspecting the database suddenly becomes possible. MySQL Workbench provides many different features when inspecting databases, all from performing queries on the tables and describing tables and fields, to monitoring the database. By simply logging in, the administrator is able to view all tables in the database and inspect them by issuing the `SELECT` query mentioned in the previous section.

EUEMTNBNUQoPLAIN	XYZFLKYRCDoEq	NIZUXSHYIBoOrder	QAXDTWURRKoADD	cdb_saltSTLRZQVTOB
1	10152355521224496996	3924097243044482960	BLOB	2202921483673219305
2	16868264031177970814	12491460493456904045	BLOB	3077565006749556258
3	8355175367105770964	6501330947525131331	BLOB	11218496465412421008
4	3897323060312706637	2790297366441961823	BLOB	61630566926686835

Figure 5.7: The resulting rows from performing a SELECT query on the encrypted table. Only the encrypted values of the employee number is shown. Observe that the values are encrypted three times under the different onions Eq, Ord and ADD.

As observed, the data is not displayed as random symbols as seen in the demo application, but rather represented as large integers. It also possible to observe the different onion encryptions of each attribute as they are encrypted as separate columns. By looking very closely at Figure 5.7 above, it is possible to spot the name of the columns, where the EQ-onion ends with *Eq*, the ORD-onion with *Order*

and the ADD-onion with *ADD*. Figure 5.8 shows a clearer picture of the available information when inspecting tables. The database administrator is able to see the number of columns, and what type of onions the column is encrypted under, as well as their raw types.

Field Types

#	Field	Schema	Table	Type	Character Set	Display Size	Precision	Scale
1	EUEMTNBNUQoPLAIN	cryptdb	table_ZUZFDCSXGE	INT UNSIGNED	binary	11	1	0
2	XYZFLKYRCDoEq	cryptdb	table_ZUZFDCSXGE	BIGINT UNSIGNED	binary	20	20	0
3	NIZUXSHYIBoOrder	cryptdb	table_ZUZFDCSXGE	BIGINT UNSIGNED	binary	20	20	0
4	QAXDTWURRKoADD	cryptdb	table_ZUZFDCSXGE	VARBINARY	binary	256	256	0
5	cdb_saltSTLRZQVTOB	cryptdb	table_ZUZFDCSXGE	BIGINT UNSIGNED	binary	8	20	0
6	ZCUBWRTFRBoEq	cryptdb	table_ZUZFDCSXGE	VARBINARY	binary	288	48	0
7	IHDFLZDFQUoOrder	cryptdb	table_ZUZFDCSXGE	BIGINT UNSIGNED	binary	20	20	0
8	cdb_saltUQKVVVHUMJ	cryptdb	table_ZUZFDCSXGE	BIGINT UNSIGNED	binary	8	20	0
9	BGPVHNOAUFoEq	cryptdb	table_ZUZFDCSXGE	BIGINT UNSIGNED	binary	20	20	0
10	EDMJIXIOEAoOrder	cryptdb	table_ZUZFDCSXGE	BIGINT UNSIGNED	binary	20	20	0
11	IVNVZXGYMKoADD	cryptdb	table_ZUZFDCSXGE	VARBINARY	binary	256	256	0
12	cdb_saltRMLBIIXPEL	cryptdb	table_ZUZFDCSXGE	BIGINT UNSIGNED	binary	8	20	0
13	FIOURYRCMYoEq	cryptdb	table_ZUZFDCSXGE	BIGINT UNSIGNED	binary	20	20	0
14	EXVTUUQASFoOrder	cryptdb	table_ZUZFDCSXGE	BIGINT UNSIGNED	binary	20	20	0
15	NEWFFJHVJJoADD	cryptdb	table_ZUZFDCSXGE	VARBINARY	binary	256	256	0
16	cdb_saltJPSBMZOACJ	cryptdb	table_ZUZFDCSXGE	BIGINT UNSIGNED	binary	8	20	0
17	CTCYOXSHZNoEq	cryptdb	table_ZUZFDCSXGE	VARBINARY	binary	288	48	0
18	OMTKUATBVFoOrder	cryptdb	table_ZUZFDCSXGE	BIGINT UNSIGNED	binary	20	20	0
19	cdb_saltERRULKTCOF	cryptdb	table_ZUZFDCSXGE	BIGINT UNSIGNED	binary	8	20	0

Figure 5.8: Overview of the fields in the encrypted table table_ZUZFDCSXGE when inspecting the database using MySQL Workbench.

5.3 Discussion on CryptDB as a Software

While being a bit challenging to install and successfully connect to a database system, CryptDB seems to work as intended when operating on encrypted data. What makes CryptDB especially easy to work with is that it is compatible with MySQL, which is a very established and heavily documented DBMS. This makes it easy and fast to connect CryptDB to existing applications or creating new applications without much knowledge about it. However, the developer should have some insight in what types of operations that are feasible, and how to securely integrate CryptDB without breaking the assumptions and restrictions presented in the recently proposed guidelines [25]. It is clear that CryptDB is not meant for commercial use, both indicated by the lack of documentation and clear guides on implementing applications.

This project has not investigated the overhead when using CryptDB compared to a regular DBMS, but the developers states that the throughput of CryptDB is 21-26 % lower than running on a regular MySQL server [24], which seems modest. However, because of lack of time and access to a suitable dataset, this project has not assessed the benchmarks of CryptDB.

The developers have taken the idea further with Mylar [6], which addresses the threats of attacks on multi-user mode. It is created with the experience from CryptDB in mind, but with focus on running multi-user web applications using the building blocks suggested in CryptDB.

Chapter 6
Comparison and Conclusion

This chapter will cover a comparison of CryptDB with a general FHE scheme, as well as the conclusion of this project. Finally, the impact and future use cases of CryptDB will be pointed out, along with the possibly future work of this project.

6.1 Comparing CryptDB to a Fully Homomorphic Encryption Scheme

6.1.1 Functionality

As discussed in Section 3.5, CryptDB is subject to some limitations when it comes to functionality. While most of these limitations are possible to avoid by using non-standard approaches or "tweaking" standard queries, it is not true that CryptDB supports all types of operations. Because of these limitations, we can state that CryptDB does not provide the functionality of a FHE scheme, where we recall that all possible operations on the encrypted data are supported.

6.1.2 Security

CryptDB is based on well-known encryption schemes where the security is tied to well-studied problems being *hard*. FHE schemes, however, are based on *newer* hard problems that are not as well-studied as the problems described in Section 2.4. For example the Learning With Errors (LWE) problem (introduced in 2005 [26]), which is based on a problem in machine learning, and the Approximate Greatest Common Divisor (GCD) problem (introduced in 1985 [28]) taking advantage of the difficulty of finding the GCD of a number given a bunch of near multiples of the number. One can argue if this makes the security of the encryption schemes used in CryptDB stronger than a FHE scheme, but the cryptography used in CryptDB is more studied than the cryptography used in today's FHE schemes.

6.1.3 Performance

As previously described, one of the biggest weaknesses of FHE schemes in today's world, is their impracticality. Especially performance-wise, a FHE system spends a tremendous amount of time when processing data in a secure manner. In this category, CryptDB is naturally way ahead of a FHE system, as it costs only a reasonable 21-26 % decrease in throughput compared to regular servers running MySQL [24]. As the demo application presented has not been exposed to large dataset, it has not been possible to verify the claims presented in the original paper on CryptDB.

6.2 Conclusion

This project has been about assessing CryptDB, its main features, and how it works. The results have been an overview of the system and its way of utilizing different types of encryption in order to create a somewhat homomorphic encryption scheme. CryptDB provides a neat software, which is easy to interact with after the installation and set-up has been successful. It comes with a brief "how to install and run" guide available in their GitHub repository. However, this guide may to some seem as a bit vague. Therefore, some of the attention of this project has been given to create a small and easy guide for future projects wishing to explore CryptDB to follow. Along with the assessment, a small demonstration application has been created for testing the various queries that CryptDB allows and to observe the behaviour of the proxy server.

CryptDB certainly provides an interesting approach for practical homomorphic encryption schemes utilizing SQL. It supports a large set of possible SQL operations to be performed while both providing confidentiality of the data and being efficient at the same time. However, when seen in connection with the attacks presented in Chapter 4, using CryptDB's approaches may not be the best direction for future implementations of HE schemes. Especially applications intended for multiple users have been proven to either leak information about data encrypted with weaker encryption scheme, or put possibly destructive restrictions on the developer with respect to usefulness.

6.3 Impact and Future Use of CryptDB

CryptDB is the first practical implementation of a somewhat homomorphic encryption scheme using SQL, and has shown a possible approach for creating encrypted databases. A lot of well-known companies such as Google, Microsoft and SAP, as well as a lot of start-ups, are using CrytpDB's building blocks and putting its ideas into their commercial software [2]. As CryptDB is enabled exclusively for single-user

applications in the nearest future, it is difficult to point out useful applications involving highly sensitive data. The possible weaknesses of CryptDB presented in Chapter 4 may indicate that the authors have made the right call when removing the multi-user possibility from CryptDB and recreated its ideas under the new project Mylar [6].

However, CryptDB seems to be a well-suited approach for creating diaries, contact lists, and other applications targeted for a single user. One might argue that it may be more convenient for the developer of such applications to use strong encryption, decrypt the queried data at the client side and perform operations on the plaintext. However, leaving sensitive information in plaintext for just a second, gives an attacker a possible window for stealing it. The single-user mode may also be a solution for developers not being able to provide their own servers. By outsourcing the hosting of the application to a cloud service, the developer is able to easily communicate with his application in a confidential and secure manner.

It would have been interesting to create a more comprehensive system and run tests with a large dataset to verify exactly how efficient CryptDB is. Unfortunately, because of the limited amount of time, this is yet to be done. It would also have been interesting to try to recreate the attack proposed by Microsoft Research, if not on a multi-user mode application, then perhaps on a single-user application.

References

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

[2] CryptDB - MIT. https://css.csail.mit.edu/cryptdb/. Accessed: 2015-12-11.

[3] Docker. https://docker.com/. Accessed: 2015-12-10.

[4] Integrated Data for Analysis, Anonymization, and SHaring (iDASH). https://idash.ucsd.edu/. Accessed: 2015-10-04.

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

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

[7] MySQL 5.7 Reference Manual. http://dev.mysql.com/doc/refman/5.7/en/can-not-connect-to-server.html. Accessed: 2015-12-10.

[8] MySQL Workbench. https://dev.mysql.com/downloads/workbench/5.2.html. Accessed: 2015-12-10.

[9] MySQLdb. http://mysql-python.sourceforge.net/MySQLdb.html. Accessed: 2015-12-10.

[10] phpBB. https://phpbb.com/. Accessed: 2015-12-10.

[11] Python. https://www.python.org/. Accessed: 2015-12-10.

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

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

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

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

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

[17] 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.

[18] Gentry, C. Computing Arbitrary Functions of Encrypted Data. *Commun. ACM* (2010), pp. 97–105.

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

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

[26] Regev, O. On Lattices, Learning with Errors, Random Linear Codes, and Cryptography. In *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing* (2005), ACM, pp. 84–93.

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

[28] Schönhage, A. Quasi-GCD Computations. *J. Complexity* (1985), 118–137.

[29] Vaikuntanathan, V. Computing Blindfolded: New Developments in Fully Homomorphic Encryption. In *Foundations of Computer Science (FOCS), 2011 IEEE 52nd Annual Symposium on* (2011), pp. 5–16.

Appendix A

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.