NTNU – Trondheim
Norwegian University of
Science and Technology

Exploring CryptDB: A Practical HE Scheme for SQL Queries

Eirik Klevstad

Submission date:	October 2015
Responsible professor:	Colin Boyd, ITEM
Supervisor:	Chris Carr, ITEM

Norwegian University of Science and Technology
Department of Telematics

Abstract

This is an awesome abstract.

Preface

This is a silly preface.

Contents

List of Figures

List of Acronyms

AES	Advanced Encryption Standard.
CBC	Cipher-Block-Chaining.
DET	Deterministic.
FHE	Fully Homomorphic Encryption.
HOM	Homomorphic Encryption.
JOIN-ADJ	Adjustable Join.
mOPE	Mutable Order-Preserving Encoding.
OPE	Order-Preserving Encryption.
PKE	Public Key Encryption.
PRP	Pseudo-Random-Permutation.
RND	Random.
SQL	Structured Query Language.
UDFs	User-Defined Functions.

Chapter 1

Introduction

1.1 Motivation

Cloud services are becoming larger and more complex. Users want their content available wherever they are, forcing applications to store content in the cloud. Companies such as Apple [1], Microsoft [3] and other corporations are looking into health information and how your personal information can be integrated into their services. iDASH [2] leads on into combining biomedical research and technology. Medical research facilities stores tremendous amounts of personal data, and 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. Another downside with this approach is that it might be inefficient as the traffic in the data center is likely to fluctuate, making some racks standing "cold" from time to time. 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, stored at a possible distrustful provider, is protected and that they will not snoop in our database?

The first solution that comes to our mind is that the user could encrypt its data with a strong block cipher such as AES and a 256-bit key, 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 [5]. 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 [9].

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

1.2 Problem

Introduction to the problem.
The aim of the project. Short explanation of results

1.3 Scope

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

1.4 Related Work

To be continued.

Chapter 2 Background

We love to describe encryption as safes where we store our data, secures it with one or more locks, and hides the secret key. Without the secret key, our data is securely stored inside the safe. Whenever we need our data, we take our 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.

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 [14]. 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 using lattices [6]. With cloud computing and BLABLA MER OM HVORFOR TING HAR TATT SEG OPP I DET SISTE.

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`). FHE schemes adds another algorithm to the toolbox, an evaluation algorithm (`Eval`), where

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 $Dec_{sk}(c) = C(m_1, ..., m_l)$. [4]

[7].

Chapter 3 Overview of CryptDB

3.1 System Architecture

CrytpDB's architecture (Figure 3.1) is divided into two pieces, more specific 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. Its purpose is to intercept queries going from the application to the database and anonymize the information of the query. By doing so, it eliminates the possibility of eavesdropping attackers to obtain any sensible information. 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 column(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.

The authors [12] 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 and are created using the `CREATE FUNCTION` command of Structured Query Language (SQL). It also holds an encrypted table of all user-keys when in multi-user mode.

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

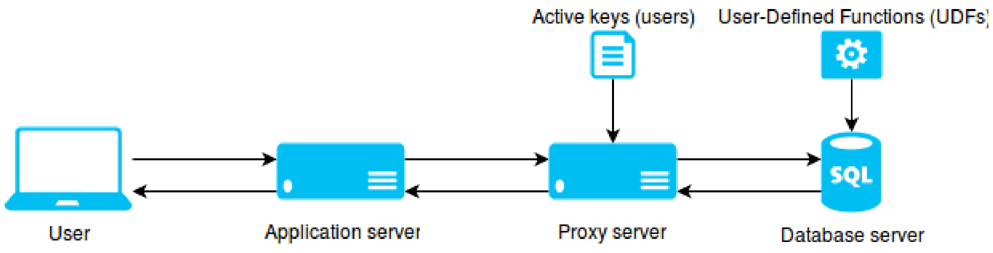

Figure 3.1: System architecture of CryptDB interacting with a client application

3.2 SQL-Aware Encryption and the Onion Scheme

CryptDB uses an encryption scheme called *SQL-aware encryption* or *onion encryption*. Basically, this means that the encryption scheme is a collection of different 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 inferior layers provide less security, but more functionality.

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 [12]. 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`.

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 $c_1 = c_2$. DET is a deterministic scheme which, to be used correctly, should be a Pseudo-Random-Permutation (PRP). In order to cope with leaking prefix equality, the authors have designed their own version of the CMC mode [12].

3.2.3 Order-Preserving Encoding

Since SQL also allows the user to compute on order relations between items, CryptDB introduces an Order-Preserving Encryption (OPE) scheme [12]. This scheme takes starting point in 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. Popa et al. [11] introduces a scheme called Mutable Order-Preserving Encoding (mOPE) where the main technique is mutable ciphertexts. Mutability (or malleability) is a cryptographic property allowing transformation of one ciphertext into another. More formally, given a plaintext m_1, it is possible to transform it into a valid encryption $f(m_1)$ by using a valid function f without further knowledge of m_1.

MOPE uses a balanced binary tree for searching which contains encryptions of all the plaintext's values and a table of OPE encodings where the encoded value of a ciphertext is its path from the root to the particular node in the binary tree. If y is greater than x, then y will be located to the right of x in the tree. It also uses an interactive protocol when inserting a value into the search tree, where the server and the client plays a query game. The client asks for the encrypted root node, decrypts it and determines whether the value to be inserted is smaller or larger than the root. It replies to the server with a 0 or 1, making the server traverse the tree to the left or right and replying with the next encrypted node. This little game continues until there are no child nodes. In order to avoid the tree being too deep, MOPE uses a tree-balancing transformation summary which describes operations (most of them split-and-merge operations) that are completely done by the server.

The balancing procedure ensures that the height of the tree is bounded by $\log(N)$, where N is the number of encrypted values. Because both client and server needs a shared understanding of the structure of the search tree, the client computes a Merkle hash of the root of the tree. A Merkle tree is a tree where every parent node is labelled with the hash of all of its children's labels [8]. By performing the

Merkle hash, the client is able to check whether the server behaves malicious or not (Malicious in this setting would be if the server performed other operations than the client requested) by comparing its own Merkle hash of the root with the one delivered by the server.

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 (HOM) scheme. As previously described, homomorphic encryption is a technique that enables computation on encrypted data without decrypting it first. CryptDB uses Pailler multiplication [10] for enabling summation operations. Pailler is originally a trapdoor mechanism based on the Composite Residuosity Class Problem which conveniently has a cryptographic property such that $HOM_k(x) * HOM_k(y) = HOM_k(x + y)$. Multiplication was not initially implemented, but has been added in later versions of CryptDB by an outside team using the El Gamal cryptosystem [13].

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 [12] 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.

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 [12]. 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.

These different layers are wrapped into four different onion classes, 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.

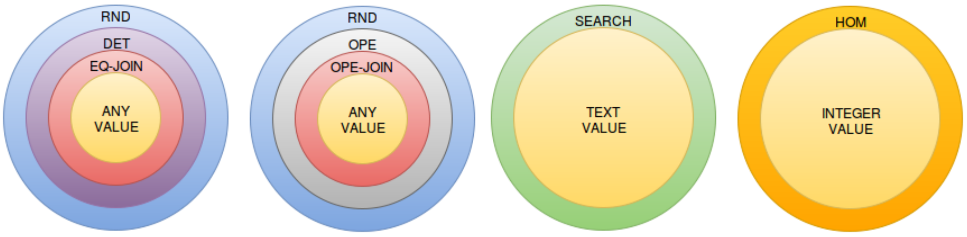

Figure 3.2: Overview of the structure of the onions. From left: The EQ-onion, ORD-onion, SEARCH-onion and ADD-onion

3.3 Adjusting the Encryption Level Based on the Query

A story about a spy, a castle and some walls.

3.4 Threats

3.5 Security

3.6 Limitations

Chapter 4 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.

References

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

[2] Integrated data for analysis, anonymization, and sharing (idash). https://idash.ucsd.edu/. Accessed: 2015-10-04.

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

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

[5] Gentry, C. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford, CA, USA, 2009. AAI3382729.

[6] Gentry, C. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 2009), STOC '09, ACM, pp. 169–178.

[7] Gentry, C. Computing arbitrary functions of encrypted data. *Commun. ACM 53*, 3 (Mar. 2010), pp. 97–105.

[8] Merkle, R. A digital signature based on a conventional encryption function. In *Advances in Cryptology — CRYPTO '87*, C. Pomerance, Ed., vol. 293 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1988, pp. 369–378.

[9] 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), ACM, pp. 113–124.

[10] 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* (Berlin, Heidelberg, 1999), EUROCRYPT'99, Springer-Verlag, pp. 223–238.

[11] 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* (Washington, DC, USA, 2013), SP '13, IEEE Computer Society, pp. 463–477.

[12] 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* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 85–100.

[13] Popa, R. A., Zeldovich, N., and Balakrishnan, H. Guidelines for using the cryptdb system securely. Cryptology ePrint Archive, Report 2015/979, 2015.

[14] Rivest, R. L., Adleman, L., and Dertouzos, M. L. On data banks and privacy homomorphisms. *Foundations of secure computation* (1978), pp. 169–180.