UNIVERSITAT DE BARCELONA

Facultat de Matemàtiques
i Informàtica

Treball Final del Grau de Matemàtiques

Una introducció a la criptografia homomòrfica. Implementació de l'esquema BGV.

Carles Albàs Boix

Director: Dr. Artur Travesa i Grau

Barcelona, 21 de juny de 2020

Abstract

Homomorphic cryptography has as its main objective being able to do operations on ciphertexts without knowing their contents or compromising their security. In this thesis we present an introduction to this new research area by exploring its fundamental principles, the Gentry's scheme, the Learning With Errors problem and the BGV scheme, as well as the required theoretical tools necessary to understand them. Finally, we make a little implementation of the BGV scheme in order to analyze the associated algorithms and to see some practical applications.

Resum

La criptografia homomòrfica té per objectiu fer operacions amb textos xifrats sense saber-ne el contingut i sense comprometre'n la seguretat. En aquest treball presentem una introducció a aquesta nova àrea de recerca explorant-ne els principis fonamentals, l'esquema de Gentry, el problema *Learning With Errors* i l'esquema BGV, així com les eines teòriques necessàries per entendre-ho. Finalment, fem una petita implementació de l'esquema BGV per analitzar-ne l'algorísmica associada i veure'n algunes aplicacions pràctiques.

2010 Mathematics Subject Classification. 06B10, 94A60, 11T55, 68W01

Agraïments

Vull agrair al meu tutor, el Dr. Artur Travesa, l'ajuda i la direcció durant la realització d'aquest treball.

Contingut

1 Introducció

El 1978, en Ron Rivest, Leonard Adleman i Michael Dertouzos [20] publiquen un article on reflexionen sobre el problema següent: Existeix algun criptosistema que permeti realitzar operacions sobre textos xifrats sense saber-ne el contingut? En concret, posen l'exemple d'una petita empresa de préstecs que utilitza un servei comercial de computació a temps compartit per guardar i processar les dades. Com que l'empresa de préstecs guarda informació sensible sobre els seus clients i hi ha altres programadors utilitzant l'ordinador a temps compartit, decideixen xifrar totes les dades. Ara bé, aquest model permet a la companyia utilitzar les capacitats d'emmagatzematge de l'ordinador a temps compartit, però no les seves capacitats computacionals, ja que per operar sobre les dades caldria desxifrar-les dins de l'ordinador de temps compartit i conseqüentment comprometre'n la seva seguretat. Per tant, les úniques opcions que té la companyia són, o bé renunciar a la idea de xifrar les dades, o bé instal·lar un ordinador local que no podrà tenir les mateixes prestacions que l'ordinador a temps compartit. A no ser que existeixi el que ells anomenen un *privacy homomorphism* per encriptar les dades i que permeti a l'ordinador de temps compartit operar sobre elles sense necessitat de desxifrar-les.
Malgrat que han passat més de 40 anys des d'aquest article, el paral·lelisme amb el món actual és immediat. Els serveis d'ordinadors a temps compartit de l'època han sigut reemplaçats pels serveis al núvol, on també nosaltres hi guardem i hi processem les nostres dades. Per tant ens podem fer la mateixa pregunta: És possible xifrar les nostres dades de tal manera que un servidor al núvol pugui operar sobre elles? Un exemple seria poder realitzar una cerca entre els nostres arxius, sense que el servidor hi tingui accés ni sàpiga què estem buscant, o realitzar algun càlcul molt complex que volem delegar a un ordinador més potent però sense comprometre'n la confidencialitat.
Aquest problema va ser resolt, malgrat que de forma purament teòrica, al 2009, a la tesi doctoral del Craig Gentry. Des d'aleshores s'han anat proposant esquemes similars i cada cop millors, però cap d'ells suficientment eficient com per permetre'n una adopció generalitzada.

1.1 Notació general

S'ha intentat mantenir una notació més o menys consistent al llarg del treball, tot i contenir resultats de varis orígens diferents. En general, una lletra minúscula en negreta $\mathbf{v}$ denotarà un vector, i una majúscula $\mathbf{B}$ una matriu. Un polinomi el denotarem tan en negreta o sense, depenent de si el veiem com a polinomi o vector de coeficients. Donat un algorisme determinista D denotarem $y = D(x_1, ..., x_n)$ el procés d'assignar a la variable y el resultat de D. Donat un algorisme probabilístic E, en canvi, ho denotarem $y \leftarrow E(x_1, ..., x_n)$, per deixar clar que $E(x_1, ..., x_n)$ pot prendre valors diferents en successives execucions. De manera similar, si χ és una distribució de probabilitat, denotarem $e \leftarrow \chi$ el procés de prendre una mostra de χ i assignar-la a la variable e, a diferència de $e = \chi$, que significaria que e és una distribució igual a χ en comptes d'una mostra d'ella.

1.2 Circuits

Comencem per introduir el concepte de circuit. Com que sempre tindrem present l'aspecte computacional d'allò que fem, en comptes de parlar de funcions o aplicacions, que no és clara la complexitat computacional que poden tenir, parlarem de circuits.

Definició 1.1. *Sigui F un conjunt no buit. Anomenem* porta *a una operació n-ària, és a dir, a una aplicació:*

$$g : F^n \to F$$

on n és el grau *de la porta. En el cas particular $n = 0$ parlem de portes constants o portes d'entrada, depenent de si el seu valor està definit en el moment de construir el circuit.*

Les portes són els blocs fonamentals amb els quals construirem un circuit:

Definició 1.2. *Un circuit C sobre un conjunt de variables $v_1, ..., v_n$ en un conjunt F és un graf finit dirigit acíclic on tots els seus vèrtexs són portes. Les arestes d'entrada de cada vèrtex estan ordenades i corresponen a les entrades de la porta, i les arestes de sortida a còpies de la sortida. Les portes de grau de sortida 0 corresponen a la sortida del circuit.*

Notació 1. *En parlar de circuits utilitzarem el llenguatge habitual, reemplaçant els termes utilitzats en teoria de grafs: als vèrtexs els anomenarem* nodes *o* portes*, i a les arestes,* cables*.*

Un circuit no deixa de ser una aplicació $C : F^n \to F^m$. Ara bé, l'avantatge de veure'l com un graf dirigit acíclic construït amb portes és que tenim clara la seva complexitat computacional, és a dir, quins passos s'ha de seguir per a fer el càlcul. Hem d'anar avaluant totes les portes de forma progressiva fins a tenir el resultat final.
Un circuit té dues mesures de complexitat associades: la *mida* i la *profunditat*. La mida (o complexitat) d'un circuit és el nombre de portes que conté. La profunditat és la llargada del seu camí més llarg. A més a més, definim el *nivell* d'un determinat node com el màxim de la llargada de tots els camins que tenen com a vèrtex final el node en qüestió.

Definició 1.3. *Direm que un circuit C està* anivellat *si totes les arestes connecten nodes de nivell consecutiu.*

Si tenim un circuit no anivellat el podem anivellar afegint portes trivials, és a dir, operacions unàries on la sortida és igual a l'entrada.
En el cas de tenir un circuit definit sobre $F = \{0, 1\}$ parlarem de circuit *booleà*, i tindrem les portes lògiques tradicionals: AND, OR, NOT, etc. En canvi, si F té estructura d'anell i tenim portes corresponents a les operacions de suma i producte de l'anell, parlarem de circuit *aritmètic*. Veiem ara com podem encadenar circuits.

Definició 1.4. *Siguin C un circuit arbitrari i D un circuit amb un únic node de sortida. El circuit C* augmentat *per D és el circuit obtingut substituint els nodes d'entrada de C per còpies de D. Ho notarem* $C \circ D$*.*

1.3 Criptografia homomòrfica

Podem intuir l'essència de la criptografia homomòrfica de la manera següent: donats els textos xifrats que encripten $\pi_1, ..., \pi_n$ un sistema completament homomòrfic hauria de permetre a qualsevol, no només a qui té les claus privades, calcular un xifrat de $f(\pi_1, ..., \pi_n)$, per a qualsevol funció f, sense que es filtri cap informació sobre els textos plans o $f(\pi_1, ..., \pi_n)$: l'entrada, la sortida i tots els valors intermedis estan sempre xifrats. Per connectar la criptografia homomòrfica amb un esquema molt conegut com és el RSA veiem la propietat següent. Si tenim una instància RSA amb mòdul n i clau pública e i xifrem dos missatges sense farcit obtenim que

$$\mathsf{Enc}(m_1) \cdot \mathsf{Enc}(m_2) \equiv m_1^e \cdot m_2^e \equiv (m_1 m_2)^e \equiv \mathsf{Enc}(m_1 m_2) \mod n.$$

Diem que el RSA sense farcit és parcialment homomòrfic, ja que ho és només pel producte. El paral·lelisme amb un homomorfisme algebraic és clar: el xifrat de la funció és la funció dels xifrats. Notem que el RSA sense farcit no és semànticament segur i per tant no s'hauria d'utilitzar mai.
Un esquema tradicional de clau pública $\mathcal{E}$ entre un espai de textos plans $\mathcal{P}$ i un espai de textos xifrats $\mathcal{X}$ consta de tres algorismes.

- $\mathsf{KeyGen}_{\mathcal{E}}$: és un algorisme probabilístic que, donat com a entrada un paràmetre de seguretat λ (normalment $\lambda \in \mathbb{R}^+$), genera una clau secreta $sk \in \mathcal{K}_s$ i una clau pública $pk \in \mathcal{K}_p$.
- $\mathsf{Encrypt}_{\mathcal{E}}$: pren un $\pi \in \mathcal{P}$ i la clau pública pk i dóna com a resultat un $\psi \in \mathcal{X}$. Pot ser probabilístic.
- $\mathsf{Decrypt}_{\mathcal{E}}$: pren un $\psi \in \mathcal{X}$ i la clau secreta sk i dóna com a resultat un $\pi \in \mathcal{P}$. És determinista.

Definició 1.5. *Diem que un esquema* $\mathcal{E}$ *és* correcte *si per a qualsevol parella de claus* $(pk, sk) \leftarrow \mathsf{KeyGen}_{\mathcal{E}}(\lambda)$ *i per a tot* $\pi \in \mathcal{P}$ *se satisfà que* $\pi = \mathsf{Decrypt}_{\mathcal{E}}(sk, \mathsf{Encrypt}_{\mathcal{E}}(pk, \pi))$*.*

Un sistema criptogràfic homomòrfic té, a més a més, un algoritme $\mathsf{Evaluate}_{\mathcal{E}}$ que pren la clau pública pk, un circuit C dins d'un conjunt de circuits permesos $\mathcal{C}_{\mathcal{E}}$ i una llista ordenada de textos xifrats $\Psi = \langle \psi_1, ..., \psi_n \rangle$ per a les entrades de C i dóna com a sortida un text xifrat $\psi \in \mathcal{X}$.

Definició 1.6. *Diem que un esquema de xifrat homomòrfic $\mathcal{E}$ és* correcte *per a circuits de $\mathcal{C}_\mathcal{E}$ si per a tota parella de claus $(pk, sk) \leftarrow \mathsf{KeyGen}_\mathcal{E}(\lambda)$, tot circuit $C \in \mathcal{C}_\mathcal{E}$ i tota família de missatges plans $\pi_1, ..., \pi_t \in \mathcal{P}$ que formen les entrades de C, si posem que $\psi_i = \mathsf{Encrypt}_\mathcal{E}(pk, \pi_i)$ i $\psi = \mathsf{Evaluate}_\mathcal{E}(pk, C, \langle \psi_1, ..., \psi_t \rangle)$, aleshores $\mathsf{Decrypt}_\mathcal{E}(sk, \psi) = C(\pi_1, ..., \pi_t)$.*

És convenient tenir una manera d'excloure els esquemes trivials, és a dir, aquells que el seu algorisme $\mathsf{Evaluate}_\mathcal{E}(pk, C, \Psi)$ senzillament es limita a retornar la parella (C, Ψ) i després és l'algorisme $\mathsf{Decrypt}_\mathcal{E}$ l'encarregat d'aplicar el circuit C un cop ha desxifrat Ψ. Aquest esquema seria correcte però no és interessant.

Definició 1.7. *Diem que un esquema de xifrat homomòrfic $\mathcal{E}$ és* compacte *si existeix un polinomi f tal que $\forall \lambda$ l'algorisme $\mathsf{Decrypt}_\mathcal{E}$ es pot expressar com un circuit $\mathcal{D}_\mathcal{E}$ de mida com a molt $f(\lambda)$.*

Per tant la complexitat de l'algorisme de desxifrat d'un esquema compacte té una cota superior que depèn només del paràmetre de seguretat λ, i per tant no pot dependre del circuit C. Diem que un esquema *avalua compactament* circuits en $\mathcal{C}_\mathcal{E}$ si és compacte i correcte per a tots els circuits en $\mathcal{C}_\mathcal{E}$. A partir d'aquí ja podem definir què és un sistema completament homomòrfic.

Definició 1.8. *Diem que un esquema de xifrat homomòrfic és* completament *homomòrfic si és compacte i correcte per a tots els circuits.*

Aquesta definició, en general, és massa forta per als esquemes amb els quals treballarem. Una condició menys forta és que l'esquema avaluï circuits de profunditat com a molt d.

Definició 1.9. *Direm que una família $\{\mathcal{E}^{(d)}\}_{d \in \mathbb{N}}$ d'esquemes completament homomòrfics és* anivellada *(leveled) si l'algorisme de desxifrat és el mateix per a tots ells i $\mathcal{E}^{(d)}$ avalua compactament tots els circuits de profunditat d.*

De forma paral·lela a la capacitat d'un esquema d'avaluar homomòrficament circuits, ens interessarà que el text xifrat resultant d'aplicar la funció $\mathsf{Evaluate}_\mathcal{E}$ no revel·li res sobre el circuit que s'ha avaluat.

Definició 1.10. *Diem que un esquema de xifrat homomòrfic $\mathcal{E}$ és* privat per a circuits *en $\mathcal{C}_\mathcal{E}$ si per a tota parella de claus $(sk, pk) \leftarrow \mathsf{KeyGen}_\mathcal{E}(\lambda)$ i per a tot $C \in \mathcal{C}_\mathcal{E}$ se satisfà que per a $\pi_1, ..., \pi_n \in \mathcal{P}$ i tot $\langle \psi_1, ..., \psi_n \rangle \leftarrow \mathsf{Encrypt}_\mathcal{E}(pk, \langle \pi_1, ..., \pi_n \rangle)$ les dues distribucions $\mathsf{Encrypt}_\mathcal{E}(pk, C(\pi_1, ..., \pi_n))$ i $\mathsf{Evaluate}_\mathcal{E}(pk, C, \psi_1, ..., \psi_n)$ són estadísticament indistingibles, és a dir, no existeix cap algorisme polinòmic que distingeixi mostres d'una de l'altre amb avantatge no negligible en λ.*

Com que treballem amb sistemes abstractes, els conjunts de textos plans i de textos xifrats poden no ser iguals. Per tant, a priori, no podem parlar de conceptes com "xifrar un text dues vegades", ja que un element de $\mathcal{X}$ pot no estar en $\mathcal{P}$. Per solucionar aquest problema introduïm el concepte de codificació o representació.

Definició 1.11. *Siguin A i B dos conjunts finits arbitraris. Anomenem* representació o codificació *del conjunt A amb elements de B a qualsevol funció injectiva*

$$r : A \to B^n$$

Anomenem n el grau *de la representació i ho denotem $n = gr(r)$.*

Notem que el grau n no pot ser qualsevol; ha de permetre que r sigui injectiva.

1.4 Bootstrapping

Aquí introduirem el concepte de *bootstrapping*. Com que no he trobat una bona traducció del terme anglès l'utilitzaré en l'original. Diem que un sistema de xifrat homomòrfic és *bootstrappable* si pot avaluar compactament el seu propi circuit de desxifrat (lleugerament augmentat). Veurem que aquesta condició és suficient per convertir el sistema de xifrat homomòrfic en un sistema completament homomòrfic. Aquesta tècnica va ser introduïda al 2009 per Craig Gentry [8].

Definició 1.12. *Sigui $\mathcal{E}$ un esquema de xifrat. Suposem que tenim representacions de $\mathcal{X}$ i $\mathcal{K}_s$ amb elements de $\mathcal{P}$, de graus m i l respectivament. Anomenem $\mathcal{D}_\mathcal{E}$ al circuit amb portes en $\mathcal{P}$ associat a l'algorisme $\mathsf{Decrypt}_\mathcal{E}(sk, \psi)$, és a dir, al circuit:*

$$\mathcal{D}_\mathcal{E} \colon \mathcal{P}^l \times \mathcal{P}^m \to \mathcal{P}$$
$$(sk_1, ..., sk_l; \psi_1, ..., \psi_m) \mapsto \mathcal{D}_\mathcal{E}(sk_1, ..., sk_l, \psi_1, ..., \psi_m)$$

tal que per a tota parella de claus (pk, sk) i tot $\pi \in \mathcal{P}$, si $\psi \leftarrow \mathsf{Encrypt}_\mathcal{E}(pk, \pi)$ i si $\langle sk_i \rangle_{i\in[1,l]}$ i $\langle \psi_j \rangle_{j\in[1,m]}$ són representacions de sk i ψ, respectivament, aleshores $\mathcal{D}_\mathcal{E}(\langle sk_i \rangle_{i\in[1,l]}, \langle \psi_j \rangle_{j\in[1,m]}) = \pi$.

Definició 1.13. *Siguin $\mathcal{E}$ un esquema de xifrat homomòrfic, $\mathcal{D}_\mathcal{E}$ el seu circuit de desxifrat. Per a tot conjunt Γ de portes definides en $\mathcal{P}$, denotem per $\mathcal{D}_\mathcal{E}(\Gamma)$ el conjunt de tots els circuits de la forma $g \circ \mathcal{D}_\mathcal{E}$ amb $g \in \Gamma$.*

Definició 1.14. *Sigui $\mathcal{E}$ un esquema de xifrat homomòrfic que avalua compactament tots els circuits en $\mathcal{C}_\mathcal{E}$. Direm que $\mathcal{E}$ és* bootstrappable *respecte de Γ si $\mathcal{D}_\mathcal{E}(\Gamma) \subseteq \mathcal{C}_\mathcal{E}$.*

Sigui $\mathcal{E}$ bootstrappable respecte de totes les portes en Γ. Aleshores, per a tot enter $d \geq 1$, podem construir un esquema $\mathcal{E}^{(d)}$ que pot avaluar compactament circuits amb portes en Γ i de profunditat d. La descripció dels algorismes que componen l'esquema és la següent.

- $\mathsf{KeyGen}_{\mathcal{E}^{(d)}}(\lambda, d)$. Pren un paràmetre de seguretat λ i un enter positiu d i per a $\ell = \ell(\lambda)$ posa:

$$(sk_i, pk_i) \leftarrow \mathsf{KeyGen}_\mathcal{E}(\lambda) \qquad \text{per a } i \in [0, d],$$

$$\overline{sk_{ij}} \leftarrow \mathsf{Encrypt}_\mathcal{E}(pk_{i-1}, sk_{ij}) \quad \text{per a } i \in [1, d], j \in [1, \ell],$$

  on $sk_{i1}, ..., sk_{i\ell}$ és la representació de sk_i amb elements de $\mathcal{P}$. Aleshores dóna:

$$sk^{(d)} = sk_0,$$

$$pk^{(d)} = (\langle pk_i \rangle_{i\in[0,d]}, \langle \overline{sk_{ij}} \rangle_{i\in[1,d],j\in[1,l]}).$$

  Denotarem per $\mathcal{E}^{(\delta)}$ el subesquema que utilitza $sk^{(\delta)} = sk_0$ i $pk^{(d)} = (\langle pk_i \rangle_{i\in[0,\delta]}, \langle \overline{sk_{ij}} \rangle_{i\in[1,\delta],j\in[1,\ell]})$, per a $\delta \leq d$.

- $\mathsf{Encrypt}_{\mathcal{E}^{(d)}}(pk^{(d)}, \pi)$. Pren la clau pública $pk^{(d)}$ i un text pla $\pi \in \mathcal{P}$. Dóna el text xifrat $\psi \leftarrow \mathsf{Encrypt}_\mathcal{E}(pk_d, \pi)$.

- $\mathsf{Decrypt}_{\mathcal{E}^{(d)}}(sk^{(d)}, \psi)$. Pren una clau secreta $sk^{(d)}$ i un text xifrat $\psi \in \mathcal{X}$ i dóna $\mathsf{Decrypt}_\mathcal{E}(sk_0, \psi)$.

Abans de definir $\mathsf{Evaluate}_{\mathcal{E}^{(d)}}$ necessitem definir dos algorismes.

- $\mathsf{Augment}_{\mathcal{E}^{(\delta)}}(pk^{(\delta)}, C_\delta, \Psi_\delta)$. Pren una clau pública $pk^{(\delta)}$, un circuit C_δ de profunditat com a molt δ i una llista ordenada de textos xifrats Ψ_δ, encriptats sota $pk^{(\delta)}$. Augmenta el circuit de desxifrat $\mathcal{D}_\mathcal{E}$ amb C_δ:

$$C^\dagger_{\delta-1} = C_\delta \circ \mathcal{D}_\mathcal{E}.$$

  Siguin (ψ_k), $k \in [1, m]$ la representació de $\psi \in \Psi_\delta$ amb elements de $\mathcal{P}$ i $\overline{\psi_k} \leftarrow \mathsf{Encrypt}_{\mathcal{E}^{(\delta-1)}}(pk^{\delta-1}, \psi_k)$. Posem:

$$\Psi^\dagger_{\delta-1} = \langle \langle \overline{sk_{\delta j}} \rangle_{j\in[1,l]}, \langle \overline{\psi_k} \rangle_{k\in[1,m]} \rangle \mid \psi \in \Psi_\delta \rangle$$

  La sortida és $(C^\dagger_{\delta-1}, \Psi^\dagger_{\delta-1})$.

- $\mathsf{Reduce}_{\mathcal{E}^{(\delta)}}(pk^{(\delta)}, C^\dagger_\delta, \Psi^\dagger_\delta)$. Pren una clau pública $pk^{(\delta)}$, un circuit $C^\dagger_\delta \in \mathcal{D}_\mathcal{E}(\Gamma, \delta + 1)$ i $\Psi^\dagger_\delta$ una llista ordenada de parelles de textos xifrats, pertanyents a la imatge de $\mathsf{Encrypt}_{\mathcal{E}^{(\delta)}}$. Definim C_δ com el sub-circuit de $C^\dagger_\delta$ consistent en els seus últims δ nivells i calcula Ψ_δ, els textos xifrats induïts d'entrada de C_δ, és a dir, per cada node d'entrada w de C_δ, es posa $\psi^{(w)}_\delta \leftarrow \mathsf{Evaluate}_\mathcal{E}(pk_\delta, C^{(w)}_\delta, \Psi^{(w)}_\delta)$, on $C^{(w)}_\delta$ és el sub-circuit de $C^\dagger_\delta$ amb node de sortida w i $\Psi^{(w)}_\delta \subseteq \Psi^\dagger_\delta$ són els textos xifrats d'entrada de $C^{(w)}_\delta$.

- $\mathsf{Evaluate}_{\mathcal{E}^{(\delta)}}(pk^{(\delta)}, C_\delta, \Psi_\delta)$. Pren una clau pública $pk^{(\delta)}$, un circuit C_δ anivellat de profunditat com a molt δ amb portes en Γ, i una llista ordenada de textos xifrats Ψ_δ (on cada text xifrat ho ha d'estar sota $pk^{(\delta)}$). Si $\delta = 0$, retorna Ψ_0 i l'algorisme acaba. En cas contrari fa el següent de forma recursiva:
  1. $(C^\dagger_{\delta-1}, \Psi^\dagger_{\delta-1}) \leftarrow \mathsf{Augment}_{\mathcal{E}^{(\delta)}}(pk^{(\delta)}, C_\delta, \Psi_\delta)$,
  2. $(C_{\delta-1}, \Psi_{\delta-1}) \leftarrow \mathsf{Reduce}_{\mathcal{E}^{(\delta-1)}}(pk^{(\delta-1)}, C^\dagger_{\delta-1}, \Psi^\dagger_{\delta-1})$,
  3. Retorna $\mathsf{Evaluate}_{\mathcal{E}^{(\delta-1)}}(pk^{(\delta-1)}, C_{\delta-1}, \Psi_{\delta-1})$.

Fem ara una explicació simplificada per entendre el procediment d'una manera més senzilla. Partim d'un circuit C_d amb portes en Γ. Per cada node d'entrada w de C_d hi ha un text xifrat ψ_w, xifrat amb la clau pública pk_d. També tenim un sistema de xifrat $\mathcal{E}$ capaç d'avaluar compactament circuits en $D_\mathcal{E}(\Gamma)$. Notem que en cap cas hem assumit que $\mathcal{E}$ pugui avaluar circuits arbitraris amb portes en Γ, sinó només aquelles portes augmentades pel circuit de desxifrat $D_\mathcal{E}$. El primer pas, realitzat pel procediment $\mathsf{Augment}$, consisteix a posar còpies de $D_\mathcal{E}$ a les entrades de C_d i així construir $C^\dagger_{d-1}$. A més a més, es xifren els textos $\psi_w \in \Psi_\delta$ (representacions d'ells amb elements de $\mathcal{P}$) amb la clau pública pk_{d-1}. En el segon pas, Reduce executa l'avaluació compacta dels sub-circuits de desxifrat afegits al pas anterior i un nivell més del circuit, prenent com a entrada textos xifrats successivament amb pk_d i pk_{d-1} així com $\overline{sk_d}$, que és un xifrat de $\overline{sk_d}$ amb la clau pk_{d-1}. D'aquesta manera, en avaluar el circuit de desxifrat, obtindrem un text xifrat únicament amb la clau pk_{d-1}. Efectivament s'ha fet un canvi de claus i s'ha avançat un nivell en el circuit que volem avaluar. El punt més important recau en el fet de que l'esquema $\mathcal{E}$ pot avaluar, no només el seu circuit de desxifrat, sinó un nivell més enllà. D'aquesta manera es pot anar avançant en l'avaluació del circuit C_d. En el cas de que només pogués avaluar el circuit de desxifrat però no més enllà, podríem fer el canvi de claus igualment, però no avaluar circuits arbitraris.
Podem visualitzar el sistema de bootstrapping de la manera següent. Prenem un objecte i el fiquem dins d'una caixa A. Després fiquem la caixa A dins d'una caixa B i la caixa interior A es destrueix. L'objecte ha canviat de caixes però mai ha estat fora de cap, garantint així la seguretat del sistema.

1.5 Seguretat

Per analitzar les implicacions del *bootstrapping* en la seguretat del sistema introduïm primer un seguit de conceptes previs. Informalment, direm que un sistema de xifrat és *semànticament* segur si cap atacant no pot extreure informació no negligible en temps polinòmic donat un missatge xifrat. Estudiarem la seguretat d'un esquema mitjançant *jocs d'atac*, que un *adversari* intenta guanyar a un *desafiador*. Tot esquema de xifrat té associat un paràmetre de seguretat, que anomenem λ, que sol ser un nombre real. El paràmetre de seguretat regula la resta de paràmetres de l'esquema per generar una instància més o menys segura depenent de les necessitats.

Joc d'atac 1.15 (seguretat semàtica)**.** Donats un esquema de clau pública $\mathcal{E} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ definit en l'espai de textos plans $\mathcal{P}$ i de textos xifrats $\mathcal{X}$ i un adversari $\mathcal{A}$, definim l'experiment $b \in \{0, 1\}$.

- El desafiador calcula $(pk, sk) \leftarrow \mathsf{KeyGen}(\lambda)$ i dóna pk a l'adversari.
- L'adversari pren $m_0, m_1 \in \mathcal{P}$, de la mateixa longitud, i els envia al desafiador.
- El desafiador calcula $c \leftarrow \mathsf{Enc}(pk, m_b)$ i envia c a l'adversari.
- L'adversari retorna $\hat{b} \in \{0, 1\}$.

Denotem W_b l'esdeveniment on $\mathcal{A}$ retorna 1 a l'experiment b i definim l'avantatge de $\mathcal{A}$ respecte $\mathcal{E}$ com

$$\mathrm{SSadv}[\mathcal{A}, \mathcal{E}] = |P(W_0) - P(W_1)|.$$

Definició 1.16. *Diem que un esquema de clau pública $\mathcal{E}$ és* semànticament segur *si per a qualsevol adversari $\mathcal{A}$, el valor $SSadv[\mathcal{A}, \mathcal{E}]$ és negligible respecte el paràmetre de seguretat λ de $\mathcal{E}$.*

Notem que si un esquema de clau pública és semànticament segur, aleshores el seu algorisme Enc ha de ser probabilístic. Si Enc fos determinista, com que l'adversari té la clau pública, podria calcular els xifrats de m_0 i m_1, comparar-los amb el text xifrat donat pel desafiador i guanyar el joc amb avantatge 1. Una definició més forta de seguretat ens ve donada a partir de l'atac de text xifrat escollit, o CCA (*Chosen Ciphertext Attack*) per les seves sigles en anglès.

Joc d'atac 1.17 (CCA)**.** Donats un esquema de clau pública $\mathcal{E} = (\mathsf{KeyGen}(\lambda), \mathsf{Enc}, \mathsf{Dec})$ definit en l'espai de textos plans $\mathcal{P}$ i de textos xifrats $\mathcal{X}$ i un adversari $\mathcal{A}$, definim l'experiment $b \in \{0, 1\}$.

- El desafiador calcula $(pk, sk) \leftarrow \mathsf{KeyGen}$ i envia pk a l'adversari.
- L'adversari $\mathcal{A}$ fa un seguit de peticions al desafiador, que poden ser de dos tipus diferents.
  - Petició de xifrat: La i-èsima petició de xifrat consisteix en un parell de missatges plans $(m_{i,0}, m_{i,1}) \in \mathcal{P}^2$, de la mateixa mida. El desafiador retorna $c_i \leftarrow \mathsf{Enc}(pk, m_{i,b})$ a $\mathcal{A}$.
  - Petició de desxifrat: La j-èsima petició de desxifrat consisteix en un text xifrat $\hat{c}_j \in \mathcal{X}$ tal que $\hat{c}_j \notin c_1, ..., c_i$. El desafiador retorna $\hat{m}_j = \mathsf{Dec}(sk, \hat{c}_j)$.
- Al final del joc $\mathcal{A}$ retorna $\hat{b} \in \{0, 1\}$.

Anomenem W_b l'esdeveniment on $\mathcal{A}$ retorna 1 a l'experiment b. Definim l'avantatge de $\mathcal{A}$ respecte $\mathcal{E}$ com

$$\mathrm{CCAadv}[\mathcal{A}, \mathcal{E}] = |P(W_0) - P(W_1)|.$$

Definició 1.18. *Un esquema de clau pública $\mathcal{E}$ s'anomena* semànticament segur contra un atac de text xifrat escollit*, o CCA-segur, si per a tot adversari $\mathcal{A}$ el valor $CCAadv[\mathcal{A}, \mathcal{E}]$ és negligible respecte λ.*

Si en el joc d'atac CCA eliminem la possibilitat de fer peticions de desxifrat, direm que és un joc d'atac de text pla escollit o CPA (*Chosen Plaintext Attack*). Es pot demostrar que per un sistema de xifrat de clau pública, que són els únics amb els que treballem, la seguretat semàntica implica la seguretat per CPA, i per tant ens centrarem en la seguretat semàntica. Per un desenvolupament més a fons d'aquestes qüestions es pot consultar el llibre de Dan Boneh i Victor Shoup [3].
Un altre concepte important és el de mal·leabilitat. Direm que un sistema de xifrat és mal·leable si donat un text xifrat c d'un missatge m aleshores un adversari pot calcular un xifrat c' d'un missatge m' que té alguna relació amb m. Es pot demostrar que la CCA-seguretat implica la no mal·leabilitat. Com que els sistemes de xifrat homomòrfics són mal·leables per definició aleshores mai podran ser CCA-segurs.
El procés de *bootstrapping* preserva la seguretat semàntica de l'esquema inicial. La demostració es troba a la tesi de Gentry [8].

Teorema 1.19. *Sigui $\mathcal{A}$ un algorisme que trenca la seguretat semàntica de $\mathcal{E}^{(d)}$ amb avantatge ϵ. Aleshores existeix un algorisme $\mathcal{B}$ que trenca la seguretat semàntica de $\mathcal{E}$ amb avantatge $\epsilon' \geq \epsilon/\ell(d+1)$, on ℓ és el grau de la representació de les sk amb elements de $\mathcal{P}$.*

Hem vist que a partir d'un esquema de xifrat *bootstrappable* $\mathcal{E}$ podem construir un esquema de xifrat homomòrfic anivellat $\mathcal{E}^{(d)}$ que és semànticament segur si $\mathcal{E}$ ho és. La mida de la clau de $\mathcal{E}^{(d)}$ és proporcional a d. El següent que ens podem preguntar és com construir un esquema de xifrat completament homomòrfic, que pugui avaluar un nombre il·limitat de nivells utilitzant una clau finita. Notem que en la construcció de $\mathcal{E}^{(d)}$ hem utilitzat una successió de claus, on cada una xifra la següent de forma lineal, sense cap cicle. Podem convertir l'esquema $\mathcal{E}^{(d)}$ en un esquema $\mathcal{E}^*$ cíclic, afegint $\overline{sk_{0,j}} = \mathsf{Encrypt}(pk_d, sk_{0,j})$ per a $j \in [1, \ell]$. Aquest sistema és completament homomòrfic, ja que el cicle de claus ens permet aplicar el procés de *bootstrapping* indefinidament, però ja no en podem demostrar la seva seguretat semàntica. Introduïm un altre concepte de seguretat, anomenat seguretat KDM (*Key Dependent Messages*), on els missatges poden contenir part de la clau. Queda formalitzat en el joc d'atac següent.

Joc d'atac 1.20 (KDM)**.** Donats un esquema de clau pública $\mathcal{E} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ definit en l'espai de textos plans $\mathcal{P}$ i de textos xifrats $\mathcal{X}$ i un adversari $\mathcal{A}$, definim l'experiment $b \in \{0, 1\}$.

- Per a un cert n polinòmic en λ el desafiador calcula $(pk_i, sk_i) \leftarrow \mathsf{KeyGen}(\lambda)$ per a $i \in [0, n-1]$. Si $b = 0$, per a tot $i \in [0, n-1]$ i tot $j \in [1, \ell]$ posa $\overline{sk_{i,j}} \leftarrow \mathsf{Encrypt}_{\mathcal{E}}(pk_{(i-1) \bmod n}, sk_{i,j})$, on $sk_{i,j}$ és el j-èsim element de la representació de sk amb elements de $\mathcal{P}$. Si $b = 1$ genera els $\overline{sk_{i,j}}$ a partir de claus privades aleatòries i no associades a les claus públiques $pk_0, ..., pk_{n-1}$. Envia les claus públiques i les claus privades xifrades a $\mathcal{A}$.
- L'adversari retorna $\hat{b} \in \{0, 1\}$.

Denotem W_b l'esdeveniment on $\mathcal{A}$ retorna 1 a l'experiment b i definim l'avantatge de $\mathcal{A}$ respecte $\mathcal{E}$ com

$$\mathrm{KDMadv}[\mathcal{A}, \mathcal{E}] = |P(W_0) - P(W_1)|.$$

Definició 1.21. *Un esquema de clau pública $\mathcal{E}$ s'anomena KDM-segur, si per a tot adversari $\mathcal{A}$ el valor $KDMadv[\mathcal{A}, \mathcal{E}]$ és negligible respecte λ.*

En la definició general del joc KDM l'adversari pot fer peticions de xifrat de qualsevol funció de les claus públiques. Ara bé, en el context de la criptografia homomòrfica l'adversari ja pot calcular aquestes funcions a partir dels xifrats individuals de les claus. La seguretat KDM no és gens trivial de demostrar i per a molts esquemes no està clar si són KDM-segurs o no. Si un esquema $\mathcal{E}$ és *bootstrappable* i KDM-segur aleshores podem construir un esquema de xifrat completament homomòrfic que sigui semànticament segur.

1.6 Aplicacions

D'entre les possibles aplicacions de la criptografia homomòrfica en destaquem les següents.

- **Processament extern de dades sensibles**. Amb el creixement de la computació al núvol moltes empreses ja no tenen servidors propis sinó que lloguen màquines virtuals a proveïdors de serveis. Ara bé, físicament aquests proveïdors poden tenir accés a les dades que s'estan processant i en alguns contextos això pot ser indesitjable. La criptografia homomòrfica permetria operar amb aquestes dades sense sacrificar-ne la confidencialitat.
- **Delegació de càlculs complexos**. De forma similar al punt anterior, es podria delegar el processament d'unes dades a una empresa externa però pel motiu que el càlcul a realitzar és massa complex i es necessita una màquina més potent. Una condició necessària per això és que l'esquema de xifrat homomòrfic permeti operar amb dades xifrades amb una eficiència similar al que es pot fer amb textos plans. Els esquemes actuals, malauradament, estan molt lluny.
- **Protecció d'algorismes**. La criptografia homomòrfica permetria a una empresa protegir un algorisme propi i a la vegada operar amb dades sensibles de tercers. Per exemple, una companyia podria desenvolupar un model matemàtic que no vol fer públic per detectar certes malalties a partir de dades biomètriques. Aleshores, un client podria enviar les seves dades mèdiques xifrades i obtenir el resultat de l'anàlisi sense perdre confidencialitat.

1.7 Estandardització

No existeix cap estàndard oficial de cap esquema de xifrat homomòrfic. Existeix un grup de treball format per diferents institucions acadèmiques i empreses amb pàgina web *https://homomorphicencryption.org* que treballa per l'estandardització de la criptografia homomòrfica. El seu esborrany [1], si bé malgrat no és un estàndard complet que es pugui implementar directament, sí que intenta unificar la notació i algunes definicions, sintetitzar i ordenar la bibliografia i donar una sèrie de recomanacions a l'hora d'ajustar paràmetres per obtenir diferents nivells de seguretat. La criptografia homomòrfica és una àrea de recerca activa i és possible que s'arribi a una estandardització completa, formal i oficial durant els pròxims anys.

2 Xarxes

En aquesta secció parlarem breument de xarxes i exposarem alguns problemes relacionats amb elles, ja que ho necessitem per desenvolupar la teoria en altres seccions.

Definició 2.1. *Una xarxa en $\mathbb{R}^n$ és un subgrup del grup additiu $\mathbb{R}^n$, isomorf a $\mathbb{Z}^n$ i que genera l'espai vectorial $\mathbb{R}^n$. Diem que n és la dimensió de la xarxa.*

Si tenim $v_1, ..., v_m \in \mathbb{R}^n$ generadors de l'espai vectorial $\mathbb{R}^n$, aleshores la xarxa generada per ells és

$$\mathcal{L}(v_1, ..., v_m) = \left\{ \sum_{i=1}^{m} x_i v_i \mid x_i \in \mathbb{Z} \right\}.$$

En el cas que $v_1, ..., v_n$ siguin una base de $\mathbb{R}^n$ també són una base de la xarxa generada per ells.

Definició 2.2. *Siguin $\Lambda \in \mathbb{R}^n$ una xarxa, i $B = \{b_1, ..., b_n\}$ una base. Anomenem* Paral·lelepípede fonamental *al conjunt*

$$\mathcal{P}(B) = \left\{ \sum_{i=1}^{m} x_i b_i \mid x_i \in [-1/2, 1/2) \subset \mathbb{R} \right\}.$$

L'únic punt de la xarxa contingut en el paral·lelepípede fonamental és el zero, com queda formalitzat en el lema següent.

Lema 2.3. *Sigui Λ una xarxa de dimensió n i $b_1, ..., b_n \in \Lambda$ vectors linealment independents de $\mathbb{R}^n$. Aleshores $b_1, ..., b_n$ són una base de Λ si i només si $\mathcal{P}(b_1, ..., b_n) \cap \Lambda = \{0\}$.*

Definició 2.4. *Dues bases B_1 i B_2 de $\mathbb{R}^n$ són equivalents si $B_2 = B_1 U$ per alguna matriu unimodular U. Dues bases equivalents generen la mateixa xarxa.*

Definició 2.5. *Sigui Λ una xarxa de dimensió n. El determinant de dues bases qualssevol de Λ només difereix, potser, en el signe, i el seu valor absolut coincideix amb la mesura de Lebesgue del paral·lelepípede fonamental en qualsevol base. Aquest valor absolut s'anomena determinant de la xarxa.*

Definició 2.6. *Sigui Λ una xarxa de dimensió n i B una base seva. Per $t \in \mathbb{R}^n$ definim $t \bmod B$ com l'únic vector $t' \in \mathcal{P}(B)$ tal que $t - t' \in \Lambda$.*

Lema 2.7. *Podem calcular $t \bmod B$ com*

$$t \bmod B = t - B \cdot \lfloor B^{-1} \cdot t \rceil,$$

on $\lfloor x \rceil$ significa el vector de coordenades enteres més proper a x.

Demostració. Posem $t' = t - B \cdot \lfloor B^{-1} \cdot t \rceil$. Aleshores, $t - t' = B \cdot \lfloor B^{-1} \cdot t \rceil \in \Lambda$, ja que $\lfloor B^{-1} \cdot t \rceil \in \mathbb{Z}^n$. Sigui ara $x \in \mathbb{R}^n$ tal que $t = Bx$. Aleshores $t' = Bx - B \cdot \lfloor B^{-1} \cdot Bx \rceil = Bx - B\lfloor x \rceil = B \cdot (x - \lfloor x \rceil) \in \mathcal{P}(B)$. □

Per a un ús posterior, donem el radi de la bola més gran circumscrita per un paral·lelepípede fonamental.

Lema 2.8. *Sigui B una base de la xarxa Λ. El radi de la bola més gran centrada al zero i continguda en $\mathcal{P}(B)$ és $r = 1/(2\|(B^{-1})^T\|)$, on $\|M\|$ és el màxim de les normes euclidianes de les columnes de M.*

2.1 Vectors curts

Una de les propietats bàsiques associades a una xarxa és la llargada del vector no nul més curt (el zero està contingut en totes les xarxes). Denotarem aquesta llargada λ_1. Una altra forma de veure el vector més curt és com el mínim r tal que els vectors de la xarxa continguts en una bola de radi r generen un espai vectorial de dimensió ≥ 1. D'aquesta manera podem generalitzar el concepte.

Definició 2.9. *Sigui Λ una xarxa de dimensió n i $\overline{\boldsymbol{B}}(0,r) = \{x \in \mathbb{R}^n \mid \|x\| \leq r\}$. Per a $i \in \{1,...,n\}$ definim el* i-èsim mínim successiu

$$\lambda_i(\Lambda) = \inf\left\{r \mid \dim\left(\langle \Lambda \cup \overline{\boldsymbol{B}}(0,r)\rangle\right) \geq i\right\},$$

on $\langle \Lambda \cup \overline{\boldsymbol{B}}(0,r)\rangle$ és el subespai vectorial generat per $\Lambda \cup \overline{\boldsymbol{B}}(0,r)$.

Podem donar una cota inferior de λ_1 a partir del procés d'ortogonalització de Gram-Schmidt.

Definició 2.10. *Donats n vectors linealment independents $b_1,...,b_n$, definim l'ortogonalització de Gram-Schmidt com la seqüència de vectors $\tilde{b}_1,...,\tilde{b}_n$*

$$\tilde{b}_i = b_i - \sum_{j=1}^{i-1} \mu_{i,j}\tilde{b}_j, \text{ on } \mu_{i,j} = \frac{\langle b_i, \tilde{b}_j\rangle}{\langle \tilde{b}_j, \tilde{b}_j\rangle}.$$

Teorema 2.11. *Siguin B una base d'una xarxa de dimensió n i $\tilde{B}$ la seva ortogonalització de Gram-Schmidt. Aleshores*

$$\lambda_1(\mathcal{L}(B)) \geq \min_{i=1,...,n} \left\|\tilde{b}_i\right\| \geq 0.$$

Ara ens interessa trobar una cota superior per a λ_1.

Teorema 2.12. *(Minkowski) Siguin $\Lambda \subseteq \mathbb{R}^n$ una xarxa de dimensió n i $S \subseteq \mathbb{R}^n$ un subconjunt mesurable amb vol$(S) \geq \det(\Lambda)$. Aleshores existeixen dos punts diferents $z_1, z_2 \in S$ tal que $z_1 - z_2 \in \Lambda$.*

Corol·lari 2.13. *Sigui Λ una xarxa de dimensió n. Per a qualsevol conjunt S centralment simètric i convex, si vol$(S) > 2^n \det(\Lambda)$, aleshores S conté un punt no nul de Λ.*

A partir d'aquí podem donar les cotes següents.

Teorema 2.14. *(Minkowski) Sigui Λ una xarxa de dimensió n. Aleshores*

$$\lambda_1(\Lambda) \leq \sqrt{n}(\det\Lambda)^{1/n},$$

$$\left(\prod_{i=1}^{n} \lambda_i(\Lambda)\right)^{1/n} \leq \sqrt{n}(\det\Lambda)^{1/n}.$$

2.2 Problemes computacionals

Un dels principals problemes computacionals en una xarxa és trobar el vector no nul més curt. El *Shortest Vector Problem*, o SVP, es presenta en les variants següents.

1. SVP de cerca: Donada una base B d'una xarxa, trobar un $v \in \mathcal{L}(B)$ tal que $\|v\| = \lambda_1(\mathcal{L}(B))$.
2. SVP d'optimització: Donada una base B, trobar $\lambda_1(\mathcal{L}(B))$.
3. SVP de decisió: Donats una base B i un $r \in \mathbb{Q}$, determinar si $\lambda_1(\mathcal{L}(B)) \leq r$.

Aquests tres problemes tenen unes respectives variants aproximades, és a dir, en comptes de cercar el vector més curt en cercarem una aproximació. El factor d'aproximació ve donat per un paràmetre $\gamma \geq 1$.

1. SVP$_\gamma$ de cerca: Donada una base B d'una xarxa, trobar $v \in \mathcal{L}(B)$ tal que $v \neq 0$ i $\|v\| \leq \gamma \cdot \lambda_1(\mathcal{L}(B))$.
2. SVP$_\gamma$ d'optimització: Donada una base B, trobar d tal que $d \leq \lambda_1(\mathcal{L}(B)) \leq \gamma \cdot d$.
3. SVP$_\gamma$ de promesa (també anomenada GapSVP$_\gamma$): Una instància del problema ve donada per una parella (B,r), on B és una base i $r \in \mathbb{Q}$. En les instàncies positives, $\lambda_1(\mathcal{L}(B)) \leq r$. En les instàncies negatives, $\lambda_1(\mathcal{L}(B)) > \gamma \cdot r$.

Un altre problema fonamental en xarxes és trobar el vector més pròxim a un punt donat. S'anomena CVP (*Closest Vector Problem*) i de forma anàloga al SVP es presenta en diferents variants de cerca, de decisió o amb factors d'aproximació. Un cas particular del CVP és el BDDP (*Bounded distance decoding problem*), que consisteix a trobar el vector més pròxim sabent que és únic, és a dir, la distància del punt al vector més pròxim de la xarxa és estrictament menor que $\lambda_1(\Lambda)$. Un punt notable de tots aquests problemes en xarxes és que no s'ha trobat i es conjectura que no existeix cap algorisme en temps polinòmic, ni clàssic ni quàntic, que els resolgui en el cas general. La presumpta resistència d'aquests problemes a la construcció eventual d'un ordinador quàntic funcional és una qualitat interessant a l'hora de justificar la criptografia basada en xarxes.

2.3 Distribucions gaussianes

Definició 2.15. *Sigui $x \in \mathbb{R}^n$. Donats els paràmetres $s \in \mathbb{R}, s > 0$ i $c \in \mathbb{R}^n$, definim $\rho_{s,c} = e^{-\pi \frac{\|x-c\|^2}{s^2}}$. Si $c = 0$ simplifiquem la notació: $\rho_s = \rho_{s,c}$. Si $S \subset \mathbb{R}$ és un subconjunt numerable, aleshores $\rho_{s,c}(S) = \sum_{x \in S} \rho_{s,c}(x)$.*

Definició 2.16. *Sigui Λ una xarxa en $\mathbb{R}^n$. Per a $r > 0$ definim la* distribució gaussiana discreta *en Λ i centrada al zero*

$$D_{\Lambda,r}(x) = \frac{\rho_r(x)}{\rho_r(\Lambda)}, \forall x \in \Lambda.$$

A continuació definim el concepte de *paràmetre allisador (smoothing parameter).* Està relacionat amb la distribució gaussiana discreta i va ser introduït per Daniele Micciancio i Oded Regev [15].

Definició 2.17. *Sigui Λ una xarxa en $\mathbb{R}^n$ i $\epsilon > 0$ real. Aleshores es defineix el* paràmetre allisador *de Λ respecte de ϵ com*

$$\eta_\epsilon(\Lambda) = \min\left\{s : \rho_{1/s}(\Lambda^* \setminus \{0\}) \leq \epsilon\right\} = \min\left\{s : \rho_{1/s}(\Lambda^*) \leq 1 + \epsilon\right\}.$$

Informalment, el paràmetre allisador indica a partir de quina desviació estàndard una distribució gaussiana s'apropa a la uniforme. Aquest concepte queda formalitzat en el lema següent.

Lema 2.18. *Sigui $\Lambda = \mathcal{L}(\boldsymbol{B})$ una xarxa de dimensió n i $\boldsymbol{B}$ una base. Per a qualssevol $s > 0$ i $\boldsymbol{c} \in \mathbb{R}^n$ la distància estadística entre $D_{s,\boldsymbol{c}} \bmod \mathcal{P}(\boldsymbol{B})$ i la distribució uniforme sobre $\mathcal{P}(\boldsymbol{B})$ és com a molt $\frac{1}{2}\rho_{1/s}(\Lambda^* \setminus \{0\})$. En particular, per a cada $\epsilon > 0$ i qualsevol $s \geq \eta_\epsilon(\Lambda)$, la distància estadística satisfà que*

$$\Delta(D_{s,\boldsymbol{c}} \bmod \mathcal{P}(\boldsymbol{B}), U(\mathcal{P}(\boldsymbol{B}))) \leq \epsilon/2.$$

Veiem com es relaciona el paràmetre allisador amb els vectors més curts.

Lema 2.19. *Sigui Λ una xarxa de $\mathbb{R}^n$. Aleshores $\eta_\epsilon(\Lambda) \leq \sqrt{n}/\lambda_1(\Lambda^*)$, on $\epsilon = 2^{-n}$.*

Lema 2.20. *Siguin Λ una xarxa de $\mathbb{R}^n$ i $\epsilon > 0$ real. Aleshores*

$$\eta_\epsilon(\Lambda) \leq \sqrt{\frac{\ln(2n(1 + 1/\epsilon))}{\pi}} \cdot \lambda_n(\Lambda).$$

La demostració d'aquests últims lemes és excessivament llarga per incorporar-la aquí. Es pot trobar a Micciancio i Regev [15].
Recordem que per a una funció $f : \mathbb{R}^n \to \mathbb{R}$, amb les condicions adequades, es defineix la transformada de Fourier $\widehat{f} : \mathbb{R}^n \to \mathbb{R}$ com

$$\widehat{f}(y) = \int_{\mathbb{R}^n} f(x) e^{-2\pi i \langle x,y \rangle} dx.$$

Teorema 2.21 (Fórmula de la suma de Poisson)**.** *Siguin $\lambda \subset \mathbb{R}$ una xarxa i $f : \mathbb{R}^n \to \mathbb{R}$ una funció. Se satisfà que*

$$\sum_{x \in \Lambda} f(x) = \frac{1}{\det(\Lambda)} \sum_{y \in \Lambda^*} \widehat{f}(y).$$

Lema 2.22. *Per a qualssevol xarxa Λ, $c \in \mathbb{R}^n$, $\epsilon > 0$ i $r \geq \eta_\epsilon(\Lambda)$ se satisfà que*

$$\rho_r(\Lambda + c) \in r^n \det(\Lambda^*)(1 \pm \epsilon).$$

Demostració. Utilitzant la fórmula de la suma de Poisson i la suposició que $\rho_{1/r}(\Lambda^* \setminus \{0\}) \leq \epsilon$ obtenim que

$$\rho_r(L + c) = \sum_{x \in \Lambda} \rho_r(x + c) = \sum_{x \in \Lambda} \rho_{r,-c}(x) = \det(\Lambda^*) \sum_{y \in \Lambda^*} \widehat{\rho_{r,-c}}(y)$$

$$= r^n \det \Lambda^* \sum_{y \in \Lambda^*} e^{2\pi i \langle c, y \rangle} \rho_{1/r}(y) = r^n \det(\Lambda^*)(1 \pm \epsilon).$$

□

Lema 2.23. *Siguin Λ una xarxa, $z, u \in \mathbb{R}^n$ vectors i $r, \alpha > 0$ reals. Suposem que $1/\sqrt{1/r^2 + (\|z\|/\alpha)^2} \geq \eta_\epsilon(\Lambda)$ per a algun $\epsilon < \frac{1}{2}$. Siguin $v \leftarrow D_{\Lambda+u,r}$ i e amb distribució normal centrada al zero i de desviació estàndard $\alpha/(\sqrt{2\pi})$. Aleshores la distribució de $\langle z, v \rangle + e$ està dins d'una distància estadística de 4ϵ d'una variable aleatòria normal centrada al zero i amb desviació estàndard $\sqrt{(r\|z\|)^2 + \alpha^2}/\sqrt{2\pi}$. En particular, com que la distància estadística no pot augmentar aplicant una funció, la distribució de $\langle z, v \rangle + e \bmod 1$ està a una distància estadística màxima de 4ϵ de $\Psi_{\sqrt{r\|z\|)^2+\alpha^2}}$.*

La demostració d'aquest lema es pot trobar a Regev [19].

2.4 Xarxes ideals

Sigui $f \in \mathbb{Z}[X]$ un polinomi mònic irreductible de grau n. Considerem l'anell quocient $R = \mathbb{Z}[X]/(f)$. Aquest anell és un grup abelià lliure de dimensió n i el conjunt de classes $(1, X, X^2, ..., X^{n-1})$ n'és una $\mathbb{Z}$-base. La identificació dels elements de R com a vectors de coordenades en aquesta base s'anomena *immersió per coeficients*. Tenim que R és isomorf, com a grup additiu, a la xarxa $\mathbb{Z}^n \subset \mathbb{R}^n$. Siguin $\mathbf{v} \in R$ i $(\mathbf{v})$ l'ideal principal generat per $\mathbf{v}$. Anomenem *base de rotació* de la xarxa ideal $(\mathbf{v})$ a $\{\mathbf{v}_i = \mathbf{v} \cdot x^i \mod f(x) : i \in [0, n-1]\}$. L'ideal $(\mathbf{v})$ i la xarxa generada per la seva base de rotació són additivament isomorfs. Les xarxes ideals són, per tant, aquelles xarxes a les quals se'ls associa un ideal en un cert anell. Una definició més general és la següent.

Definició 2.24. *Siguin R un anell isomorf, com a grup additiu, a $\mathbb{Z}^n$ i $\sigma : R \to \mathbb{R}^n$ una immersió additiva. Sigui $I \subseteq R$ un ideal. Aleshores $\sigma(I) \subseteq \mathbb{R}^n$ és la xarxa ideal associada a l'ideal I.*

En aquesta secció ens limitem a la immersió per coeficients ja que és la que s'utilitza a l'esquema de Gentry. Al capítol 4 desenvolupem l'anomenada *immersió canònica*, per tal d'estudiar el problema RLWE.

3 L'esquema de Gentry

En la seva tesi doctoral, publicada al 2009, Gentry [8] fa la primera construcció teòrica d'un sistema criptogràfic totalment homomòrfic. En aquest capítol en presentem una síntesi.

3.1 Esquema base

Sigui $R = \mathbb{Z}[x]/(f(x))$, on $f(x)$ és un polinomi mònic de grau n. La immersió $\sigma : R \to \mathbb{R}^n$ utilitzada és la immersió per coeficients, és a dir, prenent $(1, x, x^2, ..., x^{n-1})$ com a base de R s'identifiquen els elements de R amb els seus vectors de coordenades en aquesta base. Una immersió de R en $\mathbb{R}^n$ ens permet, entre d'altres coses, associar propietats geomètriques a elements de R, com ara la norma euclidiana. Siguin $I \subset R$ un ideal de R i $\mathbf{B}_I$ una base de la xarxa ideal associada a I.

Notació 2. *Denotem per* $R \bmod \boldsymbol{B}_I$ *el conjunt de representants distingits de* R/I *segons la base* $\boldsymbol{B}_I$*.*

$$R \bmod \boldsymbol{B}_I = \{t \bmod \boldsymbol{B}_I \mid t \in R\} = R \cap \mathcal{P}(\boldsymbol{B}_I).$$

Abusant de la notació també direm $R \bmod \boldsymbol{B}_I$ *per referir-nos a l'anell* R/I *juntament amb el conjunt de representants distingits* $R \bmod \boldsymbol{B}_I$*.*

L'objectiu d'associar una base $\mathbf{B}_I$ d'una xarxa a un ideal I de R és precisament la de definir un conjunt de representants distingits. Dues bases $\mathbf{B}_I$ i $\mathbf{B}'_I$ diferents de la xarxa ideal associada a I defineixen conjunts de representants diferents de l'anell quocient R/I.
Abans de definir els procediments principals de l'esquema veiem-ne uns de previs.

- $\mathsf{IdealGen}(R, \mathbf{B}_I)$: Pren aleatòriament un ideal J coprimer amb I (és a dir, $I + J = R$) i en dóna, també aleatòriament, dues bases $\mathbf{B}_J^{pk}$ i $\mathbf{B}_J^{sk}$.
- $\mathsf{Samp}(\mathbf{B}_I, x)$: És una variable aleatòria que pren la base $\mathbf{B}_I$ i un $x \in R$ i retorna, d'acord amb una certa distribució de probabilitat, valors de la classe lateral $x + I$.

El conjunt de textos plans $\mathcal{P}$ és un subconjunt de $R \bmod \mathbf{B}_I$. El conjunt de textos xifrats $\mathcal{X}$ és un subconjunt de R. Els circuits avaluables $\mathcal{C}_\mathcal{E}$ són circuits aritmètics en l'anell R/I. Els algorismes que conformen l'esquema són els següents.

- $\mathsf{KeyGen}(R, \mathbf{B}_I)$: Calcula $(\mathbf{B}_J^{sk}, \mathbf{B}_J^{pk} \leftarrow \mathsf{IdealGen}(R, \mathbf{B}_I)$. Retorna les claus pública $pk = (R, \mathbf{B}_I, \mathbf{B}_J^{pk}, Samp)$ i privada $sk = (pk, \mathbf{B}_J^{sk})$.
- $\mathsf{Encrypt}(pk, \pi)$: Pren una clau pública pk i un $\pi \in \mathcal{P}$. Posa $\psi' \leftarrow \mathsf{Samp}(\mathbf{B}_I, \pi)$ i retorna $\psi = \psi' \bmod \mathbf{B}_J^{pk}$. Notem que tot text xifrat és de la forma $\psi = \pi + i + j$, on $i \in I, j \in J$.
- $\mathsf{Decrypt}(sk, \psi)$: Pren una clau secreta sk i un text xifrat ψ. Retorna $\pi = (\psi \bmod \mathbf{B}_J^{sk}) \bmod \mathbf{B}_I$.
- $\mathsf{Evaluate}(pk, C, \Psi)$: Pren la clau pública, un circuit $C \in \mathcal{C}_\mathcal{E}$ i una llista ordenada de textos xifrats Ψ. Les portes s'avaluen de la manera següent.
  - $\mathsf{Add}(pk, \psi_1, \psi_2)$: Dóna $\psi_1 + \psi_2 \bmod \mathbf{B}_J^{pk}$.
  - $\mathsf{Mult}(pk, \psi_1, \psi_2)$: Dóna $\psi_1 \times \psi_2 \bmod \mathbf{B}_J^{pk}$.

És important tenir en compte que per a que el desxifrat es produeixi correctament, la base $\mathbf{B}_J^{sk}$ ha de ser compatible amb la distribució Samp per assegurar que $\pi + i$ és sempre el representant distingit de $\pi + i + J$ respecte $\mathbf{B}_J^{sk}$. És a dir, la imatge de Samp ha d'estar continguda en $\mathcal{P}(\mathbf{B}_J^{sk})$.
Veiem ara que l'esquema és correcte.

Definició 3.1. *Anomenem* X_{Enc} *a la imatge de* Samp*. Anomenem* X_{Dec} *a* $R \bmod \boldsymbol{B}_J^{sk}$*.*

Definició 3.2. *Denotem* r_{Enc} *el valor més petit tal que* $X_{Enc} \subseteq \mathcal{B}(r_{Enc})$ *i* r_{Dec} *el valor més gran tal que* $X_{Dec} \supseteq \mathcal{B}(r_{Dec})$*, on* $\mathcal{B}(r)$ *denota una bola de radi* r*.*

Tal i com es veu al lema 2.8 i tenint en compte que $X_{Dec} = R \bmod \mathbf{B}_J^{sk}$, tenim que $r_{Dec} = 1/(2\|\mathbf{B}_J^{sk}\|)$.

Definició 3.3. *Sigui C un circuit aritmètic en $R \bmod \boldsymbol{B}_I$. El circuit resultant de substituir totes les portes en $R \bmod \boldsymbol{B}_I$ per les portes equivalents en R l'anomenem* circuit generalitzat *de C i el denotem per $g(C)$. (És a dir, substituir les portes suma i multiplicació en $R \bmod \boldsymbol{B}_I$ per sumes i multiplicacions en R).*

Definició 3.4. *Diem que un circuit $C : (R \bmod \boldsymbol{B}_I)^n \to (R \bmod \boldsymbol{B}_I)^m$ és un circuit* permès *si $\forall (x_1, ..., x_n) \in \mathcal{B}(r_{Enc})^n$, se satisfà que $g(C)(x_1, ..., x_n) \in \mathcal{B}(r_{Dec})^m$. Denotem el conjunt de circuits permesos per $\mathcal{C}'_{\mathcal{E}}$.*

És a dir, els circuits permesos són aquells que, un cop generalitzats, les sortides estan en $\mathcal{B}(r_{Dec})$ si les entrades estan en $\mathcal{B}(r_{Enc})$.

Definició 3.5. *Diem que ψ és un* text xifrat vàlid *respecte de la clau pública pk i el conjunt de circuits permesos $\mathcal{C}_{\mathcal{E}} \subseteq \mathcal{C}'_{\mathcal{E}}$ si $\exists C \in \mathcal{C}_{\mathcal{E}}$ tal que $\psi = \mathsf{Evaluate}(pk, C, \Psi)$, on Ψ és una llista ordenada d'elements de la imatge de $\mathsf{Encrypt}(pk, \cdot)$.*

Ara demostrem que l'esquema és correcte per a circuits en $\mathcal{C}_{\mathcal{E}}$.

Teorema 3.6. *Sigui $\mathcal{C}_{\mathcal{E}}$ un conjunt de circuits permesos que conté el circuit identitat. Aleshores $\mathcal{E}$ és correcte, és a dir, $\mathsf{Decrypt}$ desxifra correctament textos xifrats vàlids.*

Demostració. Sigui $\Psi = \langle \psi_1, ..., \psi_t \rangle$ una llista ordenada de textos xifrats, on $\psi_k = \pi_k + i_k + j_k, \pi_k \in \mathcal{P}, i_k \in I, j_k \in J$ i $\pi_k + i_k \in X_{Enc}$. Aleshores

$$\mathsf{Evaluate}(pk, C, \Psi) = g(C)(\Psi) \bmod B_J^{sk} \in g(C)(\pi_1 + k_1, ..., \pi_t + i_t) + J.$$

Si $C \in \mathcal{C}'_{\mathcal{E}}$, $g(C)(\Psi) \in (X_{Dec})^n$ i per tant

$$\begin{aligned}\mathsf{Decrypt}(sk, \mathsf{Evaluate}(pk, C, \Psi)) &= g(C)(\pi_1 + i_1, ..., \pi_t + i_t) \bmod B_I \\ &= g(C)(\pi_1, ..., \pi_t) \bmod B_I = C(\pi_1, ..., \pi_t).\end{aligned}$$

□

L'objectiu d'utilitzar r_{Enc} i r_{Dec} en comptes de X_{Enc} i X_{Dec} és poder fer una anàlisi geomètrica de la profunditat dels circuits que es podran avaluar.

Definició 3.7. *Sigui $\|\cdot\|$ una norma en R. Anomenem el* factor d'expansió *de R per aquesta norma a*

$$\gamma_R := \max\left\{ \frac{\|a \times b\|}{\|a\|\|b\|} \mid a, b \in R \right\}.$$

En el nostre cas sempre treballarem amb la norma euclidiana induïda a través de la immersió per coeficients. A partir d'aquí podem veure la profunditat dels circuits que es podran avaluar correctament.

Teorema 3.8. *Siguin $r_E \geq 1$ i r_D nombres reals tals que $p = \log\log r_D - \log\log(\gamma_R \cdot r_E) > 1$ i C un circuit aritmètic en R de profunditat com a molt p on les portes suma tenen grau màxim γ_R i les portes multiplicació tenen grau màxim 2. Aleshores, $\forall (x_1, ..., x_t) \in \mathcal{B}(r_E)^t$, $C(x_1, ..., x_t) \in \mathcal{B}(r_D)^s$.*

Demostració. Sigui C un circuit de profunditat d i sigui r_i el màxim de les normes euclidianes de les entrades de les portes de nivell i. Aleshores, per la desigualtat triangular, tenim que les sortides de les portes suma de nivell i estan acotades per $\gamma_R \cdot r_i$, i la de les portes producte per $\gamma_R \cdot r_i^2$. En qualsevol cas tenim que $r_{i+1} \leq \gamma_R \cdot r_i^2$ i per tant $r_d \leq (\gamma_R \cdot r_E)^{2^d}$. Com que volem $r_d \leq r_D$, aleshores aïllem d tal que $r_D \geq (\gamma_R \cdot r_E)^{2^d}$ i obtenim que $d \leq \log\log r_D - \log\log(\gamma_R \cdot r_E)$. Per tant, si d compleix aquesta desigualtat, per entrades acotades per r_E obtenim sortides acotades per r_D. □

En particular, l'esquema $\mathcal{E}$ avalua correctament circuits (amb les anteriors condicions) de profunditat fins a $\log\log r_{\text{Dec}} - \log\log(\gamma_R \cdot r_{\text{Enc}})$. Per tal de maximitzar aquesta profunditat podem intentar o minimitzar γ_R i r_{Enc} o maximitzar r_{Dec}, sempre tenint en compte les conseqüències que poden tenir aquests canvis en la seguretat de l'esquema.

3.2 Modificacions a l'esquema base

Per tal de disminuir la profunditat del circuit de desxifrat s'ha de fer algunes modificacions a l'esquema. La primera modificació s'empara en el lema següent.

Lema 3.9. *Sigui $\boldsymbol{B}_J^{sk}$ una base secreta que desxifra correctament pel paràmetre r_{Dec}. A partir de $\boldsymbol{B}_J^{sk}$ i $\boldsymbol{B}_I$ podem calcular en temps polinòmic un vector $\boldsymbol{v}_J^{sk} \in J^{-1}$ tal que la base de rotació de $1/\boldsymbol{v}_J^{sk}$ circumscriu una bola de radi almenys $r_{Dec}/(n^{2.5}\|f\|\|\boldsymbol{B}_I\|)$. En particular, si ψ és un text xifrat de la forma $\pi + i + j$, pel text pla π, $i \in I$, $j \in J$ i $\pi + i \in \mathbb{B}(r_{Dec}/(n^{2.5}\|f\|\|\boldsymbol{B}_I\|)$, aleshores $\pi = \psi - \lfloor \boldsymbol{v}_j^{sk} \cdot \psi \rceil \bmod B_I$.*

Per tant, la modificació 1 simplifica l'equació de desxifrat de

$$\pi = \psi - \mathbf{B}_J^{sk} \cdot \lfloor (\mathbf{B}_J^{sk})^{-1} \cdot \psi \rceil \bmod \mathbf{B}_I$$

a

$$\pi = \psi - \lfloor \mathbf{v}_J^{sk} \times \psi \rceil \bmod \mathbf{B}_I.$$

La modificació 2 consisteix a redefinir el conjunt de circuits permesos $\mathcal{C}_\mathcal{E}$ reemplaçant $\mathcal{B}(r_{\text{Dec}})$ per $\mathcal{B}(r_{\text{Dec}}/2)$. D'aquesta manera assegurem que els textos xifrats estaran més aprop de la xarxa J del que és necessari i així reduir la complexitat del pas d'aproximació del desxifrat.

Lema 3.10. *Si ψ és un text xifrat vàlid després de la modificació 2, aleshores cada coeficient de $(\boldsymbol{B}_J^{sk})^{-1} \cdot \psi$ està a menys de $1/4$ d'un enter.*

3.3 Simplificació del circuit de desxifrat

L'equació general de desxifrat és

$$(\psi - \mathbf{B}_J^{sk_1} \cdot \lfloor \mathbf{B}_J^{sk_2} \cdot \psi \rceil) \bmod \mathbf{B}_I,$$

on $\psi \in \mathbb{Z}^n$, $\mathbf{B}_J^{sk_1} \in \mathbb{Z}^{n\times n}$, $\mathbf{B}_J^{sk_2} \in \mathbb{Q}^{n\times n}$ i $\mathbf{B}_I$ és la base d'un ideal I de $R = \mathbb{Z}[X]/(f)$. Per la modificació 2 sabem que els coeficients de $\mathbf{B}_J^{sk_2} \cdot \psi$ estan a una distància màxima de $1/4$ d'un enter. La modificació 1 assegura que $\mathbf{B}_J^{sk_1}$ és la matriu identitat i $\mathbf{B}_J^{sk_2}$ és una matriu de rotació. Dividirem el càlcul en les parts següents.

1. Generem n vectors $\mathbf{x}_1, ..., \mathbf{x}_n$ amb suma $\mathbf{B}_J^{sk_2} \cdot \psi$,
2. Generem vectors enters $\mathbf{y}_1, ..., \mathbf{y}_{n+1}$ amb suma $\lfloor \sum \mathbf{x}_i \rceil$,
3. Calculem $\pi = \psi - \mathbf{B}_J^{sk_1} \cdot (\sum \mathbf{y}_i) \bmod \mathbf{B}_I$.

No tenim l'espai per desenvolupar tot l'anàlisi de la complexitat d'aquest circuit, però la conclusió és que l'esquema no és *bootstrappable*. A l'hora de calcular els vectors $\mathbf{x}_i$ necessitem suficients decimals de precisió per assegurar la correcció del circuit i la profunditat acaba sent $O(\log n + \log\log r_{\text{Dec}})$, que és superior a la que l'esquema pot avaluar. S'ha de simplificar el circuit de desxifrat. La idea consisteix a expandir el procediment de xifrat de tal manera que els textos xifrats continguin una "pista" que permeti desxifrar-los amb un circuit de menys profunditat, sense que això afecti a la seguretat de l'esquema.
Sigui $\gamma(n) \in \mathbb{N}$ un paràmetre polinòmic en n i $\gamma'(n) < \gamma(n)$. Considerem els vectors $\mathbf{t}_1, ..., \mathbf{t}_{\gamma(n)} \in J^{-1}$ tals que existeix $S \in \{1, ..., \gamma(n)\}$ de cardinal $\gamma'(n)$ amb $\sum_{i\in S} \mathbf{t}_i \equiv \mathbf{v}_J^{sk} \bmod I$. Aleshores, en el moment d'encriptar, a part del text xifrat ψ es calcularà $\mathbf{c}_i = \mathbf{t}_i \bmod \mathbf{B}_I$ per a tot $i < \gamma(n)$. D'aquesta manera l'equació de desxifrat queda

$$\pi = \psi - \lfloor \sum_{i\in S} \mathbf{c}_i \rceil \bmod \mathbf{B}_I.$$

És important tenir en compte que malgrat que aquesta transformació augmenta la complexitat del circuit de desxifrat, en redueix la profunditat. La seguretat de la transformació es basa en el problema de trobar, donat un conjunt, un subconjunt on la suma dels elements sigui un valor determinat. Aquest problema no admet una resolució en temps polinòmic en el cas general.

3.4 Implementacions

Implementar l'esquema de Gentry no és senzill. No només és molt ineficient de forma intrínseca, sinó que l'esquema és més un *proof of concept* que no pas quelcom destinat a ser implementat. Molts detalls no queden explícits. Existeix una implementació duta a terme per Gentry i Halevi [9].

3.5 Conclusions

L'esquema de de Gentry és la primera demostració de que és possible construir un sistema de xifrat completament homomòrfic. Malgrat que ha estat superat per esquemes més nous i eficients, no se'n pot denegar la importància històrica. Publicat al 2009, dóna el tret de sortida d'una nova branca de la criptografia que ha estat en recerca activa fins a l'actualitat.

4 Learning With Errors

El problema *Learning With Errors* (LWE) va ser introduït per Regev [19] al 2005 i ha trobat múltiples aplicacions en el món de la criptografia. El principal punt notable del LWE és que és tan difícil com els pitjors casos d'alguns problemes sobre xarxes, a partir del qual es conjectura que no admet resolució en temps polinòmic. En aquest capítol fem una breu descripció i estudi del problema LWE i de la seva variant en anells de polinomis RLWE. Indiquem que donat un anell R i un element $q \in R$ utilitzem la notació R_q per referir-nos a l'anell quocient R/qR.
El problema LWE consisteix a trobar un vector $\mathbf{s} \in \mathbb{Z}_q^n$ donada una seqüència d'equacions lineals aleatòries aproximades en $\mathbf{s}$. Per exemple, podríem tenir el sistema en $\mathbb{Z}_{17}$:

$$\begin{aligned}
14s_1 + 15s_2 + 5s_3 + 2s_4 &\approx 8 \mod 17,\\
13s_1 + 14s_2 + 14s_3 + 6s_4 &\approx 16 \mod 17,\\
6s_1 + 10s_2 + 13s_3 + 1s_4 &\approx 3 \mod 17,\\
10s_1 + 4s_2 + 2s_3 + 16s_4 &\approx 12 \mod 17,\\
9s_1 + 5s_2 + 9s_3 + 6s_4 &\approx 9 \mod 17,\\
3s_1 + 6s_2 + 4s_3 + 5s_4 &\approx 16 \mod 17,\\
&\vdots\\
6s_1 + 7s_2 + 16s_3 + 2s_4 &\approx 3 \mod 17,
\end{aligned}$$

on cada equació és correcta excepte per un petit factor additiu (per exemple ± 1) i volem recuperar $\mathbf{s}$. En aquest cas $\mathbf{s} = (0, 13, 9, 11)$. Si no fos per l'error podríem calcular $\mathbf{s}$ fàcilment amb n equacions independents mitjançant l'algorisme d'eliminació de Gauss. Ara bé, l'error complica significativament el problema.
El nom, *Learning With Errors*, prové del fet que aquest problema es va plantejar en el context del *machine learning*, o aprenentatge automàtic, ja que treballa sobre la qüestió de quina informació es pot extreure a partir de dades aproximades. Definim formalment el problema.

Definició 4.1. *Siguin $n \geq 1, q \geq 2$ enters i sigui χ una distribució de probabilitat en $\mathbb{Z}_q$. Siguin $\boldsymbol{a} \in \mathbb{Z}_q^n$ escollit uniformement, $e \in \mathbb{Z}_q$ escollit segons la distribució χ i $A_{s,\chi} = (\boldsymbol{a}, \langle \boldsymbol{a}, \boldsymbol{s} \rangle + e) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$. Diem que un algorisme soluciona el LWE amb mòdul q i distribució d'error χ si, per a tot $\boldsymbol{s} \in \mathbb{Z}_q^n$ i donat un nombre arbitrari de mostres de la distribució $A_{s,\chi}$, pot recuperar $\boldsymbol{s}$ amb probabilitat tendint a 1.*

Una forma ingènua de solucionar el LWE consistiria a fer repetides mostres de la distribució $A_{s,\chi}$ fins a obtenir, per exemple una parella $(\mathbf{a}, b)$ amb $\mathbf{a} = (1, 0, ..., 0)$, a partir de la qual podem recuperar s_1, i igual amb els altres components de $\mathbf{s}$. La probabilitat d'obtenir una equació d'aquest tipus, donat que $\mathbf{a}$ és uniforme en $\mathbb{Z}_q^n$, és q^{-n}, i per tant l'algorisme necessitarà una mostra de $2^{O(n \log n)}$ equacions. Els millors algorismes coneguts tenen una complexitat de $2^{O(n)}$. L'objectiu és ara analitzar la complexitat del LWE analitzant les implicacions que tindria l'existència d'un algorisme polinòmic de resolució.

4.1 Conceptes previs

Abans de poder analitzar la dificultat del problema LWE necessitem alguns conceptes previs.

Definició 4.2. *$\mathbb{T}$ és el tor unidimensional. El podem pensar com el grup additiu quocient $\mathbb{R}/\mathbb{Z}$ o com el grup multiplicatiu del cercle unitat en $\mathbb{C}$. Utilitzarem el segment $[0, 1)$ amb suma mòdul 1 com a conjunt de representats distingits del grup $\mathbb{R}/\mathbb{Z}$.*

Definició 4.3. *Diem que una probabilitat p és exponencialment propera a 1 respecte un paràmetre n si $p(n)$ és com a molt $1 - 2^{-\Omega(n)}$. Sovint ometrem dir que ho és respecte n si s'entén pel context.*

Definició 4.4. *Donats $p \in \mathbb{Z}$, $\boldsymbol{s} \in \mathbb{Z}_p^n$ i una funció de densitat de probabilitat Φ en $\mathbb{T}$ definim $A_{\boldsymbol{s},\Phi}$ com la distribució en $\mathbb{Z}_p^n \times \mathbb{T}$ obtinguda escollint uniformement un vector $\boldsymbol{a} \in \mathbb{Z}_p^n$, un $e \in \mathbb{T}$*

segons la distribució Φ i retornant $(\boldsymbol{a}, \langle \boldsymbol{a}, \boldsymbol{s}\rangle/p + e)$. Anàlogament al LWE discret, direm que un algorisme soluciona el $LWE_{p,\Phi}$ si per a tot $\boldsymbol{s} \in \mathbb{Z}_p^n$, donades mostres de $A_{\boldsymbol{s},\Phi}$, retorna $\boldsymbol{s}$ amb probabilitat exponencialment propera a 1.

Per analitzar la dificultat del LWE treballarem amb famílies de distribucions normals adaptades al cercle unitat, i per tant fàcilment adaptables a $\mathbb{Z}_p$.

Definició 4.5. *Anomenem la* distribució normal embolicada *a la distribució sobre el $\mathbb{T}$ amb densitat*

$$f_{WN}(\theta; \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \sum_{k=-\infty}^{\infty} \exp\left[\frac{-(\theta - \mu + 2\pi k)^2}{2\sigma^2}\right],$$

on μ és la mitjana, σ és la desviació estàndard i $\theta \in [-\pi, \pi)$ és l'angle del punt en el cercle unitat.

Definició 4.6. *Donat $\beta \in \mathbb{R}^+$ denotem Ψ_β a la distribució normal embolicada en $\mathbb{T}$ de mitjana zero i desviació estàndard $\frac{\beta}{\sqrt{2\pi}}$, traslladada al segment $[0, 1)$, que té densitat*

$$\forall r \in [0,1), \Psi_\beta(r) = \frac{1}{\beta} \sum_{k=-\infty}^{\infty} \exp\left[-\pi \left(\frac{r-k}{\beta}\right)^2\right].$$

Definició 4.7. *Donades dues funcions de densitat de probabilitat ϕ_1, ϕ_2 en $\mathbb{R}^n$ anomenem la* distància estadística *entre elles a*

$$\Delta(\phi_1, \phi_2) = \int_{\mathbb{R}^n} |\phi_1(x) - \phi_2(x)| dx.$$

Es pot veure fàcilment que amb aquesta definició la distància estadística està sempre en el rang $[0, 2]$. A més a més, se satisfà la desigualtat triangular ($\Delta(\phi_1, \phi_3) \leq \Delta(\phi_1, \phi_2) + \Delta(\phi_2, \phi_3)$).

Proposició 4.8. *Per a qualsevol $0 < \alpha < \beta \leq 2\alpha$ se satisfà que*

$$\Delta(\Psi_\alpha, \Psi_\beta) \leq 9\left(\frac{\beta}{\alpha} - 1\right).$$

Veiem ara que donada una distribució de probabilitat contínua en $\mathbb{T}$ podem calcular el seu equivalent discret en $\mathbb{Z}_p$.

Definició 4.9. *Siguin $\phi : \mathbb{T} \to \mathbb{R}^+$ una funció de densitat de probabilitat i $p \geq 1$ un enter. Es defineix el seu equivalent discret $\bar{\phi} : \mathbb{Z}_p \to \mathbb{R}^+$ com*

$$\bar{\phi}(i) = \int_{(i-1/2)/p}^{(i+1/2)/p} \phi(x) dx.$$

Definició 4.10. *Diem que una funció $\mu : \mathbb{N} \to \mathbb{R}$ és* negligible *si $\forall c \in \mathbb{N}, \exists n_0 \in \mathbb{N}$ tal que $\forall n \geq n_0, |\mu(n)| < n^{-c}$.*

Notació 3. *Denotem per $CVP_{\Lambda,r}$ al problema: donada una xarxa Λ i un $x \in \mathbb{R}^n$ a una distància màxima r de Λ, trobar el vector de Λ més pròxim a x.*

4.2 Dificultat

El principal teorema sobre la dificultat del LWE ens ve donat per Regev [19]. En donem la versió informal.

Teorema 4.11. *Siguin n, p enters positius i $\alpha \in (0, 1)$ tal que $\alpha p > 2\sqrt{n}$, i sigui χ_α una distribució gaussiana discreta en $\mathbb{Z}_p$ de mitjana 0 i desviació estàndard αp. Si existeix un algorisme que soluciona el LWE_{p,Ψ_α} donat un nombre polinòmic de mostres aleshores existeix un algorisme quàntic eficient que aproxima els problemes sobre xarxes GapSVP (problema de decisió del vector més curt) i SIVP (problema dels vectors independents més curts) amb un marge de $\tilde{O}(n/\alpha)$ en el pitjor dels casos.*

La contraposició d'aquest teorema és que, donada una tria de paràmetres adequada, si no existeix un algorisme quàntic eficient que solucioni els problemes SIVP o GapSVP en una xarxa, aleshores tampoc existeix un algorisme polinòmic que solucioni el problema Learning With Errors. És important entendre que el component quàntic del teorema és una debilitat, ja que només caldria trobar un algorisme quàntic pel SIVP o GapSVP per no poder demostrar la dificultat del LWE. Seria ideal que fos necessari un algorisme clàssic, que és una condició més forta, però demostrar-ho roman un problema obert.
La formalització i demostració del teorema és complexa i extensa ja que entra en el terreny dels algorismes quàntics. Aquí ens conformarem amb la part clàssica de l'argument.

Teorema 4.12. *Siguin $\epsilon = \epsilon(n)$ una funció negligible, $p = p(n) \geq 2$ un nombre enter i $\alpha = \alpha(n) \in (0,1)$ un nombre real. Si existeix un algorisme que resol el LWE_{p,Ψ_α} donat un nombre polinòmic de mostres, aleshores existeixen una constant $c > 0$ i un algorisme eficient que, donada una xarxa n-dimensional Λ, un $r > \sqrt{2}p\eta_\epsilon(\Lambda)$ i n^c mostres de $D_{\Lambda,r}$, soluciona el $CVP_{\Lambda^*,\alpha p/(\sqrt{2}r)}$, és a dir, donat un punt x dins d'una distància $\frac{\alpha p}{\sqrt{2}r}$ de Λ^*, troba el punt de Λ^* més pròxim a x.*

Per demostrar aquest teorema necessitem alguns resultats previs.

Notació 4. *Al llarg del desenvolupament que ve a continuació, fent abús de notació, denotarem per Λ tant a la xarxa com a una base qualsevol d'ella.*

Definició 4.13. *Siguin Λ una xarxa n-dimensional, $0 < d < \lambda_1(\Lambda)/2$ i $p \geq 2$ un nombre enter. Diem que un algorisme soluciona el $CVP^{(p)}_{L,d}$ si, donat un punt $x \in \mathbb{R}^n$ dins d'una distància d de Λ, retorna $\Lambda^{-1}\kappa_\Lambda(x) \bmod p \in \mathbb{Z}_p^n$, el vector de coeficients del vector de Λ més pròxim a x reduït mòdul p.*

Veiem que es pot reduir el $\mathrm{CVP}_{L,d}$ al $\mathrm{CVP}^{(p)}_{L,d}$.

Lema 4.14. *Siguin Λ una xarxa, $0 < d < \lambda_1(\Lambda)$ i $p \geq 2$ un nombre enter. Si tenim accés a un oracle que soluciona el $CVP^{(p)}_{L,d}$ aleshores existeix un algorisme eficient que soluciona el $CVP_{L,d}$.*

Demostració. Partim d'un punt x dins d'una distància d de Λ. Definim la seqüència $a_i = \Lambda^{-1}\kappa_\Lambda(x_i) \in \mathbb{Z}^n$, el vector de coeficients del punt de Λ més pròxim a x_i i $x_{i+1} = (x_i - \Lambda(a_i \bmod p))/p$, amb $x_1 = x$. Unint les dues successions obtenim que $a_{i+1} = (a_i - (a_i \bmod p))/p$. La distància de x_i a Λ és com a màxim d/p^i. Aquesta seqüència es pot calcular amb l'algorisme que soluciona el $\mathrm{CVP}^{(p)}_{L,d}$.
Després de n passos tenim un punt x_{n+1} a una distància menor que d/p^n de Λ. Podem aplicar un algorisme aproximat (el de Babai per exemple, en temps polinòmic) per trobar un punt Λa a una distància $2^n d/p^n \leq d < \lambda_1(\Lambda)/2$ de x_{n+1}. Per tant, Λa és el punt més pròxim a x_{n+1} i hem recuperat $a_{n+1} = a$. De forma recursiva podem recuperar tots els altres $a_n, ..., a_1$. Això completa l'algorisme ja que Λa_1 és el punt de Λ més pròxim a $x_1 = x$. □

Pel lema 2.19 i com que $r > \sqrt{2}p\eta_\epsilon(\Lambda)$ aleshores $d < \frac{\alpha p}{\sqrt{2}r} \leq \lambda_1(\Lambda^*)/2$. Per tant, per demostrar el teorema 4.12 només cal trobar un algorisme eficient pel $\mathrm{CVP}^{(p)}_{\Lambda^*,\alpha p/(\sqrt{2}r)}$, donades les condicions que es descriuen. Necessitem alguns resultats per veure això.

Lema 4.15. *Sigui $p = p(n) \leq 1$ un enter. Existeix un algorisme eficient que, donat $\boldsymbol{s}' \in \mathbb{Z}^n$ i mostres de $A_{\boldsymbol{s},\Psi_\alpha}$ per un $\boldsymbol{s}$ desconegut i $\alpha \in (0,1)$ retorna si $\boldsymbol{s}_1 = \boldsymbol{s}$ i és correcte amb probabilitat exponencialment propera a 1.*

Demostració. L'algorisme per realitzar el test estadístic és el següent. Sigui ξ la distribució en $\mathbb{T}$ obtinguda d'agafar una mostra $(\mathbf{a}, x) \leftarrow A_{\mathbf{s},\Psi_\alpha}$ i retornar $x - \langle \mathbf{a}, \mathbf{s}' \rangle / p \in \mathbb{T}$. S'agafen n mostres $y_1, ..., y_n$ de ξ i es calcula $z = \frac{1}{n}\sum_{i=1}^n \cos(2\pi y_i)$. Si $z > 0.02$ decideix que $\mathbf{s}' = \mathbf{s}$; alternativament decideix que $\mathbf{s}' \neq \mathbf{s}$.
Analitzem ara l'algorisme. Veiem que la distribució ξ es pot obtenir prenent una mostra $e \leftarrow \Psi_\alpha$, $\mathbf{a}$ uniformement en $\mathbb{Z}_p^n$ i retornant $e + \langle \mathbf{a}, \mathbf{s} - \mathbf{s}' \rangle / p \in \mathbb{T}$. Per tant, si $\mathbf{s} = \mathbf{s}'$, aleshores $\xi = \Psi_\alpha$. Alternativament, si $\mathbf{s} \neq \mathbf{s}'$, ξ té període $1/k$ per a algun nombre enter $k \geq 2$ ja que, si prenem un index j tal que $s_j \neq s'_j$, aleshores la distribució de $a_j(s_j - s'_j) \bmod p$ és periòdica de període $\gcd(p, s_j - s'_j) < p$ i per tant la distribució de $a_j(s_j - s'_j)/p \bmod 1$ és periòdica de període $1/k$ per a

algun $k \geq 2$.
El valor esperat d'una variable aleatòria Y distribuïda segons ξ és

$$\tilde{z} = E[\cos(2\pi Y)] = \int_0^1 \cos(2\pi y)\xi(y)dy.$$

Es pot veure que si $\xi = \Psi_\alpha$, aleshores $\tilde{z} = e^{-\pi\alpha^2}$, que és major que 0.04 per a $\alpha \in (0,1)$. Si ξ és periòdica de període $1/k$ amb $k \geq 2$ es pot veure que $\tilde{z} = 0$. Aplicant la cota de Chernoff obtenim que $|z - \tilde{z}| \leq 0.01$ amb probabilitat exponencialment propera a 1. □

Lema 4.16. *Siguin $p = p(n) \geq 2$ enter i $\alpha = \alpha(n) \in (0,1)$. Assumim que tenim accés a un oracle W que soluciona LWE_{p,Ψ_α} utilitzant un nombre polinòmic de mostres. Aleshores existeix un algorisme eficient W' que, donades mostres de $A_{\boldsymbol{s},\Psi_\beta}$ per a algun $\beta \leq \alpha$ desconegut, retorna $\boldsymbol{s}$ amb probabilitat exponencialment propera a 1.*

Demostració. Assumim que el nombre de mostres que necessita W està acotat per n^c, per a algun $c > 0$. Sigui $Z = \{k\alpha^2/n^{2c} \mid k \in \mathbb{Z}, 0 \leq k \leq n^{2c}\}$. Per a cada $\gamma \in Z$ l'algorisme W fa el següent n vegades. Pren n^c mostres de $A_{\mathbf{s},\Psi_\beta}$ i afegeix al segon element de cada mostra una mostra independent de la distribució $\Psi_{\sqrt{\gamma}}$, creant així n^c mostres de $A_{\mathbf{s},\Psi_{\sqrt{\beta^2+\gamma}}}$, ja que la suma de dues distribucions normals amb igual mitjana és la distribució amb variància suma de les variàncies. Ara s'aplica W, s'obté un candidat $\mathbf{s}'$ i utilitzant el lema 4.15 comprova si $\mathbf{s} = \mathbf{s}'$. En cas afirmatiu retorna $\mathbf{s}'$, en cas contrari continua.
Comprovem que W' troba $\mathbf{s}$ amb probabilitat exponencialment propera a 1. Pel lema 4.15, si W' retorna algun valor, aquest és correcte amb probabilitat exponencialment propera a 1. Falta veure que W' retorna algun valor en alguna de les iteracions. Prenem el $\gamma \in Z$ més petit tal que $\gamma \geq \alpha^2 - \beta^2$. Definim $\alpha' = \sqrt{\beta^2 + \gamma}$. Aleshores

$$\alpha \leq \alpha' \leq \sqrt{\alpha^2 + n^{-2c}\alpha^2} \leq (1 + n^{-2c}\alpha).$$

Per la proposició 4.8, la distància estadística entre Ψ_α i $\Psi_{\alpha'}$ és com a molt $9n^{-2c}$, que implica que la distància estadística entre n^c mostres de Ψ_α i n^c mostres de $\Psi_{\alpha'}$ és com a molt $9n^{-c}$. Per tant, per la nostra tria de γ, W dóna $\mathbf{s}$ amb probabilitat com a mínim $1 - 9n^{-c}/2 - 2^{-\Omega(n)} \geq \frac{1}{2}$. La probabilitat que $\mathbf{s}$ no es trobi en cap de les n execucions és per tant com a molt 2^{-n}. □

Lema 4.17. *Siguin $\epsilon = \epsilon(n)$ una funció negligible, $p = p(n) \geq 2$ un enter i $\alpha = \alpha(n) \in (0,1)$ un real. Assumim que tenim accés a un oracle W que per a tot $\beta \leq \alpha$ desconegut, troba $\boldsymbol{s}$ donat un nombre polinòmic de mostres de $A_{\boldsymbol{s},\Psi_\beta}$. Aleshores existeix un algorisme eficient que, donats una xarxa n-dimensional Λ, un nombre $r > \sqrt{2}p\eta_\epsilon(\Lambda)$ i un nombre polinòmic de mostres de $D_{L,r}$, soluciona el $CVP^{(p)}_{\Lambda^*,\alpha p/(\sqrt{2}r)}$.*

Demostració. Descriurem un algorisme que, donat un $x \in \mathbb{R}^n$ dins d'una distància $\alpha p/(\sqrt{2}r)$ de Λ^*, genera mostres de $A_{\mathbf{s},\Psi_\beta}$ per a algun $\beta \leq \alpha$ i $\mathbf{s} = (\Lambda^*)^{-1}\kappa_{\Lambda^*}(x) \bmod p$. Amb aquest algorisme podem obtenir el nombre polinòmic de mostres requerit per W per tal de trobar $\mathbf{s}$. Un cop es té $\mathbf{s}$ és directe recuperar $\kappa_{\Lambda^*}(x) \bmod p$, que és el que ens interessa.
L'algorisme funciona de la manera següent. Es pren una mostra $v \leftarrow D_{\Lambda,r}$ i es posa $a = \Lambda^{-1}v \bmod p$. Aleshores es retorna

$$(\mathbf{a}, \langle \mathbf{x}, \mathbf{v}\rangle/p + e \bmod 1),$$

on $e \in \mathbb{R}$ es pren d'acord amb una distribució normal centrada al zero i amb desviació estàndard $\frac{\alpha}{2\sqrt{\pi}}$. Volem veure ara que aquesta distribució està a una distància estadística negligible de $A_{\mathbf{s},\Psi_\beta}$ per a algun $\beta \leq \alpha$.
Veiem primer que la distribució de $\mathbf{a}$ és molt propera a la distribució uniforme. La probabilitat d'obtenir cada $a \in \mathbb{Z}_p^n$ és proporcional a $\rho_r(p\Lambda + \Lambda\mathbf{a})$. Utilitzant que $\eta_\epsilon(p\Lambda) = p\eta_\epsilon(\Lambda) < r$ i el lema 2.22 tenim que $\rho_r(p\Lambda + \Lambda\mathbf{a}) = (r/p)^n \det(\Lambda^*)(1 \pm \epsilon)$, que implica que la distància estadística entre la distribució de $\mathbf{a}$ i la distribució uniforme és negligible.
Ara prenem un valor fix de $\mathbf{a}$ i estudiem la distribució del segon element, és a dir, de $\langle \mathbf{x}, \mathbf{v}\rangle/p + e \bmod 1$. Definint $\mathbf{x}' = \mathbf{x} \cdot \kappa_{\Lambda^*}(\mathbf{x})$ i tenint en compte que $\|\mathbf{x}'\| \leq \alpha p/(\sqrt{2}r)$,

$$\langle \mathbf{x}, \mathbf{v}\rangle/p + e \bmod 1 = \langle \mathbf{x}'/p, \mathbf{v}\rangle + e + \langle \kappa_{\Lambda^*}(\mathbf{x}), \mathbf{v}\rangle/p \bmod 1.$$

Com que $\Lambda^{-1} = (\Lambda^*)^T$, tenim que

$$\langle \kappa_{\Lambda^*}(\mathbf{x}), \mathbf{v} \rangle = \langle (\Lambda^*)^{-1} \kappa_{\Lambda^*}(\mathbf{x}), \Lambda^{-1}\mathbf{v} \rangle.$$

És a dir, i en general, el producte intern d'un vector qualsevol en Λ^* i un vector qualsevol en Λ és igual al producte intern dels seus respectius vectors de coeficients. Com que els vectors de coeficients són enters,

$$\langle \kappa_{\Lambda^*}(\mathbf{x}), \mathbf{v} \rangle \equiv \langle \mathbf{s}, \mathbf{a} \rangle \bmod p.$$

Per tant $\langle \kappa_{\Lambda^*}(\mathbf{x}), \mathbf{v} \rangle / p \equiv \langle \mathbf{s}, \mathbf{a} \rangle / p \bmod 1$. Aplicant el lema 2.23 obtenim que la distribució de la part restant $\langle \mathbf{x}'/p, \mathbf{v} \rangle + e$ està a una distància estadística negligible de Ψ_β amb $\beta = \sqrt{(r\|\mathbf{x}'\|/p)^2 + \alpha^2/2} \leq \alpha$. Aquí hem utilitzat que la distribució de $\mathbf{v}$ és $D_{p\Lambda+\Lambda\mathbf{a},r}$, la distribució de e és normal centrada al zero i amb desviació estàndard $(\alpha/\sqrt{2})/\sqrt{2\pi}$ i que

$$\frac{1}{\sqrt{1/r^2 + (\sqrt{2}\|x'\|/p\alpha)^2}} \geq \frac{r}{\sqrt{2}} > \eta_\epsilon(p\Lambda).$$

□

Fem una síntesi informal de la demostració del teorema 4.12. Partim d'un algorisme W que resol el $\mathrm{LWE}_{p,\Psi_\alpha}$ amb un nombre polinòmic de mostres i volem veure que existeix un algorisme que resol el $\mathrm{CVP}_{\Lambda^*,\alpha p/(\sqrt{2}r)}$ donat un nombre polinòmic suficientment gran de mostres d'una distribució $D_{\Lambda,r}$. Primer de tot veiem que és suficient donar un algorisme per a $\mathrm{CVP}^{(p)}_{\Lambda*,\alpha p/(\sqrt{2}r)}$ i a continuació procedim a construir aquest algorisme amb dos resultats parcials. El lema 4.16 construeix un algorisme W' que, donades mostres de $A_{\mathbf{s},\Psi_\beta}$ per a algun $\beta \leq \alpha$, retorna $\mathbf{s}$ amb probabilitat exponencialment propera a 1 utilitzant W com a oracle. Finalment, el lema 4.17 explica com utilitzar W' i les mostres de $D_{\Lambda,r}$ per tal de solucionar el $\mathrm{CVP}^{(p)}_{\Lambda*,\alpha p/(\sqrt{2}r)}$.
El teorema 4.12, malgrat el que pugui semblar a simple vista, no és una demostració clàssica de la dificultat del LWE, ja que hem assumit que tenim accés a un nombre polinòmic de mostres de $D_{\Lambda,r}$, però generar aquestes mostres és tot un altre problema en si mateix.

Definició 4.18. *Siguin Λ una xarxa i φ una funció arbitrària de l'espai de xarxes als reals. Donat un $r > \varphi(\Lambda)$ el problema del* mostreig gaussià discret*, o DGS (Discrete Gaussian Sampling) per les seves sigles en anglès, consisteix a generar una mostra de $D_{\Lambda,r}$.*

Utilitzant el teorema 4.12 es pot demostrar el següent.

Teorema 4.19. *Siguin $\epsilon = \epsilon(n)$ una funció negligible, $p = p(n)$ un enter i $\alpha = \alpha(n) \in (0,1)$ tal que $\alpha p > 2\sqrt{n}$. Si tenim un oracle W que soluciona el LWE_{p,Ψ_α} donat un nombre polinòmic de mostres aleshores existeix un algorisme quàntic eficient per al problema $DGS_{\sqrt{2n}\cdot\eta_\epsilon(\Lambda)/\alpha}$.*

Aquest teorema és el que ens caracteritza la dificultat del problema LWE, ja que després es pot veure que el DGS és equivalent al GavSVP i el SIVP. No en farem la demostració formal, ja que no és l'objectiu d'aquest treball entrar en el terreny dels algorismes quàntics, però sí que en donarem la idea principal. El primer que cal veure és que donada una desviació estàndard suficientment gran sí que podem generar mostres de $D_{\Lambda,r}$ eficientment.

Lema 4.20. *Existeix un algorisme eficient que, donada una xarxa Λ de dimensió n i $r > 2^{2n}\lambda_n(\Lambda)$, retorna una mostra que està a una distància estadística menor que $2^{-\Omega(n)}$ de $D_{\Lambda,r}$.*

A partir d'aquí podem començar un pas iteratiu. Es pren un $r > \sqrt{2}p\eta_\epsilon(\Lambda)$ i es defineix $r_i = r \cdot (\alpha p/\sqrt{n})^i$. L'algorisme comença calculant n^c mostres de $D_{\Lambda,r_{3n}}$, que es poden calcular mitjançant el lema anterior. Després, en cada pas s'utilitzen les mostres D_{Λ,r_i} com a entrada del teorema 4.12, que juntament amb l'oracle que soluciona el LWE ens dóna el $\mathrm{CVP}_{\alpha p/r_i}$. La part quàntica, en la que no entrem, consisteix a utilitzar l'algorisme que soluciona el $\mathrm{CVP}_{\alpha p/r_i}$ per generar mostres de $D_{\Lambda,r\sqrt{n}/(\alpha p)} = D_{\Lambda,r_{i-1}}$. Repetim el procediment fins a $i = 1$, obtenint així n^c mostres de $D_{\Lambda,r_0} = D_{\Lambda,r}$ tal i com volíem.

4.3 Variants del LWE

Ara considerarem algunes variants del LWE i veurem que són tan difícils com el LWE. Hem estudiat només la dificultat del cas continu. Veiem que el cas discret s'hi pot reduir.

Lema 4.21 (Discret a continu)**.** *Siguin $n, p \geq 1$ enters, ϕ una funció de densitat de probabilitat en $\mathbb{T}$ i $\bar{\phi}$ el seu equivalent discret en $\mathbb{Z}_p$. Si tenim accés a un algorisme W que soluciona el $LWE_{p,\bar{\phi}}$ aleshores tenim un algorisme eficient W' que soluciona el $LWE_{p,\phi}$.*

Demostració. L'algorisme W' pren mostres de $A_{\mathbf{s},\phi}$ i discretitza el segon element per obtenir així mostres de $A_{\mathbf{s},\bar{\phi}}$. Aleshores s'aplica W per obtenir $\mathbf{s}$. □

Una altra variant del LWE estàndard (també anomenat de cerca, ja que l'objectiu és trobar $\mathbf{s}$) és el LWE de decisió, on l'objectiu és distingir les mostres $A_{\mathbf{s},\chi}$ d'una distribució uniforme U. El LWE de decisió es pot reduir al LWE de cerca si el mòdul p és primer i polinòmic en n.

Lema 4.22 (Decisió a cerca)**.** *Siguin $n \geq 1$ enter, $2 \leq p \leq poly(n)$ un nombre enter primer i χ una distribució en $\mathbb{Z}_p$. Si tenim accés a un algorisme W que per a tot $\boldsymbol{s}$ discrimina mostres $A_{\boldsymbol{s},\chi}$ de mostres uniformes U amb un encert exponencialment proper a 1, aleshores existeix un algorisme eficient W' que donades mostres de $A_{\boldsymbol{s},\chi}$ retorna $\boldsymbol{s}$ amb probabilitat exponencialment propera a 1.*

Demostració. Mirem com es pot obtenir $s_1 \in \mathbb{Z}_p$, la primera coordenada de $\mathbf{s}$. S'agafa un $k \in \mathbb{Z}_p$ arbitrari i donada una parella $A = (\mathbf{a}, b) \in \mathbb{Z}^n \times \mathbb{Z}$ considerem la transformació $A_k = (\mathbf{a}+(l, 0, ..., 0), b+l \cdot k)$, on $l \in \mathbb{Z}_p$ s'extreu uniformement. Resulta evident que si les mostres A segueixen una distribució uniforme aleshores A_k també és uniforme. Si A segueix la distribució $A_{\mathbf{s},\chi}$ i $k = s_1$, aleshores $A_k = (\mathbf{a}+(l, 0, ..., 0), b+l \cdot s_1)$ també segueix la distribució $A_{\mathbf{s},\chi}$. Finalment, si A segueix la distribució $A_{\mathbf{s},\chi}$ però $k \neq s_1$ aleshores A_k és uniforme (requereix p primer). Per tant podem utilitzar W per comprovar si $k = s_1$, i com que només hi ha un nombre polinòmic de possibilitats les podem comprovar totes. □

Finalment veiem la reducció del cas mitjà al cas pitjor, és a dir, que si podem distingir $A_{\mathbf{s},\chi}$ de la distribució uniforme per alguns $\mathbf{s}$ aleshores ho podem fer per a tots.

Lema 4.23. *Siguin $n, p \geq 1$ enters i χ una distribució de probabilitat en $\mathbb{Z}_p$. Si tenim accés a un algorisme W que distingeix $A_{\boldsymbol{s},\chi}$ per una fracció no negligible de tots els possibles valors de $\boldsymbol{s}$ aleshores existeix un algorisme eficient W' que per a tot $\boldsymbol{s}$ distingeix $A_{\boldsymbol{s},\chi}$ de U amb probabilitat exponencialment propera a 1.*

Ajuntant els tres últims lemes obtenim el següent.

Lema 4.24. *Siguin $n \geq 1$ enter, $2 \leq p \leq poly(n)$ primer, ϕ una distribució de probabilitat en $\mathbb{T}$ i $\bar{\phi}$ el seu equivalent discret en $\mathbb{Z}_p$. Si tenim accés a un algorisme que distingeix $A_{\boldsymbol{s},\bar{\phi}}$ de U per a un subconjunt no negligible de tots els $\boldsymbol{s}$ aleshores existeix un algorisme eficient que soluciona el $LWE_{p,\phi}$.*

4.4 Ring-LWE

Un problema similar al LWE és l'anomenat *Ring Learning With Errors*. Es tracta d'una versió del LWE que en comptes de treballar en l'anell $\mathbb{Z}$ es fa en anells de polinomis. Comencem donant una versió informal del problema. Siguin n una potència de 2 i q un primer pel qual se satisfà que $q \equiv 1 \bmod 2n$. Definim l'anell quocient $R_q = \mathbb{Z}_q[x]/(x^n + 1)$. Donat un element secret $\mathbf{s} \in R_q$ generem mostres $(\mathbf{a}, \mathbf{b} = \mathbf{a} \cdot \mathbf{s} + \mathbf{e}) \in R_q \times R_q$, on $\mathbf{a} \in R_q$ s'agafa uniformement i $\mathbf{e} \in R_q$ és un terme d'error escollit d'acord amb una certa distribució sobre R_q, la més natural consistent en prendre cada coordenada segons una distribució normal independent i distribuïda idènticament (iid), és a dir, una distribució gaussiana esfèrica. L'objectiu és recuperar $\mathbf{s}$ a partir de les mostres $(\mathbf{a}, \mathbf{b})$. Ara definirem i analitzarem el problema amb més detall i donarem alguns resultats sobre la seva dificultat.

4.5 Conceptes previs

Abans de poder analitzar el problema RLWE introduïm alguns conceptes previs.

Definició 4.25. *Anomenem* cos de nombres *a tot cos extensió finita de* $\mathbb{Q}$.

Volem estudiar els anells $\mathbb{Z}[x]/(\Phi_m(x))$, on $\Phi_m(x)$ és el m-èsim polinomi ciclotòmic. Si n és una potència de 2 aleshores $\Phi_{2n}(x) = x^n + 1$, que és el cas que donem a la definició informal. Sabem que $\mathbb{Q}[x]/(\Phi_m(x)) \cong \mathbb{Q}(\zeta_m)$, on ζ_m és una arrel m-èsima primitiva de la unitat. Recordem que tot cos de nombres admet un element primitiu i que admet exactament tantes immersions en $\mathbb{C}$ com el seu grau.

Definició 4.26. *Sigui* $\sigma : K \to \mathbb{C}$ *una immersió. Si* $\sigma(K) \subset \mathbb{R}$ *direm que es tracta d'una* immersió real. *Alternativament parlarem d'una* immersió complexa no real.

Si θ és un element primitiu del cos K; és a dir, si $K = \mathbb{Q}(\theta)$, i si $f(x) = (\theta, \mathbb{Q})(x)$ és el polinomi mínim de θ, les immersions complexes de K són determinades unívocament per la imatge de θ, que pot ser qualsevol de les arrels complexes de θ. Per tant, si n és el grau de K (o sigui, el de $f(x)$), tenim exactament n immersions de K en $\mathbb{C}$. Sigui s_1 el nombre d'immersions reals de K (és a dir, d'arrels reals de $f(x)$). Les altres, s'aparellen cadascuna amb la seva complexa conjugada, de manera que $n = s_1 + 2s_2$, on s_2 és el nombre de parelles d'immersions complexes conjugades no reals de K. Notem que els nombres s_1 i s_2 no depenen de l'element primitiu, perquè són els nombres d'immersions reals de K (que només depèn de K) i el de parelles d'immersions complexes no reals (que només depèn de K).

Definició 4.27. *Siguin* s_1, s_2, n *enters tal que* $s_1 + 2s_2 = n$. *Es defineix l'espai*

$$H := \left\{(x_1, ..., x_n) \in \mathbb{R}^{s_1} \times \mathbb{C}^{2s_2} : x_{s_1+s_2+j} = \overline{x_{s_1+j}}, \forall j \in [1, s_2]\right\} \subseteq \mathbb{C}^n.$$

L'espai H amb el producte escalar induït per $\mathbb{C}^n$ és isomorf a $\mathbb{R}^n$ com a espai de Hilbert. Si ordenem les immersions $\sigma_i : K \to \mathbb{C}$ degudament podem definir el que anomenem *immersió canònica* $\sigma : K \to H$ per

$$\sigma(x) = (\sigma_1(x), ..., \sigma_n(x)).$$

Associar elements de K amb elements de H ens serveix per dotar-los d'una interpretació geomètrica, on les normes en K són les normes induïdes de $\mathbb{C}$. En l'esquema de Gentry es fa una anàlisi més simple de les xarxes ideals a través de la immersió per coeficients. Els avantatges de la immersió canònica són múltiples. Mentre que la immersió per coeficients només ens permet sumar, en la immersió canònica tant la suma com la multiplicació es poden fer component a component. Una altra molt bona propietat és que es comporta bé amb els automorfismes: simplement permuten els eixos de la immersió.
Construirem una base $\{\mathbf{h}_i\}_{i\in[1,n]}$ de H de la manera següent. Per a tot $1 \leq j \leq n$ definim $\mathbf{e}_j \in \mathbb{C}^n$ el vector amb 1 a la coordenada j i 0 a la resta. Per a $1 \leq j \leq s_1$ prenem $\mathbf{h}_j = \mathbf{e}_j$ i per a $s_1 < j \leq s_1 + s_2$ prenem $\mathbf{h}_j = \frac{1}{\sqrt{2}}(\mathbf{e}_j + \mathbf{e}_{j+s_2})$ i $\mathbf{h}_{j+s_2} = \frac{i}{\sqrt{2}}(\mathbf{e}_j - \mathbf{e}_{j+s_2})$.

Definició 4.28. *Anomenem* enter algebraic *a un nombre que és arrel d'un polinomi mònic de coeficients en* $\mathbb{Z}$. *Donat un cos de nombres* K, *els enters algebraics de* K *formen un subanell, que s'acostuma a denotar per* $\mathcal{O}_K$.

Per un cos de nombres K qualsevol de grau n el seu anell d'enters $\mathcal{O}_K$ és un $\mathbb{Z}$-mòdul lliure de dimensió n. A les bases de $\mathcal{O}_K$ les anomenem bases d'enters.

Definició 4.29. *Siguin* K *un cos numèric de grau* n *i* $\sigma_1, ..., \sigma_n$ *les immersions de* K *en* $\mathbb{C}$. *La* traça *i la* norma *de* K *en* $\mathbb{Q}$ *estan definides, respectivament, per*

$$Tr_{K/\mathbb{Q}}(\alpha) := \sum_{i=1}^{n} \sigma_i(\alpha), \quad N_{K/\mathbb{Q}} := \prod_{i=1}^{n} \sigma_i(\alpha).$$

Les imatges de la traça i la norma estan contingudes en $\mathbb{Q}$. A més a més, l'aplicació $(x, y) \mapsto \mathrm{Tr}(xy)$ és una forma bilineal, simètrica i no degenerada en K com a $\mathbb{Q}$-espai vectorial.

Definició 4.30. *Sigui K un cos de nombres. Una* xarxa *en K és el conjunt de combinacions lineals de coeficients en $\mathbb{Z}$ d'una $\mathbb{Q}$-base de K, és a dir, el $\mathbb{Z}$-submòdul lliure de K generat per una $\mathbb{Q}$-base de K.*

Exemples de xarxes en un cos de nombres K són el seu anell d'enters $\mathcal{O}_K$ o els ideals fraccionaris. Una xarxa de K genera, a través de la immersió canònica, una xarxa en H, i per tant en $\mathbb{R}^n$.

Definició 4.31. *Sigui Λ una xarxa en K. El seu* dual *es defineix com*

$$\Lambda^\vee = \{x \in K : Tr(x\Lambda) \subseteq \mathbb{Z}\}.$$

La traça fa en K la funció del producte escalar en $\mathbb{R}^n$. Sota la immersió canònica tenim que $\mathrm{Tr}(xy) = \sum_i \sigma_i(xy) = \sum_i \sigma_i(x)\sigma_i(y) = \langle \sigma(x), \overline{\sigma(y)} \rangle$. A partir d'aquí es verifica que $\sigma(\Lambda^\vee) = \overline{\sigma(\Lambda)^*}$.
Veiem ara quines distribucions utilitzarem per a l'estudi del RLWE. Recordem que per a $r > 0$ es defineix $\rho_r : H \to (0, 1]$ com $\rho_r(\mathbf{x}) = e^{-\pi\langle \mathbf{x}, \mathbf{x} \rangle / r^2}$. Normalitzant aquesta funció obtenim la distribució gaussiana contínua D_r que té densitat $r^{-n} \cdot \rho_r(\mathbf{x})$. Aquesta distribució és esfèrica, ja que té la mateixa desviació estàndard en tots els eixos. Podem estendre la definició per a distribucions gaussianes el·líptiques en la base $\{\mathbf{h}_i\}_{i \in [1,n]}$ prenent $\mathbf{r} = (r_1, ..., r_n) \in (\mathbb{R}^+)^n$ tal que $r_{j+s_1+s_2} = r_{j+s_1}$ per a cada $j \in [1, s_2]$. Aleshores una mostra de $D_{\mathbf{r}}$ s'obté calculant $\sum_{i=1}^n x_i \mathbf{h}_i$, on x_i es prenen de forma independent de les distribucions D_{r_i} en $\mathbb{R}$.
La distribució $D_{\mathbf{r}}$ no és una distribució sobre K, ja que la imatge de K per la immersió canònica no és tot H. $D_{\mathbf{r}}$ és una distribució sobre $K_{\mathbb{R}} = K \otimes_{\mathbb{Q}} \mathbb{R}$, que és isomorf a H. Hi ha la possibilitat de discretitzar la distribució $D_{\mathbf{r}}$ sobre una xarxa ideal de H per obtenir una distribució sobre una xarxa de K.

Definició 4.32. *Sigui $\alpha > 0$ real. La família de distribucions $\Psi_{\leq \alpha}$ és el conjunt de distribucions gaussianes el·líptiques $D_{\boldsymbol{r}}$ sobre $K_{\mathbb{R}}$, on cada paràmetre $r_i \leq \alpha$.*

Lema 4.33. *Sigui $K = \mathbb{Q}(\zeta_m)$ un cos ciclotòmic. Per a tot $\alpha > 0$ i $\tau \in Gal(K \mid \mathbb{Q})$ la família $\Psi_{\leq \alpha}$ és invariant per l'automorfisme τ, és a dir, $\psi \in \Psi_{\leq \alpha} \implies \tau(\psi) \in \Psi_{\leq \alpha}$.*

Demostració. Els automorfismes de K són unívocament determinats per les assignacions $\tau_k(\zeta_m) := \zeta_m^k$ per r invertible mòdul m i permuten les coordenades de la immersió canònica. Per a qualsevol $\psi = D_{\mathbf{r}} \in \Psi_{\leq \alpha}$ tenim $\tau_k(D_{\mathbf{r}}) = D_{\mathbf{r}'} \in \Psi_{\leq \alpha}$, ja que $\mathbf{r}'$ és una permutació de $\mathbf{r}$ i per tant $r'_i \leq \alpha$. □

Definició 4.34. *Siguin $K = \mathbb{Q}(\zeta_m)$ el m-èsim cos ciclotòmic i $n = \varphi(m)$. Per un $\alpha > 0$ real es defineix Υ_α com una distribució sobre distribucions gaussianes en $K_{\mathbb{R}}$. Per generar una mostra de Υ_α es prenen $x_1, ..., x_{n/2}$ independents de la distribució $\Gamma(2, 1)$ i es retorna la distribució $D_{\boldsymbol{r}}$, on $r_i^2 = r_{i+n/2}^2 = \alpha^2(1 + \sqrt{n}x_i)$.*

4.6 Formalització del RLWE

Siguin K un cos de nombres, $R = \mathcal{O}_K$ el seu anell d'enters i $q \geq 2$ un enter (racional). R és una xarxa en K i per tant podem considerar la xarxa dual $R^\vee$, que és un ideal fraccionari de K. Per un ideal fraccionari qualsevol $\mathcal{I}$ denotarem $\mathcal{I}_q = \mathcal{I}/q\mathcal{I}$. Sigui $\mathbb{T} = K_{\mathbb{R}}/R^\vee$.

Definició 4.35 (Distribució RLWE)**.** *Siguin $s \in R_q^\vee$ i ψ una distribució de probabilitat sobre $K_{\mathbb{R}}$. Es defineix la distribució $A_{s,\psi}$ en $R_q \times \mathbb{T}$ com el resultat d'escollir $a \in R_q$ uniformement, $e \in K_{\mathbb{R}}$ d'acord amb la distribució ψ i retornar $(a, b = (a \cdot s)/q + e \bmod R^\vee)$.*

Hi ha dues variants del problema RLWE: la de cerca i la de decisió. De forma anàloga al LWE, el problema de cerca consisteix a trobar s, mentre que el de decisió consisteix a distingir mostres de $A_{s,\psi}$ d'una distribució uniforme.

Definició 4.36. *Sigui Ψ una família de distribucions sobre $K_{\mathbb{R}}$. La variant de* cerca *del problema RLWE, que denotem $RLWE_{q,\Psi}$, consisteix a, donat accés a un nombre arbitrari de mostres independents de $A_{s,\psi}$ per a algun $s \in R_q^\vee$ i $\psi \in \Psi$, trobar s.*

Definició 4.37. *Sigui Υ una distribució sobre una família de distribucions sobre $K_{\mathbb{R}}$. La versió de* decisió *(cas mitjà) del problema RLWE, que denotem $RDLWE_{q,\Upsilon}$, consisteix a distingir entre un nombre arbitrari de mostres de $A_{s,\psi}$, per una tria aleatòria de $(s,\psi) \leftarrow U(R_q^\vee) \times \Upsilon$, i el mateix nombre de mostres uniformes i independents de $R_q \times \mathbb{T}$.*

El teorema principal sobre la dificultat del RLWE de cerca ens ve donat per Vadim Lyubashevsky, Chris Peikert i Oded Regev [14].

Teorema 4.38. *Siguin K un cos numèric arbitrari de grau n i $R = \mathcal{O}_K$. Siguin $\alpha = \alpha(n) > 0$ i $q = q(n) > 2$ tals que $\alpha q \geq 2 \cdot \omega(\sqrt{\log n})$. Per algun $\epsilon = \epsilon(n)$ negligible, si existeix un algorisme polinòmic pel $RLWE_{q,\Psi_{\leq\alpha}}$ aleshores existeix un algorisme quàntic eficient que soluciona el K-DGS_γ, amb*

$$\gamma = \max\left\{\eta_\epsilon(\mathcal{I}) \cdot (\sqrt{2}/\alpha) \cdot \omega(\sqrt{\log n}), \sqrt{2n}/\lambda_1(\mathcal{I}^\vee)\right\},$$

on $\omega(\sqrt{\log n})$ denota una funció fixa i arbitrària que creix asimptòticament més ràpidament que $\sqrt{\log n}$, i K-DGS_γ denota el problema de mostreig gaussià discret en K, que consisteix a, donats un ideal $\mathcal{I} \subset K$ i un $s \geq \gamma(\mathcal{I})$, generar mostres de la distribució $D_{\mathcal{I},s}$.

De forma anàloga al LWE ens trobem que hi ha una reducció quàntica del problema de mostreig gaussià discret al RLWE de cerca. Malauradament no hi ha cap demostració completament clàssica de la dificultat del problema. Notem que K és un cos numèric arbitrari; per demostrar la dificultat de cerca no és necessari que sigui un cos ciclotòmic.
L'equivalència del RLWE de cerca i el de decisió ve donada pel teorema següent.

Teorema 4.39. *Siguin ζ_m una arrel m-èsima primitiva de la unitat, $K = \mathbb{Q}(\zeta_m)$ el cos ciclotòmic associat de grau $n = \varphi(m)$, $R = \mathcal{O}_K = \mathbb{Z}[\zeta_m]$ el seu anell d'enters, $R^\vee = \mathcal{O}_K^\vee$ el seu dual i $q \equiv 1 \bmod m$ un enter acotat polinòmicament respecte de n. Sigui $\alpha = \alpha(n) > 0$ tal que $\alpha q \geq \eta_\epsilon(R^\vee)$ per algun $\epsilon = \epsilon(n)$ negligible. Aleshores hi ha una reducció en temps polinòmic del $RLWE_{q,\Psi_{\leq\alpha}}$ al $RDLWE_{q,\Upsilon_\alpha}$.*

No farem una demostració completa d'aquest teorema però sí que l'analitzarem. La demostració és un procés de quatre reduccions. Primer es veu que l'ideal $(q) \subset R^\vee$ es pot expressar com $(q) = \prod_{i\in\mathbb{Z}_m^*} \mathfrak{q}_i$, on $\mathfrak{q}_i = (q, x - \zeta_m^i)$ són ideals primers. Pel teorema xinès del residu tenim l'isomorfisme $R_q^\vee \cong \otimes_{i\in\mathbb{Z}_m^*}(R^\vee/\mathfrak{q}_i)$. Es defineix el problema $\mathfrak{q}_i$-$\mathrm{RLWE}_{q,\Psi}$ de forma anàloga al RLWE però on l'objectiu no és trobar s sinó $s \bmod \mathfrak{q}_i$.

Lema 4.40 (1ª reducció)**.** *Suposem que la família Ψ és tancada pels automorfismes de K. Aleshores, per a cada $i \in \mathbb{Z}_m^*$ existeix una reducció determinista en temps polinòmic del $RLWE_{q,\Psi}$ al $\mathfrak{q}_i$-$RLWE_{q,\Psi}$.*

Recordem que les distribucions gaussianes el·líptiques amb què treballem són, efectivament, tancades pels automorfismes de K. Per a la segona reducció necessitem unes definicions prèvies. Per conveniència identifiquem els elements de $\mathbb{Z}_m^*$ amb els seus representants en el conjunt $\{1, ..., m-1\}$ amb l'ordre de $\mathbb{Z}$. Per $i \in \mathbb{Z}_m^*$ denotem amb i^- l'element de $\mathbb{Z}_m^*$ més gran menor que i i definim $1^- = 0$.

Definició 4.41. *Siguin $i \in \mathbb{Z}_m^*$, $s \in R_q^\vee$ i ψ una distribució sobre $K_{\mathbb{R}}$. La distribució $A_{s,\psi}^i$ sobre $R_q \times \mathbb{T}$ es genera escollint $(a,b) \leftarrow A_{s,\psi}$ i retornant $(a, b + h/q)$, on $h \in R_q^\vee$ es uniforme i independent mòdul $\mathfrak{q}_j$ amb $j \leq i$ i és 0 mòdul la resta de $\mathfrak{q}_j$. Definim $A_{s,\psi}^0 = A_{s,\psi}$.*

Definició 4.42. *Siguin $i \in \mathbb{Z}_m^*$ i Ψ una família de distribucions. Es defineix el problema $WDLWE_{q,\Psi}^i$ de la manera següent: donat accés a mostres de $A_{s,\psi}^j$ per uns $s \in R_q^\vee$, $\psi \in \Psi$ i $j \in \{i^-, i\}$ arbitraris, trobar j.*

El WDLWE, anomenat així ja que es tracta del cas pitjor del problema de decisió (*worst-case decision*), consisteix doncs en diferenciar mostres de les distribucions $A_{s,\psi}^i$ i $A_{s,\psi}^{i^-}$. Tenim les reduccions següents.

Lema 4.43 (2ª reducció)**.** *Per a qualsevol $i \in \mathbb{Z}_m^*$ hi ha una reducció probabilística en temps polinòmic del $\mathfrak{q}_i$-$RLWE_{q,\Psi}$ al $WDLWE_{q,\Psi}^i$.*

Definició 4.44. *Siguin $i \in \mathbb{Z}_m^*$ i Υ una distribució sobre les distribucions d'error. Diem que un algorisme soluciona el problema $DLWE^i_{q,\Upsilon}$ si per a una probabilitat no negligible en la tria de $(s,\psi) \leftarrow U(R_q^\vee) \times \Upsilon$ accepta mostres de $A^i_{s,\psi}$ i rebutja mostres de $A^{i^-}_{s,\psi}$ amb una diferència no negligible de les probabilitats d'acceptació.*

El problema $\mathrm{DLWE}^i_{q,\Upsilon}$ és el cas mitjà del problema de decisió relatiu a $\mathfrak{q}_i$. La diferència entre el $\mathrm{WDLWE}^i_{q,\Psi}$ i el $\mathrm{DLWE}^i_{q,\Upsilon}$ és que, mentre que en el cas pitjor s i ψ es poden escollir de forma concreta, en el cas mitjà s és uniforme en $R_q^\vee$ i ψ és una mostra de la distribució Υ.

Lema 4.45 ($3^{\underline{a}}$ reducció)**.** *Per a qualsevol $\alpha > 0$ i $i \in \mathbb{Z}_m^*$ existeix una reducció en temps polinòmic del $WDLWE^i_{q,\Psi_{\leq\alpha}}$ al $DLWE^i_{q,\Upsilon_\alpha}$.*

Lema 4.46 ($4^{\underline{a}}$ reducció)**.** *Sigui Υ una distribució sobre distribucions d'error satisfent que per a qualsevol $\psi \leftarrow \Upsilon$ i qualsevol $s \in R_q^\vee$, la distribució $A^{m-1}_{s,\psi}$ està a una distància estadística negligible de la uniforme. Aleshores, per a qualsevol oracle que soluciona el problema $DLWE_{q,\Upsilon}$ existeix un algorisme eficient que soluciona el problema $DLWE_{q,\Upsilon}$.*

Una de les conseqüències del teorema 4.39, que contrasta amb els resultats sobre el problema de decisió del problema LWE, és que no es pot prendre una distribució de probabilitat d'error ψ concreta sinó que s'ha de prendre de forma aleatòria d'acord amb la distribució Υ. En el problema LWE el fet de que es conegui la distribució ψ no afecta a la dificultat del problema, però en el cas RLWE sí. Aquest fet es deu a que, tot i que la família de distribucions $\Psi_{\leq\alpha}$ és tancada respecte dels automorfismes de K, les distribucions d'aquesta família, per si soles, no tenen per què estar-ho, ja que poden ser el·líptiques. Aquest fet provoca que a la reducció del cas pitjor al cas mitjà s'hagi d'aleatoritzar la distribució d'error. Podem evitar-ho tenint en compte la variant següent del teorema 4.39.

Teorema 4.47. *Siguin R, q i α com en el teorema 4.39. Sigui $\ell \geq 1$. Existeix una reducció en temps polinòmic del $RLWE_{q,\Psi_{\leq\alpha}}$ al $RDLWE_{q,D_\xi}$ donades només ℓ mostres, on $\xi = \alpha \cdot (n\ell/\log(n\ell))^{1/4}$.*

Per tant, si l'adversari té accés a un nombre limitat de mostres podem evitar aleatoritzar la distribució d'error i fer servir una distribució gaussiana esfèrica.

4.7 Atacs

Hem estudiat la seguretat asimptòtica dels problemes LWE i RLWE: no s'ha trobat cap algorisme que els solucioni de forma general en temps polinòmic i, donat el consens més o menys acceptat sobre la dificultat d'alguns problemes en xarxes, és poc probable que existeixi. El següent que cal analitzar és la dificultat concreta, és a dir, com s'han d'ajustar els paràmetres perquè els problemes LWE i RLWE siguin segurs donats els algorismes i la capacitat computacional que tenim en l'actualitat. L'anàlisi més exhaustiva de la dificultat concreta del LWE ha estat duta a terme per Martin R. Albrecht, Rachel Player i Sam Scott [2], originalment publicat el 2015 i amb última edició el 2019, fet que demostra que és una àrea activa de recerca. No podem estendre'ns en detalls, però l'article considera diferents algorismes amb els quals solucionar el problema LWE, alguns d'ells basats en algorismes de reducció de xarxes com el LLL o el BKZ, i donen taules d'ajustament dels paràmetres per a tres possibles distribucions del secret $\mathbf{s}$: la distribució uniforme, la distribució d'error i la distribució ternària, que retorna 0 amb probabilitat 0.5 i 1 i -1 amb probabilitats 0.25.

4.8 Aplicacions

Donada la seva presumpta seguretat davant d'algorismes quàntics, a diferència d'altres problemes utilitzats en criptografia com la factorització d'enters o el logaritme discret, el RLWE podria tenir una gran rellevància en el futur si s'aconsegueixen fabricar ordinadors quàntics. Actualment, no només serveix com a base per a construir esquemes criptogràfics homomòrfics, com l'esquema BGV de Brakerski, Gentry i Vaikuntanathan [4], l'esquema BFV de Fan i Vercauteren [7] o l'esquema CKKS de Cheon, Kim, Kim i Song [6], sinó que també ha estat utilitzat per dissenyar protocols d'intercanvi de claus, com el de Ding, Xie i Lin [11], i protocols de firma digital, com el de Lyubashevsky [13].

5 El criptosistema BGV

En aquesta secció farem una exploració detallada sobre el criptosistema BGV, publicat al 2011 per Brakersi, Gentry i Vaikuntanathan [4]. Aquest esquema és interessant no només perquè millora l'eficiència i la viabilitat respecte dels criptosistemes anteriors, sinó també per la seva flexibilitat; es pot construir en l'anell dels enters o en anells de polinomis, fent ús del LWE i del RLWE, respectivament, i es pot utilitzar, o no, la tècnica del bootstrapping.

5.1 Qüestions prèvies

Notació 5. *Donats $q, a \in \mathbb{Z}$ denotem amb $[a]_q$ el representant distingint de $a \bmod q$ que està en $(-q/2, q/2]$. Si a és un vector d'enters, $[(a_1, ..., a_n)]_q = ([a_1]_q, ..., [a_n]_q)$.*

5.2 Esquema base

Comencem construint un esquema de xifrat base, no homomòrfic, a partir del qual es construirà l'esquema homomòrfic. Treballarem amb el polinomi $f(x) = x^{2^d} + 1$, $d \geq 0$ que sabem que és irreductible en $\mathbb{Z}[x]$, i considerem l'anell quocient $R = \mathbb{Z}[x]/(f(x))$. Malgrat que l'estudi del RLWE es duu a terme amb la immersió canònica, aquí utilitzarem sempre la immersió per coeficients, ja que és més senzill d'implementar. En el cas de que $d = 0$ l'anell $R = Z[x]/(x + 1) \cong Z[x]$ i per tant tindrem un esquema basat en el LWE. Si $d \geq 1$ tindrem un esquema basat en el RLWE. Els paràmetres del sistema són els següents.

- $\lambda \in \mathbb{R}^+$: És el paràmetre de seguretat. Denota una seguretat de 2^λ per a atacs coneguts. L'objectiu serà, doncs, ajustar la resta de paràmetres per assegurar una seguretat de 2^λ.
- q: És el mòdul de l'anell finit en el qual treballarem (R_q). És un primer i satisfà que $q \equiv 1 \bmod d$.
- $d = d(\lambda)$: Defineix el grau del polinomi amb el qual treballem. En el cas $d = 0$ estem en un sistema basat en enters i el LWE.
- $n = n(\lambda)$: És la dimensió del sistema. S'utilitza només en el cas LWE. Si $d > 0$ aleshores $n = 1$.
- $\chi = \chi(\lambda)$: És una variable aleatòria que pren valors en R_q d'acord amb una distribució determinada.

Definim el sistema criptogràfic amb els seus algorismes.

- $\mathsf{E.Setup}(\lambda, \mu, b)$: Inicialitza l'esquema. Pren el paràmetre de seguretat λ, un enter positiu μ i un bit $b \in \{0, 1\}$. Si $b = 0$ construïm un sistema basat en el LWE (amb $d = 1$) i si $b = 1$ construïm un sistema basat en el RLWE (amb $n = 1$). Escull un mòdul q de μ bits i els altres paràmetres $(d,\ n,\ \chi)$ de forma apropiada per garantir una seguretat de 2^λ. Posa $N = \lceil (2n + 1) \log q \rceil$. Retorna $params = (q, d, n, N, \chi)$.
- $\mathsf{E.SecretKeyGen}(params)$: Extreu $\mathbf{s}' \leftarrow \chi^n$. Posa $sk = \mathbf{s} = (1, s'_1, ..., s'_n) \in R_q^{n+1}$.
- $\mathsf{E.PublicKeyGen}(params, sk)$: Pren una clau secreta $sk = s = (1, s')$. Genera la matriu $\mathbf{A'} \in R_q^{N \times n}$ d'acord amb una distribució uniforme i extreu un vector $\mathbf{e} \leftarrow \chi^N$. Posa $\mathbf{b} = \mathbf{A's'} + 2\mathbf{e}$. Anomenem $\mathbf{A} \in R_q^{N \times (n+1)}$ a la matriu consistent en el vector columna $\mathbf{b}$ seguit de les n columnes de $-\mathbf{A'}$. La clau secreta és $pk = \mathbf{A}$. (Observem que $\mathbf{A} \cdot \mathbf{s} = 2\mathbf{e}$.)
- $\mathsf{E.Enc}(params, pk, m)$: Per xifrar un missatge $m \in R_2$ posa $\mathbf{m'} = (m, 0, ..., 0) \in R_q^{n+1}$ i extreu $\mathbf{r} \in R_2^N$ uniformement. El text xifrat és $\mathbf{c} = \mathbf{m'} + \mathbf{A}^T\mathbf{r} \in R_q^{n+1}$.
- $\mathsf{E.Dec}(params, sk, c)$: El missatge desxifrat és $m = \Big[[\langle \mathbf{c}, \mathbf{s} \rangle]_q\Big]_2$.

Veiem sota quines condicions l'esquema és correcte. Si desxifrem un missatge obtenim que

$$\Big[[\langle \mathbf{c}, \mathbf{s}\rangle]_q\Big]_2 = \Big[\Big[\langle \mathbf{m}' + \mathbf{A}^T\mathbf{r}, \mathbf{s}\rangle\Big]_q\Big]_2 = \Big[\Big[\langle \mathbf{m}', \mathbf{s}\rangle + \langle \mathbf{A}^T\mathbf{r}, \mathbf{s}\rangle\Big]_q\Big]_2$$

$$= \Big[\Big[m + \mathbf{s}^T\mathbf{A}^T\mathbf{r}\Big]_q\Big]_2 = \Big[\big[m + (2\mathbf{e})^T\mathbf{r}\big]_q\Big]_2 = \Big[[m + 2\langle \mathbf{e}, \mathbf{r}\rangle]_q\Big]_2.$$

Per tant, m es desxifrarà correctament si

$$\Big[[2\langle \mathbf{e}, \mathbf{r}\rangle]_q\Big]_2 = 0.$$

Això implica que tots els coeficients de $\langle \mathbf{e}, \mathbf{r}\rangle$ han d'estar en l'interval $[-q/4, q/4)$.

5.3 Portes homomòrfiques

L'objectiu és dotar al sistema criptogràfic anterior de la capacitat d'avaluar circuits aritmètics. El primer pas és veure com es poden calcular la suma i el producte de textos plans homomòrficament. Siguin $m_1, m_2 \in R_q$ dos missatges plans i $\mathbf{c}_1, \mathbf{c}_2 \in R_q^{n+1}$ els seus corresponents missatges xifrats amb la parella de claus pública i privada ($pk = \mathbf{A}, sk = \mathbf{s}$). Si intentem desxifrar $\mathbf{c} = \mathbf{c}_1 + \mathbf{c}_2$ obtenim que

$$\mathsf{E.Dec}(sk, \mathbf{c}) = \Big[[\langle \mathbf{c}, \mathbf{s}\rangle]_q\Big]_2 = \Big[[\langle \mathbf{c}_1 + \mathbf{c}_2, \mathbf{s}\rangle]_q\Big]_2 = \Big[[\langle \mathbf{c}_1, \mathbf{s}\rangle + \langle \mathbf{c}_2, \mathbf{s}\rangle]_q\Big]_2$$

$$= \Big[[m_1 + m_2 + 2\langle \mathbf{e}, \mathbf{r}_1\rangle + 2\langle \mathbf{e}, \mathbf{r}_2\rangle]_q\Big]_2,$$

que és igual a $m_1 + m_2$ sempre que la suma dels errors estigui dins del rang que permeti desxifrar.

Notació 6. *Donats dos vectors $u = (u_1, ..., u_n)$ i $v = (v_1, ..., v_m)$ anomenarem el producte exterior de u i v, i el denotarem per $u \otimes v$, a la matriu uv^T, de dimensió $n \times m$.*

Al desxifrar $c = c_1 \otimes c_2$ amb la clau secreta $sk' = s \otimes s$ obtenim que

$$\mathsf{E.Dec}(sk', \mathbf{c}) = \Big[[\langle \mathbf{c}_1 \otimes \mathbf{c}_2, \mathbf{s} \otimes \mathbf{s}\rangle]_q\Big]_2 = \left[\left[\sum_{i,j \le n+1} c_{1i}c_{2j}s_is_j\right]_q\right]_2$$

$$= \Big[[\langle \mathbf{c}_1, s\rangle\langle \mathbf{c}_2, \mathbf{s}\rangle]_q\Big]_2 = \Big[[(m_1 + 2\langle \mathbf{e}, \mathbf{r}_1\rangle)(m_2 + 2\langle \mathbf{e}, \mathbf{r}_2\rangle)]_q\Big]_2$$

$$= \Big[[m_1m_2 + 2m_1\langle \mathbf{e}, \mathbf{r}_2\rangle + 2m_2\langle \mathbf{e}, \mathbf{r}_1\rangle + 4\langle \mathbf{e}, \mathbf{r}_1\rangle\langle \mathbf{e}, \mathbf{r}_2\rangle]_q\Big]_2,$$

que és igual a m_1m_2 sempre que l'error, que ha augmentat de forma quadràtica, estigui dins del rang que permeti desxifrar. Hem vist per tant que l'esquema permet fer sumes i multiplicacions homomòrfiques. L'objectiu és ara fer-ho més viable.

5.4 Canvi de claus

Per avaluar homomòrficament una porta multiplicativa tant l'error com la llargada del text xifrat augmenten de forma quadràtica en cada operació (per tant de forma exponencial en la profunditat del circuit). Això no només és molt poc pràctic, ja que al cap d'unes poques multiplicacions tindríem vectors amb els quals seria impossible treballar-hi, sinó que l'esquema no seria privat per a circuits, ja que la llargada d'un text xifrat donaria informació sobre de quines operacions és el resultat. La primera de les tècniques utilitzades per solucionar aquests problemes és la de canviar les claus, és a dir, si tenim un text xifrat $\mathbf{c}$ que es desxifra amb la clau $\mathbf{s}$, convertir-lo a un text xifrat $\mathbf{c}'$ que es desxifri amb $\mathbf{s}'$, sense conèixer ni $\mathbf{s}$ ni $\mathbf{s}'$. Volem $\langle \mathbf{c}, \mathbf{s}\rangle \equiv \langle \mathbf{c}', \mathbf{s}'\rangle \mod q$. Comencem definint dues subrutines.

- $\mathsf{BitDecomp}(\mathbf{x} \in R_q^n, q)$: Descompon $\mathbf{x}$ en la seva representació en base 2. És a dir, calcula els coeficients $\mathbf{u}_i \in R_2^n$ tals que $\mathbf{x} = \sum_{i=0}^{\lfloor \log_2 q \rfloor} 2^i \mathbf{u}_i$ i retorna $(\mathbf{u}_1, ..., \mathbf{u}_n) \in R_2^{n\lceil \log_2 q \rceil}$.
- $\mathsf{PowersOf2}(\mathbf{x} \in R_q^n, q)$: Retorna el vector $(\mathbf{x}, 2\mathbf{x}, ..., 2^{\lfloor \log_2 q \rfloor}\mathbf{x}) \in R_q^{n\lceil \log_2 q \rceil}$.

Lema 5.1. *Siguin $\boldsymbol{c}, \boldsymbol{s} \in R_q^n$. Aleshores $\langle \mathsf{BitDecomp}(\boldsymbol{c}, q), \mathsf{PowersOf2}(\boldsymbol{s}, q)\rangle \equiv \langle c, s\rangle \mod 2$.*

Demostració.

$$\langle \mathsf{BitDecomp}(\mathbf{c}, q), \mathsf{PowersOf2}(\mathbf{s}, q)\rangle = \sum_{i=0}^{\lfloor \log_2 q \rfloor} \langle \mathbf{u}_i, 2^i \cdot \mathbf{s}\rangle = \sum_{i=0}^{\lfloor \log_2 q \rfloor} \langle 2^i \cdot \mathbf{u}_i, \mathbf{s}\rangle = \langle \mathbf{c}, \mathbf{s}\rangle.$$

□

Fer el canvi de claus consta de dos passos. El primer pas consisteix a, a partir de dues claus privades $\mathbf{s}_1$ i $\mathbf{s}_2$, generar una informació $\tau_{\mathbf{s}_1 \to \mathbf{s}_2}$ que permeti fer el canvi de claus sense revel·lar informació sobre elles. El segon pas és fer el canvi de claus a partir d'aquesta informació auxiliar. Tenim aquests dos procediments:

- $\mathsf{SwitchKeyGen}(\mathbf{s}_1 \in R_q^{n_1}, \mathbf{s}_2 \in R_q^{n_2})$:
  1. $\mathbf{A} \leftarrow \mathsf{E.PublicKeyGen}(\mathbf{s}_2, n_1\lceil \log_2 q \rceil)$,
  2. $\mathbf{B} = \mathbf{A} + \mathsf{PowersOf2}(\mathbf{s}_1, q)$ (Suma $\mathsf{PowersOf2}(\mathbf{s}_1, q)$ a la primera columna de $\mathbf{A}$.).

  Retorna $\tau_{\mathbf{s}_1 \to \mathbf{s}_2} = \mathbf{B}$.
- $\mathsf{SwitchKey}(\tau_{\mathbf{s}_1 \to \mathbf{s}_2}, \mathbf{c}_1)$: Retorna $\mathbf{c}_2 = \mathsf{BitDecomp}(\mathbf{c}_1, q)^T B \in R_q^{n_2}$.

Lema 5.2. *Siguin $\boldsymbol{s}_1 \in R_q^{n_1}, \boldsymbol{s}_2 \in R_q^{n_2}, \tau_{\boldsymbol{s}_1 \to \boldsymbol{s}_2} \leftarrow \mathsf{SwitchKeyGen}(\boldsymbol{s}_1, \boldsymbol{s}_2)$. Siguin $\boldsymbol{c}_1 \in R^{n_1}$ i $\boldsymbol{c}_2 = \mathsf{SwitchKey}(\tau_{\boldsymbol{s}_1 \to \boldsymbol{s}_2}, \boldsymbol{c}_1)$. Aleshores*

$$\langle \boldsymbol{c}_2, \boldsymbol{s}_2\rangle \equiv 2\langle \mathsf{BitDecomp}(\boldsymbol{c}_1), \boldsymbol{e}_2\rangle + \langle \boldsymbol{c}_1, \boldsymbol{s}_1\rangle \mod q,$$

on $\boldsymbol{A}\boldsymbol{s}_2 = 2\boldsymbol{e}_2 \in R^{n_1\lceil \log_2 q \rceil}$, amb $\boldsymbol{A}$ la matriu intermèdia de $\mathsf{SwitchKeyGen}(\boldsymbol{s}_1)$.

Demostració.

$$\langle \mathbf{c}_2, \mathbf{s}_2\rangle = \mathsf{BitDecomp}(\mathbf{c}_1)^T \mathbf{B}\mathbf{s}_2 = \mathsf{BitDecomp}(\mathbf{c}_1)^T(2\mathbf{e}_2 + \mathsf{PowersOf2}(\mathbf{s}_1))$$
$$= 2\langle \mathsf{BitDecomp}(\mathbf{c}_1), \mathbf{e}_2\rangle + \langle \mathsf{BitDecomp}(\mathbf{c}_1), \mathsf{PowersOf2}(\mathbf{s}_1)\rangle$$
$$= 2\langle \mathsf{BitDecomp}(\mathbf{c}_1), \mathbf{e}_2\rangle + \langle \mathbf{c}_1, \mathbf{s}_1\rangle.$$

□

Per tant, $\mathbf{c}_2$ encripta el mateix missatge que $\mathbf{c}_1$ però sota la clau $\mathbf{s}_2$, amb un petit augment de l'error. Aquest procediment s'utilitza per reduir la dimensió dels textos xifrats després de les operacions de multiplicació.

5.5 Canvi de mòdul

Hem vist que calcular homomòrficament el producte de dos textos xifrats implica augmentar l'error de forma quadràtica. Aquest fet limita molt la profunditat dels circuits que podrem avaluar. Per solucionar aquest problema s'introdueix la tècnica següent de canvi de mòdul.

Definició 5.3. *Sigui $\boldsymbol{x} \in \mathbb{Z}^n$ i $q > p > r$. Definim $\boldsymbol{x}' = \mathsf{Scale}(x, q, p, r)$ com el vector més pròxim a $(p/q)\boldsymbol{x}$ que satisfà que $\boldsymbol{x}' \equiv \boldsymbol{x} \mod r$.*

Definició 5.4. *Sigui $\boldsymbol{s} \in R^n$. Es defineix la norma $\ell_1^{(R)}$ com*

$$\ell_1^{(R)}(\boldsymbol{s}) = \sum_{i=0}^{n} \|s_i\|,$$

on $\|s_i\|$ és la norma euclidiana del polinomi s_i.

Lema 5.5. *Sigui $f(x) = x^{2^d} + 1$ i $R = \mathbb{Z}[x]/(f)$. Siguin $q > p > r > 0 \in \mathbb{Z}$ tals que $q \equiv p \equiv 1 \mod r$. Sigui $\boldsymbol{c} \in R^n$ i $\boldsymbol{c}' = \mathsf{Scale}(\boldsymbol{c}, q, p, r)$. Aleshores, per a qualsevol $\boldsymbol{s} \in R^n$ amb $\|[\langle \boldsymbol{c}, \boldsymbol{s}\rangle]_q\| < q/2 - (q/p)(r/2)\sqrt{2^d} \cdot \gamma_R \cdot \ell_1^{(R)}(\boldsymbol{s})$, tenim que*

$$[\langle \boldsymbol{c}', \boldsymbol{s}\rangle]_p = [\langle \boldsymbol{c}, \boldsymbol{s}\rangle]_q \bmod r \text{ i } \|[\langle \boldsymbol{c}', \boldsymbol{s}\rangle]_p\| < (p/q)\|[\langle \boldsymbol{c}, \boldsymbol{s}\rangle]_q\| + (r/2) \cdot \sqrt{2^d} \cdot \gamma_R \cdot \ell_1^{(R)}(\boldsymbol{s}).$$

Demostració. Sabem que $[\langle \mathbf{c}, \mathbf{s}\rangle]_q = \langle \mathbf{c}, \mathbf{s}\rangle - kq$, per a algun $k \in R$. Sigui ara $e_p = \langle \mathbf{c}', \mathbf{s}\rangle - kp \in R$. Resulta evident que $e_p \equiv [\langle \mathbf{c}', \mathbf{s}\rangle]_p \bmod p$. Volem veure que $e_p = [\langle \mathbf{c}', \mathbf{s}\rangle]_p$. Tenim que

$$\begin{aligned}
\|e_p\| &= \|-kp + \langle (p/q)\mathbf{c}, \mathbf{s}\rangle + \langle \mathbf{c}' - (p/q)\mathbf{c}, \mathbf{s}\rangle\| \\
&\leq \|-kp + \langle (p/q)\mathbf{c}, \mathbf{s}\rangle\| + \|\langle \mathbf{c}' - (p/q)\mathbf{c}, \mathbf{s}\rangle\| \\
&\leq (p/q)\|[\langle \mathbf{c}, \mathbf{s}\rangle]_q\| + \gamma_R \sum_{i=1}^{n} \|c_i' - (p/q)c_i\|\|s_i\| \\
&\leq (p/q)\|[\langle \mathbf{c}, \mathbf{s}\rangle]_q\| + \gamma_R(r/2) \cdot \sqrt{2^d} \cdot \ell_1^{(R)}(\mathbf{s}) \\
&< p/2.
\end{aligned}$$

A més a més, tenim que

$$[\langle \mathbf{c}', \mathbf{s}\rangle]_p = e_p = \langle \mathbf{c}', \mathbf{s}\rangle - kp \equiv \langle \mathbf{c}, \mathbf{s}\rangle - kq = [\langle \mathbf{c}, \mathbf{s}\rangle]_q \mod r.$$

□

Per tant, podem canviar el mòdul intern de l'equació de desxifrat mantenint l'esquema correcte sota la mateixa clau secreta. Aplicant aquest procediment, un avaluador que no coneix la clau secreta sinó només una cota superior de la seva norma pot canviar un text xifrat $\mathbf{c}$ que es desxifra amb $\mathbf{s}$ mòdul q a un text xifrat $\mathbf{c}'$ que es desxifra mòdul p. El més interessant és que l'error també es redueix.

Corol·lari 5.6. *Siguin p i q dos enters imparells. Suposem que $\boldsymbol{c}$ és un xifratge de m sota la clau secreta $\boldsymbol{s}$ i amb el mòdul q, és a dir $m = [[\langle \boldsymbol{c}, \boldsymbol{s}\rangle]_q]_r$. Sigui $e_q = [\langle \boldsymbol{c}, \boldsymbol{s}\rangle]_q$ i assumim que $\|e_q\| < q/2 - (q/p)(r/2) \cdot \sqrt{2^d} \cdot \gamma_R \cdot \ell_1^{(R)}(\boldsymbol{s})$. Aleshores, $\boldsymbol{c}' = \mathsf{Scale}(\boldsymbol{c}, q, p, r)$ és un xifratge de m sota la clau $\boldsymbol{s}$ i amb el mòdul p, és a dir, $m = [[\langle \boldsymbol{c}', \boldsymbol{s}\rangle]_q]_r$. L'error $e_p = [\langle \boldsymbol{c}', \boldsymbol{s}\rangle]_p$ del nou text xifrat té magnitud $\|e_p\| \leq (p/q)\|[\langle \boldsymbol{c}, \boldsymbol{s}\rangle]_q\| + \gamma_R(r/2) \cdot \sqrt{2^d} \cdot \ell_1^{(R)}(\boldsymbol{s})$.*

L'error, per tant, augmenta o disminueix de forma proporcional a la raó p/q, excepte per un factor additiu. A priori podríem pensar que no és una tècnica gaire bona, ja que si bé sí que redueix l'error, també redueix el mòdul, que és el sostre màxim d'error permès. Reduir l'error i el sostre de l'error de forma proporcional no sembla que pugui funcionar. Ara bé, el que s'ha de tenir en compte és que en les multiplicacions l'error augmenta de forma exponencial. Per exemple, si tenim un error e i $q \approx e^k$, aleshores podrem fer $\log k$ multiplicacions abans l'error no arribi al sostre. El mètode que podem fer servir per augmentar la profunditat dels circuits que es poden avaluar és utilitzar una escala decreixent de mòduls $\{q_i \approx q/x^i\}$, per $i < k$. Després de multiplicar dos textos xifrats amb el mòdul q_i canviem els textos xifrats al mòdul q_{i+1}, i l'error va de e^2 a e. D'aquesta manera podrem executar uns k nivells de multiplicacions abans d'arribar al sostre d'error. Per tant, utilitzant aquesta tècnica, hem millorat la profunditat dels circuits que es poden avaluar de forma exponencial (de $\log k$ a k).

5.6 Esquema homomòrfic anivellat

Ara modificarem l'esquema presentat anteriorment per convertir-lo en un esquema completament homomòrfic anivellat. Utilitzarem el paràmetre L per denotar la profunditat màxima dels circuits aritmètics que volem que l'esquema pugui avaluar. Procedim a la descripció de l'esquema.

- $\mathsf{FHE.Setup}(\lambda, L, b)$: Pren el paràmetre de seguretat λ, la profunditat L. S'utilitza el bit $b \in \{0,1\}$ per determinar si tenim un esquema basat en el LWE (amb $d = 1$) o un sistema basat en el RLWE (amb $n = 1$). Sigui $\mu = \theta(\log \lambda + \log L)$ un paràmetre que especificarem més endavant. Per a $j \in [0, L]$ executa $params_j = \mathsf{E.Setup}(\lambda, (j+1)\mu, v)$ per obtenir una escala decreixent de mòduls des de q_L amb $(L+1)\mu$ bits, a q_0, amb μ bits. Hem de tenir en compte que $d_j = d$ i $\chi_j = \chi$, és a dir, el grau dels polinomis i la distribució d'error són constants per a tota la instància de l'esquema, no depenen del nivell j.

- $\mathsf{FHE.KeyGen}(\{params_j\})$: Per a $j = L$ fins a 0 es fa:

  1. $\mathbf{s}_j \leftarrow \mathsf{E.SecretKeyGen}(params_j)$ i $\mathbf{A}_j \leftarrow \mathsf{E.PublicKeyGen}(params_j, \mathbf{s}_j)$,
  2. $\mathbf{s}'_j = \mathbf{s}_j \otimes \mathbf{s}_j$,
  3. $\mathbf{s}''_j = \mathsf{BitDecomp}(\mathbf{s}'_j, q_j)$,
  4. $\tau_{\mathbf{s}''_{j+1} \to \mathbf{s}_j} \leftarrow \mathsf{SwitchKeyGen}(\mathbf{s}''_j, \mathbf{s}_{j-1})$. (Ometem aquest pas quan $j = L$.)

  La clau secreta sk consisteix en tots els $\mathbf{s}_j$ i la clau pública pk consisteix en els $\mathbf{A}_j$ i $\tau_{\mathbf{s}''_{j+1} \to \mathbf{s}_j}$.

- $\mathsf{FHE.Enc}(params, pk, m)$: Pren un missatge $m \in R_2$ i retorna el valor de $\mathsf{E.Enc}(\mathbf{A}_L, m)$.

- $\mathsf{FHE.Dec}(params, sk, \mathbf{c}, j)$: Retorna $\mathsf{E.Dec}(\mathbf{s}_j, \mathbf{c})$. S'ha de saber sota quina clau $\mathbf{s}_j$ està xifrat, és a dir, quants nivells ha baixat en l'escala.

- $\mathsf{FHE.Refresh}(\mathbf{c}, \tau_{\mathbf{s}''_j \to \mathbf{s}_{j-1}}, q_j, q_{j-1})$: Pren un text xifrat amb la clau $\mathbf{s}'_j$, la informació auxiliar $\tau_{\mathbf{s}''_j \to \mathbf{s}_{j-1}}$ per fer el canvi de claus i els mòduls actual i els mòduls actual i següent q_j i q_{j-1}. Es fa el següent:

  1. $\mathbf{c}_1 = \mathsf{PowersOf2}(\mathbf{c}, q_j)$. Observem que $\langle \mathbf{c}_1, \mathbf{s}''_j \rangle \equiv \langle \mathbf{c}, \mathbf{s}'_j \rangle \mod q_j$,
  2. Canvi de mòdul: $\mathbf{c}_2 = \mathsf{Scale}(\mathbf{c}_1, q_j, q_{j-1}, 2)$,
  3. Canvi de clau: Retorna $\mathbf{c}_3 = \mathsf{SwitchKey}(\tau_{\mathbf{s}''_j \to \mathbf{s}_{j-1}}, \mathbf{c}_2, q_{j-1})$.

- $\mathsf{FHE.Add}(pk, \mathbf{c}_1, \mathbf{c}_2)$: Pren dos textos xifrats sota la mateixa clau $\mathbf{s}_j$. Posa $\mathbf{c}_3 = \mathbf{c}_1 + \mathbf{c}_2$. Donat que la primera fila (i la primera columna) de $\mathbf{s}'_j = \mathbf{s}_j \otimes \mathbf{s}_j$ és $\mathbf{s}_j$, ja que el primer element de $\mathbf{s}_j$ és 1, es pot interpretar $\mathbf{c}_3$ com un text xifrat sota la clau $\mathbf{s}'_j$, afegint els zeros que calgui. Retorna

$$\mathsf{FHE.Refresh}(\mathbf{c}_3, \tau_{\mathbf{s}''_j \to \mathbf{s}_{j-1}}, q_j, q_{j-1}).$$

- $\mathsf{FHE.Mult}(pk, \mathbf{c}_1, \mathbf{c}_2$: Pren dos textos xifrats sota la mateixa clau $\mathbf{s}_j$ i posa $\mathbf{c}_3 = \mathbf{c}_1 \otimes \mathbf{c}_2$, que és un text xifrat sota la clau $\mathbf{s}'_j = \mathbf{s}_j \otimes \mathbf{s}_j$. Retorna

$$\mathsf{FHE.Refresh}(\mathbf{c}_3, \tau_{\mathbf{s}''_j \to \mathbf{s}_{j-1}}, q_j, q_{j-1}).$$

Per avaluar circuits l'únic que s'ha de fer és anar avaluant les corresponents portes additives i multiplicatives de forma seqüencial amb els procediments FHE.Add i FHE.Mult corresponentment. El pas clau en el sistema homomòrfic anivellat és el procediment Refresh, que manté constant la llargada dels textos xifrats i va reduint l'error després de cada operació. Cal remarcar que, donat que la suma augmenta molt menys l'error que la multiplicació, no és necessari refrescar després de cada suma.
Aquest esquema no utilitza la tècnica del bootstrapping. Com que, al cap i a la fi, $\tau_{\mathbf{s}''_j \to \mathbf{s}_{j-1}}$ no deixa de ser una clau secreta xifrada amb una altra clau secreta, es podria interpretar el procediment SwitchKey com l'avaluació homomòrfica del circuit de desxifrat. Els autors del criptosistema deixen clar que no consideren que aquest mètode es pugui considerar bootstrapping, per diverses raons. El procediment SwitchKeyGen no fa una representació de la clau secreta amb elements de l'espai de textos plans (en aquest cas bits) i després els xifra, tal i com es fa en el bootstrapping, sinó que s'utilitza una

forma alternativa de codificació de la clau secreta que permet al procediment SwitchKey fer el canvi de claus sense necessitat d'avaluar homomòrficament el circuit booleà de desxifrat associat. Això fa el sistema molt més eficient. A més a més, el procediment de canvi de claus no redueix l'error, sinó que es redueix en el procediment de canvi de mòdul.
Les limitacions d'aquest mètode són evidents, ja que si bé el procediment de canvi de claus admetria una construcció circular, el de canvi de mòdul no, i per tant l'esquema sempre tindrà un màxim nombre de nivells avaluables. Per convertir-lo en un sistema completament homomòrfic l'única opció segueix sent la tècnica del *bootstrapping*.

5.7 Ajustament de paràmetres

Per tal d'ajustar els paràmetres veurem quin és l'error inicial dels textos xifrats i com es propaga a través de les operacions homomòrfiques. L'objectiu és mantenir la magnitud de l'error sempre acotada per tal de poder desxifrar. Veiem primer quin és l'error inicial.

Lema 5.7. *Sigui una instància de l'esquema completament homomòrfic anivellat amb mòdul inicial q_L, grau del polinomi 2^d i dimensió n. Sigui B_χ una cota tal que la distribució χ té probabilitat negligible fora d'una bola de radi B_χ. Aleshores, la magnitud de l'error dels textos xifrats retornats per* FHE.Enc *està acotada per* $1 + 2 \cdot \gamma_R \cdot \sqrt{2^d} \cdot ((2n+1)\log q_L) \cdot B_\chi$.

Demostració. Siguin $\mathbf{s} \leftarrow \mathsf{E.SecretKeyGen}$ i $\mathbf{A} \leftarrow \mathsf{E.PublicKeyGen}(\mathbf{s}, N)$ amb $N = (2n+1)\log q_L$. Recordem que $\mathbf{A} \cdot \mathbf{s} = 2\mathbf{e}$ per $e \leftarrow \chi$. Al xifrar un missatge es posa $\mathbf{c} = \mathbf{m} + \mathbf{A}^T\mathbf{r}$, on $\mathbf{r} \in R_2^N$. Aleshores $\|[\langle \mathbf{c}, \mathbf{s}\rangle]_q\| = |[m + 2\langle \mathbf{r}, \mathbf{e}\rangle]_q\| \leq 1 + 2 \cdot \gamma_R \cdot \sum_{i=1}^N \|r_j\| \cdot \|e_j\| \leq 1 + 2\gamma_R \cdot \sqrt{2^d} \cdot N \cdot B_\chi$. □

Analitzem com augmenta l'error en les operacions de suma i producte.

Lema 5.8. *Siguin $\boldsymbol{c}_1$ i $\boldsymbol{c}_2$ dos textos xifrats sota la clau $\boldsymbol{s}_j$ i mòdul q_j, amb $\|[\langle \boldsymbol{c}_i, \boldsymbol{s}_j\rangle]_{q_j}\| \leq B$ i $m_i = [[\langle \boldsymbol{c}_i, \boldsymbol{s}_j\rangle]_{q_j}]_2$. Siguin $\boldsymbol{s}'_j = \boldsymbol{s}_j \otimes \boldsymbol{s}_j$ i $\boldsymbol{c}' = \boldsymbol{c}_1 + \boldsymbol{c}_2$. Afegim zeros a $\boldsymbol{c}'$ per obtenir un vector $\boldsymbol{c}_3$ tal que $\langle \boldsymbol{c}_3, \boldsymbol{s}'_j\rangle = \langle \boldsymbol{c}', \boldsymbol{s}_j\rangle$. Aleshores $\|[\langle \boldsymbol{c}_3, \boldsymbol{s}'_j\rangle]_{q_j}\| \leq 2B$. Si $2B < q_j/2$, $\boldsymbol{c}_3$ és un xifrat de $m_1 + m_2$ sota la clau $\boldsymbol{s}'_j$ i mòdul q_j.*

Demostració.

$$\|[\langle \mathbf{c}_3, \mathbf{s}'_j\rangle]_{q_j}\| = \|[\langle \mathbf{c}', \mathbf{s}_j\rangle]_{q_j}\| = \|[\langle \mathbf{c}_1 + \mathbf{c}_2, \mathbf{s}_j\rangle]_{q_j}\| = \|[\langle \mathbf{c}_1, \mathbf{s}_j\rangle + \langle \mathbf{c}_2, \mathbf{s}_j\rangle]_{q_j}\|$$
$$\leq \|[\langle \mathbf{c}_1, \mathbf{s}_j\rangle]_{q_j}\| + \|[\langle \mathbf{c}_2, \mathbf{s}_j\rangle]_{q_j}\| \leq 2B.$$

□

Lema 5.9. *Siguin $\boldsymbol{c}_1$ i $\boldsymbol{c}_2$ dos textos xifrats sota la clau $\boldsymbol{s}_j$ i mòdul q_j, amb $\|[\langle \boldsymbol{c}_i, \boldsymbol{s}_j\rangle]_{q_j}\| \leq B$ i $m_i = [[\langle \boldsymbol{c}_i, \boldsymbol{s}_j\rangle]_{q_j}]_2$. Siguin $\boldsymbol{s}'_j = \boldsymbol{s}_j \otimes \boldsymbol{s}_j$ i $\boldsymbol{c}_3 = \boldsymbol{c}_1 \otimes \boldsymbol{c}_2$. Aleshores $\|[\langle \boldsymbol{c}_3, \boldsymbol{s}'_j\rangle]_{q_j}\| \leq \gamma_R \cdot B^2$. Si $\gamma_R \cdot B^2 < q_j/2$, $\boldsymbol{c}_3$ és un xifrat de $m_1 \cdot m_2$ sota la clau $\boldsymbol{s}'_j$ i mòdul q_j.*

Demostració.

$$\|[\langle \mathbf{c}_3, \mathbf{s}'_j\rangle]_{q_j}\| = \|[\langle \mathbf{c}_1 \otimes \mathbf{c}_2, \mathbf{s}_j \otimes \mathbf{s}_j\rangle]_{q_j}\| = \|[\langle \mathbf{c}_1, \mathbf{s}_j\rangle \cdot \langle \mathbf{c}_2, \mathbf{s}_j\rangle]_{q_j}\|$$
$$\leq \gamma_R \cdot \|[\langle \mathbf{c}_1, \mathbf{s}_j\rangle]_{q_j}\| \cdot \|[\langle \mathbf{c}_2, \mathbf{s}_j\rangle]_{q_j}\| \leq \gamma_R \cdot B^2.$$

□

Per ajustar els paràmetres del sistema ens basarem en l'evolució de l'error de la multiplicació ja que és el factor limitant. Després de la primera multiplicació l'error està acotat per $\gamma_R \cdot B^2$. En aplicar el procediment Scale l'error queda acotat per $(q_{j-1}/q_j) \cdot \gamma_R B^2 + \eta_{\mathsf{Scale},j}$ per a un cert terme additiu $\eta_{\mathsf{Scale},j}$. Finalment, s'aplica el procediment SwitchKey, que introdueix un altre terme additiu $\eta_{\mathsf{SwitchKey},j}$. Per tant, volem que se satisfaci que

$$(q_{j-1}/q_j) \cdot \gamma_R B^2 + \eta_{\mathsf{Scale},j} + \eta_{\mathsf{SwitchKey},j} \leq B$$

per a tot j.

5.8 *Bootstrapping*

En la seva configuració estàndard, que depèn d'un polinomi definidor $f(x) = x^{2^d} + 1$, el sistema BGV té un espai de textos plans $\mathcal{P} = \mathbb{Z}_2[x]/(f)$ i un espai de textos xifrats $\mathcal{X} = \mathbb{Z}_q[x]/(f)$, per un cert enter q. El primer problema que se'ns planteja és el de codificar elements de $\mathcal{X}$ amb elements de $\mathcal{P}$. La manera més senzilla és tenir en compte la immersió $i : \mathbb{Z}_2 \to \mathcal{P}$ donada per la identitat $i(a) = a$. És a dir, els polinomis constants de $\mathcal{P}$ formen un cos isomorf a $\mathbb{Z}_2$, amb la suma i la multiplicació sent, respectivament, les portes lògiques XOR i AND, a partir de les quals podem construir totes les altres. Per tant, podem descompondre els elements de $\mathcal{X}$ (o de qualsevol altra estructura que vulguem xifrar, per exemple enters) en bits, xifrar cada bit per separat i avaluar homomòrficament circuits binaris, àmpliament estudiats i coneguts. Cada coeficient dels polinomis en $\mathcal{X}$ té $\log q$ bits i n'hi ha 2^d, a partir del qual es veu que el grau de la representació en bits de $\mathcal{X}$ serà $2^d \log q$.
La profunditat màxima que el sistema BGV pot avaluar és lineal en el nombre de bits de q i per tant logarítmica en q. Si trobem un circuit de desxifrat de profunditat sub-logarítmica en q aleshores l'esquema serà *bootstrappable*. El primer pas del desxifrat és calcular $\langle \mathbf{c}, \mathbf{s} \rangle$. En el cas LWE tenim el producte escalar de dos vectors n-dimensionals. Cada terme $c_i \cdot s_i$ es pot veure com una suma de $\log q$ desplaçaments de s_i, on cada desplaçament té una longitud màxima de $2 \log q$ bits. Per tant tenim una suma de $n \log q$ nombres de $2 \log q$ xifres. Tal i com s'analitza al capítol 6, podem calcular aquesta suma amb profunditat $O(\log(n \log q) + \log(\log q)) = O(\log n + \log \log q)$. Per reduir el resultat de la suma mòdul q existeixen circuits de profunditat $O(\log \log q)$. La reducció final mòdul 2 consisteix a descartar els bits més significatius. Conseqüentment, a diferència del que passa amb l'esquema de Gentry, l'esquema BGV és *bootstrappable* sense haver de fer cap modificació a l'algorisme de desxifrat. El cas RLWE és anàleg.

5.9 Modificacions a l'esquema

És fàcil veure que l'esquema BGV és fàcilment adaptable a un anell de textos plans $\mathbb{Z}_p[x]/(f)$, per a algun primer p diferent de 2. El procediment $\mathsf{E.PublicKeyGen}$ s'adapta per tal que $\mathbf{A} \cdot \mathbf{s} = p \cdot \mathbf{e}$ en comptes de $2 \cdot \mathbf{e}$, els mòduls q_i s'agafen tenint en compte que $q_i \equiv q_j \equiv 1 \bmod p$ per a tot $i, j \in [0, L]$ per tal que funcioni el pas de canvi de mòduls i l'equació de desxifrat esdevé $m = [[\langle \mathbf{c}, \mathbf{s} \rangle]_q]_p$. Ara bé, aquest canvi no és gaire interessant, ja que malgrat que estem ampliant l'espai de textos plans finalment sempre ens va bé treballar en base 2, ja que és la base nativa dels ordinadors. La possibilitat existeix, però.
Un canvi més interessant és el del polinomi f. El sistema BGV estàndard treballa amb la família de polinomis $x^{2^d} + 1$, que corresponen als polinomis ciclotòmics $\Phi(m)$ amb $m \geq 2$ una potència de 2. Aquesta família té algunes bones propietats, com el fet de poder augmentar arbitràriament el grau del polinomi (i per tant la seguretat de l'esquema) sense haver de modificar l'algorisme de multiplicació i que va ser la primera família per la qual es va demostrar la seguretat del problema RLWE associat. Ara bé, els problemes d'aquesta família són també evidents. Mòdul 2 el polinomi factoritza com $x^n + 1 \equiv (x + 1)^n \bmod 2$ i per tant l'anell de textos plans $\mathbb{F}_2[x]/(f)$ no només no és un cos sinó que tampoc és un domini d'integritat i l'únic subcòs que té és isomorf a $\mathbb{F}_2$. A la vida real les dades amb les quals treballem i que volem xifrar són enters, nombres de punt flotant, cadenes de text o documents binaris, però rarament polinomis en $\mathbb{F}_2[x]/(f)$. Per tant, a l'hora de representar aquestes dades amb elements de l'anell de textos plans ho haurem de fer bit a bit, "malgastant" la resta de coeficients dels polinomis, fet que produeix una expansió de les dades xifrades molt ineficient.
Siguin p primer i $f, g \in \mathbb{F}_p[X]$ polinomis irreductibles de grau n. Se satisfà que $\mathbb{F}_p[X]/(f) \cong \mathbb{F}_p[X]/(g)$, ja que són dos cossos finits de grau n. Podem representar-los com

$$\mathbb{F}_p[X]/(f) = \left\{ a_0 + a_1\alpha + ... + a_{n-1}\alpha^{n-1} : a_i \in \mathbb{F}_p \text{ i } f(\alpha) = 0 \right\},$$

$$\mathbb{F}_p[X]/(g) = \left\{ b_0 + b_1\beta + ... + b_{n-1}\beta^{n-1} : b_i \in \mathbb{F}_p \text{ i } g(\beta) = 0 \right\}.$$

L'isomorfisme $\sigma_{f \to g}$ entre ells deixa fix $\mathbb{F}_p$ i queda explícitament determinat per la imatge $\sigma_{f \to g}(\alpha)$. Tenint en compte que

$$0 = f(\alpha) = \sigma_{f \to g}(f(\alpha)) = f(\sigma_{f \to g}(\alpha)) = 0$$

aleshores $\sigma_{f \to g}(\alpha)$ és una arrel de f en $\mathbb{F}_p[X](g)$, o dit d'una altra manera, $f(\sigma_{f \to g}(\alpha)) \equiv 0 \bmod g$. Existeixen n isomorfismes $\sigma_{f \to g}$ diferents, ja que $\mathrm{Gal}(\mathbb{F}_{p^n} \mid \mathbb{F}_p)$, que està generat per l'automorfisme

de Frobenius $\varphi_p(x) = x^p$, té ordre n. Per tant, si $f(\sigma_{f\to g}(\alpha)) \equiv 0 \bmod g$ aleshores $f(\varphi_p^i(\sigma_{f\to g}(\alpha))) \equiv 0 \bmod g$ per a tot i. Si tenim una arrel $\sigma_{f\to g}(\alpha)$ de f mòdul g aleshores podem generar tot el conjunt d'arrels fent $\varphi_p^i(\sigma_{f\to g}(\alpha)) \bmod g$ per a $i \in \{0, ..., n-1\}$.
Suposem ara que tenim un polinomi $F(x) \in \mathbb{F}_p[X]$ de grau N que factoritza en r factors irreductibles diferents $F_1(x), ..., F_r(x)$, tots ells de grau $d = N/r$. Pel teorema xinès del residu tenim que

$$\mathbb{F}_p[X]/(F) \cong \mathbb{F}_p[X]/(F_1) \times ... \times \mathbb{F}_p[X]/(F_r) \cong \mathbb{F}_{p^d} \times ... \times \mathbb{F}_{p^d},$$

és a dir, $\mathbb{F}_p[X]/(F)$ és isomorf a r còpies del cos finit $\mathbb{F}_{p^d}$. Com que $\mathbb{F}_p$ és un cos aleshores $\mathbb{F}_p[X]$ és un domini euclidià i per tant podem calcular l'isomorfisme cap a la dreta prenent el residu de la divisió euclidiana entre cada F_i

$$f(x) \mapsto (f(x) \bmod F_1, ..., f(x) \bmod F_r).$$

Calcular l'isomorfisme cap a l'esquerra és equivalent a solucionar un sistema de congruències. Volem trobar $f(x)$ tal que $f(x) \equiv f_1(x) \bmod F_1, ..., f(x) \equiv f_r(x) \bmod F_r$. Definim $H_i = F/F_i$. Fent la descomposició en fraccions parcials obtenim

$$\frac{1}{F(x)} = \sum_{i=1}^{r} \frac{G_i(x)}{F_i(x)},$$

on $G_i(x)$ són polinomis de grau menor que d. A partir d'aquí es dedueix que una solució del sistema de congruències ve donada per

$$f(x) = \sum_{i=1}^{r} f_i(x)G_i(x)H_i(x) \mod F(x). \tag{5.1}$$

Denotem per $\mathbb{K}_n := \mathbb{F}_{p^n}$ amb una certa representació canònica donada pel polinomi irreductible $K(x)$ de grau n. Sabem que si $n \mid d$ aleshores $\mathbb{K}_n$ és un subcòs de $\mathbb{F}_{p^d}$ i hi ha una immersió $\mathbb{K}_n = \mathbb{F}_p[X]/(K) \to \mathbb{F}_p[X]/(F_i)$. Per tant tindrem una immersió

$$\Gamma_{n,l} : \mathbb{K}_n^{l\leq r} \to \mathbb{F}_p[X]/(f)$$

de $l \leq r$ còpies de $\mathbb{K}_n$ a l'espai de textos plans. Aquesta construcció permet fer operacions homomòrfiques en paral·lel, dividint l'anell de textos plans en "calaixos" independents els uns dels altres. A aquest conjunt de tècniques se les anomena SIMD, de l'anglès *Single Instruction, Multiple Data*.
A la pràctica treballarem amb polinomis ciclotòmics $F(x) = \Phi_m(x)$, ja que són els únics pels quals està demostrat que la distribució RLWE és pseudoaleatòria i que per tant poden usar-se en construccions criptogràfiques. El resultat següent dóna sentit a la construcció abstracta que hem fet.

Proposició 5.10. *Siguin p un primer i m un enter no múltiple de p. El polinomi ciclotòmic $\Phi_m(x)$ factoritza sobre $\mathbb{F}_p[X]$ en $\frac{\varphi(m)}{d}$ factors irreductibles diferents de grau d, on d és l'ordre multiplicatiu de p mòdul m, és a dir, el menor enter positiu tal que $p^d \equiv 1 \bmod m$.*

Demostració. Sigui ζ_m una arrel de $\Phi_m(x)$ en un cos de descomposició K del polinomi $\Phi_m(x)$ sobre $\mathbb{F}_p$. L'ordre de ζ_m en el grup multiplicatiu K^* és m. El grau de ζ_m sobre $\mathbb{F}_p$ és l'enter $d \geq 1$ més petit tal que $\zeta_m^{p^d} = \zeta_m$, o el que és el mateix, $\zeta_m^{p^d-1} = 1$. Per tant requerim que $n \mid (p^d - 1)$, o el que és equivalent, $p^d \equiv 1 \bmod n$. □

Si l'objectiu és treballar amb dades expressades en bits ens interessaran aquells polinomis ciclotòmics que factoritzin sobre $\mathbb{F}_2$ en un nombre elevat de factors diferents en comparació amb el seu grau, és a dir, aquells amb un d petit.
A l'article original del BGV [4] només s'analitza el cas dels polinomis ciclotòmics $F(x) = x^r + 1$ amb r una potència de 2. Evidentment, no podem treballar en $\mathbb{F}_2$ ja que $F(x)$ factoritza en $\mathbb{F}_2[X]$ en un sol factor de multiplicitat r i no es pot aplicar el teorema xinès del residu. En canvi, si es pren un primer p tal que $p \equiv 1 \bmod 2r$ i a una arrel $2r$-èsima primitiva de la unitat mòdul p, aleshores $F(x)$ factoritza sobre $\mathbb{F}_p$ en factors lineals diferents dos a dos: $F(x) = \prod_{i=1}^{r}(x - a^{2i+1})$. Recordant que

l'espai de textos plans es $R_p = \mathbb{F}_p[X]/(x^r+1)$ i denotant $\mathfrak{p}_i$ l'ideal generat per $(p, x-a^{2i-1})$ aleshores $R_{\mathfrak{p}_i} = \mathbb{F}_p[X]/(x-a^{2i-1})$ i $R_p \cong R_{\mathfrak{p}_1} \times ... \times R_{\mathfrak{p}_r}$. Podem empaquetar r missatges plans sobre $R_{\mathfrak{p}_i}$ en un sol missatge xifrat i operar amb ells de forma paral·lela, però no poden interactuar entre ells. L'objectiu ara és no només calcular aquest isomorfisme en l'anell de textos plans, sinó també en el de textos xifrats. Volem agafar r textos xifrats que utilitzen només el subcòs $R_{\mathfrak{p}_1}$ i ajuntar-los en un sol text xifrat per poder operar en paral·lel, i després tornar-ho a expandir en r textos xifrats. D'aquesta manera obtindríem l'eficiència de l'empaquetament i la flexibilitat de la individualitat.
Sigui $\sigma_{i\to j} : R \to R$ un automorfisme que envia els elements de $\mathfrak{p}_i$ als elements de $\mathfrak{p}_j$ i deixa fix el terme constant. Si codifiquem un missatge m utilitzant només el terme constant i $m = [[\langle \mathbf{c}, \mathbf{s}\rangle]_q]_{\mathfrak{p}_i}$ aleshores $m = [[\langle \sigma_{i\to j}(\mathbf{c}), \sigma_{i\to j}(\mathbf{s})\rangle]_q]_{\mathfrak{p}_i}$. El nou text xifrat ho està sota una altra clau, i per tant aplicarem el procediment SwitchKey per recodificar-los sota la mateixa clau. Els procediments per empaquetar i desempaquetar textos xifrats són els següents.

- $\mathsf{PackSetup}(\mathbf{s}_1, \mathbf{s}_2)$: Pren dos claus secretes $\mathbf{s}_1$ i $\mathbf{s}_2$. Per a cada $i \in [1, r = 2^d]$ posa $\tau_{\sigma_{1\to i}(\mathbf{s}_1)\to \mathbf{s}_2} \leftarrow \mathsf{SwitchKeyGen}(\sigma_{1\to i}(\mathbf{s}_1), \mathbf{s}_2)$.
- $\mathsf{Pack}(\{\mathbf{c}_i\}_{i=1}^r, \{\tau_{\sigma_{1\to i}(\mathbf{s}_1)\to \mathbf{s}_2}\}_{i=1}^r)$: Pren textos xifrats $\mathbf{c}_1, ..., \mathbf{c}_r$ tals que $m_i = [[\langle \mathbf{c}_i, \mathbf{s}_1\rangle]_q]_{\mathfrak{p}_1}$ i $[[\langle \mathbf{c}_i, \mathbf{s}_1\rangle]_q]_{\mathfrak{p}_j} = 0$ per a tot $j \neq 1$ i la informació auxiliar donada per PackSetup. Per a cada i es fa el següent.
  1. $\mathbf{c}_i^* = \sigma_{1\to i}(\mathbf{c}_i)$,
  2. $\mathbf{c}_i^\dagger \leftarrow \mathsf{SwitchKey}(\tau_{\sigma_{1\to i}(\mathbf{s}_1)\to \mathbf{s}_2}, \mathbf{c}_i^*)$.

  Finalment es retorna $\mathbf{c} \leftarrow \sum_{i=1}^r \mathbf{c}_i^\dagger$. Assumint que l'error no excedeix el límit aleshores tenim que $m_i = [[\langle \mathbf{c}, \mathbf{s}_2\rangle]_q]_{\mathfrak{p}_i}$.
- $\mathsf{UnpackSetup}(\mathbf{s}_1, \mathbf{s}_2)$: Pren dos claus secretes $\mathbf{s}_1$ i $\mathbf{s}_2$. Per a cada $i \in [1, r = 2^d]$ es posa $\tau_{\sigma_{i\to 1}(\mathbf{s}_1)\to \mathbf{s}_2} \leftarrow \mathsf{SwitchKeyGen}(\sigma_{i\to 1}(\mathbf{s}_1), \mathbf{s}_2)$.
- $\mathsf{Unpack}(\mathbf{c}, \{\tau_{\sigma_{i\to 1}(\mathbf{s}_1)\to \mathbf{s}_2}\}_{i=1}^r)$: Pren un text xifrat tal que $m_i = [[\langle \mathbf{c}, \mathbf{s}_1\rangle]_q]_{\mathfrak{p}_i}$ i la informació auxiliar donada per UnpackSetup. Per a tot i es fa el següent.
  1. $\mathbf{c}_i = u \cdot \sigma_{i\to 1}(\mathbf{c})$, on $u \in 1 + \mathfrak{p}_1$ tal que $u \in \mathfrak{p}_j$ per a $j \neq 1$. D'aquesta manera $m_i = [[\langle \mathbf{c}_i, \sigma_{i\to 1}(\mathbf{s}_1)\rangle]_q]_{\mathfrak{p}_1}$ i $[[\langle \mathbf{c}_i, \sigma_{i\to 1}(\mathbf{s}_1)\rangle]_q]_{\mathfrak{p}_j} = 0$ per a tot $j \neq 1$.
  2. $\mathbf{c}_i^* = \mathsf{SwitchKey}(\tau_{\sigma_{i\to 1}(\mathbf{s}_1)\to \mathbf{s}_2}, \mathbf{c}_i)$.

  Retorna els $\mathbf{c}_i$.

L'expansió de la clau pública com a conseqüència d'aplicar aquesta tècnica és considerable. Veiem com podríem aplicar-ho al cas general. Suposem que tenim un polinomi ciclotòmic arbitrari $F(x) = \Phi_m(x) \in \mathbb{Z}[X]$ de grau $N = \varphi(n)$ que factoritza sobre $\mathbb{F}_p[X]$ en r factors irreductibles diferents $F_1(x), ..., F_r(x)$, tots ells de grau $d = N/r$. Denotem $R = \mathbb{Z}[X]/(F)$ i $\mathfrak{p}_i$ l'ideal de R generat per (p, F_i). Ens interessa trobar un automorfisme $\sigma_{i\to j} : R \to R$ tal que $\sigma_{i\to j}(\mathfrak{p}_i) = \mathfrak{p}_j$ i $\sigma_{i\to j}|_{\mathfrak{p}_i} : \mathfrak{p}_i \to \mathfrak{p}_j$ és un isomorfisme. Malauradament, sense imposar que $p \equiv 1 \bmod m$, i per tant que $\Phi_m(x)$ descompongui en factors lineals sobre $\mathbb{F}_p$, l'existència d'aquest automorfisme no és gens clara.
Evidentment, sempre hi ha l'opció d'avaluar homomòrficament l'isomorfisme induït pel teorema xinès del residu. Si tenim que $f \equiv f_i \bmod F_i$ i xifrats de f_i aleshores podem calcular un xifrat de f avaluant homomòrficament la suma (5.1), que evidentment requerirà tenir un xifrat dels G_i i dels H_i. Aquesta suma es pot calcular fàcilment amb profunditat logarítmica en r. El procés invers, el de calcular $f \bmod F_i$, és més complex, ja que implica avaluar homomòrficament una divisió entera de polinomis, però es pot fer igualment mitjançant un circuit de sumes i multiplicacions.

5.10 *Bootstrapping* millorat

S'han fet múltiples propostes d'ajustar els paràmetres de l'esquema BGV per tal de que sigui més senzill fer el procés de *bootstrapping*. En l'article de Gentry, Halevi i Smart [10] es proposa, entre d'altres coses, treballar amb un mòdul $q = 2^r + 1$. Veiem-ne les implicacions.

Notació 7. *Sigui a un enter. Denotem per $a\langle i\rangle$ el i-èsim bit de la descomposició binària de a. Per a un vector d'enters $\boldsymbol{a} = (a_0, ..., a_n)$ denotem $\boldsymbol{a}\langle i\rangle = (a_0\langle i\rangle, ..., a_n\langle i\rangle)$.*

Lema 5.11. *Siguin $r \geq 3$, $q = 2^r + 1$ i z un enter tal que $|z| < \frac{q^2}{4} - q$ i $[z]_q \in (-\frac{q}{4}, \frac{q}{4})$. Denotem $z' = z + (q^2 - 1)/4$. Aleshores $[[z]_q]_2 = z'\langle r\rangle \oplus z'\langle 0\rangle$.*

Per tant, si z satisfà aquestes condicions, podem calcular $[[z]_q]_2$ molt fàcilment, com la suma del bit 0 i el bit r de z'.
Un altre article interessant és el d'Emmanuela Orsini, Joop van de Pol i Nigel P. Smart [17], del 2014. En aquest article busquen una manera eficient de fer el pas de *bootstrapping* per una tria més àmplia de p i de q.

5.11 Més enllà: múltiples usuaris

Una instància de l'esquema BGV té una sola clau pública i una sola clau privada. Si múltiples usuaris volen xifrar dades i enviar-les per ser processades entre elles aleshores han de compartir clau, i això pot no ser sempre desitjable. En el context de la criptografia homomòrfica ens podem interessar, per tant, per la construcció d'un esquema que tingui diverses claus públiques i privades. D'aquesta manera múltiples usuaris podrien processar les seves dades en conjunt sense comprometre'n la seguretat individual. Existeix un esquema basat en el BGV que permet fer construccions d'aquest tipus, inventat per Long Chen, Zhenfeng Zhang i Xueqing Wang [5], però el seu anàlisi queda fora de l'abast d'aquest treball.

6 Implementació del criptosistema BGV

Presento aquí la meva implementació de l'esquema BGV i els diferents algorismes i consideracions que he tingut en compte. El llenguatge escollit ha estat el C++, per la seva eficiència i flexibilitat. He intentat limitar l'ús de llibreries de tercers al mínim, ja que part del treball consisteix, precisament, a estudiar els diferents algorismes i mètodes necessaris per implementar un esquema d'aquestes característiques, més que no pas fer una implementació altament eficient basada en llibreries professionals d'àlgebra i càlcul numèric.
Un programa informàtic no està mai complet; sempre es pot ampliar, millorar i optimitzar més. No pretenc que la meva implementació sigui cap meravella, però com a mínim sí que he tingut en compte no perdre eficiència innecessàriament, especialment tenint en compte que el que s'ha d'implementar ja és intrínsecament ineficient. El codi font es pot trobar a l'apèndix del treball.

6.1 PRNG

Com que el criptosistema BGV utilitza distribucions de probabilitat durant la generació de claus i el procés de xifrat, és necessari tenir un generador de nombres pseudo-aleatoris (PRNG en les seves sigles en anglès) que sigui criptogràficament segur. No tenim el temps ni l'espai d'estudiar la teoria de generadors aleatoris a fons, però la diferència essencial entre un generador pseudo-aleatori normal i un de criptogràficament segur és que, no només la seva sortida ha de passar tot un seguit de tests estadístics d'aleatorietat, sinó que a més a més ha de ser impredictible endavant i enrere, és a dir, cap atacant no pot calcular en temps polinòmic un determinat bit de la seqüència coneixent els que el precedeixen o el segueixen.
El llenguatge C++ té, en la seva llibreria estàndard random, un seguit de generadors de nombres pseudo-aleatoris com el Mersenne Twister i el Ranlux, molt utilitzats en simulacions per les seves bones propietats estadístiques, però cap d'ells no és criptogràficament segur. Per a aquesta implementació he decidit utilitzar, per les raons següents, el generador ISAAC, dissenyat per Robert J. Jenkins Jr. [12] el 1996.

- Malgrat que no és el més conegut i utilitzat en la indústria, en el quasi un quart de segle de la seva existència no s'ha publicat cap atac viable que trenqui la seva seguretat.
- A la pàgina web de l'autor es pot trobar la implementació de referència en C escrita per ell mateix, fet que simplifica la seva incorporació al projecte.
- És significativament ràpid, generant un nombre de 64 bits cada 19 instruccions. En un senzill test que he fet, el Mersenne Twister mt19937_64 triga $5s$ a generar 10^9 nombres de 64 bits mentre que l'ISAAC triga $3.5s$.

La llavor de l'ISAAC, en la seva versió per defecte, són 256 blocs de 64 bits. Els inicialitzem amb valors procedents del random_device, que és un generador aleatori no determinista la implementació del qual depèn del sistema operatiu i de diferents fonts d'entropia. És important tenir en compte que el random_device, mentre que és òptim per inicialitzar altres generadors, no és una bona idea utilitzar-lo per si sol, ja que no només la seva complexitat és desconeguda, sinó que no hi ha cap garantia que el sistema operatiu tingui l'entropia suficient per generar un nombre arbitràriament gran de nombres aleatoris.
Per tal de generar nombres aleatoris en l'interval $[0, q)$ uniformement, amb q un enter de b bits, el que farem serà generar nombres aleatoris de b bits fins que un d'ells sigui menor que q. En el pitjor dels casos s'haurà de fer una mitjana de dues extraccions per cada nombre aleatori desitjat. És el mètode més eficient que garanteix que la distribució resultant serà uniforme.

6.2 Aritmètica de Montgomery

El sistema BGV treballa amb polinomis de coeficients en cossos finits de la forma $\mathbb{Z}/p\mathbb{Z}$, amb p primer. Mentre que la suma i la resta són operacions trivials d'implementar, amb la multiplicació comencem a tenir problemes. Suposem que volem calcular $a \cdot b \bmod n$. El procediment més directe seria calcular primer el producte $a \cdot b$ i després reduir mòdul n. Això té dos problemes:

1. Si n és un nombre de b bits el resultat de $a \cdot b \bmod n$ també tindrà b bits, però la multiplicació intermèdia en necessitarà $2b$. Per tant, a priori només podríem treballar amb mòduls menors que l'arrel quadrada de l'enter màxim que permet el tipus de variable que tinguem, és a dir, la meitat de bits.

2. Reduir el resultat de la multiplicació mòdul n implica que l'ordinador ha de fer una divisió entera, que és un procés llarg i costós en comparació amb les altres operacions.

Per solucionar el primer problema podem utilitzar una llibreria d'aritmètica de precisió arbitrària, com la GNU Multiple Precision Arithmetic Library, però augmenta considerablement el nombre d'assignacions de memòria ja que no tenim tots els coeficients en blocs contigus i acaba causant una pèrdua considerable de rendiment. L'opció següent és utilitzar una llibreria d'aritmètica de precisió arbitrària pero fixada en el moment de compilació, com la Boost Multiprecision. Això millora el rendiment i l'administració de memòria, però segueix sent més lent que els tipus intrínsecs en el llenguatge, de manera que, en els casos possibles, he intentat evitar l'ús de cap llibreria. En el cas d'enters de 64 bits, el MSVC té la funció `__umulh` que retorna els 64 bits superiors del producte de dos enters sense signe de 64 bits. Els 64 bits inferiors es poden obtenir fent el producte normal. En el cas del GCC tenim el tipus `__int128` que permet treballar amb enters de 128 bits de forma nativa, i per tant podem calcular el producte de dos enters de 64 bits sense problema. Per calcular el producte de dos enters de 128 bits ho podem fer separant en blocs.

$$a \cdot b = (a_1 2^{64} + a_2)(b_1 2^{64} + b_2) = a_1 b_1 2^{128} + (a_1 b_2 + b_1 a_2) 2^{64} + a_2 b_2.$$

A partir d'aquí és fàcil treure els 128 bits superiors o inferiors. Per treballar amb enters de més de 64 bits en el MSVC o enters de més de 128 bits en el GCC ens veiem obligats a utilitzar llibreries.
Per solucionar el segon problema utilitzarem el que s'anomena *multiplicació de Montgomery*, en honor a Peter Lawrence Montgomery [16] que va introduir la tècnica el 1985. L'objectiu és multiplicar dos nombres mòdul n evitant dividir entre n. Prenem un enter $r > n$ tal que $\gcd(r, n) = 1$ i siguin r^{-1} i n^{-1} els seus inversos multiplicatius en els rangs $0 < r^{-1} < n$ i $0 < n^{-1} < r$.

Definició 6.1. *Anomenem la* forma de Montgomery *de $a \in \mathbb{Z}_n$ al producte $\overline{a} = ar \bmod n$.*

Com que r és invertible mòdul n la multiplicació per r és invertible i podem recuperar a a partir de $\overline{a}$. És trivial veure que la suma de dos nombres en forma de Montgomery és la forma de Montgomery de la suma.

$$\overline{a} + \overline{b} = ar + br = (a + b)r = \overline{a + b} \mod n.$$

Ara bé, amb el producte no passa el mateix ja que tindríem un factor r extra. Per tant es defineix la multiplicació de Montgomery com

$$\overline{a} * \overline{b} = \overline{a}\overline{b}r^{-1} = (ar)(br)r^{-1} = abr = \overline{ab} \mod n.$$

Per tant, per calcular el producte de dos nombres en la forma de Montgomery primer els hem de multiplicar entre ells i després multiplicar-ho per r^{-1} i reduir mòdul n. Pot semblar un procediment més complicat que el que teníem de partida però això ho soluciona l'algorisme REDC. L'algorisme REDC (algoritme 1) pren un $0 \leq T < rn$, el multiplica per r^{-1} i redueix mòdul n. Notem que

Algorisme 1 Reducció de Montgomery

```
1: procedure REDC(T)
2:     m ← (T mod r)n^{-1} mod r
3:     t ← (T − mn)/r
4:     if t < 0 then
5:         return t + n
6:     else
7:         return t
8:     end if
9: end procedure
```

$x \bmod r$ significa reduir x al rang $[0, r)$. Veiem que l'algorisme REDC és correcte.

Teorema 6.2. *L'algorisme 1 és correcte.*

Demostració. Observem que $mn \equiv Tn^{-1}n \equiv T \bmod r$ i per tant t és un enter. A més a més $tr \equiv T \bmod n$ i per tant $t \equiv Tr^{-1} \bmod n$ tal i com volem. Finalment, tenim que $-rn \leq T + mn < rn$ i aleshores $-n \leq t < n$. □

Queda vist doncs que, donada la informació auxiliar n^{-1}, podem calcular $Tr^{-1} \bmod n$ sense haver de dividir entre n sinó només entre r. Podria semblar que tornem a estar al lloc de partida però la clau està en escollir r de tal manera que les operacions de divisió entre r i mòdul r siguin trivials computacionalment. A la pràctica això implica escollir com a r una potència de 2 de la mida del tipus de variable que tinguem. Per exemple, si treballem amb variables de 64 bits el més eficient és escollir $r = 2^{64}$, de tal manera que l'operació mòdul r és totalment prescindible, ja que després de fer qualsevol multiplicació sempre ens quedaran els 64 bits inferiors i dividir entre r implica descartar els 64 bits inferiors. Per tant podem multiplicar i reduir mòdul n sense haver de fer cap divisió. Notem que r i n han de ser primers entre sí; en aquest cas implica que només podem treballar amb mòduls n senars. Això no ens suposa cap problema ja que sempre treballem amb mòduls senars (normalment primers).
Analitzem ara un parell de detalls. El primer és com calculem r^{-1} i n^{-1}. Donat que són representats de les solucions de l'equació $rr^{-1} + nn' = 1$ podríem pensar que el més senzill és calcular-los a través de l'algorisme complet d'Euclides però, donat que r és una potència de 2 i que realment no necessitem conèixer el valor de r^{-1} a l'hora d'implementar l'algorisme, podem calcular n^{-1} molt més fàcilment utilitzant el lema següent.

Lema 6.3. *Donats $a, x, k \in \mathbb{Z}$ tenim*

$$ax \equiv 1 \bmod 2^k \iff ax(2 - ax) \equiv 1 \bmod 2^{2k}.$$

Demostració. $ax \equiv 1 \bmod 2^k \Leftrightarrow ax = 1 + m \cdot 2^k$ per a algun $m \in \mathbb{Z}$. Aleshores $ax(2 - ax) = (1 + m \cdot 2^k)(1 - m \cdot 2^k) = 1 + m^2 \cdot 2^{2k} \equiv 1 \bmod 2^{2k}$. □

Podem començar amb $n_1^{-1} = 1$ com l'invers de n mòdul 2^1 i aplicar el lema successivament fins a tenir la inversa de n mòdul la potència de 2 que ens interessa. Aquest procediment és senzill i fàcil d'implementar.
El segon detall és com augmentem un nombre a la seva forma de Montgomery. Calcular $ar \bmod n$ de forma directa implica dividir entre n que és precisament el que volem evitar. Veiem que

$$a * r^2 \equiv ar^2r^{-1} \equiv ar \equiv \overline{a} \bmod n.$$

Per tant per calcular la forma de Montgomery d'un nombre fem la multiplicació de Montgomery per $r^2 \bmod n$, que és una constant que hem de tenir inicialitzada. Per calcular $r^2 \bmod n$ tenim també un truc senzill d'implementar. Inicialitzem $x_1 = -n\%n$ com a enter sense signe, que és igual a $r - n \equiv r \bmod n$, el multipliquem per 2 obtenint així $2r = \overline{2} \bmod n$ i l'anem multiplicant per ell mateix fins a obtenir $r^2 \bmod n$.
Tenim dues maneres d'incorporar l'aritmètica de Montgomery a la implementació del sistema BGV. La primera seria tenir les claus i els missatges plans en la forma de Montgomery, fer tots els passos intermedis en la forma augmentada, i reduir com a últim pas del procés de xifrat. Aquest mètode, no obstant, presenta alguns problemes, ja que no tots els passos són invariants respecte la forma de Montgomery. Per exemple, en el procediment BitDecomp, l'expressió binària d'un nombre en la forma de Montgomery no és la forma de Montgomery de l'expressió binària del nombre; el procediment Scale tampoc funciona, ja que si $\overline{a} \equiv \overline{b} \bmod r$ no podem assegurar que a i b siguin congruents mòdul r. Per tant, per aplicar aquest mètode s'hauria d'anar augmentant i reduint allà on toca i hi ha més possibilitat d'error. El segon mètode consisteix a utilitzar només la forma de Montgomery per multiplicar. Quan hem de multiplicar dos polinomis els augmentem, fem la multiplicació de Montgomery, i els reduïm. Pot semblar ineficient haver d'augmentar i reduir cada vegada, però a la pràctica la diferència és mínima.

6.3 Multiplicació de polinomis

La multiplicació estàndard de dos polinomis de grau n té una complexitat de $O(n^2)$, ja que cada coeficient del primer polinomi s'ha de multiplicar per tots els coeficients del segon polinomi. En aquesta secció estudiarem l'anomenada Transformada Ràpida de Fourier (FFT) que ens permetrà calcular el producte de dos polinomis de coeficients en un cos finit $\mathbb{Z}_p$ amb complexitat $O(n \log n)$.

Definició 6.4. *Siguin R un anell commutatiu, n un enter positiu i $\omega \in R$. Diem que ω és una arrel n-èsima de la unitat si $\omega^n = 1$. Diem que ω és una arrel n-èsima* primitiva *de la unitat si $\omega^n = 1$ i per a cada divisor primer $t \mid n$ es té que $\omega^{n/t} - 1$ no és ni zero ni un divisor de zero.*

Si R és un domini d'integritat aleshores aquesta última condició esdevé $\omega^{n/t} \neq 1$. En el que segueix, n és un enter positiu i $\omega \in R$ una arrel primitiva n-èsima de la unitat.

Definició 6.5. *Sigui $m(x) = x^n - 1 \in R[X]$. Es defineix la* Transformada Discreta de Fourier *com l'aplicació*

$$\begin{aligned} DFT : R[X]/(m) &\to R^n \\ f &\mapsto (f(1), f(\omega), f(\omega^2), ..., f(\omega^{n-1})). \end{aligned}$$

Teorema 6.6. *La Transformada Discreta de Fourier és un isomorfisme d'anells, on la suma i la multiplicació en R^n es fan component a component.*

La complexitat de multiplicar dos vectors en R^n és de $O(n)$. Per tant, si tenim un mètode sub-quadràtic per calcular la Transformada Discreta de Fourier i la seva inversa, podem calcular el producte de dos polinomis en temps sub-quadràtic. Aquest mètode és la Transformada Ràpida de Fourier, en concret l'algorisme de Cooley–Tukey. Suposem ara que n és un enter parell i que volem calcular la DFT d'un polinomi f. Si considerem les divisions enteres $f = q_0 \cdot (x^{n/2} - 1) + r_0$ i $f = q_1 \cdot (x^{n/2} + 1) + r_1$, on el grau de q_0, q_1, r_0, r_1 és menor que $n/2$ aleshores per a $0 \leq i < n/2$ obtenim que

$$\begin{aligned} f(\omega^{2i}) &= q_0(\omega^{2i}) \cdot (\omega^{ni} - 1) + r_0(\omega^{2i}) = r_0(\omega^{2i}), \\ f(\omega^{2i+1}) &= q_1(\omega^{2i+1}) \cdot (\omega^{ni}\omega^{n/2} + 1) + r_1(\omega^{2i+1}) = r_1(\omega^{2i+1}). \end{aligned}$$

Per tant per calcular la DFT de f només és necessari avaluar r_0 en les potències parelles de ω i r_1 en les potències imparelles. Donat que si ω és una arrel primitiva n-èsima de la unitat ω^2 és una arrel $n/2$-èsima de la unitat, aleshores prenent n una potència de 2 podem aplicar un algorisme recursiu per calcular la DFT. Per calcular r_0 i r_1 ho podem fer fàcilment veient que si $f = f_1 x^2 + f_0$ amb f_0 i f_1 de graus menors que n aleshores $r_0 = f_0 + f_1$ i $r_1 = f_0 - f_1$. Podem simplificar una mica més l'algorisme posant $r_1^*(\omega^{2i}) = r_1(\omega^{2i+1})$, i així r_0 i r_1^* s'avaluen en els mateixos punts. L'algorisme 2 mostra la Transformada Ràpida de Fourier en la seva variant recursiva, que és més senzilla conceptualment. A la pràctica la implemento iterativament perquè és més eficient.

En el nostre cas particular estem interessats a multiplicar polinomis en $\mathbb{F}_p[X]$, per a un cert primer

Algorisme 2 FFT de Cooley–Tukey

```
1: procedure FFT(f, ω)
2:     if n = 1 then return f_0
3:     end if
4:     r_0 = Σ_{i=0}^{n/2-1} (f_i + f_{i+n/2} x^i) x^i
5:     r_1^* = Σ_{i=0}^{n/2-1} ω^i (f_i - f_{i+n/2} x^i) x^i
6:     R_{2i} = FFT(r_0, ω^2)
7:     R_{2i+1} = FFT(r_1^*, ω^2)
8:     return (R_0, R_1, ..., R_n) = (r_0(1), r_1^*(1), r_0(ω^2), r_1^*(ω^2), ..., r_0(ω^{n-2}), r_1^*(ω^{n-2}))
9: end procedure
```

p. A diferència del cos $\mathbb{C}$, on existeixen arrels n-èsimes de la unitat per a qualsevol n, en el cos finit $\mathbb{F}_p$ aquest no és el cas.

Proposició 6.7. *Sigui p primer. Si el cos $\mathbb{Z}_p$ té una arrel primitiva n-èsima de la unitat aleshores $n \mid (p-1)$.*

Si volem multiplicar dos polinomis de grau màxim $n = 2^d$ el resultat tindrà grau màxim 2^{d+1}. Per tal de garantir l'existència d'una arrel 2^{d+1}-èsima de la unitat treballarem amb primers de la forma $p = a \cdot 2^{d+1} + 1$, amb a enter. Hem de tenir en compte aquesta condició a l'hora de generar l'escala de mòduls. Veiem ara com trobar arrels n-èsimes primitives de la unitat.

Proposició 6.8. *Sigui α una arrel primitiva mòdul p, és a dir, un generador del grup multiplicatiu cíclic $\mathbb{F}_p^*$. Aleshores $\omega = \alpha^{(p-1)/n}$ és una arrel n-èsima primitiva de la unitat en $\mathbb{F}_p$.*

El problema és ara trobar una arrel primitiva mòdul p. Tenim el resultat següent.

Proposició 6.9. *Un $\alpha \in \mathbb{F}_p^*$ és un generador de $\mathbb{F}_p^*$ si i només si per a tot factor primer $t \mid (p-1)$ se satisfà que $\alpha^{(p-1)/t} \neq 1$.*

Recordem que perquè se satisfaci la seguretat del problema de decisió del RLWE el polinomi ciclotòmic $\Phi_m(x)$ amb què treballem ha de descompondre en factors lineals en $\mathbb{F}_p[X]$, que implica que $p \equiv 1 \bmod m$. Per tant, una altra condició a l'hora de generar l'escala de mòduls és que $m \mid a \cdot 2^{d+1}$. Perquè sigui fàcil trobar una arrel primitiva mòdul p, que equival a factoritzar $p-1$, prendrem primers de la forma $p = q \cdot \operatorname{lcm}(m, 2^{d+1}) + 1$, amb q també primer.
En conclusió, hem vist que per una tria adequada de mòduls podem calcular el producte de dos polinomis amb complexitat $O(n \log n)$. En el meu programa implemento la Transformada Ràpida de Fourier fent les multiplicacions mitjançant el sistema de Montgomery.

6.4 Miller-Rabin

Per generar l'escala de mòduls necessitem un mètode per trobar nombres primers. Fer divisions successives fins l'arrel quadrada és més o menys viable amb nombres de fins a 64 bits, però més enllà es converteix en una tasca impossible. Conseqüentment, utilitzarem un test probabilístic de primeritat per generar nombres primers. El test de Miller-Rabin va ser introduït el 1977 per Michael Rabin [18] a partir del treball anterior de Gary Miller.

Definició 6.10. *Sigui p un enter senar i $p - 1 = 2^r \cdot d$, amb $r \geq 1$ i d senar. Donat un enter $1 \leq b < p$, si se satisfà una de les condicions*

- $b^{n-1} \not\equiv 1 \mod p$,
- $\exists i \in [0, r]$ *tal que* $1 < \gcd(b^{2^i \cdot d} - 1, p) < p$,

direm que b és un testimoni que p és compost, i ho denotarem per $W_p(b)$.

Està clar que si $W_p(b)$ se satisfà per a algun b aleshores p és compost. La primera condició és el teorema petit de Fermat i la segona indica que p té un divisor propi. El teorema clau que dóna validesa al test és el següent.

Teorema 6.11. *Si $4 < p$ és un enter compost, aleshores*

$$\frac{3}{4}(p-1) \leq \#\{b \mid W_p(b)\}.$$

El que ens diu és que, donat un nombre compost p, almenys tres quartes parts dels nombres $1 \leq b < p$ són testimonis que p és compost. Per tant, fent n iteracions amb nombres b_i aleatoris, la probabilitat de classificar erròniament un nombre compost com a primer és $(1/4)^n$, que ràpidament esdevé una quantitat negligible. Per implementar el test es necessita una manera eficient de fer exponenciació modular i ho aconseguim utilitzant l'algorisme binari d'exponenciació juntament amb la multiplicació de Montgomery.

6.5 Procediment Scale

En el procediment Scale del sistema BGV, donats primers $p < q$ i un enter $a < q$ hem de trobar $x = \lfloor \frac{ap}{q} \rfloor$. Una possibilitat és utilitzar aritmètica de punt flotant, però amb enters grans perdem

precisió i el resultat és incorrecte. L'opció següent és calcular ap en un enter de doble precisió i després fer la divisió entera entre q, però és lent i complex fer una divisió amb un enter partit en les meitats alta i baixa. El que farem serà veure que $ap = xq + r$ amb $0 < r < q$, ja que $q \nmid ap$. Aleshores calculem $r \equiv ap \bmod q$ i $x \equiv (-r)q^{-1} \bmod p$. Les operacions mòdul p i q són senzilles utilitzant el sistema de Montgomery i per trobar $q^{-1} \bmod p$ apliquem el Teorema Petit de Fermat $q^{-1} \equiv q^{p-2} \bmod p$ utilitzant exponenciació binària. Com que per a cada parella (p, q) s'ha d'aplicar aquest procediment a molts coeficients no és necessari calcular q^{-1} cada vegada.

6.6 Operacions binàries

Donat que per la naturalesa de la criptografia homomòrfica ens trobem amb la necessitat d'implementar circuits binaris en *software* hem de tenir en compte una sèrie de mètodes que en circumstàncies normals delegaríem a l'electrònica de l'ordinador. Suposem que tenim dos enters de n bits $a = a_0 + a_1 \cdot 2 + ... + a_n \cdot 2^n$ i $b = b_0 + b_1 \cdot 2 + ... + b_n \cdot 2^n$, amb $a_i, b_i \in \mathbb{Z}_2, \forall i$. La forma més senzilla de sumar-los és mitjançant l'anomenat sumador *ripple-carry*. Es comença pel bit menys significant i es posa $c_0 = a_0 + b_0$ i $carry_0 = a_0 \cdot b_0$, on $+$ i $\cdot$ denoten la suma i el producte algebraics en $\mathbb{Z}_2$ i corresponen, respectivament, a les portes lògiques XOR i AND. A partir d'aquí es va iterant posant $c_i = a_i + b_i + carry_{i-1}$ i $carry_i = a_i \cdot b_i + a_i \cdot carry_{i-1} + b_i \cdot carry_{i-1}$. Al final $c_0, ..., c_n$ contindrà la suma i $carry_n$ indicarà si hi ha hagut desbordament. És el mateix mètode que la suma clàssica amb llapis i paper. La profunditat d'aquest circuit és de $O(n)$, ja que fem un nombre fix d'operacions per cada bit. Malauradament, no podem utilitzar aquest mètode. Donat que el nombre de nivells que pot avaluar el sistema BGV és lineal en el nombre de bits del mòdul, amb constant arbitràriament petita (dependrà de la distribució d'error i la dimensió de la instància), i per fer la suma homomòrfica d'enters menors que el mòdul necessitem un circuit de profunditat lineal en el nombre de bits, amb constant fixa, no podem garantir que es pugui avaluar el circuit sumador *ripple-carry*.
Per solucionar aquest problema utilitzarem un circuit sumador de profunditat logarítmica, també anomenat sumador d'arbre. En cada nivell es redueix el nombre de bits a la meitat. Per sumar dos nombres a, b de $n = 2^N$ bits es fa el següent.

1. Valors inicials: $G_{1,i} = a_i \cdot b_i$, $P_{1,i} = a_i + b_i$, per a $0 \leq i < n$.
2. S'itera endavant per calcular els grups.
   - $G_{2,i} = G_{1,2i+1} + G_{1,2i} P_{1,2i+1}$; $P_{2,i} = P_{1,2i} P_{1,2i+1}$, per a $0 \leq i < n/2$,
   - $G_{4,i} = G_{2,2i+1} + G_{2,2i} P_{2,2i+1}$; $P_{4,i} = P_{2,2i} P_{2,2i+1}$, per a $0 \leq i < n/4$,
   - ...,
   - $G_{n,i} = G_{n/2,2i+1} + G_{n/2,2i} P_{n/2,2i+1}$; $P_{n,i} = P_{n/2,2i} P_{n/2,2i+1}$, per $0, \leq i < n/2^N = 1$.
3. Es posa $C_{n,0} = G_{n,0}$ i s'itera enrere per calcular el *carry*.
   - $C_{n/2,2i+1} = G_{n/2,2i} + C_{n,i} P_{n/2,2i}$, $C_{n/2,2i} = C_{n,i}$, per $0 \leq i < n/2^N = 1$,
   - ...,
   - $C_{2,2i+1} = G_{2,2i} + C_{4,i} P_{2,2i}$, $C_{2,2i} = C_{4,i}$, per a $0 \leq i < n/2$,
   - $C_{1,2i+1} = G_{1,2i} + C_{2,i} P_{1,2i}$, $C_{1,2i} = C_{2,i}$, per a $0 \leq i < n$.
4. Es calcula la suma final: $(a + b)_i = a_i + b_i + C_{1,i}$, per a $0 \leq i < n$.

Per tant, podem sumar dos nombres de n bits amb profunditat $O(\log n)$. Aquest algorisme és essencial per poder aplicar la tècnica del *bootstrapping*. En la meva implementació, el sumador *ripple-carry* té una profunditat de n i el sumador logarítmic de $2(\log_2 n + 1)$. Fins a nombres de 8 bits és més eficient utilitzar el *ripple-carry* i per nombres de més de 8 bits el sumador logarítmic.
Si hem de sumar m nombres de n bits podem agafar-los per parelles i sumar-los utilitzant un sumador logarítmic, reduint així en cada iteració la quantitat de nombres a la meitat. La profunditat d'aquest circuit seria de $O(\log n \cdot \log m)$. Ho podem millorar fàcilment utilitzant el sumador *carry-save*, que pren tres nombres binaris i en produeix dos amb la mateixa suma. Com el seu nom indica, es basa en no propagar el *carry*. Donats a, b i c per sumar es calcula $d_i = a_i + b_i + c_i$ i $e_{i+1} = a_i \cdot b_i + a_i \cdot c_i + b_i \cdot c_i$.

Se satisfà que $a + b + c = d + e$. La profunditat del *carry-save adder* és per tant constant. Per sumar m nombres els anem reduint successivament utilitzant aquest mètode fins tenir-ne només 2, i aleshores utilitzarem un sumador logarítmic per calcular el resultat final. Aquest circuit té profunditat $O(\log n + \log m)$.

6.7 Distribucions

La distribució d'error pel problema LWE és senzilla de calcular. L'estudi de la dificultat que es fa al capítol 4 ens diu que el problema és segur agafant una discretització de la distribució normal embolicada. A la pràctica utilitzarem una binomial centrada al zero que sembla una aproximació suficientment bona. La distribució pel RLWE és més complicada, no només per l'aleatorització de la distribució necessària per fer segur el problema donat un nombre il·limitat de mostres, sinó perquè l'estudi de la seguretat es fa amb distribucions gaussianes el·líptiques en la immersió canònica i, en canvi, a l'hora d'implementar-ho es fa una immersió per coeficients. A la proposta d'estàndard [1] es donen alguns mètodes per calcular aquestes distribucions. En la meva implementació, per falta de temps, utilitzo una distribució binomial. És important remarcar que pel que se sap fins ara no satisfaria un bon nivell de seguretat.

6.8 Detalls tècnics

6.8.1 Gestió de la memòria

La gestió de la memòria no és trivial. Freqüentment ens trobem que estem treballant amb vectors de vectors de matrius de polinomis i costa trobar un esquema suficientment flexible que permeti experimentar i a la vegada suficientment rígid per no perdre eficiència innecessàriament. En la meva implementació el grau del polinomi quocient i la longitud en bits dels seus coeficients estan fixats en el moment de compilació. Com que un polinomi és de longitud fixe es pot construir una matriu de polinomis amb una sola reserva de memòria, tota en bloc. A l'hora de xifrar dades en múltiples missatges, malgrat allò més eficient seria fer una matriu, he utilitzat un vector de vectors, ja que és més flexible.

6.8.2 Processament multi-fil

Donat un problema inherentment ineficient no ens podem permetre el luxe de perdre eficiència utilitzant només un nucli del processador. El millor lloc per aplicar un processament en paral·lel, donada la seva naturalesa, és en la multiplicació de matrius. El més fàcil seria calcular cada coeficient de la matriu resultant en un fil separat, però no és òptim, ja que en el sistema RLWE la majoria de matrius resultants són petites, sovint amb menys elements que nuclis té l'ordinador. Utilitzant la llibreria de multi-processament *OpenMP* el que he fet ha estat, per a cada coeficient de la matriu resultant, dividir el càlcul entre diversos fils i al final sumar el resultat total.

6.9 Casos pràctics

Per tal de comprovar la viabilitat de la criptografia homomòrfica i posar a prova la meva implementació de l'esquema BGV analitzem algunes aplicacions pràctiques.

6.9.1 Cerca en textos xifrats

Suposem que tenim una sèrie de documents xifrats en un servidor i volem realitzar una cerca sense comprometre'n la privadesa. Aquí treballarem amb documents de text però el procediment és fàcilment extensible a qualsevol altre tipus de fitxer. Hem de xifrar la paraula que volem buscar, enviar-la al servidor i que aquest apliqui un circuit de cerca per a cada fitxer. Evidentment, el resultat del circuit de cerca estarà xifrat. El servidor ha de retornar una llista de tots els fitxers amb el resultat del circuit. Localment es desxifren els resultats, es veu a quins documents s'ha trobat el

text, i es fa una nova petició al servidor perquè retorni aquests fitxers.
En aquest cas el més adequat és utilitzar un polinomi ciclotòmic irreductible en $\mathbb{F}_2[X]$. La raó és que l'única operació aritmètica que volem realitzar sobre les dades és una suma bit a bit i per tant l'estructura multiplicativa de l'espai de textos plans és irrellevant. El que sí que ens interessa és que sigui un domini d'integritat, per tal d'assegurar que si el resultat d'un producte és 0 almenys un dels factors és 0. Siguin $(\psi_1, ..., \psi_n)$ la llista ordenada de textos xifrats que formen el document i $(\varphi_1, ..., \varphi_m)$, amb $m < n$, la llista ordenada de textos xifrats que volem buscar. A nivell lògic, el circuit que volem avaluar és

$$\text{Existeix} = \text{AND}_{i=1}^{n-m}\big[(\psi_i \text{ XOR } \varphi_1) \text{ OR } ... \text{ OR } (\psi_{i+m-1} \text{ XOR } \varphi_m)\big],$$

que retorna 0 si la cerca és positiva i 1 si és negativa. Les portes XOR i AND les tenim de forma nativa, ja que corresponen a la suma i al producte del cos amb el que treballem. Implementem la porta OR tenint en compte que A OR $B = A$ XOR B XOR (A AND B). Agafant els termes de cada bloc per parelles podem calcular tots els OR amb profunditat $O(\log m)$. Fent el mateix amb tots els termes AND calculem el producte amb profunditat $O(\log n)$. Per tant el circuit té una profunditat de $O(\log n + \log m)$.
A l'hora d'escollir un polinomi és important que tingui grau proper però menor a una potència de 2. D'aquesta manera aconseguim la relació seguretat/eficiència més bona possible, ja que per calcular la Tansformada Ràpida de Fourier s'ha d'ampliar el grau dels polinomis fins a la potència de 2 més propera. També és preferible que tingui el major nombre possible de coeficients iguals a zero, ja que així es disminueix significativament el temps de reduir mòdul $\Phi_m(x)$. Evitarem, doncs, els m primers. Un bon polinomi, en aquest cas, és el $\Phi_{625}(x) = x^{500} + x^{375} + x^{250} + x^{125} + 1$. Aquesta construcció ens limita a xifrar caràcter a caràcter, és a dir, blocs de 8 bits, si volem la màxima flexibilitat. Si xifréssim blocs de n caràcters només podríem buscar cadenes de text múltiples de n caràcters. Una altra limitació evident i intrínseca a la criptografia homomòrfica és que, si bé un algorisme de cerca tradicional pararia de buscar en el moment de trobar una coincidència, fer-ho segur implica que s'ha d'analitzar el document sencer, ja que en el moment de fer les comparacions no se sap si es produeix una coincidència o no.
Utilitzant una longitud de q inicial de 150 bits i una relació $q_i/q_{i-1} \approx 2000$ obtenim que es triguen uns 78 segons a generar les claus i uns 22 minuts a realitzar la cerca del text "BGV" al títol d'aquest treball, utilitzant 10 nivells de l'esquema.

6.9.2 Processament centralitzat de dades

Suposem que en una universitat els professors pengen les notes al campus virtual i aquest n'ha de calcular la mitjana, però no volem comprometre la seguretat de les dades exposant-les al servidor. Malgrat ser un cas bastant simple ens serveix com a prova de concepte i, a més a més, ens permet utilitzar les tècniques SIMD. Tenim alguns polinomis interessants. Per exemple, $\Phi_{255}(x)$ factoritza sobre $\mathbb{F}_2$ en 16 polinomis de grau 8 ($\varphi(255) = 128$); $\Phi_{257}(x)$ ho fa en 16 polinomis de grau 16 ($\varphi(257) = 256$); $\Phi_{771}(x)$ en 32 polinomis de grau 16 ($\varphi(771) = 512$). Per motius pràctics ens centrarem en el $F(x) = \Phi_{257}(x)$, però si volem un esquema realment segur hauríem d'utilitzar un polinomi de grau més alt. A l'annex es pot trobar la factorització de $\Phi_{257}(x)$ sobre $\mathbb{F}_2[X]$ i l'expansió en fraccions parcials necessària per poder calcular l'isomorfisme del teorema xinès del residu.
Per simplificar suposem que hi ha 16 alumnes. Com que treballarem en bits, és a dir, en textos plans en $\mathbb{F}_2 \subset \mathbb{F}_{2^{16}}$, només utilitzarem els termes constants dels factors F_i de $F(x)$, i per tant no ens fa falta un polinomi que faci de representant canònic. En el cas de treballar en un espai de textos plans $\mathbb{F}_{2^d}$, amb $d \mid 16$, necessitaríem un polinomi $K(x)$ de grau d que fes de representant canònic i calcular explícitament les immersions $\mathbb{F}_2[X]/(K) \to \mathbb{F}_2[X]/(F_i)$.
Prenem 16 blocs de 4 enters de 8 bits per sumar entre ells. En total tindrem, doncs, 32 missatges xifrats. Utilitzant una longitud de q inicial de 128 bits i una relació $q_i/q_{i-1} \approx 1000$ obtenim que es triguen uns 14 segons a generar les claus i uns 40 segons a aplicar el circuit binari de suma sobre els textos xifrats. Notem l'efectivitat de les operacions SIMD: es triga el mateix a sumar 4 enters que 16 blocs de 4 enters.

6.10 Resultats

He fet una petita implementació de l'esquema BGV utilitzant les diferents tècniques exposades al llarg d'aquest capítol i ha quedat ben clara la ineficiència intrínseca d'aquest tipus d'esquemes. Circuits senzills com sumar enters o buscar lletres, que un ordinador qualsevol pot fer en microsegons, avaluar-los homomòrficament és una tasca feixuga de diversos minuts. Segur que un equip professional amb més mitjans, temps i coneixements que jo pot fer una millor feina, però per l'altra banda el meu esquema tampoc verifica una seguretat estricte, que encara el faria més ineficient. Ara bé, no tot són males notícies. Fa tan sols una dècada hauria estat impossible fer el que fa el meu programa senzill. Un fet que és important tenir sempre present és que no tot algorisme es pot expressar com a circuit, i per tant no tot algorisme serà avaluable homomòrficament. L'exemple més senzill són les branques condicionals. Un circuit no pot tenir branques condicionals i la criptografia homomòrfica no permet treballar amb elles. Per exemple, seria impossible factoritzar un nombre enter homomòrficament, almenys per divisions successives. Si bé podem dividir dos nombres i tenir un xifrat del resultat, no podem saber si aquest és zero o no en el moment d'avaluar. Hi ha la possibilitat de calcular totes les branques exhaustivament i retornar-les en conjunt a l'usuari, sempre i quan això tingui sentit. En el cas de la factorització això equivaldria a donar totes les possibles factoritzacions, que evidentment no té cap sentit i és el mateix que no fer res.
Els esquemes homomòrfics només poden avaluar circuits anivellats. Una conseqüència d'aquest fet és que es perd eficiència anivellant els circuits. Per exemple, la clau de l'esquema BGV és lineal respecte la quantitat de nivells avaluables ja que conté tots els $\tau_{\mathbf{s}_i \to \mathbf{s}_{i+1}}$ per passar un text xifrat del nivell i al nivell $i+1$. Si es necessita passar un text xifrat del nivell i al nivell j, per anivellar el circuit, cal aplicar el procés $\mathsf{Refresh}$ $j-i$ vegades, que és bastant costós. En canvi, si calculem els $\tau_{\mathbf{s}_i \to \mathbf{s}_j}$ per a tot $i \neq j$ aleshores es pot fer en un sol pas. La clau, aleshores, creixeria quadràticament. Una altra alternativa és calcular els τ no successius a partir dels τ successius al començament del càlcul homomòrfic, si és suficientment llarg com per justificar-ho.

6.11 Treball futur

Hi ha moltes qüestions interessants que no hem tingut temps d'investigar o implementar, com per exemple completar el procés de *bootstrapping* per fer una instància completament homomòrfica de l'esquema BGV (tot i que no se sap si l'esquema BGV és KDM-segur) o aconseguir empaquetar i desempaquetar missatges xifrats (no només plans) homomòrficament. També hi ha qüestions més generals, com la d'estudiar maneres de representar dades en cossos finits $\mathbb{F}_{2^d}$ que no sigui la descomposició binària, que ens limita al subcòs $\mathbb{F}_2$, malgastant la resta de l'espai de textos plans; o si hi ha maneres més eficients de treballar en cossos de nombres que la representació per coeficients. Es podria intentar reformular l'esquema BGV per treballar amb la immersió canònica i així simplificar considerablement el producte d'elements del cos, ja que en la immersió canònica el producte es fa coordenada a coordenada. En general no és trivial trobar la manera més eficient de treballar computacionalment amb certes estructures algebraiques ja que, al cap i a la fi, l'únic que saben fer els ordinadors són operacions bit a bit i operacions aritmètiques en enters mòdul una potència de 2.

7 Conclusions

La criptografia homomòrfica és una àrea jove i activa de recerca que ha gaudit d'una enorme expansió durant les dues últimes dècades. En aquest treball n'hem estudiat breument els principis fonamentals. L'esquema de Gentry, publicat al 2009, demostra per primer cop que és possible construir un sistema criptogràfic que permeti operar amb dades xifrades sense revelar-ne el contingut. Els esquemes més eficients coneguts a dia d'avui es fonamenten en el problema RLWE, que redueix la seva seguretat en la seguretat quàntica d'alguns problemes de xarxes. D'entre els múltiples esquemes basats en el RLWE hem analitzat l'esquema BGV, del qual n'hem fet una petita implementació.

La principal dificultat del treball ha estat poder tenir una visió més o menys global de la criptografia homomòrfica. Donada la novetat de la matèria encara no hi ha llibres de text que en parlin, de manera que tota la informació prové d'articles o seminaris. Hi ha moltíssims articles interessants i que no han pogut ser analitzats per fer aquest treball, així com molts d'altres que senzillament no hem arribat a conèixer mai. És per tot això que és possible que aquest treball contingui detalls o explicacions que ja no són rellevants o fins i tot incorrectes perquè han estat millorats o esmenats en articles posteriors. És un risc que s'ha de tenir present en aquests casos.

Tots els esquemes criptogràfics homomòrfics coneguts es basen en afegir un cert error a una estructura matemàtica, de manera que treure aquest error és difícil sense el coneixement d'una informació extra que és la clau secreta. Per exemple, en l'esquema de Gentry s'afegeix un error en punts d'una xarxa i en el problema LWE s'afegeix un error a un sistema d'equacions lineals. Aquest error inherent als esquemes creix a mida que es realitzen operacions amb els textos xifrats, fins que arriba un punt que el resultat no es pot desxifrar correctament. L'única tècnica coneguda que permet fer un nombre indefinit d'operacions amb una clau pública acotada és la del *bootstrapping*, consistent a avaluar homomòrficament el circuit associat a l'algorisme de desxifrat, obtenint així un xifrat nou amb menys error.

Malgrat els avenços fets els últims anys la criptografia homomòrfica, en el seu estat actual, presenta clars desavantatges. L'expansió de la mida dels textos xifrats respecte dels textos plans és considerable, amb factors sovint superiors a 1000. Això vol dir que per xifrar un MB d'informació necessitaríem un text xifrat de vàries GB, amb les conseqüències computacionals i d'emmagatzematge que això implica, de manera que no podem utilitzar esquemes homomòrfics per xifrar grans quantitats de dades com faríem amb el AES o xifrats de flux. A més a més, la falta, encara, d'una estandardització completa i oficial d'algun sistema de xifrat homomòrfic i d'un període de *peer review* suficientment llarg i profund dels resultats emprats fan que difícilment es pugui recomanar l'ús dels esquemes homomòrfics actuals per a xifrar dades realment sensibles. Ara bé, donada la ràpida evolució de la matèria els últims anys, és possible que no estiguem gaire lluny de començar a veure els primers estàndards i aplicacions pràctiques d'esquemes de xifrat homomòrfic, si bé segurament no amb l'objectiu de delegar càlculs complexos, degut a la ineficiència intrínseca dels esquemes actualment coneguts, sí per a fer un processament centralitzat de dades sense comprometre'n la seguretat.

Potser algun dia es descobreix algun sistema de xifrat homomòrfic que no tingui el problema de l'error, o un que, malgrat tenir-lo, no depengui de la tècnica del *bootstrapping* sinó d'una més eficient per fer-lo completament homomòrfic. En cas contrari, no hi ha cap motiu pel qual no es puguin començar a utilitzar els esquemes actuals una vegada hi hagi millors estàndards i implementacions professionals, però la criptografia homomòrfica quedarà relegada a un segon pla, potser per ser utilitzada en ocasions puntuals que la seva naturalesa ho demani, però no a gran escala.

Referències

[1] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. Homomorphic encryption security standard. Technical report, HomomorphicEncryption.org, Toronto, Canada, November 2018.

[2] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. Cryptology ePrint Archive, Report 2015/046, 2015. https://eprint.iacr.org/2015/046.

[3] Dan Boneh and Victor Shoup. *A Graduate Course in Applied Cryptography.* http://toc.cryptobook.us/.

[4] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Fully homomorphic encryption without bootstrapping. Cryptology ePrint Archive, Report 2011/277, 2011. https://eprint.iacr.org/2011/277.

[5] Long Chen, Zhenfeng Zhang, and Xueqing Wang. Batched multi-hop multi-key fhe from ring-lwe with compact ciphertext extension. Cryptology ePrint Archive, Report 2017/923, 2017. https://eprint.iacr.org/2017/923.

[6] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. Cryptology ePrint Archive, Report 2016/421, 2016. https://eprint.iacr.org/2016/421.

[7] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. https://eprint.iacr.org/2012/144.

[8] Craig Gentry. *A fully homomorphic encryption scheme.* PhD thesis, Stanford University, 2009. crypto.stanford.edu/craig.

[9] Craig Gentry and Shai Halevi. Implementing gentry's fully-homomorphic encryption scheme. Cryptology ePrint Archive, Report 2010/520, 2010. https://eprint.iacr.org/2010/520.

[10] Craig Gentry, Shai Halevi, and Nigel P. Smart. Better bootstrapping in fully homomorphic encryption. Cryptology ePrint Archive, Report 2011/680, 2011. https://eprint.iacr.org/2011/680.

[11] Xiaodong Lin Jintai Ding, Xiang Xie. A simple provably secure key exchange scheme based on the learning with errors problem. Cryptology ePrint Archive, Report 2012/688, 2012. https://eprint.iacr.org/2012/688.

[12] Robert J. Jenkins Jr. Fast software encryption. pages 41–49, 1996.

[13] Vadim Lyubashevsky. Lattice signatures without trapdoors. Cryptology ePrint Archive, Report 2011/537, 2011. https://eprint.iacr.org/2011/537.

[14] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. Cryptology ePrint Archive, Report 2012/230, 2012. https://eprint.iacr.org/2012/230.

[15] Daniele Micciancio and Oded Regev. Worst-case to average-case reductions based on gaussian measures. *SIAM J. Comput.*, 37(1):267–302, April 2007.

[16] Peter L. Montgomery. Modular multiplication without trial division. *Math. Comp. 44*, pages 519–521, 1985.

[17] Emmanuela Orsini, Joop van de Pol, and Nigel P. Smart. Bootstrapping bgv ciphertexts with a wider choice of p and q. Cryptology ePrint Archive, Report 2014/408, 2014. https://eprint.iacr.org/2014/408.

[18] Michael O Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128 – 138, 1980.

[19] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '05, page 84–93, New York, NY, USA, 2005. Association for Computing Machinery.

[20] R. Rivest, L. Adleman, and M. Dertouzos. On data banks and privacy homomorphisms. Foundations of Secure Computation, pages 169–180, 1978. https://lucagiuzzi.unibs.it/corsi/Support/papers-cryptography/RAD78.pdf.

8 Annex

8.1 Estudi del $\Phi_{257}(x)$ en $\mathbb{F}_2[X]$

El polinomi $F(x) = \Phi_{257}(x)$ té grau $\varphi(257) = 256$ i descompon sobre $\mathbb{F}_2[X]$ en 16 factors irreductibles diferents de grau 16.

- $F_1 = x^{16} + x^{12} + x^{11} + x^8 + x^5 + x^4 + 1$
- $F_2 = x^{16} + x^{13} + x^8 + x^3 + 1$
- $F_3 = x^{16} + x^{13} + x^{12} + x^{10} + x^8 + x^6 + x^4 + x^3 + 1$
- $F_4 = x^{16} + x^{14} + x^{12} + x^{11} + x^8 + x^5 + x^4 + x^2 + 1$
- $F_5 = x^{16} + x^{14} + x^{13} + x^{11} + x^{10} + x^9 + x^8 + x^7 + x^6 + x^5 + x^3 + x^2 + 1$
- $F_6 = x^{16} + x^{14} + x^{13} + x^{12} + x^{10} + x^8 + x^6 + x^4 + x^3 + x^2 + 1$
- $F_7 = x^{16} + x^{14} + x^{13} + x^{12} + x^{11} + x^9 + x^8 + x^7 + x^5 + x^4 + x^3 + x^2 + 1$
- $F_8 = x^{16} + x^{15} + x^8 + x + 1$
- $F_9 = x^{16} + x^{15} + x^{13} + x^9 + x^8 + x^7 + x^3 + x + 1$
- $F_{10} = x^{16} + x^{15} + x^{13} + x^{11} + x^{10} + x^8 + x^6 + x^5 + x^3 + x + 1$
- $F_{11} = x^{16} + x^{15} + x^{13} + x^{12} + x^{10} + x^9 + x^8 + x^7 + x^6 + x^4 + x^3 + x + 1$
- $F_{12} = x^{16} + x^{15} + x^{14} + x^8 + x^2 + x + 1$
- $F_{13} = x^{16} + x^{15} + x^{14} + x^{12} + x^{10} + x^8 + x^6 + x^4 + x^2 + x + 1$
- $F_{14} = x^{16} + x^{15} + x^{14} + x^{13} + x^9 + x^8 + x^7 + x^3 + x^2 + x + 1$
- $F_{15} = x^{16} + x^{15} + x^{14} + x^{13} + x^{11} + x^{10} + x^8 + x^6 + x^5 + x^3 + x^2 + x + 1$
- $F_{16} = x^{16} + x^{15} + x^{14} + x^{13} + x^{12} + x^{11} + x^8 + x^5 + x^4 + x^3 + x^2 + x + 1$

Calculem l'expansió en fraccions parcials de

$$\frac{1}{F(x)} = \sum_{i=1}^{r} \frac{G_i(x)}{F_i(x)}.$$

- $G_1 = x^{12} + x^{11} + x^6 + x^5$
- $G_2 = x^{14} + x^{13} + x^4 + x^3$
- $G_3 = x^{14} + x^{13} + x^4 + x^3$
- $G_4 = x^{12} + x^{11} + x^6 + x^5$
- $G_5 = x^{14} + x^{13} + x^{12} + x^{11} + x^{10} + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3$
- $G_6 = x^{14} + x^{13} + x^4 + x^3$
- $G_7 = x^{14} + x^{13} + x^{12} + x^{11} + x^{10} + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3$
- $G_8 = x^8 + x^2 + 1$
- $G_9 = x^{14} + x^{10} + x^4 + x^2 + 1$
- $G_{10} = x^{14} + x^{12} + x^{10} + x^8 + x^4 + x^2 + 1$
- $G_{11} = x^{14} + x^{12} + x^6 + x^2 + 1$
- $G_{12} = x^{14} + x^8 + 1$
- $G_{13} = x^{14} + x^{12} + x^{10} + x^8 + x^6 + x^4 + 1$
- $G_{14} = x^{10} + x^4 + 1$
- $G_{15} = x^{12} + x^{10} + x^8 + x^4 + 1$
- $G_{16} = x^8 + x^6 + 1$

Per fer aquests càlculs s'ha utilitzat el programa PARI/GP.

8.2 Codi font

8.2.1 Setup.cpp

```
#ifndef SETUP_H
#define SETUP_H
#include <boost/multiprecision/cpp_int.hpp>
#include <map>

//typedef boost::multiprecision::int256_t Z;
//typedef boost::multiprecision::uint256_t uZ;
//#define sizeZ 32
//
//const int phi_m = 625;
//const int d = 500;
//const int n = 1;
//const int primrootpow2 = 1024;

typedef __int128_t Z;
typedef __uint128_t uZ;
#define sizeZ 16

const int phi_m = 257;
const int d = 256;
const int n = 1;
const int primrootpow2 = 512;

// Global variables

std::map<Z, Z> primroots;
#endif
```

8.2.2 Main.cpp

```
   #include <iostream>
#include <array>
#include <vector>
#include <random>
#include <chrono>
#include <string>
#include <type_traits>
#include <omp.h>
//#include <gmpxx.h>
#include <boost/multiprecision/cpp_int.hpp>
#include "Montgomery.cpp"
#include "Poly.cpp"
#include "Matrix.cpp"
#include "PrintInt128.cpp"
#include "isaac/isaac64.c"
#include "RandomDist.cpp"
#include "Setup.cpp"
#include "SIMD.cpp"

using namespace std;

typedef vector<Matrix<Poly<d, Z>>> Mvector;

struct FHEKey {
    vector<Z> q;
    Mvector s;
    Mvector A;
    Mvector tau;
};

Matrix<Poly<d, Z>> SecretKeyGen(int n);
Matrix<Poly<d, Z>> PublicKeyGen(Matrix<Poly<d, Z>>& s, Z q, int N);
Matrix<Poly<d, Z>> Enc(Matrix<Poly<d, Z>>& A, Poly<d, Z>& m);
Matrix<Poly<d, Z>> Enc(Poly<d, Z>& m, FHEKey key, int index);
Matrix<Poly<d, Z>> EncZero(FHEKey key, int index);
Poly<d, Z> Dec(Matrix<Poly<d, Z>> c, Matrix<Poly<d, Z>> s, Z q);
Poly<d, Z> Dec(Matrix<Poly<d, Z>> c, FHEKey& key, int index);
```

```
Matrix<Poly<d, Z>> BitDecomp(Matrix<Poly<d, Z>> x, Z q);
Matrix<Poly<d, Z>> PowersOf2(Matrix<Poly<d, Z>> x, Z q);
Matrix<Poly<d, Z>> SwitchKeyGen(Matrix<Poly<d, Z>> s1, Matrix<Poly<d, Z>> s2, Z q);
Matrix<Poly<d, Z>> SwitchKey(Matrix<Poly<d, Z>> c, Matrix<Poly<d, Z>> B, Z q);
Matrix<Poly<d, Z>> Scale(Matrix<Poly<d, Z>> x, Z q, Z p, int r);

double Error(Matrix<Poly<d, Z>> c, Matrix<Poly<d, Z>> s, Z q);

Mvector EncryptBuffer(Matrix<Poly<d, Z>>& A, uint8_t* data, size_t length, int
    blocksize);
Mvector EncryptInteger(Matrix<Poly<d, Z>>& A, Z n);
Mvector EncryptInteger(Z n, FHEKey& key, int index);
Mvector EncryptIntegerPack(vector<int> pack, FHEKey& key, int index);
Mvector EncryptString(string str, FHEKey& key, int index);
Z DecryptInteger(Mvector c, Matrix<Poly<d, Z>>& s, Z q);
Z DecryptInteger(Mvector c, FHEKey& key, int index);
vector<int> DecryptIntegerPack(Mvector c, FHEKey& key, int index);
Mvector SumIntegersZ2(Mvector& c1, Mvector& c2, FHEKey& key, int index);
int SumIntegersLogZ2(Mvector c1, Mvector c2, Mvector& out, FHEKey& key, int index);
void ThreeForTwoZ2(Mvector c1, Mvector c2, Mvector c3, Mvector& sumOut, Mvector&
    carryOut, FHEKey& key, int index);
int ThreeForTwoZ2(vector<Mvector> numbers, Mvector& sumOut, Mvector& carryOut, FHEKey
    & key, int index);
Matrix<Poly<d, Z>> Search(Mvector data, Mvector word, FHEKey& key, int& index);

void FHEKeyGen(FHEKey& key);
Matrix<Poly<d, Z>> Refresh(Matrix<Poly<d, Z>> c, FHEKey& key, int index);
Mvector Refresh(Mvector c, FHEKey& key, int index);

bool MillerRabin(Z p);
Z firstPrimeAbove(Z n);
Z firstPrimeAboveWithConditions(Z n);
Z gcdExtended(Z a, Z b, Z& x, Z& y);

int main()
{
    random_device r;
    for (int i = 0; i < RANDSIZ; i++) randrsl[i] = r();
    randinit(TRUE);
    InitBatching();

    auto t1 = std::chrono::high_resolution_clock::now();
    auto t2 = std::chrono::high_resolution_clock::now();

    uZ q = 1;
    q <<= 125;

    t1 = std::chrono::high_resolution_clock::now();
    FHEKey key;
    for (int i = 0; i < 12; i++) {
        key.q.push_back(firstPrimeAboveWithConditions(q));
        cout << key.q[i] << endl;
        q /= 800;
    }

    //Poly<d, Z>::push_q(key.q[2]);
    //InitBatching();
    //vector<Poly<d, Z>> moduli = {
    //    {1,1,0,1},
    //    {1},
    //    {0,1}
    //};
    //cout << Batch(moduli) << endl;
    //Poly<d, Z> F1 = { 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1 };
    //cout << Unbatch(Batch(moduli))[0] << endl;

    FHEKeyGen(key);
    t2 = std::chrono::high_resolution_clock::now();
    cout << "Clau generada en " << (double)chrono::duration_cast<std::chrono::
        milliseconds>(t2 - t1).count() / 1000 << " s" << endl;

    Poly<d, Z> m1 = { 1 , 0, 1, 1, 0, 0, 1 };
```

```
    Poly<d, Z> m2 = { 0 , 1, 0, 0, 0, 0, 1 };
    Poly<d, Z> m3 = { 1 , 1 };
    cout << (m1 * m2) % q << endl;
    Poly<d, Z>::push_q(key.q[0]);

    //cout << (A[0] * s[0]) % key.q[0] << endl;
    auto c1 = Enc(key.A[0], m3);
    cout << Dec(c1, key.s[0], key.q[0]) << endl;
    cout << Error(c1, key.s[0], key.q[0]) << endl;

    for (int i = 0; i < 10; i++) {
        auto c2 = c1.outer(c1);
        c1 = Refresh(c2, key, i);
        Poly<d, Z>::push_q(key.q[i + 1]);
        cout << Dec(c1, key.s[i + 1], key.q[i + 1]) << endl;
        cout << Error(c1, key.s[i + 1], key.q[i + 1]) << endl << endl;
    }

    Mvector str1 = EncryptString("Una introducci  a la criptografia homom rfica.
        Implementaci  de l'esquema BGV.", key, 0);
    Mvector word = EncryptString("BGV", key, 0);
    int index = 0;
    t1 = std::chrono::high_resolution_clock::now();
    auto exists = Search(str1, word, key, index);
    t2 = std::chrono::high_resolution_clock::now();
    cout << "Cerca realitzada en " << (double)chrono::duration_cast<std::chrono::
        milliseconds>(t2 - t1).count() / 1000 << " s" << endl;
    cout << Dec(exists, key, index);
    cout << index << endl;
    //return 0;

    vector<int> notes1 = { 2, 7, 8, 9, 2, 1, 5, 8, 2, 2, 1, 0, 0, 5, 10, 10 };
    vector<int> notes2 = { 5, 9, 5, 8, 2, 7, 2, 7, 8, 7, 5, 9, 0, 1, 3, 3 };
    vector<int> notes3 = { 4, 7, 6, 6, 7, 7, 7, 7, 8, 8, 8, 10, 0, 1, 1, 1 };
    vector<int> notes4 = { 4, 2, 5, 9, 2 , 5, 6, 7, 8, 9, 10, 10, 2, 3, 1, 0};

    vector<Mvector> IntegersPack = { EncryptIntegerPack(notes1, key, 0),
        EncryptIntegerPack(notes2, key, 0), EncryptIntegerPack(notes3, key, 0),
        EncryptIntegerPack(notes4, key, 0) };
    cout << DecryptIntegerPack(IntegersPack[0], key, 0)[0] << endl;
    Mvector sum(8), carry(9);
    int iter = ThreeForTwoZ2(IntegersPack, sum, carry, key, 0);
    iter = SumIntegersLogZ2(sum, carry, sum, key, 0 + iter);
    auto integers = DecryptIntegerPack(sum, key, iter);
    for (int i = 0; i < integers.size(); i++) {
        cout << integers[i] << " ";
    }
    cout << endl;
    cout << iter << endl;

    //vector<Mvector>Integers = { EncryptInteger(100, key, 0)
    //    , EncryptInteger(13, key, 0), EncryptInteger(19, key, 0), EncryptInteger
        (10, key, 0), EncryptInteger(1, key, 0) };
    //Mvector sum(8), carry(9);
    //int iter = ThreeForTwoZ2(Integers, sum, carry, key, 0);
    //iter = SumIntegersLogZ2(sum, carry, sum, key, 0 + iter);
    //cout << DecryptInteger(sum, key, iter) << endl;
    //cout << iter << endl;

    //auto t1 = std::chrono::high_resolution_clock::now();

    //auto t2 = std::chrono::high_resolution_clock::now();
    //cout << chrono::duration_cast<std::chrono::milliseconds>(t2 - t1).count() <<
        endl;

    cout << Montgomery::counter << endl;
    return 0;
}

Mvector EncryptBuffer(Matrix<Poly<d, Z>>& A, uint8_t* data, size_t length, int
    blocksize) {
    Mvector res;
```

```
    Poly<d, Z> m;
    for (int i = 0, j = 0; i < length*8; i++, j++) {
        m[j] = (data[i / 8] >> (i % 8)) & 1;
        if (j == blocksize - 1 || i == length*8 - 1) {
            res.push_back(Enc(A, m));
            j = -1;
        }
    }
    return res;
}

Mvector EncryptInteger(Matrix<Poly<d, Z>>& A, Z n) {
    return EncryptBuffer(A, (uint8_t*)&n, 1, 1);
}

Mvector EncryptInteger(Z n, FHEKey& key, int index) {
    Poly<d, Z>::push_q(key.q[index]);
    auto c = EncryptBuffer(key.A[index], (uint8_t*)&n, 1, 1);
    Poly<d, Z>::pop_q();
    return c;
}

Mvector EncryptIntegerPack(vector<int> pack, FHEKey& key, int index) {
    Mvector res;
    Poly<d, Z>::push_q(key.q[index]);
    for (int i = 0; i < 8; i++) {
        vector<Poly<d, Z>> bits;
        for (int j = 0; j < pack.size(); j++) {
            bits.push_back({ (pack[j] >> i) & 1 });
        }
        Poly<d, Z> m = Batch(bits);
        res.push_back(Enc(m, key, index));
    }
    Poly<d, Z>::pop_q();
    return res;
}

Mvector EncryptString(string str, FHEKey& key, int index) {
    Poly<d, Z>::push_q(key.q[index]);
    auto c = EncryptBuffer(key.A[index], (uint8_t*)str.data(), str.length(), 8);
    Poly<d, Z>::pop_q();
    return c;
}

Z DecryptInteger(Mvector c, Matrix<Poly<d, Z>>& s, Z q) {
    Z n = 0;
    for (int i = c.size() - 1; i >= 0; i--) {
        n <<= 1;
        n += Dec(c[i], s, q)[0];
    }
    return n;
}

Z DecryptInteger(Mvector c, FHEKey& key, int index) {
    return DecryptInteger(c, key.s[index], key.q[index]);
}

vector<int> DecryptIntegerPack(Mvector c, FHEKey& key, int index) {
    vector<int> integers(16);
    for (int i = c.size() - 1; i >= 0; i--) {
        //cout << c[i] << endl;
        Poly<d, Z> m = Dec(c[i], key, index);
        auto bits = Unbatch(m);
        //cout << bits[0] << endl;
        for (int j = 0; j < bits.size(); j++) {
            integers[j] <<= 1;
            integers[j] += (int)bits[j][0];
        }
    }
    return integers;
}
```

```
Mvector SumIntegersZ2(Mvector& c1, Mvector& c2, FHEKey& key, int index) {
    int n = c1.size();
    cout << c1.size() << " " << c2.size() << endl;
    Mvector res(n);
    Poly<d, Z>::push_q(key.q[index]);
    res[0] = Refresh(c1[0] + c2[0], key, index);
    auto carry = Refresh(c1[0].outer(c2[0]), key, index);
    //cout << Dec(c1[0], key.s[index], key.q[index]) << endl;
    //cout << Dec(c2[0], key.s[index], key.q[index]) << endl;
    //cout << Dec(res[0], key.s[index + 1], key.q[index + 1]) << endl;
    //cout << Dec(carry, key.s[index + 1], key.q[index + 1]) << endl << endl;
    for (int i = 1; i < n; i++) {
        auto c1_q = c1[i], c2_q = c2[i];
        for (int j = 0; j < i; j++) {
            c1_q = Refresh(c1_q, key, index + j);
            c2_q = Refresh(c2_q, key, index + j);
        }
        Poly<d, Z>::push_q(key.q[index + i]);
        res[i] = Refresh(c1_q + c2_q + carry, key, index + i);
        carry = Refresh(c1_q.outer(c2_q) + c1_q.outer(carry) + c2_q.outer(carry), key
            , index + i);

        //cout << Dec(c1_q, key.s[index + i], key.q[index + i]) << endl;
        //cout << Dec(c2_q, key.s[index + i], key.q[index + i]) << endl;
        //cout << Dec(res[i], key.s[index + i +1], key.q[index + i +1]) << endl;
        //cout << Dec(carry, key.s[index + i +1], key.q[index + i +1]) << endl <<
            endl;
    }
    for (int i = 0; i < n - 1; i++) {
        for(int j = 1; j < n - i; j++) res[i] = Refresh(res[i], key, index + i + j);
    }
    return res;
}

int SumIntegersLogZ2(Mvector c1, Mvector c2, Mvector& out, FHEKey& key, int index) {
    int n = c1.size();
    int N = log2(n);
    cout << n << " " << N << endl;
    int indexInit = index;
    vector<Mvector> G, P, C;
    Mvector Gi(n), Pi(n), Ci(n), a(n), b(n);
    Poly<d, Z>::push_q(key.q[index]);
    for (int i = 0; i < n; i++) {
        Gi[i] = Refresh(c1[i].outer(c2[i]), key, index);
        Pi[i] = Refresh(c1[i] + c2[i], key, index);
    }
    a = Refresh(c1, key, index);
    b = Refresh(c2, key, index);
    index++;
    Poly<d, Z>::pop_q();
    G.push_back(Gi);
    P.push_back(Pi);
    for (int k = 1; k <= N; k++) {
        Poly<d, Z>::push_q(key.q[index]);
        Gi.clear();
        Pi.clear();
        for (int i = 0; i < (n >> k); i++) {
            //cout << "k=" << k << " i=" << i << endl;
            Gi.push_back(Refresh(G[k - 1][2 * i].outer(P[k - 1][2 * i + 1]).
                addToFirstRow(G[k - 1][2 * i + 1]), key, index));
            Pi.push_back(Refresh(P[k - 1][2 * i].outer(P[k - 1][2 * i + 1]), key,
                index));
            //cout << Dec(Pi[i], key.s[index + 1], key.q[index + 1]) << endl;
        }
        G.push_back(Gi);
        P.push_back(Pi);
        a = Refresh(a, key, index);
        b = Refresh(b, key, index);
        Poly<d, Z>::pop_q();
        index++;
    }
    C.push_back(G.back());
```

```
    for (int k = 0; k < N; k++) {
        Poly<d, Z>::push_q(key.q[index]);
        Pi = P[N - k - 1];
        Gi = G[N - k - 1];
        for (int j = indexInit + N - k; j < index; j++) {
            Pi = Refresh(Pi, key, j);
            Gi = Refresh(Gi, key, j);
        }
        for (int i = 0; i < (1 << k); i++) {
            //cout << "k=" << k << " i=" << i << endl;
            Ci[2 * i + 1] = Refresh(C[k][i].outer(Pi[2 * i]).addToFirstRow(Gi[2 * i])
                , key, index);
            cout << Dec(Ci[2 * i + 1], key.s[index + 1], key.q[index + 1]) << endl;
            Ci[2 * i] = Refresh(C[k][i], key, index);
            //cout << Dec(Ci[2*i+1], key.s[index + 1], key.q[index + 1]) << endl;
        }
        C.push_back(Ci);
        a = Refresh(a, key, index);
        b = Refresh(b, key, index);
        Poly<d, Z>::pop_q();
        index++;
    }
    Poly<d, Z>::push_q(key.q[index]);
    Mvector res(n);
    for (int i = 0; i < n; i++) {
        res[i] = Refresh(a[i] + b[i] + C[N][i], key, index);
    }
    Poly<d, Z>::pop_q();
    index++;
    out = res;
    return index;
}

void ThreeForTwoZ2(Mvector c1, Mvector c2, Mvector c3, Mvector& sumOut, Mvector&
    carryOut, FHEKey& key, int index) {
    int n = c1.size();
    carryOut[0] = EncZero(key, index + 1);
    Poly<d, Z>::push_q(key.q[index]);
    for (int i = 0; i < n; i++) {
        sumOut[i] = Refresh(c1[i] + c2[i] + c3[i], key, index);
        carryOut[i + 1] = Refresh(c1[i].outer(c2[i]) + c1[i].outer(c3[i]) + c2[i].
            outer(c3[i]), key, index);
    }
    Poly<d, Z>::pop_q();
}

int ThreeForTwoZ2(vector<Mvector> numbers, Mvector& sumOut, Mvector& carryOut, FHEKey
    & key, int index) {
    int n = numbers[0].size();
    int iter = 0;
    vector<Mvector> numbersIn = numbers, numbersOut;
    Mvector innerSumOut(n), innerCarryOut(n + 1);
    while (numbersIn.size() != 2) {
        int blocks = numbersIn.size() / 3;
        for (int i = 0; i < blocks*3; i += 3) {
            ThreeForTwoZ2(numbersIn[i], numbersIn[i + 1], numbersIn[i + 2],
                innerSumOut, innerCarryOut, key, index + iter);
            numbersOut.push_back(innerSumOut);
            numbersOut.push_back(innerCarryOut);
        }
        for (int i = blocks * 3; i < numbersIn.size(); i++) {
            numbersOut.push_back(Refresh(numbersIn[i], key, index + iter));
        }
        numbersIn = numbersOut;
        numbersOut.clear();
        iter++;
    }
    sumOut = numbersIn[0];
    carryOut = numbersIn[1];
    return index + iter;
}
```

```
int MultZ2(Mvector c1, Mvector c2, FHEKey& key, int index) {
    vector<Mvector> numbers;
    int length = c1.size() * c2.size();
    for (int i = 0; i < c2.size(); i++) {
        Mvector number(length);
        for (int j = 0; j < length; j++) {
            if (j < i) {
                //number[i] =
            }
        }
    }
}

Matrix<Poly<d, Z>> Search(Mvector data, Mvector word, FHEKey& key, int& index) {
    Mvector blocksIn, blocksOut;
    int n = data.size();
    int m = word.size();
    int iter = index;
    for (int i = 0; i < n - m; i++) {
        cout << "Block " << i << endl;
        Mvector innerBlocksIn, innerBlocksOut;
        Poly<d, Z>::push_q(key.q[index]);
        for (int j = 0; j < m; j++) {
            innerBlocksIn.push_back(Refresh(data[i + j] + word[j], key, index));
        }
        iter = index + 1;
        while (innerBlocksIn.size() != 1) {
            int blocks = innerBlocksIn.size() / 2;
            Poly<d, Z>::push_q(key.q[iter]);
            for (int i = 0; i < blocks * 2; i += 2) {
                innerBlocksOut.push_back(Refresh(innerBlocksIn[i].outer(innerBlocksIn
                    [i+1]).addToFirstRow(innerBlocksIn[i] + innerBlocksIn[i+1]), key,
                    iter));
            }
            for (int i = blocks * 2; i < innerBlocksIn.size(); i++) {
                innerBlocksOut.push_back(Refresh(innerBlocksIn[i], key, iter));
            }
            innerBlocksIn = innerBlocksOut;
            innerBlocksOut.clear();
            iter++;
        }
        blocksIn.push_back(innerBlocksIn[0]);
    }
    while (blocksIn.size() != 1) {
        int blocks = blocksIn.size() / 2;
        Poly<d, Z>::push_q(key.q[iter]);
        for (int i = 0; i < blocks * 2; i += 2) {
            blocksOut.push_back(Refresh(blocksIn[i].outer(blocksIn[i + 1]), key, iter
                ));
        }
        for (int i = blocks * 2; i < blocksIn.size(); i++) {
            blocksOut.push_back(Refresh(blocksIn[i], key, iter));
        }
        blocksIn = blocksOut;
        blocksOut.clear();
        iter++;
    }
    index = iter;
    return blocksIn[0];
}

void FHEKeyGen(FHEKey& key) {
    for (int i = 0; i < key.q.size(); i++) {
        Poly<d, Z>::push_q(key.q[i]);
        int N = ceil((2 * n + 1) * log2((double)key.q[i]));
        key.s.push_back(SecretKeyGen(n));
        key.A.push_back(PublicKeyGen(key.s[i], key.q[i], N));
        if (i > 0) {
            Poly<d, Z>::push_q(key.q[i - 1]);
            auto sp = key.s[i - 1].outer(key.s[i - 1]) % key.q[i - 1];
            auto spp = BitDecomp(sp, key.q[i - 1]);
```

```
            //Poly<d, Z>::push_q(qs[i]);
            key.tau.push_back(SwitchKeyGen(sp, key.s[i], key.q[i - 1]));
        }
    }
}

Matrix<Poly<d, Z>> Refresh(Matrix<Poly<d, Z>> c, FHEKey& key, int index) {
    Poly<d, Z>::push_q(key.q[index]);
    auto c2 = SwitchKey(c, key.tau[index], key.q[index]);
    auto c3 = Scale(c2, key.q[index], key.q[index + 1], 2);
    //Poly<d, Z>::push_q(key.q[index + 1]);
    Poly<d, Z>::pop_q();
    return c3;
}

Mvector Refresh(Mvector c, FHEKey& key, int index) {
    Mvector res;
    for (int i = 0; i < c.size(); i++) {
        res.push_back(Refresh(c[i], key, index));
    }
    return res;
}

Matrix<Poly<d, Z>> SecretKeyGen(int n) {
    Matrix<Poly<d, Z>> s(n + 1, 1);
    s[0] = Poly<d, Z>::One();
    for (int i = 0; i < n; i++) {
        s[i + 1] = Poly<d, Z>::Chi();
    }
    return s;
}

Matrix<Poly<d, Z>> PublicKeyGen(Matrix<Poly<d, Z>>& s, Z q, int N) {
    int n = s.size() - 1;
    Matrix<Poly<d, Z>> Ap = Matrix<Poly<d, Z>>::Uniform(q, N, n);
    Matrix<Poly<d, Z>> e = Matrix<Poly<d, Z>>::Chi(N, 1);
    Matrix<Poly<d, Z>> b = Ap * s.range(1, 0, -1, -1) + 2*e;
    Matrix<Poly<d, Z>> A = b | -Ap;
    return A;
}

Matrix<Poly<d, Z>> Enc(Matrix<Poly<d, Z>>& A, Poly<d, Z>& m) {
    int N = A.rows();
    int n = A.cols();
    Matrix<Poly<d, Z>> mm(n, 1);
    mm[0] = m;
    Matrix<Poly<d, Z>> c = (mm + A.randomRowSum().transpose());
    return c;
}

Matrix<Poly<d, Z>> Enc(Poly<d, Z>& m, FHEKey key, int index) {
    Poly<d, Z>::push_q(key.q[index]);
    auto c = Enc(key.A[index], m);
    Poly<d, Z>::pop_q();
    return c;
}

Matrix<Poly<d, Z>> EncZero(FHEKey key, int index) {
    Poly<d, Z>::push_q(key.q[index]);
    Poly<d, Z> m;
    auto c = Enc(key.A[index], m);
    Poly<d, Z>::pop_q();
    return c;
}

Matrix<Poly<d, Z>> GetZero(FHEKey& key, int index) {
    int n = key.A[index].cols();
    Matrix<Poly<d, Z>> mm(n, 1);
    return mm;
}

inline Poly<d, Z> Dec(Matrix<Poly<d, Z>> c, Matrix<Poly<d, Z>> s, Z q) {
```

```
    Poly<d, Z>::push_q(q);
    auto res = c.dot(s) % q % 2;
    Poly<d, Z>::pop_q();
    return res;
}

Poly<d, Z> Dec(Matrix<Poly<d, Z>> c, FHEKey& key, int index) {
    Poly<d, Z>::push_q(key.q[index]);
    auto res = c.dot(key.s[index]) % key.q[index] % 2;
    Poly<d, Z>::pop_q();
    return res;
}

Matrix<Poly<d, Z>> BitDecomp(Matrix<Poly<d, Z>> x, Z q) {
    int n = x.size();
    int places = ceil(log2((double)q));
    //cout << "places= " << places << endl;
    Matrix<Poly<d, Z>> res(n*places, 1);
    for (int i = 0; i < n; i++) {
        Poly<d, Z> temp = x[i].modPos(q);
        //cout << "x= " << temp << endl;
        for (int j = 0; j < places; j++) {
            res[j * n + i] = temp % 2;
            temp /= 2;
        }
    }
    return res;
}

Matrix<Poly<d, Z>> PowersOf2(Matrix<Poly<d, Z>> x, Z q) {
    int n = x.size();
    int places = ceil(log2((double)q));
    Matrix<Poly<d, Z>> res(n * places, 1);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < places; j++) {
            Z power = 1;
            power = power << j;
            res[j*n + i] = power * x[i];
        }
    }
    return res;
}

Matrix<Poly<d, Z>> SwitchKeyGen(Matrix<Poly<d, Z>> s1, Matrix<Poly<d, Z>> s2, Z q) {
    int N = s1.size() * ceil(log2((double)q));
    Matrix<Poly<d, Z>> A = PublicKeyGen(s2, q, N);
    Matrix<Poly<d, Z>> P = PowersOf2(s1, q);
    for (int i = 0; i < N; i++) {
        A(i, 0) += P[i];
    }
    return A.transpose();
}

Matrix<Poly<d, Z>> SwitchKey(Matrix<Poly<d, Z>> c, Matrix<Poly<d, Z>> B, Z q) {
    int size = (n + 1) * (n + 1);
    Matrix<Poly<d, Z>> c2(n + 1, n + 1);
    if (c.size() < (n + 1) * (n + 1)) {
        for (int i = 0; i < c.size(); i++) c2[i] = c[i];
    }
    else {
        c2 = c;
    }
    return B*BitDecomp(c2, q);
}

Matrix<Poly<d, Z>> Scale(Matrix<Poly<d, Z>> x, Z q, Z p, int r) {
    int n = x.rows();
    int m = x.cols();
    Matrix<Poly<d, Z>> res(n, m);
    Poly<d, Z>::push_q(q);
    for (int i = 0; i < n * m; i++) {
        res[i] = q - p * x[i];
```

```
    }
    Poly<d, Z>::pop_q();
    Poly<d, Z>::push_q(p);
    Z qinv = Montgomery::inv(q % p);
    for (int i = 0; i < n * m; i++) {
        res[i] = qinv * res[i];
        for (int j = 0; j < d; j++) {
            res[i][j] += x[i][j] % r - res[i][j] % r;
        }
    }
    Poly<d, Z>::pop_q();
    return res;
}

double Error(Matrix<Poly<d, Z>> c, Matrix<Poly<d, Z>> s, Z q) {
    //Poly<d, Z> m = (c.dot(s) % q) % 2;
    //cout << (c.dot(s)) % q << endl;
    return (double)((c.dot(s)) % q).maxAbsCoeff() / (double)q;
}

bool isPrimeLog(Z p) {
    if (p % 2 == 0) return false;
    for (int i = 3; i < min(sqrt((double)p), 100); i += 2) {
        if (p % i == 0) return false;
    }
    return true;
}

bool MillerRabin(Z p) {
    UniformDist dist(p);
    Montgomery::set_mod(p);
    Z d = p - 1;
    int r = 0;
    while (d % 2 == 0) {
        d >>= 1;
        r += 1;
    }
    bool cont = false;
    for (int i = 0; i < 64; i++) {
        Z a = dist.get();
        if (a == p - 1) a--;
        else if (a == 0 || a == 1) a = 2;
        Z x = Montgomery::modPow(a, d);
        if (x == 1 || x == p - 1) continue;
        cont = false;
        for (int j = 0; j < r - 1; j++) {
            x = Montgomery::modPow(x, 2);
            if (x == p - 1) {
                cont = true;
                break;
            }
        }
        if (cont) continue;
        return false;
    }
    return true;
}

bool isPrime(Z p) {
    if (!isPrimeLog(p)) return false;
    else return MillerRabin(p);
}

Z firstPrimeAbove(Z n) {
    if (n <= 1) return 2;
    if (n % 2 == 0) n++;
    while (!isPrimeLog(n) || !MillerRabin(n)) {
        n += 2;
    }
    return n;
}
```

```
//Calcula el primer primer m s gran que n de la forma a*lcm(m, 2^{d+1}) + 1, amb a
    primer.
Z firstPrimeAboveWithConditions(Z n) {
    if (n <= 1) return 2;
    Z pow2 = 1, lcm = phi_m;
    while (pow2 < 2 * d) {
        pow2 <<= 1;
        if (lcm % 2 == 0) lcm >>= 1;
    }
    lcm = lcm * pow2;
    Z a = n / lcm + 1;
    if (a % 2 == 0) a++;
    Z p = a * lcm + 1;
    while (1) {
        p = a * lcm + 1;
        if (!isPrime(a) || !isPrime(p)) {
            a += 2;
        }
        else {
            // Trobem una arrel primitiva de la unitat m dul p
            vector<Z> mfactors;
            for (int i = 3; i < sqrt(phi_m); i += 2) {
                if (isPrime(i) && (phi_m % i == 0)) mfactors.push_back(i);
            }
            Montgomery::set_mod(p);
            for (int i = 3; i < p; i++) {
                if (Montgomery::modPow(i, (p-1) / a) != 1 && Montgomery::modPow(i, (p
                    -1) / 2) != 1) {
                    bool isPrimRoot = true;
                    for (int j = 0; j < mfactors.size(); j++) {
                        if (Montgomery::modPow(i, (p - 1) / mfactors[j]) == 1) {
                            isPrimRoot = false;
                            break;
                        }
                    }
                    if (isPrimRoot) {
                        primroots[p] = Montgomery::modPow(i, (p-1) / pow2);
                        break;
                    }
                }
            }
            break;
        }
    }
    return p;
}

Z gcdExtended(Z a, Z b, Z& x, Z& y){
    if (a == 0){
        x = 0;
        y = 1;
        return b;
    }

    Z x1, y1;
    Z gcd = gcdExtended(b % a, a, x1, y1);
    x = y1 - (b / a) * x1;
    y = x1;
    return gcd;
}
```

8.2.3 Montgomery.cpp

```
#ifndef MONTGOMERY_H
#define MONTGOMERY_H

#include <iostream>
#include <type_traits>
#include <boost/multiprecision/cpp_int.hpp>
#include "Setup.cpp"
#include "PrintInt128.cpp"
```

```
#ifdef _MSC_VER
#include <intrin.h>
#endif
using namespace std;
typedef Z T;
typedef uZ uT;

class Montgomery {
protected:
  static uT _n;
  static int _d;
  static uT _np;
  static uT _r2;
public:
  inline static T mul_high_half(uint64_t x, uint64_t y) {
    uint32_t a = x >> 32, b = x;
    uint32_t c = y >> 32, d = y;
    uint64_t ac = (uint64_t)a * c;
    uint64_t ad = (uint64_t)a * d;
    uint64_t bc = (uint64_t)b * c;
    uint64_t bd = (uint64_t)b * d;
    uint64_t carry = (uint64_t)(uint32_t)ad + (uint64_t)(uint32_t)bc + (bd >> 32u);
    uint64_t high = ac + (ad >> 32u) + (bc >> 32u) + (carry >> 32u);
    return high;
  }
#ifdef __GNUC__
  typedef unsigned __int128 uint128;
  typedef __int128 int128;
  inline static void mult_full(uint64_t x, uint64_t y, uint64_t& high, uint64_t& low)
      {
    uint128 prod = (int128)x * y;
    high = (uint64_t)(prod >> 64);
    low = (uint64_t)prod;
  }
  inline static void mult_full(int128 x, int128 y, int128& high, int128& low) {
    counter++;
    uint64_t a = x >> 64, b = x;
    uint64_t c = y >> 64, d = y;
    uint128 ac = (uint128)a * c;
    uint128 ad = (uint128)a * d;
    uint128 bc = (uint128)b * c;
    uint128 bd = (uint128)b * d;
    uint128 carry = (uint128)(uint64_t)ad + (uint128)(uint64_t)bc + (bd >> 64u);
    high = ac + (ad >> 64u) + (bc >> 64u) + (carry >> 64u);
    low = (ad << 64u) + (bc << 64u) + bd;
  }
#else
  inline static void mult_full(uint64_t x, uint64_t y, uint64_t& high, uint64_t& low)
      {
    high = __umulh(x, y);
    low = x * y;
  }
#endif
  typedef boost::multiprecision::int128_t b128;
  typedef boost::multiprecision::int256_t b256;
  typedef boost::multiprecision::int512_t b512;
  inline static void mult_full(b128 x, b128 y, b128& high, b128& low) {
    b256 res = boost::multiprecision::multiply(res, x, y);
    low = static_cast<b128>(res);
    res >>= 128;
    high = static_cast<b128>(res);
  }
  inline static void mult_full(b256 x, b256 y, b256& high, b256& low) {
    b512 res = boost::multiprecision::multiply(res, x, y);
    low = static_cast<b256>(res);
    res >>= 256;
    high = static_cast<b256>(res);
  }


  static void set_mod(T n) {
    if (n == _n) return;
```

```
  _n = static_cast<uT>(n);
  _d = sizeof(T) * 8;
  if (n == 0) {
    _np = 0;
    _r2 = 0;
    return;
  }
  uT inv = 1;
  for (int i = 0; i < 7; i++)
    inv *= 2 - _n * inv;
  _np = inv;
  //std::cout << inv << std::endl;

  _r2 = (1 - _n) % _n - 1;
  //cout << _r2 << endl;
  for (int i = 0; i < 4; i++) {
    _r2 <<= 1;
    if (_r2 >= _n)
      _r2 -= _n;

  }
  for (int i = 0; i < log2(sizeZ) + 1; i++)
    _r2 = static_cast<uT>(mul(_r2, _r2));
  //std::cout << _r2 << std::endl;
}
inline static T augment(T x) {
  //x %= _n;
  //cout << x << endl;
  if (x < 0) x += _n;
  return mul((T)x, _r2);
}
inline static T reduce(T x) {
  if (x < 0) x += _n;
  T high, low;
  T m = x * _np;
  mult_full(m, _n, high, low);
  T t = high;
  if (t == 0) return 0;
  else return _n - t;
}
inline static T reduce(T high, T low) {
  T high2, low2;
  T m = low * _np;
  mult_full(m, _n, high2, low2);
  T t = high - high2;

  if (t < 0) return t + _n;
  else return t;
}
inline static T mul(T x, T y) {
  T high, low, high2, low2;

  mult_full(x, y, high, low);

  T m = low * _np;
  mult_full(m, _n, high2, low2);
  T t = high - high2;

  if (t < 0) return t + _n;
  else return t;
}
inline static T add(T x, T y) {
  T s = x + y;
  if (s >= _n) s -= _n;
  return s;
}
inline static T add(T x, T y, T z) {
  T s = x + y + x;
  if (s >= _n) s -= _n;
  return s;
}
inline static T sub(T x, T y) {
```

```
    if (y > x) return _n + x - y;
    else return x - y;
  }
  static T modPow(T b, T e) {
    T res = augment((T)1);
    T bM = augment(b);
    while (e > 0) {
      if (e % 2 == 1) {
        res = mul(res, bM);
      }
      e >>= 1;
      bM = mul(bM, bM);
    }
    return reduce(res);
  }
  static T inv(T x) {
    return modPow(x, _n - 2);
  }
  static uint64_t counter;
};

uT Montgomery::_n;
uT Montgomery::_np;

int Montgomery::_d;
uT Montgomery::_r2 = 0;

uint64_t Montgomery::counter = 0;

#endif
```

8.2.4 Poly.cpp

```
#ifndef POLY_H
#define POLY_H

#include <iostream>
#include <array>
#include <vector>
#include <stack>
#include <random>
#include <initializer_list>
//#include <type_traits>
#include <boost/multiprecision/cpp_int.hpp>
#include <boost/type_traits.hpp>

#include "Montgomery.cpp"
#include "isaac/isaac64.h"
#include "RandomDist.cpp"
#include "Setup.cpp"

using namespace std;

Z primroot;
Z primrootinv;

vector<int> phi625 = { 0, 125, 250, 375, 500 };

template<size_t d, class T>
class Poly
{
    typedef uZ uT;
    friend Poly operator+(const Poly& lhs, const Poly& rhs)
    {
        Poly<d, T> res;
        if (_q == 0)
            for (int i = 0; i < d; i++)
                res[i] = lhs[i] + rhs[i];
        else
            for (int i = 0; i < d; i++) {
                res[i] = lhs[i] + rhs[i];
```

```
            if (res[i] > _q) res[i] -= _q;
            else if (res[i] < -_q) res[i] += _q;
            res[i] = Montgomery::add(lhs[i], rhs[i]);
        }
    return res;
}
friend Poly operator-(const Poly& lhs, const Poly& rhs)
{
    Poly<d, T> res;
    if (_q == 0)
        for (int i = 0; i < d; i++)
            res[i] = lhs[i] - rhs[i];
    else
        for (int i = 0; i < d; i++)
            res[i] = Montgomery::sub(lhs[i], rhs[i]);
    return res;
}
friend Poly operator*(T scalar, const Poly& rhs) {
    Poly<d, T> res;
    if (_q == 0) {
        for (int i = 0; i < d; i++) {
            res[i] = scalar * rhs[i];
        }
    }
    else {
        Poly<d, T> rhsM;
        for (int i = 0; i < d; i++) {rhsM[i] = Montgomery::augment(rhs[i]);}
        T Mscalar = Montgomery::augment(scalar);
        for (int i = 0; i < d; i++) {
            res[i] = Montgomery::mul(Mscalar, rhsM[i]);
        }
        for (int i = 0; i < d; i++) { res[i] = Montgomery::reduce(res[i]); }
    }
    return res;
}
friend Poly operator/(const Poly& lhs, T scalar) {
    Poly<d, T> res;
    for (int i = 0; i < d; i++) {
        res[i] = lhs[i] / scalar;
    }
    return res;
}
friend Poly operator*(const Poly& lhs, const Poly& rhs) {
    Poly<d, T> res;
    if (_q == 0) {
        for (int i = 0; i < d; i++) {
            for (int j = 0; j < d; j++) {
                if (j < i) {
                    res[j] -= lhs[i] * rhs[d - i + j];
                }
                else {
                    res[j] += lhs[i] * rhs[j - i];
                }
            }
        }
    }
    else {
        bool useFFT = true;
        if (useFFT) {
            return multFFT(lhs, rhs);
        }
        else {
            Poly<d, T> lhsM, rhsM;
            for (int i = 0; i < d; i++) {
                lhsM[i] = Montgomery::augment(lhs[i]);
                rhsM[i] = Montgomery::augment(rhs[i]);
            }
            for (int i = 0; i < d; i++) {
                for (int j = 0; j < d; j++) {
                    if (j < i) {
                        res[j] = Montgomery::sub(res[j], Montgomery::mul(lhsM[i],
                            rhsM[d - i + j]));
```

```
                        }
                        else {
                            res[j] = Montgomery::add(res[j], Montgomery::mul(lhsM[i],
                                rhsM[j - i]));
                        }
                    }
                }
                //cout << res << endl;
                for (int i = 0; i < d; i++) {
                    res[i] = Montgomery::reduce(res[i]);
                }
            }
        }
        return res;
    }
    friend Poly operator%(const Poly& lhs, const T modulo) {
        Poly<d, T> res;
        for (int i = 0; i < d; i++) {
            res[i] = lhs[i] % modulo;
            if (res[i] > modulo / 2) res[i] -= modulo;
            else if (res[i] <= -modulo / 2) res[i] += modulo;
        }
        return res;
    }
    friend ostream& operator<<(ostream& os, const Poly& rhs)
    {
        bool prettyPoly = false;
        bool firstPrint = true;
        if (prettyPoly) {
            for (int i = 0; i < d; i++) {
                if (rhs.coeffs[i] != 0) {
                    if (rhs.coeffs[i] > 0 && !firstPrint) os << "+";
                    os << rhs.coeffs[i];
                    if (i > 1) {
                        os << "x^" << i;
                    }
                    else if (i > 0) {
                        os << "x";
                    }
                    firstPrint = false;
                }
            }
            if (firstPrint) os << "0";
        }
        else {
            os << "[" << rhs.coeffs[0];
            for (int i = 1; i < d; i++) {
                os << " " << rhs.coeffs[i];
            }
            os << "]";
        }
        return os;
    }

protected:
    static T _q;
    static stack<T> _qStack;
    static UniformDist uniform_dist;
    std::array<T, d> coeffs;
public:
    Poly() : coeffs() {
        //coeffs[9] = 3;
    }
    Poly(T c) : coeffs() {
        coeffs[0] = c;
    }
    Poly(const initializer_list<T>& list) : coeffs() {
        int i = 0;
        for (T c : list) {
            coeffs[i] = c;
            i++;
        }
```

```
}
Poly operator-() {
    Poly<d, T> res;
    if(_q == 0)
        for (int i = 0; i < d; i++)
            res[i] = -this->coeffs[i];
    else
        for (int i = 0; i < d; i++) {
            if (coeffs[i] > 0) res[i] = _q - this->coeffs[i];
            else res[i] = -this->coeffs[i];
        }
    return res;
}
Poly& operator+=(const Poly& rhs) {
    if (_q == 0)
        for (int i = 0; i < d; i++)
            this->coeffs[i] += rhs.coeffs[i];
    else
        for (int i = 0; i < d; i++)
            this->coeffs[i] = Montgomery::add(this->coeffs[i], rhs.coeffs[i]);
    return *this;
}
Poly& operator/=(const T scalar) {
    for (int i = 0; i < d; i++) {
        this->coeffs[i] /= scalar;
    }
    return *this;
}
Poly modPos(const T modulo) {
    Poly<d, T> res;
    for (int i = 0; i < d; i++) {
        res[i] = (*this)[i] % modulo;
        if (res[i] < 0) res[i] += modulo;
    }
    return res;
}
Poly mod2(const Poly& rhs) {
    Poly<d, T> res = *this;
    int ldeg = res.deg(), rdeg = rhs.deg();
    if (ldeg < rdeg) return res;
    for (int i = ldeg; i >= rdeg; i--) {
        for (int j = 0; j <= rdeg; j++) {
            res[i + j - rdeg] = abs((res[i + j - rdeg] - res[i]*rhs[j]) % 2);
        }
    }
    return res;
}
Poly Maugment() {
    Poly<d, T> res;
    if (_q == 0)
        for (int i = 0; i < d; i++)
            res[i] = (*this)[i];
    else
        for (int i = 0; i < d; i++)
            res[i] = Montgomery::augment((*this)[i]);
    return res;
}
Poly Mreduce() {
    Poly<d, T> res;
    if (_q == 0)
        for (int i = 0; i < d; i++)
            res[i] = (*this)[i];
    else
        for (int i = 0; i < d; i++)
            res[i] = Montgomery::reduce((*this)[i]);
    return res;
}
T& operator[](size_t c)
{
    return coeffs[c];
}
T operator[](size_t c) const {
```

```
    return coeffs[c];
}
T shift() {
    T temp = coeffs[d];
    for (int i = d - 1; i > 0; i--) {
        coeffs[i] = coeffs[i - 1];
    }
    coeffs[0] = 0;
    return temp;
}
T maxAbsCoeff() const {
    T max = 0;
    for (int i = 0; i < d; i++) {
        T coeff = abs((*this)[i]);
        if (coeff > max) max = coeff;
    }
    return max;
}
int deg() const {
    int degree = 0;
    for (int i = 0; i < d; i++) {
        if (coeffs[i] != 0) degree = i;
    }
    return degree;
}
static Poly One() {
    Poly<d, T> res;
    //if (_q == 0)
    res[0] = 1;
    //else res[0] = Montgomery::augment(1);
    return res;
}
static void set_q(T q) {
    if (q != _q) {
        _q = q;
        Montgomery::set_mod(q);
        uniform_dist = UniformDist(q);
        primRootUnity();
    }
}
static void push_q(T q) {
    _qStack.push(q);
    //cout << "push length " << _qStack.size() << endl;
    set_q(q);
}
static void pop_q() {
    _qStack.pop();
    //cout << "pop length " << _qStack.size() << endl;

    set_q(_qStack.top());
}

static Poly Uniform() {
    Poly<d, T> res;
    for (int i = 0; i < d; i++) {
        res[i] = uniform_dist.get();
    }
    return res;
}
static Poly Chi() {
    Poly<d, T> res;
    //if(_q == 0)
    for (int i = 0; i < d; i++)
        res[i] = TernaryDist::get();
    /*else
        for(int i = 0; i < d; i++)
            res[i] = Montgomery::augment((T)dist(prng) - 1);*/
    return res;
}

static void primRootUnity() {
    //Z pow2 = 1;
```

```
    //while (pow2 < 2 * d) pow2 <<= 1;
    //Z a = (_q - 1) / pow2;
    //for (int i = 2; i < _q; i++) {
    //    if (Montgomery::modPow(i, pow2) != 1 && Montgomery::modPow(i, a*pow2/2)
         != 1) {
    //        primroot = Montgomery::modPow(i, a);
    //        break;
    //    }
    //}
    ////primrootpow2 = pow2;
    primroot = primroots[_q];
    primrootinv = Montgomery::inv(primroot);
}
static void reducePoly(array<Z, primrootpow2>& a) {
    if ((phi_m & (phi_m - 1)) == 0) {
        for (int i = primrootpow2 - 1; i >= d; i--) {
            a[i - d] = Montgomery::sub(a[i - d], a[i]);
        }
    }
    else if(phi_m == 625){
        auto phi = phi625;
        for (int i = primrootpow2 - 1; i >= d; i--) {
            for (int j = 0; j < phi.size() - 1; j++) {
                a[i - d + phi[j]] = Montgomery::sub(a[i - d + phi[j]], a[i]);
            }
        }
    }
    else {
        for (int i = primrootpow2 - 1; i >= d; i--) {
            for (int j = i - d; j <= i - 1; j++) {
                a[j] = Montgomery::sub(a[j], a[i]);
            }
        }
    }
}
static Poly multFFT(const Poly& lhs, const Poly& rhs) {
    array<Z, primrootpow2> a = {}, b = {};
    for (int i = 0; i < d; i++) {
        a[i] = Montgomery::augment(lhs[i]);
        b[i] = Montgomery::augment(rhs[i]);
    }
    DFT(a, false);
    DFT(b, false);
    for (int i = 0; i < primrootpow2; i++) a[i] = Montgomery::mul(a[i], b[i]);
    DFT(a, true);
    Poly<d, Z> res;
    reducePoly(a);
    for (int i = 0; i < d; i++) res[i] = Montgomery::reduce(a[i]);
    return res;
}
static Poly multFFTaugmented(array<Z, primrootpow2>& a, array<Z, primrootpow2>& b
    ) {
    array<Z, primrootpow2> c;
    for (int i = 0; i < primrootpow2; i++) c[i] = Montgomery::mul(a[i], b[i]);
    DFT(c, true);
    Poly<d, Z> res;
    for (int i = primrootpow2 - 1; i >= d; i--) {
        c[i - d] = Montgomery::sub(c[i - d], c[i]);
    }
    for (int i = 0; i < d; i++) res[i] = Montgomery::reduce(c[i]);
    return res;
}
static void DFT(array<Z, primrootpow2>& a, bool inv) {
    int n = primrootpow2;
    Z one = Montgomery::augment(1);
    for (int i = 1, j = 0; i < n; i++) {
        int bit = n >> 1;
        for (; j & bit; bit >>= 1)
            j ^= bit;
        j ^= bit;

        if (i < j)
```

```
                swap(a[i], a[j]);
        }

        for (int len = 2; len <= n; len <<= 1) {
            Z wlen = inv ? primrootinv : primroot;
            wlen = Montgomery::augment(wlen);
            for (int i = len; i < primrootpow2; i <<= 1)
                wlen = Montgomery::mul(wlen, wlen);

            for (int i = 0; i < n; i += len) {
                Z w = one;
                for (int j = 0; j < len / 2; j++) {
                    Z u = a[i + j];
                    Z v = Montgomery::mul(a[i + j + len / 2], w);
                    a[i + j] = Montgomery::add(u, v);
                    a[i + j + len / 2] = Montgomery::sub(u, v);
                    w = Montgomery::mul(w, wlen);
                }
            }
        }

        if (inv) {
            Z n_1 = Montgomery::inv(n);
            n_1 = Montgomery::augment(n_1);
            for (Z& x : a)
                x = Montgomery::mul(x, n_1);
        }
    }
};

template<size_t d, class T>
T Poly<d, T>::_q = 0;
template<size_t d, class T>
stack<T> Poly<d, T>::_qStack;

template<size_t d, class T>
UniformDist Poly<d, T>::uniform_dist;

#endif
```

8.2.5 Matrix.cpp

```
#include <iostream>
#include <vector>
#include "isaac/isaac64.h"
#include "Poly.cpp"
#include "Setup.cpp"

using namespace std;

extern const int d;

template<class T>
class Matrix {
    friend Matrix operator+(const Matrix& lhs, const Matrix& rhs) {
        if (lhs._cols != rhs._cols || lhs._rows != rhs._rows) {
            throw "Size mismatch";
        }
        Matrix<T> res(lhs._rows, lhs._cols);
        for (int i = 0; i < lhs._rows; i++) {
            for (int j = 0; j < lhs._cols; j++) {
                res(i, j) = lhs(i, j) + rhs(i, j);
            }
        }
        return res;
    }
    friend Matrix operator-(const Matrix& lhs, const Matrix& rhs) {
        if (lhs._cols != rhs._cols || lhs._rows != rhs._rows) {
            throw "Size mismatch";
        }
        Matrix<T> res(lhs._rows, lhs._cols);
```

```
        for (int i = 0; i < lhs._rows; i++) {
            for (int j = 0; j < lhs._cols; j++) {
                res(i, j) = lhs(i, j) - rhs(i, j);
            }
        }
        return res;
    }

    template<class S>
    friend Matrix operator*(S scalar, const Matrix& rhs) {
        Matrix<T> res(rhs._rows, rhs._cols);
        for (int i = 0; i < rhs._rows; i++) {
            for (int j = 0; j < rhs._cols; j++) {
                res(i, j) = scalar * rhs(i, j);
            }
        }
        return res;
    }

    friend Matrix operator*(const Matrix& lhs, const Matrix& rhs) {
        //cout << "(" << lhs._rows << "," << lhs._cols << ")x(" << rhs._rows << ","
            << rhs._cols << ")" << endl;
        if (lhs._cols != rhs._rows) throw "Size mismatch";
        Matrix<T> res(lhs._rows, rhs._cols);
        for (int i = 0; i < res._rows; i++) {
            for (int j = 0; j < res._cols; j++) {
                vector<T> sum(8);
#pragma omp parallel for
                for (int k = 0; k < lhs._cols; k++) {
                    sum[omp_get_thread_num()] += lhs(i, k) * rhs(k, j);
                }
                T totalsum = T(0);
                for (int k = 0; k < sum.size(); k++) totalsum += sum[k];
                res(i, j) = totalsum;
            }
        }
        return res;
        Matrix<array<Z, primrootpow2>> a(lhs.rows(), lhs.cols()), b(rhs.rows(), rhs.
            cols());
#pragma omp parallel for collapse(2)
        for (int i = 0; i < lhs.rows(); i++) {
            for (int j = 0; j < lhs.cols(); j++) {
                for (int k = 0; k < d; k++) {
                    a(i, j)[k] = Montgomery::augment(lhs(i, j)[k]);
                }
                Poly<d, Z>::DFT(a(i, j), false);
            }
        }
#pragma omp parallel for collapse(2)
        for (int i = 0; i < rhs.rows(); i++) {
            for (int j = 0; j < rhs.cols(); j++) {
                for (int k = 0; k < d; k++) {
                    b(i, j)[k] = Montgomery::augment(rhs(i, j)[k]);
                }
                Poly<d, Z>::DFT(b(i, j), false);
            }
        }
        for (int i = 0; i < res._rows; i++) {
            for (int j = 0; j < res._cols; j++) {
                vector<T> sum(8);
#pragma omp parallel for
                for (int k = 0; k < lhs._cols; k++) {
                    sum[omp_get_thread_num()] += Poly<d, Z>::multFFTaugmented(a(i, k)
                        , b(k, j));
                }
                T totalsum = T(0);
                for (int k = 0; k < sum.size(); k++) totalsum += sum[k];
                res(i, j) = totalsum;
            }
        }
        return res;
    }
```

```
    template<class S>
    friend Matrix operator%(const Matrix& lhs, const S modulo) {
        Matrix<T> res(lhs._rows, lhs._cols);
#pragma omp parallel for
        for (int i = 0; i < res.size(); i++) {
            res[i] = lhs[i] % modulo;
        }
        return res;
    }

    friend Matrix operator|(const Matrix& lhs, const Matrix& rhs) {
        if (lhs._rows != rhs._rows) throw "Size mismath";
        Matrix<T> res(lhs._rows, lhs._cols + rhs._cols);
        for (int i = 0; i < lhs._rows; i++) {
            for (int j = 0; j < lhs._cols; j++) res(i, j) = lhs(i, j);
            for (int j = 0; j < rhs._cols; j++) res(i, j + lhs._cols) = rhs(i, j);
        }
        return res;
    }
    friend ostream& operator<<(ostream& os, const Matrix& rhs)
    {
        for (int i = 0; i < rhs._rows; i++) {
            for (int j = 0; j < rhs._cols; j++) {
                os << rhs(i, j) << "   ";
            }
            os << endl;
        }
        return os;
    }
protected:
    std::vector<T> elems;
    size_t _rows, _cols;
public:
    Matrix() : elems(0) {
        _rows = 0;
        _cols = 0;
    }
    Matrix(size_t rows, size_t cols) : elems(rows* cols) {
        _rows = rows;
        _cols = cols;
    }
    Matrix operator-() {
        Matrix<T> res(this->_rows, this->_cols);
        for (size_t i = 0; i < res.size(); i++) {
            res[i] = -(*this)[i];
        }
        return res;
    }
    T& operator[](size_t e) {
        return elems[e];
    }
    T operator[](size_t e) const {
        return elems[e];
    }
    T& operator()(size_t row, size_t col) {
        return elems[row * _cols + col];
    }
    T operator()(size_t row, size_t col) const {
        return elems[row * _cols + col];
    }
    size_t size() const {
        return _rows * _cols;
    }
    size_t rows() const {
        return _rows;
    }
    size_t cols() const {
        return _cols;
    }
    Matrix range(int row_start, int col_start, int row_end, int col_end) {
        if (row_start < 0) row_start = this->_rows + row_start;
```

```
        if (row_end < 0) row_end = this->_rows + row_end;
        if (col_start < 0) col_start = this->_cols + col_start;
        if (col_end < 0) col_end = this->_cols + col_end;

        Matrix<T> res(row_end - row_start + 1, col_end - col_start + 1);
        for (size_t i = row_start; i <= row_end; i++) {
            for (size_t j = col_start; j <= col_end; j++) {
                res(i - row_start, j - col_start) = (*this)(i, j);
            }
        }
        return res;
    }
    Matrix transpose() {
        Matrix<T> res(this->_cols, this->_rows);
        for (int i = 0; i < this->_rows; i++) {
            for (int j = 0; j < this->_cols; j++) {
                res(j, i) = (*this)(i, j);
            }
        }
        return res;
    }
    T dot(const Matrix& rhs) {
        if (this->size() != rhs.size()) throw "Size mismath";
        T sum = T(0);
        for (int i = 0; i < this->size(); i++) {
            sum += (*this)[i] * rhs[i];
        }
        return sum;
    }
    Matrix outer(const Matrix& rhs) {
        Matrix<T> res(this->size(), rhs.size());

#pragma omp parallel for
        for (int i = 0; i < res.rows(); i++) {
            for (int j = 0; j < res.cols(); j++) {
                res(i, j) = (*this)[i] * rhs[j];
            }
        }
        return res;
    }
    Matrix addToFirstRow(const Matrix& rhs) {
        if (_cols != rhs.size()) {
            throw "Size mismatch";
        }
        Matrix<T> res(_rows, _cols);
        for (int j = 0; j < _cols; j++) {
            res(0, j) = (*this)(0, j) + rhs[j];
        }
        for (int i = 1; i < _rows; i++) {
            for (int j = 0; j < _cols; j++) {
                res(i, j) = (*this)(i, j);
            }
        }
        return res;
    }
    Matrix randomRowSum() {
        Matrix<T> sum(1, this->cols());
        for (int i = 0; i < this->rows(); i++) {
            if ((irand64() & 1) == 1) {
                for (int j = 0; j < this->cols(); j++) {
                    sum(0, j) += (*this)(i, j);
                }
            }
        }
        return sum;
    }
    Matrix Maugment() {
        Matrix<T> res(this->_rows, this->_cols);
        for (int i = 0; i < this->size(); i++) {
            res[i] = elems[i].Maugment();
        }
        return res;
```

```
    }
    Matrix Mreduce() {
        Matrix<T> res(this->_rows, this->_cols);
        for (int i = 0; i < this->size(); i++) {
            res[i] = elems[i].Mreduce();
        }
        return res;
    }
    template <typename F>
    static Matrix Random(F r, size_t rows, size_t cols) {
        Matrix<T> res(rows, cols);
        for (int i = 0; i < res.size(); i++) {
            res[i] = r();
        }
        return res;
    }
    template<typename S>
    static Matrix Uniform(S q, size_t rows, size_t cols) {
        return Random([q]() {return T::Uniform(); }, rows, cols);
    }
    static Matrix Chi(size_t rows, size_t cols) {
        return Random(T::Chi, rows, cols);
    }
};
```

8.2.6 RandomDist.cpp

```
#ifndef RandomDistCPP
#define RandomDistCPP

#include "isaac/isaac64.h"
#include "Setup.cpp"

class UniformDist {
  typedef Z T;
  typedef uZ uT;
protected:
  int bits = 0;
  int blocks64 = 0;
  int Tsize = 0;
  T max = 0;
public:
  UniformDist() {};
  UniformDist(T _max) {
    bits = ceil(log2(static_cast<double>(_max)));
    blocks64 = ceil((double)bits / 64);
    Tsize = sizeZ * 8;
    max = _max;
  }
  T get() {
    uT rnd;
    do {
      rnd = irand64();
      for (int i = 1; i < blocks64; i++) {
        uT t = irand64();
        t <<= 64 * i;
        rnd += t;
      }
      rnd <<= Tsize - bits;
      rnd >>= Tsize - bits;
    } while (rnd >= max);
    return rnd;
  }
};

class TernaryDist {
public:
  static int get() {
    uint64_t a = irand64();
    uint8_t b = a & 1, c = (a & 2) >> 1;
    if (b == c) return 0;
```

```
    else if (b == 1) return 1;
    else return -1;
  }
};

#endif
```

8.2.7 SIMD.cpp

```
#ifndef SIMD_H
#define SIMD_H
#include "Setup.cpp"
#include "Poly.cpp"

vector<Poly<d, Z>> F257 = {
  {1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1},
  {1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1},
  {1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1},
  {1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1},
  {1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1},
  {1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1},
  {1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1},
  {1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1},
  {1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1},
  {1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1},
  {1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1},
  {1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1},
  {1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1},
  {1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1},
  {1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1},
  {1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1}
};

vector<Poly<d, Z>> G257 = {
  {0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1},
  {0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1},
  {0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1},
  {0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1},
  {0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
  {0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1},
  {0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
  {1, 0, 1, 0, 0, 0, 0, 0, 1},
  {1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1},
  {1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1},
  {1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1},
  {1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1},
  {1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1},
  {1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1},
  {1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1},
  {1, 0, 0, 0, 0, 0, 1, 0, 1}
};

vector<Poly<d, Z>> B257(16);

void InitBatching() {
  //Poly<d, Z> F = F257[0];
  //for (int i = 1; i < 16; i++) F = F * F257[i];
  //cout << (F) % 2 << endl;
  for (int i = 0; i < 16; i++) {
    B257[i] = G257[i];
    for (int j = 0; j < 16; j++) {
      if (i != j) B257[i] = B257[i] * F257[j];
    }
  }
}

Poly<d, Z> Batch(vector<Poly<d, Z>> moduli) {
  Poly<d, Z> res;
  for (int i = 0; i < moduli.size(); i++) {
    res += B257[i] * moduli[i];
  }
```

```
  res = res % 2;
  return res;
}

vector<Poly<d, Z>> Unbatch(Poly<d, Z> p) {
  vector<Poly<d, Z>> res(16);
  for (int i = 0; i < 16; i++) {
    res[i] = p.mod2(F257[i]);
  }
  return res;
}

#endif
```