KU LEUVEN

FACULTY OF
ENGINEERING SCIENCE

PIRANA implementation using GBFVtest

Ing. Antoine Janssens van der Maelen

Thesis submitted for the degree of
Master of Science in Cybersecurity

Supervisor
Prof. Dr. Ir. F. Vercauteren

Assessor
placeholder

Assistant-supervisors
Dr. Ir. R. Geelen
Dr. J. Kang
Ir. J. Spiessens

Academic year 2025 – 2026

Preface

I would like to thank XXX.

Ing. Antoine Janssens van der Maelen

Contents

Abstract

The `abstract` environment contains a more extensive overview of the work. But it should be limited to one page.

List of Figures and Tables

List of Figures

List of Tables

List of Abbreviations and Symbols

Abbreviations

HE	Homomorphic encryption
PHE	Partially homomorphic encryption
SHE	Somewhat homomorphic encryption
FHE	Fully homomorphic encryption
RLWE	Ring learning with errors
LWE	Learning with errors
BFV	Brakerski-Fan-Vercauteren scheme
CLPX	Chen-Laine-Player-Xia
GBFV	Generalized Brakerski-Fan-Vercauteren scheme
SIMD	Single instruction multiple data
PIR	Private information retrieval
NTT	Number theoretic transform

Symbols

$\Phi_m(x)$	m-th cyclotomic polynomial (degree $\varphi(m)$)
$\varphi(m)$	Euler's totient function
ω_m	Primitive m-th root of unity
$\mathbb{Z}_m^\times$	Units modulo m (indices in Φ_m product)
$\mathcal{R}$	Cyclotomic ring $\mathbb{Z}[x]/(\Phi_m(x))$
$t,\ t(x)$	Plaintext modulus: integer (BFV) or polynomial (CLPX/GBFV)
q	Ciphertext modulus
Δ	Scaling factor q/t (or $q/t(x)$)
m	Codeword length in PIRANA; also cyclotomic index when clear
k	Hamming weight of a codeword (PIRANA)
r	Slots per ciphertext (rows in PIR matrix)
c	Database columns $= n/r$ (PIRANA)
a_i	Uniform LWE/RLWE sample coefficient in $\mathbb{Z}_q$
$s_i,\ s$	Secret key coefficient / polynomial
e	Error term sampled from χ_{err}
m	Message polynomial/plaintext element
$\mathbf{ct}$	Ciphertext pair $(c_0, c_1) \in \mathcal{R}_q^2$
p_{mod}	Prime used to set plaintext modulus in implementations
$\tau(x)$	Factor of $t(x)$ used to define SIMD slot decomposition
b	LWE second component $b = \mathbf{a} \cdot \mathbf{s} + e + \Delta m$

Chapter 1

Introduction

1.1 Privacy-enhancing/preserving technologies

In a world where AI becomes data-driven and in a world where people are getting more aware of how sensitive their data can be, protection of data is crucial. Some legislation sets up a framework to protect data, such as the General Data Protection Regulation (GDPR) for personal data. This requires companies and individuals to implement data protection technologies to ensure data privacy, security and compliance to the legislation.

So far, encryption technologies are shown to be effective to protect data at rest and in transit. However, when data is used for computation, preserving privacy becomes more complex. To achieve this, several privacy-enhancing or privacy-preserving technologies (PETs) are available. PETs are defined by the European Union Agency for Cybersecurity (ENISA) as "a coherent system of information and communication technology (ICT) measures that protect privacy by eliminating or reducing personal data or by preventing unnecessary and/or undesired processing of personal data, all without losing the functionality of the information system" [15].

The least secure way to handle data used in computation is to compute in clear, on an unprotected device, since sensitive data can be seen by the computing party. If this party is malicious or if the data gets leaked, a malicious party could exploit it. To mitigate this risk, several different PETs can be used.

1.1.1 Confidential computing

A computer is built on different layers, going from the hardware layer at the bottom to the application layer on top. Traditionally privacy technologies aimed to defend the bottom layers, such as the operating system. However, with the widespread adoption of hosting and cloud services, there is a switch to protecting the application from the lower layers. In a normal use case, the application has to trust the layers underneath, such as the OS kernel and hypervisor layer. If an attacker or a bug is dissimilated in those lower layers, it is possible to attack the application. Therefore, enclaves are used in confidential computing, creating a trusted execution environment (TEE). Memory inside an enclave is encrypted and decrypted on the fly, inside the isolated enclave, only code running in the enclave is allowed to access the data. So, by essentially only trusting the enclave and the CPU, data in use is protected from the outside of the enclave. Even privileged software, for instance the operating system, cannot access the memory directly. The susceptibility to side-channel attacks limits the use of enclaves; for example information can be revealed from the way the TEE interacts with other parts of the system, thus potentially revealing what is happening inside the enclave [35]. A second limitation when using confidential computing is the fact one is trusting the cloud service provider or the specific computer system to properly set up and use the enclave, but this party can be untrustworthy. To mitigate this risk, attestation mechanisms can be employed to verify the trustworthiness of the provider or system.

1.1.2 Federated computation

Another way to protect data while being processed is to perform federated computation. Firstly, a local computation is performed which will hide information about the local database. Secondly, the different parties who did the local computation share their results with a central co¨ordinator. A second computation is performed on all data collected from the different parties, yielding the desired final output, without any party having to reveal its raw data.

This approach however has some limitations. Federated computation is a relatively easy and efficient way to operate, but can become challenging when one wants to perform non-linear operations on the data that require intermediate sharing[1] (e.g. high-order moments). An attacker can retrieve some information from the parties, as the first computation result (local computation by each party) is sent in clear to the co¨ordinator. Also, an attacker can use a combination of engineered queries to get some information about a certain subgroup of data or users (differential attack). To prevent this, one can add noise to the result of the local computation, to obtain differential privacy. However, adding too much noise will make the data useless. Also, outliers will increase the noise significantly which can detoriate the accuracy of the results. So, setting the right noise parameters is tricky and requires a certain amount of expertise.

1.1.3 Multi-party computation

Multi-party computation uses multiple computing parties, with no authority party. There is no computation on the original data, but data is first split up into shares and these shares are then distributed between the computing parties. These parties then perform computations using the shares they possess. The results of these computations are then combined together to find the result of the desired computation, on the original data.

When one wants to compute non-linear functions, however, MPC becomes more challenging. To perform the computation there is a need for interaction between the parties, who need to exchange their shares between each other. This leaks

[1]Exchange of partial results or intermediate results during multi-step computations.

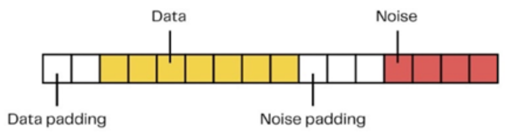

FIGURE 1.1: Ciphertext HE [12]

information about the data. To solve this, for example, two random numbers can be chosen. These random numbers are used to mask the share, while reconstructing of the answer remains possible [14]. A risk of MPC is collusion. If some of the parties collude, secret sharing amongst them can make it possible to reconstruct the original data. Additionally, MPC can bring along high communication overhead, leading to scalability issues.

1.1.4 Homomorphic encryption

Homomorphic encryption allows computation on encrypted data without decrypting. No party performing computations has access to the plaintext data, these data remain encrypted. Partially homomorphic encryption (PHE) is a type of HE that only supports homomorphic multiplication or addition, but not both. Full HE (FHE) and leveled HE (LHE) support both multiplications and additions, but with LHE only up to a limited computation depth. To enhance security, noise is added to the encrypted data when using HE, in the least significant bits as illustrated in Figure 1.1. However, when performing computations the noise can grow beyond the noise padding bits, eventually corrupting the data in LHE. To mitigate this, bootstrapping can be performed in FHE to reduce the amount of noise, thus allowing more computations to be done whilst maintaining data correctness.

FHE is however computationally intensive (thus slow) for large and unstructured data. It also requires specialized expertise to implement. To address this, some organisations are supporting adoption (e.g. Google has released an open source compiler for FHE) [30].

1.1.5 Comparison of PETs

As already mentioned, each PET has its strengths and flaws. Also, the suitability of a PET depends on the envisaged application.

With the rise of cloud computing, users realise their data in the cloud is at risk. As a result computing in clear, without any protection or encryption, will phase out. Federated computation seems useful when data is naturally distributed, for instance when training an AI model with data collected from different user devices. An important vulnerability however is the fact that data privacy is not guaranteed by itself, the exchanged model parameters can still leak information about the underlying data. Therefore, federated computation should be combined with other PETs to enhance the security of the underlying data.

Multi-party computation is a good option to maintain privacy of information while computing, but inherently needs multiple devices each having a different authority. In MPC, sensitive information can be restored when parties collude. Although this risk can be lowered when devices are distributed under different authorities, this makes the implementation of MPC challenging, as finding nodes that will never collude is a difficult (impossible?) task. Therefore, there is still some risk the sensitive information leaks.

Confidential computing is still evolving and manufacturers are using different approaches to implement enclaves, this technique offers a high level of protection by keeping data and code secure in an enclave. The enclave is made on a single machine, there is no need for multiple devices (in contrast to MPC, federated computation). Confidential computing already found ground in multiple applications, such as Nitro Enclaves at AWS and Intel SGX CPU's.

FHE offers strong advantages when compared to other PETs. If an implementation of FHE can be proven, we can be (mathematically) sure the data can not be decrypted while being processed. Also, less communication is needed during computation when compared to MPC (multi-party computation) and it has a better track record in terms of security vulnerability when compared to TEE (trusted execution environment). [22]

On the other hand, there are some disadvantages too. Like some other PETS, FHE requires specialized expertise to implement. But, most importantly, FHE is computationally intensive (thus slow) for large and unstructured data. According to Ulf Mattsson, general FHE processing is 1,000 to 1,000,000 times slower than equivalent plaintext operations. [29].

Enhancing the speed of FHE is an attractive research topic, as it would make FHE more suitable for real-world applications. Therefore, in this thesis, we focus on improving the performance of a private information retrieval (PIR) scheme based on FHE.

However, seeing every PET independent would be a mistake. For instance, when creating a FHE blockchain network, there is a need for one key for the whole network. Who holds the decryption key will define the security level, given it to one party is insecure. MPC could be used as a means to distribute the key to all nodes of the blockchain, thus making the blockchain network more secure. Thus, combining PETs could enhance security for certain applications[2].

1.2 Thesis outline

This thesis aims to implement the PIRANA-protocol[3] using a GBFV[4]scheme.

The thesis begins by presenting the theoretical background required for this work. First, the different generations of homomorphic encryption are introduced. Since

[2]The blockchain is made (very) secure by FHE, but by distributing the key using MPC, the security is shifted to the MPC.

[3]The PIRANA-protocol is a private information retrieval protocol.

[4]GBFV is a FHE-scheme

the GBFV scheme relies on the hardness of the Ring Learning With Errors (RLWE) problem, cyclotomic rings, LWE, and RLWE are subsequently discussed. The GBFV scheme and its scheme functions are then described in detail. As GBFV supports single-instruction multiple-data (SIMD) operations, SIMD is also addressed.

Next, private information retrieval is introduced, with a distinction between single-server and multi-server PIR schemes. In Section 2.7, the PIRANA protocol is described for two single-query implementations: PIRANA for small payloads and PIRANA for large payloads. Finally, the Fheanor library, a Rust-based homomorphic encryption framework on which the easyGBFV implementation is built, is presented.

Chapter 3 describes the implementation of GBFV-PIRANA for both single-query small and large payloads. In addition, a GBFV-based approach using one-hot encoding is introduced.

The experimental results are presented and analyzed in the subsequent chapter. These results are then discussed, and the thesis concludes with a summary of the findings and conclusions drawn from this research.

Chapter 2

Theoretical background

2.1 Homomorphic encryption

Homomorphic encryption (HE) allows computations to be performed on encrypted data without needing to decrypt. As discussed above, there are multiple types of HE: PHE which only supports multiplication or addition and LHE/FHE which supports both addition and multiplication.

Whilst performing a large number of (complex) operations, noise added to the HE ciphertext will grow and overwrite data. To avoid this, two methods are used: using big integers as ciphertext modulus and bootstrapping. By using big integers as ciphertext modulus, enough space is provided for the noise to grow for the full computation - the so-called leveled schemes (LHE)[1]. To have more computation depth possible (FHE), bootstrapping operations are performed to reduce the amount of noise in between chains of computations. One should note that bootstrapping is computationally and memory-intensive.

FHE schemes can be divided into multiple generations, depending on the type of bootstrapping techniques [18]. First generation schemes include schemes like the Gen09 bootstrapping technique, described in 2009, which is illustrated in Figure 2.1 [12]. FHE-encrypted data are FHE-encrypted a second time, with a lower level of noise compared to the initial encryption. Subsequently a bootstrapping key is sent to the computing node, which is the secret key of the initial encryption, encrypted with the public key of the second encryption. Decryption with this bootstrapping key removes the first encryption layer, and one ends up with data solely encrypted via the second FHE-scheme, with a lower level of noise. This type of scheme is no longer used in practical implementations.

Second generation schemes are defined by having a slow and complex bootstrapping. However, bootstrapping cost is compensated by the use of SIMD (Single Instruction Multiple Data) operations, which will distribute this cost over many slot, so many parallel computations. Examples are BGV, BFV and CKKS. Third generation schemes are characterized by a very simple and fast bootstrapping procedure.

[1]BFV and CKKS are often implemented without bootstrapping, as a leveled scheme, but are bootstrappable.

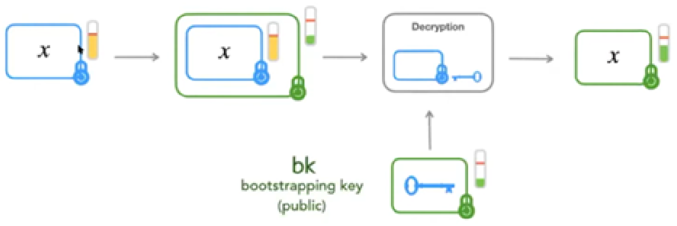

FIGURE 2.1: Gen09 bootstrapping [12]

These exhibit lower circuit complexity, faster execution times, and less noise growth when compared to the second generation schemes. On the other hand, they will not offer SIMD slots for parallel processing. Examples include torus FHE (TFHE).

Next to these generations, some leveled homomorphic encryption schemes are known without any efficient bootstrapping technique. The CLPX scheme from Chen et al. [10] is an example, where the parameters of the scheme are set as such level to allow deep circuit evaluation before noise corrupts the result.

2.2 Cyclotomic rings and (R)LWE

2.2.1 Cyclotomic rings

BFV (Brakerski-Fan-Vercauteren) is built on the RLWE problem (ring learning with errors), which is a hardness problem used in cryptography. We define the m-th cyclotomic polynomial as follows:

$$\Phi_m(x) = \prod_{j \in \mathbb{Z}_m^\times} (x - \omega_m^j)$$

- $w_m \in \mathbb{C}$ is a primitive m-th root of unity, where $m \geq 1$
- $\mathbb{Z}_m^\times$ is the unit group of integer modulo m.

The degree of the cyclotomic polynomial is equal to $\varphi(m)$, the result of Euler's totient function of m. Although the cyclotomic polynomial has complex roots, it has been proven that the coefficients are integer numbers and the polynomials are monic (leading coefficient is 1) and irreducible[3]. The RLWE problem is then defined over the ring $\mathbb{R} = \mathbb{Z}[x]/(\Phi_m(x))$. This ring is a subring of the cyclotomic number field $\mathbb{Q}[x]/(\Phi_m(x))$.

2.2.2 LWE

The ciphertext is constituted of two parts: uniform random numbers a_i and b, where b is the sum of the multiplication of the uniform random numbers a_i with the secret key s_i, some Gaussian distributed noise e and the message m, normalized by a delta coefficient. The corresponding ciphertext can be represented on a ring, all the possible values of m are put on a ring, at a spacing Δ from each other. To decrypt, the decryption formula (2.2) is used, which is equivalent to rounding the value on the ring (which corresponds to Δ times the message plus the error) to the closest possible value for m. If the error becomes too large, the value will round to the wrong message value, so returning a faulty message.

$$\text{Encryption:} \quad ct = (a_0, \ldots, a_{n-1}, b) \quad \text{where} \quad b = \sum_{i=0}^{n-1} a_i s_i + e + \Delta m \tag{2.1}$$

$$\text{Decryption:} \quad m \approx \frac{b - \mathbf{a} \cdot \mathbf{s}}{\Delta} \tag{2.2}$$

$a_i \in \mathbb{Z}_q$ are chosen uniformly at random
$s_i \in \mathbb{Z}_q$ are the secret key coefficients
$e \in \mathbb{Z}_q$ is a small error term (typically Gaussian)
m is the message encoded in the ring
Δ is the message scaling factor
$b \in \mathbb{Z}_q$ is the second component of the ciphertext

This scheme already allows to do some operations on the ciphertext: we can add two ciphertexts and perform multiplications of the ciphertext with non-encrypted integers[2].

2.2.3 RLWE

Ring LWE is similar to LWE but the message, secret key, random values and error are all polynomials in the cyclotomic ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(f(x))$ rather than vectors over integers. Ciphertexts are constructed using polynomial arithmetic modulo a cyclotomic polynomial. When encrypting, we will normalize the message and add a Gaussian error, like in LWE. For every coefficient of the polynomial, the value of $\Delta \ s$ plus the error will again be represented on a ring. Rounding will give the polynomial coefficients of the message m. The RLWE scheme allows to perform additions between ciphertexts and multiplication with non-encrypted constant polynomial functions.

The RLWE distribution consists of pairs $(a, b) \in \mathcal{R}_q \times \mathcal{R}_q$, where a is chosen uniformly at random from $\mathcal{R}_q$ and $b = a \cdot s + e$, with e and s sampled at random over $\mathcal{R}_q$.

There exist two variants of the RLWE-problem, search RLWE and decision RLWE.

Search RLWE: *Let $\mathcal{R}$ be an RLWE instance. The search RLWE problem, denoted by SRLWE($\mathcal{R}$), is to discover s given access to arbitrarily many independent samples (a, b).*

Decision RLWE: *Let $\mathcal{R}$ be an RLWE instance. The decision RLWE problem, denoted by DRLWE($\mathcal{R}$), is to distinguish between the same number of independent*

[2]An operation on ciphertexts will, in FHE, correspond to an operation on plaintexts.

samples in two distributions on $\mathcal{R}_q \times \mathcal{R}_q$. The first is the RLWE distribution of $\mathcal{R}$, and the second consists of uniformly random and independent samples from $\mathcal{R}_q \times \mathcal{R}_q$.

Both variants are supposed to be hard[4]. Note that the search RLWE problem can be reduced to the decision RLWE problem [11].

2.3 BFV and CLPX

In BFV by Kim et al.[24, 7, 16], a subquotientring $\mathcal{R}_t$ of $\mathcal{R}$ is created by taking modulo t the ring $\mathcal{R}$. In the case of BFV, we fix this t to a prime integer p. The ciphertext also has a modulus q and the message is scaled by a factor $\delta = q/t$. The plaintext space corresponds to $R_t = \mathbb{Z}[x]/(\Phi_m(x), p)$. Encryption is then done via following formula:

$$\text{Ciphertext:} \quad \mathbf{ct} = \left(\left[\lfloor \Delta \cdot m \rceil + \mathbf{a} \cdot \mathbf{s} + e\right]_q, -\mathbf{a}\right) \tag{2.3}$$

$$\text{Decryption:} \quad m = \left\lfloor \frac{c_0 + c_1 \cdot \mathbf{s}}{\Delta} \right\rceil \tag{2.4}$$

The scheme can be implemented as a leveled scheme or can be bootstrapped to a fully homomorphic encryption scheme. In BFV, one can perform addition, multiplication and automorphism over the plaintext space.

In CLPX, the idea is to use a plaintext ring modulo t, with $t = x - b$ instead of an integer p, as in BFV. The plaintext space is now defined as

$$\mathcal{R}_t = \mathbb{Z}[x]/(\Phi_m(x), x - b) = \mathbb{Z}[x]/(x - b, p) \cong \mathbb{Z}_p \text{while} p = \Phi_m(b) \tag{2.5}$$

In this CLPX-scheme, m is a k-th power of 2. When encrypting, a hat encoding is performed first on the message m, by taking the modulus quotient ring of R modulo t. We get $\hat{m}$ which only has small coefficients. Encryption is done as follows:

$$\mathbf{c} = \left(\left[\Delta \cdot \hat{m} + \mathbf{a} \cdot \mathbf{s} + e\right]_q, -\mathbf{a}\right) \tag{2.6}$$

Decryption is performed using the secret key $\mathbf{s}$:

$$\hat{m} = \left\lfloor \frac{t}{q} \cdot \left(c_0 + c_1 \cdot \mathbf{s}\right) \right\rceil \tag{2.7}$$

In CLPX, addition and multiplication can be performed homomorphically [10].

CLPX can encrypt a single huge integer modulo $\Phi_m(b)$ and has much lower noise growth when compared to BFV - the growth is sublinear in the desired precision, it depends on b instead of $\Phi_m(b)$ [19]. This makes CLPX suitable for high-precision arithmetic HE operations. However, in CLPX Only a single element is encrypted, so no SIMD operations can be performed. Also, since the size of p is exponential in m, bootstrapping is made challenging. However, a recent work by Kim demonstrated [25].

2.4 GBFV

CLPX has much lower noise growth when compared to BFV, but does not support SIMD operations and is not known to be efficiently bootstrappable for cryptographically secure parameters. Geelen and Vercauteren propose the GBFV scheme which combines the SIMD and bootstrapping capabilities of BFV with the lower noise growth of CLPX, by tuning the parameters m and $t(x)$. Combining both properties would either yield a scheme capable of evaluating deeper circuits or would yield a scheme capable of working with smaller ring dimensions. While BFV allows for slots, it always creates N slots, with N the degree of the cyclotomic polynomial. GBFV allows to create a number of slots which is a divisor of N, so one can create less slots than N, but with bigger plaintext modulus, which can be useful in some applications.

GBFV operates over the cyclotomic ring $\mathcal{R} = \mathbb{Z}[x]/\phi_m(x)$. The plaintext space is defined modulo an arbitrary non-zero principal ideal generated by a polynomial $t = t(x)$. This ring is $R_t = R/tR$. A plaintext $m \in \mathcal{R}_{\sqcup}$ is encrypted into a ciphertext $ct \in \mathcal{R}_q^2$ with RLWE:

$$ct = \left(\left[\lfloor \Delta \cdot m \rceil + a \cdot s + e \right]_q, \ -a \right)^{3} \tag{2.8}$$

The ciphertext space will be a ring $\mathcal{R}_q^2$ with $q \geq 2$. The scaling factor Δ is defined by q/t, with q the ciphertext modulus. This scaling factor is not rounded to $\mathcal{R}$, resulting in a conceptually simpler scheme definition when compared to BFV and CLPX.

For correct decryption, the canonical infinity norm of the plaintext modulus must be much smaller than the ciphertext modulus. This ensures that the decryption correctly recovers plaintexts without modular wrap-around or rounding errors, all contributions from $t(x)$ (i.e. $t(x) \cdot m(x)$) and the noise must be much smaller than q.

2.4.1 Scheme functions

The GBFV scheme has several functions which it can perform:

- Secret key generation: Samples a secret key s from a key distributon χ_{key}, $s \in \mathcal{R}$, returns s.
- Relinearization key: after multiplication of ciphertexts, one gets a hig

  This thesis aims to implement the PIRANA protocol using the GBFV fully homomorphic encryption scheme.

  The thesis begins by presenting the theoretical background required for this work. First, the different types of homomorphic encryption are introduced. Since the GBFV scheme relies on the hardness of the Ring Learning With Errors (RLWE) problem, cyclotomic rings, LWE, and RLWE are subsequently

[3] $\lfloor x \rceil$ is rounding to the nearest integer

discussed. The GBFV scheme and its core functions are then described in detail. As GBFV supports single-instruction multiple-data (SIMD) operations, SIMD techniques are also addressed.

Next, private information retrieval is introduced, with a distinction between single-server and multi-server PIR schemes. In Section 2.7, the PIRANA protocol is described, including two specific single-query implementations: PIRANA for small payloads and PIRANA for large payloads. Finally, the Fheanor library, a Rust-based homomorphic encryption framework on which the easyGBFV implementation is built, is presented.

Chapter 3 describes the implementation of GBFV-PIRANA for both single-query small and large payloads. In addition, a GBFV-based approach using one-hot encoding is introduced.

The experimental results are presented and analyzed in the subsequent chapter. These results are then discussed, and the thesis concludes with a summary of the findings and conclusions drawn from this research. her order polynomial in s, which can not be decrypted since the scheme only knows s and not s to a higher power. The relinearization key approximates the ciphertext back to a linear equation in s. Returns the relinearisation key.

- Decryption: A ciphertext is decrypted using $m = \lfloor \frac{c_0 + c_1 \cdot s}{\Delta} \rceil$. Returns m.

GBFV supports standard homomorphic operations on ciphertexts, using following functions:

- Ciphertext-ciphertext addition: ciphertext addition is done component-wise modulo q and returns ct_{add}.

- Plaintext-ciphertext addition: the plaintext is encrypted as follows:
$$ct' = \left([\lfloor \Delta \cdot m \rceil]_q, \, 0 \right)$$
After this stage, add the original ciphertext with ct' (ciphertext-ciphertext addition).

- Key switching: reduces the result of the ciphertext-ciphertext multiplication back to two components.

- Ciphertext-ciphertext multiplication: two ciphertexts $ct = (c_0, c_1)$ and $ct' = (c'_0, c'_1)$ are multiplied as follows:
$$\mathbf{c}'' = \left([\lfloor \frac{c_0 \cdot c'_0}{\Delta} \rceil]_q, \, [\lfloor \frac{c_0 \cdot c'_1 + c_1 \cdot c'_0}{\Delta} \rceil]_q \right), \tag{2.9}$$
$$c''_2 = \left[\lfloor \frac{c_1 \cdot c'_1}{\Delta} \rceil \right]_q \tag{2.10}$$
Since the c''_2 contains a second-order term in s, a relinearization is performed using the relinearization key, creating ct'''. This ciphertext is then added to ct''.

- Ciphertext-plaintext multiplication: takes the ciphertext and multiplies both parts with the flattened message m. Flattening involves reducing the coefficients from, ensuring the coefficients are reduced modulo t but expressed in $\mathcal{R}$. Following formula is used:

$$\text{Flatten} : \mathcal{R}_t \to \mathcal{R} : \quad m \mapsto t \cdot \left[\frac{m}{t}\right]_1$$

- Automorphism. Applying an automorphism to the ciphertext polynomials, this permutes slots in packed plaintexts after decryption. Then, the plaintext moduli are corrected if they changed under the automorphism. A key switching is performed to bring back the secret key to s. Finally, the adjusted ciphertexts are combined to form the final output.

2.5 SIMD

Smart and Vercauteren [34] noticed that it was possible to encode multiple elements in one plaintext/ciphertext, using the Chinese Remainder Theorem. This splitting in slots can allow SIMD operation - performing a single operation on multiple data.

SIMD allows to perform following operations such as:

- Addition: component-wise addition of slots.
- Multiplication: component-wise multiplication of slots.
- Slot rotations: shifting slots in the plaintext/ciphertext. The shifting is performed different in BFV and GBFV. In BFV, the slots are usually structured in a 2D-hypercube, so rotations can be performed along the two dimensions. In GBFV, the slots are structured as a 1D-array, so rotation is a slot-shift along the axis.

BFV can support packing in slots and thus SIMD operations. However, doing this puts a restriction on the use of BFV. If $p = \Phi_m(b)$ is the modulus of the plaintext, the upper bound of the output noise will grow proportional to the product of this factor and the sum of the upper bounds on the input noise. When one wants to have a high precision arithmetic, one chooses a high p, which will result in more output noise. This makes BFV SIMD-schemes impractical when performing precise arithmetic calculations, often needed in HE-applications. For instance, privacy-preserving machine learning [21] uses moduli up to 80 bits. Also, a higher value of p enables a higher packing density [19]. The packing density, which is equal to the number of slots divided by the ring dimension, is equal to $1/d$. d is the multiplicative order of p modulo the cyclotomix index m. To achieve full packing, p needs to be larger than m. When using power-of-two cyclotomics, the number of slots will be upper bounded by $(p{+}1)/2$ [17]. To conclude, while a large value of p allows for greater packing density and more precise arithmetic operations, it simultaneously exacerbates noise growth.

GBFV works modulo a plaintext modulus polynomial $t(x)$, hereby defining a plaintext ring $\mathcal{R}_t = \mathcal{R}/(t(x))$. To enable SIMD computation we want this plaintext space to be isomorphic to a product of fields. By the Chinese Remainder Theorema, this decomposition in fields allows a plaintext to be represented as a number of slots, so SIMD operations are also possible when using GBFV. Homomorphic operations will then be performed component-wise on these slots. This enables parallel computation and rotation of slots can be performed via automorphisms.

2.6 Private information retrieval

When retrieving information from a remote server, the database holder will know which elements are queried if no security measures are implemented. To protect the user, one wants to hide which elements are queried from the server. Private information retrieval or PIR is often considered to achieve this goal.

The goal of PIR is to ensure the server does not learn anything about the index from the user query. This will enhance the privacy of the user, since no information will be leaked to the (remote) server(s), potentially causing serious privacy issues. PIR finds application in multiple scenarios where sensitive data are used. For example, a doctor querying a database with a patient's medical data will get back the requested medical information, without the server learning which patient or which patient record was requested. PIR also finds application in technology. Apple uses PIR to provide caller ID information of an incoming phone call, without them learning who is calling who [28].

PIR protocols can be categorized in two groups: single-server PIR and multi-server PIR. The single-server PIR is the most straightforward setting: one server holds the full dataset, and the client queries the server to get the data of interest. In multi-server PIR, there are multiple servers holding a copy of the full dataset, and the client queries multiple servers to obtain the data of interest. The core idea when using multi-server PIR is that, although the dataset is replicated on multiple servers, the query is split into parts. In this way, none of the servers learn which bit is requested but the requested bit can be recovered from the results of the different servers.

A multi-server PIR scheme is demonstrated in Figure 2.2. Let D be the database with 4 bits $[b_1, b_2, b_3, b_4]$, and the client wants to retrieve bit b_3 privately. There are two non-colluding servers holding a copy of the database. The client will sample a random binary vector q_1 and will make q_2 by XORing q_1 withe a zero vector with a one at the position of the wanted element e_3. Subsequently both queries are sent to both servers. The servers compute a_1 and a_2, and return them to the client. Afther XORing both responses the client can retrieve b_3. Each server sees a random-looking query, so neither learns which index was requested as long as they do not collude.

Security holds as long as the servers do not collude. This assumption is difficult to achieve, because the database is usally under one authority and distributing the database on multiple servers with different authorities is mostly not feasible in practice. Therefore, single-server PIR schemes are often preferred since they rely

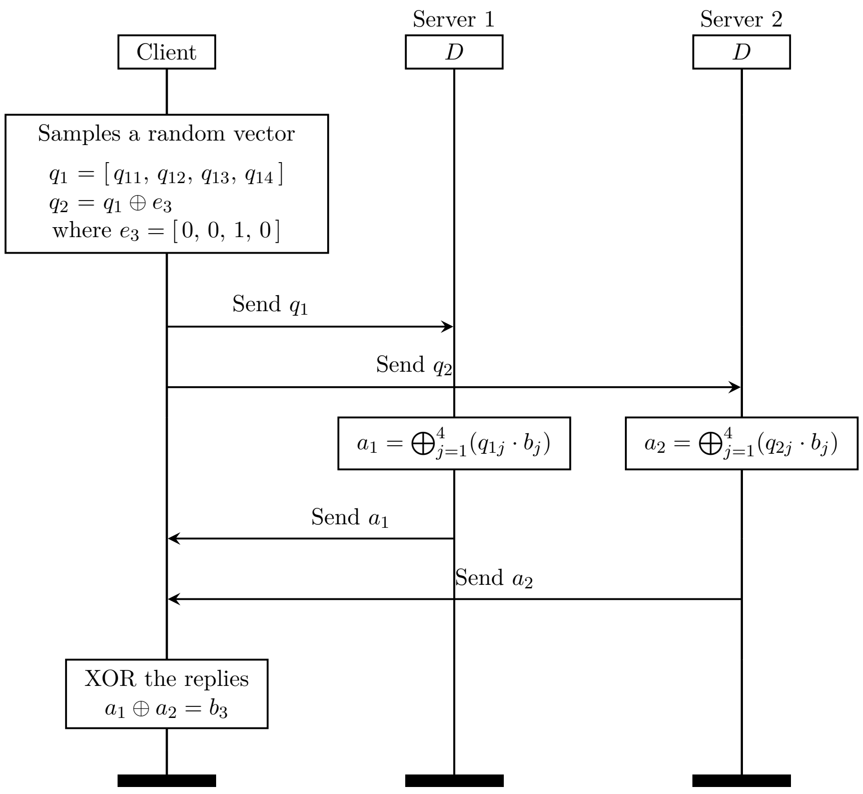
Server 1
Server 2
Client
D
D
Samples a random vector
$q_1 = [q_{11}, q_{12}, q_{13}, q_{14}]$
$q_2 = q_1 \oplus e_3$
where $e_3 = [0, 0, 1, 0]$
Send q_1
Send q_2
$a_1 = \bigoplus_{j=1}^{4}(q_{1j} \cdot b_j)$
$a_2 = \bigoplus_{j=1}^{4}(q_{2j} \cdot b_j)$
Send a_1
Send a_2
XOR the replies
$a_1 \oplus a_2 = b_3$

FIGURE 2.2: Multi-server PIR protocol example

on cryptographic hardness assumptions, but at the cost of incurring a performance overhead. In this thesis, we will further focus on single-server PIR.

A scheme is information-theoretically secure when the queries asked by the user give no information whatsoever about the requested bit to the server. Multi-server PIR schemes can achieve information-theoretic security, such as the PIR-protocol proposed by Ghoshal et al. [20]. Single-server PIR schemes can not achieve information-theoretic security. One exception, when the single server sends the full database to the client. The client can then query the database and the server will not learn anything about the requested bit. However, this trivial scheme has a communication cost of $\mathcal{O}(n)$, with n the size of the database. Single-server PIR schemes with smaller communication cost are only computationally secure; computationally secure schemes only guarantee that the server can not compute the requested bit in a reasonable amount of time, given the queries. Kushilevitz and Ostrovsky made use of the number-theoretic assumption to deduce a single-server computationally secure PIR with subpolynomial communication cost [26]. The scheme has a communication complexity of $\mathcal{O}(n^{\epsilon})$ for any $\epsilon > 0$. This scheme however requires n big integer multiplications. Cachin et al. proposed a two-round computationally secure PIR using the ϕ-hiding assumption where communication complexity is polylogarithmic in n. This scheme requires n modular exponentiations, with large moduli, which makes multiplication slower then in Kuhilevitz's scheme [13]. Chang subsequently proposed a scheme with logarithmic communication complexity, using Paillier's cryptosystem [32, 9]. In 2007, Sion and Carbunar pointed out that these single-server PIR protocols are mostly orders of magnitude slower than the trivial transfer of the entire database to the client [33]. However, later work by Aguilar-Melchor et al. showed that this argument is incorrect: single-server PIR can be faster than downloading the entire database, when using lattice-based cryptographic methods. These methods have smaller per-bit computation cost when used in a batched fashion [1].

More recent PIR protocols make use of fully homomorphic encryption. FHE typically incurs significant communication overhead due to the ciphertext expansion factor. However, keeping the query size as low as possible while maintaining computation cost reasonable is the objective in these protocols. Arranging the database as a hypercube will increase the computation efficiency, as used in Respire [8]. Furthermore, transciphering can be used to further lower the query size. In transciphering, the client will use a symmetric encryption scheme to encrypt the query, which is then homomorphically evaluated on the server side. The server will homomorphically decrypt the query and evaluate it on the database, returning the encrypted result to the client. The client will then decrypt the result symmetrically. This method reduces communication cost since FHE ciphertexts are only used for the query and result, while symmetric encryption is used for the database. Kang proposed a novel transciphering method to further reduce the communication cost when compared to (T-)Respire [6, 5]. In this scheme, the client transmits only one part of the LWE ciphertext. The full LWE ciphertext is reconstructed using a pseudo-random generator seed shared between the client and server. By sending only a single LWE component and a short seed, Pirouette succesfully achieves query compression while keeping computational cost associated with transciphering reasonable [23].

2.7 PIRANA

PIRANA is a single-server protocol developed at Zhejiang University [27]. The protocol is based on constant-weight codes, which is a way to encode the queries. In constant-weight codes, all codewords have a length m and have the same Hamming weight, meaning they have the same number of ones. In later steps, it will be clear how the database is structured and how information is retrieved. There should at least be the same amount of codewords as there are columns in the database. To estimate the length of the codeword in bits, we need to know the number of columns and the Hamming weight k. The number of codewords is equal to the binomial coefficient $\binom{m}{k}$. To estimate the code length m, knowing the Hamming weight k and the number of columns n, we can then use following formula:

$$m \in O\left(\sqrt[k]{k!\,n} + k\right) \tag{2.11}$$

According to the Mahdavi-Kerschbaum mapping method, every index i of a column is mapped to the i-th codeword.

PIRANA can be used in single-query and multi-query set-up. In the following part, single-query PIRANA will be discussed for small and large payloads, as well as a comparison of PIRANA with constant-weight PIR (cwPIR).

2.7.1 Single-query PIRANA for small payloads

In single-query PIRANA for small payloads, the client wants to retrieve a single element from the database. Small payloads means the elements in the database are smaller than the plaintext modulus p. So multiple elements can be packed in one ciphertext. The database of n elements is structured as a 2D-matrix with r rows and t columns, where $n = r \cdot t$ and r is the number of slots in a ciphertext. Every column is represented by one codeword of length m and Hamming weight k. To determine m and k, one wants to find the smallest m such that $\binom{m}{k} \geq t$. Figure 2.3 shows how to retrieve an element in position (i, j) using single-query PIRANA for small payloads. The client constructs a query made out of m ciphertexts $\widetilde{q}_{1\ldots m}$, with r ciphertext slots, thereby constructing a matrix of size $m \times r$. This is a all-zero matrix, except for the i-th row, which contains the codeword corresponding to column j. This query is then sent to the server. The sever will receive this query and, for every column, will take the corresponding codeword. For every bit equal in the codeword, the server will take the corresponding column from the query and perform $k-1$ homomorpic multiplications. The server will do this for all columns, thus $(k-1) \cdot n$ ciphertext-ciphertext multiplications. Hereby creating a selection matrix of t ciphertexts $\widetilde{w}_{1\ldots t}$, where every ciphertext contains r slots. Thus, the selection matrix has a size $r \times t$. All values in these matrix are zero, except for the position (i, j), which contains the one. The server will then perform a ciphertext-plaintext multiplication between the columns of the selection matrix $\widetilde{w}$ and the columns of the database $d_{1\ldots t}$. This can be done because the dimensions of both matrices match. This will return t columns $\widetilde{u}_{1\ldots t}$. Every ciphertext contains all zeros, except the j-th ciphertext, which will contain the payload at slot i. To reduce the amount of ciphertexts sent back to the client, the server will sum up all columns, returning a ciphertext $\widetilde{v}$ with r slots, where all slots are zero, except for slot i, which contains the requested element. This ciphertext is sent back to the client, who will decrypt and get the requested element.

This protocol has some flaws. First of all, if ns is large, the amount of ciphertext-ciphertext multiplications becomes huge. Secondly, the communication cost to retrieve one element is high, since the client needs to send m ciphertexts to the server, and the server will return one ciphertext with r slots. Lastly, when choosing a large k, the amount of ciphertext-ciphertext multiplications increase and m increases too, which will increase the communication cost.

2.7.2 Single-query PIRANA for large payloads

PIRANA can also be used for large payloads, i.e. elements that are bigger than the plaintext modulus p. Every element is split into multiple chunks, each chunk ch

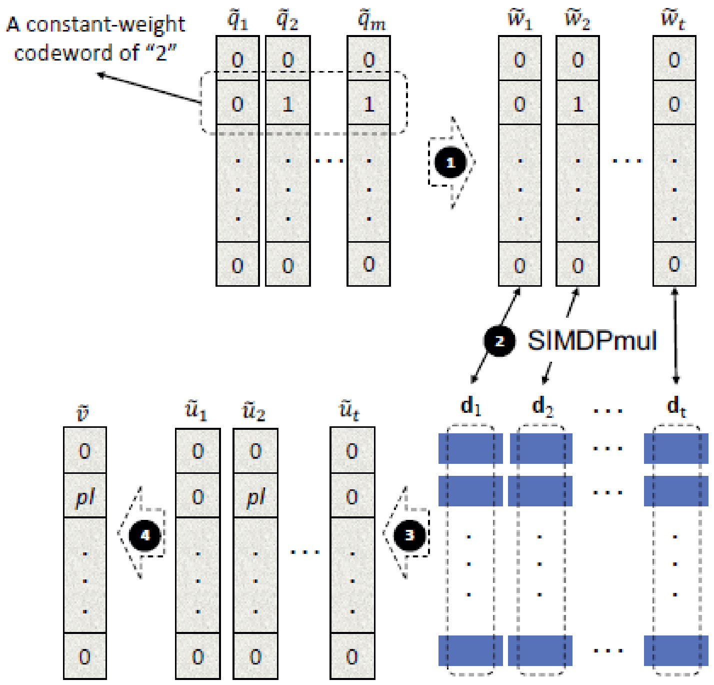

Figure 2.3: Single-query PIRANA for small payloads [27]

smaller than p. The database is now a 3D-matrix of size $r \times t \times \ell$, where ℓ equals the number chunks per element. The selection matrix is created in the same way as for small payloads, resulting in a matrix of size $r \times t$. This selection matrix is now multiplied column-wise with every layer of the database. The result of this multiplication gives a 3D-matrix of size $r \times t \times \ell$. In the same way as for small payloads, every column $\widetilde{u}_{\cdot,n}$ of a layer n is added to create $\widetilde{v}_n$. This is done for every layer, thereby giving $\widetilde{v}_{1 \ldots \ell}$. To reduce the amount of $\widetilde{v}$ ciphertexts sent back to the client, rotate and sum is performed as displayed in Figure 2.4. Every column $\widetilde{v}$ is a ciphertext with all zeros, except at position i. These positions contain one chunk ch_n of the requested element. An accumulator acc is initialized with the first column/ciphertext, rotated by one position. Subsequently, the second column is added to this accumulator. The accumulator is then rotated by one position again, and the third column is added to the accumulator. This is repeated until all columns are added. If there are more chunks than slots in a ciphertext, ℓ/r ciphertexts are needed to retrieve the remaining chunks. By performing the rotate-and-sum operation, the response query goes from ℓ to ℓ/r ciphertexts. The final ciphertexts are sent back to the client, which will decrypt and can reconstruct the requested element, by combining the chunks.

For small database size n, one could pre-compute the rotations in the set-up time of the database. This will reduce the computation time when performing a query to the database.

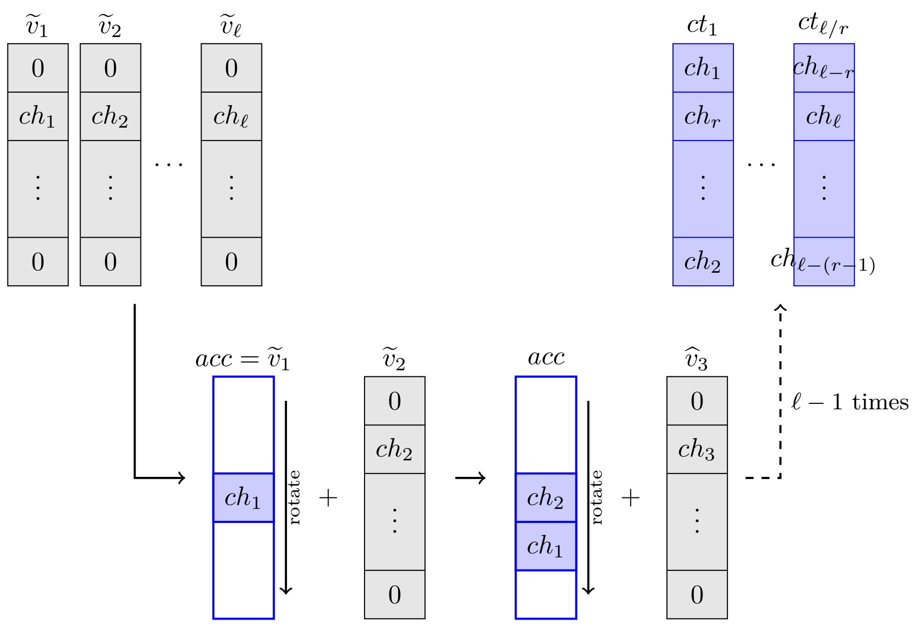

FIGURE 2.4: Rotate-and-sum operation

2.7.3 PIRANA performance comparison

Liu et al. compared the performance of PIRANA with constant-weight PIR in table 2.1. PIRANA was implemented in C++ based on Microsoft SEAL HE library [4], and the BFV scheme was used with $N \in \{4096, 8192\}$. Tests are performed on an Intel Xeon Cooper Lake with a base frequency of 3.4 GHz and turbo frequency of 3.8 GHz. The server was running on Ubuntu 20.04. This set-up is done similar to the set-up of cwPIR [2]. PIRANA outperforms cwPIR in terms of selection vector generation time. As expected: in PIRANA $c \cdot (k-1)$ ciphertext-ciphertext multiplications are needed, while in cwPIR $n \cdot (k-1)$ multiplications are needed. When the database size n increases, the difference in performance becomes larger. Inner product calculation is also faster in PIRANA when compared to cwPIR. In cwPIR, every ciphertext needs to be transformed using NTT (number theoretic transform) before multiplying with the database. In PIRANA, there are only m ciphertexts to transform, which is r times smaller than n. The query size of PIRANA is up to 2.5 times larger when compared to cwPIR, and the response size is equivalent. Thus, commmunication cost is higher in PIRANA. But, one can query $\lfloor \frac{N}{1.5} \rfloor$ elements for the same communication cost in PIRANA (multi-query).

PIRANA was also compared to some state-of-the-art PIR schemes by Liu et al. [27]. To answer a single query, PIRANA is mostly slower than other PIR schemes. However, PIRANA becomes more competitive when the number of queries increases.

[4] https://github.com/microsoft/SEAL

Table 2.1: Performance comparison of CwPIR and PIRANA [27]

	# elements n DB Size (MB)	2^8 5.2	2^9 10	2^{10} 21	2^{11} 42	2^{12} 84	2^{13} 170	2^{14} 340	2^{15} 670	2^{16} 1300
CwPIR [2]	Selection Vec. (s)	3.9	7.8	15.5	31.0	61.7	123.1	246.2	492.7	983.3
	Inner Product (s)	0.2	0.4	0.8	1.6	3.3	6.5	13.1	26.2	52.3
	Total server (s)	4.1	8.2	16.3	32.6	65.0	129.7	259.4	518.9	1035.6
PIRANA (single-query)	Selection Vec. (s)	0.001	0.001	0.001	0.001	0.001	0.001	0.027	0.05	0.1
	Inner Product (s)	0.22	0.24	0.28	0.36	0.52	0.86	1.57	2.86	5.39
	Total server (s)	0.22	0.24	0.28	0.36	0.52	0.86	1.6	2.9	5.49
	Speedup	18.6×	34.2×	58.2×	90.6×	125×	151×	162.1×	178.9×	188.6×

2.8 Fheanor

In this thesis, we will implement GBFV in the Fheanor library[5]. Feanor is a Rust library containing building blocks for homomorphic encryption, implementing several FHE schemes, including BFV and CLPX[6]. Fheanor is build on feanor-math[7], a Rust library for number theory and algebra. Both libraries are open-source and can be found on GitHub.

The library supports implementations over both power-of-two and genaral cyclotomics [31]. This is interesting for FHE implementations, in particular because the use of non-power-of-two cyclotomics can allow greater SIMD capabilities by having a larger number of slots with small plaintext moduli. Fheanor also explicitly models arithmetic circuits, providing tools for their computation. The Fheanor library is close in performance to the HElib and SEAL libraries, which are state-of-the-art.

Spiessens created a wrapper called easyGBFV, which will use the Fheanor library to create a GBFV-scheme. This wrapper offers some easy-to-use functions. Some functions that will be used in this GBFV implementation are:

- `DenseMatrixMul::new(ring, columns, data, desc)`: data is a 2D-matrix, flattened in a row major manner, where elements at position (i, j) are stored at index $i \cdot columns + j$ of the flattened matrix. The description argument is a name that will be given to the cache file used to store the precomputed data. The function returns a MatrixMul struct that can be used in homomorphic matrix multiplication.

- `PlaintextFromSlots(&GBFV, ptslots: I)`[8]: takes a iterator over elements of a SlotRIng and returns an element of a plaintext ring.

- `HomMatMul(&GBFV, &MatrixMul, &[Ciphertext], &PublicKey)`: multiplication of the 2D-matrix, or one chunck of the 3D-matrix for large payloads, with the ciphertext column selector. This function creates a circuit that is evaluated when performing the homomorphic matrix multiplication.

[5]https://github.com/FeanorTheElf/fheanor
[6]CLPX is not implemented in any other major library [31]
[7]https://github.com/FeanorTheElf/feanor-math
[8]This function is not offered by easyGBFV, but is implemented in this thesis.

Chapter 3

PIR implementation

This chapter discusses the implementation of different PIR protocols. First, some utility functions that are used in the implementations are explained. Next, the implementation of the PIRANA protocol for small and large payloads is presented. After that, an alternative implementation using one-hot encoding is discussed. Finally, the testbench used to benchmark the different implementations with large payloads is explained.

The implementations can be found in the thesis github[1]. The examples in the gihub from 4 to 7 are created to test and debug the different implementations. Therefore, a significant amount of steps are printed and the user can choose which element is queried from the database at runtime. This is not suitable for benchmarking. Therefore, examples 8 and 9 are created to benchmark PIRANA and one-hot encoding for large payloads. These implementations do not print unncessary information, the queried element is chosen at random and timing measurements are performed in order to benchmark the implementations.

3.1 Utility functions

Some utility functions are created to help with the implementation of the PIR protocols. These functions are implemented in `util.rs`.

- `GenFromI32(&Ring, &[i32])`: Generates a vector of ring elements, for the specified ring, from a vector of i32 values.
- `GenFromBigInt(&Ring, &[BigInt])`: Generates a vector of ring elements, for the specified ring, from a vector of BigInt values.
- `d3_finder(element_size_bit: usize, p_mod: &str)`: This function will determine in how many chunks a large payload can be split. It will return the amount of chunks. To achieve this, the function needs to know the integer modulus `p_mod` and the maximal size of an element in the database (in bits).

[1] https://github.com/antoinejvdm/easygbfv_PIR

Equation 3.1 shows how the number of chunks are calculated.

$$\text{number of chunks} = \left\lceil \frac{\texttt{element\_size\_bit}}{\lfloor log_2(\texttt{p\_mod}) \rfloor} \right\rceil \tag{3.1}$$

- `calculate_cw_len(k,columns)`: The constant-weight codeword length m is calculated with this function. The function takes the Hamming weight k and the amount of columns in the database as input arguments. If k equals 2 (default parameter), the function calculates m via this formula:

$$\frac{m \cdot (m-1)}{2} \geq n_{\text{columns}} \implies m = \left\lceil \frac{1 + \sqrt{1 + 8 \cdot n_{\text{columns}}}}{2} \right\rceil \tag{3.2}$$

. In any other case, m will be calaculated via a iterative approach. It will return the minimal m such that $\binom{m}{k} \geq columns$.

- `base_p_decompose(n,p, chunks)`: This function will do a base-p decomposition of a big integer n into chunks. Euclidian division of the integer n by the plaintext modulus p is used. The division will be done *chunks* amount of times, and every chunk will keep the remainder of the division. This will create a vector of size *chunks*, containing numbers smaller than p.

- `recompose_base_p_to_str(digits, p)`: This function will recompose the chunks back into one large integer. It will take the vector of chunks and the plaintext modulus p. The recomposition is done by multiplying every chunk with p^i, with i the index of the chunk in the vector. The results are summed together to create one large integer, which is then converted to a string and returned.

$$n = \sum_{i=0}^{chunks-1} digits[i] \cdot p^i \tag{3.3}$$

- `get_rand_matrix(nr_elements, element_size_bits, nr_slots, p)`: To create the database, this function is used. It will create a 3D-matrix with size $r \times t \times chunks$, with r the number of slots in a ciphertext, t the number of elements divided by the number of slots, and *chunks* the amount of chunks needed to split one large element. Every element in the database is a large integer with size equal to *element_size_bits*. Every large integer is split into *chunks* chunks via base-p decomposition. This function will return the 3D-matrix. To avoid recreating the matrix every time the function is called upon, the matrix is stored in a cache file. If the cache file already exists, the matrix is loaded from the file instead of being created again.

3.2 GBFV-PIRANA, single-query small payload

In this section, the implementation of the PIRANA protocol using the easyGBFV library is presented. The implementation is a single-query implementation for small payloads. This means that the elements of the database are smaller when compared to the plaintext modulus p. The single-query small payload implementation can be found in the `examples\5_GBFV_PIRANA_Spayload` folder of the thesis github.

First, a database/matrix is created with size $r \cdot c$, where r is equal to the number of slots in the ciphertext and c is equal to the amount of elements divided by the number of slots in the ciphertext[2]. All indices of the columns of the matrix are substituted with a constant weight codeword. To achieve this, m and k have to be chosen properly. In this implementation, k will be set to 2, meaning that every codeword has a Hamming weight of 2. This will keep the amount of ciphertext-ciphertext multiplications low. Knowing k, m can be calculated as $c \leq \binom{m}{2}$, with c the amount of columns in the matrix. Later, every column will be multiplied with a ciphertext. Therefore, the plaintext elements of one column are set into a plaintext ring.

Subsequently, an instance of GBFV is created. When creating this instance, one has to set m the cyclotomic order, p the integer modulus and t the plaintext modulus. EasyGBFV has some GBFV parameters already set, to create GBFV instances of 16/32/64 bits of plaintext modulus.

Having a database and having created a GBFV instance, the PIRANA set-up is finished. The client can now create a query for an element in the database. Imagine the client wants to retrieve element (i, j) from the database. First, the client will look up which codeword corresponds to column j. The client will create the query matrix, which is a matrix of size $r \cdot m$. This matrix is an all-zero matrix, except for the i-th row, which is subsituted with the codeword.

Before sending the query, the client has to encrypt the query. Therefore, he generates a secret key and a public key. Every column of length r (amount of slots in a ciphertext) will be encrypted. The client sends m ciphertexts to the server.

The server will, for each column, take the codewords of length m and look at which position the codeword has a 1. In our case, there are only two one's (remember, the Hamming weight equals 2). The server will take the corresponding ciphertexts of these two positions in the query and multiply them with each other. This new ciphertext is one column of the selection matrix. This process is repeated for all c columns of the database. After creating the selection matrix, the server will perform a homomorphic plaintext-ciphertext multiplication between every column of the selection matrix and the corresponding column of the database. Finally, all the columns of the resulting matrix are summed together, by going through all the columns and adding them via an accumulator. The result is then sent back as one ciphertext to the client.

The client will receive the ciphertext from the server and will decrypt using his secret key. Every ciphertext is decrypted and will return a vector of slot ring elements. All elements are equal to zero, except for the i-th element, which is equal to the desired element in the database. The client can now format this element and retrieve the desired value.

[2]When working with small payloads, all elements in the database are of maximal size i32 or plaintext modulo.

3.3 GBFV-PIRANA, single-query large payload

In this section, the implementation of a single-query PIR protocol for large payloads using the easyGBFV library is presented. The implementation can be found in the `examples\7_GBFV_PIRANA_Lpayload` folder of the thesis github. The implementation is similar to the small payload implementation, with some differences. First, the database is created. The database is now a 3D-matrix of size $r \cdot t \cdot chunks$, where r is equal to the number of slots in the ciphertext, t is equal to the amount of elements divided by the number of slots and $chunks$ is equal to the amount of chunks needed to split one large element. Every large element in the database is split into $chunks$ chunks via base-p decomposition, with p the plaintext modulus. Every chunk is smaller than p. The query generation works in the same way as for small payloads, where a constant-weight codeword is set at the i-th row of the query matrix. At the server side, the implementation works in the same way as the small payload implementation. The selection matrix is created in the same way, but when multiplying the selection matrix with the database, this operation is repeated for every chunk of the database. This will result in $chunks$ ciphertexts. To reduce the number of ciphertexts sent back to the client, a rotate-and-sum operation is performed on every ciphertext. This will result in $\frac{chunks}{slots}$ ciphertexts which are sent back to the client. The client will decrypt every ciphertext and recompose the chunks into one large integer via base-p recomposition. The client can now retrieve the desired element from the database.

pseudocode rotate and sum

3.4 GBFV one-hot encoding

As shown in the PIRANA paper [27], PIRANA is slower when compared to most other PIR protocols when querying one element. PIRANA becomes more competitive when querying multiple elements at once. Therefore, using the PIRANA protocol to get one element is suboptimal. An alternative way to retrieve one element from a database is to use one-hot encoding. The implementation can be found in the `example\6_OneHot_Lpayload`. Instead of sending a query matrix, as in PIRANA, with one-hot encoding only two vectors are sent. Both vectors are all-zero vectors, except at position i for the first vector (row vector) and at position j for the second vector (column vector). When using small elements, the database in one-hot encoding has to be structured in a 2D-matrix. The first dimension equals the number of slots, while the second dimension equals t, where $t = \frac{n}{s}$ with n the amount of elements in the database and s the number of slots in a ciphertext. Algorithm ?? shows the query generation for large or small payloads. The matrx multiplication used in 2 needs a vector of length equal to a multiple of the number of slots. Therefore, padding might be needed, when the amount of columns is not a multiple of the number of slots.

The two query vectors are encrypted and sent to the server. The server will perform a matrix multiplication between the column vector and the database, resulting

Algorithm 1 One-hot query generation

```
1: $columns \leftarrow \frac{elements}{slots}$
2: if $columns \bmod slots \neq 0$ then
3:     $padded\_columns \leftarrow columns + (slots - (columns \bmod slots))$
4: end if
5: $j \in \{0, \dots, columns - 1\}$
6: $i \in \{0, \dots, slots - 1\}$
7: $col\_selector \leftarrow$ array of zeros of length $padded\_columns$
8: $elem\_selector \leftarrow$ array of zeros of length $slots$
9: $col\_selector[j] \leftarrow 1$
10: $elem\_selector[i] \leftarrow 1$
11: $row\_selector\_slots \leftarrow \texttt{GenFromI32}(slotring, elem\_selector)$
12: $col\_selector\_slots \leftarrow \texttt{GenFromI32}(slotring, col\_selector)$
```

in a ciphertext containing only the j-th column of the database. This ciphertext is then multiplied with the row vector, resulting in a ciphertext containing only the desired element. This ciphertext is sent back to the client, who can decrypt and retrieve the desired element. When handling large elements, the database is set up as a 3D-matrix, where the third dimension equals the amount of chunks needed to split one large element. The server will need to perform the matrix multiplication for every chunk, resulting in multiple ciphertexts. Algorithm 2 shows the server-side operations for both large payloads.

A rotate-and-sum operation is performed to reduce the amount of ciphertexts sent back to the client. Communication cost can be reduced when using one-hot encoding instead of PIRANA, as discussed in Chapter 4.

3.5 Testbench

To benchmark the different implementations, two testbenches are created. The first testbench is testing PIRANA for large payloads while the second is testing one-hot encoding for large payloads. All operations performed by the server are set in a seperate Rust file. This file will have one function that mimics the working of the server. This function takes the GBFV instance, the private key, the encrypted queries and the database as input arguments. The output of this function will return the selected element as a ciphertext. Both testbenches work in a similar manner. They start by making the GBFV instance. When running the testbench, the user can pass arguments to choose the size of the plaintext modulus and the N, the cyclotomic order. The creation of the instance is timed. Next, the database is created. The user can pass arguments to choose the amount of elements in the database and the size of each element (in bits). The database is processed. For GBFV every column of `BigInt` is converted into a slotted plaintext, that can be used for ciphertext-plaintext multiplications. Algorithm 3 shows the database processing

Algorithm 2 One-hot encoding search algorithm

```
1:  $nrChunks \leftarrow$ number of chunks per element
2:  for $ch = 0$ to $numChunks - 1$ do
3:      $layer \leftarrow$ empty 2D array with capacity $numChunks \times slots$
4:      $colInChunk \leftarrow$ number of columns in one $ch$ of the matrix
5:      for $r = 0$ to $slots - 1$ do
6:          $rowVals \leftarrow$ empty array of length $paddedColumns$
7:          for $c = 0$ to $paddedColumns - 1$ do
8:              if $c < colInChunk$ then
9:                  Append $matrix[ch][r][c]$ to $rowVals$
10:             else
11:                 Append 0 (zero element in ring) to $rowVals$
12:             end if
13:         end for
14:         $slotsRow \leftarrow \textsc{GenFromBigInt}(slotring, rowVals)$
15:         Append $slotsRow$ to $layer$
16:     end for
17:     $data \leftarrow$ empty array with capacity $slots \times paddedColumns \times paddedColumns$
18:     for $rowIdx = 0$ to number of rows in $layer - 1$ do
19:         for $colIdx = 0$ to $paddedColumns - 1$ do
20:             Append $layer[rowIdx][colIdx]$ to $data$
21:         end for
22:     end for
23:     $mm \leftarrow \textsc{DenseMatrixMul}(slotring, padded\_columns, data)$
24:     $ct\_col \leftarrow \textsc{HomMatMul}(gbfv, mm, ct\_col\_select, pk)$
25:     $ct\_chunk\_element \leftarrow$ element-wise homomorphic multiplication of $ct\_col$
        and $ct\_row\_select$ using $gbfv$
26:     Append $ct\_chunk\_element$ to $ct\_elements$
27: end for
```

for PIRANA. The original matrix contains all `BigInt` elements of the database, split into chunks. The algorithm returns a processed matrix, which is a 2D-matrix of size $chunks \times columns$, where every element is a slotted plaintext. For one-hot encoding every chunk is set into a dense `DenseMatrixMul`, which is a type which is used for the matrix multiplication. The database creation is also timed. In order to produce realistic measurements of the performance of the server, one element is chosen at random and is queried from the database. This is done multiple times and the average time is taken. Every iteration, the query generation time, the server response time and the decryption and recomposition time are measured. The testbenches will print the average times of every operation. Next to these three timing measurements, every time an addition, multiplication (ct-ct, pt-ct) and rotation is performed by the server, a counter is increased. The time spent for each of these specific operations is also measured. At the end of the testbench, the total amount of operations and the average time spent per operation is printed. This will give insight into which operations are the most costly and where future optimizations can be made. At the end of every iteration, a check is done to verify the correctness of the retrieved element. This retrieved element is compared to the original element in the database.

Algorithm 3 Processing of database for PIRANA

```
 1: $chunks \leftarrow$ number of chunks per element
 2: $colums \leftarrow \frac{elements}{slots}$
 3: $matrix \leftarrow$ 3D-matrix of size $slots \times columns \times chunks$
 4: $ProcessedMatrix \leftarrow$ empty matrix with capacity $chunks \times colums$
 5: for $ch = 0$ to $chunks - 1$ do
 6:     $chRows \leftarrow$ empty vector with capacity $colums$
 7:     for $c = 0$ to $columns - 1$ do
 8:         $rowVals \leftarrow$ empty array of length $slots$
 9:         for $r = 0$ to $slots - 1$ do
10:             if $c < colInChunk$ then
11:                 Append $rowVals$ with $matrix[ch][r][c]$
12:             else
13:                 Append $rowVals$ with 0 (zero element in ring)
14:             end if
15:         end for
16:         $slotsRow \leftarrow \textsc{GenFromBigInt}(slotring, rowVals)$
17:         $slotsRowPt \leftarrow \textsc{PlaintextFromSlots}(slotsRow)$
18:         Append $chRows$ with $slotsRowPt$
19:     end for
20:     $ProcessedMatrix[ch] \leftarrow chRows$
21: end for
```

Chapter 4

Results

4.1 GBFV-PIRANA versus one-hot encoding

4.1.1 Communication cost

One-hot encoding can be used to reduce the communication cost when querying one element from a database. The communication cost in PIRANA when querying one element is m ciphertexts, where m is equal to $O\left(\sqrt[k]{k!\,n}+k\right)$. The communication cost in one-hot encoding is equal to one ciphertext for the rows, because there are as many rows as slots, and t/s ciphertexts for the columns, since there are more columns than slots in a ciphertext. So the query communication cost, which is the amount of ciphertexts sent from client to server, is equal to:

$$\texttt{query communication cost} = 1 + \frac{n}{s^2} \tag{4.1}$$

As can be deduced from the formulas above, the communication cost in one-hot encoding will be lower for small databases (small n). In Table ?? the number of elements is shown from which on PIRANA has a lower query communication cost (number of ciphertexts that constitute the query) than one-hot encoding. The results are shown for different amount of slots s and a Hamming weight of $k = 2$.

Slots s	2	4	8	16	32	64	128
Elements n	46	574	8446	132094	2101246	33570814	536936446
Query ct's	11	35	131	515	2052	8196	32772

Table 4.1: Number of elements n from when PIRANA has less communication cost than one-hot encoding (results are indicated for different amount of slots s and Hamming weight $k = 2$).

4.1.2 Performance

In this section, one-hot encoding and PIRANA are compared, both implemented with a GBFV scheme. To compare both protocols with each other, the same GBFV

Table 4.2: Performance comparison of one-hot encoding and GBFV-PIRANA for plaintext modulus $\sim$ 32 bits

	# elements n DB Size (KB)	2^8 32	2^9 64	2^{10} 128	2^{11} 256	2^{12} 512	2^{13} 1024
PIRANA	Query generation (s)	0.010	0.011	0.014	0.019	0.026	0.038
	Server response (s)	0.0356	0.0818	0.1689	0.3560	0.7409	1.5703
	Dec. and recomposition (s)	0.006	0.006	0.006	0.007	0.007	0.008
One-hot	Query generation (s)	0.006	0.007	0.008	0.011	0.016	0.027
	Server response (s)	0.0705	0.1206	0.2259	0.4643	1.0582	2.7632
	Dec. and recomposition (s)	0.005	0.005	0.005	0.005	0.005	0.005

Table 4.3: Performance comparison of one-hot encoding and GBFV-PIRANA for plaintext modulus $\sim$ 64 bits

	# elements n DB Size (KB)	2^8 32	2^9 64	2^{10} 128	2^{11} 256	2^{12} 512	2^{13} 1024
PIRANA	Query generation (s)	0.067	0.080	0.102	0.136	0.186	0.251
	Server response (s)	0.4431	0.9560	1.9430	4.4696	10.5309	26.1016
	Dec. and recomposition (s)	0.028	0.028	0.029	0.031	0.033	0.036
One-hot	Query generation (s)	0.052	0.068	0.101	0.166	0.300	0.536
	Server response (s)	0.2196	0.4619	1.1564	3.2916	10.7572	38.2325
	Dec. and recomposition (s)	0.028	0.028	0.028	0.028	0.028	0.028

scheme is implemented. The scheme will use $N = 2^6$ and $N = 2^8$ as degree of the cyclotomic polynomial ring for respective plaintext moduli of $32bits$ and $64bits$. One element will be randomly selected from the database. Both schemes are compared for speed of query generation, server response time and decryption and recomposition time. The comparison is done for multiple databases, containing a number of elements varying from 2^8 to 2^{13}. The element size is 1024 bits. The respective times are shown in Table 4.2 and 4.3.

4.1.3 Database processing time

Before doing any measurements, the database has to be created and processed. The database creation time is the time it takes to create the database of big integers. The database processing time is the time it takes to preprocess the database for querying. For GBFV-PIRANA, preprocessing the databases consists of converting every column of the database into a slotted plaintext, that can later be used for a ciphertext-plaintext multiplication. On the other hand, for one-hot encoding, preprocessing the database consists of setting every chunk of the big integer database into a DenseMatrixMul, which is a type of plaintext matrix that can later be used for a matrix-vector multiplication with ciphertexts.

Note that for one-hot encoding, the server will perform a matrix-vector multiplication. To do this multiplication, a circuit is created, which is evaluated to perform

Table 4.4: Performance comparison of GBFV-PIRANA with different plaintext moduli (N^8, 512-bits element size, 2^{12} elements)

plaintext modulus (bits)	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}	2^{11}
number of slots s	16	64	8	32	16	8	4	2
Query generation (s)	0.136	0.108	0.229	0.155	0.216	0.398	1.140	2.758
Server response (s)	3.0285	0.2676	6.0063	0.2630	2.0646	1.5483	1.3357	2.2908
Dec. and recomposition (s)	0.021	0.041	0.027	0.037	0.058	0.073	0.071	0.090

the multiplication. A downside of this approach is that the circuit has to be created for every chunk of a database. This leads to huge server response time when calling this multiplication for a first time. Once the circuit is created, evaluating the circuit becomes faster, since the circuit was already created. This is not a step of the database processing, but it is still a set-up time that has to be taken into account.

The database processing times (including the server response time) for one-hot encoding and GBFV-PIRANA, for different database sizes and different plaintext moduli (so different amount of slots), are shown in Table **??**.

4.2 GBFV-PIRANA - parameter optimization

To find the optimal parameters for a GBFV-PIRANA implementation, N is set to 2^8, the number of elements is set to 2^{12} and the size of one element is 512-bit. The amount of slots will be determined by the plaintext modulus. This section compares slots with plaintext modulus going from 16-bit up to 2048-bit. Query generation time, server response time and decryption and recomposition times are shown in Table 4.4.

4.3 GBFV-PIRANA versus BFV-PIRANA

The PIRANA-paper by Liu et al. describes a PIR protocol based on BFV. In this thesis, the same protocol is implemented but using a GBFV scheme. This section compares the performance of both implementations. The performance results for the BFV-PIRANA were already shown in Table 2.1. The GBFV implementation of PIRANA uses $N = 2^{13}$, has a payload size of 20 KB (as in the PIRANA-paper). The plaintext modulus size is set to 128 bits, hereby creating 1024 slots. The plaintext modulus size is chosen based on the results of the previous section, where using this plaintext modulus showed lower query generation, server response and decryption and recomposition times.

The implementation of BFV-PIRANA is limited: the maximum plaintext modulus has size 32 bits, and the implementation is only available for small elements (where the element size is smaller than the plaintext size).

Two comparisons are performed for plaintext sizes of 16 and 32 bits with $N = 2^{10}$. For GBFV, the polynomials $t(x)$ are chosen as $t(x) = X^{2^8} - 2$ and $t(x) = X^{2^6} - 288$, thus creating 256 and 64 slots for the 16- and 32-bit cases, respectively. The same

Table 4.5: Performance of GBFV-PIRANA, when using $N = 2^{13}$ and payload size 20 KB)

# elements n 2^8	2^9	2^{10}	2^{11}	2^{12}
DB Size (KB)	32	64	128	256
Query generation (s)	15.988	15.583	15.280	15.986
Server response (s)	13.6678	13.5988	13.5965	22.8538
Dec. and recomposition (s)	56.418	56.548	56.349	63.738

Table 4.6: Performance of GBFV-PIRANA and BFV-PIRANA, when using $N = 2^{10}$ and payload size 1024bits)

	Query generation (s)	Server response (s)	Decryption (s)	Total (s)
GBFV ($p = 16$) (s)	0.505	0.107	0.086	0.698
BFV ($p = 16$) (s)	0.012	0.006	0.105	0.124
GBFV ($p = 32$) (s)	0.530	0.028	0.445	1.003
BFV ($p = 32$) (s)	-	-	-	-
Speedup ($p = 16$)	41.83x	18.42	0.82x	5.65x

database is used for both comparisons, with 1024 elements of size 16 or 32 bits, depending on the plaintext modulus. The results of the performance comparison of these two implementations are shown in Table 2.

4.4 Overhead large versus small payload

Two GBFV-PIRANA protocols are implemented in the code of this thesis. The first protocol can only take small payloads, where the payload is smaller than the plaintext modulus. The second protocol allows for handling large playloads, larger than the plaintext modulus. Here these two implementations are compared in terms of performance (query generation time, server response time and decryption and recomposition time).

Chapter 5

Discussion

5.1 Implementation

In this thesis, PIRANA is implemented with a GBFV-scheme. GBFV-PIRANA is hereby built upon the Fheanor library. Testbenches are created to measure the performance of the implementation, these testbenches measure the query generation time, server response time and decryption and recomposition time for different database sizes, element sizes, cyclotomic orders and plaintext moduli. The results of these measurements are shown in chapter 4 and the implementatons can be found in the thesis github.

5.2 GBFV-PIRANA versus one-hot encoding

When increasing the number of slots, the communication cost will be lower in one-hot encoding up to a larger database size. For example, when taking a Hamming weight of 2 for PIRANA and the amount of slots equal to 16, one-hot encoding has a lower communication cost up to a databse size of 131583 elements (see Table ??). When lowering the number of slot to 1, one-hot encoding only has a lower communication cost for a database up to 3 elements. From that moment on, PIRANA will have a lower communication cost. This is due to the fact that, when the database grows, the amount of queries for one-hot encoding will grow faster when compared to PIRANA. Every time the length m of the constant-weight codewords grows, it allows for more columns in the database. In PIRANA increasing n, the amount of elements in the database, will lead to a k-th square root increase in the length m of the codewords, while for one-hot encoding increasing n will lead to a linear increase in the length m of the codewords. Therefore, there exists a point, a certain database size, from which on PIRANA will have a lower communication cost.

Next to communcation cost analysis, the performance of both protocols are also compared. The results are shown in table 4.2 and 4.3. From these results, it can be seen that GBFV-PIRANA outperforms one-hot encoding in server response time for all database sizes when using a plaintext modulus of 32 bits and for $N = 2^6$. Query generation time and decryption and recomposition time are slightly worse for GBFV-PIRANA, but these times can be neglected in the total protocol time, since the server response time is much larger. When using a plaintext modulus of 64 bits and using $N = 2^8$, GBFV-PIRANA outperforms one-hot encoding in server response time for database sizes larger or equal to 2^{12} elements. For smaller databases, one-hot encoding has a slightly better server response time. Again, query generation time and decryption and recomposition time are slightly worse for GBFV-PIRANA for small databases, better for databases larger or equal to 2^{12} elements, but these times can be neglected in the total protocol time. It can indeed be expected that the query generation time is smaller for one-hot encoding for small databases, since there are less ciphertexts in one query, the communication cost is lower (see previous part). When the database size grows, GBFV-PIRANA will have a smaller amount of ciphertexts in one query, so the query generation time will be smaller when compared to one-hot encoding for large database sizes. Regarding the decryption and recomposition times, one can expect a similar behaviour. The measurements were done on elements of the same size, so with the same amount of chunks. Therefore, after the rotate-and-sum operation, both protocols will have the same amount of returned ciphertexts. So the client should take the same amount of time to decrypt these ciphertexts. In table 4.2 and 4.3 it can be seen that the decryption and recomposition times are indeed very similar for both protocols. The server response time can differ between both protocols because of the different underlying operations performed. Remember, in one-hot encoding, a circuit has to created when evaluating the matrix-vector multiplication. Therefore, the first element request will create a huge server response time. In the results we omitted this first server response time, since it only occurs once. For example, for a database size of 2^{13}, element size of 1 Kb, plaintext modulus of 64 bits, the first querying an element will take a server response time of 726.3 seconds. This while the average of the next queries is 38.2 seconds. One advantage of GBFV-PIRANA is that there is no circuit created, so there is no huge first server response time. Note that table 4.2 and 4.3 are not to be compared, since the cyclotomic order N differs.

Also, the database processing time is very high when using $N = 2^{13}$, which makes testing cumbersome. For every scheme with a different amount of slots, the database has to be created and processed for different dimensions.

5.3 GBFV-PIRANA parameter optimization

To find the optimal parameters for a GBFV-PIRANA implementation, N was set to 2^8, the number of elements was set to 2^{12} and the size of one element was 512-bit. The goal is to find an optimal plaintext modulus (and so a number of slots) for this implementation. Table 4.4 shows that in the range of plaintext moduli from 2^8 to 2^{11}, the server response time is the lowest at 2^{10}. This corresponds to 4 slots in the ciphertext. The query generation time and decryption and recomposition time are increasing when increasing the plaintext modulus. The results for plaintext moduli smaller than 2^8 show anomalies. One should expect that, when evaluating different plaintext moduli, an optimal plaintext modulus exists over the full range of possible plaintext moduli, which is difficult to define in the case of our results.

5.4 GBFV-PIRANA versus BFV-PIRANA

The performance of GBFV-PIRANA was evaluated and compared to BFV-PIRANA, using the same parameters as in the PIRANA-paper. The results show that the GBFV-PIRANA implementation is in all means slower than the BFV-PIRANA implementation.

But, as mentioned in the results section, this comparison is suboptimal since different hardware and libraries are used. Therefore, BFV-PIRANA was also implemented on the Fheanor library, to make a fair comparison between both implementations. The results of this comparison show that ....

5.5 Overhead large versus small payload

5.6 Future work

In this thesis GBFV-PIRANA was implemented for single-query and built upon the Fheanor library. Multiple further optimizatons and extensions can be made to this work. First of all, the implementation can be extended to multi-query PIRANA. Untill now, only single-query PIRANA is implemented. However, PIRANA supports multi-query natively. Implementing multi-query PIRANA could increase the performance and make the protocol more competitive in comparison to other PIR-protocols. Next to this, to increase the performance even more, GBFV-PIRANA should be implemented on a more performant library. Fheanor is a library created for researchers, and not optimized for performance. However, to implement GBFV-PIRANA on a more performant library, this library has to implement the GBFV-scheme. Due to the low performance of Fheanor, testing performance for different parameter sets is very time consuming for large databases or big cyclotomic orders. Therefore, implementing GBFV-PIRANA on a more performant library would also allow to test more parameter sets and find the optimal parameters for GBFV-PIRANA more easily. In the thesis implementation of BFV-PIRANA, the highest plaintext modulus is of size i32. This is a huge limitation and blocks comparison for big integers. Therfore, implemeting BFV with big integers would allow to compare GBFV-PIRANA and BFV-PIRANA for big integers.

Chapter 6

Conclusion

In this thesis, PIRANA is for a first time successfully implemented with a GBFV scheme. To this end, GBFV-PIRANA was implemented on top of the Fheanor library using the easyGBFV wrapper. Both single-query PIRANA variants, targeting small and large payloads, were implemented. A comprehensive testbench was developed to evaluate performance of the developed implementation for different database sizes, payload sizes, and plaintext moduli.

This is of particular interest, since GBFV allows for greater flexibility in parameter selection; fewer but bigger slots are possible when compared to BFV (cf. PIRANA was implemented with BFV by Liu et al. [27]). Also, GBFV could allow to evaluate deeper circuits or could allow to work with smaller ring dimensons, by combining the properties of BFV and CLPX (Section 2.4). By decoupling the number of slots from the ring dimension and allowing larger plaintext moduli, GBFV can accommodate larger payload chunks and reduce the number of ciphertexts required for large-payload retrieval. This can lead to lower communication overhead.

The experimental results demonstrate that GBFV-PIRANA offers clear advantages over one-hot encoding in terms of communication efficiency once the database reaches a certain size threshold, even for single-query PIRANA. This threshold depends on the number of SIMD slots and the chosen constant-weight code parameters, but aligns well with theoretical expectations.

Experimental results show that implementing GBFV-PIRANA is feasible, but performance lags when compared to the original PIRANA-paper implemented on a performant library. This is mainly due to the fact that Fheanor is not optimized for performance, but rather for research purposes. This opens directions for future work, such as implementing GBFV-PIRANA on a more performant library and implementing multi-query GBFV-PIRANA thereon.

This thesis showed the feasibility of implemeting PIRANA with a GBFV scheme and provided insights into its performance characteristics.

Bibliography

[1] C. Aguilar-Melchor, J. Barrier, L. Fousse, and M.-O. Killijian. XPIR: Private information retrieval for everyone. Cryptology ePrint Archive, Paper 2014/1025, 2014.

[2] R. Akhavan Mahdavi and F. Kerschbaum. Constant-weight pir: Single-round keyword pir via constant-weight equality operators. In *Proceedings of the 31st USENIX Security Symposium*, page 1723–1740. USENIX Association, 2022.

[3] A. Q. M. Al-Kateeb. *Structures and Properties of Cyclotomic Polynomials.* PhD thesis, North Carolina State University, 2016. Ph.D. dissertation.

[4] D. Balbás. The hardness of LWE and ring-LWE: A survey. Cryptology ePrint Archive, Paper 2021/1358, 2021.

[5] S. Belaïd, N. Bon, A. Boudguiga, R. Sirdey, D. Trama, and N. Ye. Further improvements in AES execution over TFHE: Towards breaking the 1 sec barrier. Cryptology ePrint Archive, Paper 2025/075, 2025.

[6] N. Bon, D. Pointcheval, and M. Rivain. Optimized homomorphic evaluation of boolean functions. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(3):302–341, Jul. 2024.

[7] Z. Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. Cryptology ePrint Archive, Paper 2012/078, 2012.

[8] A. Burton, S. J. Menon, and D. J. Wu. Respire: High-rate PIR for databases with small records. Cryptology ePrint Archive, Paper 2024/1165, 2024.

[9] Y.-C. Chang. Single database private information retrieval with logarithmic communication. In *Information Security and Privacy*, pages 50–61, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[10] H. Chen, K. Laine, R. Player, and Y. Xia. High-precision arithmetic in homomorphic encryption. In *Topics in Cryptology – CT-RSA 2018*, Lecture Notes in Computer Science, pages 116–136. Springer, Mar. 2018. International Conference on Cryptographers Track at the RSA Conference on Topics in Cryptology, CT-RSA 2018 ; Conference date: 16-04-2018 Through 20-04-2018.

[11] H. Chen, K. E. Lauter, and K. E. Stange. Attacks on the search-RLWE problem with small error. Cryptology ePrint Archive, Paper 2015/971, 2015.

[12] I. Chillotti. TFHE Deep Dive - Ilaria Chillotti, FHE.org. https://www.youtube.com/watch?v=LZuEr4jpyUw, Aug. 2022. YouTube video, Accessed: 2025-11-13.

[13] C. Christian, M. Silviio, and S. Markus. Computationally private information retrieval with polylogarithmic communication. In J. Stern, editor, *Advances in Cryptology — EUROCRYPT '99: International Conference on the Theory and Application of Cryptographic Techniques*, volume 1592 of *Lecture Notes in Computer Science*, pages 202–214, Prague, Czech Republic, 1999. Springer.

[14] B. DeCoste. Secret sharing explained. https://medium.com/dropoutlabs/secret-sharing-explained-acf092660d97, Nov. 2018. Accessed: 2025-12-24.

[15] European Data Protection Supervisor. Glossary. https://www.edps.europa.eu/data-protection/data-protection/glossary/p_en#pets. Accessed: 24 December 2025.

[16] J. Fan and F. Vercauteren. Somewhat practical fully homomorphic encryption. Crytology ePrint Archive, Paper 2012/144, 2012.

[17] R. Geelen. Revisiting the slot-to-coefficient transformation for BGV and BFV. Cryptology ePrint Archive, Paper 2024/153, 2024.

[18] R. Geelen and F. Vercauteren. Bootstrapping for bgv and bfv revisited. *Journal of Cryptology*, 36(12), 2024.

[19] R. Geelen and F. Vercauteren. Fully homomorphic encryption for cyclotomic prime moduli. Cryptology ePrint Archive, Paper 2024/1587, 2024.

[20] A. Ghoshal, B. Li, Y. Ma, C. Dai, and E. Shi. Scalable multi-server private information retrieval. Cryptology ePrint Archive, Paper 2024/765, 2024.

[21] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In M. F. Balcan and K. Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 201–210, New York, USA, 20–22 Jun 2016. PMLR.

[22] N. J. Bouman. Comparison of Privacy Enhancing Technologies and MPC, Aug. 2024.

[23] J. Kang and L. Schild. Pirouette: Query efficient single-server PIR. Cryptology ePrint Archive, Paper 2025/680, 2025.

[24] A. Kim, Y. Polyakov, and V. Zucca. Revisiting homomorphic encryption schemes for finite fields. In *Lecture Notes in Computer Science*, volume 1309, pages 608–639. Springer, 2021.

[25] J. Kim. Bootstrapping GBFV with CKKS. Cryptology ePrint Archive, Paper 2025/888, 2025.

[26] E. Kushilevitz and R. Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 364–373, 1997.

[27] J. Liu, J. Li, D. Wu, and K. Ren. PIRANA: Faster multi-query PIR via constant-weight codes. Cryptology ePrint Archive, Paper 2022/1401, 2022.

[28] Machine Learning Research. Combining machine learning and homomorphic encryption in the apple ecosystem. URL: `https://machinelearning.apple.com/research/homomorphic-encryption`, last checked on 2025-27-10.

[29] U. Mattsson. Security and Performance of Homomorphic Encryption, Apr. 2025.

[30] C. Meghan, D. Deeksha, A. Armin, T. Caroline, L. T. Vincent, and R. Brandon. Porcupine: a synthesizing compiler for vectorized homomorphic encryption. *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 375–389, June 2021.

[31] H. Okada, R. Player, and S. Pohmann. Fheanor: a new, modular FHE library for designing and optimising schemes. Cryptology ePrint Archive, Paper 2025/864, 2025.

[32] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer, 1999.

[33] R. Sion and B. Carbunar. On the computational practicality of private information retrieval. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*, pages 364–373, San Diego, California, USA, 2007. Internet Society (ISOC).

[34] N. P. Smart and F. Vercauteren. Fully homomorphic simd operations. *Designs, Codes and Cryptography*, 71:57 – 81, 2012.

[35] Y. Xu, W. Cui, and M. Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. *2015 IEEE Symposium on Security and Privacy*, pages 640–656, May 2015. ISSN: 2375-1207.