Exploring CryptDB:
A Practical HE Scheme for SQL Queries

Eirik Klevstad

Abstract

This is an awesome abstract.

Preface

This is a silly preface.

Contents

List of Figures

List of Tables

List of Acronyms

AES Advanced Encryption Standard.

CBC Cipher-Block-Chaining.

DET Deterministic.

FHE Fully Homomorphic Encryption.

HOM Homomorphic Encryption Onion.

JOIN-ADJ Adjustable Join.

mOPE Mutable Order-Preserving Encoding.

OPE Order-Preserving Encryption.

PHE Practical Homomorphic Encryption.

PKE Public Key Encryption.

PRP Pseudo-Random-Permutation.

RND Random.

SQL Structured Query Language.

UDFs User-Defined Functions.

Introduction

1.1 Motivation

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

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

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

In short terms, homomorphic encryption is a cryptographic property describing the ability to perform certain operations on encrypted data (ciphertexts) without decrypting it first. Fully homomorphic encryption is the enhanced version where the encryption scheme is capable of performing all efficient functions [8]. While still in research mode, fully homomorphic encryption schemes' biggest challenge is efficiency. As for practicality, fully homomorphic encryption schemes still have a long way to go [11].

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

Maybe we can achieve FHE in another way?

For now: What is practically possible?

1.2   Problem

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

1.3   Scope

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

1.4   Related Work

To be continued.

Background

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

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

2.1 Symmetric Encryption

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

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

2.2 Asymmetric Encryption

In contrast to symmetric key encryption, asymmetric key encryption does not depend on a pre-shared secret key. Asymmetric key encryption, or Public-Key Encryption, is based upon the fact that some mathematical problems are considered *hard*, such as the integer factorization, discrete logarithms and elliptic curves. By computing a key-

pair consisting of a public key and a private key, two users are able to exchange keys over a public channel without worrying about their secret keys being compromised. The public key is used for encrypting data sent to the user, and the private key is used to decrypt received data that is encrypted with said public key. Two of the most recognized public-key cryptosystems are Rivest - Shamir - Adleman (RSA), which relies of the integer factorization problem, and El Gamal, which relies on the discrete logarithms problem. In addition to secure communication between multiple parties, public key cryptography is also applied to create digital signatures, which provides authentication and data integrity.

2.3   Homomorphic Encryption

We love to describe encryption as safes where we store our data, then secures it with one or more locks, and hide the secret key. Without the secret key, the data is securely stored inside the safe. Whenever we need our data, we take the hidden key out from its hideout and opens all the locks of the safe, where the data is as intact as we left it. The, perhaps, holiest grail of all the holy grails in cryptography is called *homomorphic encryption.* This is a special case of encryption where operations on the encrypted data is 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.

Regular Public Key Encryption (PKE) schemes consists of three algorithms, namely a key generation algorithm (`KeyGen`), an encryption algorithm (`Enc`) and a decryption algorithm (`Dec`). Homomorphic Encryption (HE) schemes adds another algorithm to the toolbox, an evaluation algorithm (`Eval`), where

Theorem 2.1. [6] *Eval, given a well-formed public key pk, a boolean circuit C with fan-in of size t and well-formed ciphertexts $c_1, ..., c_l$ encrypting $m_1, ..., m_l$ respectively, outputs a ciphertext c such that $Dec_{sk}(c) = C(m_1, ..., m_l)$.*

For Practical Homomorphic Encryption (PHE) schemes, *Eval* will be associated to a set of permitted functions f. These are functions that the algorithm can handle, and which guarantee a meaningful result when executed. These functions can be expressed as circuits consisting of logical gates such as AND, OR and NOT. Gentry [10] presented a homomorphic encryption scheme consisting of three functions; addition (Add_ϵ), subtraction (Sub_ϵ) and multiplication ($Mult_\epsilon$). When performing homomorphic operations using functions from f, an N-bit noise is generated and added to the encrypted ciphertext, making the relation between the encrypted result and its corresponding plaintext weaker. By performing multiple operations on the ciphertexts, the noise grows larger. Problems arise if the noisy part gets large enough, as the decryption algorithm might not be able to decrypt in a reliable way in order to obtain the correct result.

2.4   Fully Homomorphic Encryption

Fully Homomorphic Encryption (FHE), which has no restrictions to what types of operations that can be performed on the encrypted data, was first suggested in 1978 by Rivest, Adleman, and Dertouzos [15]. 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 [9], and a year later another scheme using a *bootstrappable* approach based on asymmetric encryption [10]. Nearly all modern FHE schemes are based on this bootstrappable concept using Gentry's blueprints.

As mentioned in the previous section, a homomorphic operation creates noise when executed. Multiple operations adds more noise which may change the decrypted result beyond the recognizable. But what if we had a scheme that was able to reduce the noise generated by such operations?

By using an approach where decryption keys are bootstrapped to

When allowing the encryption function to handle its own decryption at the same time, the modified algorithm encrypts a message under two different public keys, Gentry showed that the noise added by the homomorphic operations was less than the noise removed by the additional decryption, and the breakthrough was made in hunt for a fully homomorphic encryption scheme. The modified algorithm $Recrypt(pk_{i+1}, D_\epsilon, \overline{sk_i}, c_i)$ consists of two operations.

$$\overline{c_i} = Enc(pk_{i+1}, c_i)$$

and

$$c = Eval(pk_{i+1}, D_\epsilon, \overline{sk_i}, \overline{c_i})$$

encrypts both the ciphertext c_i and the secret key sk_i under the new public key pk_{i+1}, D_ϵ is a decryption function

before it uses the *Eval* algorithm to decrypt the inner ciphertext using the provided secret key sk_i.

What is the point of an encryption algorithm to be able to decrypt at the same time? Well, this somewhat untraditional way of encrypting something holds an interesting property.

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

By recursively encrypting the secret keys, a chain of encrypted keys are created such that sk_i is encrypted under pk_{i+1}.

To give this chain a bit more context, imagine a set of boxes encrypted with a

Assume a secret key sk_1 and a plaintext message m.

$$c_1 = Enc(sk_1, m)$$

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

After the the recent breakthroughs in homomorphic encryption, mainly by Gentry [9] [8] [10], most of the HE schemes follow the basic ideas, handling noisy ciphertexts and using the bootstrappable approach in order to obtain FHE.

While Gentry's system is great in theory, it has been shown that creating FHE schemes is practical hard. However, some other cryptographic schemes holds different types of homomorphic properties which when combined can provide different homomorphic operations. Addition is, for example, the homomorphic property of the Pailler cryptosystem [12]. Pailler is originally a trapdoor mechanism based on the Composite Residuosity Class Problem which conveniently has the cryptographic property such that

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

Another practical cryptosystem providing a similar property, is the ElGamal cryptosystem. While originally being a cryptosystem used for encryption, it has the cryptographic property shown below, which enables it to perform homomorphic multiplications if used correctly.

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

2.5   El Gamal

By taking the advantage of the discrete logarithms problem, Taher El Gamal described a cryptosystem [7] based on the Diffie-Hellman key exchange. Hmm.

2.6   Searchable Encryption

2.7   Database Security

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

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

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

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

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

Overview of CryptDB

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

3.1 System Architecture

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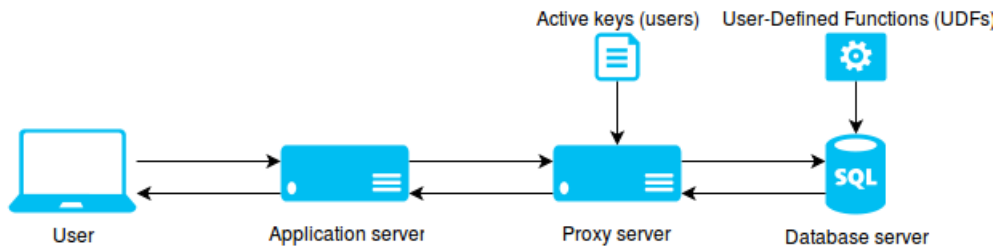

Figure 3.1: System architecture of CryptDB interacting with a client application

Another vital part of the proxy server's domain is to issue queries to the database

that adjusts the encryption layer (or onion layer) of the data items(s) that the user has issued a query on, which will be discussed in section 3.3. The last responsibility of the proxy server is to keep track of the users that are currently logged in using a table of active keys. 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.

In addition to keeping the list of active keys, the proxy server also keeps an embedded database with the schemes of the encrypted tables in the database. By storing this, the proxy server keeps a continuous picture of the encryption layer for each column, which is used for adjusting the encryption level (or layer) for a column in order to allow certain types of functionality.

The authors [13] 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.

3.1.1   Single-user Mode

If an application is created with a single user in mind, then it should be running in the single-user mode where the keys are derived from one secret master key which is rooted in the user's password. By running in this particular mode, the user has access to all data encrypted through the proxy server.

Proxy server and application considered trusted

3.1.2   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 2.7, 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.

In CryptDB, annotations are used when creating tables to specify which entities that should have access to the data items, as well as allowing entities to perform operations on behalf of user within its group.

```
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) );
```

Table 3.1: Use of policy annotations when creating multi-user applications

Table 3.1 shows a basic use of such annotations. `PRINCTYPE` is used to annotate external users, which are authenticated through the access control by providing their password, and internal users, which are entities inside the database. `ENC_FOR` is used to indicate which columns that consist sensitive data, and in the next column CryptDB needs to store which principals that should have access to the sensitive column.

How can a user read messages on the discussion board if they are encrypted by another user's keys? What about private messages, with suffers from the same case, as they will be encrypted with the sender's key? CryptDB solves these cases by letting developers use annotations in their schemas.

In multi-user mode the proxy server holds an encrypted table of the keys of all users.

Database server, Proxy server and application considered untrusted.

3.1.3   Something more?

As of the most recent versions of the CryptDB software, only single-user mode is supported. One may react to the fact that the developers has stopped maintaining such a quite regular and useful way of developing applications with multiple user's and interaction between users. Mylar [4] is the team that is behind CryptDB's new project, where they have taken some of CryptDB's best features and created a

Todo:
Annotations of SQL schemas.
Key-chaining. Speaks for. Encrypts for.

3.1.4   Something more?

As of the most recent versions of the CryptDB software, only single-user mode is supported. One may react to the fact that the developers has stopped maintaining such a quite regular and useful way of developing applications with multiple user's and interaction between users. Mylar [4] is the team that is behind CryptDB's new project, where they have taken some of CryptDB's best features and created a

   Todo:
Annotations of SQL schemas.
Key-chaining. Speaks for. Encrypts for.

3.2   SQL-Aware Encryption and the Onion Scheme

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

   As a scenario in order to describe the different operations and situations in CryptDB, assume a simple employee application with a table structure and some example records as shown in Table 3.2. Our application is created in order to provide some insight in the salary distribution within our firm, with respect to age, number of years within the firm and employment division. EmplNum is the number associated with the employee in other systems and applications, and SSN is the Social Security Number of the employee used for creating pay checks and similar operations.

Id	EmplNum	Name	SSN	Age	Salary	Division
int	int	varchar(255)	varchar(255)	int	int	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 3.2: Employee_table for a simple employee application with example records

3.2.1   Random

Random (RND) is the highest security level in CryptDB and provides the maximum security found in encryption scheme. It uses a strong block cipher such as Blowfish or Advanced Encryption Standard (AES) in Cipher-Block-Chaining (CBC) mode and a random initialization vector (IV) to ensure that the block cipher is probabilistic [13]. RND, being the maximum security level provided, does not allow any computation to be done on the encrypted data. In other terms, this level is a natural choice for sensitive data that are only meant to be read. When a block cipher is probabilistic, it has properties such that when encrypting the same message m_1 multiple times, the resulting ciphertexts are unequal.

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

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

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

```
SELECT * FROM employee_table;

UPDATE employee_table SET salary = 0;

DELETE FROM employee_table;
```

3.2.2   Deterministic

Where RND allows no computation to be done on the encrypted data, the next layer does. Deterministic (DET) is an encryption scheme enabling the application to perform standard SQL operations such as equality checks, distinct, group by and

count. By allowing these sorts of computation, the application leaks information to an adversary. In particular, it leaks which ciphertexts that decrypts to the same plaintext value. Following the previous example; if the scheme encrypted the message m_1 two times, the resulting ciphertexts c_1 and c_2 are such that $c_1 = c_2$. For our sample application, DET would be the scheme used when allowing operations such as `GROUP BY`, `DISTINCT` and `COUNT`. Continuing with our sample application, we now want to find out what types of departments that exists in our table

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

which returns `'Engineering'`, `'Marketing'` and `'Management'`. As described, a deterministic scheme encrypts the same plaintext to the same ciphertext every time. By taking advantage of this, CryptDB is capable of iterating over the encrypted data and return a distinct selection of the different ciphertexts observed. 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 [13]. In order to cope with leaking prefix equality, the authors have designed their own version of the CMC mode.

3.2.3   Order-Preserving Encoding

Since Structured Query Language (SQL) also allows the user to compute on order relations between items, CryptDB introduces an Order-Preserving Encryption (OPE) scheme [13]. 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. This common operation in SQL supports order comparison, which can be used for range checks, ranking, sorting, and extracting minimum and maximum values.

CryptDB uses a modified version the scheme proposed by Boldyreva et. al [5] 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. It uses a Hyper Geometric Distribution (HGD), and uses a lazily sampling to generate ciphertext elements from the set on demand.

Popa et al. has also proposed

The OPE layer is the encryption layer that adds the most functionality When searching for the persons in our imaginary firm that has a yearly salary over 500 000,

the query uses the `WHERE` clause and then performs an order check where it navigates down the binary three for items that has a salary larger than the requested value.

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

3.2.4 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 [12] for enabling summation operations. If our application is in need for finding out the total sum of the salaries within the firm, we do something like this

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

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

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

Multiplication was not initially implemented, but has been implemented in some versions of CryptDB by an outside team using the El Gamal cryptosystem [14].

3.2.5 Equality Join and Order-Preserving Join

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

The authors [13] introduces a new cryptographic primitive for equality joins, Adjustable Join (JOIN-ADJ), which is a deterministic function. The idea is to

let the proxy server adjust the encryption keys of columns in real-time, based on the observed query. By using this approach, CryptDB avoids the database server to compute joins on its own as attempt to learn about relations between different columns. When a join query is observed at the proxy, it sends an adjusted key to the database server enabling it to adjust the values in one of the two affected columns. When an adjustment has been done, the columns share the key until the proxy server issues a new adjustment key to either one of the columns.

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

Example here: join on empl_num

3.2.6   Search

In order to perform search for words in the encrypted texts, CryptDB has a scheme called SEARCH. This was an implementation of the cryptographic protocol suggested by Song et al. making it almost as secure as the RND scheme [13]. The main idea is to split a text encrypted with SEARCH into individual keywords on a given delimiter specified by the application developer. Removal of duplicate keywords and randomly permute them are also added to enhance the security. When executing a search, the server would be given an encrypted token of the keyword, and would retrieve encrypted values matching the token. Originally, SEARCH could perform `LIKE` operations as other database systems, but in the most recent version of CryptDB's software it has been deprecated.

3.2.7   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 3.2. Each onion has a special purpose in means of supporting certain operations, and every data item is encrypted each of the different onions using the different layers. However, the application does not necessary maintain all of onions. There is, for example, no need for maintaining the ADD-onion if the data item is a string.
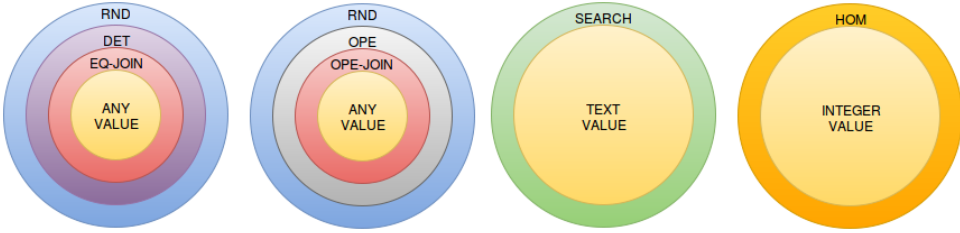
Figure 3.2: Overview of the structure of the SQL-aware Encryption Scheme. From left: The EQ-onion, ORD-onion, SEARCH-onion and ADD-onion

3.3 Adjusting the Encryption Level Based on the Query

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

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

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

Have in mind that all data items at this point is encrypted with RND as the utmost layer. What the proxy does, is that before sending the rewritten query to the database server, it sends an `UPDATE` query first. This query orders the database server to remove the outer encryption level of the ORD-onion to match the OPE layer, before it sends the rewritten query. Figure 3.3 illustrates how the proxy server observes the query, and issues the `UPDATE`. `SET` is used to change the current encryption level of the ORD-onion, and `cryptdb_decrypt_int_sem` is the name of the UDF that the server will use to adjust the encryption layer to support the query. The curious string `'???]???G??+?,?#'` is the encrypted key which the server will use along with the UDF.

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

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

Figure 3.3: Encryption layer adjustment performed by proxy server upon receiving a query. (Is the figure too small/hard to read?. Adjust colors?)

After performing the encryption layer adjustment, the proxy rewrites the query as shown in Figure 3.4. As described, each of the columns are anonymized upon the creation of a table. The proxy server keeps table schemes in order to substitute the different parts of the query so it matches the encrypted columns in the database. For the record, 'test' is the name of the current database.

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

3.4   Security

Overview of the security aspects of CryptDB

3.5   Limitations

Data structures that are not supported:

Float, Double, Timestamp, Date, Time, DateTime, Year, Newdate, Bit, Enum, Set, Geometry

Nested queries

On the hack queries and other things that simply does not work with CryptDB.

Attacks

4.1 Microsoft and their Frequency Analysis

The developers of CryptDB have released their "counter-answer" to the Microsoft Research along with a paper on how to safely use CryptDB. More to come. My next paper to read.

CryptDB as a Software

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

5.1 Installation of CryptDB's Software

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

5.1.1 Docker to the Rescue

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

can be shipped and run in other environments such as Windows, OS X and Linux without issues.

A thorough guide for installing CryptDB can be found at https://github.com/klevstad/CryptDB_Docker

5.2   A Small Demo Application

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

5.3   Discussion of CryptDB as a Software

Should I compare CryptDB to another HE-database. Perhaps MuteDB ? http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6977940

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

Chapter 6
Conclusion

6.1   Impact of CryptDB

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

6.2   Useful applications

Suggestions

6.3   Conclusion

Some conclusion.

References

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

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

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

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

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

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

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

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

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] POPA, R. A., ZELDOVICH, N., AND BALAKRISHNAN, H. Guidelines for Using the CryptDB System Securely. Cryptology ePrint Archive, Report 2015/979, 2015.

[15] RIVEST, R. L., ADLEMAN, L., AND DERTOUZOS, M. L. On Data Banks and Privacy Homomorphisms. *Foundations of Secure Computation* (1978), pp. 169–180.