QUERYING OVER ENCRYPTED DATABASES IN A CLOUD ENVIRONMENT

by

Jake Douglas

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

May 2019

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the thesis submitted by

Jake Douglas

Thesis Title: Querying Over Encrypted Databases in a Cloud Environment

Date of Final Oral Examination: 8th March 2019

The following individuals read and discussed the thesis submitted by student Jake Douglas, and they evaluated the presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

Jyh-haw Yeh, Ph.D.	Chair, Supervisory Committee
Yantian Hou, Ph.D.	Member, Supervisory Committee
Min Long, Ph.D.	Member, Supervisory Committee

The final reading approval of the thesis was granted by Jyh-haw Yeh, Ph.D., Chair of the Supervisory Committee. The thesis was approved by the Graduate College.

ABSTRACT

The adoption of cloud computing has created a huge shift in where data is processed and stored. Increasingly, organizations opt to store their data outside of their own network to gain the benefits offered by shared cloud resources. With these benefits also come risks; namely, another organization has access to all of the data. A malicious insider at the cloud services provider could steal any personal information contained on the cloud or could use the data for the cloud service provider's business advantage. By encrypting the data, some of these risks can be mitigated. Unfortunately, encrypting the data also means that some commonly used operations, such as equality testing or search, do not work because encryption also obfuscates these properties.

This thesis proposes a system that allows for data to be encrypted with a minimal impact on data accessibility and usability in its encrypted format. This is achieved by carefully selecting the encryption methods used with the goal of preserving properties of the data that are required for the SQL server's functionality. By preserving only order, equality, and the ability to perform addition, common data operations can still be performed. The system was implemented in Java as a proof-of-concept to show that the encrypted data is still operable on, and to compare it to existing systems. The impact from implementing this system on the database size, query encryption and decryption time, and data security is measured and compared to a similar system, showing that it is feasible for use.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Data storage in a cloud environment is extremely popular and offers benefits to organizations of all sizes. Gartner, a research group, predicts that global public cloud revenue will grow 17.3% to $206.2 billion in 2019 [9]. Any amount of data stored by an organization, and the services that rely on that data, can take advantage of the security, reliability, and scalability provided by cloud services. For example, smaller organizations that host online content often won't have dedicated security engineers or experienced server technicians, which can lead to vulnerabilities. Misconfigured servers were number six in the Open Web Application Security Project (OWASP) Top 10 vulnerabilities in 2017 [40]. Even large organizations often lack the required security workforce: the 2017 Global Information Security Workforce Study by Frost & Sullivan, a business consulting firm, revealed that 66% of 19,175 respondents said that there were "too few information security workers in my department" [7]. Outsourcing a database to the cloud allows for at least some of the risk to be mitigated since the cloud service provider will have security engineers on staff, reducing the possibility of a data breach by an outside attacker.

Additionally, hosting data in a cloud environment also allows for improved reliability and scalability in the event of a power outage or natural disaster. Major cloud providers have data centers in multiple locations across the country or world, such as

Azure's geo-redundant storage [8] or Amazon Web Service's Disaster Recovery plan [43]. During normal day-to-day operation, data centers are capable of handling large amounts of network traffic. Some businesses, especially small and home businesses, may not have the capacity to process large amounts of data on their own network. They also allow for quicker scaling since resources can be almost instantly expanded or reduced based on current traffic. For a business-critical application, a long downtime caused by an influx of new customers or small bandwidth could result in lost profits and reduced competitiveness. For large data sets, an important consideration is that cloud storage is much cheaper than physical storage. For this reason, many large data sets are outsourced to the cloud.

Using these services does not come without risk; while attacks from outsiders might be mitigated, the data owner is trusting another entity with their data. LogicMonitor, a cloud configuration platform, surveyed 283 professionals and found that 66% reported security as a major challenge in cloud adoption [3]. In some cases, such as storing personal information or credit card data, it might be unwise. In others, such as secret government data or certain types of personal information, it could be illegal. Encrypting the data allows users to gain the benefits of cloud service without exposing their data to potentially malicious actors. However, encrypting data without regard to how it will be processed causes some problems with the usability of the data: encryption destroys certain properties of data. For example, encrypting multiple messages using the RSA encryption scheme means that the messages can't be sorted or summed, but can still be multiplied [42]. Encryption using AES means that all properties of the data are destroyed, rendering it equivalent to a random number in usefulness.

1.1 Objective of this work

By leveraging the properties that **are** preserved after certain types of encryption, we aim to build a database encryption scheme which allows for operation on the data while it is still encrypted. The goal is to create a system that is transparent to the database server which also provides confidentiality using a minimum amount of additional resources. In this context, transparency means that the database management server (DBMS) does not need to be modified for the software to function. This is important to allow the system to be used on a commercially available public cloud database system.

1.2 Motivation

The use case scenario for this thesis is a database system used by only one organization. The organization might want to put their data on the cloud for a variety of reasons, including low start-up costs, quick scaling, or geographic redundancy. A few possible scenarios could be:

- **Email server**: Emails may contain sensitive company or personal information
- **Web shop back end**: Customers' credit card and address information are likely stored
- **Personal data in a smart city**: Some information might be applicable to group computations (weather, traffic volume), but other information (license plate numbers, facial recognition data) should not

Email servers and web shops are both areas where the high availability afforded by cloud systems is crucial and where sensitive data is likely being stored and processed.

Downtime for a web shop directly leads to lost sales, especially when another provider is likely a few clicks away. In a business setting, a lack of access to email can lead to a loss in productivity.

Smart city systems is a growing area where sensitive private data is being stored on cloud servers. In a smart city, data is collected from various sensors including surveillance cameras, traffic congestion sensors, and fire detection sensors. Since they are often running all the time, these sensors can generate a large amount of data that might become costly for a government entity to store. While some data may not be sensitive (e.g. air quality, temperature), other types of data may contain personally identifiable information, such as license plate numbers, movement tracking information, or video feeds. AWS provides support for integrating city government functions into their cloud environment. City governments are already using this service to store their information, including the New York City Department of Transportation, the Singapore Land Transport Authority, and the City of Chicago [2].

These use case scenarios support the need for a database structure where only one organization is accessing the data. This is in contrast to privacy-preserving computing, where statistical metadata is shared between multiple entities.

CHAPTER 2

RELATED WORK

Other methods to preserve the confidentiality of a database have been proposed in the past. They range from encryption schemes without regard to SQL operations to complex interactive tree-based systems.

2.1 Early Attempts

Hakan Hacigümüş' 2002 paper [23] is an early example of attempts to solve this problem. It has a similar general setup as both this system and CryptDB [39], but relies more heavily on client-side processing than either system. This becomes a major disadvantage as the size of the data grows. The large ciphertexts need to be transported to the client to be decrypted and filtered. This also severely weakens one of the advantages of cloud computing: increased processing power.

2.2 Enterprise Solution

Microsoft Always Encrypted offers client-side encryption integrated directly with Azure. Sensitive columns can be encrypted with either deterministic or random encryption. Advanced Encryption Standard (AES) is used to encrypt all values in this system. When random encryption is being used, the initialization vector (IV) is a random number. When deterministic encryption is being used, a hash

of the plaintext value is used as the IV. Encrypted columns are accessed through special drivers installed on the client's computer, which perform the encryption and decryption. Columns with random encryption can't have any operations performed on them besides a plain select and cannot be used in any statement's clauses. Columns with deterministic encryption can be used for equality testing and as primary keys, however other comparisons (e.g. order, substring matching) are not available. There are also no homomorphic encryption options, meaning functions like sum or average would be performed on the client machine. Instead, Always Encrypted does not support any mathematical operations on encrypted data.

2.3 CryptDB

CryptDB [39] approaches the problem in a similar manner as this paper, but with more complexity. All data is encrypted up to seven times to achieve different security properties. For example, a string would be encrypted, then that ciphertext would be encrypted, then that ciphertext would be encrypted again, creating three "layers" of encryption. Together, the multiple layers of encryption are called an "onion". The same string would be encrypted again three times with different methods of encryption to form a second onion, then separately encrypted again one more time. This results in three cipher texts (three onions) representing the same data, and seven encryption keys that need to be stored per column. Any sensitive portions of queries over this data also need to be encrypted up to three times and data returned will need to be decrypted up to three times. As sensitive properties of the data are required for the database server to perform a query (e.g. order, equality), the data will be permanently decrypted to a lower security level. Encryption keys are derived from

user passwords as they log in so that a compromised client server doesn't release keys of not-logged-in users.

The system outlined in this thesis takes a simpler approach with less initial security guarantees, but similar guarantees to CryptDB after the system has been in use. The system in this thesis encrypts and stores data only once or twice per datum depending on the data type. This gives faster processing times per query and lower space requirements (see section 5 for a comparison). When the client pays for their data storage based on its size or network usage, the costs can quickly add up when the data is inflated. The per-query execution time is also lower, meaning constantly faster queries. As the system is used, the ciphertexts can reach their lowest layer of encryption, meaning that CryptDB would provide no additional security guarantees while still requiring additional storage and processing power.

Finally, CryptDB cannot be adapted for use in many popular cloud systems because of its dependence on User Defined Functions (UDFs). UDFs are SQL functions written in C or C++ that are added to the server but do not require the recompilation of the DBMS. CryptDB uses UDFs for changing the layer of encryption (i.e., decrypting the onion to reveal a ciphertext with more properties) on a given column and other operations. Popular cloud DBMSs, including Microsoft Azure and Amazon Web Services (AWS), don't allow for the installation of UDFs on their database-only services (such as AWS's Relational Database Service) as not to allow untrusted software run on a shared resource (the underlying server). In order to use a cloud service, one must instead use a virtual machine with their SQL server of choice installed. This preserves many of the benefits of cloud computing, but does not allow for as simple storage size scaling or as simple setup.

2.4 Tree-Based Solutions

Poddar's, Boelter's, and Popa's Arx [37]; Egorov's and Wilkison's ZeroDB [19]; and Popa's, Li's, and Zeldovich's encryption scheme [38] encrypt all data with the randomized AES encryption scheme and use encrypted tree structures to perform operations over the data. The DBMS remains unmodified, but two additional servers are required: a client proxy server that rewrites queries and a server proxy that performs interactive tree traversals with the DBMS. The storage and time penalties for the extra trees and the encryption are high. In the case of Arx, the encrypted database is approximately sixteen times larger than the original database. ZeroDB, a commercial solution, can require multiple decryptions, interactions between the server and client, and traversals of a B-tree for a single query. This introduces a high performance penalty and network load.

2.5 Hardware-Based Soltions

TrustedDB [14] and Cipherbase [13] use a trusted hardware extension of the server to process sensitive data. This allows for faster execution of queries as they don't need to be transformed before they are handled. The queries are either handled by the tamper-proof hardware or are forwarded to the regular query processor if they don't contain secret data. Using the secure processor does incur a performance penalty, but it is less than encryption-only systems. This system requires that a secure module is allowed to be integrated into the system, which is not possible when using commodity data storage and breaks the transparency requirement of this paper.

2.6 Secure Multiparty Computing

Secure multiparty computing, or privacy-preserving computing, is a related but distinct research area: the product of this thesis would be used to secure a production database where the data is owned by a single entity or shared between entities where data does not need to be hidden. Privacy-preserving computing, on the other hand, is used to generate aggregate information from data sets with different owners. For example, Wang et al. [45] is a system to allow a third party auditor to access an organization's data to ensure integrity. In the use case for this thesis, allowing an external user to do any kind of analysis of the data should not be allowed. This is discussed in section 1.2.

CHAPTER 3

CRYPTOSYSTEMS

Multiple cryptosystems, or methods for the encryption of data, are necessary to reveal different properties of the data while still maintaining confidentiality. To achieve our goal, we require a cryptosystem that reveals the order of the plaintext data and one that allows addition without revealing the terms or the sum to the machine performing the computation. We also make use of a hashing method to protect certain types of data.

In this chapter, the following conventions apply for readability:

- m refers to the plaintext message
- e refers to the encrypted message
- encrypt() is the encryption function of the cryptosystem such that $\text{encrypt}(m) = e$
- decrypt() is the decryption function of the cryptosystem such that $\text{decrypt}(e) = m$
- $\text{hash}(x)$ is the SHA-3 hash of x
- $\text{lcm}(x, y)$ is the lowest common multiple of x and y
- $||$ indicates concatenation

3.1 Paillier Encryption

Paillier encryption was created in 1999 by Pascal Paillier to introduce a new additively homomorphic encryption scheme. It is based on the discrete log problem, i.e., the problem stating that it is computationally hard to find m given y, n and the value $y^m \mod n$ [35]. This cryptosystem was selected for this application instead of a more popular cryptosystem such as RSA or ElGamal because it preserves the additive property, while RSA and ElGamal only preserve multiplication [42] [20]. Preservation of addition was chosen over multiplication because commonly used SQL functions like SUM or AVERAGE depend on the availability of addition. Furthermore, multiplication of an encrypted value is still possible if it is being multiplied by a non-sensitive constant. Raising the column to the power of the constant will produce the encrypted product of the plaintext value and the multiplier, i.e. $\text{decrypt}(e^x \mod n^2) = m \cdot x$ where x is the multiplier.

Much like RSA, Paillier keys are generated from two large primes, p and q. To find the complete key, find n, λ, and μ.

$$n = p \cdot q$$

$$\lambda = \text{lcm}((p-1) \cdot (q-1))$$

$$\mu = [\frac{(g^\lambda \mod n^2) - 1}{n}]^{-1} \mod n$$

Let g be a positive non-zero integer less than n^2. The public key is (n, g) and the private key is (λ, μ). To encrypt a message, compute

$$c = g^m \cdot r^n \mod n^2$$

where r is a random number regenerated with every encryption. To decrypt a message, compute

$$m = \frac{(c^\lambda \mod n^2) - 1}{n} \cdot \mu \mod n$$

Addition is just multiplication modulo n^2.

$$m_1 + m_2 = \text{decrypt}(\text{encrypt}(m_1) \times \text{encrypt}(m_2) \mod n^2)$$

Multiplication can be achieved by computing the ciphertext to the power of the multiplier, i.e. $m \cdot x = \text{decrypt}(e^x \mod n^2)$. This reveals the multiplier to the server and can only be used if x is not sensitive. The value of m is never revealed to the entity performing the multiplication.

In section 5 of Paillier's original paper [36], he describes an alternative encryption scheme by the same method. To fit a longer message into a single ciphertext, it is split into two parts, m_1 and m_2 where m_1 can be any message and $m_2 \in \mathbb{Z}_n^*$. m_2 is used in place of r, and the rest of the equation remains unchanged. Using this method, messages less than n^2 can be encrypted in one ciphertext. Both m_1 and m_2 are recoverable.

3.1.1 Derivation from the Original

In order to test the equality of two values after an encrypted addition has occurred (i.e. $a \stackrel{?}{=} b+c$), the encryption of the numeric values must be deterministic. In the original Paillier encryption scheme, the random r prevents this. To make the cryptosystem deterministic, the r is replaced with the SHA-3 hash of the value. This allows for a deterministic encryption without risking exposure of small messages (when the message is small, factoring g^m can be easy if it's not obfuscated by another value).

The trapdoor from section 5 can't be used directly here because it breaks the additive property. Instead of splitting the message m as required by the system, we are defining our m as the paper's m_1 and the hash of m as m_2. So, the paper's m is our $m||hash(m)$. We ignore m_2 during the decryption phase since it will become corrupted by any additions and it is unnecessary. The original paper warns that m_2 ($hash(m)$) must be in $\mathbb{Z}_n^*$. This is guaranteed in our setting because the hash is at most 512 bits, the size of either prime and therefore less than the product of them.

3.1.2 Hardness of the derivation

Given $c = (g^m \mod n) \times \text{hash}(m) \mod n^2$, assume m is not computationally difficult to find. To isolate m, the attacker will first need to find $g^m \mod n$ or hash(m) to begin solving for m. Assuming this is possible, the attacker is left with $g^m \mod n$ and hash(m). Assuming the attacker is able to find m from $g^m \mod n$, the discrete log problem [1] is not hard. Assuming the attacker is able to use hash(m) to find m, the hash function does not provide preimage resistance. Since the discrete log problem is known to be hard, modern hash alogrithms provide preimage resistance [35], and the attacker has no way to find $g^m \mod n$ or hash(m), finding m is computationally difficult.

3.2 Boldyreva's Order Preserving Encryption

Order preserving encryption is a cryptosystem where the ciphertexts of a set of messages appear in the same order as the original messages. This is done by de-

[1] The discrete log problem (DLP) says that "Given the finite cyclic group $\mathbb{Z}_p^*$ of order $p-1$ and a primitive element $\alpha \epsilon \mathbb{Z}_p^*$ and another element $\beta \epsilon \mathbb{Z}_p^*$. The DLP is the problem of determining the integer $1 \leq x \leq p-1$ such that: $\alpha^x \equiv \beta \mod p$" [35].

terministically assigning each message a value in a subrange of the total encryption range. This is illustrated in Figure 3.1. Since the plaintexts are assigned to a subrange in the same order as the original messages, the ciphertexts remain in the correct order.

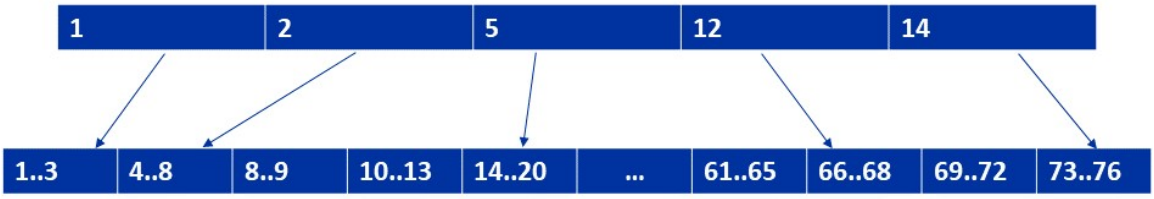


Figure 3.1: An illustration of order preserving encryption.

The order preserving encryption used in this scheme is Boldyreva's Order Preserving Encryption (OPE) [15], the first provably secure order preserving method. OPE allows for direct comparison between any two ciphertexts, allowing for efficient and transparent "where" and "order by" clauses when both ciphertexts are encrypted by the same key. A consequence of using any order preserving encryption scheme is that up to half of the plaintext bits can be derived from the ciphertext [38]. In fact, Boldyreva's original paper shows that it is impossible for any OPE scheme to produce ciphertexts that are indistinguishable under a chosen-plaintext attack. Since the client does not provide an encryption oracle, the attacker shouldn't have access to a chosen plaintext. Tree-based solutions like Popa's "An Ideal-Security Protocol for Order-Preserving Encoding" [38] have been proposed, but incur large space and time penalties, and require interaction between the client and the server. Even using a tree-based cryptosystem, the data can often be derived using a non-crossing attack [22].

3.3 Secure Hash Algorithm 3

Hashing algorithms generate a numeric representation of some data by operating on it in such a way that any original meaning of the data is destroyed. Famous hashing algorithms include Rivest's MD5 [41] and the Secure Hash Algorithm family [18] [26]. Since the plaintext is not recoverable from the hash, hash functions are not encryption methods per se, but are still useful in many applications. The irreversibly of hash functions is called pre-image resistance and is integral to their use. Other important characteristics of a good hash function are second pre-image resistance and collision resistance. Second pre-image resistance means that, given a message, it is hard to find another message with the same hash. Collision resistance means that it should be hard to find any two messages with the same hash. Furthermore, a hash function is especially useful if it generates any given hash with about the same probability as any other hash.

Secure Hash Algorithm 3 (SHA-3) [17] is used in this program to protect table and column names. A hashing algorithm was chosen over encryption for speed: generating the hash of a string is generally much faster than encrypting it. Additionally, these names need to be stored by the client regardless, so there is no need to recover them from the server. SHA-3 was chosen because it is the most recent National Institute of Standards and Technology (NIST) approved hashing method. Older hashing schemes are still commonly used, but are becoming less secure due to both increased computing power availability and inherent weaknesses. For example, a collision in SHA-1 was found last year by researchers at Google [44]. Using flaws in the mathematical foundations of the hashing method, the researchers were able to find the collision 100,000 times faster than using a brute-force method.

CHAPTER 4

DESCRIPTION OF THE SYSTEM

In this section, I will describe how the system components work together to create a lightweight encrypted database system.

4.1 Definitions

- Client: The data owners
- Proxy: The software that encrypts queries and decrypts result sets
- Key store: A SQL server that contains the encryption keys, hashes, and other database metadata. It does not contain any of the user's data and is always owned by the client.
- Cloud server: The SQL server hosted by another organization, which is accessed via the internet

4.2 Access Models

This system can work with two main access models with different benefits for each.

4.2.1 Web Server Model

In this model, one central proxy server - owned by the data owner and residing on their network - processes all query encryptions and result set decryptions. This server could be queried by either connecting client applications to it as an intermediary, or by providing a web interface with which the clients could interact. The main advantage of this model is that it allows for increased portability since a user could access the databases from any computer without having to install any additional software. Another advantage of this model is that key revocation becomes a non issue since all cryptographic operations take place on the server. The primary disadvantage of this model is that the server may become a performance bottleneck since the cryptographic functions require significant computational power. Additionally, it imposes a higher start up cost on the client because they must invest in a server that is capable of performing multiple encryption and decryption requests at a time, removing a main advantage of using cloud computing. This setup is outlined in Figure 4.1.

4.2.2 Decentralized Model

In the decentralized model, encryptions and decryptions are performed by the users' machines themselves, i.e. the proxy software is installed on every user's machine. The users' machines would retrieve the encryption keys from a small local SQL server or keep the key store on their own machine, then use the proxy software installed on their devices to interact with the cloud database. This model would scale more easily than the centralized model since the server doesn't have to perform the cryptographic functions for the users; as the number of users increases, so does the computing power. This also means that the organization does not need to host a powerful

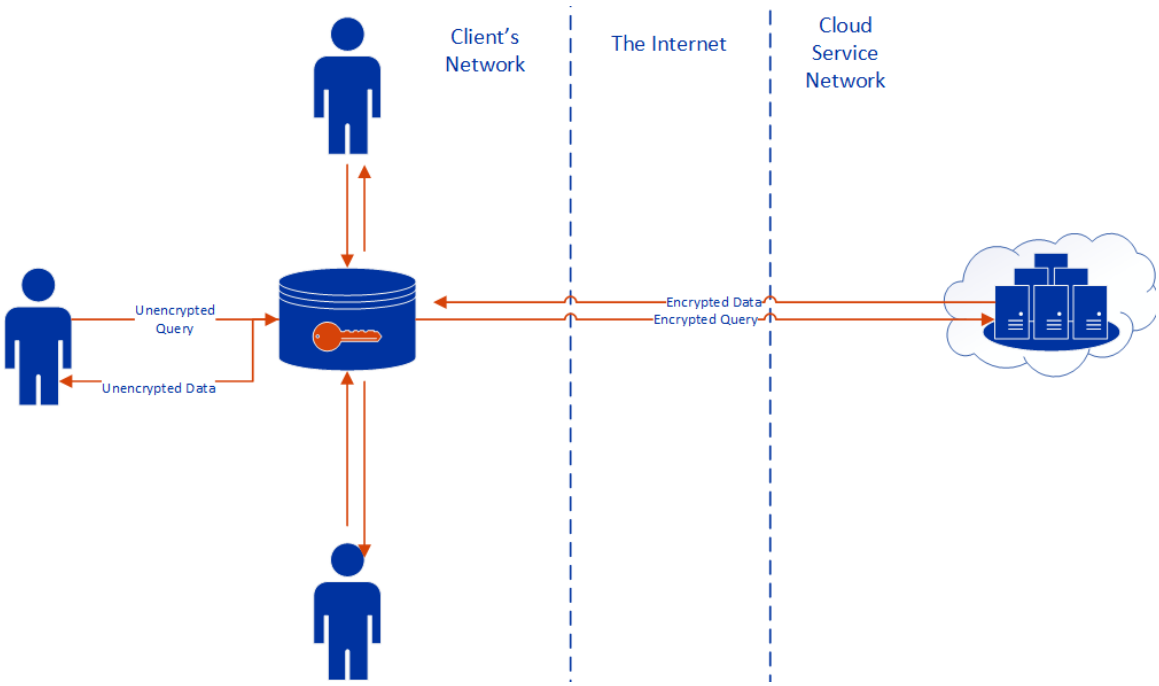


Figure 4.1: A centralized web server model.

server, but rather one can distribute small amounts of data when a user connects to a database for the first time. Key revocation would be difficult as the users would have copies of the keys on their local machines. Changing the keys would mean that the data would have to be downloaded and re-encrypted. Additionally this would require installation of some software, so the data would not be accessible from public workstations. Figure 4.2 demonstrates how interactions in this model could look.

4.3 Encryption Types

There are four types of data that are encrypted in this scheme: **I**dentity (I-type), **N**umeric (N-type), **S**tring (S-type), and **S**ub**S**tring (SS-type).

I-type data includes database, table, and column names. This is hashed using SHA-3. It is hashed instead of encrypted as a time saving measure, since hashing

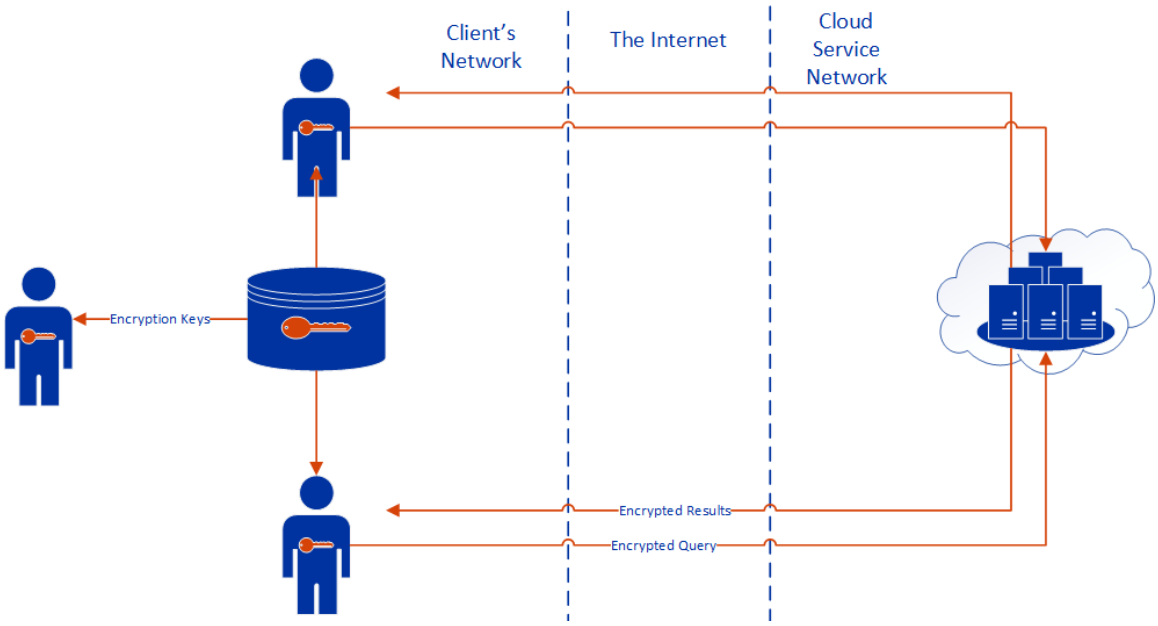


Figure 4.2: A distributed key-sharing model.

is faster than encrypting. An implication of hashing is that the database, table, and column names cannot be recovered from the hash. This is acceptable because reversing the hash is not necessary since the plaintext names must be stored by the proxy anyway so that they can be tied to their encryption keys. Also, because the proxy has access to the plaintext query, it knows what data was requested.

The N-type label is used for any numeric data. The data are encrypted once using OPE and one using Paillier encryption. This is because each type provides a different necessary property for the database functionality. OPE provides both order checks and equality checks. Paillier ciphertexts provide homomorphic operations for addition, subtraction, and non-sensitive multiplication. Since they are maintained as two separate columns, when one is updated, they both need to be changed. This is straightforward when the column is being set to a constant (e.g. myint = 10) since the value can be encrypted with each cryptosystem and uploaded to the server. However,

it is more complicated when the homomorphic property is being taken advantage of during an update query (e.g. **UPDATE** tbl **SET** myint = myint + 10). This is discussed further in section 4.4.6.

S-type data are encrypted using only OPE. This is possible because addition is not applicable to strings. In S-mode, the entire string is encrypted together and substring matching is not possible. SS-type data are encrypted block-by-block, by default using whitespace as a delimiter. The user can select a custom delimiter to better complement their data set, such as a hyphen ("-"). Date type data are automatically encrypted as SS-type data with hyphens as the delimiter to allow for year, month, or day matching. Since the dates are saved in the order year-month-day by default, no adjustment is needed to preserve order [31].

4.4 Walkthough of Software

When the user sends a SQL query, it is first parsed by the proxy software using JSQLParser [10]. JSQLParser transforms a SQL statement into hierarchical Java classes which can be traversed using the visitor pattern [27]. For example, the statement "**SELECT** col **FROM table**" would be transformed into a Select object, which has references to a Column object and a Table object. Using the visitor pattern, a Visitor class would first visit the Select object. Inside the Select object the Column and Table objects would also be recursively visited and hashed. As each object is visited, the relevant content is added to a StringBuilder object that is owned by the visitor object and shared between recursive calls. The result of the StringBuilder is then sent to the DBMS. Data is returned to the proxy over the internet in its encrypted format. The proxy decrypts the results, performs any additional formatting or delayed

arithmetic operations, and finally reveals the data to the user.

4.4.1 Key Store

Keys are stored in a small MySQL database instance that is owned by the client. In the decentralized model, this would be a database on a server on the client's network. In the centralized model, the proxy server and this key store would run on the same server. Each encrypted database is represented by a local database and each table is represented by a local table. Each column is represented as a row in the table. The local tables contain four columns: the data column name, the hashed data column name, the data type, and the encryption type. None of the data itself is stored on the local key server. A local MySQL instance is used because it allows for quick key finding and manipulation. Storing the encryption keys in a MySQL instance also allows for easy key privilege partitioning. The permissions built in to MySQL can be used to only allow only those who should have access to retrieve the keys by setting their select privilege on a per-database or per-table basis.

In this implementation, 512-bit Paillier keys and 128-bit OPE keys are used in order to allow for a better comparison against CryptDB, which uses keys of those sizes. Ideally, a 256-bit OPE and a 2048-bit Paillier key would be used to provide a higher level of security.

4.4.2 Encrypted Data Format

The database retains it's original structure. Database names, table names, and column names are represented by their hashed names. Besides the hashed names, only the data types and number of columns are different. All columns are stored as blobs to allow for sufficient storage space. A blob is a variable-length binary string

datatype, i.e. it holds any amount of binary data [32]. Numeric data has two values, so it is stored in two columns: one that is just the hash of the original column name and one that is the hash with "_hom" (for homomorphic) appended to it.

Because the encryption methods take only numeric inputs, strings must be presented as integers. To convert strings to a numeric format, they are represented as a byte array of ASCII values. The byte array is passed to the BigInteger constructor, which interprets it as a number which can be encrypted. If the string is longer than 32 bits, it is broken into 32-bit sections. Each section is encrypted independently. If the final section is not 32 bits long, it is right-padded before encrypting. Encrypting this way maintains order. Numeric values are encrypted without modification.

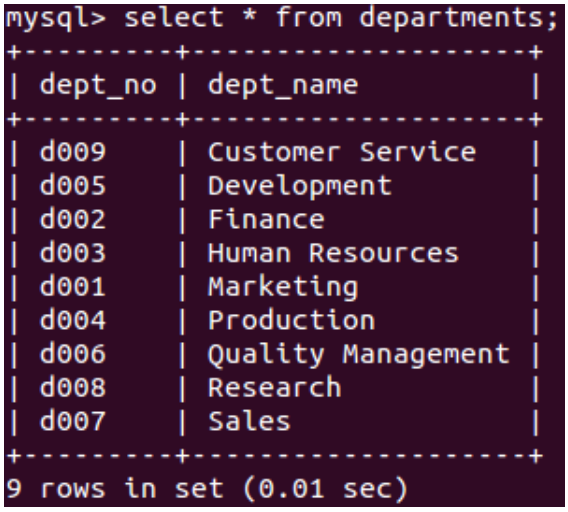
Figure 4.3 shows how the keys are stored in the client-side database. Figure 4.4 shows how the data is stored on the cloud server. Figure 4.5 is the same data if it were stored in an unencrypted database instance.

```
mysql> SELECT id, secType, SUBSTRING(secKey, 1, 30), hash FROM employees.departments;
+------------+---------+--------------------------------+------------------------------------------------------------------+
| id         | secType | SUBSTRING(secKey, 1, 30)       | hash                                                             |
+------------+---------+--------------------------------+------------------------------------------------------------------+
| dept_no    | S       | e55d7e3cb01d95eb7188956e167b64 | 271c3a28b87bdb866f6d8bff7775ef8d68abbf344e581a5d1856e44c         |
| dept_name  | S       | cc2a647b81ce06ed1d85885b12a211 | 27195b528714ec25a5c0bd8ca01008a7999162b02e39dea51dcc22ea         |
+------------+---------+--------------------------------+------------------------------------------------------------------+
2 rows in set (0.03 sec)
```

Figure 4.3: An example of the table the keys are stored in. The keys are truncated due to their length.

```
mysql> SELECT HEX(`271c3a28b87bdb866f6d8bff7775ef8d68abbf344e581a5d1856e44c`), HEX(`27195b528714ec25a5c0bd8ca01008a7999162b02e39dea51dcc22ea`) FROM `f4771
490302912`;
+------------------------------------------------------------------+------------------------------------------------------------------------------------------------------------------------------------+
| HEX(`271c3a28b87bdb866f6d8bff7775ef8d68abbf344e581a5d1856e44c`) | HEX(`27195b528714ec25a5c0bd8ca01008a7999162b02e39dea51dcc22ea`)                                                                    |
+------------------------------------------------------------------+------------------------------------------------------------------------------------------------------------------------------------+
| 0000006431942800                                                 | 0000074688B040C00000073739AC0FC0000006D68A1F46F0000000000414802                                                                    |
| 0000006431942801                                                 | 000000616D4D70780000000004678CB0B                                                                                                  |
| 000000643194286A                                                 | 0000007262935C06000000657254B819000000616D0A9223000000004882AB1D                                                                   |
| 0000006431942A01                                                 | 0000074688B040C0000006F63ABCC010000000000504603                                                                                    |
| 0000006431942A03                                                 | 0000006D64A15CA0000000656B513881000000004473FE01                                                                                   |
| 0000006431942DE0                                                 | 0000006D64A15CA00000006E60A0891A000000791F87A004000000616B5441730000000000515011                                                  |
| 0000006431943006                                                 | 000000616B5020720000000000000499F                                                                                                  |
| 000000643194300C                                                 | 00000061714D81010000005264E8E0A8                                                                                                   |
| 0000006431943100                                                 | 0000076686D10170000000205348A8560000006F6C9FBB2000000043752052AB                                                                   |
+------------------------------------------------------------------+------------------------------------------------------------------------------------------------------------------------------------+
9 rows in set (0.00 sec)
```

Figure 4.4: An example of data stored in it's encrypted format.



```
mysql> select * from departments;
+---------+--------------------+
| dept_no | dept_name          |
+---------+--------------------+
| d009    | Customer Service   |
| d005    | Development        |
| d002    | Finance            |
| d003    | Human Resources    |
| d001    | Marketing          |
| d004    | Production         |
| d006    | Quality Management |
| d008    | Research           |
| d007    | Sales              |
+---------+--------------------+
9 rows in set (0.01 sec)
```

Figure 4.5: The same data as Figure 4.4 in its unencrypted format.

4.4.3 Select Statements

Select statements allow users to retrieve data from the database. As the parser traverses the statement and encounters column names, it inserts the hashed column names into the encrypted version of the statement. If all columns are being selected ("SELECT *"), the * is replaced by the list of column names to avoid selecting the homomorphically encrypted data, which are not necessary and whose ciphertexts are very large relative to the datasize and to the OPE ciphertext size. When a "where" clause is encountered, the column name is hashed and the value is encrypted with its key or processed accordingly if it is an expression. The encrypted query is then forwarded to the server. As an example, the query

```
SELECT fname FROM employee;
```

becomes

```
SELECT
    f0b6dbb5b50ed10963c726f9be05c3e3384e990e8bbaebf9b72bf070
```

```
FROM be93f62e2730e77dfa0a990bf511c76498fd49eb0594f3e99824f82a
;
```

on this system.

When the server returns the requested data, the proxy can begin to decrypt it. The proxy does not need to look up the table names because it has the unencrypted query for reference. Before it begins, it retrieves all the necessary encryption keys from the key store. The proxy then retrieves the data from the cloud server line-by-line, decrypts each column's data, and prints the result. The columns that contain the Paillier encryption of the data are never selected unless as part of an aggregate function due to their larger ciphertext size. The OPE ciphertexts is used for simplicity and speed: simplicity because we only need to perform one type of decryption for all columns, and speed because OPE decryption is much faster than Paillier decryption. Additionally, the size of the ciphertexts are smaller resulting in a faster transmission.

4.4.4 Substring Matching

When the column is marked as applicable to substring matching (SS-type), the data is encrypted on a word-by-word basis. By default, words are delimited by white space and hyphens. This setting is changeable by the user. Encrypting using this method will induce a storage and security penalty. The storage penalty will result from when a plaintext message that would fit in a single ciphertext is split into multiple messages, resulting in an increase in the number of ciphertexts. The security penalty is introduced because frequency analysis attacks [25] have a more granular view of the data.

Joining Tables

Joined tables within a select statement do not need any special treatment and can be hashed as usual, i.e. "tbl1 JOIN tbl2" becomes "68..7d JOIN 42..77". Joins are only available for columns related by foreign key. This is because foreign key columns are encrypted deterministically with the same encryption key, so matching can occur without modification. MySQL will only index 3072 bytes per table between all of the columns used as an index [33]. Because of this, only the first 1000 bytes of each ciphertext column are matched, which corresponds to the first 500 bytes of the plaintext. 1000 bytes is used instead of 3000 to allow for indexes with multiple columns.

Functions

Aggregate functions require UDFs to compute the sum or average of a column. UDFs are SQL functions that are written in C and added to MySQL as a library. They allow the application of powerful C functionality to data in a SQL database. In this case, UDFs are required for aggregate functions because MySQL does not support calculation using large integers.

To find the sum of two Paillier ciphers, a modular multiplication with a large modulus (i.e., n^2, see 3.2) needs to be performed. If the encryption key (and consequently the modulus) were small enough to be operated on by MySQL, it would be insecure. To allow for a large key, a UDF must be used. The BigDigits C library is used to provide operations on large numbers. The use of UDFs is not usually considered modifying the MySQL software [38] because adding them does not modify the SQL server itself, does not require recompilation, and is a legitimate extension of

the MySQL DBMS software [34]. When UDFs can't be installed, sums and averages can be found by first retrieving all the data from the cloud server, then finding the sum of the data after it has been downloaded.

On the cloud server, the sum UDF retrieves each value and homomorphically adds it to the running total. When there are no more values, the result is sent to the proxy for decryption. If the sum were to be larger than n^2, the sum would become corrupted. Because n^2 is larger than the maximum integer size in MySQL, no functionality is lost due to encryption. In an unencrypted database, the too-large sum would overflow and cause an error anyway.

To compute the average, a sum is computed on the column homomorphically and a count is retrieved from the database server in clouds. The sum is decrypted by the proxy, divided by the count, and returned to the user. A division could be performed by the database server by bringing the sum σ to the power of the modular inverse of the column count x (modular division, $\sigma^{x^{-1}} \mod n^2$), but this will fail if the sum is not a multiple of of the count. Instead, both the sum and the column count are retrieved and the division is done on the proxy.

4.4.5 Insert Statements

Insert statements require hashing column and table names as with select statements, but also involve retrieving the necessary data types and keys to encrypt the data to be inserted. If no columns are specified, the data is encrypted in order of the table's columns. When the data is numeric type, it is encrypted once using OPE and once using Paillier and inserted into the appropriate columns. "ON DUPLICATE KEY" subqueries tell the server what to do when it tries to insert a tuple where the primary or foreign key values already exist in the database. These subqueries work

without modification because the data is deterministically encrypted. Data is sent as hexadecimal numbers in the format "x‘0F1..E82’". For example the query

```
INSERT INTO tbl VALUES ('hi');
```

becomes

```
INSERT INTO ‘401
    bfd8f5c0573d62d5b65fb1a01b54a0ba47219ef83863ea873335d‘
    VALUES (x'd0d2113abf');
```

4.4.6 Update Statements

Update statements change already existing rows. This is simple when the update is changing a column to a static value, e.g. "**UPDATE** tbl **SET** col1 = 10;". However, when the additive Paillier property is being used, e.g. "**UPDATE** tbl **SET** col2 = col2 + 2;", the update on the homomorphic (Paillier) column can be done directly by the database server, but the OPE column must be updated manually. Manual updates involve downloading the modified ciphertexts to the client, decrypting them, reencrypting them using OPE, and reuploading them. This process could take a large amount of resources, but is unavoidable since a cryptosystem that preserves both addition and order would allow for easy guessing with any known plaintext.

When a non-static update is performed, the update of the OPE column could happen in two places: either at the time the update is performed or when the order preserving property is required. A delayed update can be implemented by overwriting the old OPE value to NULL when the homomorphic update is initially performed. The reencryption can be triggered when all of the following are true: the order preserving value is requested, the OPE column is null, and the Pailler column is not

null (indicating the OPE value should also not be null). The benefit of reencrypting only when the order preserving property is used is that if an order query is never executed, the expensive computations can be avoided. The advantage of doing the reencryption when the initial update is performed is predictability/reliability. If the reencryption is performed at a later time, a time-sensitive select query with an "order by" clause may trigger the reencryption without the user realizing the query will take a long time.

4.4.7 Delete Statements

Delete statements remove entire rows from a database. Lines can be targeted for deletion by "where" and "limit" clauses. To delete encrypted data, no special processing is required beyond hashing the I-type data and encrypting any values in a where clause.

4.4.8 Create Statements

Create statements are used to add new tables to a database. While the sensitive information is being hashed and added to the encrypted query, encryption keys must be generated and added to the key store in the proxy. First, the table name is extracted from the unencrypted query. A table is created in the key store with the same name. Then, as each column name and data type is encountered, a key is generated using Java's SecureRandom [28]. For Paillier keys, two probable primes are found using the BigInteger [29] class in combination with the SecureRandom class. The key is stored as one prime concatenated with the other and the required values are recalculated when the key is used. For numeric data, the OPE key is the first 128 bits of the first prime. String-type columns generate a random 128 bit number. When

the column is a foreign key reference, a new encryption key is not generated. Instead, the encryption key for the foreign column being referenced is copied. If the column is string-type, the user is prompted to indicate whether the data will require substring matching. The column name, type, encryption key, and hash are then inserted into the key store. All data types are replaced with "BLOB" in the encrypted statement. Finally, the hashed create statement is sent to the database server.

4.4.9 Alter Statements

Alter statements allow users to make changes to existing tables. Their usability in an encrypted database system might be limited depending on the specific alteration being requested, but most operations are still possible. Adding columns is no more difficult than the initial creation of a column, and only involves hashing the column name and adding its encryption key to the key store. Similarly, dropping a column only requires the row containing the key to be removed from the key store before sending the hashed drop statement to the server. Adding primary keys is also easy and requires no actual alterations to the data or key store; only hashing the column name and primary or foreign key name is needed. Renaming a column is as simple as updating the key store with the new name. Adding a foreign key relationship requires more work since the encryption key must change. In order to add a foreign key relationship after the initial table creation, the data in the column must be downloaded, reencrypted, and uploaded again.

CHAPTER 5

EXPERIMENTATION

In order to test the penalties introduced by the system, we compare it against an unencrypted instance of MySQL server and against CryptDB. We want to test these systems based on the following metrics: storage penalty, database creation time penalty, query time penalty, and security.

To guarantee the fitness of the system in a cloud environment, this system and the unencrypted system were also tested for correctness using a virtual machine instance on Microsoft Azure. Since the available implementation of CryptDB does not allow for a remote database, it is not tested in the cloud. This data was only checked for correctness.

The proxy was developed in Java and the MySQL User Defined Functions (UDFs) were developed in C. The proxy uses JSQLParser [10] to parse the SQL statements, SECRAM [6] and BouncyCastle [12] for cryptography, and Oracle's **J**ava **D**ata**b**ase **C**onnectivity (JDBC) [30] for interaction with the databases. In the UDFs, the BigDigits C library [16] was used to support operations on very large numbers, namely ciphertexts and public keys.

5.1 Experimental Method

All three systems were tested on a virtual machine with 4GB of RAM and 1 virtual processor. VirtualBox was used as the hypervisor, Ubuntu as the operating system, and MySQL from Oracle as the SQL server. Each virtual machine was reset to a snapshot between each round of testing to ensure a consistent test environment. The tests were run with the proxy system and the encrypted database on the same system. This is required for testing because the implementation of CryptDB publicly available on Github does not allow the encrypted database to be located anywhere besides the local machine. This also removes the network load variable since all operations happen on the same machine.

We used the employees sample database set provided by Oracle [11] for testing. The database consists of 6 tables with a combined 24 columns and 3,919,015 total rows. Tables connecting each specific employee's information (employees, salaries, titles) are linked together with foreign keys by the employee's id (emp_no). The department tables (departments, dept_emp, dept_manager) are linked by the department's id number (dept_id). A complete list of tables and columns can be found in Appendix D.

5.2 Space Overhead

The size of our encrypted database is heavily impacted by the number of numeric (N-type) columns. This is partially because the data is inserted into the database twice (once encrypted with order preserving encryption (OPE) and once with Paillier encryption). The size increase is also because the ciphertexts of an asymmetric cryptosystem like the Paillier cryptosystem are much larger than those of a sym-

metric cryptosystem like Boldyreva's OPE. The Paillier ciphertexts are in the rage of $n < c < n^2$, where n is the product of two large primes. Specifically in this implementation, the Paillier ciphertexts are in the range of 2^{1023} to 2^{2048} since each prime is 512 bits. On the other hand, the OPE ciphertexts are $\lceil \frac{s}{32} \rceil \cdot 64$ bits where s is the bit size of the plaintext value. We know this because each plaintext is split into maximum 32 bit blocks, then each block is encrypted to form a 64 bit ciphertext block.

This data set contains 18 string values and 6 numeric values, corresponding to 24 OPE ciphertexts and 6 Paillier ciphertexts. For this implementation of OPE, the plaintext space is 32 bits and the ciphertext space is 64 bits, resulting in ciphertexts about twice the size of the plaintexts. Paillier encryption is much more expensive, resulting in about a 64 times increase from plaintext to ciphertext. Specifically, 4 byte integers become 256 byte ciphertexts. Minimizing the amount of data stored as an integer can help reduce the size of the encrypted database. Below, we compare the size of this database to the size of the unencrypted database and to the size of CryptDB.

Table 5.1: Size comparisons among encryption schemes.

Method	Size (MB)	Increase Factor
Unencrypted	160.6	0
Thesis	1205.5	6.5
CryptDB	3229.4	19.1

Table 5.1 shows the impact of encryption on the database's size. Increase Factor is defined as

$$\text{Increase Factor} = \frac{e - u}{u}$$

where e is the encrypted database size and u is the unencrypted database size.

Encryption with our method causes a 6.5 times increase in data size, while encrypting the same data set with CryptDB created a 19.1 times increase.

As discussed above, integers make up a disproportionate amount of the increase for this method. Of the 1205.5 megabytes (MB) of encrypted data, 579.5 MB (48%) are Paillier ciphertexts. Of the 160.6 MB of original data, only 43 MB (26%) were integers. This demonstrates a 13.4 times increase in integers, which is much higher than the general expansion rate. CryptDB's expansion rate is much greater because the same data is stored in the database three times regardless of the data's type, and because each layer of encryption creates a larger ciphertext than the last.

5.3 Timing Overhead

The time required to perform various operations can be loosely compared between implementations. Since they are programmed in different languages, the underlying technology can change the time taken to run. For example, Gherardi's, Brugali's and Comotti's paper [21] shows that a Java implementation of a commonly used robotics algorithm is 1.09 to 1.91 times slower than the same algorithm in C++, which CryptDB is implemented in. Even so, the reduced amount of cryptographic operations performed shows a clear reduction in the time required to interact with the database.

5.3.1 Initial Database Creation and Insert Statements

Creation of the initial database requires generating encryption keys and encrypting the database schema. The employees database is provided as a schema file containing

the create statements and several dump files containing the data itself as a series of insert statements.

Encryption of the create statements is fast, since only the table and column names need to be hidden and the keys generated. Each create statement encryption and key generation took around 1 millisecond. Encryption of the actual initial data is done via a series of insert statements, taking a total of 14,614,472 ms (4.05 hours). By comparison, not encrypting the data took 123 ms to create and load the database, and CryptDB took 25,404,584 ms (7.05 hours) to create and load it. That is the expected behavior since this method is encrypting the data fewer times compared to CryptDB. Table 5.2 shows the time necessary to create the database using each method, as well as the increase factor. Here, the increase factor is $\frac{e-u}{u}$ where e is the time taken to encrypt the database using the specified method and u is the time taken to create and load the unencrypted database.

Table 5.2: Time comparison for encrypting the same database among encryption schemes.

Method	Time (ms)	(min)	Increase Factor
Unencrypted	123	.002	0
Thesis	14614472	243.5	118815.84
CryptDB	25404584	423.4	206540.33

5.3.2 Select Statements

Select statements are used to retrieve data from the database. When received by the proxy, the statements are converted using the process described in section 4.4.3. This causes a small increase in size, since the 256-bit hash names are likely larger than the original table and column names. This is especially true for select all ("**SELECT** *..") queries, since each column needs to be enumerated to avoid also selecting the

Paillier encrypted columns. It is beneficial to avoid downloading those columns since they are not usually necessary and are much larger than the OPE-encrypted columns, resulting in a reduction in the amount of data transmitted even with the larger query. The majority of the total transmission size consists of the requested data itself.

Table 5.3 shows the size of the data that is transmitted back to the user for the query "**SELECT** $*$ **FROM** employees **LIMIT** 100000;". The size of the data returned by this method is about 3.5 times larger than the unencrypted data, while CryptDB caused an about 7.7 times size increase compared to the unencrypted query. This is a result of CryptDB's larger ciphertexts, which are a result of the layers of encryption creating a blow-up effect. Since only the OPE encrypted columns need to be transmitted, the size of numeric data in transit is less than the size of the same data at rest on the cloud. Reductions in the size of the returned data are especially helpful in cloud systems that charge for outbound data usage, such as AWS [1] or Google's cloud solution [5].

Table 5.3: Size increase of returned data.

Method	Returned Data Size (MB)	Increase Factor
Unencrypted	4.7	0
Thesis	16.5	2.5
CryptDB	36.2	6.7

Figure 5.1 (tabulated in Table 5.4) compares the time taken by the three systems to perform the same select queries. 5.1a shows the average total time for each query (including the client side encryptions and decryptions as well as the server side SQL operations), while 5.1b shows the time taken by the server to execute the query. Total time is used instead of client processing time to minimize the amount of modifications made to CryptDB to enable testing. It also is useful because it

represents the real impact on the user. The data points represent the retrieval of 1,000; 5,000; 10,000; 50,000; and 100,000 rows from the employees table, i.e. "**SELECT** $*$ **FROM** employees **LIMIT** 1000". The complete query list can be found in Appendix A. Before the testing phase, one row is selected from the database to allow for any required initialization of the systems (encryption key retrieval, etc.). This more accurately reflects the impact of the system since the proxy isn't likely be restarted between queries.

When selecting a smaller amount of records, this method takes more time than CryptDB because the implementation of OPE that is used has a high start-up time. When the number of rows is higher, the cost is shared between the rows and the total decryption time is less. This can be verified by looking at the per-row decryption time. For 1,000 rows, the average query time per row is 1.19 ms. For 10,000 to 100,000 rows, the average is about .2 ms, less than one fifth of the time required for a lesser amount of rows and about a third of the time required by CryptDB. The increase in time required for a select query as the number of rows returned grows is about linear once the impact is shared among enough rows. The server-side processing time for this method is about the same as that of an unencrypted database, while CryptDB is consistently higher. This is probably because of the larger amount of data that the server needs to retrieve from the disk.

Throughput

Throughput is the amount of data that is transferred in a given amount of time. Faster throughput makes more responsive applications, and is especially important in the web service industry. Measuring throughput reveals the real effect on the end user, who is ultimately affected by the slowdown. The throughput penalty can be

(a) Average total processing time

(b) Average server processing time

Figure 5.1: Average query execution times as the number of returned rows increases for select statements. Note that the x-axis is in log scale, so the linear increase in time appears to be exponential.

Table 5.4: The impact of the various systems on the time taken to retrieve information from the database. (Tabulated data from Figure 5.1a).

Method	Rows Affected	Total Time (ms)	Time / Row (ms)
Unencrypted	1000	7	0.006
	5000	8	0.002
	10000	18	0.002
	50000	63	0.001
	100000	129	0.001
Thesis	1000	1190	1.19
	5000	1657	.33
	10000	2817	.28
	50000	10850	.22
	100000	21526	.22
CryptDB	1000	714	0.7140563557
	5000	3725	0.75
	10000	6031	0.60
	50000	41712	0.83
	100000	67338	0.67

measured by taking the size of the actual, plaintext information being transmitted divided by the time taken for any query. Throughput of an encrypted database is impacted by the larger size of the data being transferred and the time taken to decrypt the results. Table 5.5 compares the average plaintext bytes transferred per second of a select statement ($\frac{\text{plaintext data size}}{\text{Average query time}}$), and the percentage of the transmission speed that represents ($\frac{\text{Method's throughput}}{\text{Unencrypted throughput}} \times 100$). To calculate these results, the data from the query returning 100,000 rows is used. Our method has a significant impact (slow down) on throughput, but is still six times faster than CryptDB. These results are shown in Table 5.5.

Table 5.5: The methods' effect on throughput speed. Throughput is represented as bytes per second.

Method	Throughput (B/s)	Percent Throughput
Unencrypted	35890.9	100
Thesis	215.4	0.6
CryptDB	68.9	0.1

"Where" Clauses

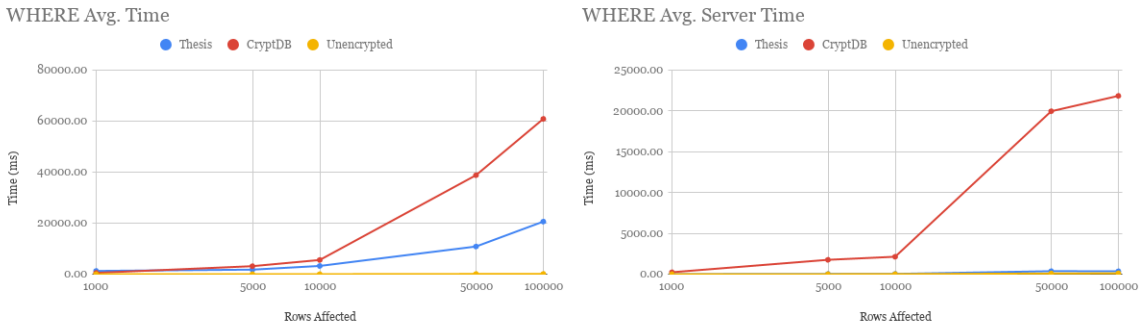
"Where" clauses allow the user to filter the data that is returned to them. The difference between the select statement with the "where" clause and the select statement without is measured. This measures the impact of including a where clause in a select query for the three systems we tested. The "where" clauses were chosen so that they should return the same data as the statement without the clause. See Appendix A.1 for the list of statements used.

When the "where" clause is introduced, CryptDB must lower its security level to allow for equality matching. The outermost layer of encryption is removed to reveal deterministic ciphertexts that allow for equality matching. This operation was

performed before the testing takes place since it only happens once and therefore does not impact day-to-day operations.

Figure 5.2 (tabulated in Table 5.6) shows the time required for a select query with a "where" clause. Including a "where" clause doesn't change the overall form of the trends; CryptDB begins taking more time than this method after retrieving around 1200 records and this method still creates about the same server load as an unencrypted database. Figure 5.3 graphs the time taken for a select statement with a where clause minus the time for a select statement without one, i.e. how much longer the query takes when the where clause is added. Initially, CryptDB showed a decrease on both the client and server side. This is because the security level was downgraded, meaning that less decryptions needed to be performed in order to find the plaintext data. In order to test the actual difference between a select statement with and without a "where" clause, the outermost layer of encryption needed to be first removed. Figure 5.4 shows the new comparison. When the select without a "where" is run at a lower security level, i.e. the security level the system is at after equality matching is required, all three systems register little difference between plain select statements and those with "where" clauses until about 5,000 records are returned, when CryptDB starts requiring more computation time. As with the plain select statement, the increase is approximately linear after about 10,000 rows are returned due to the start up cost of the OPE implementation.

From these results, we see that none of the three systems require much extra time when a "where" statement is introduced as compared to the same statement without it. This is encouraging because it shows that introducing encryption does not always significantly increase the time needed to perform equality matches on data despite the data's larger size.



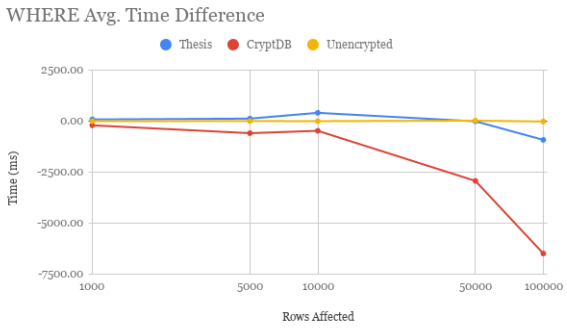
(a) Average total processing time

(b) Average server processing time

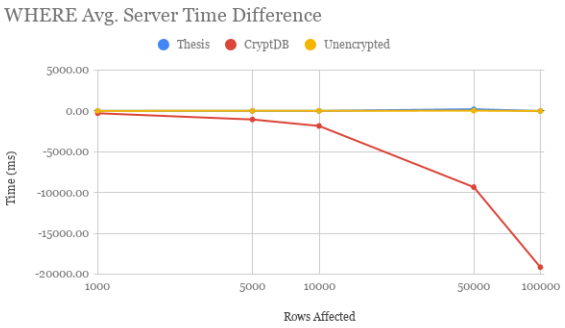
Figure 5.2: Average query execution times as the number of returned rows increases.

Table 5.6: The impact of the various systems on the time taken to retrieve information from the database when a "where" clause is added to the query. (Tabulated data from Figure 5.2).

Method	Rows Affected	Total Time (ms)	Time / Row (ms)
Unencrypted	1000	0.75	0.00075
	5000	7.785666667	0.002
	10000	13.537	0.001
	50000	82.797	0.002
	100000	107.5026667	0.001
Thesis	1000	1270	1.27
	5000	1776	0.36
	10000	3220	0.32
	50000	10838	0.22
	100000	20605	0.21
CryptDB	1000	5057	0.50
	5000	3131	0.63
	10000	5554	0.56
	50000	38780	0.78
	100000	60848	0.61

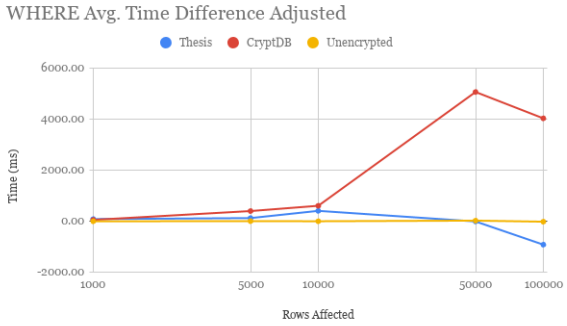


(a) Average total processing time difference

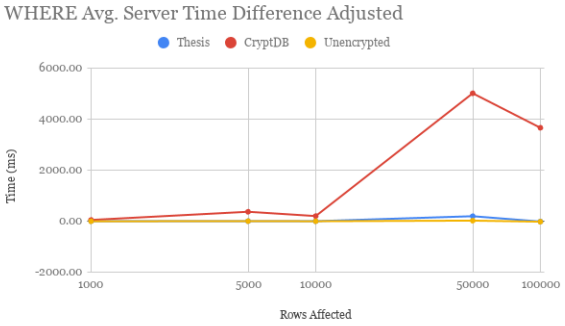


(b) Average server processing time difference

Figure 5.3: The difference between a plain select statement and a select statement with a where clause. CryptDB shows a greatly reduced time because the encryption level changed. See 5.4 for an equal comparison.



(a) Average total processing time difference



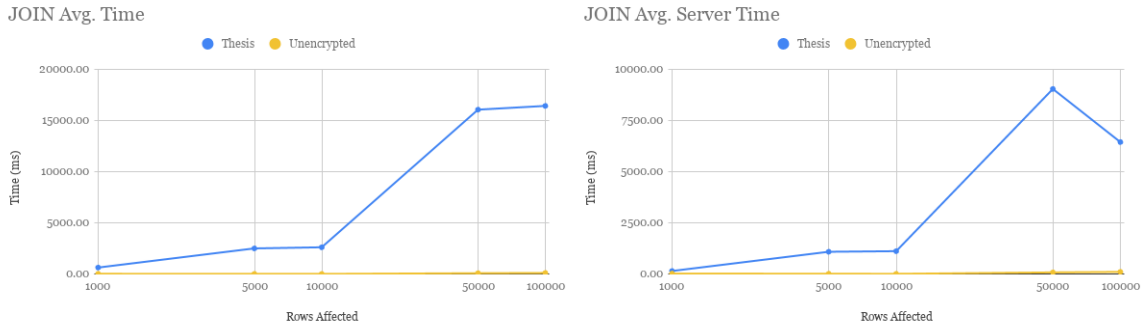
(b) Average server processing time difference

Figure 5.4: The difference between a plain select statement and a select statement with a where clause when CryptDB is at a lower encryption level.

Join Clauses

Join clauses allow for data from two tables to be temporarily combined, e.g. pair an address from the employees table with a salary from the salaries table. Join statements couldn't be tested on CryptDB because the version available on Github does not have foreign key assignments correctly implemented.

Joins using this method do not require any special transformations by the SQL proxy, but do take extra time on the server due to the larger size of the data. The result of the experiment is shown in Figure 5.5 (tabulated in Table 5.7). The queries used are listed in Appendix A.2.



(a) Average total processing time

(b) Average server processing time

Figure 5.5: Select queries with joined tables.

Table 5.7: The impact of the various systems on the time taken to retrieve information from the database when a join clause is added to the query. (Tabulated data from Figure 5.5).

Method	Rows Affected	Total Time (ms)	Time / Row (ms)
Unencrypted	1000	20.11	0.020
	5000	11.90	0.002
	10000	7.92	0.001
	50000	77.81	0.002
	100000	103.01	0.001
Thesis	1000	627.33	0.63
	5000	2498.33	0.50
	10000	2608.00	0.26
	50000	16052.67	0.32
	100000	16418.67	0.16

Aggregate Functions

Aggregate functions, such as sum, average, or count, are important tools for a SQL programmer. Both this method and CryptDB make use of Paillier encryption's additive homomorphic property in order to compute the sum of any numeric data. Computing the encrypted sum requires modular multiplications, which are computationally more intensive than a regular addition. Consequently, the amount of time

required to compute the encrypted sum is increased above an unencrypted sum query. To test the time required to compute a sum, the entire salary column, with a total of 2844047 rows, is summed in the salaries table. This allows for testing the function without any possible interference from a "where" or a "limit" clause. This also shows that large numbers can be added without causing overflow issues as all three methods calculated the same value. Table 5.8 shows the amount of time needed to compute the sum of the column.

```
SELECT SUM(salary) FROM salaries;
```

Table 5.8: Time taken to compute the sum of the salary column with a total of 2844047 rows.

Method	Sum Function Time (ms)	(min)	Increase Factor
Unencrypted	1.4	0.00002	0
Thesis	620312.6	10.3	443079.4
CryptDB	2450771.314	40.8	1750549.9

The count function in our system does not need to be modified from the one used in the unencrypted database, i.e. only the column, table, and database names are changed, not the number of rows. This is possible because "null" (empty) values are not encrypted, so they will not be counted. In the encrypted databases, the count takes longer to compute because of the larger data size and because of the delay introduced by the intermediary software. Table 5.9 shows the results of the test.

```
SELECT COUNT(salary) FROM salaries;
```

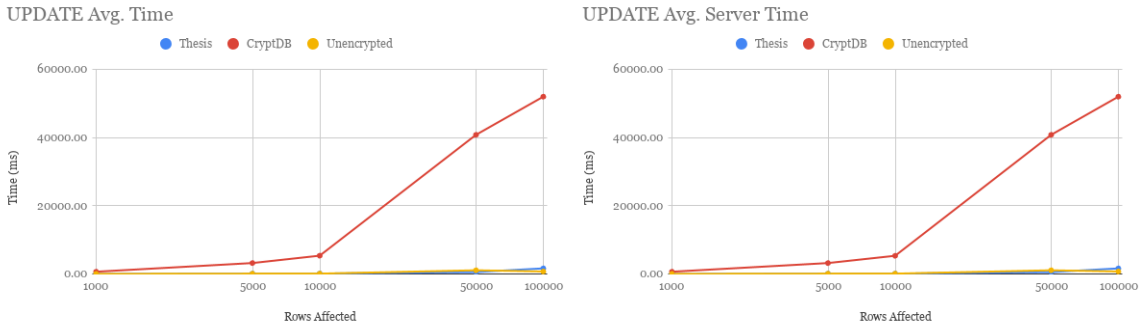
5.3.3 Update Statements

Update statements allow users to change columns in rows that already exist. In the proposed system, statements do not take much longer than those on an unencrypted

Table 5.9: Time taken to count the number rows in the salary column with a total of 2844047 rows.

Method	Count Function Time (ms)	(min)	Increase Factor
Unencrypted	1.5	0.00002	0
Thesis	14126.3	.235	9397.9
CryptDB	1319385.0	21.9	877854.6

database. For the query we tested, the data only had to be encrypted once and could be applied to all rows in the column. Work done by the server accounted for most of the work for both encrypted database schemes. For large batch updates, this method takes about the same amount of time as an unencrypted update since a single encryption does not take much time. CryptDB takes much longer because it must generate a different ciphertext for each row, even when it represents the same data. While this may not be a common operation, batch updates were tested to follow a similar form as the other tests. The timing results of the sample updates (Appendix B) can be seen in Figure 5.6 (tabulated in Table 5.10).



(a) Average total processing time

(b) Average server processing time

Figure 5.6: Update statements as the number of rows affected increases.

Table 5.10: The impact of the various systems on the time taken to update information in the database. (Tabulated data from figure 5.6)

Method	Rows Affected	Total Time (ms)	Time / Row (ms)
Unencrypted	1000	35.60	0.036
	5000	95.69	0.019
	10000	105.31	0.011
	50000	1083.56	0.022
	100000	732.37	0.007
Thesis	1000	14.67	0.01
	5000	95.00	0.02
	10000	118.00	0.01
	50000	564.67	0.01
	100000	1637.67	0.02
CryptDB	1000	663.42	0.66
	5000	3205.58	0.64
	10000	5364.61	0.54
	50000	40786.59	0.82
	100000	51964.32	0.52

As a more common use scenario, updates that change a single line were also tested. This simulates users entering information into an application that changes previously entered information, such as updating an email address or their phone number. Since these types of updates seem most commonly to be strings, the last name column

is updated here as the experiment. In this scenario, 100 different rows receive 100 different encrypted values in the form "**UPDATE** employees **SET** last_name = 'x ' WHERE emp_no = y" where each x is a different value and each y is a unique employee id (emp_id) number. The CryptDB encryption layer needed to be changed to support the equality in the "where" clause, but the time taken to perform that decryption is not included in the results shown in Table 5.11.

Table 5.11: The average time taken to update existing rows with a unique ciphertext.

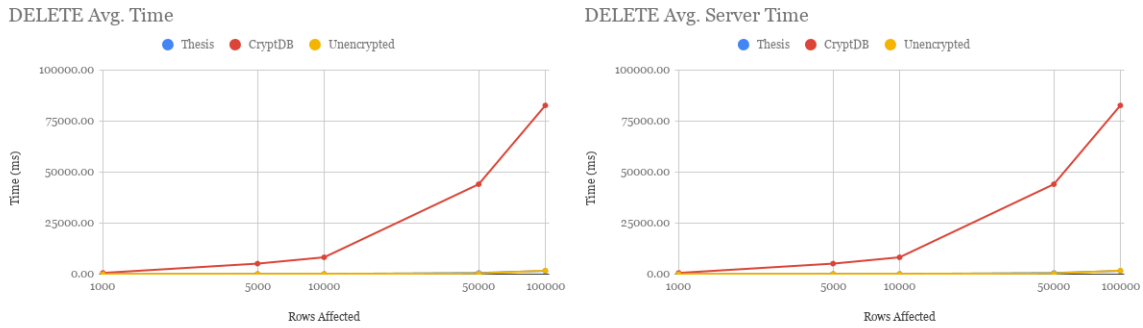
Method	Time / 100 Updates (ms)	Time / Update (ms)	Increase Factor
Unencrypted	231.1	2.3	0
Thesis	1660.3	16.6	6.2
CryptDB	79618.3	796.2	343.6

For our method, updating the "last_name" column takes much less time than a general insertion because only one datum needs to be encrypted and because no Paillier encryption is required. Paillier encryption is much more computationally expensive than order preserving encryption. Since only one encryption is required, much less time is taken than the multiple encryptions required to update one datum in CryptDB. Even though equality matching is revealed to the server (meaning that the security of the system in this thesis and that of CryptDB are the same), CryptDB will still encrypt each entry for "last_name" six times and update three columns.

5.3.4 Delete Statements

Delete statements remove data from the database. Delete statements require no encryption or decryption, only hashing of the I-type data. Accordingly, this method takes about the same amount of time to delete data as an unencrypted database. The comparison can be seen in Figure 5.7 (tabulated in Table 5.12). Although it

also doesn't require extra encryption, CryptDB takes longer to delete because of the larger amount of data that needs to be marked for removal from the disk and the increased number of columns.



(a) Average total processing time

(b) Average server processing time

Figure 5.7: Average time in ms required to execute delete queries as the amount of data being removed increases.

Table 5.12: The impact of the various systems on the time taken to delete information from the database. (Tabulated data from Figure 5.7)

Method	Rows Affected	Total Time (ms)	Time / Row (ms)
Unencrypted	1000	51.74	0.052
	5000	63.45	0.013
	10000	116.83	0.012
	50000	354.96	0.007
	100000	1723.28	0.017
Thesis	1000	47.00	0.044
	5000	104.33	0.020
	10000	124.00	0.011
	50000	584.00	0.011
	100000	1610.33	0.016
CryptDB	1000	612.03	0.607
	5000	5167.47	1.033
	10000	8253.49	0.825
	50000	44071.93	0.881
	100000	82637.76	0.826

5.4 Security

Cloud systems generally have strong security against outside attackers. With this in mind, this thesis is mainly concerned with attacks by insiders of the cloud system: the system administrators, server technicians, or the network administrators employed by the cloud provider. We are also concerned that the hosting company might use another business' sensitive data to increase its advantage in the market. These attackers have persistent access to the data and are able to monitor it as it changes. This thesis is mainly concerned with providing confidentiality. Integrity and authenticity are also discussed.

5.4.1 Confidentiality

While CryptDB requires many encryptions for each datum, its encryption guarantee is only stronger than the proposed system's as long as any order operation (a greater or less than operation or an "order by" clause) is not performed on a column. Once that has happened, their security is no stronger than the system proposed in this thesis. At that point, both systems have the Boldyreva order preserving encryption (OPE) ciphertext and the Paillier cryptosystem ciphertext exposed to the cloud system.

Some columns may never require sorting, while others' OPE values will be quickly revealed. For example, names will need to be sorted alphabetically, dates will need to be sorted to determine seniority, or sensor values from a certain time period will need to be retrieved. Using the order property at any point in CryptDB leaves the column permanently in the less secure state since any order-preserving query causes a decryption to the order-revealing ciphertext.

As an example, any column besides title and dept_name in the sample employees

database would likely require the order property. This corresponds to 22 of the 24 (91.6%) columns in the CryptDB instance being reduced to the same security guarantee as the instance in the proposed system. Of course, both systems provide more confidentiality than an unencrypted database.

While cryptosystems that reveal order are inherently weaker than those that do not, even computationally expensive tree-based ordering systems reveal enough information for a motivated attacker to breach the data while still consuming a high amount of resources. Boldyreva's order preserving encryption, used here, is known to satisfy the "pseudorandom order-preserving function against chosen-ciphertext attack" (POPF-CCA) security definition [15]. Tree-based systems use nondeterministic, very secure cryptosystems in conjunction with some tree data structure to achieve indistinguishably under an ordered chosen plaintext attack (IND-OCPA), which is normally considered a very high standard for order-based encryption. Grubs et al.'s paper [22] shows that any encryption method that contains ordering information is susceptible to leakage-abuse attacks by exploiting a non-crossing attack. Systems that use tree structures to reveal order consume significantly more resources while still leaving data vulnerable.

5.4.2 Authenticity

Authenticity is provided when an attacker cannot insert new data into the database that appears to be provided by a legitimate user. Here, it guaranteed for string (S- and SS-type) data because the order-preserving encryption keys are computationally hard to find from the encrypted values. An attacker cannot insert new data into the database without the encryption key. The only data an attacker would be able to

insert is data that is already in the database, or a random value which may or may not be able to be decrypted.

Because the Paillier cryptosystem is a public key cryptosystem, the attacker could insert valid Paillier ciphertexts into the database. The OPE cipertext won't be updated to reflect the new value until the user sends an update query that involves using the homomorphic value. The user will continue to see the old value until that update is executed, until the user selects homomorphically updated data (e.g. **SELECT** col + 10 **FROM** tbl;), or the user overwrites the modified ciphertext with a new value (erasing the damage from the attack). This is the case for both this system and for CryptDB. Of course, the attacker could insert any type of data into an unencrypted database without detection.

5.4.3 Integrity

Integrity concerns the ability for the attacker to change data that is already in the database without detection. This system can provide some integrity protection for string values (S- and SS-type data), but not numeric values (N-type). Integrity is provided for OPE encrypted values because new data cannot be inserted without the encryption key, which is not revealed to the server. Any change to the ciphertext without the key would be unlikely to render an intelligible plaintext. However, without another system in place designed for integrity checking, there is not a systematic way to check if the data has been modified. The changes can only be detected once a user tries to interact with the data and notices that the results are corrupted. To provide integrity that is verifiable by the system itself, better systems are available [24].

Because of the malleability provided by Paillier ciphertexts that allows aggregate functions in the system to work, these ciphertexts can be modified by the attacker and remain valid encryptions. This is achieved by multiplying two ciphertexts retrieved from the database modulo n^2 (which is public), or by raising a ciphertext to the power of an unencrypted number modulo n^2 and overwriting the old value with the result. Again, this won't affect the user until the user uses the homomorphic property or overwrites the changed data.

Integrity of Paillier encrypted data could be checked in this system by decrypting the cipher to retrieve the hash of the number ($\text{hash}(m_1)$, which is m_2 in section 5 of [36]) and comparing it with the hash of the retrieved number (m_1).

$$(c \times g^{-m_1})^{n^{-1}} \mod n \stackrel{?}{=} \text{hash}(m_1)$$

This is an expensive operation since transmission and decryption of the Paillier ciphertext would not otherwise need to occur. Additionally, any data that has been added to homomorphically would fail the hash check, with no way of knowing whether the failure was caused by a legitimate addition or by data tampering. This is because when multiplying the ciphertexts, ($c = g^m \times \text{hash}(m) \mod n$) the hashes are also multiplied together modulo n.

CHAPTER 6

CONCLUSION

The use of cloud services for storing and processing data has become an irreversible trend. As data collection becomes more ubiquitous and more personal, protecting that data becomes significantly more important. Data encryption allows for consumer trust to be retained in businesses that are using their data while allowing cost savings for the business as well as for the customer by storing it on the cloud. Quick data processing and data privacy are both important goals, but are often perceived as at odds with each other by those implementing the systems. By using a minimum amount of targeted encryption, we can provide fast data access and operation while not releasing the data outside of its owner's organization.

In this thesis, I have proposed a method of encrypting data such that it can still be operated on without revealing the data to a third party (the cloud server), that is more efficient than other systems, and that can be deployed on a commodity database-as-a-service server or on a cloud virtual machine. A minimal amount of encryption is used in order to preserve one of the benefits of cloud computing - increased processing power - while still preserving confidentiality. It also preserves the cost-saving component of the cloud, since less encryption corresponds to smaller encrypted data sizes, meaning less cloud storage needs to be purchased and less data needs to be transferred. I have tested the system in order to ascertain the impact of

the system and compared it with other storage possibilities. These results show that the transparency requirement is not broken (i.e. the queries were runnable without modifying the server software) and that the impact of the encryption on the size of the database and the time taken to execute queries is less than other database encryption methods.

6.1 Limitations

The tests performed in this work were not done with actual data, and the sample data used may be limited in its reflection of a database system used in a production environment. Similarly, the queries used to benchmark the system may not reflect queries used in day-to-day operations. However, they should give a general idea about how the systems compare for basic operations. This is especially true since the systems are implemented with different technologies.

Additionally, this system assumes that table relations do not change after the tables' initial creation, i.e. that foreign keys are not added to the tables at a later time. Any addition of a foreign key constraint would require the entire column to be re-encrypted with the alternate key.

6.2 Future Work

1. A web interface, like that outlined in Figure 4.1, could be implemented and tested.

2. Testing and comparison using real data sets could give a picture of the day-to-day impact of this system. Testing using larger or more complex data sets could also be beneficial.

3. Testing was performed using MySQL. Other SQL dialects could be implemented.

4. The implementation used for experimentation here did not include any optimizations such as multithreading or caching. Implementing those could give a better idea of the commercial viability of this thesis as a product.

5. Integrity assurance products (like Karki's thesis [24]) could be integrated with this confidentiality-preserving model to provide more security aspects.

REFERENCES

[1] Amazon rds faqs. https://aws.amazon.com/rds/faqs/. Accessed: 2019-2-23.

[2] Aws for smart, connected and sustainable cities. https://aws.amazon.com/smart-cities/. Accessed: 2018-10-10.

[3] *Cloud Vision 2020: The Future of the Cloud*.

[4] Employees structure. https://dev.mysql.com/doc/employee/en/sakila-structure.html. Accessed: 2019-2-24.

[5] Pricing summary. https://cloud.google.com/storage/pricing-summary/. Accessed: 2019-2-23.

[6] Secram. https://github.com/acs6610987/secram, 2016.

[7] *2017 Global Information Security Workforce Study*. 2017.

[8] Data replication in azure storage, Oct 2018.

[9] Gartner forecasts worldwide public cloud revenue to grow 17.3 percent in 2019. *Gartner*, Sep 2018.

[10] Jsqlparser. https://github.com/JSQLParser/JSqlParser, 2018.

[11] test_db. https://github.com/datacharmer/test_db, 2018.

[12] Bouncycastle. https://www.bouncycastle.org/java.html, 2019.

[13] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, and Ramarathnam Venkatesan. Orthogonal security with cipherbase. In *CIDR*. Citeseer, 2013.

[14] Sumeet Bajaj and Radu Sion. Trusteddb: A trusted hardware-based database with privacy and data confidentiality. *IEEE Transactions on Knowledge and Data Engineering*, 26:752–765, 2011.

[15] Alexandra Boldyreva, Nathan Chenette, and Adam ONeill. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *Annual Cryptology Conference*, pages 578–595. Springer, 2011.

[16] DI Management. *BigDigits multiple-precision arithmetic source code.*

[17] Morris J Dworkin. Sha-3 standard: Permutation-based hash and extendable-output functions. Technical report, 2015.

[18] D Eastlake 3rd and Paul Jones. Us secure hash algorithm 1 (sha1). Technical report, 2001.

[19] Michael Egorov and MacLane Wilkison. Zerodb white paper. *CoRR*, abs/1602.07168, 2016.

[20] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472, 1985.

[21] Luca Gherardi, Davide Brugali, and Daniele Comotti. A java vs. c++ performance evaluation: a 3d modeling benchmark. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, pages 161–172. Springer, 2012.

[22] Paul Grubbs, Kevin Sekniqi, Vincent Bindschaedler, Muhammad Naveed, and Thomas Ristenpart. Leakage-abuse attacks against order-revealing encryption. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 655–672. IEEE, 2017.

[23] Hakan Hacigümüş, Bala Iyer, Chen Li, and Sharad Mehrotra. Executing sql over encrypted data in the database-service-provider model. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 216–227. ACM, 2002.

[24] Ujwal Karki. Integrity coded databases: Ensuring correctness and freshness of outsourced databases. 2017.

[25] Muhammad Naveed, Seny Kamara, and Charles V Wright. Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 644–655. ACM, 2015.

[26] National Institute of Standards and Technology. Secure hash standard (shs). Technical report, 2015.

[27] Bruno C.d.S. Oliveira, Meng Wang, and Jeremy Gibbons. The visitor pattern as a reusable, generic, type-safe component. *SIGPLAN Not.*, 43(10):439–456, October 2008.

[28] Oracle. *Java Platform SE 8*. Class SecureRandom.

[29] Oracle. *Java Platform SE 8*. Class BigInteger.

[30] Oracle. *Java SE Technologies*. Database.

[31] Oracle. *MySQL 8.0 Reference Manual*. Section 11.3.1 The DATE, DATETIME, and TIMESTAMP Types.

[32] Oracle. *MySQL 8.0 Reference Manual*. 11.4.3 The BLOB and TEXT Types.

[33] Oracle. *MySQL 8.0 Reference Manual*. 13.1.15 CREATE INDEX Syntax.

[34] Oracle. *MySQL 8.0 Reference Manual*. Section 28.4 Adding New Functions to MySQL.

[35] Christof Paar and Jan Pelzl. *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009.

[36] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 223–238. Springer, 1999.

[37] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. Arx: A strongly encrypted database system. *IACR Cryptology ePrint Archive*, 2016:591, 2016.

[38] Raluca Ada Popa, Frank H Li, and Nickolai Zeldovich. An ideal-security protocol for order-preserving encoding. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 463–477. IEEE, 2013.

[39] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 85–100. ACM, 2011.

[40] Open Web Application Security Project. Owasp top 10 - 2017. 2017.

[41] Ronald Rivest. The md5 message-digest algorithm. Technical report, 1992.

[42] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[43] Glen Robinson, Attila Narin, and Chris Ellemen. *Using Amazon Web Services for Disaster Recovery*. Oct 2014.

[44] M Stevens, E Bursztein, P Karpman, A Albertini, Y Markov, A Petit Bianco, and C Baisse. Announcing the first sha1 collision. *Google Security Blog*, 2017.

[45] Cong Wang, Qian Wang, Kui Ren, and Wenjing Lou. Privacy-preserving public auditing for data storage security in cloud computing. In *Infocom, 2010 proceedings ieee*, pages 1–9. Ieee, 2010.

APPENDIX A

SELECT STATEMENT LIST

```
SELECT * FROM employees LIMIT 1;
SELECT * FROM employees LIMIT 1000;
SELECT * FROM employees LIMIT 5000;
SELECT * FROM employees LIMIT 10000;
SELECT * FROM employees LIMIT 50000;
SELECT * FROM employees LIMIT 100000;
```

A.1 Where Clause List

```
SELECT * FROM employees WHERE emp_no < 499991 LIMIT 1;
SELECT * FROM employees WHERE emp_no < 499994 LIMIT 1000;
SELECT * FROM employees WHERE emp_no < 499995 LIMIT 5000;
SELECT * FROM employees WHERE emp_no < 499996 LIMIT 10000;
SELECT * FROM employees WHERE emp_no < 499997 LIMIT 50000;
SELECT * FROM employees WHERE emp_no < 499998 LIMIT 100000;
```

A.2 Join List

```
SELECT salary, first_name FROM salaries JOIN employees ON
    salaries.emp_no = employees.emp_no LIMIT 1;
SELECT salary, first_name FROM salaries JOIN employees ON
    salaries.emp_no = employees.emp_no LIMIT 1000;
SELECT salary, first_name FROM salaries JOIN employees ON
    salaries.emp_no = employees.emp_no LIMIT 5000;
SELECT salary, first_name FROM salaries JOIN employees ON
    salaries.emp_no = employees.emp_no LIMIT 10000;
SELECT salary, first_name FROM salaries JOIN employees ON
    salaries.emp_no = employees.emp_no LIMIT 50000;
SELECT salary, first_name FROM salaries JOIN employees ON
    salaries.emp_no = employees.emp_no LIMIT 100000;
```

APPENDIX B

UPDATE STATEMENT LIST

```
UPDATE employees SET last_name = 'Smith' LIMIT 1;
UPDATE employees SET last_name = 'Smith' LIMIT 1000;
UPDATE employees SET last_name = 'Smith' LIMIT 5000;
UPDATE employees SET last_name = 'Smith' LIMIT 10000;
UPDATE employees SET last_name = 'Smith' LIMIT 50000;
UPDATE employees SET last_name = 'Smith' LIMIT 100000;
```

APPENDIX C

DELETE STATEMENT LIST

```
DELETE FROM employees LIMIT 1;
DELETE FROM employees LIMIT 1000;
DELETE FROM employees LIMIT 5000;
DELETE FROM employees LIMIT 10000;
DELETE FROM employees LIMIT 50000;
DELETE FROM employees LIMIT 100000;
```

APPENDIX D

EMPLOYEES DATABASE TABLE AND COLUMN LIST

The database is presented in the format:

table_name

1. column_name
2. ...

Below are the tables and files for the employees database provided by Oracle. The relationship diagram is available on their website [4].

departments

1. dept_no
2. dept_name

dept_manager

1. dept_no
2. emp_no
3. from_date
4. to_date

dept_emp

1. emp_no
2. dept_no
3. from_date
4. to_date

employees

1. emp_no
2. first_name
3. last_name
4. gender
5. hire_date

salaries

1. emp_no
2. salary
3. from_date
4. to_date

titles

1. emp_no
2. title
3. from_date
4. to_date