EDUARDO LOPES COMINETTI

IMPROVING CLOUD BASED ENCRYPTED DATABASES

Master Dissertation presented to the Departamento de Engenharia de Computação e Sistemas Digitais at the Escola Politécnica, Universidade de São Paulo, Brazil to obtain the degree of Master of Science.

São Paulo
2019

EDUARDO LOPES COMINETTI

IMPROVING CLOUD BASED ENCRYPTED DATABASES

Master Dissertation presented to the Departamento de Engenharia de Computação e Sistemas Digitais at the Escola Politécnica, Universidade de São Paulo, Brazil to obtain the degree of Master of Science.

Concentration area:

Computer Engineering

Advisor:

Marcos Antonio Simplicio Junior

São Paulo
2019

Este exemplar foi revisado e corrigido em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, _____ de _____ de _____

Assinatura do autor: _____

Assinatura do orientador: _____

Catalogação-na-publicação

Cominetti, Eduardo Lopes
Improving Cloud Based Encrypted Databases / E. L. Cominetti -- versão corr. -- São Paulo, 2019.
143 p.

Dissertação (Mestrado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1.Criptologia 2.Algoritmos 3.Segurança de computadores 4.Banco de dados I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais II.t.

RESUMO

Bancos de Dados são essenciais para a operação de diversos serviços, como bancos, lojas onlines e até mesmo assistência médica. O custo de manutenção local dessa grande coleção de dados é alto, e a nuvem pode ser utilizada para compartilhar recursos computacionais e atenuar esse problema. Infelizmente, grande parte desses dados pode ser confidencial ou privada, necessitando, portanto, de proteção contra terceiros. Além disso, esses dados precisam ser manipulados para que seu dono consiga extrair informações relevantes. Nesse cenário, bancos de dados cifrados na nuvem que permitam a manipulação de seus dados foram desenvolvidos nos últimos anos. Embora promissoras, as soluções propostas até então apresentam oportunidades de melhorias em termos de eficiência, flexibilidade e também segurança. Neste trabalho, modificações são propostas para o CryptDB, uma solução de banco de dados cifrado na nuvem que faz parte do estado da arte, visando melhorar sua eficiência, flexibilidade e segurança, através do aprimoramento ou troca das primitivas criptográficas utilizadas. A eficiência foi melhorada através da substituição do algoritmo de Paillier presente no CryptDB por um novo algoritmo homomórfico proposto neste trabalho. A flexibilidade foi aprimorada através de uma modificação prévia no texto antes de sua cifração com o algoritmo de Song, Wagner e Perrig, o que permite a busca por wildcards no banco de dados. Por fim, a segurança foi incrementada através da substituição do algoritmo AES em modo CMC na camada determinística do banco de dados pelo algoritmo de Song, Wagner e Perrig.

ABSTRACT

Databases are a cornerstone for the operation of many services, such as banking, web stores and even health care. The cost of maintaining such a large collection of data on-premise is high, and the cloud can be used to share computational resources and mitigate this problem. Unfortunately, a great amount of data may be private or confidential, thus requiring to be protected from agents. Moreover, this data needs to be manipulated to provide useful information to its owner. Hence, encrypted databases that allow the manipulation of data without compromising its privacy have surfaced in the recent years. Albeit promising, the solutions available in the literature can still be improved in terms of efficiency, flexibility and even security. In this work, we propose modifications to CryptDB, a state-of-the-art encrypted cloud database, aiming to enhance its efficiency, flexibility and security; this is accomplished by improving or changing its underlying cryptographic primitives. The efficiency of CryptDB was improved by substituting a new homomorphic algorithm proposed by us for the Paillier cryptosystem. The flexibility of the cloud database was augmented by modifying how a text is encrypted using the Song, Wagner and Perrig algorithm, thus enabling wildcard searches. Finally, the security of the system was enhanced by substituting the Song, Wagner and Perrig algorithm for the AES in CMC mode at the deterministic layer.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ACD	Approximate Common Divisor
ACPC	Activation Codes for Pseudonym Certificates
AES	Advanced Encryption Standard
AJE	Adjustable Join Encryption
BCLO	Boldyreva, Chenette, Lee and O'Neil Order Preserving Encryption
CBC	Cipher Block Chaining
CMC	CBC-Mask-CBC
CRL	Certificate Revocation List
DB	Database
DBMS	Database Management System
DET	CryptDB's Deterministic Layer
EC	Elliptic Curve
ECEGES	Elliptic Curve ElGamal Encryption Scheme
EGES	ElGamal Encryption Scheme
FAHE	Fast Additive Homomorphic Encryption Scheme
FHE	Fully Homomorphic Encryption
GCD	Greatest Common Divisor
HOM	CryptDB's Homomorphic Addition Layer
IND-CCA1	Indistinguishability under non-adaptive Chosen-Ciphertext Attack

IND-CCA2	Indistinguishability under adaptive Chosen-Ciphertext Attack
IND-CPA	Indistinguishability under Chosen-Plaintext Attack
IV	Initialization Vector
JOIN	CryptDB's Join Layer
KDF	Key Derivation Function
mOPE	Mutable Order Preserving Encoding
NIST	National Institute of Standards and Technology
OPE	CryptDB's Order Preserving Layer
OPE-JOIN	CryptDB's Order Join Layer
PHE	Partially Homomorphic Encryption
PHPE	Paillier Homomorphic Probabilist Encryption
PRF	Pseudo-Random Function
PRP	Pseudo-Random Permutation
RND	CryptDB's Random Layer
SCMS	Security Credential Management System
SEARCH	CryptDB's Search Layer
stOPE	Storage-Aware Order Preserving Encoding
SWP	Song, Wagner and Perrig Searchable Encryption
UTF-8	Unicode Transformation Format using 8 bits
UTF-16	Unicode Transformation Format using 16 bits
UTF-32	Unicode Transformation Format using 32 bits
XOR	Exclusive-or operation

LIST OF SYMBOLS

$*$	Database wildcard operator
$a \ll b$	Left Shift operation of string a by b positions
$a \gg b$	Right Shift operation of string a by b positions
$\|\|$	Concatenation operator
$\|x\|$	String x size in bits
$\oplus$	Exclusive-or operator
$\mathbb{F}_p$	Finite Field modulo p
$gcd(a, b)$	Greatest Common Divisor of a and b
$\mathcal{H}$	Cryptographic Hash Function
$\mathcal{O}$	Big O notation

CONTENTS

1 INTRODUCTION

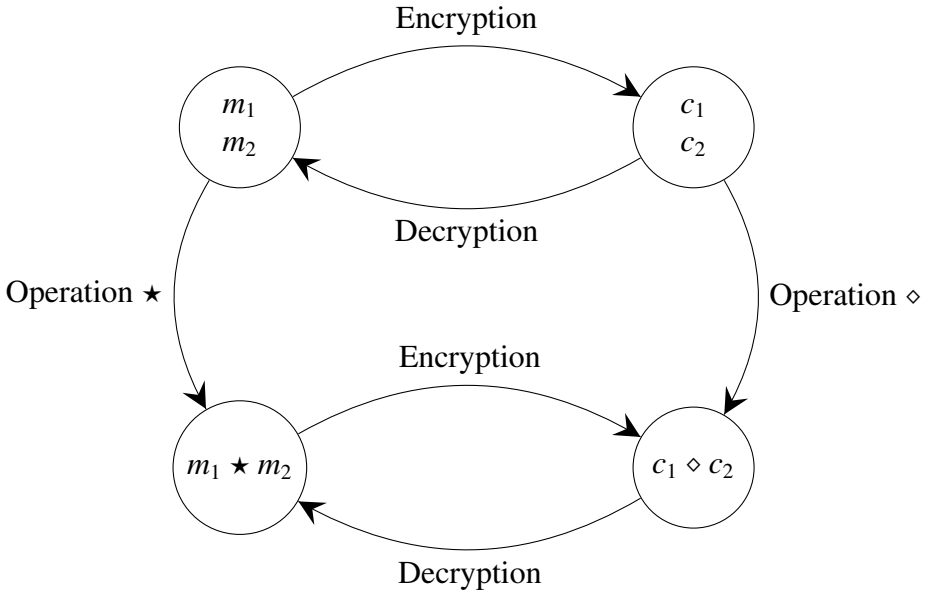
A database is a usually large collection of data organized especially for rapid search and retrieval (Merriam-Webster, 2017). Databases are used in processes that need to correlate information, such as maintaining and operating a web store, which requires the association of a person, his/her address, a purchase, and a payment method to process a sale.

Generally, a database is stored on-premise, which means that its owner is responsible for providing the infrastructure and for maintaining the system. However, on-premise costs for large databases are quite high (BUCKEL, 2013). In comparison, using the cloud to store a database can help mitigate the costs by sharing resources among different companies. (ARASU et al., 2013; ROGGERO, 2013) Unfortunately, though, the data stored may be private or confidential, which is the case for credit card numbers or medical history, for example. Therefore, it must be protected from internal and external agents. As a result, privacy-preserving cloud databases become essential to the deployment of confidential data in the cloud environment.

1.1 Motivation

Security and privacy concerns remain among the major cornerstones for the widespread adoption of cloud solutions (ORACLE, 2015; SCHULZE, 2016). These concerns are legitimate, since the number of online attacks that try to recover confidential data is considerable: only in the United States, more than 5,000 data sets

Figure 1: Homomorphic Encryption: there is a morphism between operation $\star$ performed in the plaintexts m_1 and m_2 and operation $\diamond$ performed in the ciphertexts c_1 and c_2.



Source: Author.

were made public, which translates to more than 9 million records compromised since 2005 (Privacy Rights Clearinghouse, 2017).

Data must be encrypted to prevent its disclosure in case of attacks. However, it is not always possible to use traditional encryption schemes in databases, as then the encrypted data cannot be manipulated and correlated without decryption. This limitation of traditional schemes brings forward a challenge: is it possible to encrypt data and still be able to compute on it without decryption?

In 1978, Rivest, Adleman, and Dertouzos proposed a class of special encryption functions they called "privacy homomorphisms" (RIVEST; ADLEMAN; DERTOUZOS, 1978). These special encryption functions allow encrypted data to be operated without decryption. Figure 1 presents a visual reference of homomorphic encryption. Given two plaintexts m_1 and m_2 and their encryptions c_1 and c_2, there is an operation $\diamond$ performed on the ciphertexts that is equivalent to an operation $\star$ performed on the plaintexts. In other words, the decryption of $c_1 \diamond c_2$ is equal to $m_1 \star m_2$. If they

can be used on a database, the underlying data can be protected and operated on the cloud without revealing classified information. For instance, an addition of multiple rows can be simply achieved by homomorphically adding these rows. Furthermore, convoluted database operations can be expressed as logic gate digital circuits. Hence, if there is an homomorphic encryption that allows any operation to be performed, it can be used to create logic gates and any database operation can be performed homomorphically. Following this concept, many Partially Homomorphic Encryption (PHE) schemes that allow one function to be computed over encrypted data were proposed. Unfortunately, schemes that allow any operation to be performed on encrypted data, called Fully Homomorphic Encryption (FHE), are still impractical for real world applications (BAJAJ; SION, 2011).

FHE's poor performance compels cloud database developers to adopt new and creative strategies. One of such strategies is to use a collection of different encryption functions, each one allowing a specific database operation to be performed. An example is CryptDB, a cloud database designed by MIT in 2011 (POPA et al., 2011). It uses a myriad of encryption schemes and encryption modes to permit the database to operate on data without publicly exposing it. As the solution relies on many different algorithms, overall database performance, functionality and even security is greatly affected by their individual behaviour. Thus, the study, improvement and modification of these algorithms is essential to enhance cloud databases and make them a better alternative to on-premise systems.

1.2 Goals

Our main goal is to improve the cryptographic schemes used on privacy preserving cloud databases in order to enhance their security, functionality and efficiency. To accomplish this, we use CryptDB as a basis, for it is considered one of the state-of-the-art privacy preserving cloud databases and its framework and source code are publicly

available.

1.3 Related Works

In this section, we present some cloud database solutions that provide data security, giving an overview of the state-of-the-art. The solutions are presented in chronological order.

1.3.1 CryptDB

CryptDB (POPA et al., 2011) is an encrypted database designed by MIT. As it is the basis for this work, a more thorough description is presented in chapter 3.

CryptDB uses a (User)-(Secure Proxy)-(Server) framework, so that (1) the user interacts with the secure proxy as if the proxy was a plain database, and (2) the secure proxy is responsible for encrypting data and storing it on the server. The data encryption by the secure proxy in done in "onion layers". There are multiple onions for each data and each onion has multiple layers. Data is encrypted from "more information revealing" layers to "less information revealing" layers. Each layer uses a different algorithm and is responsible for a specific database operation. Moreover, CryptDB's secure proxy is responsible for data encryption, storage of keys and conversion of a plain SQL query provided by the user to an encrypted query sent to the server.

Improvements to CryptDB's security, functionality and efficiency can be achieved by changing individual layer algorithms, since each layer deals with specific SQL operations. This modularity is an important feature of CryptDB, since it allows a more structured analysis when looking for improvement opportunities.

1.3.2 Cipherbase

Cipherbase (ARASU et al., 2013) is a cloud database that relies on secure hardware to maintain privacy. Its architecture is divided in two groups. The first group is known as “Untrusted Machine”, which comprises the traditional cloud environment. The second group, called “Trusted Machine”, comprises trusted Field-Programmable Gate Arrays (FPGAs) deployed inside the cloud by trusted authorities.

Data is stored encrypted in the unstrusted machine and, when some computation on it is required, it is sent to the trusted machine. The trusted machine decrypts the data, performs the requested operation, and finally encrypts the result and corresponding data before returning them to the untrusted machine. For some functions, PHEs can be used on data so it can be operated directly on the untrusted machine.

This approach requires the deployment and maintenance of specific hardware on the cloud environment. Moreover, it also requires the existence of a trusted third party that will be able to access the whole database.

1.3.3 Ciphercloud

Ciphercloud (CipherCloud, 2015) is a patented cloud database system that relies on AES to provide cloud security. Its white paper describes the use of AES in tokenization schemes, local stored mapping tables, and secure gateways to provide functionality while maintaining database privacy. Such description does not provide enough information for a security analysis of the solution. The only achievable conclusion is that Ciphercloud provides security through obscurity.

1.3.4 ZeroDB

ZeroDB (EGOROV; WILKISON, 2016) is an end-to-end encrypted database that performs its operations by traversing b-trees. This is achieved with a client-server

cooperation.

Data is encrypted into buckets at client-side. Afterwards, these buckets are stored and indexed logically on a b-tree at server-side.

Whenever the client queries the database, it chooses a tree whose logical index matches the query. The server sends the client the tree root for decryption and analysis. After the decryption, the client is able to compare the data with the desired data and informs the server to which sequential leaf it should go. This process is repeated until the leaf with the desired data is reached.

AES or any other secure cipher is used to encrypt the client information and PHE can be used to perform some operations without the need to send the client every matching bucket. Additionally, frequent and small sub-trees can be prefetched to the client to speed up the system.

This approach relies heavily on client-server communication and information exchange, thus network traffic is a constraining factor.

1.3.5 Arx

Arx (PODDAR; BOELTER; POPA, 2016) is an encrypted database that builds upon ZeroDB. It provides a series of schemes to enable database operations. Arx-EQ, Arx-AGG and Arx-Range enable equality checks, aggregations and order operations respectively. Arx-EQ is constructed using a block cipher and a key derivation function, KDF. Arx-AGG is implemented using Paillier's cryptosystem (PAILLIER, 1999). Arx-Range expands over the ZeroDB idea. Arx-Range also uses a binary tree to implement order. Differently from ZeroDB, though, this tree is constructed with garbled circuits (YAO, 1986). Garbled circuits allow the tree to be traversed without exposing the searched value. The garbled circuit receives an encrypted value and outputs another encrypted value together with the next leaf to be used (left or right). This allows

the tree to be traversed without a client-server interaction.

Although "order by" operations are now performed without overburdening the network, garbled circuits maintain the secrecy property only for a single use. Therefore, every time the binary tree is used, the database must destroy all utilized nodes and the client must reconstruct them. Another problem is that a tree can be used by just one client, which limits the application's parallelism capabilities.

1.4 Contributions

Aiming to improve the CryptDB database system, in this work we:

1. enhanced CryptDB's homomorphic layer performance by up to 1300 times through the replacement of the layer's algorithm by a novel PHE scheme presented in chapter 4, at the cost of additional storage space;
2. enhanced CryptDB's deterministic layer performance by up to 7.4 times through the replacement of the layer's algorithm by a hash function, at the cost of additional storage space;
3. enabled wildcard search, thus expanding the system's functionality, through the modification of the CryptDB's search layer algorithm, at the cost of additional storage space and additional performance overhead; and
4. improved CryptDB's deterministic layer security through the replacement of CryptDB's deterministic layer algorithm by CryptDB's search layer algorithm, at the cost of additional perfomance overhead.

1.5 Outline

The rest of this document is organized as follows. Chapter 2 presents the concepts, algorithms, and encryption schemes already in use by CryptDB database. Chapter 3

gives a more in-depth view of CryptDB database system, discussing its limitations and flaws. Chapter 4 introduces a novel symmetric PHE, created specifically for the scenario of privacy-preserving cloud databases. Chapter 5 presents our modifications to CryptDB together with their security analysis. The results of these modifications are analyzed in Chapter 6. Finally, we present our concluding remarks and ideas for future work in Chapter 7.

2 BUILDING BLOCKS

In this chapter, we present fundamental concepts and algorithms that are required to understand and construct CryptDB and our solution.

2.1 Some Basic Notation

The symbol $\oplus$ is used to represent the exclusive-or (XOR) operation. The symbol $||$ is used to represent the concatenation operation.

2.2 Pseudo-Random Function (PRF), Pseudo-Random Permutation (PRP) and Block Cipher

The definitions for Pseudo-Random Function (PRF), Pseudo-Random Permutation (PRP) and Block Cipher are taken from (BELLARE; ROGAWAY, 2005) and (GOLDWASSER; BELLARE, 1996).

Let $F : K \times D \rightarrow R$ be a family of functions, where $K = \{0,1\}^k$ is the key space of F, $D = \{0,1\}^l$ is the domain of F, $R = \{0,1\}^L$ is the range of F, and $k, l, L \geq 1$.

F is a **pseudo-random function** (PRF) if:

1. there is a polynomial-time algorithm to compute F and
2. a random instance of F is poly-time indistinguishable from a random function.

"Poly-time indistinguishable" means that there is no adversary A who can distinguish F from a real random function f with a probability greater than $\frac{1}{Q(n)}$, where $Q(n)$ is a polynomial and n is a security parameter.

F is a **pseudo-random permutation** (PRP) if:

1. F is a PRF;
2. F is a bijection with $D = R$ and
3. there is a polynomial-time algorithm to compute F^{-1}.

A **block cipher** is a family of PRPs with fixed K, D and R. Block ciphers are **symmetric** as they use the same key $k \in K$ to encrypt and to decrypt data. Block ciphers are **deterministic** as their output is always the same for a given input under the same key. The Advanced Encryption Standard (AES) is an example of block cipher.

2.2.1 Advanced Encryption Standard (AES)

The **Advanced Encryption Standard** (AES) is a block cipher with $K = \{0,1\}^k$, $D = \{0,1\}^n$, $R = \{0,1\}^n$, where $k = 128$ or $k = 192$ or $k = 256$ and $n = 128$. It was standardized by the National Institute of Standards and Technology (NIST) in 2001 (Federal Information Processing Standards Publication 197, 2001).

AES is based on the Rijndael cipher (DAEMEN; RIJMEN, 1999) and is considered to be a secure cipher (BARKER; ROGINSKY, 2015). AES works by first organizing the plaintext (i.e., the unencrypted information that needs to be secured) in a 4×4 matrix, where every element of the matrix corresponds to one byte of the plaintext. The encryption key is also organized in a 4×4 matrix. Next, it performs n rounds of encryption, where $n = 10$ if $k = 128$, $n = 12$ if $k = 192$, and $n = 14$ if $k = 256$. The encryption key matrix is expanded, producing $n + 1$ RoundKey 4×4 matrices. Each round but the last performs the following transformations:

1. **SubBytes**: each byte of the matrix is substituted by another according to a lookup table;
2. **ShiftRows**: each matrix row is cyclically left shifted according to its number, the first row is not shifted, the second is shifted one position, the third is shifted two positions, and the fourth is shifted three positions;
3. **MixColumns**: each matrix column is left multiplied by a fixed 4×4 matrix;
4. **AddRoundKey**: each byte of the matrix is added to the corresponding round key byte

Additions are performed using the exclusive-or (XOR) operation, and for multiplications the vectors are treated as polynomials over a finite field modulo the polynomial $x^8 + x^4 + x^3 + x + 1$. The last AES round is similar to the intermediate rounds, the main difference being that it does not execute the *MixColumns* step. Having defined how each round is set, AES's overall operation can be described as follows:

1. The plaintext is organized in a matrix and the key is organized in a matrix and expanded;
2. An *AddRoundKey* with the first RoundKey matrix is performed;
3. The intermediate rounds are executed using each one their RoundKey matrix;
4. The last round is executed using the last RoundKey matrix;

The decryption process follows the reverse order of these steps, so the transformations in each round are also individually applied in reverse order.

Like any block cipher, AES provides security for just one block of plaintext, comprising 128 bits. To provide security for larger plaintexts, AES must be used together with a mode of operation.

2.2.2 Modes of Operation

A **mode of operation** is an algorithm that uses as its core a block cipher to provide security for plaintexts larger than one block(NIST, 2017). Although many modes of operation exist, we focus on **Cipher Block Chaining** (CBC), **Counter** (CTR), and **CBC-Mask-CBC** (CMC) as these modes are used in CryptDB and in the solutions hereby proposed.

2.2.2.1 Cipher Block Chaining (CBC)

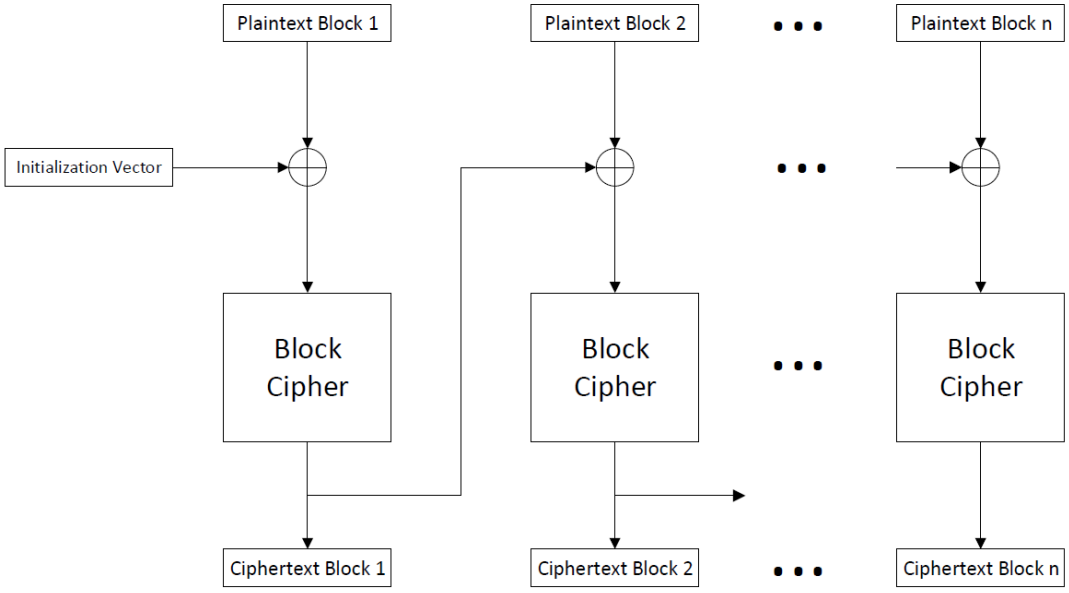
Cipher Block Chaining (CBC) was patented in 1978 by Ehrsam, Meyer, Smith and Tuchman (EHRSAM et al., 1978). It divides a large plaintext in multiple AES-sized blocks, i.e., 128 bits. If the plaintext size is not a multiple of the AES-size block, it is padded: bits are added to the plaintext until it achieves the block length. There are various recommended padding schemes, one of which is PKCS#7 (IETF, 2009, Section 6.3).

Once the (padded) plaintext is divided into AES-size blocks, numbered from 1 to n, the encryption follows these steps:

1. Plaintext block 1 is XORed with a random **initialization vector** (IV);
2. The new block is encrypted with the AES producing the **ciphertext** block 1;
3. Ciphertext block 1 is XORed with plaintext block 2;
4. The new block is encrypted with the AES producing the ciphertext block 2;
5. The process is repeated until plaintext block n is encrypted.

Figure 2 illustrates the CBC mode of operation. The **initialization vector** is a 128 bits value chosen at random, which guarantees that identical plaintext blocks

Figure 2: CBC encryption.



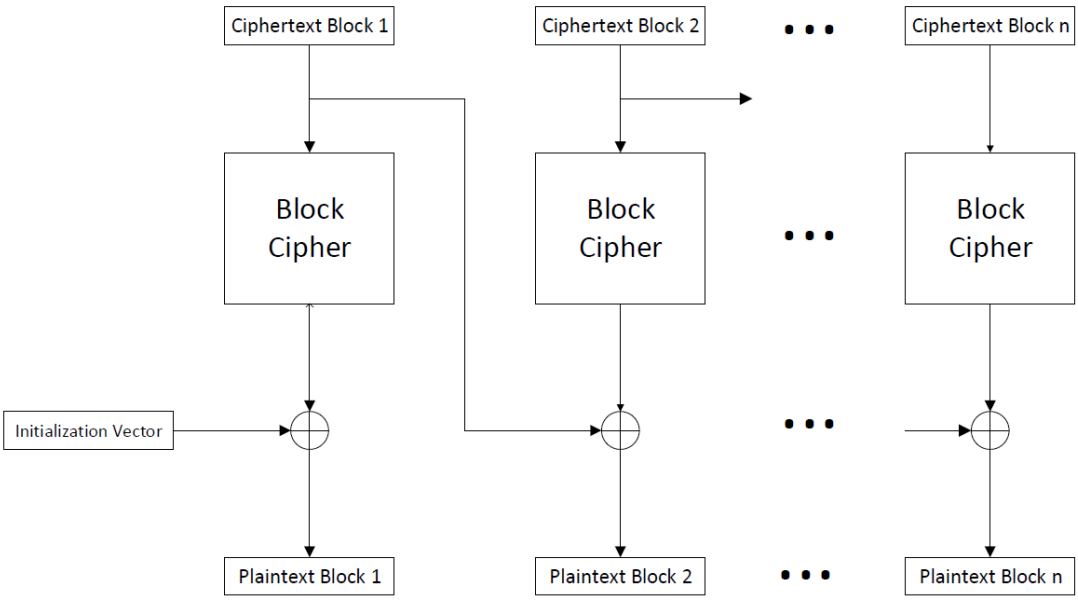
Source: Author.

are encrypted into different ciphertext blocks. A **ciphertext** is the encrypted result of a plaintext. Thus, CBC is a **probabilistic** algorithm, since identical plaintexts are encrypted into different ciphertexts. The resulting ciphertext is set as $IV \,||\, ciphertext\,1 \,||\, ciphertext\,2 \,||\, \cdots \,||\, ciphertext\,n$.

To decrypt the ciphertext, it must be divided again in blocks and the process can be reverted by applying the following steps (and illustrated in Figure 3):

1. Ciphertext block 1 is decrypted by the AES;
2. The decrypted vector is XORed with the IV;
3. The resultant vector is plaintext block 1;
4. Ciphertext block 2 is decrypted by the AES;
5. The decrypted vector is XORed with ciphertext block 1;
6. The resultant vector is plaintext block 2;
7. The process is repeated until ciphertext block n is decrypted.

Figure 3: CBC decryption.



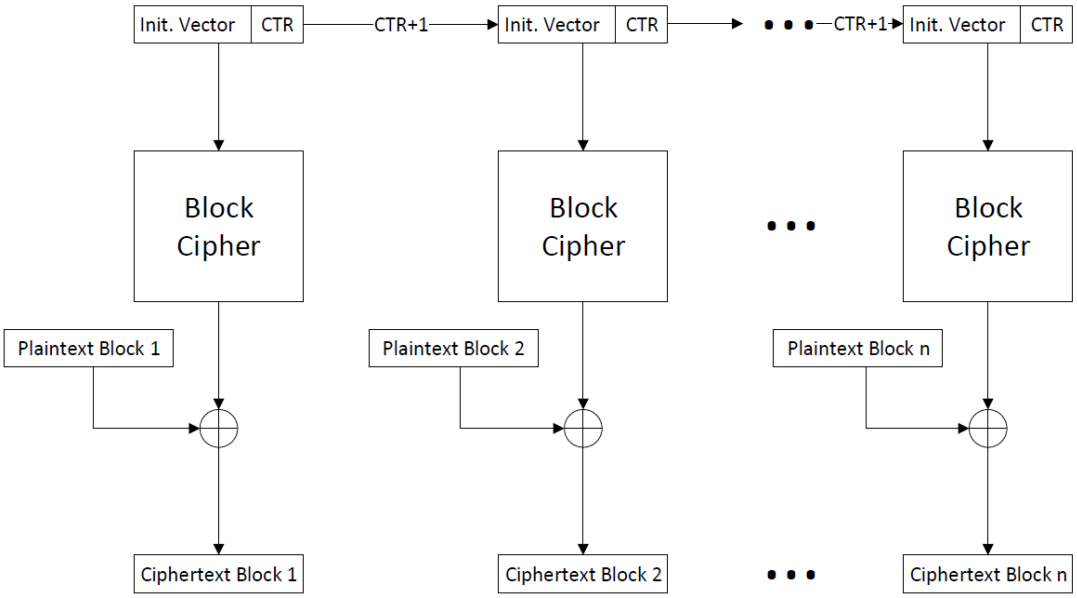
Source: Author.

2.2.2.2 Counter (CTR)

Counter (CTR) was proposed by Diffie and Hellman in 1979 (LIPMAA; ROGAWAY; WAGNER, 2000). It uses a $(blocksize - x)$ bits random IV concatenated with a x bits counter, $IV \,||\, ctr$. The plaintext is divided into AES-size blocks, from 1 to n. Differently from CBC, though, the plaintext does not need to be padded. The last block remains with any number of bits less than or equal to the AES-size block. The counter is first set to 0 and is incremented by one every plaintext block. The algorithm follows these steps:

1. $IV \,||\, ctr$ is encrypted with the AES;
2. The result is XORed with plaintext 1 producing ciphertext 1;
3. The counter is incremented by one;
4. The process is repeated until plaintext block n is encrypted.

Figure 4: CTR encryption.



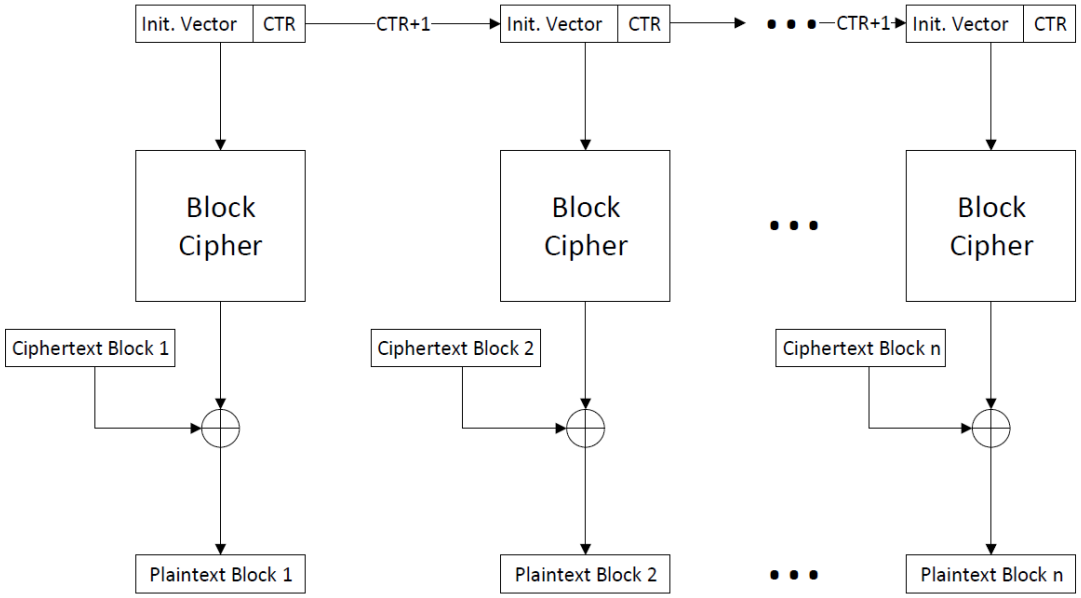
Source: Author.

Figure 4 illustrates CTR encryption process. Similar to CBC, the randomness in IV also makes CTR mode probabilistic. Another approach is to choose a random IV, cipher it with a different key and treat the result vector as a number which would serve as a counter. The rest of the algorithm remains the same. The final ciphertext is set as $IV \,||\, ciphertext\,1 \,||\, ciphertext\,2 \,||\, \cdots \,||\, ciphertext\,n$. The decryption process is similar to the encryption and follow these steps:

1. $IV \,||\, ctr$ is encrypted with the AES;
2. The result is XORed with ciphertext 1 producing plaintext 1;
3. The counter is incremented by one;
4. The process is repeated until ciphertext block n is decrypted.

Figure 5 shows CTR decryption process. Notice that the encryption and decryption circuits are symmetric. This is an implementation advantage, as the mode just needs to be written once.

Figure 5: CTR decryption.



Source: Author.

2.2.2.3 CBC-Mask-CBC (CMC)

CBC-Mask-CBC (CMC) was introduced by Halevi and Rogaway in 2003 (HALEVI; ROGAWAY, 2003). It is a mode of operation proposed to encrypt disk sectors. It works executing CBC twice on the plaintext. First, it uses CBC as usual on the plaintext. Next, it calculates a mask and XORs it with every ciphertext block. Finally, it executes an algorithm similar to CBC decryption from the last to the first block. The overall process follows these steps:

1. CBC is applied to the plaintext, creating the intermediate ciphertext blocks ct' 1 to n;
2. A mask $M = 2(ct'\ 1 \oplus ct'\ n)$ is created;
3. The mask is applied to every ciphertext block ct' creating ciphertext blocks ct'' 1 to n;
4. Ciphertext block $ct''\ n$ is encrypted using AES resulting in $ct'''\ n$;

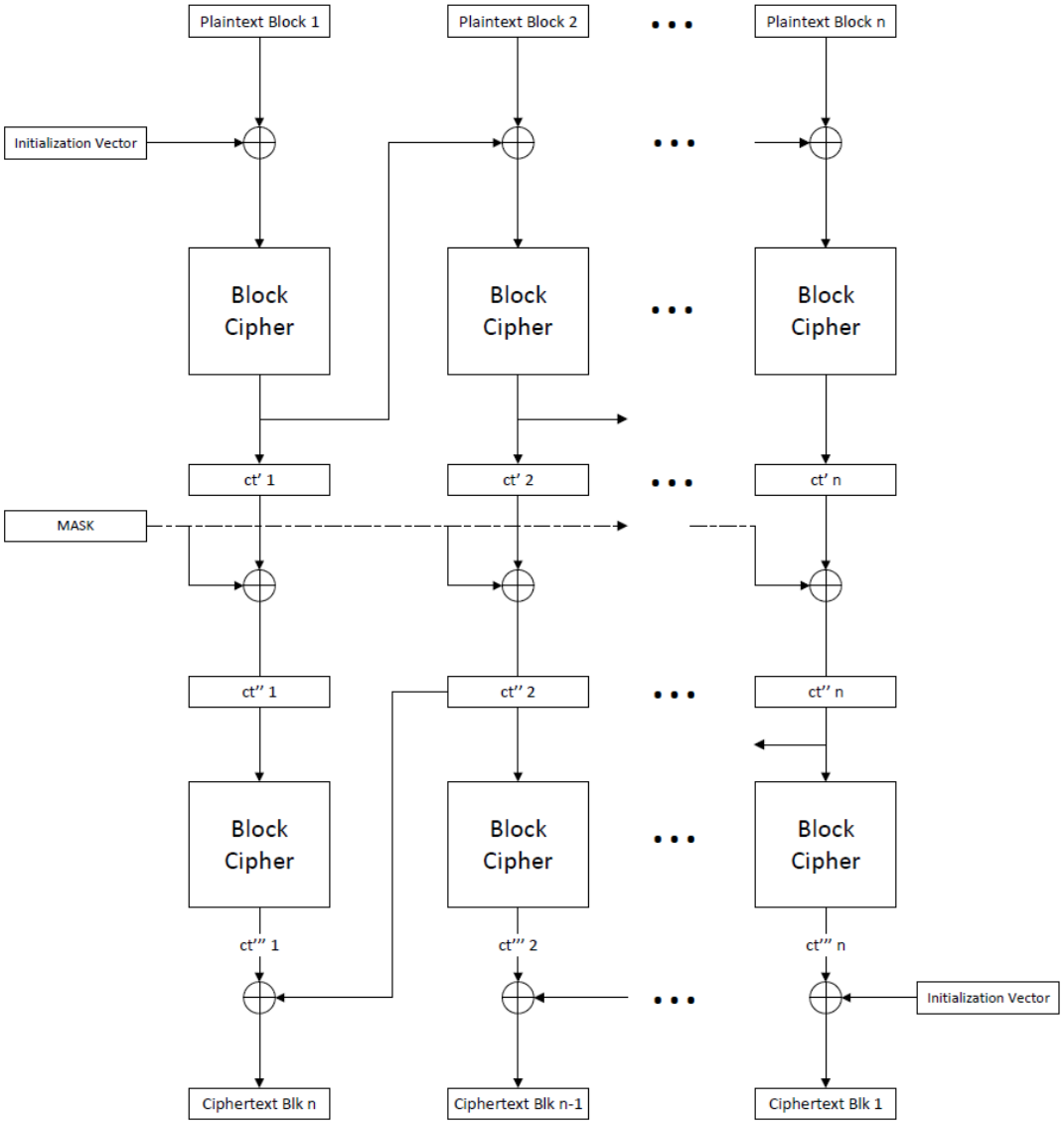
5. Ciphertext block $ct''' n$ is XORed with the IV creating the final ciphertext block 1;

6. Ciphertext block $ct''' (n-1)$ is XORed with $ct'' n$ creating the final ciphertext block 2;

7. Steps 4 to 6 are repeated until the final ciphertext block n is created.

The symbol $\oplus$ represents the XOR operation. Figure 6 illustrates CMC encryption pro-

Figure 6: CMC encryption.



Source: Author.

cess. The final ciphertext is set as $IV \,||\, ciphertext\,1 \,||\, ciphertext\,2 \,||\, \cdots \,||\, ciphertext\,n$. The decryption process follows these steps:

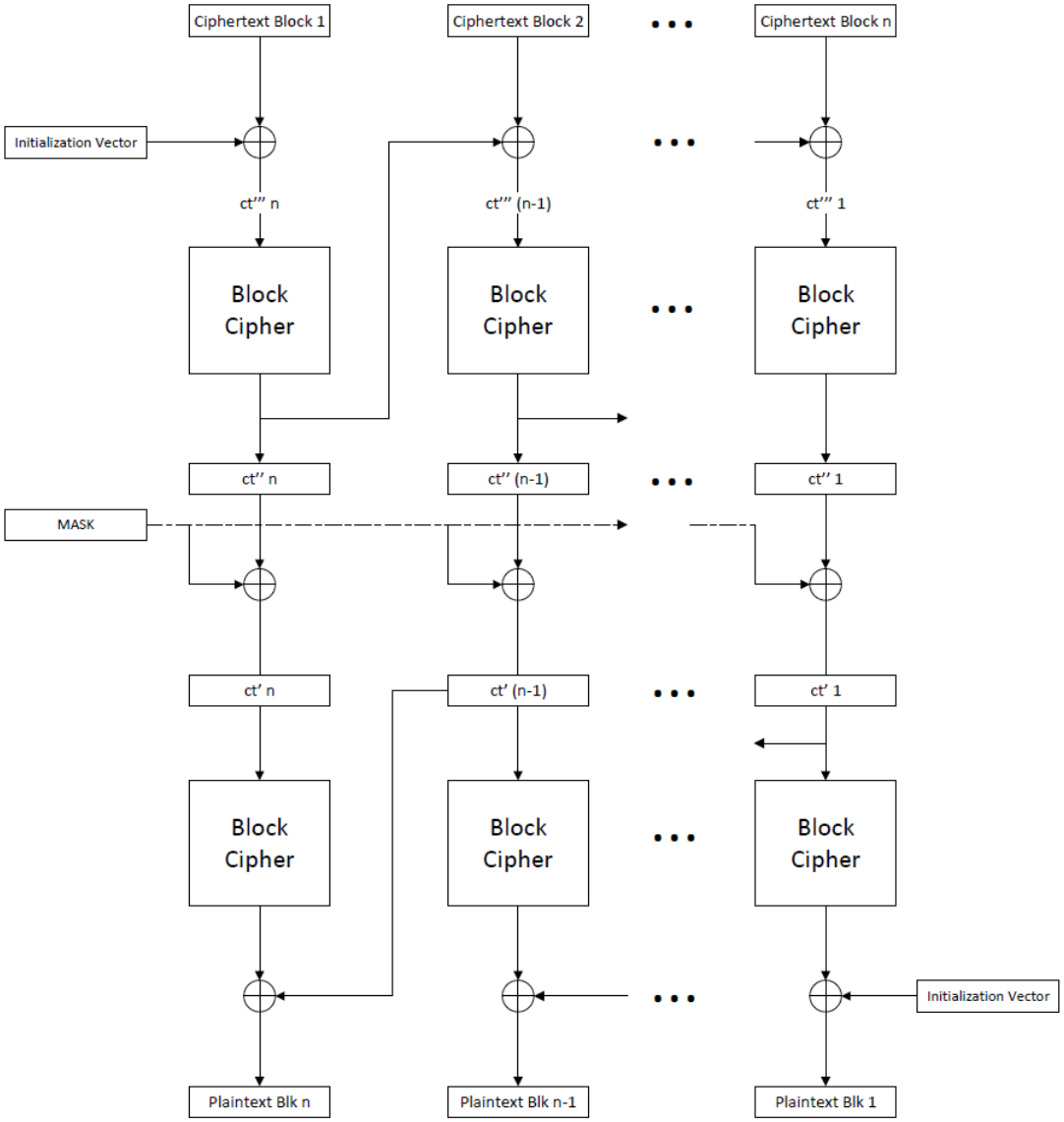
1. Ciphertext block 1 is XORed with the IV, resulting in $ct'''\,n$;
2. $ct'''\,n$ is decrypted using AES producing $ct''\,n$;
3. Ciphertext block 2 is XORed with $ct''\,n$ resulting in $ct'''\,(n-1)$;
4. $ct'''\,(n-1)$ is decrypted using AES producing $ct''\,(n-1)$;
5. Steps 3 to 4 are repeated until $ct''\,1$ to $ct''\,n$ are generated;
6. The mask $M = 2(ct''\,1 \oplus ct''\,n)$ is created;
7. The mask is applied to every ciphertext block ct'' creating ciphertext blocks ct' 1 to n;
8. CBC decryption is applied to ciphertext $ct'\,1 \;||\; ct'\,2 \;||\; \cdots \;||\; ct'\,n$;
9. The decryption will output the final plaintext blocks 1 to n.

Figure 7 shows CMC decryption process. Notice that CMC encryption and decryption circuits are also symmetric.

It is possible to define and evaluate the security of modes of operation. To do so, a scheme can be evaluated by its goals and the possible attack models (BELLARE et al., 1998). In this work, we use the definition of indistinguishability of encryptions and the three different attacks. They are indistinguishability under chosen-plaintext and indistinguishability under chosen-ciphertext (non-adaptative and adaptative).

The presented modes of operation, CBC, CTR and CMC, only achieve Indistinguishability under Chosen-Plaintext (IND-CPA) security as they do not provide ciphertext integrity (BONEH, 2017, Week 2, Week 4).

Figure 7: CMC decryption.



Source: Author.

2.2.3 Indistinguishability of Encryptions

Indistinguishability of encryptions is defined as the result of an experiment. Let A be an adversary regarded as two polynomial-time algorithms, A_1 and A_2.

In A_1, the adversary can ask for the encryption of messages of the same length. Same length messages are a requirement since encrypted ciphertexts reveal information about the original plaintext length. At the end of A_1, the adversary outputs a triple

(m_0, m_1, s), where m_0 and m_1 are different messages of the same length and s is a state information, which can be all previously encrypted messages. As a requirement, m_0 and m_1 must not have been previously encrypted in A_1.

In A_2, the adversary is given the output of A_1 plus a challenge ciphertext y, which is the encryption of either m_0 or m_1. The adversary goal is to determine $b = \{0, 1\}$ such that m_b is the plaintext corresponding to ciphertext y. The adversary is successful if he can, with some advantage, *distinguish* the two messages m_0 and m_1. The adversary fails if he cannot make such distinction with a non-negligible probability. If the adversary fails, we consider that the encryption algorithm passes the experiment. Otherwise, we consider that the encryption algorithm fails the experiment. This probability, Adv_A, is given by:

$$Adv_A = 2 \cdot Pr[(m_0, m_1, s) \leftarrow A_1^{atk}; b \leftarrow \{0, 1\};$$
$$y \leftarrow Encryption(m_b) : A_2^{atk}(m_0, m_1, s, y) = b] - 1$$

where $x \leftarrow A$ denotes either that x is the output of algorithm A (if A is an algorithm), or x is the uniform selection of an element from A (if A is a set). The parameter *atk* represents each of the possible attacks. The adversary capabilities changes in A_1 and A_2 depending on which attack is considered. For *atk=cpa*, a chosen-plaintext attack is considered; *atk=cca1* represents a non-adaptive chosen-ciphertext attack, and *atk=cca2* is an adaptive chosen-ciphertext attack. The probability is considered negligible for $Adv_A < 2^{-k}$. The parameter k is the level of security, in bits, achieved by the encryption scheme. In AES and its modes of operation, k is equal to the chosen key size.

2.2.3.1 Indistinguishability under Chosen-Plaintext

A **chosen-plaintext attack** is the simplest form of attack possible.

In a chosen-plaintext attack, the adversary A is only capable of *encrypting* plain-

texts both in A_1 and A_2. The only restriction is that he cannot ask for the encryption of m_0 or m_1 in A_2. If by just encrypting different plaintexts the adversary is capable of distinguish between the encryption of plaintexts m_0 and m_1, then the attack is successful. Otherwise, the attack fails and the scheme is said to be **indistinguishable under a chosen-plaintext attack** (IND-CPA).

2.2.3.2 Indistinguishability under Chosen-Ciphertext

There are two different **chosen-ciphertext attacks** possible for an adversary. First, we describe the simplest one, the non-adaptive, and next the more complex adaptive variant.

In a **non-adaptive chosen-ciphertext attack**, the adversary has the same capabilities as the chosen-plaintext attack. Moreover, he is also capable of *decrypting* ciphertexts in A_1. If the adversary is capable of distinguishing between the encryption of m_0 and m_1 given this improved capability, the attack is successful. If not, the attack fails and the scheme is said to be **indistinguishable under a non-adaptive chosen-ciphertext attack** (IND-CCA1).

For the more complex **adaptive chosen-ciphertext attack**, the adversary has the same capabilities as the simpler non-adaptive attack. But, furthermore, he is also capable of *decrypting* ciphertexts in A_2. The restriction is the impossibility of directly decrypting the challenge ciphertext y. Similar to the previous scenarios, if he can distinguish between plaintexts m_0 and m_1, he is successful. Otherwise, he fails and the scheme is said to be **indistinguishable under an adaptive chosen-ciphertext attack** (IND-CCA2).

The number following the acronym can be viewed as the stage, 1 or 2, at which the adversary is still capable of decrypting ciphertexts.

2.2.4 Cryptographic Hash Function ($\mathcal{H}$)

A **Cryptographic Hash Function** ($\mathcal{H}$) is a function $\mathcal{H} : D \rightarrow R$, where $D = \{0,1\}^t$ and $R = \{0,1\}^n,\ t > n$ (MENEZES; OORSCHOT; VANSTONE, 1996, Chapter 9). It also needs to satisfy the following properties:

1. **Compression**: $\mathcal{H}$ maps an input string from an arbitrary length up to t bits into a fixed length string of n bits.

2. **Ease of Computation**: $\mathcal{H}$ is easy to compute for an input string x.

3. **Preimage Resistance**: Given an output string y of $\mathcal{H}$, it is infeasible to compute x such that $\mathcal{H}(x) = y$.

4. **2nd-preimage Resistance**: Given an input x, it is infeasible to find an input $x' \neq x$ such that $\mathcal{H}(x') = \mathcal{H}(x)$.

5. **Collision Resistance**: It is infeasible to find $x \neq x'$ such that $\mathcal{H}(x) = \mathcal{H}(x')$ (Note: differently from property 4, the choice of x, x' is free)

By the pigeonhole principle, the probability of mapping a message of size t bits to the same output of n bits is 2^{-n}. The **pigeonhole principle** states that given a itens to be put in b containers, $a > b > 0$, at least one of these containers will have more than 1 item. This gives that the infeasibility of property 4 is true as long as 2^{-n} is small enough.

By the birthday paradox, finding two different x, x' that provide the same output $\mathcal{H}(x) = \mathcal{H}(x') = y$ with a high probability takes $2^{n/2}$ tries. The **birthday paradox** states that given a population set P which has the property $\alpha \in S$, being S an equiprobable set of size m, the number i of elements $p_i \in P$ that must be drawn until $\alpha_{p_{i'}} = \alpha_{p_i}$, for $0 < i' < i$, is $i = \sqrt{m}$. This gives that the infeasibility of property 5 is true as long as $2^{-n/2}$ is small enough.

As $2^{-n/2}$ is bigger than 2^{-n}, the security parameter for a hash function $\mathcal{H}$ is always half of its output length. At the time of this writing, examples of good secure cryptographic hash functions are SHA-2 (Federal Information Processing Standards Publication 180-4, 2015), SHA-3 (Federal Information Processing Standards Publication 202, 2015), and BLAKE2 (SAARINEN; AUMASSON, 2015). Although SHA-1 was considered a secure hash by NIST in the previous papers, it is **no longer** secure and **should not** be considered a secure cryptographic hash function (BARKER; ROGINSKY, 2015; STEVENS et al., 2017).

2.3 Other Encryption Algorithms

Apart from the concepts and algorithms previously presented, there are other encryption algorithms of interest to this work, described in what follows.

2.3.1 ElGamal Encryption Scheme (EGES)

The **ElGamal Encryption Scheme** (EGES) was proposed by Taher ElGamal in 1984 (ELGAMAL, 1984). The security of the algorithm is based on the difficulty of computing discrete logarithms over cyclic groups. Let p be the prime order of the multiplicative cyclic group G and let g be its generator. A generator g is an element of this group so that $\forall a \in G$, there is an integer j such that $g^j = a$ (LIDL; NIEDERREITER, 1994, Chapter 1). The computation of a discrete logarithm over the cyclic group G of order p is finding the solution x in $a = g^x$, where a, g, and p are known.

EGES is an asymmetric encryption scheme. An **asymmetric encryption** scheme is an encryption where there are two keys, known as public key and private key. The **public key** is a key publicly disclosed and is used to encrypt a plaintext. The **private key** is a key kept in secrecy and used to decrypt a ciphertext. Both keys are correlated and the computation of the private key from the public key is infeasible.

Let A and B be the users Alice and Bob. Bob wants to send a message to Alice. A starts by choosing a prime p, a generator $g \in G$ and a private key $k \in G$, $G = \{1, 2, \ldots, p-1\}$. A computes the public key $pk = g^k \mod p$ and sends B the tuple (G, p, g, pk). B then chooses $y \in G$ and calculates $c_1 = g^y \mod p$ and $h = pk^y \mod p$. Let m be the plaintext Bob wants to send to Alice mapped in G. B proceeds by computing $c_2 = m \cdot h \mod p$ and sending Alice the tuple (c_1, c_2). Now, to reveal m, A calculates $h = c_1^k \mod p$ and finds its inverse h^{-1}. Finally, $c_2 \cdot h^{-1} \mod p = m \cdot h \cdot h^{-1} \mod p = m$.

As a remark that will be later used, observe that for a given (c_1, c_2) that encrypts message m and (c_1', c_2') that encrypts message m', the tuple $(c_1'', c_2'') = (c_1 \cdot c_1', c_2 \cdot c_2')$ encrypts the message $m'' = m \cdot m'$. This is a **multiplicative homomorphism**.

2.3.2 Elliptic Curve ElGamal Encryption Scheme (ECEGES)

The **Elliptic Curve ElGamal Encryption Scheme** is a modification to the original EGES proposed by Neal Koblitz in 1987 (KOBLITZ, 1987). Instead of using integers and the discrete logarithm problem, it uses elliptic curves (EC) and the discrete logarithm problem over elliptic curves.

"An **elliptic curve** E_K defined over a field K of characteristic $\neq 2$ or 3 is the set of solutions $(x, y) \in K^2$ to the equation $y^2 = x^3 + ax + b$, $a, b \in K$ (where the cubic on the right has no multiple roots)." (KOBLITZ, 1987). The set of points that satisfies this equation is an additive cyclic group with the addition operation defined by the chord-tangent composition. A more detailed and extended explanation can be found in (SILVERMAN, 2009, Chapter III).

The protocol works similarly to the classical EGES, substituting the integer equations by EC equations. Let, again, A and B be the users Alice and Bob. Bob wants to send Alice a message. A chooses an EC E_K, and an EC generator G. A also picks a private key k and computes the public key $P = k \cdot G$. The tuple (E_K, G, P) is sent to

B. B then picks an integer y and calculates $C_1 = y \cdot G$ and $H = y \cdot P$. B proceeds by mapping the plaintext into an EC point M and produces $C_2 = M + H$. B finally sends A the tuple (C_1, C_2). To reveal the message M, A computes $H = k \cdot C_1$ and performs the subtraction $C_2 - H = M + H - H = M$.

Differently from the EGES, note that for a given (C_1, C_2) that encrypts message M and (C'_1, C'_2) that encrypts message M', the tuple $(C''_1, C''_2) = (C_1 + C'_1, C_2 + C'_2)$ encrypts the message $M'' = M + M'$. Hence, this scheme provides **additive homomorphism**.

2.3.3 Paillier Homomorphic Probabilistic Encryption (PHPE)

The **Paillier Homomorphic Probabilistic Encryption** (PHPE) is an encryption scheme proposed by Pascal Paillier in 1999 (PAILLIER, 1999). It is an asymmetric probabilistic scheme based on the composite residuosity class problem. Consider an integer $n = p \cdot q$, p, q primes and λ the Carmichael function of n (CARMICHAEL, 1914, Chapter 4.6). Let $x \in \mathbb{Z}_n$, $y \in \mathbb{Z}_n^*$, $g, w \in \mathbb{Z}_{n^2}^*$, where $\mathbb{Z}_n$ is the set of integers modulo n, $\mathbb{Z}_n^*$ is $\mathbb{Z}_n \setminus \{0\}$, $\mathbb{Z}_{n^2}^*$ is $\mathbb{Z}_{n^2} \setminus \{0\}$, and the order of g is different from a nonzero multiple of n. The order of an element a from a group A is the number of elements in the subgroup of A consisting of all the powers of a (LIDL; NIEDERREITER, 1994, Chapter 1). The n-th residuosity class of w with respect to g is the unique number $x \in \mathbb{Z}_n$ for which there exists $y \in \mathbb{Z}_n^*$ such that

$$g^x \cdot y^n \mod n^2 = w$$

The **n-th composite residuosity class problem** is the problem of finding the number x when w and n are known. This is considered a hard problem (BENALOH, 1987, Chapter 2.8: The Prime Residuosity Assumption). Additionally, for $S_n = \{u < n^2 \mid u \equiv 1 \mod n\}$, let $L(u) = (u - 1)/n$.

Let A and B be the users Alice and Bob. Bob wants to send a message to Alice. A starts by choosing $n = p \cdot q$, p, q primes and a random suitable g. To verify if g

is suitable, A must only verify if $gcd(L(g^{\lambda} \mod n^2), n) = 1$, where $gcd(a, b)$ is the greatest common divisor of a and b. The private key is the Carmichael function λ of n and the public keys are the numbers n and g. A sends B the public parameters (n, g). Let m be the plaintext Bob wants to send to Alice, $m < n$. B then chooses a random $r < n$ and calculates $c = g^m \cdot r^n \mod n^2$. B proceeds by sending Alice the ciphertext c. Now, to reveal m, A checks if $c < n^2$ and, in that case, computes

$$m = L(c^{\lambda} \mod n^2) \cdot \left(L(g^{\lambda} \mod n^2)\right)^{-1} \mod n.$$

Also, PHPE has an **additive homomorphism**. Notice that for $c_1 = g_1^m \cdot r_1^n \mod n^2$ and $c_2 = g_2^m \cdot r_2^n \mod n^2$, the ciphertext c_3 is such that

$$\begin{aligned} c_3 &= c_1 \cdot c_2 \mod n^2 \\ &= ((g^{m_1} \cdot r_1^n) \cdot (g^{m_2} \cdot r_2^n)) \mod n^2 \\ &= (g^{m_1+m_2} \cdot (r_1 \cdot r_2)^n) \mod n^2 \\ &= (g^{m_1+m_2} \cdot r_3^n) \mod n^2 \end{aligned}$$

carries the encryption of $m_3 = m_1 + m_2$. Hence, the *multiplication* of ciphertexts results in the *addition* of plaintexts.

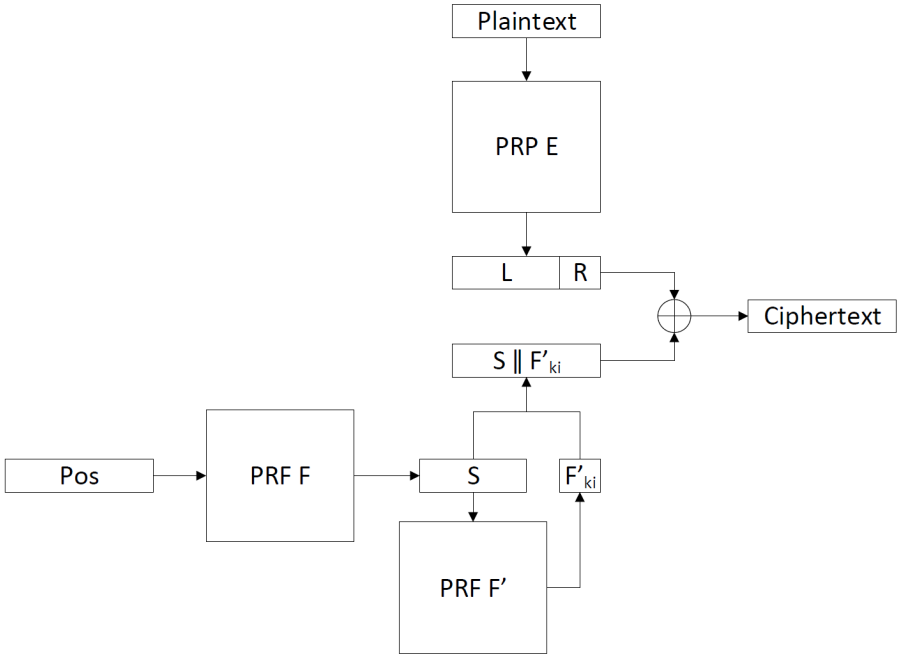
2.3.4 Song, Wagner and Perrig Searchable Encryption (SWP)

The **Song, Wagner and Perrig Searchable Encryption** (SWP) is a encryption scheme designed in 2000 to allow searches to be performed on encrypted data (SONG; WAGNER; PERRIG, 2000). SWP works by encrypting the plaintext with a deterministic PRP and XORing it with a block of the same length, derived from the plaintext data.

To encrypt a plaintext P, the following steps are performed (see Figure 8):

1. The plaintext P is encrypted using a deterministic PRP E and key k, $E_k(P)$;

Figure 8: SWP encryption.



Source: Adapted from (SONG; WAGNER; PERRIG, 2000).

2. $E_k(P)$ is divided into a left block L of size m and a right block R of size n;

3. Some information derived from P, such as the number of its position in the text, Pos, is encrypted using a PRF F and key k', resulting in $S = F_{k'}(Pos)$. S has a size of m;

4. A new key k_i is computed from the left block L, a PRF f and key k'', $k_i = f_{k''}(L)$;

5. This new key is used to create an encryption of S of size n using a PRF F', $F'_{k_i}(S)$;

6. S and $F'_{k_i}(S)$ are concatenated to make the mask which is applied to $E_k(P)$;

7. The mask is XORed to $E_k(P)$, producing the final ciphertext C.

To perform a word search, the plaintext encryption $E_k(P)$ and the key k_i are given to the search engine. The search engine picks every ciphertext C it has stored and XORs them with $E_k(P)$. It divides the result in m and n size blocks. The first block is a candidate for S and the second a candidate for $F'_{k_i}(S)$. Let the first block be called

S'. Next, the engine encrypts S' with the provided key k_i, $F'_{k_i}(S')$. If the encryption matches the second block, the ciphertext C is the encryption of P, otherwise the engine tries the next C.

2.4 Approximate Common Divisor

The **Approximate Common Divisor** (ACD) problem was introduced by Howgrave-Graham in 2001 (HOWGRAVE-GRAHAM, 2001).

Definition 1: The Approximate Common Divisor (ACD) Problem
Let p be a prime number of size η, in bits. Let q be an integer in the interval $[0, 2^\gamma/p)$, where the γ parameter gives the number final size. And let r be a positive or negative random noise whose size in bits is defined by the ρ parameter. Define the efficiently sampleable distribution $\mathcal{D}_{\gamma,\rho}(p)$ as

$$\mathcal{D}_{\gamma,\rho}(p) = \{p \cdot q + r \mid q \leftarrow \mathbb{Z} \cap [0, 2^\gamma/p), r \leftarrow \mathbb{Z} \cap (-2^\rho, 2^\rho)\}.$$

The ACD problem consists in computing p from polynomially-many samples x_i drawn from $\mathcal{D}_{\gamma,\rho}(p)$.

Later, Cheon and Stehlè proved that the decisional problem of ACD (i.e., to distinguish between a sample from $\mathcal{D}_{\gamma,\rho}(p)$ and an integer uniformly chosen in an interval) is not easier than the Learning with Errors (LWE) problem (CHEON; STEHLÈ, 2016). Therefore, since LWE (REGEV, 2009) is the basis of many lattice-based, post-quantum secure cryptographic primitives (e.g. CRYSTALS (AVANZI et al., 2018; DUCAS et al., 2018), FRODO (ALKIM et al., 2017b), NewHope (ALKIM et al., 2017a) and qTesla (BINDEL et al., 2018)), the decisional problem of ACD is also expected to be hard even in a setting where quantum computers are available.

Methods to solve the ACD were also studied by Galbraith et al.(GALBRAITH; GEBREGIYORGIS; MURPHY, 2016). Their work corroborates the results of Cheon

and Stehlè. They also state that all the methods they studied are essentially equivalent. All of the algorithms that solve the ACD problem start by using the many samples x_i from $\mathcal{D}_{\gamma,\rho}(p)$ to construct a lattice matrix. Once it is constructed, a lattice basis reduction should be performed to solve the problem.

Using the results provided, a choice for the parameters $\rho, \eta, \gamma \in \mathbb{N}$ to provide a desired security level against classical computers of λ in bits is:

$$\begin{aligned} \rho &\geq \lambda \\ \eta &> \rho \\ \gamma &\geq \Omega\left(\frac{\rho}{\log(\rho)} \cdot (\eta-\rho)^2\right) \\ \frac{\gamma-\rho}{\eta-\rho} &\geq 800 \end{aligned}$$

The first three equations were presented by Cheon and Stehlè (CHEON; STEHLÈ, 2016). The noise parameter ρ is greater than λ in order to prevent brute force attacks on the noise. The parameter γ is chosen in order to prevent lattice attacks. The parameter η is selected to allow correct decryption of the ciphertext.

The final equation, the relation between ρ, η and γ, was set heuristically by Galbraith et al.(GALBRAITH; GEBREGIYORGIS; MURPHY, 2016) to prevent any practical lattice attack.

2.5 Summary

In this chapter, we described the basic concepts that are used throughout this work: pseudo-random function and permutation, block cipher and its modes of operation, indistinguishability of encryptions, cryptographic hash function, and the approximate common divisor problem. Moreover, some important cryptographic algorithms that are later used were detailed. In particular, we use the approximate common divisor problem in chapter 4 as a basis to a new partially additive homomorphic encryption

algorithm. In addition, we adopt the Paillier Homomorphic Probabilistic Encryption as a performance reference for our new algorithm in chapter 6. We also use the Advanced Encryption Standard and its modes of operation together with hash functions and the Song, Wagner and Perrig Searchable Encryption in chapter 5 to improve the CryptDB database system.

3 CRYPTDB

CryptDB (POPA et al., 2011) is a property-preserving encrypted database designed by MIT in 2011. It works by encrypting the same data multiple times using different algorithms. Each of these algorithms enables a specific database functionality. Algorithms may encrypt the data sequentially generating layers. Multiple layers form a structure called onion. Each onion is responsible for a DB macro operation, such as equality, ordering, searching, and addition.

CryptDB utilizes a client proxy to make the encrypted database transparent to the user. Thus, the end user operates on the database as if it was not encrypted, which is useful from a usability standpoint.

The next section gives an overview of the system, followed by an in-depth analysis of its components, limitations and flaws.

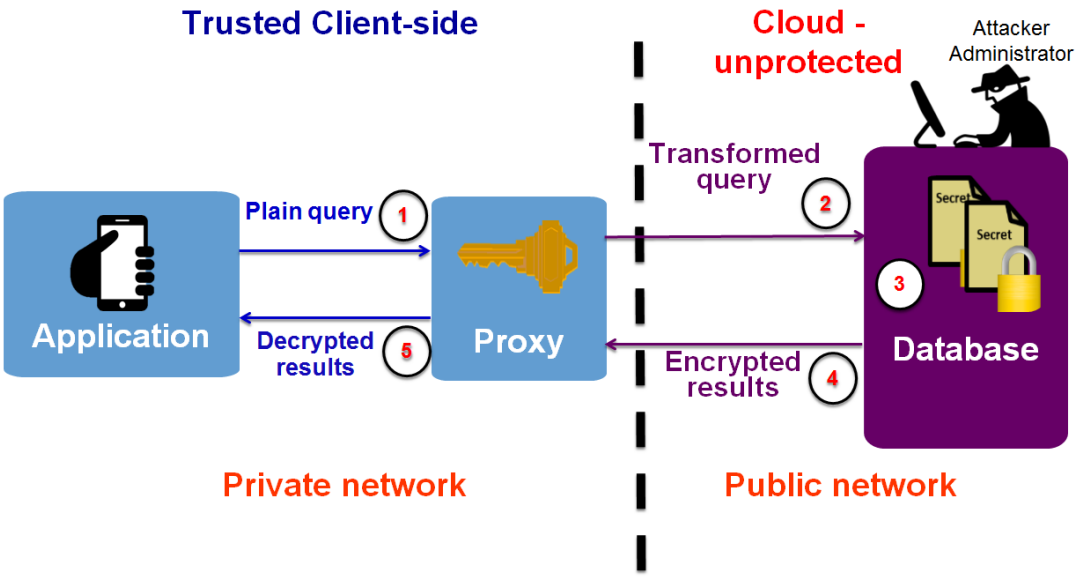
3.1 Overview

CryptDB's architecture can be divided into client-side and server-side components. On the client-side, there is the end user and the proxy, while on the server side there is the database itself. This architecture is illustrated in figure 9.

3.1.1 Client-side

The end user and proxy are on the client-side.

Figure 9: CryptDB architecture



Source: Adapted from (POPA et al., 2011).

The **end user** is the data owner or someone authorized to access and manipulate the data. The encrypted cloud DB is completely transparent to him. He writes plain SQL queries, that are forwarded to the proxy for adequate processing.

The **proxy** is a secure application. It is responsible for key management and storage, and query rewriting.

Every DB table, column and encryption layer have their own encryption key. When a client wants to insert data in the DB, the proxy analyses which operations are required for that field. It proceeds by determining what onion and layers should be created for that value and by fetching the appropriate encryption keys. The keys are created if they do not exist. Afterwards, the value is encrypted for every needed layer, for every needed onion. Now that the proxy has the encryption of the required onions, it rewrites the SQL query. First, it encrypts the table and column name to obfuscate them. Next, it changes the data to the encrypted onions that have been created. Finally, it sends this rewritten query to the DB.

The proxy works in a similar way when the client needs to operate on the database.

Instead of generating multiple onions, the proxy determines which onion and which layer is required to compute the operation. It encrypts the operand so it matches the designed layer. Finally, it rewrites the query to operate on the specific layer with the encrypted operand and sends it to the DB.

3.1.2 Server-side

The database and its management system are on the server side.

The **database** itself acts as an ordinary database, simply storing the values provided by the proxy.

The **database management system** (DBMS) has some slight modifications compared to an unencrypted database version. These modifications are simply a set of user defined functions, UDFs. They allow the system to interpret and process the encrypted SQL queries provided by the proxy. Ergo, the DBMS is responsible for the correct functionality of the database.

3.2 CryptDB's Structure

As previously mentioned, CryptDB stores the same data encrypted multiple times in onions. Each onion has multiple layers which are encrypted using different algorithms. Figure 10 illustrates the Onion-layer architecture.

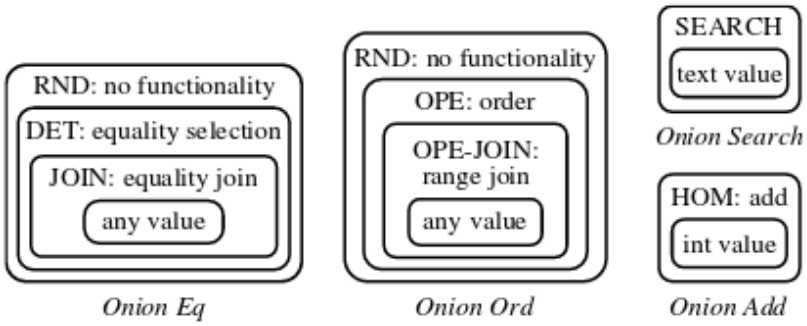
Next, we briefly describe each onion and, subsequently, each individual layer.

3.2.1 Onions

A description for each onion ensues.

The **equality onion** is responsible for DB operations that require the comparison of provided and stored data. Simple matching selection and table join are performed

Figure 10: CryptDB multiple onions with layers



Source: (POPA et al., 2011).

by this onion.

The **order onion** is responsible for DB operations that require order relations between provided and stored data. ORDER BY, smaller/greater than and table range joins are performed by this onion.

The **search onion** is responsible for the DB operations that require encrypted text search between a provided data and stored data. The LIKE operator is implemented by this onion.

The **addition onion** is responsible for the DB operations that require sums between provided data and stored data or multiple stored data. SUM, averages and value increment are performed by this onion.

3.2.2 Encryption Layers

Each onion is composed of one or multiple layers that store the data appropriately for a given operation. Figure 10 also shows the layers that constitute each onion.

The **Random Layer** (RND) is designed to provide IND-CPA security. It has no database operational function otherwise. This layer uses a block cipher in CBC mode with a random initialization vector (IV). As a result, the Random Layer implements a probabilistic encryption.

The **Deterministic Layer** (DET) provides the equality check functionality to the Equality Onion. It enables the computation of selects with equality operators, GROUP BY, COUNT, DISTINCT, among others. This layer uses a PRP in CMC mode with a zero IV. The outcome is a deterministic encryption. This allows the equality check between values without exposing the unencrypted value.

The **Join Layer** (JOIN) is responsible for allowing the operation JOIN between two database columns. This layer uses an algorithm called Adjustable Join Encryption (AJE). This algorithm was proposed by the creators of CryptDB (POPA; ZELDOVICH, 2012). AJE allows the columns' ciphertexts to be modified to share the same key under a deterministic encryption scheme. Hence, equality comparisons can be made between different columns.

The **Order Preserving Layer** (OPE) enables order relations between data in the same column. Thus, ordering is enabled by this layer for the Order Onion. This layer was first implemented using the Boldyreva, Chenette, Lee and O'Neil Order Preserving Encryption (BCLO) algorithm (BOLDYREVA et al., 2009). Later, BCLO was found insecure and it was substituted for the mOPE/stOPE algorithm proposed by the CryptDB creators (POPA; LI; ZELDOVICH, 2013).

The **Order Join Layer** (OPE-JOIN) permits order joins to be executed by the database. It uses the same algorithm as the Join Layer to modify ciphertexts on different columns to share the same encryption key. However, the underlying encryption scheme is provided by the OPE layer. Consequently, data from two columns can now be order related.

The **Search Layer** (SEARCH) allows a word to be searched on the database without revealing it. This layer enables the SQL operand LIKE in the database. It uses the SWP encryption scheme, presented in chapter 2.

The **Homomorphic Addition Layer** (HOM) permits the sum of values without

first decrypting them. This allows the database to perform the SQL SUM operation and averages. The layer is implemented by the PHPE algorithm presented in chapter 2.

3.3 CryptDB's Flaws, Limitations and Efficiency

CryptDB has flaws, limitations and efficiency problems. In this work, we propose to correct some of them, which are presented next.

CryptDB's HOM uses the PHPE. PHPE is a slow encryption algorithm as it is similar to RSA. Since it is a performance bottleneck, we studied possible solutions to increase HOM's efficiency. As a result, we designed a new partially homomorphic encryption that is presented in chapter 4. CryptDB's modification and other considerations regarding the HOM are presented in chapter 5, section 5.1.

CryptDB's SEARCH has the limitation of only allowing full word searches to be performed on encrypted data. The user cannot look up for word fragments or use the traditional database wildcard operator. This limits the ability of CryptDB to produce useful information, as a user may not have the entire word at his disposal or wants to look for multiple cases (e.g., prefixes or suffixes). We propose a work around to this situation on chapter 5, section 5.2.

Finally, as the encrypted data is deterministic in DET, this can be used to perform a series of attacks described at (NAVEED; KAMARA; WRIGHT, 2015). Inference attacks are performed by counting the frequency of a given ciphertext on the database and matching this ciphertext with a plaintext with a similar frequency on an auxiliary database. The l_p-Optimization attack is an evolution of the inference attack; instead of analyzing each ciphertext individually, it creates a table of all ciphertexts and their frequency and a table of all plaintexts and their frequency of an auxiliary database. After that, it tries to combine these tables until an optimization cost function achieves

its minimum value. The resulting combination is likely the plaintext-ciphertext key table for this database. Both these attacks relies heavily on the fact that the same plaintext is encrypted to the same ciphertext in the database. We propose a modification to CryptDB to mitigate these kind of attacks on chapter 5, section 5.3.

For a final consideration, the OPE is susceptible to some attacks, presented in (NAVEED; KAMARA; WRIGHT, 2015) and (KOLESNIKOV; SHIKFA, 2012). Although we initially planned to mitigate these attacks, an article by Durak, DuBuisson and Cash (DURAK; DUBUISSON; CASH, 2016) showed that attacks to OPE are inherent to its concept. In other words, if encrypted words are simply ordered, their unencrypted values can be inferred using an auxiliary database. Because of this, we decided to not treat attacks to OPE.

3.4 Summary

In this chapter we presented the cloud encrypted database CryptDB. Firstly, we described CryptDB's overall framework, with the client-server side separation and components. Next, we detailed CryptDB's internal structure, explaining the Onions construction and their internal layers. Furthermore, we described which operation each layer is responsible for. Finally, we briefly analyzed CryptDB's flaws, limitations and efficiency, outlining our proposals to improve each aspect.

4 NEW FAST ADDITIVE PARTIALLY HOMOMORPHIC ENCRYPTION

In this chapter, we present two novel additive, partially homomorphic encryption schemes built upon the Approximate Common Divisor (ACD) Problem presented in section 2.4. The constructions are inspired by similar works that use the ACD problem (e.g. DGHV (DIJK et al., 2010), Batch DGHV (CHEON et al., 2013), AHE (CHEON; STEHLÈ, 2016)). We name them Fast Additive Homomorphic Encryption (FAHE) 1 and FAHE2. One of the main particularities of the proposed solutions, which enable relevant simplifications and optimizations, is that they rely on symmetric keys for data encryption and decryption. Hence, on one hand, just a trusted party can encrypt and decrypt data. Homomorphic additions, on the other hand, can be performed very efficiently by any entity (e.g., cloud servers).

In a nutshell, FAHE1 and FAHE2 are symmetric probabilistic encryption algorithms that use a prime number as private key. Both schemes rely on the ACD as underlying security problem. However, whereas FAHE1 is a simple application of the ACD, FAHE2 provides shorter ciphertexts but requires slightly stronger security assumptions.

For both protocols, we assume the following: the security parameter is λ; the maximum plaintext message size is $|m_{max}|$; the noise size is ρ; the secret key size is η; the final ciphertext's maximum size is γ; and the total number of additions supported is at least $2^{\alpha-1}$. This notation is summarized in Table 1.

Table 1: General notation for FAHE.

Symbol	Definition
λ	Security parameter
$\|m_{max}\|$	Maximum plaintext message size
ρ	Noise size
η	Secret prime number size
γ	Ciphertext size
α	Number of supported additions (log)

Source: Author.

4.1 Notation

The following notation is used throughout this chapter.

The notation $a \leftarrow A$ refers to sampling uniformly at random an element from the set A. We write $|a|$ to denote the size of a bit string a, and 0^l to refer to a l-bit long string composed of 0s. The concatenation of bit strings a and b is denoted $a \parallel b$. The operation $x \ll y$ (resp. $x \gg y$) corresponds to the left (resp. right) shift of x by y positions, where $x, y \in \mathbb{N}$. Finally, $\log x$ is the base-2 logarithm of x.

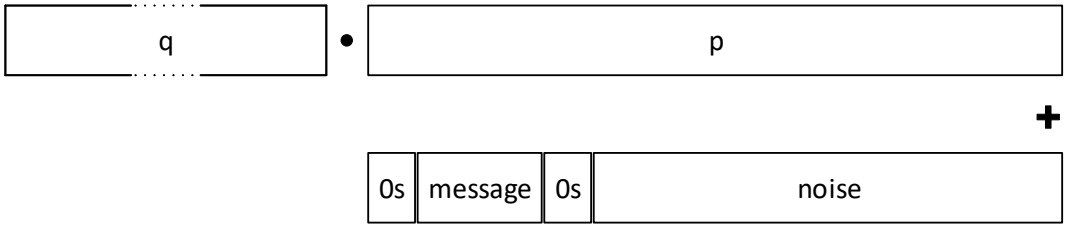
4.2 FAHE1

The main idea for FAHE1 is to use the ACD problem and append the message m to be encrypted at the end of the noise `noise`, before adding the result to $p \cdot q$. Since the resulting string containing m and `noise` remains smaller than the η-bit prime p, the corresponding plaintext can be recovered via modular reduction, during the decryption procedure. Figure 11 illustrates FAHE1's ciphertext structure.

4.2.1 Key generation, encryption, decryption and homomorphic addition

The key generation, encryption, decryption and homomorphic addition processes of FAHE1 are defined as follows:

Figure 11: A visual description of FAHE1's encryption process. Parameter q is larger than p and is partially represented.



Source: Author.

FAHE1.Keygen($\lambda, |m_{max}|, \alpha$). Choose a suitable security parameter λ, the maximum message size $|m_{max}|$ and the parameter α that defines the total number of supported additions. Then, compute the set of parameters (ρ, η, γ), given by: $\rho = \lambda,\ \eta = \rho + 2\,\alpha + |m_{max}|,\ \gamma = (\frac{\rho}{\log \rho} \cdot (\eta - \rho)^2)$. Finally, pick a prime p of size η and set $X = 2^{\gamma}/p$.

Set the scheme's key to $k = (p, |m_{max}|, X, \rho, \alpha)$. In the encryption process, the subset $ek = (p, X, \rho, \alpha)$ is required. For decryption, the user employs the subset $dk = (p, |m_{max}|, \rho, \alpha)$.

FAHE1.Enc$_{ek}$(m). Given a message m, sample $q \leftarrow [0, X)$, $\texttt{noise} \leftarrow \{0,1\}^{\rho}$ and let $M = (m \ll (\rho + \alpha)) + \texttt{noise}$. Then, compute $n = p \cdot q$ and output

$$c = n + M.$$

FAHE1.Add(c_1, c_2). Given two ciphertexts c_1 and c_2, output $c_{add} = c_1 + c_2$. Note that the ciphertext size can increase during this operation due to carries.

FAHE1.Dec$_{dk}$(c). Given the ciphertext c, output the least significant $|m_{max}|$ bits of

$$m = (c \bmod p) \gg (\rho + \alpha)$$

4.2.2 Correctness

The correctness of the encryption and decryption processes is quite simple to verify. First, the encryption of plaintext m consists in placing, via addition, its encoded form $M = m \ll (\rho + \alpha) + \texttt{noise}$ into a fixed position of $n = p \cdot q$. Since $M < p$ by construction, the resulting ciphertext $c = n + M$ can be decrypted as:

$$\begin{aligned} m &= (c \bmod p) \gg (\rho + \alpha) \\ &= ((p \cdot q + M) \bmod p) \gg (\rho + \alpha) \\ &= M \gg (\rho + \alpha) \\ &= ((m \ll (\rho + \alpha)) + \texttt{noise}) \gg (\rho + \alpha) \\ &= ((m \ll (\rho + \alpha)) \gg (\rho + \alpha)) + (\texttt{noise} \gg (\rho + \alpha)) \\ &= m \end{aligned}$$

The correctness of the homomorphic operations, in turn, is highly dependent on the α-bit sequences of 0's placed around the plaintext m while it is encoded into M. The reason is that such sequences handle the carry propagation resulting from additions, acting as buffers for those carry bits. Namely, the buffer placed on the right side of message m (i.e., between its least significant bit and the `noise`) is reserved for the noise growth resulting from each addition, preventing such noise from mixing with the message itself. Similarly, the buffer placed on the left side of m is reserved for the carry-outs that may result when the plaintext messages are themselves added together. For a concrete example, suppose that ciphertexts $\{c_1, \ldots, c_a\}$ are (homomorphically) added together; the resulting ciphertext c_+ is then computed as:

$$\begin{aligned} c_+ &= \textstyle\sum_{i=1}^{a}(c_i) \\ &= \textstyle\sum_{i=1}^{a}(n_i + M_i) \\ &= \textstyle\sum_{i=1}^{a}(n_i) + \sum_{i=1}^{a}(m_i) \ll (\rho + \alpha) + \sum_{i=1}^{a}(\texttt{noise}_i) \end{aligned}$$

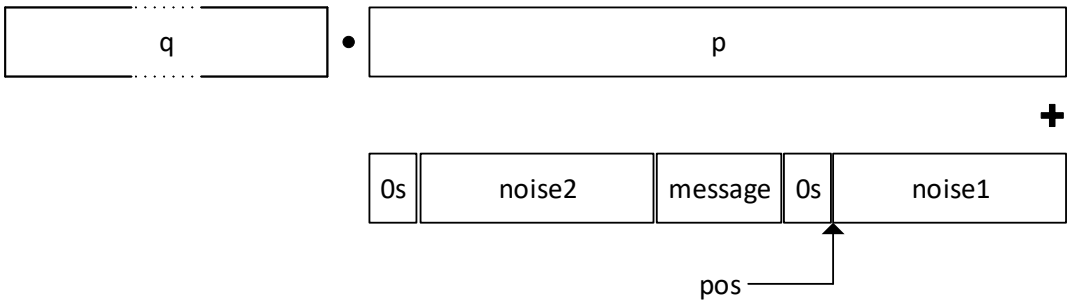
Even though the $\sum_{i=1}^{a}(\texttt{noise}_i)$ expression can generate carries, those carries would only affect the buffer on the right side of the message, not the bits on the message itself. For an α-bit long buffer, up to 2^α bits can be captured in this manner, meaning that the addition of noises would only start to affect the bits on the expression $\sum_{i=1}^{a}(m_i) \ll (\rho+\alpha)$ after at least 2^α additions are performed. Analogously, even if $\sum_{i=1}^{a}(m_i) \ll (\rho+\alpha)$ results in carry-outs, the corresponding bits are captured by the α-bit long buffer on the left side of each m_i. As long as less than $2^{\alpha-1}$ additions are performed, it is ensured that the most significant bit of this buffer remains at 0. As a result, we have $\sum_{i=1}^{a}(M_i) < p$, and the decryption of c_+ is correctly performed as:

$$
\begin{aligned}
m_+ &= (c_+ \mod p) \gg (\rho+\alpha) \\
&= (\textstyle\sum_{i=1}^{a}(n_i + M_i) \mod p) \gg (\rho+\alpha) \\
&= (\textstyle\sum_{i=1}^{a}(M_i)) \gg (\rho+\alpha) \\
&= (\textstyle\sum_{i=1}^{a}(m_i \ll (\rho+\alpha) + \texttt{noise}_i)) \gg (\rho+\alpha) \\
&= \textstyle\sum_{i=1}^{a}(m_i)
\end{aligned}
$$

Therefore, by choosing a suitable α parameter, it is ensured that $2^{\alpha-1}$ homomorphic additions can be performed in FAHE1. Those additions can include both unsigned numbers and signed numbers represented in two's complement, since the carry-out bit of the resulting operation is ignored anyway (as dictated by the two's complement arithmetic).

As a final remark, we note that, for some values of α, the ciphertext size $\gamma = (\frac{\rho}{\log\rho} \cdot (\eta-\rho)^2)$ could be theoretically smaller than the prime size η. However, for the security reasons presented in section 4.4 and to prevent invalid values for γ, we suggest setting the minimum value of α in FAHE1 as described in section 4.5.

Figure 12: A visual description of FAHE2's encryption process. Parameter q is larger than p and is partially represented.



Source: Author.

4.3 FAHE2

The basic idea behind FAHE2's design is to create an open space at a given position `pos` inside the noise employed in the ACD problem. Then, we embed the message in that position before adding the result to $p \cdot q$. As a result, the difference between the noise size ρ and the key size η is smaller than in FAHE1. Consequently, the ciphertext size is also smaller than in the previous variant. Figure 12 illustrates FAHE2's structure.

4.3.1 Key Generation, Encryption and Decryption

The key generation, encryption, decryption and homomorphic addition processes of FAHE2 are defined as follows:

FAHE2.Keygen(λ, $|m_{max}|$, α). Choose a suitable security parameter λ, the maximum message size $|m_{max}|$ and the parameter α that defines the total number of supported additions. Then, compute the set of parameters $(\rho, \eta, \gamma, p, X, \texttt{pos})$, given by: $\rho = \lambda + \alpha + |m_{max}|$, $\eta = \rho + \alpha$, $\gamma = (\frac{\rho}{\log \rho} \cdot (\eta - \rho)^2)$. Finally, pick a prime p of size η and set $X = 2^\gamma / p$ and $\texttt{pos} \leftarrow [0, \lambda]$.

Set the scheme's key to $k = (p, X, \texttt{pos}, |m_{max}|, \lambda, \alpha)$. In the encryption process,

the subset $ek = (p, X, \texttt{pos}, |m_{max}|, \lambda, \alpha)$ is required. For decryption, the user employs the subset $dk = (p, \texttt{pos}, |m_{max}|, \alpha)$.

FAHE2.Enc$_{ek}(m)$. Given a message m, sample $q \leftarrow [0, X)$, $\texttt{noise1} \leftarrow \{0,1\}^{\texttt{pos}}$, $\texttt{noise2} \leftarrow \{0,1\}^{\lambda-\texttt{pos}}$, and make $M = (\texttt{noise2} \ll (\texttt{pos} + |m_{max}| + \alpha)) + (m \ll (\texttt{pos} + \alpha)) + \texttt{noise1}$. Next, compute $n = p \cdot q$ and output

$$c = n + M.$$

FAHE2.Add(c_1, c_2). Given two ciphertexts c_1 and c_2, output $c_{add} = c_1 + c_2$. Note that the ciphertext size can increase during this operation due to carries.

FAHE2.Dec$_{dk}(c)$. Given the ciphertext c, output the least significant $|m_{max}|$ bits of

$$m = (c \bmod p) \gg (\texttt{pos} + \alpha)$$

4.3.2 Correctness

The correctness of the encryption and decryption processes can be verified similarly to FAHE1. Indeed, encryption consists in placing the plaintext's encoded form $M = (\texttt{noise2} \ll (\texttt{pos} + |m_{max}| + \alpha)) + (m \ll (\texttt{pos} + \alpha)) + \texttt{noise1}$ into $n = p \cdot q$, at the position defined by the pos variable. Since we once again have $M < p$ (by

construction), the ciphertext $c = n + M$ can be decrypted as:

$$
\begin{aligned}
m &= (c \bmod p) \gg (\texttt{pos} + \alpha) \\
&= ((p \cdot q + M) \bmod p) \gg (\texttt{pos} + \alpha) \\
&= M \gg (\texttt{pos} + \alpha) \\
&= ((\texttt{noise2} \ll (\texttt{pos} + |m_{max}| + \alpha)) + (m \ll (\texttt{pos} + \alpha)) + \texttt{noise1}) \gg (\texttt{pos} + \alpha) \\
&= ((\texttt{noise2} \ll (\texttt{pos} + |m_{max}| + \alpha)) + (m \ll (\texttt{pos} + \alpha))) \gg (\texttt{pos} + \alpha) \\
&\qquad + (\texttt{noise1}) \gg (\texttt{pos} + \alpha) \\
&= (\texttt{noise2} \ll |m_{max}|) + m \\
&= m \qquad \triangleright \text{ note: only the } |m_{max}| \text{ least significant bits are output}
\end{aligned}
$$

The correctness of the homomorphic operations, in turn, is also dependent on the α-bit sequences of 0's placed at the end of both `noise1` and `noise2`, which work as buffers for carry bits. Namely, the buffer placed after `noise1` handles its growth due to additions, thus preventing this part of the noise from reaching the message itself for less than 2^{α} additions. Then, the buffer placed after `noise2` prevents M from growing larger than p as long as less than $2^{\alpha-1}$ additions are performed. This can be verified if we once again consider the (homomorphic) addition of ciphertexts $\{c_1, \cdot, c_a\}$, leading to ciphertext c_+:

$$c_+ = \sum_{i=1}^{a}(c_i)$$

$$= \sum_{i=1}^{a}(n_i + M_i)$$

$$= \sum_{i=1}^{a}(n_i) + \sum_{i=1}^{a}((\texttt{noise2}_i \ll (\texttt{pos} + |m_{max}| + \alpha)) + (m_i \ll (\texttt{pos} + \alpha)) + \texttt{noise1}_i)$$

$$= \sum_{i=1}^{a}(n_i) + \sum_{i=1}^{a}(\texttt{noise2}_i \ll (\texttt{pos} + |m_{max}| + \alpha))$$

$$+ \sum_{i=1}^{a}(m_i \ll (\texttt{pos} + \alpha))$$

$$+ \sum_{i=1}^{a}(\texttt{noise1}_i)$$

Similarly to FAHE1, we have $\sum_{i=1}^{a}(M_i) < p$ for less than $2^{\alpha-1}$ addition, so the decryption of c_+ is correctly performed as:

$$m_+ = (c_+ \bmod p) \gg (\texttt{pos} + \alpha)$$

$$= (\sum_{i=1}^{a}((n_i + M_i) \bmod p)) \gg (\texttt{pos} + \alpha)$$

$$= (\sum_{i=1}^{a}(M_i)) \gg (\texttt{pos} + \alpha)$$

$$= (\sum_{i=1}^{a}((\texttt{noise2}_i \ll (\texttt{pos} + |m_{max}| + \alpha)) + (m_i \ll (\texttt{pos} + \alpha)) + \texttt{noise1}_i)) \gg (\texttt{pos} + \alpha)$$

$$= (\sum_{i=1}^{a}(\texttt{noise2}_i \ll (\texttt{pos} + |m_{max}| + \alpha)) + (\sum_{i=1}^{a}(m_i \ll (\texttt{pos} + \alpha)))) \gg (\texttt{pos} + \alpha)$$

$$+ (\sum_{i=1}^{a}(\texttt{noise1}_i)) \gg (\texttt{pos} + \alpha)$$

$$= \sum_{i=1}^{a}(\texttt{noise2}_i \ll |m_{max}|) + \sum_{i=1}^{a}(m_i)$$

$$= \sum_{i=1}^{a}(m_i)) \qquad \triangleright \text{ note: only the } |m_{max}| \text{ least significant bits are output}$$

Also like in FAHE1, FAHE2 support additions for both unsigned numbers and signed numbers represented in two's complement. After all, carry-outs in this case are

stored in `noise2`, which is discarded during the decryption of ciphertexts.

Finally, the same remark regarding how the ciphertext sizes in FAHE1 relate to α is also valid for FAHE2. More precisely, for some values of α the ciphertext size $\gamma = (\frac{\rho}{\log \rho} \cdot (\eta - \rho)^2)$ could be theoretically smaller than the prime size η. For the security reasons presented in section 4.4 and to prevent invalid values for γ, though, we suggest the minimum value of α in FAHE2 as described in section 4.5.

4.4 Security analysis

Both FAHE and FAHE2 rely on the post-quantum secure *Approximate Common Divisor* (ACD) as underlying security problem. Nevertheless, to avoid possible borrow-ins from the message text when performing homomorphic additions, we only use the positive interval for the noise added to $p \cdot q$. Therefore, and even though the ACD problem states that the noise interval comprehends negative and positive values, we argue that using only positive values does not affect the noise's distribution and, hence, the overall security of the ACD decisional problem. This argument is based on the result shown in Theorem 1.

Theorem 1. *Let $\mathcal{D}$ be a sampleable distribution built from the original ACD problem, and $\mathcal{D}'$ be a sampleable distribution built from the ACD problem where the noise consists only in positive integers. Samples taken from $\mathcal{D}$ are as indistinguisable from random as samples taken from $\mathcal{D}'$.*

Proof. (Sketch) Let x_i be elements sampled from the ACD distribution $\mathcal{D}_{\gamma,\rho}(p) = \{p \cdot q + r \mid q \leftarrow \mathbb{Z} \cap [0, 2^\gamma/p), r \leftarrow \mathbb{Z} \cap (-2^\rho, 2^\rho)\}$. Remember that every x_i sampled in this manner is indistinguishable from random (see section 2.4). Now, take $t = 2^\rho$ and compute $x'_i = x_i + t$ for every x_i, thus "shifting" those elements by the fixed threshold 2^ρ. The result of this sum $x'_i = x_i + t$, where the biggest addend x_i is indistinguishable from random and t is a constant, is still indistinguishable from random. Therefore, the

new element x'_i can then be rewritten as $x'_i = \{p \cdot q + r' \mid r' \leftarrow \mathbb{Z} \cap (0, 2^{\rho+1})\}$, which is an instance of the ACD problem where the noise consists only in positive integers. □

Building upon Theorem 1, we analyze the security of FAHE1 and FAHE2 in what follows.

4.4.1 Security of FAHE1

FAHE1 uses just the distribution with positive noise to encrypt messages and it is indistinguishable under a chosen plaintext attack (IND-CPA) (see subsubsection 2.2.3.1). In other words, given polynomial encryption samples $c_1, c_2, \ldots, c_a$ corresponding to plaintexts messages $m_1, m_2, \ldots, m_a$ of the same size and a challenge ciphertext c_x from m_{c0} or m_{c1}, an adversary cannot determine if $x = c0$ or $x = c1$ with non-negligible probability.

Theorem 2. *Encryption with FAHE1 is indistinguishable under a chosen plaintext attack (IND-CPA).*

Proof. (Sketch) A sample c_n is computed by adding together the message m_n and an element $n \leftarrow \mathcal{D}_{\gamma,\rho}(p) = \{p \cdot q + r \mid q \leftarrow \mathbb{Z} \cap [0, 2^\gamma/p), r \leftarrow \mathbb{Z} \cap (0, 2^{\rho+1})\}$ from the ACD distribution with positive noise. Since the element n is indistinguishable from random, the sampling process acts as a secure pseudorandom function (PRF). After all, for every message, a distinct element is chosen with high probability: namely, the probability of repeating an element from $\mathcal{D}_{\gamma,\rho}(p)$ is approximately $2^{-\frac{(\gamma-\eta)}{2}}$, which is the probability of drawing the same q considering the birthday paradox. Therefore, n can be interpreted as an independent pseudorandom one time pad when $(\gamma - \eta)/2$ is big; in particular, in FAHE1 we have $(\gamma - \eta)/2 > 1000$, so the probability of repeating an element from $\mathcal{D}_{\gamma,\rho}(p)$ is negligible. Then, when we perform the addition $n + m_n = c_n$, the element n acts as random mask for every bit of the message m_n, since $n > m_n$. The

output c_n is, thus, a pseudorandom ciphertext produced from a secure PRF, which is IND-CPA (ROSULEK, 2018, Chapter 8.3). □

4.4.2 Security of FAHE2

FAHE2 embeds the message inside the noise. The noise can then be seen as two distinct components, $\texttt{noise1} \leftarrow [0, 2^{\texttt{pos}})$, and $\texttt{noise2} \leftarrow [2^{\texttt{pos}+\alpha+|m_{max}|}, 2^{\eta-\alpha})$ (note: noise2 can also be 0). Another possible interpretation, which is more useful in our scenario, is that noise2 is a noise r' from the interval $[0, 2^{\lambda-\texttt{pos}})$ multiplied by $Q = 2^{\texttt{pos}+\alpha+|m_{max}|}$. Therefore, the resulting construction resembles the Chinese Remainder Theorem (CRT) message encryption of the batch DGHV scheme (CHEON et al., 2013), where Q and the key p are coprimes: after all, p is prime and Q a power of two. The message encryption in FAHE2 can then be rewritten as $(Qr' + qm + \texttt{noise1})$, where $q = 2^{\texttt{pos}+\alpha}$.

There is evidence in the literature to believe the CRT variant of the ACD to be hard, but it is still an open problem to provide an algorithm that uses the CRT structure to solve the problem (GALBRAITH; GEBREGIYORGIS; MURPHY, 2016). Since we make the total size of the random noise in FAHE2 $|\texttt{noise1}| + |\texttt{noise2}| = \lambda$, brute force attacks on the noise are still prevented. Nevertheless, since FAHE2 relies on an approximation of the CRT variant of the ACD, the resulting scheme ends up having stronger security assumptions than FAHE1.

Finally, we note that FAHE2 also uses the ACD distribution with positive noise for generating a random mask for the message to be encrypted. Hence, using the same arguments presented for FAHE1 in Theorem 2, we can show that FAHE2 is also indistinguishable under a chosen plaintext attack (IND-CPA).

4.4.3 Security against the Simultaneous Diophantine Approximation (SDA) attack

As a final remark, we consider the heuristic analysis of the Simultaneous Diophantine Approximation (SDA) attack performed in (GALBRAITH; GEBREGIYORGIS; MURPHY, 2016). Specifically, (GALBRAITH; GEBREGIYORGIS; MURPHY, 2016) concluded that the dimension d of the lattice created by the ACD samples must present the following condition for the SDA algorithm to succeed:

$$d + 1 \geq \frac{\gamma - \rho}{\eta - \rho}$$

The same analysis also concludes that a value of $(\gamma - \rho)/(\eta - \rho) \geq 800$ is sufficient to prevent any practical lattice attack. It corresponds to the last relation presented in section 2.4. To achieve this in our proposed schemes, the term α must have a minimum value for each algorithm, which are presented in section 4.5.

4.4.4 A Key Recovery Attack with (Adaptive) Decryption Queries

If a decryption oracle is available for attackers, a key recovery attack can be mounted against both FAHE1 and FAHE2. Basically, the attack works by finding a multiple of the key p after sending (adaptively) chosen ciphertext decryption queries to this oracle, as follows.

First, the attacker asks for the decryption of the arbitrary ciphertext $c = 100\ldots000$ of size $|c| = \gamma$, and saves the output message m. The ciphertext can be interpreted as $c = x \cdot p + M$, where x is an integer. Then, the attacker scans that ciphertext, starting from the bit η (i.e., the position corresponding to the most significant bit of the secret prime p) to the least significant bit, until a bit 0 is found. That bit 0 is flipped to 1, and a new decryption oracle query is made with the modified ciphertext. As a result, there are two possibilities for this new ciphertext: (1) it corresponds to $x \cdot p + M'$, where M' and M are identical except for the flipped bit; or (2) it can be written as

$(x + 1) \cdot p + W$, where $W \neq M$. The latter case happens whenever the bit-flipping leads to an encoded plaintext that is larger than p, so it is affected by the $\pmod p$ operation during decryption; to correct this, the affected bit is flipped back to 0, so the multiplicand of p in the ciphertext remains as x. Otherwise, the affected bit is kept flipped. Whichever the case, the attacker repeats the scanning process, looking for the next bit 0 to be flipped.

When all bits between η and the least significant bit are processed in this manner, the resulting ciphertext can be written as $c' = x \cdot p + M^*$, where $M^* = p - 1$. After all, the attack ensures that all bits of the encoded message M were flipped so it is just barely smaller than p. Finally, then, by adding 1 to the modified ciphertext, the attacker obtains $(x \cdot p + p - 1) + 1 = (x + 1) \cdot p$, which is an integer multiple of p. By repeating the attack with another arbitrary ciphertext with the η-least significant bits cleared (e.g. $c = 1100 \ldots 000$, where $|c| = \gamma$), the attacker can simply compute the greatest common divisor (GCD) between the two ciphertexts obtained in this manner to recover p or a small multiple of p. We present an example of the key recovery attack on Appendix A.

Albeit powerful, we note that this attack shows that FAHE1 and FAHE2 cannot be proved to be IND-CCA2 secure, which is actually the case for any malleable (e.g., homomorphic) scheme (BELLARE et al., 1998, Theorem 3.3).

4.5 Parameter Selection Example

In this section, we present some suggested parameters size ρ, η, γ for both FAHE1 and FAHE2. We consider a security level λ of 128 bits against classical computers, the message size $|m_{max}|$ of 32 and 64 bits, and different values for the minimum number of supported additions, $2^{\alpha-1}$.

To obtain the value $(\gamma - \rho)/(\eta - \rho) \geq 800$ for FAHE1, we need to enforce $\alpha \geq 6$ for $\lambda = 128$ and $|m_{max}| = 32$. For FAHE2, we set $\alpha \geq 32$ for $\lambda = 128$ and $|m_{max}| = 32$. The

Table 2: Parameters for FAHE1 and FAHE2 for $\lambda = 128$.

	FAHE1				FAHE2			
	$\|m_{max}\| = 32$		$\|m_{max}\| = 64$		$\|m_{max}\| = 32$		$\|m_{max}\| = 64$	
	$\alpha = 6$	$\alpha = 33$	$\alpha = 6$	$\alpha = 33$	$\alpha = 32$	$\alpha = 33$	$\alpha = 29$	$\alpha = 33$
ρ	128	128	128	128	192	193	221	225
η	172	226	204	258	224	226	250	258
γ	35402	175616	105619	309029	25921	27683	23866	31359

Source: Author.

minimum α can be set lower for $|m_{max}| = 64$. For FAHE1, we decided to keep the same value as it is already small. For FAHE2, $\alpha \geq 29$ considering $\lambda = 128$ and $|m_{max}| = 64$.

We present the results for both schemes using the minimum α allowed and also $\alpha = 33$ (i.e., so at least 2^{32} sums are supported by the schemes). The rationale for the parameters $|m_{max}| = 64$ and $\alpha = 33$ were that this enables additions among as many long integers as commonly found in very large databases (e.g., 167 million items in the Library of Congress (Library of Congress, 2018) and 12 billion total credit card operations in Brazil in 2016 (Diretoria de Regulação Prudencial, Riscos e Assuntos Econômicos, 2016, p. 103)).

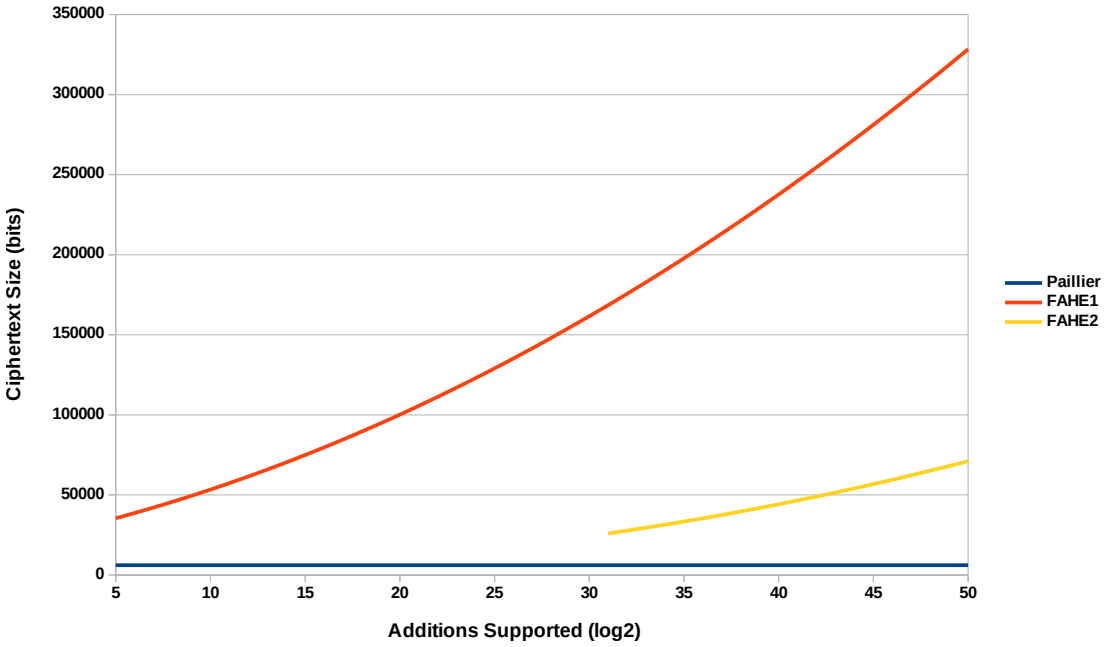
The parameters are presented in Table 2. We note that the ciphertext size varies almost linearly with the security parameter λ and almost quadratically with the number of supported sums α plus the message size $|m_{max}|$.

For FAHE1, the smallest ciphertext size is obtained when $\rho = 128$, $\eta = 172$, $\gamma = 35402$ for $\lambda = 128$, $|m_{max}| = 32$, $\alpha = 6$. The largest appears at $\rho = 128$, $\eta = 258$, $\gamma = 309029$ for $\lambda = 128$, $|m_{max}| = 64$, $\alpha = 33$.

For FAHE2, since the noise size is bigger for $|m_{max}| = 64$, α can be reduced. As a result, the smallest ciphertext size occurs at $\rho = 221$, $\eta = 150$, $\gamma = 23866$ for $\lambda = 128$, $|m_{max}| = 64$, $\alpha = 29$. The biggest γ is then obtained with $\rho = 225$, $\eta = 258$, $\gamma = 31359$ for $\lambda = 128$, $|m_{max}| = 64$, $\alpha = 33$.

Figure 13 presents the variation of ciphertext size with the number of supported

Figure 13: Variation of ciphertext size for FAHE1 and FAHE2 for different numbers of additions supported. Paillier is included as a reference.



Source: Author.

additions for $\lambda = 128$, $|m_{max}| = 32$ and valid values of α for FAHE1 and FAHE2. When we compare these numbers with those obtained with the Paillier scheme, we observe that, the latter's ciphertexts have the same length independently of the number of additions supported. Although FAHE1 and FAHE2 present ciphertexts larger than Paillier's, the operations from our algorithms are orders of magnitude more efficient, as further discussed in chapter 6.

We also evaluated how FAHE1 and FAHE2 would behave for $\lambda = 256$, which is expected to provide a security level of 128 bits against quantum computers (since it prevents brute force attacks on the noise) (GROVER, 1996). Again, to obtain $(\gamma - \rho)/(\eta - \rho) \geq 800$ for FAHE1, we set $\alpha \geq 6$. Actually, α can be set lower than 6 for this configuration, but as it is already small, we suggest keeping the same value calculated for $\lambda = 128$. In FAHE2, we suggest setting $\alpha \geq 22$ for $|m_{max}| = 32$ and $\alpha \geq 21$ for $|m_{max}| = 64$. The parameters are presented in Table 3.

Table 3: Parameters for FAHE1 and FAHE2 for $\lambda = 256$.

	FAHE1				FAHE2			
	$\lvert m_{max}\rvert = 32$		$\lvert m_{max}\rvert = 64$		$\lvert m_{max}\rvert = 32$		$\lvert m_{max}\rvert = 64$	
	$\alpha = 6$	$\alpha = 33$	$\alpha = 6$	$\alpha = 33$	$\alpha = 22$	$\alpha = 33$	$\alpha = 21$	$\alpha = 33$
ρ	256	256	256	256	310	321	341	353
η	300	354	332	386	332	354	362	386
γ	61952	307328	184832	540800	18130	41984	17874	45421

Source: Author.

4.6 Summary

In this chapter, we presented two partially additive homomorphic encryption schemes based on the approximate common divisor problem. Both schemes are symmetric and probabilistic but whilst the first has larger ciphertext size, the second relies on slightly stronger security assumptions. Another noteworthy point is that the schemes are believed to be resistant against quantum computers because they are built upon the approximate common divisor problem. We describe the key generation, encryption, decryption and homomorphic addition for them both. We also present their security analysis and parameter selection for a number of practical scenarios.

5 MODIFICATIONS

In this chapter, we present our proposed modifications to CryptDB.

Firstly, we present a modification in CryptDB's homomorphic addition layer (HOM) that improves its efficiency by changing the current cryptographic algorithm to our proposed scheme presented in chapter 4. We also discuss other attempted solutions and why they were not adopted.

Secondly, we present a modification in CryptDB's search layer (SEARCH) that improves its functionality by allowing wildcard searches. In addition, we present an important drawback to it.

Finally, we present a modification in CryptDB's deterministic layer (DET) that improves its security by mitigating the attacks presented in chapter 3. A drawback is also presented to our solution.

5.1 Efficiency Improvement in CryptDB's homomorphic addition layer (HOM)

We propose the substitution of the Paillier Homomorphic Probabilistic Encryption (PHPE) on CryptDB's homomorphic addition layer (HOM) by our algorithm Fast Additive Homomorphic Encryption (FAHE) 2, presented in chapter 4.

CryptDB's HOM uses PHPE to provide additive homomorphism in the encrypted database values. PHPE's n-th composite residuosity class problem (see subsec-

tion 2.3.3 for the definition) requires a public key n of size similar to RSA. For a desired security level $\lambda = 128$ bits, $n = 3072$ (BARKER; DANG, 2016, Table 2). Since PHPE uses the square of the key, n^2, to operate, the ciphertext final size is 6144 bits. As a result, the encryption and decryption require modular exponentiations and multiplications over a large cyclic group. Moreover, PHPE is not secure against quantum computer attacks, as the integer factorization of n allows the computation of the private key (SHOR, 1997).

In opposition, our algorithms allow considerable speed ups in every process (key generation, encryption, decryption, and homomorphic sum). Furthermore, our schemes are based on the Approximate Common Divisor (ACD) problem, believed to be resistant to quantum computer attacks.

Another difference between PHPE and FAHE is that the first is an asymmetric scheme while ours is symmetric. It means that PHPE allows multiple entities to encrypt data, but just a single one (the one that possesses the private key) is capable of decrypting it. Meanwhile, in FAHE, just the entity that can decrypt data can also encrypt it. Although this limits the scenarios where FAHE can substitute PHPE, CryptDB is one of the scenarios where the substitution can be made. In CryptDB, a single entity, the proxy, is responsible for encrypting the client data before sending it to the cloud database, as well as decrypting the cloud database data before returning it to the client. Hence, a symmetric key can be used and PHPE can be substituted by FAHE.

As a result of the substitution, the entire HOM benefits from a performance gain. Improvements to all the processes, including the key generation, aid the layer's efficiency. Albeit for most systems the key generation process is performed only once, every addition enabled column of a CryptDB's database is encrypted using a different key. Thus, the generation of multiple keys is necessary and its enhancement represents a benefit.

Although PHPE achieves Indistinguishability under Chosen-Ciphertext 1 (IND-

CCA1) (ARMKNECHT; KATZENBEISSER; PETER, 2013) while FAHE provides only Indistinguishability under Chosen-Plaintext (IND-CPA) (see section 4.4), this does not represent a problem. Firstly, only the proxy knows the cryptographic keys and only it is capable of encrypting or decrypting data. In CryptDB, it is assumed that the proxy is a secure entity. Hence, a decryption oracle is not provided to a possible attacker. Secondly, CryptDB's Random layer (RND) is responsible for providing data security to the database. RND is implemented using AES in Cipher Block Chaining (CBC) mode, which also only achieves IND-CPA (see subsection 2.2.2). Consequently, CryptDB is already limited by an IND-CPA algorithm and the change from PHPE to FAHE does not incur in a security loss.

The major drawback of the modification is the increased ciphertext size. FAHE2 presents a 5 times increase, approximately, in ciphertext size compared to PHPE. FAHE2's ciphertext size is 31359 bits (Table 2; $\lambda = 128$, $|m_{max}| = 64$, $\alpha = 33$) and PHPE's ciphertext size is 6144 bits ($\lambda = 128$, $n^2 = 6144$).

5.1.1 Other Attempted Solutions

In chapter 2, we presented the Eliptic Curve ElGamal Encryption Scheme (ECEGES). This algorithm also has an additive homomorphism property.

Our first attempt to improve the HOM was replacing PHPE by the ECEGES. Unfortunately, as presented on the algorithm description, a message should be mapped to an Elliptic Curve (EC) point before the encryption. Although the homomorphism is preserved for the points, the homomorphism is not preserved for the map and reverse map of the messages to points and the opposite. Using the Elligator2 map (BERNSTEIN et al., 2013), we observed that the addition of the points was not inverted back to the addition of the messages. Further studies showed that the range size of any map from a string to an EC point should have twice the domain size (FOUQUE; JOUX; TIBOUCHI, 2013). Therefore, a map from a string to a point is an injective function

only and there are elements in the range that do not have a corresponding string.

Another solution would be to map a message to an EC point by multiplying the message value by the curve generator point G. Despite the fact that this map would correctly preserve the additive homomorphism, the decryption process would have a major drawback. To invert the map, it is required to compute the EC discrete logarithm. The decryption of a ciphertext would result in the sum of messages multiplied by the curve generator. Although the message space could be made small (e.g., 2^{32}), this is still a hard problem. The time complexity of the EC discrete logarithm problem can be traded for space complexity by storing a precomputed table on the proxy. For this, all the message possibilities should be computed and the result should be stored in this table. Hence, when the map needs to be inverted, the proxy simply looks at the table to find the correspondence between EC point and message. The drawback is that this table reaches a couple of hundred gigabytes that must be stored in the proxy. The storage of large data on the local proxy goes against the idea of a cloud database Hence, this approach is not ideal.

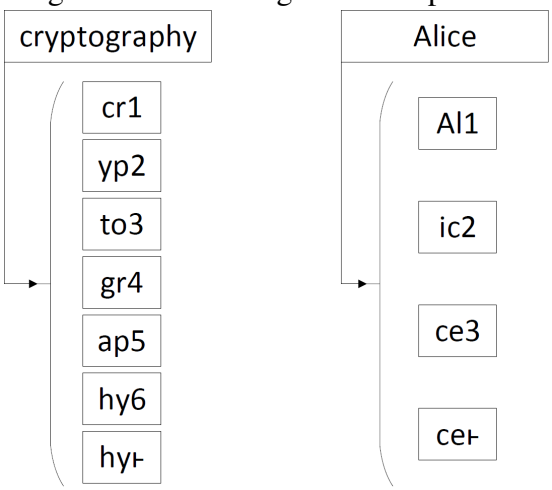
5.2 Functionality Improvement in SEARCH

CryptDB's search layer (SEARCH) uses the Song, Wagner and Perrig Searchable Encryption (SWP) algorithm presented in chapter 2 to enable search on encrypted data. The algorithm, as it is implemented, is able to search only for full words. In a database, a fragment word search is desirable, as the user may want to look for data patterns. To allow word fragments to be searched, we propose a modification on how data is encrypted by SWP.

Instead of encrypting the whole word, we propose that the user chose a token size. This token size is the fragment word size to be encrypted. The word will be divided in multiple tokens. Appended to the token, the position of the fragment in the word

is added, forming the final plaintext to be encrypted. The last token also has a second copy with the special terminator, ⊢. Moreover, if the last token is smaller than the token size, the final token repeats part of the previous fragment to reach the token size. For instance, the word “cryptography”, with a token size 2, will be divided in the fragments “cr1”, “yp2”, “to3”, “gr4”, “ap5”, “hy6”, and “hy⊢”. The word “Alice”, with the same token size, will be divided into “Al1”, “ic”, “ce3”, and “ce⊢”. These fragments will be then encrypted instead of the whole word. This process is depicted in Figure 14.

Figure 14: Word fragmentation process.

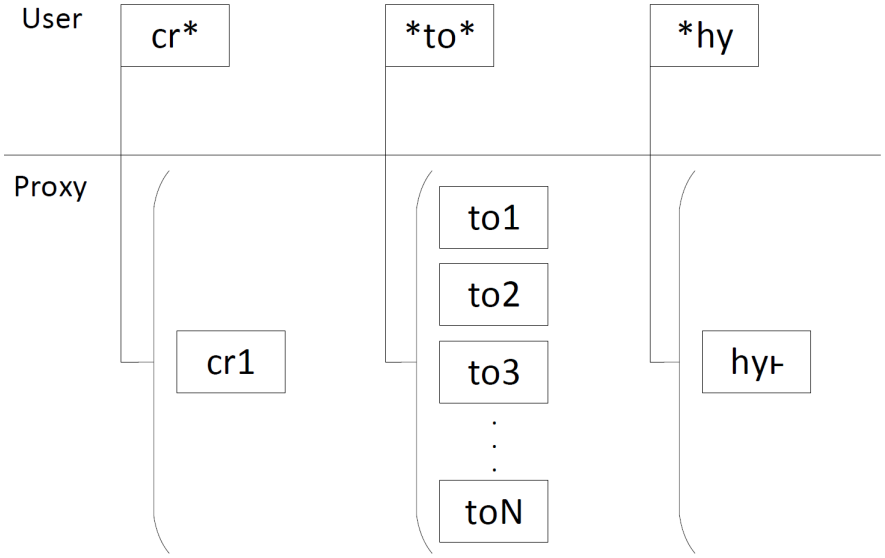


Source: Author.

When the user searches for a fragment, the proxy rewrites the search query and adds the appropriate positions to the fragment. One search query may have to be rewritten as many times as there are possible positions for the fragment. This modified query is then used to perform the search. An overview of the query rewrite step is presented in Figure 15.

Our solution presents no modification to the algorithm security since it is changing the data which will be encrypted, and not how the data is encrypted. Nonetheless, there are fewer possible fragments as the token gets smaller. When the token reaches the value of 1, there are only 52 possible fragments (the letters “a” to “z” and their capitals), followed by their position in the word. Assuming that the number of positions is finite and small, all possible combinations make up a small group. SWP uses an

Figure 15: Query rewrite process. “cr*” is a search for words that begin with “cr”; “*to*” is a search for words that contain “to”; “*hy” is a search for words that end with “hy”.



Source: Author.

information about the text to be encrypted to create the final ciphertext. In a database this information can be the primary key of the entry the fragments belong to. This guarantees that even equal fragments will be encrypted different. However, when a search is performed, the proxy must provide the encryption of the fragment. With few possible fragments, it is possible to infer what fragments the proxy is searching for after analyzing a number of queries. This problem was already present on the SEARCH, but the number of possible words is greater than the number of possible fragments (i.e., there are more possible full words than numbered word fragments). Hence, our modification makes it easier to deduce what is being searched given enough queries and worsens the already presented issue.

Unfortunately, the solution has drawbacks and one limitation. The fragment word search is still limited to fragments of a giving size. Moreover, if the fragment is partially in one token and partially in other (e.g. searching for the fragment “ry” in our example), this result will not be produced. Because of this, the wildcard operator is not fully enabled. To fully enable the wildcard operator, the token size must be made

equal to 1. This brings forth the important drawback of our solution.

The major drawback of our solution is that it increases the ciphertext size. Since each fragment must be individually encrypted to allow its search, there is a ciphertext expansion compared to the unmodified protocol. In the unmodified protocol, the words are divided to entirely fill the underlying SWP Pseudo Random Permutation (PRP) block. As a result, every PRP block encrypts as much plaintext as possible. However, on the modified SEARCH, each fragment demands an entire block. Consequently, each block is not entirely filled and more blocks are needed to encrypt the same plaintext. The need of more blocks to encrypt the same plaintext produces larger ciphertexts. However, the ciphertext size and consequently its expansion rate are independent from individual plaintext sizes. All plaintexts should be expanded to the same size (e.g., by adding space characters at the end) to avoid inference attacks. If plaintexts of multiple sizes are simple encrypted, the resulting ciphertexts will also have different sizes. Then, an attacker who looks at the encrypted database can associate larger ciphertexts with larger plaintexts and infer the original plaintext size. As a consequence, the data storage expansion ratio from the original to the modified search is constant.

The final drawback of our solution is that the query translation by the proxy into multiple queries increases the network data transmitted as the overall query size will be greater. In the original scheme, the full encrypted word must be sent by the proxy to the cloud database. In our modified scheme, the proxy must send the fragment with every possible word position. Although the search for a first or last character only demands one fragment (the one which is searched with the position 1 or $\vdash$), the proxy must still complete the query data with other (invalid) fragments until the final ciphertext size is achieved. The reasoning behind sending unneeded data is similar to the one presented for the immutability of SEARCH's ciphertext size. If the proxy sends just one fragment, an attacker can infer that the database is searching for a starting or

ending character. This way, the network data transmitted also increases by the same ratio as the ciphertext size.

5.3 Security Improvement in DET

CryptDB's deterministic layer (DET) uses the CBC-masks-CBC (CMC) mode with a zero initialization vector (IV) presented in chapter 2 to encrypt plaintexts and allow equality checks. The use of a zero IV makes the algorithm deterministic and equal plaintexts are encrypted to equal ciphertexts. Also, CMC is used in the place of CBC because CBC leaks prefix information. As CMC applies a mask created using information about the entire plaintext, only full word equalities can be checked.

Encrypting equal plaintexts to equal ciphertexts poses a security issue. As described in section 3.3, inference and l_p-Optimization attacks are possible. Fields such as age, date of birth, cities and so on are the ideal scenario for those attacks because the diversity of plaintexts and, consequently, ciphertexts is small. Unfortunately, these fields are also very likely to be encrypted for DET.

To mitigate these problems, we propose a modification to DET. CMC encryption mode should be substituted by the SWP algorithm already used in SEARCH with a singularity allowed by a database. Noting that SWP uses an information about the plaintext to encrypt it, we propose the algorithm to use the database primary key associated with that plaintext. The primary key is guaranteed to be unique for every and each entry in a database. As the primary key is unique, its encryption will also be unique and when the exclusive-or (XOR) with the encrypted plaintext is performed by SWP, the result will also be unique. It stands true even if equal plaintexts are encrypted, as each will have a different primary key and, then, different ciphertexts. Although each ciphertext is unique, equality checks will continue to be possible due to SWP structure.

The substitution creates some drawbacks. Firstly, it is not possible to build an index to perform the equality check as every single ciphertext has to be searched to determine if a match occurred. In addition, SWP has to perform an XOR, an encryption and a comparison to check for a match, while CMC mode required a simple comparison. This increases the time to perform an equality check. Secondly, operations that required a simple ciphertext mismatch, such as COUNT DISTINCT, need the proxy's help to be performed. The proxy chooses one entry at random, decrypts it and sends the server the SWP search term for that entry. The server goes through each entry in the database and performs the equality check, marking every successful check. Next, the proxy chooses one not marked entry at random and repeats the process. The server goes through every non-marked entry and performs the equality check, putting a different mark on every successful result. The process is repeated until every entry is marked. Then, the server can simple perform a COUNT DISTINCT on the marks to produce the desired result. This process increases the time to perform the operation greatly. Furthermore, now the server and the proxy have to engage in an interactive protocol, also increasing the network traffic.

5.3.1 Alternate modification to DET

Another possible modification to DET is the substitution of CMC mode by a simple hash with a pepper. A pepper is a constant secret string concatenated with the hash's input.

Since a hash output is different for every different input, even plaintexts that share all bits but one have different hash values. The pepper's function is to be an affix to randomize the hash's output. Because the pepper is a secret, an attacker cannot build a lookup table to discover the hash input. Additionaly, an attacker cannot recover a secure hash preimage, by definition. Also, the hash's output has a fixed length, preventing inference attacks due to the database entry size.

The modification promotes an efficiency gain and has one drawback. For the reason that a hash is faster than CMC, the cost to create the DET is greatly reduced. Moreover, the database can perform any DET operation using the hash values.

The major drawback is that the ciphertext can no longer be decrypted since it would be necessary to compute a hash preimage. As CryptDB works with multiple onions, this can be circumvented by separating the DET layer even further. The new DET layer would be used just for DET operations and the ciphertext of other onions would be used to provide the data for decryption.

5.4 Summary

In this chapter, we presented our proposed modifications to the CryptDB system. The table 4 gives a brief summary of the planned modifications with the improvements they bring and their drawbacks.

Table 4: Proposed modifications

Improvement	Modification	Drawback
Efficiency	Replaces PHPE for FAHE in HOM	Increased ciphertext size (security is limited by RND)
	Replaces CMC for peppered hash in DET	Additional exclusive onion required for DET
Functionality	Modification in SEARCH to allow wildcard operator	Still limited to token size; ciphertext expansion; bigger SQL query
Security	Replaces AES-CMC for SWP in DET	Operation takes longer to process; Increased network traffic

Source: Author.

6 RESULTS

In this chapter, we present the results of the modifications proposed in chapter 5.

The modifications were compared outside the CryptDB's system for two reasons. The first is to prevent noise from other CryptDB's operations not related with the tested solutions. The second is that CryptDB's code is convoluted and any modification to it is extremely time consuming. Nonetheless, another team is implementing the changes in CryptDB to compare both versions.

6.1 Efficiency Improvement in CryptDB's Homomorphic Addition Layer (HOM)

In this section we present the experimental results of the Fast Additive Homomorphic Encryption (FAHE) algorithm and we compare them to the Paillier Homomorphic Probabilistic Encryption (PHPE).

For completeness, we present the results and comparison of PHPE and both FAHE1 and FAHE2.

6.1.1 Experimental Results

We implemented FAHE1 and FAHE2 using the RELIC toolkit (ARANHA; GOUVÊA,) version 0.4.1 in C language, and used the Paillier implementation provided by the same library. The benchmark was performed on an Intel Core i7-7700K CPU,

operating at 4.2 Ghz and running on an Ubuntu 16.04.4 LTS operating system. The results are measured in cycles using the instruction RDTSCP (Read Time-Stamp Counter and Processor ID) (PAOLONI, 2010). We set the big number precision to twice the size of the ciphertext. We noted that RELIC's Paillier decryption computes the $L(g^{\lambda} \bmod n^2)^{-1} \bmod n$ term for every call. As this term can be precomputed, we subtracted the cycles needed for this operation from the decryption result.

The benchmarked operations were key generation, encryption, decryption and homomorphic addition. The results hereby presented are given in cycles, and correspond to the average of 10000 executions of each operation, which resulted in a standard deviation below 1%. Aiming at obtaining an uniform distribution for input messages, random samples were generated beforehand for each test (the generation time is not included in the benchmark).

For Paillier, we adopt 3072 bits for n. This is equivalent to RSA-3072, which provides a 128-bits security level (BARKER; DANG, 2016, Table 2). For FAHE1 and FAHE2, we use $\lambda = 128$, $|m_{max}| = 32$, and the minimum α for each scheme; we also consider $\lambda = 128$, $|m_{max}| = 64$, and $\alpha = 33$. Aiming at obtaining a post-quantum secure implementation, we also benchmark both schemes for $\lambda = 256$, $|m_{max}| = 64$, $\alpha = 33$.

The cycles needed for each FAHE1 and FAHE2 process (key generation, encryption, homomorphic addition and decryption) are presented, respectively, in Tables 5 and 6. For convenience, we repeat in both tables the number of cycles needed for the same processes in the Paillier cryptosystem, presenting the comparative gain of each proposed scheme. As shown in these tables, the more time consuming process is all schemes is the key generation. The reason is that FAHE1 and FAHE2 require the creation of a single prime of size η, whereas Paillier involves the the generation of two primes of size around $n/2$ (i.e., 1536 bits).

Finally, the cycles needed for FAHE1 and FAHE2 considering a quantum computer scenario are presented in Table 7. We show only the results for $|m_{max}| = 64$ and

Table 5: FAHE1 results (in cycles) compared to Pailler at the same (pre-quantum) $\lambda = 128$ security level.

Process	$\|m_{max}\| = 32$ $\alpha = 6$	$\|m_{max}\| = 64$ $\alpha = 33$	Pailler	Gain (Paillier/FAHE1)
KeyGen	16341812	117428129	2253611712	19.19
Enc	330027	2895834	351458572	121.37
Add	3045	25574	211829	8.28
Dec	293469	13388410	347251790	25.94

Source: Author.

Table 6: FAHE2 results (in cycles) compared to Pailler at the same (pre-quantum) $\lambda = 128$ security level.

Process	$\|m_{max}\| = 32$ $\alpha = 32$	$\|m_{max}\| = 64$ $\alpha = 33$	Pailler	Gain (Paillier/FAHE2)
KeyGen	17651582	23832773	2253611712	94.56
Enc	254619	294839	351458572	1192.04
Add	1820	2384	211829	88.85
Dec	198096	262719	347251790	1321.76

Source: Author.

$\alpha = 33$ for both algorithms, since this is the worst-case performance scenario for both schemes. Since Paillier is not quantum secure, we do not provide any direct comparison with it.

6.1.2 Analysis and comparison with Paillier

As shown in in Tables 5 and 6, improved performance is the main advantage of FAHE1 and FAHE2 in comparison to Paillier. Specifically, FAHE1 is around 20 times faster than Paillier for generating keys. It is also over 120 times faster for encrypting a message and over 25 times quicker to decrypt a ciphertext. There is also an 8

Table 7: FAHE1 and FAHE2 results (in cycles) for $\lambda = 256$, $|m_{max}| = 64$ and $\alpha = 33$.

Process	FAHE1	FAHE2
KeyGen	213369339	52678469
Enc	5119900	426348
Add	45287	3413
Dec	40875523	480605

Source: Author.

times gain in the homomorphic operation. FAHE2 takes advantage of the shorter ciphertext relatively to FAHE1 and is even faster. The key generation gain is almost 95 times. The encryption is approximately 1200 times faster and the decryption is over 1300 times quicker than Paillier. Finally, the homomorphic operation has a gain of around 90 times. As an additional benefit, the proposed schemes can also be made resistant to quantum computer attacks, whereas the underlying problem in the Paillier cryptosystem is known to be vulnerable to attacks by quantum computers.

On the other hand, FAHE1 and FAHE2 also present some drawbacks compared to Paillier. In particular, the performance gains of both schemes are traded by larger ciphertexts than those computed with Paillier. For example, with the suggested parameters the ciphertext expansion is approximately 50 and 5 times using FAHE1 and FAHE2, respectively, assuming the largest message space and number of supported additions.

In addition, Paillier can encrypt messages in the $[0, n-1]$ interval and perform homomorphic additions until an overflow in the message. Therefore, when dealing with 32-bit and 64-bit input messages, it would support up to $2^{|n|-32}$ and $2^{|n|-64}$ additions, respectively. For concrete parameters, such as 3072 bits for n, this means that Paillier supports a practically unlimited number of additions. In comparison, the number of additions of FAHE1 and FAHE2 is bounded by $2^{\alpha-1}$, where α is a chosen parameter. In our examples, we chose the largest α as 33, although some scenarios may employ a larger value. All in all, FAHE1 and FAHE2 allow speed-ups for many practical values of α. This flexibility is not available in the Paillier cryptosystem, for which the number of supported operations is way beyond the need of any conceivable real-world application.

Finally, another distinctive feature of FAHE1 and FAHE2 is that they rely on symmetric keys, in contrast with Paillier's asymmetric cryptosystem. More precisely, Paillier allows several entities to produce encrypted data using the target user's public key;

this enables the construction, for example, of collaborative databases where in principle even untrusted users can input data. Conversely, FAHE1 and FAHE2 are symmetric cryptosystems, since the same secret key is required for encryption and decryption (although not for homomorphic additions); therefore, every data source must be entrusted with the secret key before it can participate in the system.

In CryptDB, the proxy is the only entity entrusted with secret keys. Hence, Paillier can be substituted by our schemes without loss of functionality. For this reason, we recommend the substitution of Paillier by FAHE2 in CryptDB.

6.2 Efficiency Improvement in CryptDB's Deterministic Layer (DET)

In this section we present the experimental results of the exchange of the AES algorithm using CBC-masks-CBC (CMC) mode with a constant initialization vector (IV) for a peppered hash function. The hash function is the SHA2 (Federal Information Processing Standards Publication 180-4, 2015).

We present the cycles needed to encrypt data using both methods as well as the additional storage required by our alternative. The operation performed by the DET layer is not impacted in any way since our approach also produces deterministic ciphertexts. Actually, the deterministic ciphertext produced by our solution is smaller than the one produced by the original algorithm. Hence, a simple equality check is quicker than the unmodified version as the comparison element is smaller.

6.2.1 Experimental Results

Similar to subsection 6.1.1, we implemented both AES-CMC and the peppered SHA2 method using the RELIC toolkit version 0.4.1 in C language. The benchmark was performed on the same Intel Core i7-7700K CPU, but running on an Ubuntu

18.04.1 LTS operating system. The results are once again measured in cycles using the instruction Read Time-Stamp Counter and Processor ID (RDTSCP). The results are the mean of 10000 of each operation so it results in a standard deviation below 1%.

We used Thunderbird's English dictionary as the input to be encrypted. The dictionary has 49057 entries encoded using the Unicode Transformation Format using 8 bits (UTF-8) (Unicode Consortium, 2018). For the AES-CMC method, each word in the dictionary is expanded to 256 bytes by padding 0's to it. We expanded the word to 256 bytes as this is the maximum length supported by CryptDB's rows and every word has to have the same size to avoid inference attacks. If the word was not expanded, each AES-CMC ciphertext would have a different size and an attacker could infer the plaintext word size using this information. Differently, the word does not need to be expanded for the peppered SHA2 method. Since the output of a hash has a fixed length independently of the input size, the expansion is not required. Additionally, the constant IV for AES-CMC is set to the string of 0s.

Table 8 presents the hiding (encrypting or hashing) cycles results for both methods. We also present the cycles required to perform a comparison where the data is found (positive) and not found (negative). Considering that both methods provide deterministic outputs, the search structure (e.g., indexed, binary tree) is the same and any difference comes from the individual search time. We remember that the peppered SHA2 method cannot be decrypted and needs an additional layer on the database. The additional layer adds 256 bits for each hidden row. To decrypt data using our method, another onion of CryptDB must be used. Hence, a decryption comparison is not possible. Finally, the key generation for both methods is not compared because it is a random selection of 128 bits for the AES-CMC and a random selection of 256 bits for the peppered hash. Considering that random selections of bytes are extremely fast, any difference is negligible.

Table 8 shows an improvement of around 7.4 times for hiding the entire dictionary

Table 8: AES-CMC and peppered SHA2 results (in cycles) to hide a 49057 entries dictionary and to search a single entry.

		AES-CMC	Peppered SHA2
Hiding (entire dictionary)		697580460	93961082
Search (single entry)	Positive	2110	444
	Negative	2018	290

Source: Author.

using our method. It also shows an improvement of 4.75 and 6.95 times for searching a single entry when the result is “found” and “not found”, respectively. Although the search function is not isochronous, it does not represent a problem and is beneficial in this scenario. It does not represent a problem in this scenario because an adversary does not gain any advantage from it. The search function is a simple bitwise comparison between two known strings. The time analysis shows whether one of the strings matches or does not match the other, which is exactly the output of the function. It is beneficial because in the majority of cases, the number of strings that does not match the search term is greater than the number of strings that match the search term. Hence, it is beneficial to spend less time in this case that represents the larger part of the data.

6.3 Functionality Improvement in CryptDB’s Search Layer (SEARCH)

In this section, we present the experimental results of the modification in the SEARCH layer to allow wildcard searches to be performed.

We briefly present a theoretical storage expansion from the presented scheme to the modified version. Next, we present the experimental results in cycles to encrypt and search data using both schemes. We also present the experimental storage size of a dictionary using both methods.

6.3.1 Theoretical Storage Expansion

Consider w as the largest unencrypted word size possible in bytes. Take b as the Song, Wagner and Perrig Searcheable Encryption (SWP) Pseudo Random Permutation (PRP) block size in bytes. For the traditional search, to obtain the maximum ciphertext size we need to first divide w by b and round up:

$$r_t = \left\lceil \frac{w}{b} \right\rceil$$

The number of PRP blocks required do encrypt a word is given by r_t. Next, to find the final size s_t we need to multiply r_t by the block size b again:

$$\begin{aligned} s_t &= r_t \cdot b \\ &= b \cdot \left\lceil \frac{w}{b} \right\rceil \end{aligned}$$

For our fragmented search, to find the maximum ciphertext size, first we have to determine the token size t in bytes. For that, we need to discover how many bytes the word fragment has and how many bytes is required to represent the maximum fragment position in the word. Hence:

$$t = size_{word} + size_{pos}$$

Next, we need to discover how many PRP blocks are required to encrypt the token:

$$r_f = \left\lceil \frac{t}{b} \right\rceil$$

Finally, we multiply the previous result r_f by the maximum possible number of fragments n and by the PRP block size b. Therefore, the maximum ciphertext size in bytes for the fragmented search is:

$$\begin{aligned} s_f &= n \cdot b \cdot r_f \\ &= n \cdot b \cdot \left\lceil \frac{t}{b} \right\rceil \end{aligned}$$

Let us assume that the Unicode Transformation Format using 8 bits (UTF-8) is

used to represent the words. In CryptDB, the largest word possible has 256 characters and each character is represented by 1 byte. Additionally, let the word fragment be 1 character (i.e., perfect wildcard functionality with one letter each fragment). Consequently, there are 257 possible positions for each fragment, the 256 positions from the word plus the supplementary end character ⊢. Finally, we use AES as the PRP block, providing $b = 16$ bytes. Then, for the traditional search, we have $w = 256$ bytes and:

$$\begin{aligned} s_t &= b \cdot \left\lceil \frac{w}{b} \right\rceil \\ s_t &= 16 \cdot \left\lceil \frac{256}{16} \right\rceil \\ s_t &= 256 \end{aligned}$$

For the modified search, the word fragment size is 1 byte and we need 2 bytes to represent the maximum fragment position. This gives:

$$\begin{aligned} t &= size_{word} + size_{pos} \\ t &= 1 + 2 \\ t &= 3 \end{aligned}$$

And:

$$\begin{aligned} s_f &= n \cdot b \cdot r_f \\ s_f &= 256 \cdot 16 \cdot \left\lceil \frac{3}{16} \right\rceil \\ s_f &= 256 \cdot 16 \cdot 1 \\ s_f &= 4096 \end{aligned}$$

Thus, the storage expansion from the traditional search to the fragmented search is 16 times.

Finally, we note that using UTF-8 represents the worse scenario compared to using UTF-16 or UTF-32 (UTF using 16 and 32 bits). This is true because UTF-8 allows 16 characters to be encrypted within a single PRP block in the traditional algorithm, while UTF-16 allows 8 characters and UTF-32 allows only 4 characters. For the modified scheme, in the perfect wildcard scenario, the number of PRP blocks required to encrypt a word fragment is still 1 for both UTF-16 and UTF-32. If UTF-16 or UTF-32 are used,

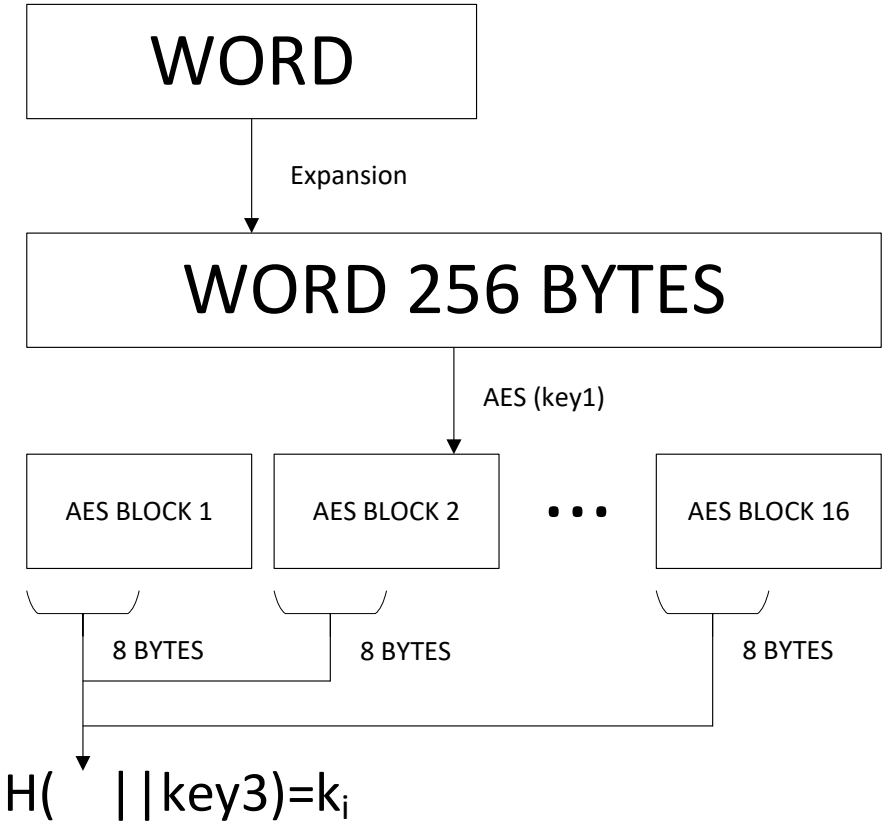
the storage expansion is 8 and 4 times, respectively.

6.3.2 Experimental Results

Similar to subsection 6.2.1, we implemented both search schemes using the RELIC toolkit version 0.4.1 in C language. The benchmark was performed on an Intel Core i7-7700K CPU running on an Ubuntu 18.04.1 LTS operating system. The results are measured in cycles using the instruction Read Time-Stamp Counter and Processor ID (RDTSCP). The results are the mean of 10000 of each operation so it results in a standard deviation below 1%.

We used Thunderbird's English dictionary as the input to be encrypted. The dictionary has 49057 entries encoded using the Unicode Transformation Format using 8 bits (UTF-8). For similar reasons presented in subsection 6.2.1, the traditional search has each word expanded to 256 bytes by padding 0's to it to prevent inference attacks. We create 3 keys, $key1$ of 128 bits, $key2$ of 256 bits and $key3$ of 256 bits. Then, the word is encrypted using the SWP algorithm. First, we encrypt the expanded word using AES-CBC with a constant IV (composed of 0's) and $key1$. Next, we produce an intermediate key k_i by hashing the concatenation of the first 8 bytes of every AES block and $key3$. These steps are illustrated in Figure 16. Then, we use the position of the word in the dictionary concatenated with $key2$ to generate the S block of SWP using SHA2. The output is successively concatenated with $key2$ and hashed until enough S blocks are created for the entire word. Following that, each S block is concatenated with k_i and hashed to produce the F blocks. The process is shown in Figure 17. Finally, each S and F are concatenated and XORed (exclusive-or operation) with the correct AES block, creating the final ciphertext, as presented in Figure 18. The only difference from this process to the practical CryptDB system is that instead of using the word position in the dictionary, CryptDB would use an encryption of the word's primary key in the database.

Figure 16: Word expansion and encryption together with the creation of the intermediate key k_i for the traditional search.

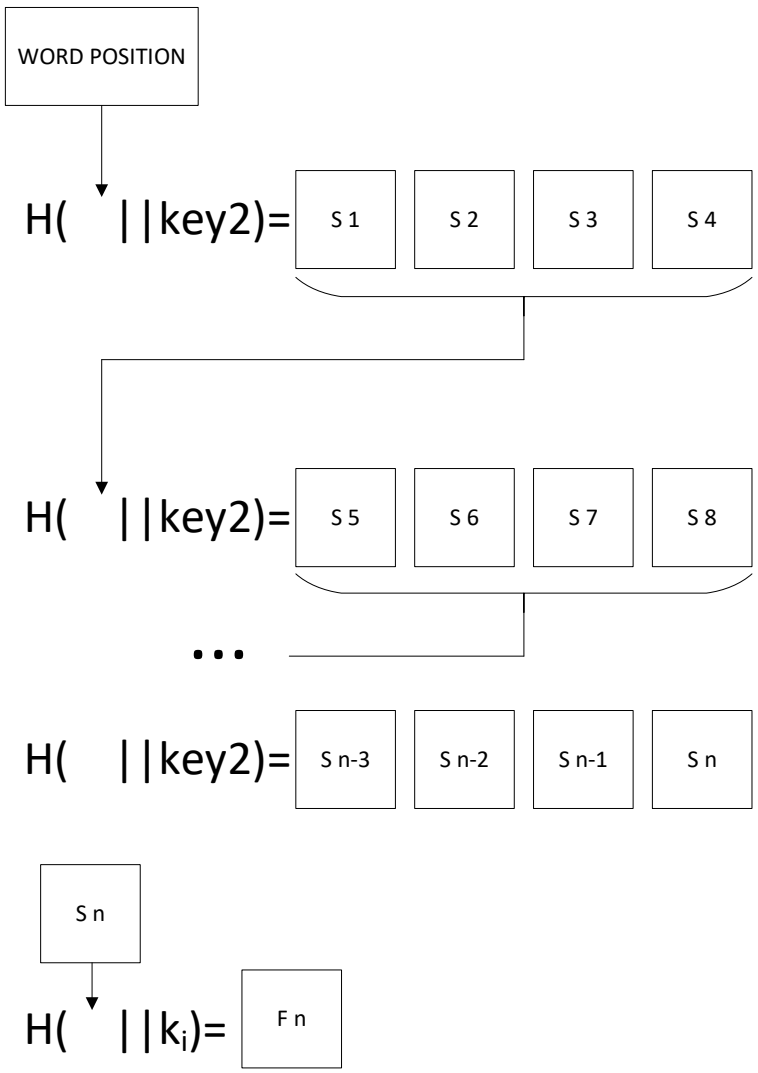


Source: Author.

For the modified search, the process is similar, but each character is encrypted individually. First, each character is concatenated with its position in the word. The last one is replicated with the supplementary end character ⊢. In our implementation, we used the 2 bytes `0xFFFF` to represent it. As every word has to have 256 characters to avoid inference attacks, we use random 16-byte strings to fill the remaining positions. Next, each fragment is encrypted using the AES cipher directly. Differently from the traditional search, multiple k_i are created, one for each fragment. The above steps are illustrated in Figure 19. After that, the S blocks are computed similarly to the traditional search and the F blocks are generated individually for each block with their respective k_i. Finally, the final ciphertext is created by XORing the AES block with the concatenation of the correct S and F.

To execute a search, a search token is created. The search token consists of the

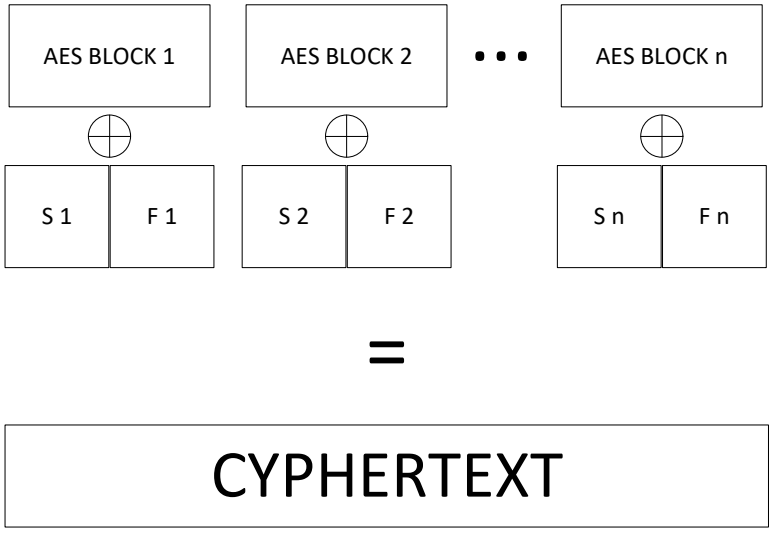
Figure 17: Creation of S and F blocks for both search algorithms.



Source: Author.

encrypted AES block and k_i. They are created analogously to the methods described above for the traditional and modified versions. We executed two tests, one for the traditional search and one for the wildcard search. For the traditional algorithm, a search token of an entire word is created. For the modified search, one token is created using a single character in a specific position to simulate a search for words that begin or end with it (e.g., “a*” or “*a”). The search for a single character in any position (a true wildcard search) can be estimated by multiplying the previous result by 256, which is the number of tokens required to cover every possible position for a character. Then the entire database is searched since the SWP algorithm is probabilistic and it is not

Figure 18: Creation of the final ciphertext for both search algorithms.



Source: Author.

Table 9: Traditional and modified Search results (in cycles) to encrypt a 49057 entries dictionary, to generate the search token and to search all encrypted entries.

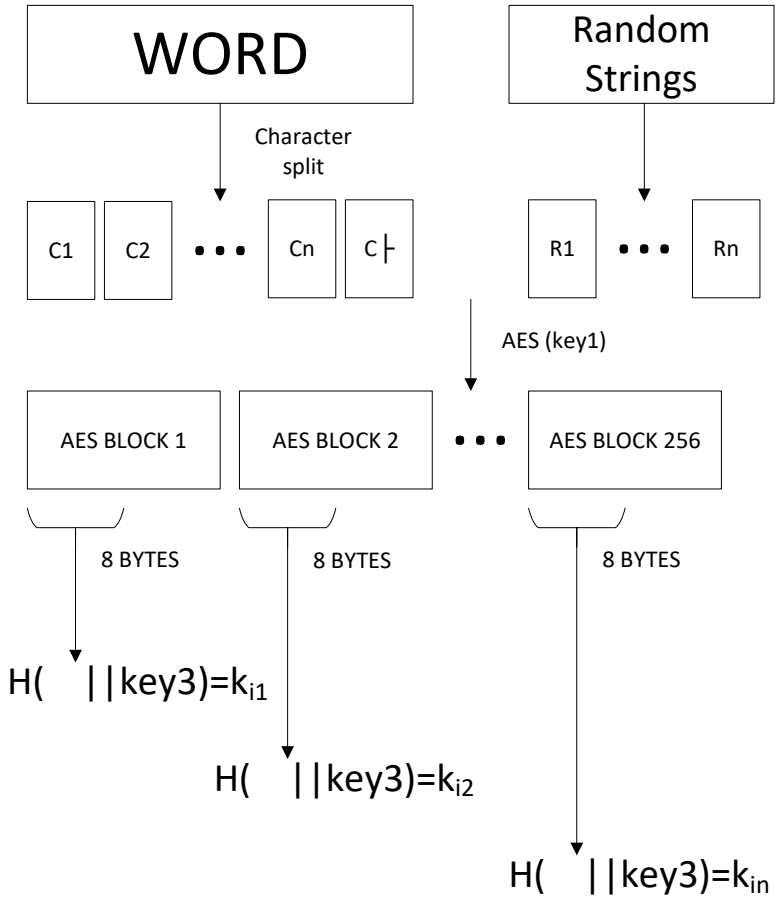
	Traditional (Full words)	Modified (Single character, fixed position)
Encrypting (entire dictionary)	2486218658	59039053083
Generate search token	9140	8420
Search (all encrypted entries)	1366376385	21827182139

Source: Author.

possible to order the ciphertexts in a search structure (e.g., different unrelated ciphertexts can be the encryption of the same word/fragment). Additionally, the wildcard search does not reveal the position of the fragment in the word. Since every character is marked with its original position, the characters can be shuffled before encryption. The result is that there is no correlation between the position of the encrypted fragment in the ciphertext and its plaintext position. Hence, if there is a positive match of the search token in the n-th block of the ciphertext, the match may not correspond to the n-th position of the plaintext word. If decryption is required, the ciphertext is decrypted and the plaintext is reassembled using the position associated with each character to yield the final output.

Table 9 shows the result in cycles to encrypt the dictionary using both methods. It

Figure 19: Character split and individual encryption together with the creation of the multiple intermediate keys k_i for the wildcard search.



Source: Author.

also presents the results to create a search token for one word in the traditional scheme and for one fragment in a fixed position in the modified version. Finally, it also gives the result in cycles to search the entire encrypted dictionary using this token.

The modified search needs to perform 16 times more AES encryptions and 256 times more hashes to generate the intermediate key k_i than the traditional algorithm. As a result, it is around 23.75 times slower to encrypt the entire dictionary compared to the full word scheme. To generate the search token, both schemes take around the same time, with a slight advantage to the modified scheme. Additionally, to search all the entries, the new search is about 16 times slower than the original search. The result is explained by the encrypted dictionary file size. The modified scheme has a file size of $201,220,704$ bytes and the original scheme possesses a file size of $12,841,824$

bytes. Hence, the result is expected since the data expansion between the two methods is 15.67 times and all data needs to be traversed for the process execution. We note that the file size expansion is not exactly the predicted 16 times from the theoretical analysis for we added an index to each entry. It was done purely to verify if the search was returning the correct result using the unencrypted dictionary as a reference.

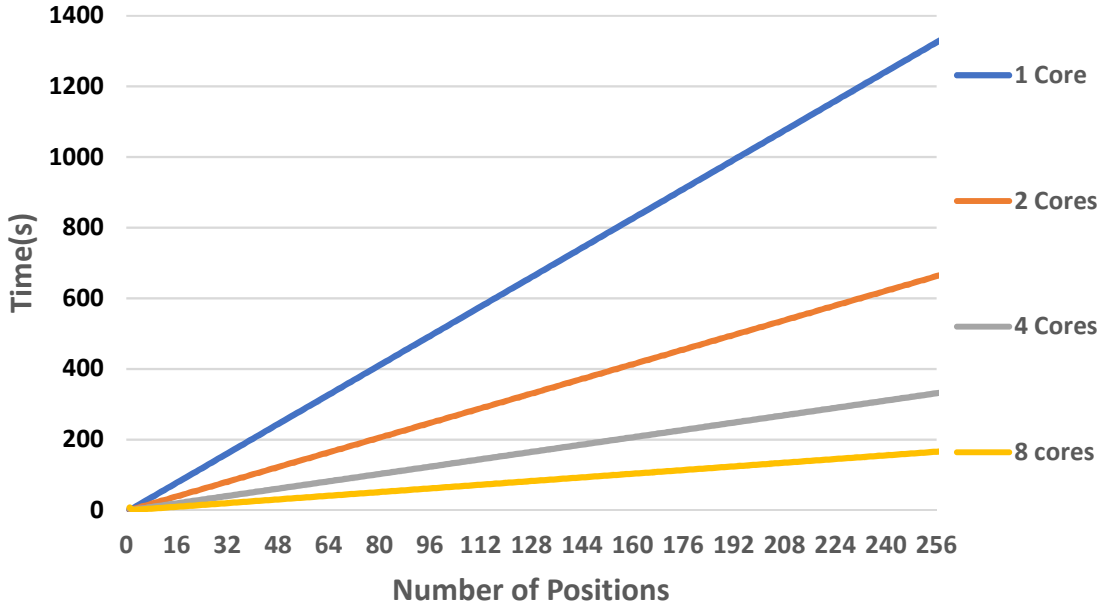
Finally, we can predict the number of cycles required to perform a search for a character in any position of the word. For this, we multiply the modified search result by 256, which is the number of tokens needed to cover every possible position in the word, resulting in 5587758627584 cycles. If we consider that our processor has a clock of 4.2GHz, this translates to roughly 1330 seconds, or 22 minutes. Although it seems impractical to implement a wildcard search in a real world database, we recall some facts that can improve the process: (1) the search can be performed simultaneously by multiple cores instead of a single core as in our test, since data is static during the algorithm execution; (2) considering that words are stored separately, we can create a smaller number of tokens (e.g., 30) and still cover every possible word since no word has 256 characters. As a result, if we use 30 tokens, the estimated number of cycles is 654815464170 which translates to 156 seconds. And using 4 cores to execute the search, the final time is estimated to be around 40 seconds to execute an encrypted wildcard search on a dictionary of 49057 entries. In Figure 20, we present the execution time to perform a wildcard search considering various numbers of possible positions and multiple number of cores available.

6.4 Security Improvement in CryptDB's Deterministic Layer (DET)

In this section, we present a discussion about the exchange of the AES-CMC by the SWP in the DET layer of CryptDB.

The swap of the algorithms requires some database operations performed by the

Figure 20: Execution time to perform a wildcard search using multiple cores and multiple possible positions.



Source: Author.

DET layer to be modified to ensure their correctness. We use data from section 6.2 and from section 6.3 for the AES-CMC and the SWP, respectively, together with a theoretical approach to evaluate the impact of the required changes.

6.4.1 Security Improvement Discussion

The replacement of AES-CMC by SWP makes the data from the DET layer probabilistic. The substitution thwarts inference attacks on these data, but some modifications have to be made so the layer's functionality is not affected. Moreover, some processes present an overhead compared to the original scheme. Let us assume the total number of entries present in the database is n and the number of different entries is n_d.

When a comparison is made (i.e., SQL operator "="), the Database Management System (DBMS) does not need to look all the database's entries when AES-CMC is used. Since the algorithm is deterministic, the DBMS can build an index for these

Table 10: Summary of the comparison operation using the deterministic AES-CMC algorithm with a b-tree and the probabilistic SWP algorithm, for n total entries and n_d number of different entries.

	AES-CMC	SWP
Complexity	$O(\log_b(n_d))$	$O(n)$
Cycles for each step	2018	27853

Source: Author.

values. We assume this index to be a b-tree, where each node has one of the different values stored in the database together with the primary keys of the rows where this value is present. To perform the operation, the DBMS has to simply search this tree and obtain the primary keys of the correct node. The complexity of this operation is $O(\log_b(n_d))$ and the cycles required for each step is the AES-CMC negative search for a single entry from Table 8, 2018 cycles. The negative search value is used since each step except the last is a negative match for the searched value.

If SWP is used, the data becomes probabilistic and no index can be created for them. As a result, the operation has to traverse the entire database, which has a complexity of $O(n)$. Moreover, the cycles needed to execute each step can be obtained by dividing the traditional search result from Table 9 by the dictionary size, 49057, which equals to 27853 cycles.

The summary of the comparison operation is presented in Table 10

Another operation that uses the deterministic layer is the SQL instruction "`COUNT DISTINCT`". When AES-CMC is used, the DBMS can perform this operation in a similar way as if the data were unencrypted. Since data is deterministic, the DBMS has to just count how many different entries are present and return this value. However, if the database adopts the SWP algorithm, the CryptDB proxy's help is required to implement the operation. Since data is probabilistic, equal values are recorded using different strings. To solve the problem, first the DBMS creates a counter and selects one random entry to be sent to the proxy. The proxy decrypts it and generates the search token for this value, which is returned. The DBMS searches all the data using

Table 11: Summary of the modifications required by the SWP algorithm for the `COUNT DISTINCT` operation, considering worst case scenario for n total entries and n_d number of different entries.

Interactions Database-Proxy	n_d
Number of elements searched	$\frac{n_d}{2} \cdot (2 \cdot n - (n_d - 1))$
Cycles spent on each element	27853

Source: Author.

this token and whenever a positive result is found, the row is marked with the counter value. After that, the counter is incremented and one unmarked value is sent to the proxy, where it will be decrypted and a search token created and returned. The DBMS searches all the unmarked data, marks the positive results and repeats the process until all data has been marked. Finally, the DBMS counts the different marks. The computation requires n_d interactions between DBMS and proxy. The worst case search scenario is when the data set is unbalanced and $n_d - 1$ values appear once and the n_d-th value composes the remaining entries and the n_d-th element is the one last searched. The number of elements searched in the worst case scenario is:

$$S = \frac{n_d}{2} \cdot (2 \cdot n - (n_d - 1))$$

We remember that each element takes 27853 cycles to be searched. A summary of the modifications required for the correctness of the `COUNT DISTINCT` operation is presented in Table 11. As a final remark, the `COUNT DISTINCT` removes the probabilistic nature for all the values currently presented in the database. If the data set does not change regularly, the use of SWP is discouraged if `COUNT DISTINCT` is expected to be used. After all, the algorithm exceedingly increases the complexity of the process without providing any practical benefits.

Any other operation which is performed by DET can be reduced to either the comparison and/or the `COUNT DISTINCT` method. Therefore, no further analysis about the exchange of the AES-CMC by the SWP is required.

6.5 Results Summary

In this chapter we presented the results of the modifications proposed in chapter 5.

Firstly, we showed the results of the main modification proposed, the exchange of the Paillier Homomorphic Probabilistic Encryption for the Fast Additive Homomorphic Encryption, described in chapter 4, in the homomorphic layer. The result reveals a performance gain for key generation, encryption, decryption and homomorphic addition. In particular, the second variation of our algorithm (FAHE2) offers a gain of almost 95 times, 1192 times, 1322 times, and 89 times respectively for the above mentioned processes.

Secondly, we presented the results of the exchange of the AES CBC-masks-CBC algorithm for a peppered hash function in the deterministic layer. The result shows a gain of approximately 7.5 times to hide the entire dictionary and a gain of 4.75 and 7 times to search a matching and non-matching entry, respectively. We remember that this modification has the drawback of requiring an additional layer in the database, which increments its storage space by 32 bytes per entry.

Thirdly, we showed the results of the modification in the search layer that allow wildcard searches to be performed. We enabled the wildcard search in the database but we increased the overhead required to perform the operation. In particular, the wildcard method is 23.75 times slower to encrypt the entire dictionary and around 16 times slower to search for a single character in a specified position. To perform a true wildcard search (i.e., a single character in any position) in a reasonable time, we had to make some workarounds, such as reducing the number of possible positions to 30 and using multiple cores to search the database.

Lastly, we discussed the exchange of the AES CBC-masks-CBC algorithm by the Song, Wagner and Perrig Searcheable Encryption (SWP) algorithm in the deterministic layer to improve the layer's security. We were able to improve the layer's secu-

rity by exchanging a deterministic algorithm for a probabilistic one while maintaining the layer's functionality. Nonetheless, we greatly increased the layer's overhead. In particular, the SQL operator "=" went from a complexity $O(\log_b(n_d))$ to $O(n)$, where b comes from the b-tree data structure, n is the number of entries in the database, and n_d is the number of different entries in the database. Moreover, the time to perform the comparison increased 13.8 times for each entry. Finally, the execution of the "`COUNT DISTINCT`" operation reduces all the probabilistic data currently present in the database to deterministic data. Hence, the modification is not recommended if the data set does not change regularly and the `COUNT DISTINCT` operation is expected to be used.

7 CONCLUSION

In this work, we presented methods to improve the CryptDB system, a cloud based encrypted database. We described the inner-workings of CryptDB, enumerating the various algorithms used and how they work together to provide multiple SQL functionality. Then, we raised some weaknesses of the system regarding its efficiency, functionality and security. We proceeded to address these issues by: (1) designing a new partially additive homomorphic algorithm which is more efficient than the current state of the art used on CryptDB; (2) exchanging the deterministic layer's AES-CMC algorithm for the SHA2 hash function to improve the layer's performance; (3) modifying the search layer's SWP algorithm to allow wildcard searches to be executed, adding a new functionality to the system; (4) replacing the deterministic layer's AES-CMC algorithm for the SWP to make data encryption probabilistic, increasing the layer's security.

According to our experimental results and analysis, all our solutions except 4 are satisfactory. Solutions 1 and 2 are able to improve the system performance by increasing the system storage. Solution 3 enables wildcard search if the size of the words to be stored are decreased to a still reasonable value. Finally, solution 4 increases the security of the database but imposes a considerable performance overhead and one operation requires special attention.

7.1 Publications

As a direct result of this work, a paper describing the new homomomorphic algorithm FAHE was submitted to the *IEEE Transactions of Information, Forensics and Security*. This paper is currently under review. Also, during the period of this work, we participated in the project Key Management for Vehicular Communications, supported by LG Electronics. This project produced the following publications:

- **(SIMPLICIO et al., 2018b):** In this article, we improve the Security Credential Management System (SCMS), which is one the leading candidates for protecting Vehicle-to-Infrastructure and Vehicle-to-Vehicle communications. In particular, we present two birthday attacks against SCMS's certificate revocation process and mitigate these attacks by applying security strings to the process. Our solution also further improves the certificate revocation procedure by increasing its flexibility through the use of linkage hooks.

- **(SIMPLICIO et al., 2018a):** In this article, we further expand upon the SCMS's certificate revocation process. We propose a new design called Activation Codes for Pseudonym Certificates (ACPC) in order to shorten the system's Certificate Revocation List (CRL). With ACPC, vehicles require short bit-strings to activate their pseudonym certificates. These bit-strings are periodically distributed by the system to non-revoked vehicles, while revoked vehicles cannot obtain their activation codes. Hence, a revoked vehicle needs to be kept in the CRL only during the time their current activation code is valid. Afterwards, the vehicle can be removed from the CRL, which shortens the list.

- **(SIMPLICIO et al., 2018c):** In this article, we expand upon the SCMS's butterfly key expansion process. The butterfly key expansion process allows SCMS to issue large batches of pseudonym certificates with a single request by a vehicle. The request consists of two separate private/public key pairs. We improve

the process by unifying both keys without reducing SCMS's security, flexibility or scalability. As a result, the bandwidth utilization for certificate provisioning is reduced. Moreover, we are able to remove a signature from the certificate process, lessening the system's processing overhead.

7.2 Future Work

The new partially additive homomorphic encryption algorithm described in chapter 4 is planned to be submitted to a journal or conference. We let as future work the integration of our solutions into the CryptDB system, as well as their benchmarking in a real cloud environment. This step can measure the real impact of our modifications in the system, where they will be subject to other CryptDB's operation noise and network overheads.

Further research on encrypted search algorithms are required. In particular, protocols which allow partial information to be matched (i.e., wildcard search) can be useful. Moreover, it is interesting to study a search mechanism that does not allow future data to be searched using previous search terms (e.g., a single use search token). Such mechanism would provide stronger forward secrecy to the system, as no future data could be linked to a previous one.

REFERENCES

ALKIM, E.; AVANZI, R.; BOS, J. W.; DUCAS, L.; PIEDRA, A. d. l.; PÖPPELMANN, T.; SCHWABE, P.; STEBILA, D. *NewHope*. 2017. <https://newhopecrypto.org/data/NewHope_2017_12_21.pdf>.

ALKIM, E.; BOS, J. W.; DUCAS, L.; LONGA, P.; MIRONOV, I.; NAEHRIG, M.; NIKOLAENKO, V.; PEIKERT, C.; RAGHUNATHAN, A.; STEBILA, D.; EASTERBROOK, K.; LAMACCHIA, B. *FrodoKEM*. 2017. <https://frodokem.org/files/FrodoKEM-specification-20171130.pdf>.

ARANHA, D. F.; GOUVÊA, C. P. L. *RELIC is an Efficient LIbrary for Cryptography*. <https://github.com/relic-toolkit/relic>.

ARASU, A.; BLANAS, S.; EGURO, K.; KAUSHIK, R.; KOSSMANN, D.; RAMAMURTHY, R.; VENKATESAN, R. Orthogonal security with cipherbase. In CITESEER. *CIDR*. Asilomar, CA, USA, 2013.

ARMKNECHT, F.; KATZENBEISSER, S.; PETER, A. Group homomorphic encryption: characterizations, impossibility results, and applications. *Designs, codes and cryptography*, Springer, p. 1–24, 2013.

AVANZI, R.; BOS, J.; DUCAS, L.; KILTZ, E.; LEPOINT, T.; LYUBASHEVSKY, V.; SCHANCK, J. M.; SCHWABE, P.; SEILER, G.; STEHLÉ, D. *CRYSTALS-Kyber: a CCA-secure module-lattice-based KEM*. 2018. <https://pq-crystals.org/kyber/data/kyber-specification.pdf>.

BAJAJ, S.; SION, R. Trusteddb: A trusted hardware based database with privacy and data confidentiality. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 2011. (SIGMOD '11), p. 205–216. ISBN 978-1-4503-0661-4. Available from Internet: <http://doi.acm.org/10.1145/1989323.1989346>.

BARKER, E.; DANG, Q. *NIST Special Publication 800–57 Part 1, Revision 4*. 2016.

BARKER, E.; ROGINSKY, A. Transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths. In NIST. 2015. Accessed August 16, 2017. Available from Internet: <http://dx.doi.org/10.6028/NIST.SP.800-131Ar1>.

BELLARE, M.; DESAI, A.; POINTCHEVAL, D.; ROGAWAY, P. Relations among notions of security for public-key encryption schemes. In SPRINGER. *Advances in Cryptology-CRYPTO'98*. Santa Barbara, CA, USA, 1998. p. 26–45.

BELLARE, M.; ROGAWAY, P. *Introduction to modern cryptography*. UC San Diego, 2005. Available from Internet: <https://cseweb.ucsd.edu/~mihir/cse207/classnotes.html>.

BENALOH, J. D. C. *Verifiable Secret-Ballot Elections,*. PhD Thesis (PhD) — Yale University, 1987.

BERNSTEIN, D. J.; HAMBURG, M.; KRASNOVA, A.; LANGE, T. Elligator: Elliptic-curve points indistinguishable from uniform random strings. In ACM. *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. Berlin, Germany, 2013. p. 967–980.

BINDEL, N.; AKLEYLEK, S.; ALKIM, E.; BARRETO, P. S. L. M.; BUCHMANN, J.; EATON, E.; GUTOSKI, G.; KRÄMER, J.; LONGA, P.; POLAT, H.; RICARDINI, J. E.; ZANON, G. *qTESLA*. 2018. <https://qtesla.org/wp-content/uploads/2018/07/qTESLA_v2.1_06.30.2018.pdf>.

BOLDYREVA, A.; CHENETTE, N.; LEE, Y.; O'NEILL, A. et al. Order-preserving symmetric encryption. In SPRINGER. *Eurocrypt*. Cologne, Germany, 2009. vol. 5479, p. 224–241.

BONEH, D. *Cryptography I*. 2017. Accessed July 12, 2017. Available from Internet: <https://www.coursera.org/learn/crypto/home/welcome>.

BUCKEL, C. *The Real Cost of Enterprise Database Software*. 2013. Accessed April 3, 2017. Available from Internet: <https://flashdba.com/2013/09/10/the-real-cost-of-enterprise-database-software/>.

CARMICHAEL, R. D. *The Theory of Numbers*. JOHN WILEY & SONS, Inc., NEW YORK, 1914. Accessed August 16, 2017. Available from Internet: <http://www.gutenberg.org/files/13693/13693-pdf.pdf>.

CHEON, J. H.; CORON, J.-S.; KIM, J.; LEE, M. S.; LEPOINT, T.; TIBOUCHI, M.; YUN, A. Batch fully homomorphic encryption over the integers. In JOHANSSON, T.; NGUYEN, P. Q. (Ed.). *Advances in Cryptology – EUROCRYPT 2013*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 315–335. ISBN 978-3-642-38348-9.

CHEON, J. H.; STEHLÈ, D. *Fully Homomophic Encryption over the Integers Revisited.* 2016. Cryptology ePrint Archive, Report 2016/837. <https://eprint.iacr.org/2016/837>.

CipherCloud. *Guide to Cloud Data Protection*. 2015. Available from Internet: <http://pages.ciphercloud.com/Guide-to-Cloud-Data-Protection.html>.

DAEMEN, J.; RIJMEN, V. AES proposal: Rijndael. 1999.

DIJK, M. v.; GENTRY, C.; HALEVI, S.; VAIKUNTANATHAN, V. Fully homomorphic encryption over the integers. In SPRINGER. *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. French Riviera, France, 2010. p. 24–43.

Diretoria de Regulação Prudencial, Riscos e Assuntos Econômicos. *Painel Econômico e Financeiro*. 2016. <https://cmsportal.febraban.org.br/Arquivos/documentos/PDF/-L06_painel_port.pdf>.

DUCAS, L.; KILTZ, E.; LEPOINT, T.; LYUBASHEVSKY, V.; SCHWABE, P.; SEILER, G.; STEHLÉ, D. *CRYSTALS-Dilithium: a Digital Signatures from Module Lattices*. 2018. <https://pq-crystals.org/dilithium/data/dilithium-specification.pdf>.

DURAK, F. B.; DUBUISSON, T. M.; CASH, D. What else is revealed by order-revealing encryption? In ACM. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. Vienna, Austria, 2016. p. 1155–1166.

EGOROV, M.; WILKISON, M. Zerodb white paper. *arXiv preprint arXiv:1602.07168*, 2016.

EHRSAM, W. F.; MEYER, C. H.; SMITH, J. L.; TUCHMAN, W. L. *Message verification and transmission error detection by block chaining*. 1978. US Patent 4,074,066.

ELGAMAL, T. A public key cryptosystem and a signature scheme based on discrete logarithms. In SPRINGER. *Advances in cryptology*. Santa Barbara, CA, USA, 1984. p. 10–18.

Federal Information Processing Standards Publication 180-4. Secure hash standard (SHS). In NIST. 2015. Available from Internet: <http://dx.doi.org/10.6028/NIST.FIPS.180-4>.

Federal Information Processing Standards Publication 197. Announcing the advanced encryption standard (AES). In NIST. 2001. Available from Internet: <https://doi.org/10.6028/NIST.FIPS.197>.

Federal Information Processing Standards Publication 202. SHA-3 standard: Permutation-based hash and extendable-output functions. In NIST. 2015. Available from Internet: <http://dx.doi.org/10.6028/NIST.FIPS.202>.

FOUQUE, P.-A.; JOUX, A.; TIBOUCHI, M. Injective encodings to elliptic curves. In SPRINGER. *Australasian Conference on Information Security and Privacy*. Brisbane, Australia, 2013. p. 203–218.

GALBRAITH, S. D.; GEBREGIYORGIS, S. W.; MURPHY, S. Algorithms for the approximate common divisor problem. *LMS Journal of Computation and Mathematics*, London Mathematical Society, vol. 19, no. A, p. 58–72, 2016.

GOLDWASSER, S.; BELLARE, M. *Lecture notes on cryptography*. 1996.

GROVER, L. K. A fast quantum mechanical algorithm for database search. In ACM. *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. Philadelphia, PA, USA, 1996. p. 212–219.

HALEVI, S.; ROGAWAY, P. A tweakable enciphering mode. In SPRINGER. *Annual International Cryptology Conference*. Santa Barbara, CA, USA, 2003. p. 482–499.

HOWGRAVE-GRAHAM, N. Approximate integer common divisors. In SPRINGER. *CaLC*. Providence, RI, USA, 2001. vol. 1, p. 51–66.

IETF. *Request for Comments (RFC) 5652*. 2009. Accessed July 3, 2017. Available from Internet: <https://tools.ietf.org/html/rfc5652>.

KOBLITZ, N. Elliptic curve cryptosystems. *Mathematics of computation*, vol. 48, no. 177, p. 203–209, 1987.

KOLESNIKOV, V.; SHIKFA, A. On the limits of privacy provided by order-preserving encryption. *Bell Labs Technical Journal*, Wiley Online Library, vol. 17, no. 3, p. 135–146, 2012.

Library of Congress. Library of Congress - Statistics. 2018. <https://www.loc.gov/about/fascinating-facts/>.

LIDL, R.; NIEDERREITER, H. *Introduction to finite fields and their applications*. Cambridge, United Kingdom: Cambridge university press, 1994.

LIPMAA, H.; ROGAWAY, P.; WAGNER, D. *CTR-mode encryption*. Baltimore, MD,USA: First NIST Workshop on Modes of Operation, 2000.

MENEZES, A. J.; OORSCHOT, P. C. V.; VANSTONE, S. A. *Handbook of applied cryptography*. Boca Raton, FL, United States: CRC press, 1996.

Merriam-Webster. *Database*. 2017. Accessed March 29, 2017. Available from Internet: <https://www.merriam-webster.com/dictionary/database>.

NAVEED, M.; KAMARA, S.; WRIGHT, C. V. Inference attacks on property-preserving encrypted databases. In ACM. *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. Denver, CO, USA, 2015. p. 644–655.

NIST. Block cipher modes. In . 2017. Accessed July 2, 2017. Available from Internet: <http://csrc.nist.gov/groups/ST/toolkit/BCM/index.html>.

ORACLE. *Cloud Computing Comes of Age*. 2015. Available from Internet: <https://www.oracle.com/webfolder/s/delivery_production/docs/FY15h1/doc16/HBR-Oracle-Report-webview.pdf>.

PAILLIER, P. Public-key cryptosystems based on composite degree residuosity classes. In SPRINGER. *Advances in cryptology - EUROCRYPT'99*. Prague, Czech Republic, 1999. p. 223–238.

PAOLONI, G. How to benchmark code execution times on intel ia-32 and ia-64 instruction set architectures. *Intel White Paper*, p. 324264–001, 2010.

PODDAR, R.; BOELTER, T.; POPA, R. A. *Arx: A Strongly Encrypted Database System*. 2016. Cryptology ePrint Archive, Report 2016/591. <http://eprint.iacr.org/2016/591>.

POPA, R. A.; LI, F. H.; ZELDOVICH, N. An ideal-security protocol for order-preserving encoding. In IEEE. *2013 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA, 2013. p. 463–477.

POPA, R. A.; REDFIELD, C.; ZELDOVICH, N.; BALAKRISHNAN, H. Cryptdb: protecting confidentiality with encrypted query processing. In ACM. *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. Cascais, Portugal, 2011. p. 85–100.

POPA, R. A.; ZELDOVICH, N. Cryptographic treatment of cryptdb's adjustable join. 2012.

Privacy Rights Clearinghouse. *Chronology of data breaches*. 2017. Accessed March 29, 2017. Available from Internet: <http://privacyrights.org/data-breach>.

REGEV, O. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, ACM, vol. 56, no. 6, p. 34, 2009.

RIVEST, R.; ADLEMAN, L.; DERTOUZOS, M. On data banks and privacy homomorphisms. *Foundations of secure computation*, Citeseer, vol. 4, no. 11, p. 169–180, 1978.

ROGGERO, H. *Sample Pricing Comparison: On-Premise vs. Private Hosting vs. Cloud Computing*. 2013. Accessed March 29, 2017. Available from Internet: <http://wblo.gs/dbE>.

ROSULEK, M. *The Joy of Cryptography*. Oregon State University, 2018. Accessed August 22, 2018. Available from Internet: <https://web.engr.oregonstate.edu/~rosulekm/crypto/>.

SAARINEN, M.; AUMASSON, J. *The BLAKE2 cryptographic hash and message authentication code (MAC)*. 2015.

SCHULZE, H. *Cloud Security Spotlight Report*. 2016. Available from Internet: <https://pages.cloudpassage.com/rs/857-FXQ-213/images/cloud-security-survey-report-2016.pdf>.

SHOR, P. W. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, vol. 26, no. 5, p. 1484–1509, 1997.

SILVERMAN, J. H. *The arithmetic of elliptic curves*. Berlin, Germany: Springer Science & Business Media, 2009. vol. 106.

SIMPLICIO, M. A.; COMINETTI, E. L.; PATIL, H. K.; RICARDINI, J. E.; SILVA, M. V. M. ACPC: Efficient revocation of pseudonym certificates using activation codes. *Ad Hoc Networks*, 2018. ISSN 1570-8705. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S1570870518304761>.

SIMPLICIO, M. A.; COMINETTI, E. L.; PATIL, H. K.; RICARDINI, J. E.; FERRAZ, L. T. D.; SILVA, M. V. M. da. Privacy-preserving method for temporarily linking/revoking pseudonym certificates in vanets. In *17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications / 12th IEEE International Conference On Big Data Science And Engineering, TrustCom/BigDataSE 2018, New York, NY, USA, August*

1-3, 2018. IEEE, 2018. p. 1322–1329. Available from Internet: <https://doi.org/10.1109/TrustCom/BigDataSE.2018.00182>.

SIMPLICIO, M. A.; COMINETTI, E. L.; PATIL, H. K.; RICARDINI, J. E.; SILVA, M. V. M. da. The unified butterfly effect: Efficient security credential management system for vehicular communications. In *IEEE Vehicular Networking Conference (VNC), Taipei, Taiwan, December 5-7, 2018*. IEEE, 2018. p. 1–9. Available from Internet: <https://eprint.iacr.org/2018/089.pdf>.

SONG, D. X.; WAGNER, D.; PERRIG, A. Practical techniques for searches on encrypted data. In *2000 IEEE Symposium on Security and Privacy*. Berkeley, CA, USA: IEEE, 2000. p. 44–55. ISSN 1081-6011.

STEVENS, M.; BURSZTEIN, E.; KARPMAN, P.; ALBERTINI, A.; MARKOV, Y. The first collision for full sha-1. *IACR Cryptology ePrint Archive*, vol. 2017, p. 23, 2017.

Unicode Consortium. *The Unicode Standard - Version 11.0 - Core Specification*. 2018. <http://www.unicode.org/versions/Unicode11.0.0/UnicodeStandard-11.0.pdf>.

YAO, A. C.-C. How to generate and exchange secrets. In IEEE. *27th Annual Symposium on Foundations of Computer Science, 1986.* Toronto, Canada, 1986. p. 162–167.

APPENDIX A – A KEY RECOVERY ATTACK EXAMPLE ON FAHE

In this appendix, we implement the key recovery attack to the encryption scheme proposed in chapter 4.

WE implemented the attack using FAHE2 with security parameter $\lambda = 128$, maximum message size $|m_{max}| = 64$, and total number of supported additions $\alpha = 29$. We decided to use FAHE2 for its shorter ciphertext size. Nonetheless, the steps are analogous for FAHE1.

We executed the key generation and obtained the parameters $pos = 98$ and key

$$p = \texttt{0x03482A6BC4304B9816A2E25C08997689}$$
$$\texttt{8F72E333B7C9B16AA3E6BA90FBD95FCB}$$

Finally, as discussed in subsection 4.4.4, we must provide two arbitrary ciphertexts of length $\gamma = 23866$ bits. We choose

$$c1 = \texttt{0x200}\ldots\texttt{00000000000000000000000000000000}$$
$$\texttt{00000000000000000000000000000000}$$

and

$$c2 = \texttt{0x200}\ldots\texttt{10000000000000000000000000000000}$$
$$\texttt{00000000000000000000000000000000}$$

Initially, for ciphertext $c1$ we obtained the decryption $d = \texttt{0xF2F69369DBED0E1B}$.

We show now a step-by-step example of the attack using ciphertext $c1$. For convenience, we present only the last 32 bytes of $c1$ since every bit flip occurs in this range. We denote as Q the ciphertext query to the decryption oracle and as A its answer. The ciphertext's bits are flipped from the most significant to the least significant bit, one at a time. We remember that if the decryption changes drastically from the initial one, the bit is returned to 0. Otherwise, the bit is kept at 1. Additionally, when we change the bits in the same position where the message is stored (i.e., bits 191 to 127 in this example), the decryption is affected. A small change in the decryption from the initial value (i.e., 1 or 2 bits) represents that the bit must be kept at 1. Moreover, this decryption value is assumed as the new base value for future comparisons.

Q: 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: C5 B0 CE B1 CA BA 21 08

Q: 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: C5 B0 CE B1 CA BA 21 08

Q: 00 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: F2 F6 93 69 DB ED 0E 1B

Q: 00 C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: F2 F6 93 69 DB ED 0E 1B

Q: 00 E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: C5 B0 CE B1 CA BA 21 08

Q: 00 D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: C5 B0 CE B1 CA BA 21 08

Q: 00 C8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: C5 B0 CE B1 CA BA 21 08

Q: 00 C8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: C5 B0 CE B1 CA BA 21 08

Q: 00 C4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: C5 B0 CE B1 CA BA 21 08

Q: 00 C2 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: C5 B0 CE B1 CA BA 21 08

Q: 00 C1 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: F2 F6 93 69 DB ED 0E 1B

Q: 00 C1 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: C5 B0 CE B1 CA BA 21 08

Q: 00 C1 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: F2 F6 93 69 DB ED 0E 1B

Q: 00 C1 60 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: F2 F6 93 69 DB ED 0E 1B

Q: 00 C1 70 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: C5 B0 CE B1 CA BA 21 08

Q: 00 C1 68 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: F2 F6 93 69 DB ED 0E 1B

Q: 00 C1 6C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: C5 B0 CE B1 CA BA 21 08

Q: 00 C1 6A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: F2 F6 93 69 DB ED 0E 1B

Q: 00 C1 6B 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: F2 F6 93 69 DB ED 0E 1B

Q: 00 C1 6B 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: F2 F6 93 69 DB ED 0E 1B

Q: 00 C1 6B C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00

A: C5 B0 CE B1 CA BA 21 08

Q: 00 C1 6B A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00

A: F2 F6 93 69 DB ED 0E 1B

Q: 00 C1 6B B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00

A: C5 B0 CE B1 CA BA 21 08

Q: 00 C1 6B A8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00

A: C5 B0 CE B1 CA BA 21 08

Q: 00 C1 6B A4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00

A: F2 F6 93 69 DB ED 0E 1B

Q: 00 C1 6B A6 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00

A: F2 F6 93 69 DB ED 0E 1B

Q: 00 C1 6B A7 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00

A: C5 B0 CE B1 CA BA 21 08

Q: 00 C1 6B A6 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: F2 F6 93 69 DB ED 0E 1B

Q: 00 C1 6B A6 C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: C5 B0 CE B1 CA BA 21 08

Q: 00 C1 6B A6 A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: F2 F6 93 69 DB ED 0E 1B

Q: 00 C1 6B A6 B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: F2 F6 93 69 DB ED 0E 1B

Q: 00 C1 6B A6 B8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: C5 B0 CE B1 CA BA 21 08

Q: 00 C1 6B A6 B4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: F2 F6 93 69 DB ED 0E 1B

Q: 00 C1 6B A6 B6 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: C5 B0 CE B1 CA BA 21 08

Q: 00 C1 6B A6 B5 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: F2 F6 93 69 DB ED 0E 1B

Q: 00 C1 6B A6 B5 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: C5 B0 CE B1 CA BA 21 08

Q: 00 C1 6B A6 B5 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: F2 F6 93 69 DB ED 0E 1B

Q: 00 C1 6B A6 B5 60 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: C5 B0 CE B1 CA BA 21 08

Q: 00 C1 6B A6 B5 50 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: C5 B0 CE B1 CA BA 21 08

Q: 00 C1 6B A6 B5 48 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: C5 B0 CE B1 CA BA 21 08

Q: 00 C1 6B A6 B5 44 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: C5 B0 CE B1 CA BA 21 08

Q: 00 C1 6B A6 B5 42 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: C5 B0 CE B1 CA BA 21 08

Q: 00 C1 6B A6 B5 41 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: F2 F6 93 69 DB ED 0E 1B

Q: 00 C1 6B A6 B5 41 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: C5 B0 CE B1 CA BA 21 08

Q: 00 C1 6B A6 B5 41 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: F2 F6 93 69 DB ED 0E 1B

Q: 00 C1 6B A6 B5 41 60 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: F2 F6 93 69 DB ED 0E 1B

Q: 00 C1 6B A6 B5 41 70 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: F2 F6 93 69 DB ED 0E 1B

Q: 00 C1 6B A6 B5 41 78 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: C5 B0 CE B1 CA BA 21 08

Q: 00 C1 6B A6 B5 41 74 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: C5 B0 CE B1 CA BA 21 08

Q: 00 C1 6B A6 B5 41 72 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: C5 B0 CE B1 CA BA 21 08

Q: 00 C1 6B A6 B5 41 71 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: C5 B0 CE B1 CA BA 21 08

Q: 00 C1 6B A6 B5 41 70 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: C5 B0 CE B1 CA BA 21 08

Q: 00 C1 6B A6 B5 41 70 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: F2 F6 93 69 DB ED 0E 1B

Q: 00 C1 6B A6 B5 41 70 60 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: C5 B0 CE B1 CA BA 21 08

Q: 00 C1 6B A6 B5 41 70 50 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: F2 F6 93 69 DB ED 0E 1B

Q: 00 C1 6B A6 B5 41 70 58 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: F2 F6 93 69 DB ED 0E 1B

Q: 00 C1 6B A6 B5 41 70 5C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: C5 B0 CE B1 CA BA 21 08

Q: 00 C1 6B A6 B5 41 70 5A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: C5 B0 CE B1 CA BA 21 08

Q: 00 C1 6B A6 B5 41 70 59 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: F2 F6 93 69 DB ED 0E 1B

Q: 00 C1 6B A6 B5 41 70 59 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: F2 F6 93 69 DB ED 0E 1B

Q: 00 C1 6B A6 B5 41 70 59 C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 45 B0 CE B1 CA BA 21 08 (message bits reached, change in every bit)

Q: 00 C1 6B A6 B5 41 70 59 A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 05 B0 CE B1 CA BA 21 08 (message bits reached, change in every bit)

Q: 00 C1 6B A6 B5 41 70 59 90 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 12 F6 93 69 DB ED 0E 1B (message bits reached, small change, new value)

Q: 00 C1 6B A6 B5 41 70 59 98 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 22 F6 93 69 DB ED 0E 1B (message bits reached, small change, new value)

Q: 00 C1 6B A6 B5 41 70 59 9C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2A F6 93 69 DB ED 0E 1B (message bits reached, small change, new value)

Q: 00 C1 6B A6 B5 41 70 59 9E 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 01 B0 CE B1 CA BA 21 08 (message bits reached, change in every bit)

Q: 00 C1 6B A6 B5 41 70 59 9D 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2C F6 93 69 DB ED 0E 1B (message bits reached, small change, new value)

Q: 00 C1 6B A6 B5 41 70 59 9D 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 00 B0 CE B1 CA BA 21 08 (message bits reached, change in every bit)

Q: 00 C1 6B A6 B5 41 70 59 9D 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 00 30 CE B1 CA BA 21 08 (message bits reached, change in every bit)

Q: 00 C1 6B A6 B5 41 70 59 9D 20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 36 93 69 DB ED 0E 1B (message bits reached, small change, new value)

Q: 00 C1 6B A6 B5 41 70 59 9D 30 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 00 10 CE B1 CA BA 21 08 (message bits reached, change in every bit)

Q: 00 C1 6B A6 B5 41 70 59 9D 28 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 00 00 CE B1 CA BA 21 08 (message bits reached, change in every bit)

Q: 00 C1 6B A6 B5 41 70 59 9D 24 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 3E 93 69 DB ED 0E 1B (message bits reached, small change, new value)

Q: 00 C1 6B A6 B5 41 70 59 9D 26 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 42 93 69 DB ED 0E 1B (message bits reached, small change, new value)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 44 93 69 DB ED 0E 1B (message bits reached, small change, new value)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 93 69 DB ED 0E 1B (message bits reached, small change, new value)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 00 00 4E B1 CA BA 21 08 (message bits reached, change in every bit)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 00 00 0E B1 CA BA 21 08 (message bits reached, change in every bit)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 90 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 B3 69 DB ED 0E 1B (message bits reached, small change, new value)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 93 69 DB ED 0E 1B (message bits reached, small change, new value)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 9C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 00 00 06 B1 CA BA 21 08 (message bits reached, change in every bit)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 9A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 00 00 02 B1 CA BA 21 08 (message bits reached, change in every bit)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 99 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 00 00 00 B1 CA BA 21 08 (message bits reached, change in every bit)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 69 DB ED 0E 1B (message bits reached, small change, new value)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 00 00 00 31 CA BA 21 08 (message bits reached, change in every bit)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 A9 DB ED 0E 1B (message bits reached, small change, new value)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 00 00 00 11 CA BA 21 08 (message bits reached, change in every bit)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 00 00 00 01 CA BA 21 08 (message bits reached, change in every bit)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B1 DB ED 0E 1B (message bits reached, small change, new value)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A6 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B5 DB ED 0E 1B (message bits reached, small change, new value)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B7 DB ED 0E 1B (message bits reached, small change, new value)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 00 00 00 00 CA BA 21 08 (message bits reached, change in every bit)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 00 00 00 00 4A BA 21 08 (message bits reached, change in every bit)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 00 00 00 00 0A BA 21 08 (message bits reached, change in every bit)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B7 FB ED 0E 1B (message bits reached, small change, new value)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 18 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 0B ED 0E 1B (message bits reached, small change, new value)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 00 00 00 00 02 BA 21 08 (message bits reached, change in every bit)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 0F ED 0E 1B (message bits reached, small change, new value)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1B 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 00 00 00 00 00 BA 21 08 (message bits reached, change in every bit)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 10 ED 0E 1B (message bits reached, small change, new value)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 00 00 00 00 00 3A 21 08 (message bits reached, change in every bit)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 2D 0E 1B (message bits reached, small change, new value)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 00 00 00 00 00 1A 21 08 (message bits reached, change in every bit)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 00 00 00 00 00 0A 21 08 (message bits reached, change in every bit)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 00 00 00 00 00 02 21 08 (message bits reached, change in every bit)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 31 0E 1B (message bits reached, small change, new value)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A3 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 00 00 00 00 00 00 21 08 (message bits reached, change in every bit)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 0E 1B (message bits reached, small change, new value)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 8E 1B (message bits reached, small change, new value)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 CE 1B (message bits reached, small change, new value)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 00 00 00 00 00 00 01 08 (message bits reached, change in every bit)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 E8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 DE 1B (message bits reached, small change, new value)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EC 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 E6 1B (message bits reached, small change, new value)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EE 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 EA 1B (message bits reached, small change, new value)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 EC 1B (message bits reached, small change, new value)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 00 00 00 00 00 00 00 08 (message bits reached, change in every bit)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 EC 9B (message bits reached, small change, new value)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 60 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 EC DB (message bits reached, small change, new value)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 70 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 EC FB (message bits reached, small change, new value)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 78 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 0B (message bits reached, small change, new value)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00 (message bits reached, change in every bit)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 0F (message bits reached, small change, new value)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 11 (message bits reached, small change, new value)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 12 (message bits reached, small change, new value)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00 (message bits reached, change in every bit)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13 (message bits reached, small change, new value)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00 (message bits reached, change in every bit)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00 (message bits reached, change in every bit)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00 (message bits reached, change in every bit)

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A2 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 60 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 50 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 58 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 5C 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 5A 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 00 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 80 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 C0 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 E0 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F0 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F8 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F4 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F2 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 00 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 80 00 00 00 00 00 00 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 40 00 00 00 00 00 00 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 20 00 00 00 00 00 00 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 10 00 00 00 00 00 00 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 08 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0C 00 00 00 00 00 00 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0A 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 00 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 80 00 00 00 00 00 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 40 00 00 00 00 00 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 20 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 30 00 00 00 00 00 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 28 00 00 00 00 00 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 24 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 00 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 27 00 00 00 00 00 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 80 00 00 00 00 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 40 00 00 00 00 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 20 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 30 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 38 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 00 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3E 00 00 00 00 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3D 00 00 00 00 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 80 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C C0 00 00 00 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C A0 00 00 00 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 90 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 98 00 00 00 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 94 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 96 00 00 00 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 00 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 80 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C0 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 E0 00 00 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 D0 00 00 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 00 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 CC 00 00 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 CA 00 00 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C9 00 00 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 80 00 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 40 00 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 20 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 30 00 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 28 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2C 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2E 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 00 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 80 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 40 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 20 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 10 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 18 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1C 00 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1A 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 00 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 80 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 40 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 20 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 10 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 08 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 04 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 06 00 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 00 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 80 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 40 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 20 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 10 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 08 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 04 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 02 00 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 03 00 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 02 80 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 02 40 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 02 20 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 02 10 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 02 18 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 02 14 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 02 12 00 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 02 13 00 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 02 12 80 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 02 12 40 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 02 12 60 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 02 12 70 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 02 12 68 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 02 12 6C 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 02 12 6E 00 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 02 12 6D 00 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 02 12 6D 80 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 02 12 6D 40 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 02 12 6D 60 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 02 12 6D 70 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 02 12 6D 68 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 02 12 6D 64 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 02 12 6D 62 00

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 02 12 6D 61 00

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 02 12 6D 61 80

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 02 12 6D 61 40

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 02 12 6D 61 20

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 02 12 6D 61 30

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 02 12 6D 61 38

A: 2D 45 C4 B8 11 32 ED 13

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 02 12 6D 61 3C

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 02 12 6D 61 3A

A: 21 00 00 00 00 00 00 00

Q: 00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B A3 59 F1 0B 26 3C 95 C8 2F 1B 05 02 12 6D 61 39

A: 21 00 00 00 00 00 00 00

Therefore, the final value for ciphertext $c1$ ends with the following bytes:

$c1 =$ ...00 C1 6B A6 B5 41 70 59 9D 27 98 A7 1A A2 EF 7B
A3 59 F1 0B 26 3C 95 C8 2F 1B 05 02 12 6D 61 38

Indeed, if we add 1 to this ciphertext, we get a multiple of the key p.

Now, we must repeat the same process with ciphertext $c2$. As the process is analogous to ciphertext $c1$, we only present the initial decryption d = *0x1099BBD185EE6CBC* and the final ciphertext $c2$. The final value for ciphertext $c2$ ends with the following bytes:

$$c2 = \ldots 11\ 2A\ 3F\ C1\ 8A\ 32\ EA\ 52\ 0E\ 56\ 04\ 73\ 45\ A2\ 40\ 2B$$
$$70\ 98\ 61\ 0D\ BD\ 2D\ 0C\ DD\ 62\ 9C\ A9\ D6\ FD\ AC\ 40\ 2F$$

Similar to $c1$, by adding 1 to $c2$ we get a multiple of key p.

Finally, computing $\gcd(c1 + 1, c2 + 1)$, we obtain:

$$\gcd(c1 + 1, c2 + 1) = \texttt{0x03482A6BC4304B9816A2E25C08997689}$$
$$\texttt{8F72E333B7C9B16AA3E6BA90FBD95FCB}$$

This is exactly key p.