Università degli Studi di Milano
Facoltà di Scienze e Tecnologie
Corso di Laurea in Sicurezza dei sistemi e delle reti informatiche

CRITTOGRAFIA POST-QUANTISTICA BASATA SUI RETICOLI: IMPLEMENTAZIONE E CRITTOANALISI DI GGH

Relatore: Prof. Stelvio CIMATO

Tesi di:
Gabriele BOTTANI
Matricola: 01701A

Anno Accademico 2023-2024

Quando la crittografia sarà messa fuori legge,
fbyb v shbevyrttr nienaab cevinpl.
- John Perry Barlow

Ringraziamenti

Desidero esprimere la mia gratitudine a diverse persone che mi hanno assistito, in maniera diretta o indiretta, nel processo di stesura di questo elaborato. In primo luogo, rivolgo un sincero ringraziamento al mio relatore, il professor Stelvio Cimato, per avermi offerto l'opportunità di intraprendere questa tesi e per il suo prezioso supporto durante tutto il suo sviluppo. Un secondo ringraziamento va ai miei amici, in particolare quelli che ho conosciuto durante i miei tre anni di università e quelli che invece conosco da una vita. La vostra presenza è stata importante perchè è grazie a voi se vivo sempre con il sorriso e con la tranquillità di avere sempre qualcuno su cui contare e con cui distrarmi nei momenti in cui ne avevo bisogno. Non posso tralasciare, sebbene possa sembrare scontato, di esprimere la mia più profonda gratitudine anche ai miei genitori, Serena e Antonio. Il loro costante supporto e incoraggiamento sono stati pilastri fondamentali nel mio percorso. Senza di loro, non solo non sarei qui a scrivere queste parole, ma non avrei nemmeno avuto la possibilità di conseguire questo traguardo accademico. Il loro ruolo è stato determinante non solo nel rendermi la persona che sono oggi, ma anche nel permettermi di raggiungere questo importante obiettivo formativo. Un doveroso ringraziamento va anche ai revisori, il cui contributo è stato fondamentale per il perfezionamento di questo lavoro. La loro attenzione ai dettagli e i loro suggerimenti hanno notevolmente arricchito la qualità dell'elaborato, permettendomi di affinare le mie argomentazioni e di presentare un lavoro più completo e rigoroso. Alessandro, il mio migliore amico, per i suoi preziosi consigli in ambito informatico; Filippo, conosciuto relativamente da poco, ma che ha fornito un contributo fondamentale nella precisazione e correzione a livello matematico; mio zio Vittorino, per il suo aiuto nella riformulazione e correzione logico-sintattica delle frasi. Un ringraziamento speciale va a quest'ultimo non solo per avermi messo a disposizione la macchina su cui sono stati eseguiti tutti gli esperimenti presentati, senza la quale non avrei potuto fornire risultati così precisi e validi. Devo inoltre riconoscere che è grazie a lui se sono arrivato a questo punto del mio percorso: la sua passione per l'informatica non solo mi ha dato l'ispirazione necessaria per intraprendere questa strada, ma mi ha anche fornito un modello da seguire. Un ultimo ringraziamento, ma indubbiamente non per importanza, va alla mia ragazza, la mia Chia. Lei è stata la fonte costante di supporto e incoraggiamento durante tutto il

mio percorso da quasi tre anni a questa parte, la persona con cui ho potuto sfogarmi e allo stesso tempo festeggiare. La sua pazienza, comprensione e il suo amore incondizionato sono stati fondamentali per superare questi anni e lo saranno anche per i prossimi avvenire. Un'ultima volta grazie a tutti voi che avete contribuito in modi diversi ma ugualmente preziosi al mio cammino. Questo traguardo non è solo mio, ma il risultato del sostegno, dell'amicizia e della compagnia che ognuno di voi mi ha fornito. Grazie di cuore.

Indice

Elenco delle tabelle

Elenco delle figure

Capitolo 1

Introduzione

Nell'era digitale odierna, la sicurezza delle informazioni è diventata una priorità cruciale. La crittografia, in particolare quella basata sui reticoli, svolge un ruolo fondamentale in questo contesto. I reticoli offrono una solidità matematica e resistenza anche ai computer quantistici, una minaccia emergente per molti sistemi crittografici tradizionali. Il sistema GGH, proposto nel 1997, rappresenta una pietra miliare nella crittografia basata sui reticoli, sfruttando la complessità del problema dei vettori più vicini sui reticoli. Tuttavia, il sistema originale GGH ha mostrato vulnerabilità, stimolando la ricerca di miglioramenti come la variante GGH-HNF. Questa tesi si propone di rivisitare approfonditamente GGH e GGH-HNF, valutandone la potenziale applicabilità pratica nel contesto tecnologico attuale.
Il Capitolo 2 si concentra sulle proprietà e i problemi relativi ai reticoli. Viene innanzitutto presentata una panoramica dettagliata sui reticoli, tra cui la definizione formale, le nozioni base come la proprietà dei coefficienti integrali, e il concetto del dominio fondamentale. Vengono inoltre introdotti i principali problemi reticolari di interesse crittografico, ovvero il Problema del Vettore più Corto (SVP) e il Problema del Vettore più Vicino (CVP). Il capitolo prosegue illustrando algoritmi chiave per la riduzione di basi reticolari, introducendo prima l'ortogonalizzazione Gram-Schmidt necessaria per la riduzione e successivamente l'algoritmo di Lenstra-Lenstra-Lovász (LLL) e sue varianti migliorate. Questi algoritmi permettono di trasformare basi "cattive" in basi "buone", migliorando il rapporto di Hadamard che misura la qualità della base. Infine, il capitolo si concentra su algoritmi per la risoluzione approssimata del CVP, come gli algoritmi di Babai e la tecnica di incorporamento, che sfruttano basi reticolari ridotte per ottenere soluzioni efficaci.
Il Capitolo 3 descrive e crittoanalizza il crittosistema a chiave pubblica Goldreich Goldwasser Halevi (GGH), oggetto di questa tesi. Viene spiegato in dettaglio il funzionamento di GGH, con particolare focus sulla generazione delle chiavi pubblica e privata. Il capitolo analizza poi le principali vulnerabilità del sistema GGH originale,

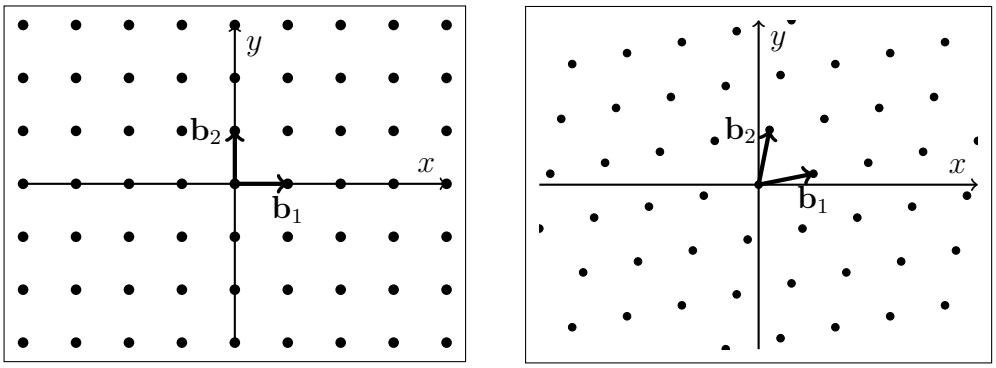
come gli attacchi di calcolo della chiave privata, risoluzione diretta del CVP, l'attacco di Nguyen che sfrutta la struttura del vettore di errore, e gli attacchi basati su informazioni parziali del messaggio. Per ciascun attacco sono forniti esempi pratici e discusse le possibili contromisure proposte.
Il Capitolo 4 presenta la variante migliorata del sistema GGH, nota come GGH-HNF, proposta da Daniele Micciancio nel 2001. Questa versione mira a risolvere le vulnerabilità della versione originale di GGH aumentandone sia le performance che la sicurezza. Il capitolo descrive in dettaglio la struttura e il funzionamento di GGH-HNF, spiegando come Micciancio abbia scelto di utilizzare la forma normale di Hermite (HNF) per generare la chiave pubblica invece della costruzione casuale originale. In questo capitolo vengono anche riportati anche i dati riguardanti i limiti pratici di GGH-HNF, rilevati da un rapporto tecnico di Christoph Ludwig nel 2004.
Il Capitolo 5 esplora l'implementazione dei crittosistemi GGH e GGH-HNF, strutturata come un pacchetto Python. Questa sezione offre una dettagliata giustificazione delle scelte progettuali e presenta un'analisi approfondita dei tre moduli principali che compongono il pacchetto: GGH e GGH-HNF, che implementano rispettivamente i due sistemi in esame, e Utils, un modulo di supporto che fornisce funzioni e metodi di utilità generale utilizzati da entrambi i crittosistemi.
Il Capitolo 6 espone i risultati sperimentali dei test condotti sui sistemi crittografici precedentemente discussi. Presenta dati dettagliati sui tempi di esecuzione per ciascuna fase operativa e sulle dimensioni delle chiavi e dei testi cifrati. L'analisi si conclude con un confronto tra questi risultati e quelli ottenuti da studi precedenti sugli stessi crittosistemi, nonché con schemi crittografici tradizionali come RSA ed ElGamal.
Il Capitolo 7, infine, discute i risultati ottenuti e risponde alle domande poste dalla tesi, con particolare attenzione ai vantaggi e, soprattutto, gli svantaggi rilevati.

Capitolo 2

Proprietà e problemi sui reticoli

2.1 Reticoli

2.1.1 Nozioni base



(a) $\mathcal{L} = \mathbb{Z}^n$ (b) $\mathcal{L} = \mathbf{B}\mathbb{Z}^n$ dove $\mathbf{B} \in \mathbb{R}^{m \times n}$

Figura 1: Due esempi di strutture reticolari.

Un reticolo è un insieme di punti in uno spazio di dimensione n che forma una struttura periodica. Ogni punto del reticolo può essere generato come combinazione lineare di n vettori, chiamati base, che sono linearmente indipendenti tra loro. La struttura e le proprietà di un reticolo dipendono dai vettori di base che, partendo dall'origine, definiscono il suo pattern di disposizione indicando le direzioni e le distanze tra i punti del reticolo.
Una proprietà fondamentale su cui si basa la definizione di reticolo è la proprietà dei coefficienti integrali: la base di un reticolo ha sempre coefficienti integrali, il che significa che tutti i vettori nella base sono combinazioni lineari intere l'uno dell'altro.

I reticoli possono essere formati in diversi modi, il più comune è il reticolo quadrato (Figura 1a) nel quale la base è ortogonale con gli assi cartesiani. Le altre varianti sono ottenibili applicando delle trasformazioni lineari alla base del reticolo quadrato (Figura 1b).
I reticoli sono normalmente definiti in uno spazio bidimensionale o tridimensionale, ma il concetto può essere esteso a spazi di dimensioni superiori. La rappresentazione dei vettori in questa tesi è quella per riga, al contrario della scelta fatta dagli autori di [4] che hanno utilizzato una notazione per colonna nel loro crittosistema a chiave pubblica Goldreich Goldwasser Halevi (GGH), oggetto di questa tesi. Quindi per esempio, una matrice $\mathbf{B} \in \mathbb{R}^{m \times n}$ sarà divisa in vettori $[\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n]$.
Una base può essere rappresentata da una matrice $\mathbf{B} = [\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n] \in \mathbb{R}^{n \times n}$ avente, come precedentemente anticipato, i vettori base come righe. Utilizzando la matrice come notazione, il reticolo generato da una matrice $\mathbf{B} \in \mathbb{R}^{n \times n}$ può essere definito come $\mathcal{L}(\mathbf{B}) = \{\mathbf{xB} : \mathbf{x} \in \mathbb{Z}^n\}$, dove $\mathbf{xB}$ è una comune moltiplicazione matriciale.
Si definisca ora l'i-esimo minimo $\lambda_i(\mathcal{L})$ come il raggio della sfera più piccola, centrata nell'origine, che contiene i vettori linearmente indipendenti del reticolo. Si chiami "gap" il rapporto tra il secondo e il primo minimo, $\frac{\lambda_1(\mathcal{L})}{\lambda_2(\mathcal{L})}$. Questo valore misura la differenza relativa tra i due vettori più corti linearmente indipendenti del reticolo, fornendo un'indicazione importante sulla sua struttura. Più formalmente, dati n vettori linearmente indipendenti $\mathbf{b}_1, \ldots, \mathbf{b}_n \in \mathbb{R}^n$, il reticolo generato da essi è un set di vettori

$$\mathcal{L}(\mathbf{b}_1, \ldots, \mathbf{b}_n) = \sum_{i=1}^{n} \mathbf{b}_i \cdot \mathbb{Z} = \{\mathbf{xB} : \mathbf{x} \in \mathbb{Z}^n\}.$$

Lo stesso reticolo può essere generato da più basi composte ciascuna da vettori diversi

$$\mathcal{L} = \sum_{i=1}^{n} \mathbf{c}_i \cdot \mathbb{Z}.$$

Il determinante di un reticolo è il valore assoluto del determinante della matrice base $\det(\mathcal{L}(\mathbf{B})) = |\det(\mathbf{B})|$. Di conseguenza, per ogni matrice unimodulare (ovvero avente determinante +1 o -1) $\mathbf{U} \in \mathbf{Z}^{n \times n}$, $\mathbf{UB}$ è una base di $\mathcal{L}(\mathbf{B})$. Per verificare se due basi $\mathbf{R}$ e $\mathbf{B}$ generano lo stesso reticolo, è possibile utilizzare la matrice pseudo-inversa e trovare un $\mathbf{U}$ tale per cui $\mathbf{UR} = \mathbf{B}$.
Computando $\mathbf{R}^+$, ovvero la matrice pseudo-inversa di $\mathbf{R}$, si ha che:

$$\mathbf{U} = \mathbf{B}\ \mathbf{R}^+.$$

$\mathbf{R}^+$ è particolarmente facile da ottenere in questo caso in quanto $\mathbf{R}$ è una matrice quadrata e i suoi vettori riga di sono linearmente indipendenti per definizione. Di

conseguenza la matrice pseudo-inversa nient'altro è che la normale inversa:

$$\mathbf{R}^{+} = \mathbf{R}^{-1}.$$

Si ottiene quindi che:

$$\mathbf{U} = \mathbf{B}\mathbf{R}^{-1}.$$

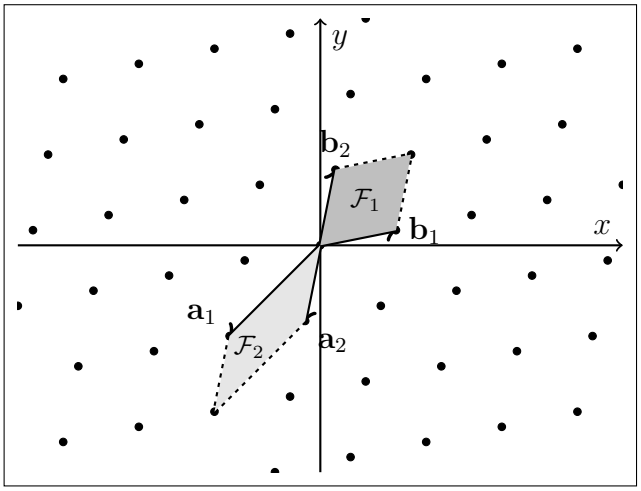


Figura 2: Un reticolo con due suoi domini fondamentali.

Esempio 2.1.1. (Verificare che due basi generino lo stesso reticolo)
Siano $\mathbf{R}$ e $\mathbf{B}$ due basi generanti entrambi il reticolo $\mathcal{L}$ con

$$\mathbf{R} = \begin{bmatrix} 1 & 2 \\ 3 & 0 \end{bmatrix} \quad \text{e} \quad \mathbf{B} = \begin{bmatrix} 5 & 4 \\ -6 & -6 \end{bmatrix}$$

allora deve esistere una matrice unimodulare $\mathbf{U}$ tale che $\mathbf{U}\mathbf{R} = \mathbf{B}$. Per trovare $\mathbf{U}$ è possibile calcolare:

$$\mathbf{U} = \mathbf{B}\mathbf{R}^{-1} = \begin{bmatrix} 2 & 1 \\ -3 & -1 \end{bmatrix}.$$

Ora è sufficiente controllare che

$$\mathbf{U}\mathbf{R} = \mathbf{B} \quad \text{ovvero} \quad \begin{bmatrix} 2 & 1 \\ -3 & -1 \end{bmatrix} \times \begin{bmatrix} 1 & 2 \\ 3 & 0 \end{bmatrix} = \begin{bmatrix} 5 & 4 \\ -6 & -6 \end{bmatrix}$$

inoltre dato che $\det(\mathbf{U}) = 1$, si può affermare che $\mathbf{R}$ e $\mathbf{B}$ sono entrambe basi di $\mathcal{L}$.

2.1.2 Dominio Fondamentale

Il dominio fondamentale è un concetto molto importante nei reticoli, grazie al quale è possibile capire la struttura matematica che li compone. Data una base arbitraria $\mathbf{B}$ e un reticolo $\mathcal{L}$ è possibile immaginare il dominio fondamentale come un parallelepipedo che ha come vertice l'origine e come lati i vettori base $\mathbf{b}$ generanti il reticolo.
Di tale parallelepipedo è possibile calcolarne il volume $\mathcal{F}(\mathbf{B})$, il quale è strettamente legato al determinante del reticolo. È possibile osservare in Figura 2 un reticolo con due sue basi: nonostante i domini fondamentali abbiano forme diverse, l'area coperta dal loro volume è la medesima. Come dimostrato in [16, sezione 7.4], proprio come per il determinante, il dominio fondamentale è un'invariante che è indipendente dalla scelta delle basi per il reticolo. Inoltre ne deriva la proprietà:

$$\mathcal{F}(\mathbf{B}) = \det(\mathcal{L})$$

e ricollegandoci a quanto detto nella sezione 2.1.1: $\mathcal{F}(\mathbf{B}) = \det(\mathcal{L}) = |\det(\mathbf{B})|$.
Una seconda proprietà fondamentale, sempre dimostrata in [16] è che tramite il dominio fondamentale è possibile ricostruire l'intero reticolo (Figura 3). In altre parole, ogni vettore $\mathbf{t} \in \mathbb{R}^n$ con $\mathcal{L} \subset \mathbb{R}^n$ può essere ottenuto sommando ripetutamente a un vettore $\mathbf{f} \in \mathcal{F}$ un altro vettore $\mathbf{v} \in \mathcal{L}$. Più formalmente:

$$\mathcal{F} + \mathbf{v} = \{\mathbf{f} + \mathbf{v} \mid \mathbf{f} \in \mathcal{F}, \mathbf{v} \in \mathcal{L}\}$$

comprende esattamente tutti i vettori nel reticolo $\mathcal{L}$.

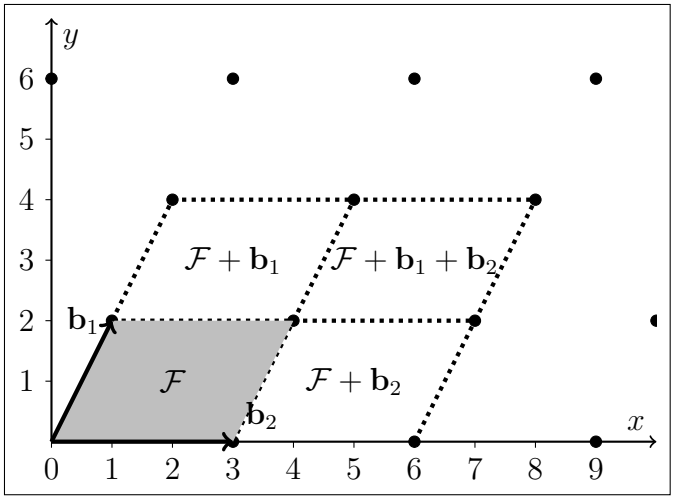


Figura 3: Il dominio fondamentale comprende esattamente tutti i vettori di $\mathcal{L}$.

2.2 Problemi sui reticoli

L'utilizzo della crittografia basata su reticoli parte dall'assunto che, soprattutto nei casi di spazi multidimensionali, la complessità computazionale derivante da determinati problemi su di essi, sia un limite invalicabile. I problemi reticolari più conosciuti e usati in ambito crittografico sono i seguenti:

- Problema del Vettore più Corto (SVP): Data una base $\mathbf{B}$ di un reticolo, trovare il vettore non nullo di lunghezza minima in $\mathcal{L}(\mathbf{B})$.
- Problema del Vettore più Vicino (CVP): Data una base $\mathbf{B}$ di un reticolo e un vettore target $\mathbf{t}$ (non necessariamente nel reticolo), trovare il vettore $\mathbf{w} \in \mathcal{L}(\mathbf{B})$ più vicino a $\mathbf{t}$ minimizzando $\|\mathbf{t} - \mathbf{w}\|_2$.
- Problema dei Vettori Lineramente Indipendenti più Corti (SIVP): Data una base $\mathbf{B} \in \mathbb{Z}^{n \times n}$ di un reticolo , trovare n vettori linearmente indipendenti $\mathbf{S} = [\mathbf{s}_1, \ldots, \mathbf{s}_n]$ (dove $\mathbf{s}_i \in \mathcal{L}(\mathbf{B})$) per tutte le i) minimizzando la quantità $||\mathbf{S}|| = \max_i ||\mathbf{s}_i||_2$. SIVP è una variante di SVP, ma a differenza di quest'ultimo, SIVP mira a identificare un insieme di vettori indipendenti che siano i più corti possibile, in altre parole la ricerca di una base ortogonale o ortonormale che generi il reticolo e che minimizzi la lunghezza dei suoi vettori.

La complessità per risolvere CVP è stata provata essere NP-difficile[1], stessa cosa vale per SVP, ma sotto alcune circostanze specifiche[9]. Per questi motivi vengono comparati come problemi dalla stessa difficoltà anche se, in pratica, risolvere CVP è considerato essere un po' più difficile di SVP a parità di dimensione. Ognuno di questi due problemi ha un relativo sotto-problema che nient'altro è che una variante approssimativa: il Problema del Vettore più Vicino Approssimato (apprCVP) e Problema del Vettore più Corto Approssimato (apprSVP). Questi sotto-problemi sono riferibili alla necessità di trovare un vettore non nullo la cui lunghezza sia maggiore di un fattore dato $\Psi(n)$, rispetto ad un vettore non nullo corretto che risulti essere più corto o più vicino, a seconda del problema.
In particolare GGH si basa sulla risoluzione del CVP basandosi su una delle proprietà fondamentali dei reticoli: la possibilità di usare più basi per lo stesso reticolo. Utilizzando due basi $\mathbf{A}$ e $\mathbf{B}$, definite rispettivamente come "buona" e "cattiva", ma che generano lo stesso reticolo, diventa più agevole risolvere determinati problemi sui reticoli utilizzando la base $\mathbf{A}$ piuttosto che la base $\mathbf{B}$. Per questi motivi il CVP sarà il fulcro dei problemi discussi in questa tesi assieme al SVP, il quale verrà trattato prevalentemente per quanto riguarda la crittoanalisi di GGH.

2.3 Riduzione di un reticolo

2.3.1 Rapporto di Hadamard

Si supponga di avere a disposizione due basi $\mathbf{A}$ e $\mathbf{B}$ che godono della proprietà di generare lo stesso reticolo. Seppur condividendo tale caratteristica, $\mathbf{A}$ e $\mathbf{B}$ sono in realtà molto diverse nella loro struttura; in particolare $\mathbf{A}$ è composta da vettori corti e quasi ortogonali fra loro mentre $\mathbf{B}$ è composta da vettori lunghi e quasi paralleli fra loro.
La qualità di una base risiede in queste differenze dei vettori costituenti le basi, chiamiamo quindi base "buona" $\mathbf{A}$ e base "cattiva" $\mathbf{B}$. È necessario però definire una metrica per valutare quanto una base sia buona o meno; a tal proposito Hadamard[16] introdusse una formula quantitativa per misurare la qualità di una base reticolare, il cosiddetto rapporto di Hadamard.

Data una base $\mathbf{B} = [\mathbf{b_1}, \mathbf{b_2}. \ldots, \mathbf{b}_n]$ e un reticolo $\mathcal{L}$ di dimensione n generato da $\mathbf{B}$, il rapporto di Hadamard della base $\mathbf{B}$ è definito dal valore:

$$\mathcal{H}(\mathbf{B}) = \left(\frac{\det(\mathcal{L})}{\|\mathbf{b}_1\|_2 \cdot \|\mathbf{b}_2\|_2, \cdots \|\mathbf{b}_n\|_2} \right)^{\frac{1}{n}}$$

Il rapporto di Hadamard si configura nell'intervallo $(0, 1]$, dove i valori che si avvicinano a 1 rappresentano la bontà della base, mentre i valori tendenti a zero sono indice di una base "cattiva". Questa formula verrà utilizzata come unica misura per verificare la qualità degli esempi di basi che verranno presentate più avanti in questa tesi.

2.3.2 Ortogonalizzazione Gram-Schmidt

Ora che è possibile giudicare una base dato il suo rapporto di Hadamard, utilizziamo la base $\mathbf{B}$ definita nella precedente sezione, la quale ipotizziamo abbia un $\mathcal{H}(\mathbf{B})$ prossimo allo zero. Se volessimo utilizzare questa base per risolvere uno dei problemi dei reticoli, molto probabilmente non riusciremmo mai a raggiungere una soluzione che sia valida o quantomeno che sia vicina alla soluzione ottima. A questo proposito sono stati ideati degli algoritmi in grado di ortogonalizzare una base cattiva per convertirla in una buona e mantenere le proprietà del reticolo iniziale, si ottiene quindi una base $\mathbf{B}'$ tale che: $\mathcal{H}(\mathbf{B}^*) \approxeq 1$ e che $\det(\mathbf{B}) = \det(\mathbf{B}^*)$. Nell'ambito della riduzione di reticoli è importante notare come il gap di un reticolo giochi un ruolo chiave: è noto che più il gap è grande e più la riduzione è semplice.

Prima di discutere questo tipo di algoritmi è necessario affrontare brevemente l'algoritmo di Gram-Schmidt, il quale, esegue un tipo di ortogonalizzazione che viene applicata su spazi vettoriali e che è anche chiamata Ortogonalizzazione Gram-Schmidt (GSO). Questo algoritmo non è adottabile direttamente sulle basi reticolari in quanto esso andrebbe a violare la proprietà dei coefficienti integrali, di fondamentale importanza nella definizione di reticolo. Nonostante ciò, questo algoritmo gode di una proprietà chiave che viene utilizzata in algoritmi di riduzione dei reticoli. Come dimostrato in [16, Teorema 7.13]:
siano $\text{span}(\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n)$ e $\text{span}(\mathbf{b}_1^*, \mathbf{b}_2^*, \ldots, \mathbf{b}_n^*)$ gli spazi vettoriali generati rispettivamente dalle righe di $\mathbf{B}$ e $\mathbf{B}^*$, allora se $\mathbf{B}^*$ è il risultato di GSO applicato a $\mathbf{B}$:

$$\text{span}(\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n) = \text{span}(\mathbf{b}_1^*, \mathbf{b}_2^*, \ldots, \mathbf{b}_n^*).$$

Quindi facendo uso del GSO come subroutine di un algoritmo per la trasformazione delle matrici di spazi vettoriali, si ottiene una importante riduzione del costo computazionale e nel contempo, si semplifica l'implementazione di algoritmi per la riduzione di reticoli.

Algoritmo 1: Algoritmo di Gram-Schmidt

Input: Una matrice $\mathbf{B}$ tale che rango($\mathbf{B}$) = righe($\mathbf{B}$)
Output: Una matrice $\mathbf{B}^*$ ortogonale
$\mathbf{b}_1^* = \mathbf{b}_1$
for $i = 2$ **to** n **do**
  for $j = 1$ **to** $i - 1$ **do**
    $\mu_{i,j} = \frac{\langle \mathbf{b}_i, \mathbf{b}_j^* \rangle}{\langle \mathbf{b}_j^*, \mathbf{b}_j^* \rangle}$
  end
  $\mathbf{b}_i^* = \mathbf{b}_i - \sum_{j=1}^{i-1} \mu_{i,j} \mathbf{b}_j^*$
end
return $\mathbf{B}^*$

Dove rango() indica il rango e righe() il numero delle righe.

2.3.3 Algoritmo di Lenstra-Lenstra-Lovász

L'algoritmo di Lenstra-Lenstra-Lovász (LLL)[5, 16] è noto come uno dei più famosi algoritmi per la riduzione dei reticoli. In teoria, opera con un tempo polinomiale $O(n^6(\log \mathcal{E})^3)$, dove n è la dimensione di un reticolo $\mathcal{L}$ dato ed $\mathcal{E}$ rappresenta la massima lunghezza euclidea dei vettori nella base fornita. Il risultato di LLL è una base

ridotta, il cui vettore più breve può essere considerato una soluzione approssimata dell'apprSVP con un fattore di $2^{(n-1)/2}$ rispetto alla lunghezza del vettore corretto più corto. Nonostante questo fattore sia esponenziale, l'algoritmo si dimostra sorprendentemente efficiente nella pratica e ciò lo rende un tassello fondamentale negli algoritmi di risoluzione del CVP e del SVP.
Una base $\mathbf{B}^*$ per essere considerata LLL-ridotta deve soddisfare due condizioni:

- Condizione di grandezza: $\mu_{i,j} = \frac{\langle \mathbf{b}_i, \mathbf{b}_j^* \rangle}{\langle \mathbf{b}_j^*, \mathbf{b}_j^* \rangle} \leq \eta$ per ogni $1 \leq j < i \leq n$.
- Condizione di Lovász: $\langle \mathbf{b}_i^*, \mathbf{b}_i^* \rangle \geq (\delta - \mu_{i,i-1}^2) \langle \mathbf{b}_{i-1}^*, \mathbf{b}_{i-1}^* \rangle$ per ogni $1 < i \leq n$.

I valori δ e η sono standardizzati rispettivamente a $\frac{3}{4}$ e $\frac{1}{2}$.

Algoritmo 2: Algoritmo di LLL, con $\delta = \frac{3}{4}, \eta = \frac{1}{2}$

Input: Una matrice $\mathbf{B}$ che sia base di un reticolo
Output: Una matrice $\mathbf{B}^*$ LLL-ridotta
$k = 1$
$\mathbf{b}_1^* = \mathbf{b}_1$
while $k <= n$ **do**
  for $j = k - 1$ **to** 0 **do**
    if $|\mu_{k,j}| > \eta$ **then** `// Condizione di grandezza`
      $\mathbf{b}_k = \mathbf{b}_k - \lfloor \mu_{k,j} \rceil \mathbf{b}_j$
      GramSchmidt($\mathbf{B}$)
    end
  end
  if $\langle \mathbf{b}_k^*, \mathbf{b}_k^* \rangle \geq (\delta - \mu_{k,k-1}^2) \langle \mathbf{b}_{k-1}^*, \mathbf{b}_{k-1}^* \rangle$ **then** `// Condizione di Lovász`
    $k = k + 1$
  end
  else
    Scambia $\mathbf{b}_{k-1}$ e $\mathbf{b}_k$
    GramSchmidt($\mathbf{B}$)
    $k = \max(k - 1, 1)$
  end
end

Esempio 2.3.1. (Riduzione di una base con LLL) Sia $\mathcal{L}$ un reticolo generato dalla matrice:

$$\mathbf{B} = \begin{bmatrix} 87634 & 88432 & 94345 \\ 32323 & 27883 & 40323 \\ -21221 & -11234 & -32123 \end{bmatrix}$$

$$\text{con } \mathcal{H}(\mathbf{B}) = 0.14318 \text{ e } \det(\mathbf{B}) = -1079906335101.$$

Si applichi ora una riduzione LLL alla matrice $\mathbf{B}$:

$$\mathbf{B}^* = \begin{bmatrix} -784 & 632 & 2701 \\ -14823 & 9207 & -7717 \\ -13454 & -14753 & -97 \end{bmatrix}$$

Ricalcolando $\mathcal{H}(\mathbf{B}^*)$ è possibile osservare che:

$$\mathcal{H}(\mathbf{B}^*) = 0.99442 \text{ e } \det(\mathbf{B}^*) = 1079906335101.$$

È stato ottenuto un incremento notevole del rapporto di Hadamard grazie alla riduzione LLL senza interferire con le proprietà della base $\mathbf{B}$. Infatti $|\det(\mathbf{B})| = |\det(\mathbf{B}^*)|$ e, usando la formula descritta nella Sezione 2.1, è possibile trovare una matrice di interi

$$\mathbf{U} = \begin{bmatrix} -1 & 4 & 2 \\ -6 & 25 & 14 \\ -3 & 11 & 5 \end{bmatrix} \quad \text{con } \det(\mathbf{U}) = -1$$

tale per cui $\mathbf{UB} = \mathbf{B}^*$.

2.3.4 Varianti di LLL

LLL è un eccellente algoritmo in grado di restituire in un tempo polinomiale una matrice quasi ortogonale partendo da una con vettori quasi paralleli, o che comunque è ritenibile di bassa qualità. Esistono però varianti che ne velocizzano i calcoli, così come altri algoritmi capaci di restituire una base di qualità ancora superiore rispetto a quella ottenuta con la riduzione LLL. Di seguito vengono presentate brevemente tre versioni dell'algoritmo.

La prima versione discussa è quella in virgola mobile (FPLLL)[12], la quale utilizza aritmetica in virgola mobile a precisione arbitraria per accelerare i calcoli razionali dell'algoritmo originale. Questa versione ha come vantaggio un aumento delle performance: in comparazione con LLL il tempo di computazione nel caso peggiore è $O(n^3(\log \mathcal{E})^2)$. È importante notare che l'utilizzo di aritmetica a virgola mobile per velocizzare i calcoli è una tecnica comune e utilizzabile per tutti gli algoritmi di riduzione dei reticoli spiegati in questa sezione.

La seconda versione è stata presentata da Schnorr-Euchner [14] ed il suo nome originale è "deep insertions" ovvero inserzioni profonde. In LLL (Algoritmo 2), è presente un passaggio in cui avviene uno scambio tra il vettore $\mathbf{b}_{k-1}$ e $\mathbf{b}_k$, il quale di solito permette qualche riduzione di grandezza ulteriore del nuovo $\mathbf{b}_k$. Nella variante deep insertions, viene invece inserito $\mathbf{b}_k$ tra $\mathbf{b}_{i-1}$ e $\mathbf{b}_i$ con i che viene scelta in modo

da apportare una maggiore riduzione di grandezza. L'algoritmo risultante, nel caso peggiore, potrebbe non terminare in un tempo polinomiale, ma in pratica, quando eseguito sulla maggioranza dei reticoli, termina rapidamente e può fornire in output una base ridotta significativamente migliore di quella di LLL standard.
L'ultima variante discussa è basata sull'algoritmo di riduzione Korkin–Zolotarev (KZ)[3, sezione 18.5]. Le caratteristiche di una base KZ-ridotta sono generalmente migliori rispetto a quelle di LLL, ma richiedono una complessità maggiore e un tempo di computazione non polinomiale; per le proprietà complete si veda il riferimento sopracitato. Più nel dettaglio, il problema principale, è che non esiste un algoritmo in grado di computare una base KZ in tempo polinomiale, infatti, l'algoritmo più veloce conosciuto, richiede un tempo di computazione esponenziale rispetto alla dimensione. KZ offre un notevole vantaggio per la riduzione del carico computazionale, in quanto il primo vettore di una base KZ-ridotta è sempre una soluzione al SVP. Dato che la complessità di KZ cresce con n, è logico pensare che a basse dimensioni sia comunque sufficientemente veloce. Un'idea è quindi quella di computare una riduzione di proiezioni a dimensioni più basse del reticolo originale. L'algoritmo in questa configurazione prende il nome di Korkine-Zolotarev a blocco (BKZ), il quale, se combinato con LLL, diventa una variante di quest'ultimo chiamata LLL-BKZ. Questa variante è in grado di bilanciare costo computazionale e qualità di riduzione ottenendo così l'algoritmo più efficiente per SVP in grandi dimensioni, dimostrando anche una qualità di riduzione significativamente migliore di quella di LLL standard.

Per reticoli di dimensioni ancora maggiori, dove anche BKZ potrebbe risultare computazionalmente oneroso, è stata sviluppata una versione ulteriormente ottimizzata chiamata BKZ "pruned" o potata [15]. Questa variante mantiene l'efficacia di BKZ nel bilanciare costo computazionale e qualità della riduzione, ma introduce una tecnica di potatura nell'enumerazione dei vettori. Tale tecnica è spesso usata in informatica al fine di ottimizzare algoritmi riducendo lo spazio di ricerca, permettendo così di ottenere soluzioni approssimate (e solitamente corrette) in tempi significativamente minori rispetto all'esplorazione completa.

2.4 Algoritmi per la risoluzione del CVP

2.4.1 Algoritmi di Babai

Nel 1986 Babai[2] propose due algoritmi per la risoluzione di apprCVP, i cosiddetti: "Metodo del Piano più Vicino" e "Tecnica di Arrotondamento". Ai fini di questa tesi, entrambi verranno trattati, sebbene il primo sarà discusso in modo più conciso poiché, come verrà spiegato nei prossimi capitoli, non è stato utilizzato nelle implementazioni proposte.

Il metodo del piano più vicino è il primo algoritmo presentato da Babai; esso si basa sull'impiegare l'ortogonalizzazione di Gram-Schmidt per semplificare il problema. L'algoritmo infatti, in una fase iniziale, provvede ad ortogonalizzare la base del reticolo fornita in input. Successivamente, procede in maniera iterativa a partire dalla dimensione più alta: il vettore input viene proiettato sul vettore base ortogonale corrispondente e questa proiezione viene approssimata al multiplo intero più vicino del vettore della base originale. Tale approssimazione viene sottratta dal vettore input, generando un nuovo vettore residuo. Questo processo viene ripetuto per le dimensioni inferiori, una alla volta, fino a coprire tutte le dimensioni. Al termine, l'algoritmo fornisce come risultato una soluzione all'apprCVP. Grazie alla sua complessità polinomiale, l'algoritmo riesce a bilanciare efficacemente l'accuratezza dell'approssimazione con il tempo di esecuzione. Per ulteriori dettagli e informazioni sull'algoritmo si veda [3].
Il secondo algoritmo è la tecnica di arrotondamento che, come da nome, si basa principalmente sull'arrotondare dei valori frazionari all'intero più vicino. Seppur la sua implementazione risulti semplice e banale, in realtà la sua dimostrazione teorica è tutt'altro che immediata. A differenza del precedente, non utilizza l'ortogonalizzazione di Gram-Schmidt, ma mantiene comunque una complessità polinomiale. Di seguito viene fornita una spiegazione del suo funzionamento.
Come discusso nella Sezione 2.1.2, dati un reticolo $\mathcal{L}$ di dimensione n e una sua base $\mathbf{B}$, per ogni vettore $\mathbf{t} \in \mathbb{R}^n$, con $\mathbf{t} \notin \mathcal{L}$, un'unica decomposizione $\mathbf{t} = \mathbf{f} + \mathbf{v}$ può essere sempre trovata in modo tale che $\mathbf{v} \in \mathcal{L}$ e $\mathbf{f}$ si collochi nel dominio fondamentale $\mathcal{F}$ di $\mathbf{B}$. Questa proprietà fornisce l'idea che sta dietro alla risoluzione dell'apprCVP usata da questo algoritmo: identificare il dominio fondamentale (traslato) rispettivamente a $\mathbf{v} \in \mathcal{L}$, nel quale il vettore target $\mathbf{t}$ si trova. Sia $\mathcal{L}$ un reticolo con dimensione n generato da una base (buona) $\mathbf{B}$ e sia $\mathbf{t}$ un vettore tale che $\mathbf{t} \in \mathbb{R}^n$ e $\mathbf{t} \notin \mathcal{L}$. Dato che $\mathbf{B}$ è una matrice di rango massimo, è possibile calcolare:

$$\mathbf{x} = \mathbf{t}\mathbf{B}^{-1}$$

Da qui si applica la tecnica di arrotondamento, la quale è semplicemente:

$$\mathbf{w} = \sum_{i=1}^{n} \lfloor \mathbf{x}_i \rceil \mathbf{b}_i$$

con $\mathbf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n]$ e $\lfloor \mathbf{x} \rceil$ che significa prendere l'intero più vicino al numero reale $\mathbf{x}$. Questo algoritmo mira ad identificare il dominio fondamentale (traslato) che il vettore $\mathbf{t}$ localizza e la sua correttezza è strettamente legata alla forma geometrica del dominio fondamentale, è necessaria quindi una base di alta qualità al fine di avere risultati validi.

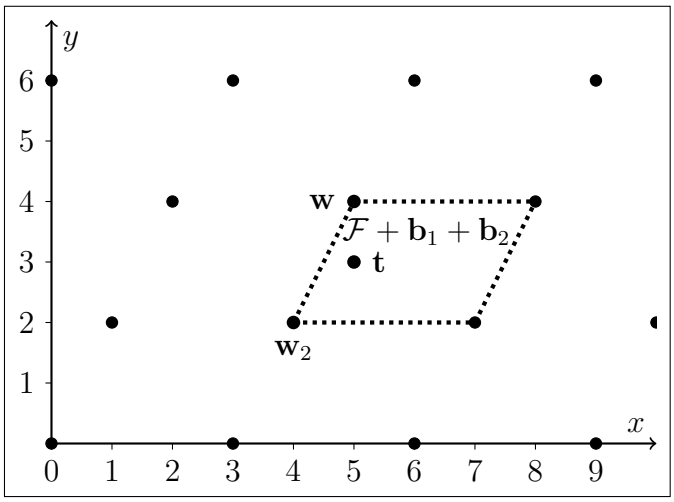


Figura 4: Risoluzione del CVP usando la tecnica di arrotondamento di Babai: $\mathbf{w}$ è il vertice del dominio fondamentale traslato localizzato da $\mathbf{t}$, quindi è soluzione per apprCVP.

Esempio 2.4.1. (Risoluzione del CVP usando la tecnica di arrotondamento di Babai)

Siano $\mathbf{R}$ e $\mathbf{B}$ le stesse basi definite nell'Esempio 2.1.1 e sia $\mathbf{t} \in \mathbb{R}^n, \mathbf{t} \notin \mathcal{L}$ con

$$\mathbf{t} = \begin{bmatrix} 5 & 3 \end{bmatrix}.$$

Si inizi con l'applicare l'algoritmo, dal primo passo si ottiene:

$$\mathbf{R}^{-1} = \begin{bmatrix} 0 & 0.33 \\ 0.5 & -0.16 \end{bmatrix} \quad \text{e} \quad \mathbf{x} = \mathbf{t}\mathbf{R}^{-1} = \begin{bmatrix} 1.5 & 1.16 \end{bmatrix}$$

si applichi ora la tecnica di arrotondamento a $\mathbf{x}$:

$$\lfloor \mathbf{x} \rceil = \begin{bmatrix} 2 & 1 \end{bmatrix}$$

si proceda infine con l'ottenere il risultato finale:

$$\mathbf{w} = \mathbf{x}\mathbf{R} = \begin{bmatrix} 5 & 4 \end{bmatrix}$$

che, come mostrato in Figura 4, è il vettore più vicino a $\mathbf{t}$ con $\|\mathbf{t} - \mathbf{w}\|_2 = 1$. Se si dovesse valutare la qualità della base, si otterrebbe che $\mathcal{H}(\mathbf{R}) = 0.94574$ e, grazie a tali proprietà ortogonali di $\mathbf{R}$, il dominio fondamentale derivante assume una forma geometrica tale per cui l'algoritmo è in grado di raggiungere facilmente la soluzione. Si riesegua ora l'algoritmo su $\mathbf{B}$. Calcolando $\mathcal{H}(\mathbf{B}) = 0.33231$ si scopre che $\mathbf{B}$ offre

una qualità molto più bassa rispetto a $\mathbf{R}$. Procedendo si ottiene che:

$$\mathbf{x}_2 = \mathbf{t}\mathbf{B}^{-1} = \begin{bmatrix} 1.5 & 1.16 \end{bmatrix} \quad \text{e quindi} \quad \lfloor \mathbf{x}_2 \rceil = \begin{bmatrix} 2 & 1 \end{bmatrix}.$$

Computando l'ultimo passaggio, il vettore risultante è:

$$\mathbf{w}_2 = \mathbf{x}_2\mathbf{B} = \begin{bmatrix} 4 & 2 \end{bmatrix}$$

il quale non è soluzione corretta all'apprCVP in quanto $\|\mathbf{t} - \mathbf{w}_2\|_2 = 1.41 > \|\mathbf{t} - \mathbf{w}\|_2$.

La principale differenza tra i due algoritmi di Babai è che il metodo del piano più vicino risulta essere più preciso in quanto i valori frazionari vengono arrotondati in maniera adattiva piuttosto che tutti insieme in un'unica volta. Inoltre l'utilizzo dell'aritmetica in virgola mobile, introdotta nella Sezione 2.3.4, consente di ottenere tempi di esecuzione ulteriormente più rapidi.

2.4.2 Tecnica di incorporamento

Babai, oltre alla presentazione dei due algoritmi precedentemente trattati, ha dimostrato anche quanto una base ridotta migliori l'approssimazione della soluzione ad apprCVP. In particolare, con una base LLL-ridotta, questo porta ad un fattore di approssimazione esponenziale per entrambi i suoi algoritmi. Nella pratica però, il metodo migliore per risolvere apprCVP, è la cosiddetta tecnica di incorporamento[3], tecnica euristica che si basa sul ridurre il CVP a un SVP.
Sia $\mathbf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n]$ la base di un reticolo $\mathcal{L}$ di dimensione n e sia $\mathbf{t} \in \mathbb{R}^n, \mathbf{t} \notin \mathcal{L}$. La tecnica di incorporamento impone la costruzione di un reticolo di dimensione $n+1$ con la seguente struttura:

$$\mathbf{M} = \begin{bmatrix} \cdots & \mathbf{b}_1 & \cdots & 0 \\ \cdots & \vdots & \cdots & 0 \\ \cdots & \mathbf{b}_n & \cdots & 0 \\ \cdots & \mathbf{t}_1 & \cdots & 1 \end{bmatrix}$$

Il nuovo reticolo $\mathcal{L}(\mathbf{M})$ è strutturato in modo tale da avere lo stesso determinante di $\mathcal{L}(\mathbf{B})$ e quasi la stessa dimensione, ci si può quindi aspettare che il vettore più corto di $\mathcal{L}(\mathbf{M})$ abbia quasi la stessa lunghezza di quello di $\mathcal{L}(\mathbf{B})$. Si assuma che $\mathbf{w} \in \mathcal{L}$ minimizzi la distanza per $\mathbf{t}$ e sia $\mathbf{u} = \mathbf{t} - \mathbf{w}$, allora il vettore

$$\mathbf{v} = \begin{bmatrix} \mathbf{u} & 1 \end{bmatrix}$$

appartiene a $\mathcal{L}(\mathbf{M})$ e, se dovesse anche essere il suo vettore più corto, si potrebbe risolvere l'apprCVP di $\mathcal{L}(\mathbf{B})$ determinando l'apprSVP di $\mathcal{L}(\mathbf{M})$. Per ottenere $\mathbf{v}$ è sufficiente ridurre $\mathbf{M}$ mediante algoritmi come LLL (o meglio BKZ) per poi ottenere

$\mathbf{w}$ calcolando $\mathbf{t} - \mathbf{u}$. È importante notare che il gap del reticolo di $\mathcal{L}(\mathbf{M})$ è approssimativamente il rapporto tra la lunghezza del vettore più corto di $\mathcal{L}(\mathbf{B})$ e la lunghezza di $\mathbf{u}$. Aumentare la lunghezza del vettore più corto di $\mathcal{L}(\mathbf{B})$ rende il gap del reticolo di $\mathcal{L}(\mathbf{M})$ più ampio, facilitando così la riduzione. Quando si discute del gap del reticolo in relazione a un'istanza del CVP, è importante chiarire che ci si riferisce in realtà al gap del reticolo dell'istanza SVP corrispondente. Questa istanza SVP viene creata attraverso una tecnica di embedding che trasforma l'istanza CVP originale in un'istanza SVP equivalente. Pertanto, il concetto di gap del reticolo, originariamente definito per SVP, viene esteso indirettamente alle istanze CVP attraverso questa trasformazione. Un problema nella pratica sta nella scelta di $\mathbf{t}$: teoricamente $\mathbf{t}$ può appartenere all'insieme $\mathbb{R}$, ma questo creerebbe problemi nella costruzione della nuova base $\mathbf{M}$ la quale non soddisferebbe più la proprietà dei coefficienti integrali che sta alla base della definizione di reticolo. Tale problema viene discusso e affrontato nell'attacco di Nguyen contro GGH presentato nella Sezione 3.2.2. Nel concreto si tenta di mantenere $\mathbf{t} \in \mathbb{Z}^n$ in modo da evitare problemi di questa natura.

Esempio 2.4.2. (Risoluzione del CVP usando la tecnica di incorporamento) Siano $\mathbf{R}$, $\mathbf{B}$ e $\mathbf{t}$ le stesse basi definite nell'Esempio 2.4.1. Seguendo quanto descritto nella tecnica di incorporamento, si costruisca la matrice

$$\mathbf{M} = \begin{bmatrix} \mathbf{r}_{0,0} & \mathbf{r}_{1,0} & 0 \\ \mathbf{r}_{0,1} & \mathbf{r}_{1,1} & 0 \\ \mathbf{t}_{0,0} & \mathbf{t}_{1,0} & 1 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 0 \\ 3 & 0 & 0 \\ 5 & 3 & 1 \end{bmatrix}$$

e la si riduca usando un algoritmo di riduzione, in questo caso LLL:

$$\mathbf{M}^* = \begin{bmatrix} 0 & -1 & 1 \\ 1 & 1 & 1 \\ 2 & -1 & -1 \end{bmatrix}.$$

Infine è necessario che si estraggano i primi n valori dal vettore riga più corto di $\mathbf{M}^*$ e sottrarli poi a $\mathbf{t}$. In questo caso il vettore più corto risulta essere il primo, quindi $\mathbf{u} = [0 \ -1]$. Completando questo passaggio si deduce che:

$$\mathbf{w} = \mathbf{t} - \mathbf{u} = \begin{bmatrix} 5 & 3 \end{bmatrix} - \begin{bmatrix} 0 & -1 \end{bmatrix} = \begin{bmatrix} 5 & 4 \end{bmatrix}$$

la quale è soluzione all' apprCVP con $\|\mathbf{t} - \mathbf{w}\|_2 = 1$. Utilizzando la base cattiva $\mathbf{B}$, invece, si otterrebbe:

$$\mathbf{M}_2 = \begin{bmatrix} 5 & 4 & 0 \\ -6 & -6 & 0 \\ 5 & 3 & 1 \end{bmatrix} \quad \text{con} \quad \mathbf{M}_2^* = \begin{bmatrix} 0 & -1 & 1 \\ -1 & -1 & -1 \\ 2 & -1 & -1 \end{bmatrix}.$$

Ed effettuando l'ultimo passaggio:

$$\mathbf{w}_2 = \mathbf{t} - \mathbf{u}_2 = \begin{bmatrix} 5 & 3 \end{bmatrix} - \begin{bmatrix} 0 & -1 \end{bmatrix} = \begin{bmatrix} 5 & 4 \end{bmatrix}$$

che è la stessa soluzione ottenuta con la base $\mathbf{R}$.

Grazie a questo esempio si può comprendere meglio l'efficacia nella pratica di questa tecnica: in comparazione con l'algoritmo di arrotondamento di Babai, nonostante la bassa qualità della seconda base, si è riusciti comunque a trovare una soluzione corretta.

Capitolo 3

Crittosistema a chiave pubblica GGH

3.1 Struttura e funzionamento di GGH

Nel 1996, Oded Goldreich, Shafi Goldwasser e Shai Halevi[4] hanno introdotto un nuovo sistema crittografico a chiave pubblica basato sulla difficoltà di risolvere CVP in reticoli di dimensioni elevate.
L'idea dietro GGH è la seguente: si supponga di avere un messaggio $\mathbf{m}$ codificato in un vettore appartenente ad un reticolo $\mathcal{L}$, un vettore target $\mathbf{t} \notin \mathcal{L}$ vicino ad $\mathbf{m}$ e due basi $\mathbf{R}$ e $\mathbf{B}$ entrambe generanti $\mathcal{L}$ e rappresentanti rispettivamente base privata e base pubblica. Siano $\mathbf{R}$ una base buona e $\mathbf{B}$ una base cattiva, allora, tramite l'utilizzo di uno degli algoritmi di risoluzione del CVP (Sezione 2.4), sarà possibile ritrovare il vettore più vicino a $\mathbf{t}$ (che risulterà essere $\mathbf{m}$) usando la base privata, ma non usando la base pubblica.
Più formalmente, GGH è definito da una funzione trapdoor (ovvero una funzione matematica che è facile da calcolare in una direzione, ma molto difficile da invertire senza dei dati segreti), la quale è composta da quattro funzioni probabilistiche di complessità polinomiale:

- Generate: Dato in input un intero positivo n vengono generate due basi $\mathbf{R}$ e $\mathbf{B}$ di rango massimo in $\mathbb{Z}^n$ e un numero positivo reale σ. Le basi $\mathbf{R}$ e $\mathbf{B}$ sono rappresentate da matrici $n \times n$ e sono rispettivamente denominate base privata e base pubblica. Sia $\mathbf{R}$ che $\mathbf{B}$ generano lo stesso reticolo $\mathcal{L}$ e, insieme a σ, danno origine a chiave privata e chiave pubblica. Per maggiori dettagli riguardo la generazione delle chiavi si veda la prossima sezione.

- Sample: Dati in input $\mathbf{B}, \sigma$ vengono originati i vettori $\mathbf{m}, \mathbf{e} \in \mathbb{R}^n$.
  Il vettore $\mathbf{m}$ viene scelto casualmente da un cubo in $\mathbb{Z}^n$ che sia sufficientemente

grande. Gli autori suggeriscono di scegliere in maniera casuale ogni valore di $\mathbf{m}$ uniformemente dall'intervallo $[-n^2, -n^2+1, \ldots, +n^2]$, sottolineando però che la scelta di n^2 è arbitraria e che non hanno prove di come essa possa influenzare la sicurezza del crittosistema stesso. Un intervallo sufficientemente grande che viene normalmente utilizzato è $[-128, 127]$.
Il vettore $\mathbf{e}$ invece, viene scelto casualmente in $\mathbb{R}^n$ in modo tale la media dei valori sia zero e la varianza sia σ^2. Il metodo più semplice per generare tale vettore è quello di scegliere ogni valore di $\mathbf{e}$ come $\pm\sigma$ con probabilità $\frac{1}{2}$. Questo vettore ha l'importante funzione di essere un errore che viene aggiunto al calcolo del testo cifrato per complicarne la decifratura.

- Evaluate: Dati in input $\mathbf{B}, \sigma, \mathbf{m}, \mathbf{e}$ si calcola $\mathbf{c} = \mathbf{mB} + \mathbf{e}$. Con questo calcolo si ottiene il messaggio cifrato che è rappresentato da $\mathbf{c}$.

- Invert: Dati in input $\mathbf{R}, \mathbf{c}$ si utilizza la tecnica di arrotondamento di Babai per invertire la funzione trapdoor e ricavare il messaggio originale.

3.1.1 Generazione delle chiavi

La generazione delle chiavi è un elemento cruciale in tutti i crittosistemi asimmetrici. In GGH, per costruire le chiavi, è indispensabile ottenere prima due basi: una pubblica e una privata. La sicurezza di questo crittosistema si basa sul fatto che la base pubblica non sia di qualità sufficientemente alta, in modo tale da impedire l'applicazione efficace di un algoritmo di risoluzione del CVP al testo cifrato, evitando così di recuperare il messaggio originale. È dunque fondamentale il modo in cui vengono generate la base privata e, soprattutto, la base pubblica, per garantire le caratteristiche necessarie a mantenere la sicurezza del crittosistema. In questa sezione verrà quindi analizzata la struttura della funzione Generate, la quale si occupa di quanto introdotto precedentemente.
Questa funzione prende come unico parametro in input la dimensione n dalla quale dipende la grandezza delle basi generate. In linea con quanto detto nella Sezione 2.2, più n cresce e più i problemi sui reticoli si fanno complessi, rendendo quindi più sicuro il crittosistema. A discapito di ciò però, man mano che la complessità aumenta, il tempo di esecuzione delle funzioni e lo spazio in bits delle basi diventano più onerosi. Gli autori di [4, sezione 3.3.1] a tal proposito ipotizzarono che, presi in considerazione gli algoritmi di riduzione disponibili al tempo, un n tra 150 e 200 fosse sufficiente, anche se ciò si rivelerà essere sbagliato. Dopo aver scelto un n adeguato, si procede con il generare la base privata $\mathbf{R}$ e, successivamente, decidere la distribuzione con la quale essa verrà originata. Due sono le proposte avanzate:

1. Generare una base $\mathbf{R}$ casuale: ogni elemento viene scelto in maniera casuale uniformemente nell'intervallo $[-l, \ldots, l]$ per qualche valore l. In [4] è stato provato

che la scelta di l non influenza particolarmente la qualità della base generata, per cui è stato scelto un l tra ± 4 al fine di semplificare alcune operazioni di calcolo.

2. Generare una base $\mathbf{R}$ rettangolare: si inizia con il moltiplicare la matrice identità $\mathbf{I}$ per qualche numero k ottenendo così $k\mathbf{I}$. Si genera poi una matrice $\mathbf{R}'$ casuale (punto 1.) per poi computare $\mathbf{R} = \mathbf{R}' + k\mathbf{I}$.

Come preannunciato, una volta generata la base privata $\mathbf{R}$, è necessario derivare la base pubblica rappresentata da un'altra base $\mathbf{B}$, tale che $\mathbf{R}$ e $\mathbf{B}$ generino lo stesso reticolo $\mathcal{L}$. Dato che ogni base di $\mathcal{L}(\mathbf{R})$ è ottenuta con $\mathbf{B} = \mathbf{UR}$ per qualche matrice unimodulare $\mathbf{U}$, allora ottenere $\mathbf{B}$ equivale ad ottenere una matrice unimodulare casuale. Anche in questo caso, due sono i metodi proposti per generare tali matrici:

1. Il primo metodo consiste nell'applicare una sequenza di operazioni elementari sulle colonne della matrice identità, mantenendo però gli uni sulla diagonale principale. Ad ogni step viene aggiunta alla i-esima colonna una combinazione lineare intera casuale delle altre colonne. I coefficienti della combinazione lineare sono scelti casualmente in $\{-1, 0, 1\}$ con un bias verso zero (probabilità $\frac{5}{7}$), in modo che i numeri non crescano troppo velocemente. Viene suggerito dagli autori stessi di eseguire l'algoritmo almeno due volte.

2. Il secondo metodo si basa sul generare delle matrici triangolari superiori ($\mathbf{S}$) e inferiori ($\mathbf{L}$) con ± 1 sulla diagonale principale. I restanti elementi della matrice che non sono zeri vengono scelti casualmente tra $\{-1, 0, 1\}$. In particolare sarà necessario moltiplicare $\mathbf{R}$ per almeno 4 paia di $\mathbf{SL}$ al fine di ottenere un $\mathbf{B}$ sufficientemente sicuro.

È stato provato dagli stessi autori che entrambi i metodi offrono lo stesso livello di sicurezza, anche se il secondo, in comparazione, genera matrici con numeri più grandi andando quindi a complicare i calcoli successivi.
Dopo aver generato due $\mathbf{R}$ e $\mathbf{B}$ con le qualità necessarie, non rimane altro che determinare σ. Questo valore è molto importante perchè esso aggiunge una complessità maggiore per quanto riguarda l'inversione della funzione trapdoor, diventando così un fattore di bilanciamento. Richiamando quanto definito nelle Sezioni 2.2 e 2.4.1, la tecnica di arrotondamento di Babai è una proposta per la risoluzione dell'apprCVP, il quale, ritorna un vettore più vicino che non sempre risulta essere la soluzione più corretta. Dato che GGH si basa su questo algoritmo per decifrare un messaggio, è possibile definire questo crittosistema come probabilistico: in certe situazioni neanche la base privata può essere usata per ritrovare il messaggio originale $\mathbf{m}$ e, viceversa, in altre situazioni, la base pubblica potrebbe essere usata per decifrare il messaggio. Per evitare questi casi è stato ideato il parametro σ, il quale, viene utilizzato per

generare il vettore di errore $\mathbf{e}$ che, una volta aggiunto a $\mathbf{c}$, complicherà ulteriormente l'inversione. L'idea è che la qualità di $\mathbf{B}$ sia sufficientemente bassa da non poter correggere l'errore, ma allo stesso tempo, permettere a $\mathbf{R}$ di essere in grado di farlo. È cruciale che σ non sia né troppo grande, altrimenti $\mathbf{R}$ non riuscirebbe a recuperare il messaggio, né troppo piccolo, per evitare che $\mathbf{B}$ possa riuscirci.

In [4, sezione 3.2] vengono proposte due metriche, ciascuna basata rispettivamente sulla norma L_1 e L_∞, per definire un limite a σ in maniera che non possa causare errori di inversione usando la base privata. La prima metrica è la più solida, poiché limita σ a un valore massimo che garantisce sempre il successo dell'inversione. La seconda, invece, restringe σ a un livello in cui la probabilità di errori d'inversione è molto bassa. In entrambi i casi, con dimensioni elevate, il valore massimo di σ si aggira intorno a 3, risultando in un valore standard che bilancia sicurezza e affidabilità. Ora che tutti i parametri sono stati determinati è possibile costruire le due chiavi:

- La chiave pubblica è definita semplicemente dalla coppia $(\mathbf{B}, \sigma)$
- La chiave privata, invece, è rappresentata semplicemente da $\mathbf{R}$. in modo da velocizzare la decifratura.

La decifratura avviene tramite la tecnica di arrotondamento di Babai spiegata in sezione 2.4.1:

$$\mathbf{m} = \lfloor \mathbf{c}\mathbf{R}^{-1} \rceil \mathbf{R}\mathbf{B}^{-1}$$

dove, per semplicità:

$$\mathbf{m} = \mathbf{w}\mathbf{B}^{-1} \quad \text{con} \quad \mathbf{w} = \lfloor \mathbf{c}\mathbf{R}^{-1} \rceil \mathbf{R}.$$

3.1.2 Esempio pratico

Prima di affrontare le varie tipologie di attacchi a GGH, viene mostrato un semplice esempio (a dimensione 3) di come due entità, rispettivamente Alice e Bob, possano utilizzare questo crittosistema per scambiare messaggi.

Esempio 3.1.1. (Esempio di funzionamento di GGH) Sia $\mathbf{R}$ la base privata di Alice definita come:

$$\mathbf{R} = \begin{bmatrix} 12 & -4 & -1 \\ 1 & 8 & -1 \\ -4 & 1 & 14 \end{bmatrix} \quad \text{con } \mathcal{H}(\mathbf{R}) = 0.96762$$

Alice procede col generare la sua base pubblica $\mathbf{B}$ moltiplicando $\mathbf{R}$ con una matrice unimodulare casuale $\mathbf{U}$:

$$\mathbf{U} = \begin{bmatrix} 12 & -3 & -1 \\ -3 & 1 & 1 \\ -14 & 3 & 0 \end{bmatrix} \quad \text{quindi } \mathbf{B} = \mathbf{U}\mathbf{R} = \begin{bmatrix} 145 & -73 & -23 \\ -39 & 21 & 16 \\ -165 & 80 & 11 \end{bmatrix}.$$

È possibile osservare come $\mathbf{B}$ abbia un rapporto di Hadamard molto basso, più precisamente $\mathcal{H}(\mathbf{B}) = 0.07403$. Infine, utilizzando $\sigma = 3$, Alice compone le sue due chiavi:

$$\mathbf{K}_{private} = \mathbf{R} \text{ e } \mathbf{K}_{public} = (\mathbf{B}, \sigma).$$

Bob decide di mandare un messaggio $\mathbf{m} = [-48\ 29 - 76]$ con vettore di errore $\mathbf{e} = [3\ 3\ 3]$. Utilizza quindi la chiave pubblica di Alice e ottiene il corrispondente testo cifrato:

$$\mathbf{c} = \begin{bmatrix} -48 & 29 & -76 \end{bmatrix} \begin{bmatrix} 145 & -73 & -23 \\ -39 & 21 & 16 \\ -165 & 80 & 11 \end{bmatrix} . + \begin{bmatrix} 3 & 3 & 3 \end{bmatrix} = \begin{bmatrix} 4452 & -1964 & 735 \end{bmatrix}.$$

Alice, una volta ricevuto il messaggio cifrato, è in grado di decifrarlo in maniera efficiente usando la sua chiave privata. Infatti, avendo a disposizione

$$\mathbf{R}^{-1} = \begin{bmatrix} \frac{113}{1363} & \frac{55}{1363} & \frac{12}{1363} \\ -\frac{10}{1363} & \frac{164}{1363} & \frac{11}{1363} \\ \frac{33}{1363} & \frac{4}{1363} & \frac{100}{1363} \end{bmatrix} \text{ e } \mathbf{B}^{-1} = \begin{bmatrix} -\frac{1049}{1363} & -\frac{1037}{1363} & -\frac{685}{1363} \\ -\frac{2211}{1363} & -\frac{2200}{1363} & -\frac{1423}{1363} \\ \frac{345}{1363} & \frac{445}{1363} & \frac{198}{1363} \end{bmatrix}$$

Alice, ottiene il messaggio originale calcolando:

$$\mathbf{x} = \lfloor \mathbf{c}\mathbf{R}^{-1} \rceil = \begin{bmatrix} 401 & -55 & 77 \end{bmatrix} \text{ e } \mathbf{m} = \mathbf{x}\mathbf{R}\mathbf{B}^{-1} = \begin{bmatrix} -48 & 29 & -76 \end{bmatrix}.$$

Si supponga ora che ci sia una terza persona, chiamata Eve, in ascolto nel canale di comunicazione tra Alice e Bob. Eve riesce ad ottenere la chiave pubblica di Alice e il messaggio cifrato inviato da Bob. Decide quindi di provare a decifrarlo usando la base pubblica invece della privata. Dato che non è in possesso della chiave privata di Alice, Eve tenterà la decifrazione usando solo la base pubblica $\mathbf{B}$.
Dato che $\mathbf{B}\mathbf{B}^{-1} = 1$, la tecnica di arrotondamento di Babai si semplifica alla seguente formula:

$$\mathbf{m}' = \lfloor \mathbf{c}\mathbf{B}^{-1} \rceil = \begin{bmatrix} -54 & 23 & -80 \end{bmatrix}$$

Il vettore $\mathbf{m}'$ ottenuto presenta evidenti similitudini con il messaggio originale $\mathbf{m}$, differenziandosi solo per alcune cifre. Sebbene in questo caso l'errore possa apparire quasi trascurabile è importante precisare che l'esempio è stato presentato in una dimensione molto bassa. Infatti la grandezza dell'errore è direttamente proporzionale all'aumentare della dimensione delle chiavi usate. Di conseguenza, il solo uso della base pubblica, non è sufficiente ad ottenere il messaggio originale.

3.2 Crittoanalisi di GGH

In questa sezione saranno esaminate le vulnerabilità di GGH e gli attacchi derivanti da esse. I principali attacchi a cui GGH è soggetto includono:

- Computazione di una chiave privata: eseguendo una riduzione della base pubblica $\mathbf{B}$ si tenta di ottenere una chiave privata $\mathbf{B}'$ di qualità pari o simile a quella originale.
- Risoluzione diretta del CVP: tentare di risolvere il CVP del testo cifrato $\mathbf{c}$ rispetto al reticolo definito dalla base pubblica $\mathbf{B}$.
- Attacco di Nguyen: sfruttando la particolare struttura del vettore di errore $\mathbf{e}$ adottata dagli autori del crittosistema, è possibile ricondursi ad un'istanza del CVP molto più semplice di quella proposta da GGH.
- Attacco basato su informazioni parziali: conoscendo sufficienti elementi del messaggio originale è possibile costruire un'istanza del CVP ancora più semplice di quella ottenuta tramite l'attacco di Nguyen.

3.2.1 Crittoanalisi originale

L'attacco più ovvio e semplice tra quelli proposti è la computazione di una chiave privata per invertire la funzione trapdoor. Uno studio dettagliato e combinato con esperimenti pratici ha portato però gli autori a considerarlo inefficace per una dimensione maggiore di 100. Un miglioramento dell'attacco appena descritto consiste nell'utilizzo di uno degli algoritmi per approssimare il CVP presentati nella Sezione 2.4.1, si rientra quindi nell'attacco basato su risoluzione diretta del CVP. Gli autori, basandosi su quanto descritto finora, hanno ipotizzato che, se l'algoritmo di riduzione utilizzato è LLL, il loro schema risulti sicuro per dimensioni superiori a 150 indipendentemente dal tipo di algoritmo scelto per risolvere il CVP. Tuttavia, poiché esistono algoritmi di riduzione migliori (Sezione 2.3.4), la loro conclusione è che la funzione trapdoor di GGH dovrebbe essere sicura per dimensioni comprese tra 250 e 300.
Di seguito viene presentato un esempio in dimensione 3 dell'attacco basato su risoluzione diretta del CVP. Per eseguire tale attacco sono stati utilizzati l'algoritmo LLL e la tecnica di incorporamento. Nonostante BKZ sia l'opzione più efficace, la bassa dimensionalità del problema rende i risultati ottenuti con LLL molto simili se non uguali. Pertanto, per semplicità, è stato scelto l'algoritmo LLL.

Esempio 3.2.1. (Esempio di risoluzione diretta del CVP tramite incorporamento) Siano $(\mathbf{B}, \sigma)$ e $\mathbf{c}$ rispettivamente chiave pubblica e testo cifrato utilizzati tra Alice e Bob nell'esempio 3.1.2. Supponiamo che Eve abbia intercettato il testo cifrato

e la chiave pubblica, e stia cercando di attaccare il crittosistema GGH risolvendo direttamente il CVP.
Decide di procedere tramite tecnica di incorporamento costruendo quindi la seguente matrice:

$$\mathbf{M} = \begin{bmatrix} 145 & -73 & -23 & 0 \\ -39 & 21 & 16 & 0 \\ -165 & 80 & 11 & 0 \\ 4452 & -1964 & 735 & 1 \end{bmatrix}.$$

Come secondo passaggio riduce $\mathbf{M}$ tramite LLL:

$$\mathbf{M}^* = \begin{bmatrix} -2 & 1 & -1 & -4 \\ 3 & 3 & 3 & 1 \\ 0 & 4 & -3 & 3 \\ 7 & -2 & -8 & -2 \end{bmatrix}.$$

Eve a questo punto, secondo quanto definito in Sezione 2.4.2, dovrebbe prelevare i primi n valori del vettore riga di $\mathbf{M}^*$ più corto. A causa della composizione del vettore di errore usato in GGH però la selezione del vettore da $\mathbf{M}^*$ risulta essere diversa. In particolare sapendo che $\sigma = 3$ Eve preleverà il vettore riga di forma $[\pm\sigma, \cdots \pm \sigma, 1]$, che non per forza è il vettore più corto di $\mathbf{M}^*$. In questo caso nella matrice è presente un vettore con tale forma, ovvero:

$$\begin{bmatrix} 3 & 3 & 3 & 1 \end{bmatrix} \text{ con conseguente } \mathbf{u} = \begin{bmatrix} 3 & 3 & 3 \end{bmatrix}.$$

Come si può notare $\mathbf{u}$ è uguale al vettore di errore $\mathbf{e}$ utilizzato da Bob nell'esempio 3.1.2, indice del corretto andamento dell'attacco. Come penultimo passaggio Eve calcola il vettore $\mathbf{w}$ più vicino a $\mathbf{c}$:

$$\mathbf{w} = \mathbf{c} - \mathbf{e} = \begin{bmatrix} 4452 & -1964 & 735 \end{bmatrix} - \begin{bmatrix} 3 & 3 & 3 \end{bmatrix} = \begin{bmatrix} 4449 & -1967 & 732 \end{bmatrix}$$

e ottiene infine il messaggio originale $\mathbf{m}$ tramite:

$$\mathbf{m} = \mathbf{w}\mathbf{B}^{-1} = \begin{bmatrix} -48 & 29 & -76 \end{bmatrix}.$$

Contromisure

La principale debolezza di GGH è intrinseca alla sua costruzione: il vettore di errore $\mathbf{e}$ è sempre notevolmente più corto dei vettori nel reticolo. Ciò favorisce quindi un gap di dimensione maggiore nel reticolo incorporato. Tale vulnerabilità viene sfruttata con successo dalla tecnica di incorporamento fino ad una certa dimensione, la quale si colloca tra 250 e 300. Non esiste un modo semplice per risolvere questo problema senza sconvolgere la struttura di GGH, è dunque noto che le istanze CVP derivanti

da tale schema risultano più facili da risolvere rispetto alle istanze CVP generali. L'unica soluzione è anche la più veloce e ovvia: aumentare la dimensione del reticolo oltre 300, in modo da evitare del tutto la possibilità di attacchi analoghi.

3.2.2 Attacco di Nguyen

Questo attacco prende il nome dal suo autore Phong Nguyen[11] il quale, nel 1999, scoprì una vulnerabilità nel crittosistema GGH che permise ad attacchi, come la risoluzione diretta del CVP, di funzionare a dimensioni ancora più elevate di quelle già precedentemente raggiunte. Nguyen notò che la particolare scelta di composizione del vettore di errore in GGH introdusse un "indizio" utilizzabile per ottenere informazioni relative al messaggio $\mathbf{m}$ e addirittura semplificare il CVP del relativo testo cifrato. Richiamando quanto detto nella sezione 3.1:

$$\mathbf{c} = \mathbf{mB} + \mathbf{e} \tag{1}$$

con $\mathbf{e} = \{\pm\sigma\}$. Data la speciale forma di $\mathbf{e}$ è possibile, tramite una precisa scelta di modulo, far scomparire il vettore di errore dall'equazione 1. Definendo quindi un vettore $\mathbf{s} = (\sigma, \ldots, \sigma) \in \mathbb{Z}^n$ e utilizzando come modulo 2σ si ottiene che:

$$\mathbf{e} + \mathbf{s} \equiv 0 \pmod{2\sigma}$$

e di conseguenza:

$$\mathbf{c} + \mathbf{s} \equiv \mathbf{mB} \pmod{2\sigma}.$$

Definendo $\mathbf{cs} = \mathbf{c} + \mathbf{s}$ si arriva ad un sistema modulare di tipo $\mathbf{y} \equiv \mathbf{Bx} \pmod{2\sigma}$ che come unica incognita ha $\mathbf{x}$ (ovvero $\mathbf{m}$). Questa tipologia di sistemi modulari si risolve banalmente quando la matrice $\mathbf{B}$ è invertibile modulo 2σ, permettendo di calcolare direttamente una soluzione unica. Tuttavia, se $\mathbf{B}$ non è invertibile, il processo di risoluzione diventa significativamente più complesso. In queste circostanze, si presentano diverse complicazioni: il sistema può ammettere soluzioni multiple, manca un approccio risolutivo diretto e i metodi di risoluzione devono essere adattati al modulo specifico del sistema in esame. Nguyen, in [11], stabilisce inizialmente che esiste una probabilità significativa che la matrice $\mathbf{B}$ sia invertibile modulo 2σ. Questa dimostrazione implica che in una porzione rilevante dei casi, il sistema modulare può essere risolto in modo diretto e semplice. Quando la matrice non è invertibile invece, Nguyen dimostra come il kernel (e quindi il numero delle soluzioni) sia generalmente molto piccolo. In particolare viene rilevato che solo una parte molto piccola delle matrici modulo 6 (che è il doppio del parametro $\sigma = 3$ suggerito) ha un kernel con più di 12 elementi.
Nguyen conclude quindi che, per la scelta suggerita di parametri (n, σ) e per qualsiasi

testo cifrato $\mathbf{c}$, il sistema lineare ha, molto probabilmente, pochissime soluzioni.
Si denoti con $\mathbf{m}_{2\sigma}$ il messaggio in chiaro modulo 2σ ottenuto risolvendo il precedente sistema modulare. Si supponga ora che $\mathbf{B}$ sia invertibile modulo 2σ, allora il sistema ha una sola soluzione $\mathbf{m}_{2\sigma} = (\mathbf{c} + \mathbf{s})\mathbf{B}^{-1}$. Sottraendo $\mathbf{m}_{2\sigma}\mathbf{B}$ in entrambe le parti dell'equazione 1 si consegue:

$$\mathbf{c} - \mathbf{m}_{2\sigma}\mathbf{B} = \mathbf{m}\mathbf{B} + \mathbf{e} - \mathbf{m}_{2\sigma}\mathbf{B}$$

e, raccogliendo $\mathbf{B}$ nella seconda parte dell'equazione, si ottiene quindi:

$$\mathbf{c} - \mathbf{m}_{2\sigma}\mathbf{B} = (\mathbf{m} - \mathbf{m}_{2\sigma})\mathbf{B} + \mathbf{e}. \tag{2}$$

Un'importante osservazione è che, essendo $\mathbf{m}_{2\sigma}$ congruente a $\mathbf{m}$ modulo 2σ, la differenza $(\mathbf{m}-\mathbf{m}_{2\sigma})$ risulta per definizione divisibile per 2σ. Questa proprietà consente di rappresentare tale differenza come il prodotto tra 2σ e un nuovo intero $\mathbf{m}'$, esprimendola nella forma $\mathbf{m} - \mathbf{m}_{2\sigma} = 2\sigma\mathbf{m}'$, dove $\mathbf{m}'$ costituisce il quoziente intero derivante da questa divisione. È possibile quindi riscrivere 2 come:

$$\mathbf{c} - \mathbf{m}_{2\sigma}\mathbf{B} = (2\sigma\mathbf{m}')\mathbf{B} + \mathbf{e}$$

e, dividendo per 2σ in entrambe le parti, si ottiene infine:

$$\frac{\mathbf{c} - \mathbf{m}_{2\sigma}\mathbf{B}}{2\sigma} = \mathbf{m}'\mathbf{B} + \frac{\mathbf{e}}{2\sigma}. \tag{3}$$

L'equazione 3 nella sua forma finale mostra una chiara divisione in due parti: la prima rappresenta un punto razionale con tutti gli elementi noti, permettendone così un calcolo diretto; la seconda mantiene la struttura della formula 1, differenziandosi unicamente per la presenza del messaggio $\mathbf{m}'$. Ne consegue quindi che tale equazione può essere letta come un CVP per il quale il vettore di errore $\frac{\mathbf{e}}{2\sigma} \in \{\pm\frac{1}{2}\}^n$ risulta essere molto più piccolo di quello proposto da GGH. Data la relazione tra i due errori, conseguente dal procedimento appena illustrato, se si è in grado di risolvere il ben più semplice CVP posto dall'equazione 3 allora è possibile risolvere anche il CVP originale. In altre parole Nguyen, grazie alla sua intuzione, è riuscito a ridurre l'istanza del CVP di GGH in una più semplice.
L'attacco di Nguyen può essere meglio descritto come una semplificazione del CVP di GGH, semplificazione che può essere sfruttata da algoritmi di risoluzione del CVP come la tecnica di incorporamento. Infatti, una volta risolto il CVP semplificato, si otterrà $\mathbf{m}'$ con il quale sarà possibile calcolare il messaggio originale attraverso:

$$\mathbf{m} = \mathbf{m}_{2\sigma} + 2\sigma\mathbf{m}'.$$

Successivamente alla pubblicazione di GGH nel 1997, vennero pubblicate delle "internet challenges": delle sfide lanciate dagli autori su internet al fine di testare quanto il loro schema fosse sicuro. Le sfide erano composte da 5 istanze di GGH delle quali si era a conoscenza solo del testo cifrato e della chiave pubblica. Ogni sfida era più difficile della precedente, spaziando più precisamente nelle seguenti dimensioni: 200, 250, 300, 350 e 400. Per validare il suo attacco, Nguyen, riuscì a recuperare il messaggio originale in tutte le sfide ad eccezione dell'ultima in dimensione 400, dove ottenne solo informazioni parziali. La sua strategia si articolò in due fasi: per dimensioni fino a 300, impiegò la tecnica di incorporamento con BKZ a blocchi di 20, mentre per le dimensioni superiori combinò lo stesso algoritmo di risoluzione per il CVP con una versione potata di BKZ a blocchi di 60. Per migliorare la stabilità, entrambe le varianti di BKZ furono implementate utilizzando l'aritmetica a virgola mobile. Un problema precedentemente introdotto nella Sezione 2.4.2 è l'uso di valori non interi nella costruzione della matrice secondo la tecnica di incorporamento. Infatti, in accordo con quanto ottenuto nell'equazione 3, $\mathbf{e} \in \{\pm\frac{1}{2}\}^n$ ne consegue quindi che la parte sinistra dell'equazione non sia più un vettore di soli elementi interi. Per risolvere tale problema Nguyen propose due soluzioni:

1. Moltiplicare per 2 l'equazione 3 ottenendo così $\mathbf{e} \in \{\pm 1\}^n$. Ciò però consegue che anche la base pubblica $\mathbf{B}$ sarà moltiplicata per 2 causando un aumento di complessità dei calcoli con reticoli di grandi dimensioni.

2. Aggiungere un vettore costante $\mathbf{s} = (\sigma, \ldots, \sigma)$ e successivamente scalare l'intero sistema per un fattore 2σ. Questa manipolazione matematica semplifica i calcoli, poiché il vettore di errore risultante contiene solo valori 0 o 1. Tuttavia, è importante notare che questa trasformazione comporta un leggero aumento della lunghezza prevista del vettore di errore. Per un esempio più dettagliato si veda [11, sezione 5]

Esempio 3.2.2. (Esempio dell'attacco di Nguyen a GGH tramite incorporamento) Siano $(\mathbf{B}, \sigma)$ e $\mathbf{c}$ rispettivamente chiave pubblica e testo cifrato utilizzati tra Alice e Bob nell'esempio 3.1.2. Si supponga che Eve abbia intercettato il testo cifrato e la chiave pubblica. Eve, venuta a conoscenza della scoperta di Nguyen, tenta così di decifrare il messaggio cifrato sfruttando tale informazione. Eve innanzitutto verifica se la base pubblica $\mathbf{B}$ sia invertibile modulo 2σ. Per fare ciò calcola $\det(\mathbf{B}) = 781$ e controlla se esso sia coprimo con $2\sigma = 6$. Scopre così che $\mathbf{B}$ è effettivamente invertibile, di conseguenza, il sistema modulare ha un'unica soluzione, che può essere determinata direttamente:

$$(\mathbf{c} + \mathbf{s})\mathbf{B}^{-1} \equiv \mathbf{m} \pmod{2\sigma}$$

$$\mathbf{m}_{2\sigma} = (\mathbf{c} + \mathbf{s})\mathbf{B}^{-1} \pmod{2\sigma}$$

$$\mathbf{m}_{2\sigma} = \left(\begin{bmatrix} 4452 & -1964 & 735 \end{bmatrix} + \begin{bmatrix} 3 & 3 & 3 \end{bmatrix} \right) \begin{bmatrix} 1 & 1 & 5 \\ 3 & 2 & 5 \\ 3 & 1 & 0 \end{bmatrix} \pmod{2\sigma} = \begin{bmatrix} 0 & 5 & 2 \end{bmatrix}.$$

Eve, ottenuto $\mathbf{m}_{2\sigma}$, procede con il calcolare il CVP semplificato tramite l'equazione 3. Dato che vuole utilizzare la tecnica di incorporamento per risolverlo nel passaggio successivo, moltiplica per 2 la frazione in modo da liberarsi di valori con la virgola. Tale moltiplicazione andrà poi riflessa su $\mathbf{B}$ anche nei passaggi successivi all'estrazione del vettore $\mathbf{e}$.

$$\mathbf{c}^* = 2\Big(\frac{\mathbf{c} - \mathbf{m}_{2\sigma}\mathbf{B}}{2\sigma}\Big) = \begin{bmatrix} 1659 & -743 & 211 \end{bmatrix}.$$

Ora che Eve ha ottenuto un'istanza semplificata del CVP originale, procede con gli stessi passaggi presentati nell'esempio 3.2.1, ma utilizzando il nuovo $\mathbf{c}^*$ invece che $\mathbf{c}$.

$$\mathbf{M} = \begin{bmatrix} 145 & -73 & -23 & 0 \\ -39 & 21 & 16 & 0 \\ -165 & 80 & 11 & 0 \\ 1659 & -743 & 211 & 1 \end{bmatrix}.$$

Nell'esempio 3.2.1 Eve usò LLL come algoritmo di riduzione. In questo attacco, per una maggiore sicurezza, decide di usare BKZ. Ottiene quindi la matrice:

$$\mathbf{M}^* = \begin{bmatrix} 1 & 1 & 1 & 1 \\ -1 & 6 & -3 & -2 \\ 3 & -1 & 6 & -6 \\ 9 & 0 & -6 & -4 \end{bmatrix} \quad \text{con} \quad \mathbf{e} = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}.$$

Dopo aver costruito il vettore $\mathbf{u}$, Eve è ora in grado di calcolare $\mathbf{m}'$, ricordandosì però che ora è necessario raddoppiare $\mathbf{B}$.

$$\mathbf{m}' = (\mathbf{c}^* - \mathbf{u})(2\mathbf{B})^{-1} = \begin{bmatrix} -8 & 4 & -13 \end{bmatrix}.$$

Eve, come ultimo passaggio, decifra il messaggio originale tramite:

$$\mathbf{m} = \mathbf{m}_{2\sigma} + 2\sigma\mathbf{m}' = \begin{bmatrix} 0 & 5 & 2 \end{bmatrix} + \left(6 \begin{bmatrix} -8 & 4 & -13 \end{bmatrix} \right) = \begin{bmatrix} -48 & 29 & -76 \end{bmatrix}.$$

Contromisure

Nguyen stesso propose delle modifiche allo schema originale in [11, sezione 7] per contrastare la vulnerabilità da lui scoperta e riparare lo schema GGH. La vulnerabilità principale del sistema è riconducibile alla particolare struttura del vettore di errore $\mathbf{e}$, come definito dagli autori in [4]. Per mitigare questo problema, un approccio

intuitivo consiste nel modificare l'intervallo dei possibili valori che le componenti del vettore possono assumere. Specificamente, Nguyen propose di adottare un intervallo più ampio $[-\sigma, \ldots, +\sigma]$, in sostituzione del più ristretto insieme $\pm\sigma$ originariamente utilizzato. Il nuovo vettore di errore risolve con successo la vulnerabilità sfruttata da Nguyen, ma rende il vettore stesso più corto, aumentando così il gap del reticolo incorporato e rendendo lo schema più vulnerabile ad attacchi basati su tecnica di incorporamento.

3.2.3 Attacco basato su informazioni parziali

Per quanto la vulnerabilità scoperta da Nguyen renda molto più facile l'attacco a GGH, essa non si rivelò sufficiente per dimensioni superiori a 400. A tal proposito nel 2010, Moon Sung Lee e Sang Geun Hahn[10], proposero un attacco in grado di superare la barriera dimensionale a cui i precedenti attacchi si fermarono. Mentre sia questo attacco che quello di Nguyen mirano a semplificare il CVP, essi differiscono nel metodo: Nguyen riduce la lunghezza del vettore di errore $\mathbf{e}$, mentre questo nuovo approccio aumenta la lunghezza del vettore più corto nel reticolo definito dalla base pubblica. Per fare ciò però è necessario che un numero k di valori del messaggio originale siano noti, tale conoscenza risulta essere possibile solo in alcuni casi. Il metodo su cui si basa l'attacco è il seguente.
Sia $\mathbf{m}^1$ il vettore composto dai primi k degli n valori di $\mathbf{m}$ (noti) e sia $\mathbf{m}^2$ il vettore composto dai restanti valori di $\mathbf{m}$. Similmente, sia $\mathbf{B}^1$ la matrice composta dalle prime $\mathbf{k}$ righe della base pubblica $\mathbf{B}$ e sia $\mathbf{B}^2$ la matrice composta dalle righe rimanenti di $\mathbf{B}$. Allora si ha che:

$$\mathbf{c} = \mathbf{mB} + \mathbf{e} = (\mathbf{m}^1\ \mathbf{m}^2)\begin{pmatrix}\mathbf{B}^1\\ \mathbf{B}^2\end{pmatrix} + \mathbf{e} = \mathbf{m}^1\mathbf{B}^1 + \mathbf{m}^2\mathbf{B}^2 + \mathbf{e}$$

da cui si deriva:

$$\mathbf{c} - \mathbf{m}^1\mathbf{B}^1 = \mathbf{m}^2\mathbf{B}^2 + \mathbf{e}. \tag{4}$$

Data l'assunzione iniziale, la prima componente dell'equazione 4 è conosciuta. La seconda componente, analogamente all'equazione 3 discussa in precedenza, può essere ricondotta a una versione semplificata del CVP originale. Tuttavia, in questo caso, il problema è definito su un reticolo $\mathcal{L}(\mathbf{B}_2)$ che è un sottoinsieme del reticolo originale $\mathcal{L}(\mathbf{B})$, ma distinto da esso. La risoluzione del nuovo CVP implica la risoluzione dell'istanza originale del problema. Tale affermazione sussiste in quanto il rango della matrice su cui viene risolta risulta essere $n - k$ e quindi molto più piccola dell'originale. Per validare il loro metodo, gli autori di [10] applicarono l'attacco alla sfida rimanente in dimensione 400, sfruttando anche la vulnerabilità scoperta da Nguyen. Il corretto funzionamento richiedeva la determinazione di k valori del

messaggio originale. Sapendo che il messaggio era composto da 400 numeri interi $\mathbf{m} = (\mathbf{m}_1, \mathbf{m}_2, \ldots, \mathbf{m}_{400})$, con $\mathbf{m}_i \in [-128, 127]$, e che $\mathbf{m}_1 \bmod 6 = 5$ grazie alle informazioni parziali rilevate da Nguyen, si dedusse l'esistenza di sole 43 possibilità per $\mathbf{m}^1 = (\mathbf{m}_1)$, ovvero $(-127), (-121), \ldots, (125)$. Questa deduzione permise di restringere significativamente lo spazio di ricerca per l'attacco, rendendo il metodo più efficace e praticabile. In conclusione quindi, gli autori di [10], riuscirono a superare la sfida (challenge) imposta dalla dimensione 400 avendo a dispozione un solo valore del messaggio originale.

Contromisure

Dato l'utilizzo della vulnerabilità scoperta da Nguyen al fine di indovinare valori del messaggio originale è logico pensare che la contromisura proposta da Nguyen sia sufficiente al fine di proteggere l'algoritmo da questo attacco. Questa affermazione risulta vera, ma solo se effettivamente non c'è una perdita di informazione relativa al messaggio originale. In [10] viene discussa la possibilità di un attacco alle challenge 350 e 400 senza utilzzare la tecnica di Nguyen. Gli autori scoprirono che per decifrare con successo il messaggio originale, era necessario fare supposizioni su 9 e 17 valori (k) per le rispettive sfide. Considerando che lo schema GGH originale definisce i valori del messaggio nell'intervallo $[-128, +127]$, indovinare un singolo elemento richiederebbe 2^8 tentativi, mentre per k elementi sarebbero necessari $2^{(8k)}$ tentativi. Risulta evidente che un approccio di questo tipo sarebbe impraticabile sia in termini di tempo che di risorse computazionali.

Capitolo 4

Migliorare GGH usando la Forma Normale di Hermite

Considerando i vari attacchi discussi nella Sezione 3.2 e le note vulnerabilità del crittosistema GGH, è evidente che, per un suo utilizzo sicuro, la dimensione delle chiavi deve essere almeno superiore a 400. Tuttavia, una tale dimensione comporta complessità spaziali e temporali tali da rendere il crittosistema poco competitivo rispetto ad altri attualmente in uso come RSA o DSS. Nel 2001, Daniele Micciancio [7] ha proposto una versione migliorata del crittosistema basata sulla forma normale di Hermite, nota come GGH-HNF. Questo schema mira ad aumentare sia le performance che la sicurezza in comparazione alle risorse necessarie rispetto alla versione originale di GGH.

4.1 Struttura e funzionamento di GGH-HNF

Sulla base di quanto precedentemente esposto nella Sezione 3.1, in GGH il messaggio originale $\mathbf{m}$ viene codificato in un vettore $\mathbf{x}$ appartenente al reticolo, e il testo cifrato risulta come $\mathbf{c} = \mathbf{xB} + \mathbf{e}$. Le ottimizzazioni sviluppate da Micciancio hanno portato a una modifica di questo approccio. Invece di generare in maniera casuale sia il vettore $\mathbf{x}$ che la base $\mathbf{B}$, Micciancio ha scelto di codificare il messaggio direttamente nel vettore di errore $\mathbf{e}$, procedendo con un approccio deterministico per la generazione dei precedentemente citati parametri. Questa scelta nasce dalla difficoltà nel generare vettori e basi random che abbiano una sicurezza intrinseca e dimostrabile. Questa difficoltà si ripercuote sulla sicurezza del crittosistema: $\mathbf{B}$ scelta casualmente rilascia spesso informazioni parziali relative a $\mathbf{R}$ permettendo così una facile riduzione di essa. Per superare questo problema, Micciancio decide di non generare più $\mathbf{B}$ tramite la costruzione di matrici unimodulari casuali moltiplicate per $\mathbf{R}$. Invece, opta per un approccio deterministico basato sulla forma normale di Hermite (HNF) di $\mathbf{R}$. La forma

normale di Hermite è una rappresentazione canonica e unica per una data matrice, ottenuta mediante operazioni elementari di riga e colonna. Essa presenta una struttura triangolare e garantisce che gli elementi sulla diagonale principale siano ordinati in modo decrescente. Poiché l'HNF è unica per ogni reticolo la chiave pubblica $\mathbf{B}$ non rivela informazioni sulla chiave privata $\mathbf{R}$, se non il reticolo $\mathcal{L}$ che genera. Inoltre, qualsiasi informazione su $\mathbf{R}$ che possa essere efficacemente calcolata da $\mathbf{B}$ può essere altrettanto efficacemente calcolata a partire da qualsiasi altra base $\mathbf{B}'$ che genera lo stesso reticolo $\mathcal{L}$. Questo perché $\mathbf{B} = \mathrm{HNF}(\mathbf{R}) = \mathrm{HNF}(\mathbf{B}')$.
Ottenuto $\mathbf{B}$ è necessario quindi calcolare un vettore $\mathbf{xB}$ appartenente al reticolo che verrà poi aggiunto a $\mathbf{e}$ come da equazione 1. L'idea migliore sarebbe scegliere il vettore in modo casuale e uniforme, ma questa scelta non è praticabile. Tuttavia, Micciancio in [7, sezione 4.1] dimostra che tale risultato può essere ottenuto mediante il semplice calcolo di $\mathbf{x} = \mathbf{e} \bmod \mathbf{B}$. Quindi, invece di aggiungere a $\mathbf{e}$ un vettore casuale $\mathbf{xB}$, si riduce $\mathbf{e}$ modulo la base pubblica. Data la particolare struttura della matrice $\mathbf{B}$ nella sua forma HNF, questo calcolo risulta particolarmente semplice da effettuare. Partendo da un vettore $\mathbf{x}$ inizialmente nullo, si può calcolare un valore di $\mathbf{x}$ alla volta, iniziando dall'ultimo componente $\mathbf{x}_n$, tramite la seguente formula:

$$\mathbf{x}_i = \left\lfloor \frac{\mathbf{e}_i - \sum_{j=i+1}^{n-1} \mathbf{B}_{j,i}\mathbf{x}_j}{\mathbf{B}_{i,i}} \right\rfloor \tag{5}$$

e ottenere infine:

$$\mathbf{c} = \mathbf{e} - \mathbf{xB}. \tag{6}$$

Come si può notare le due equazioni 1 e 6 sono diverse, ma come dimostrato in [7, sezione 4.3] esse garantiscono lo stesso livello di sicurezza.
Un ulteriore cambiamento, conseguente dalla scelta di Micciancio di usare $\mathbf{e}$ come vettore rappresentante il messaggio, è la totale mancanza di un fattore di bilanciamento, ruolo che nella versione originale del crittosistema veniva ricoperto da σ. Il processo di decifratura infatti, basato sulla tecnica di arrotondamento di Babai, rimane invariato. Pertanto, quanto detto in Sezione 3.1.1, è ancora vero anche per GGH-HNF: il crittosistema è probabilistico e necessità di un parametro per bilanciarne la probabilità di decifratura con chiave pubblica e privata. Nella versione originale di GGH, σ, veniva derivato direttamente dalla base privata e veniva utilizzato come parametro assoluto per la costruzione di $\mathbf{e}$.
In GGH-HNF invece, Micciancio, decide di creare un nuovo parametro ρ derivandolo sempre dalla base privata, ma con un approccio differente. La base privata $\mathbf{R}$ viene ortogonalizzata utilizzando l'algoritmo di Gram-Schmidt, producendo la base

ortogonale $\mathbf{R}^*$. Successivamente ρ è calcolato attraverso:

$$\rho = \frac{1}{2} min_i \|\mathbf{r}_i^*\|_2. \tag{7}$$

ρ rappresenta un raggio di correzione: se la lunghezza del vettore di errore è minore di questo raggio la decifratura avrà successo. Poiché la base privata è conosciuta esclusivamente dal destinatario, è essenziale che il parametro ρ sia incluso nella chiave pubblica, insieme alla base pubblica $\mathbf{B}$. Questa inclusione è fondamentale affinché il mittente possa generare messaggi appropriati, codificandoli nel vettore di errore $\mathbf{e}$. In questo modo, il mittente può assicurarsi che i messaggi cifrati siano compatibili con i parametri di decifratura del destinatario, garantendo che possano essere decifrati correttamente utilizzando la base privata del ricevente.
Un'ultima modifica proposta riguarda la generazione di $\mathbf{R}$. Mentre GGH optava per la creazione di una matrice rettangolare successivamente moltiplicata per una matrice casuale, Micciancio suggerisce un metodo diverso basato sui suoi esperimenti. Il nuovo approccio consiste nel generare direttamente una matrice casuale i cui elementi sono interi compresi nell'intervallo $[-n, \ldots, n]$. A questa matrice viene poi applicata una riduzione LLL. Gli esperimenti provarono che questo metodo produce basi con un ρ sufficientemente grande, più precisamente $\rho = \frac{n}{2}$.

4.1.1 Esempio pratico

Esempio 4.1.1. (Esempio di funzionamento di GGH) Sia $\mathbf{R}$ la base privata di Alice definita nell'esempio 3.1.2. Sia $\mathbf{B}$ la forma normale di Hermite di $\mathbf{R}$:

$$\mathbf{B} = \mathrm{HNF}(\mathbf{R}) = \begin{bmatrix} 1 & 0 & 327 \\ 0 & 1 & 1322 \\ 0 & 0 & 1363 \end{bmatrix}.$$

Se si dovesse calcolare il rapporto di Hadamard di $\mathbf{B}$ si otterrebbe che $\mathcal{H}(\mathbf{B}) = 0.01322$ che è ancora minore di quello relativo alla base pubblica dell'esempio 3.1.2, facendo intuire quanto l'HNF sia utile per la generazione di basi reticolari di bassa qualità.
Alice procede col calcolare il ρ della sua chiave privata ottenendo $\rho = 3.99242$ e conclude con la generazione della sue sue due chiavi:

$$\mathbf{K}_{private} = \mathbf{R} \quad \text{e} \quad \mathbf{K}_{public} = (\mathbf{B}, \rho).$$

Bob vuole ora mandare un messaggio ad Alice. Inizia con il selezionare un vettore $\mathbf{e}$ la cui lunghezza sia minore del ρ di Alice:

$$\mathbf{e} = \begin{bmatrix} 1 & 1 & 2 \end{bmatrix} \quad \text{con} \quad \|\mathbf{e}\|_2 = 2.44948.$$

Una volta ottenuto $\mathbf{e}$, Bob, calcola il testo cifrato attraverso le equazioni 5 e 6:

$$\mathbf{c} = \mathbf{e} - \mathbf{xB} = \begin{bmatrix} 1 & 1 & 2 \end{bmatrix} - \begin{bmatrix} 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 327 \\ 0 & 1 & 1322 \\ 0 & 0 & 1363 \end{bmatrix} = \begin{bmatrix} 0 & 0 & -1647 \end{bmatrix}.$$

Alice, una volta ricevuto $\mathbf{c}$, utilizza la sua chiave privata per decifrarlo attraverso la tecnica di arrotondamento di Babai:

$$\mathbf{x} = \lfloor \mathbf{cR}^{-1} \rceil = \begin{bmatrix} -40 & -5 & -121 \end{bmatrix} \text{ ed } \mathbf{e} = \mathbf{c} - \mathbf{xR} = \begin{bmatrix} 1 & 1 & 2 \end{bmatrix}.$$

Si supponga ora che Eve, una volta intercettato il testo cifrato e la chiave pubblica, tenti di ottenere il messaggio originale usando solo la base $\mathbf{B}$:

$$\mathbf{e}' = \mathbf{c} - (\lfloor \mathbf{cB}^{-1} \rceil \mathbf{B}) = \begin{bmatrix} 0 & 0 & -284 \end{bmatrix}.$$

È possibile osservare una significativa differenza tra $\mathbf{e}$ ed $\mathbf{e}'$, dimostrando ancora una volta l'aumento di sicurezza apportato dall'uso della forma normale di Hermite. Inoltre è possibile verificare che $\mathbf{B}$ non è in grado di correggere l'errore $\mathbf{e}$ attraverso il calcolo del suo ρ, il quale, è pari a 0.50042.
Mentre nell'esempio precedente (esempio 3.1.2) il messaggio decifrato $\mathbf{m}'$ mostrava poche cifre di distanza dal messaggio originale $\mathbf{m}$, in questo caso la situazione cambia notevolmente, presentando differenze molto più marcate.

4.2 Limiti pratici di GGH-HNF

GGH-HNF riesce a risolvere i problemi di GGH con successo, diventando una variante migliorata a tutti gli effetti. Nelle conclusioni di [7], Micciancio suggerisce che una dimensione di 500 potrebbe offrire un livello di sicurezza adeguato, mantenendo al contempo una dimensione delle chiavi accettabile grazie all'impiego della forma normale di Hermite. Sfortunatamente però le supposizioni di Micciancio si sono rivelate troppo ottimistiche.
Nel Gennaio del 2004 Christoph Ludwig stilò un report tecnico [6] nel quale crittoanlizzò GGH-HNF e ne testò i suoi limiti pratici.

Generazione delle chiavi

Una serie di esperimenti vennero condotti sulla generazione delle chiavi di GGH-HNF. Il lavoro si è concentrato su diverse dimensioni dei reticoli, fino a 475, con un caso speciale in dimensione 800. Per le chiavi private, il processo più impegnativo è stato la riduzione LLL delle basi scelte casualmente. Secondo gli esperimenti di Ludwig

questo ha richiesto fino a 58 minuti nelle dimensioni più alte. Molto più pesante invece il dato riguardante la generazione della chiave pubblica: il miglior algoritmo a disposizione impiegò 4 ore. Per quanto riguarda il caso in dimensione 800 i tempi rilevati furono di 4 ore e mezza per la chiave privata e 46 ore per quella pubblica.

Cifratura e decifratura

Come descritto precedentemente, la particolare struttura della forma normale di Hermite consente una cifratura molto veloce. Ciò venne confermato dai test di Ludwig i quali impiegarono in media solo 0.29 secondi in dimensione 800. Le cose cambiano drasticamente con la decifratura: a causa dell'ortogonalizzazione Gram-Schmidt e della precisione richiesta, lo spazio occupato e il tempo richiesto per i calcoli cresce a livelli non accettabili. Gli esperimenti richiedettero 40 minuti per ortogonalizzare e rispettivamente 13 e 73 minuti per decifrare in dimensione 475 e 800. È però importante precisare che Ludwig utilizzò il metodo del piano più vicino di Babai che restituisce una soluzione più precisa, ma contemporaneamente richiede più tempo per trovarla a causa della sua complessità computazionale maggiore.

Attacchi a GGH-HNF

Gli attacchi a GGH-HNF furono condotti su reticoli di dimensioni fino a 280, con vettori di errore di lunghezza variabile tra il 10% e il 100% del ρ. Gli attacchi furono configurati con la tecnica di incorporamento, usando LLL, BKZ-20 e BKZ-60 potato. LLL si dimostrò efficace in dimensione 280 con vettori di errore corti, ma inefficiente per dimensioni da 180 in su con vettori più lunghi. L'aumento di lunghezza dei vettori in dimensioni più alte richiese necessario l'impiego di algoritmi più avanzati. Inizialmente, si tentò con BKZ-20, ma quando questo si rivelò insufficiente, si passò al più potente BKZ-60 potato. Ludwig, estrapolando dai dati sperimentali, stimò l'efficacia degli attacchi su dimensioni maggiori che non erano state oggetto di verifica, offrendo una prospettiva sulla sicurezza futura del sistema. Considerando scenari di complessità esponenziale e subesponenziale, suggerì che per garantire la sicurezza del GGH-HNF sarebbero necessarie dimensioni del reticolo di almeno 800, ben oltre le stime iniziali di Micciancio di 500. Questi risultati portarono alla conclusione che le scarse prestazioni di GGH-HNF su alte dimensioni lo rendessero impraticabile, specialmente considerando la necessità di dimensioni superiori a 800. Tuttavia, è cruciale notare che tali dati sono ormai superati dal progresso tecnologico. Come dimostrano i risultati sperimentali in Sezione 6.3, gli algoritmi moderni consentono di generare e decifrare chiavi con risorse notevolmente inferiori rispetto a quelle richieste all'epoca della pubblicazione dell'algoritmo, circa vent'anni fa.

Capitolo 5

Implementazione

Nel seguente capitolo vengono presentate e discusse le implementazioni dei crittosistemi GGH e GGH-HNF in linguaggio Python. Inoltre, vengono illustrati alcuni strumenti fondamentali per la crittografia basata sui reticoli, comunemente impiegati da entrambi i sistemi crittografici. L'attenzione si concentra sulle tecniche di programmazione, i moduli e i metodi utilizzati per implementare questi algoritmi crittografici, elementi necessari per grarantire l'efficienza e la precisione delle operazioni richieste. La prima sezione è di carattere preliminare: illustra le tecnologie impiegate, le motivazioni e le conseguenze derivanti da esse, nonché la struttura del progetto organizzato in un pacchetto Python di tre moduli. Le sezioni successive invece, introdurranno più nello specifico ogni singolo modulo descrivendone problemi, soluzioni e funzionalità.

5.1 Tecnologie adottate e motivazioni

La prima scelta che deve essere presa prima di iniziare a strutturare le implementazioni è il linguaggio di programmazione. Questo gioca un ruolo fondamentale sia per quanto riguarda l'efficienza del codice e sia per quanto riguarda la sua usabilità sui diversi sistemi operativi. Spesso, quando si tratta di operazioni matematiche complesse, la scelta di linguaggi di programmazione ad alte prestazioni è fondamentale. I reticoli, ed in particolare i problemi legati ad essi, richiedono calcoli precisi e veloci, talvolta con numeri molto grandi o molto piccoli. Questo compito viene spesso affidato al linguaggio di programmazione C, alle sue varianti come C++ o a linguaggi specializzati in calcoli matematici come Mathematica.
Alcuni esempi di librerie C specifiche utilizzate possono essere osservati direttamente negli studi svolti dagli autori citati nei precedenti capitoli:

- [7, 6, 10], dove viene utilizzata la Number Theory Library (NTL).

- [4, 11], in cui è stata impiegata la libreria LiDiA.

La scelta di C risulta quindi essere molto popolare e giustificata dalla sua efficienza. È però necessario precisare che, dati gli anni di pubblicazione degli studi originali, molti dei linguaggi di programmazione attualmente in circolazione non potevano essere presi in considerazione, poichè semplicemente non esistenti o non sufficientemente maturi. Alcuni dei più recenti linguaggi popolari al giorno d'oggi riescono non solo a pareggiare o superare la velocità di C, ma risolvono anche altri suoi problemi intrinseci, come il non supportare più piattaforme e non disporre di sistemi di sicurezza a protezione dei dati in memoria.
Alcuni esempi includono Rust, nato nel 2015, e Python, introdotto nel 1991, molto prima che venisse pubblicato GGH e le sue varianti. Nello specifico, quest'ultimo, ha attraversato un'evoluzione significativa nel corso degli anni, diventando oggi il linguaggio di programmazione più richiesto e utilizzato dalle aziende. In particolare Python gode delle seguenti proprietà:

- Leggibilità e semplicità del codice: Python utilizza una sintassi chiara e concisa che rende il codice facile da leggere e mantenere riducendo il rischio di errori.
- Compatibilità multipiattaforma: Python è un linguaggio interpretato e multipiattaforma, il che significa che il codice può essere eseguito su diversi sistemi operativi (Windows, macOS, Linux) senza richiedere modifiche sostanziali.
- Gestione automatica della memoria: Python gestisce automaticamente la memoria attraverso un garbage collector, riducendo il rischio di memory leaks e semplificando lo sviluppo rispetto a linguaggi come C, dove viceversa è richiesta la gestione manuale della memoria.
- Sicurezza: Python offre protezioni intrinseche contro problemi di sicurezza comuni, come buffer overflow, che sono invece frequenti in linguaggi a basso livello come C.
- Estendibilità: Python può essere facilmente esteso con moduli scritti in C, C++ o Cython per migliorare le prestazioni e l'efficienza.
- Gestione di numeri con precisione illimitata: Python è dotato di funzionalità native per manipolare interi di qualsiasi grandezza senza restrizioni. Inoltre, tramite il modulo `Decimal`, offre la possibilità di operare con numeri decimali a precisione arbitraria.

Alla luce di queste caratteristiche, Python si rivela un'opzione ottimale per lo sviluppo dei progetti richiesti. Sebbene sia generalmente noto come meno veloce rispetto a linguaggi come C, questa potenziale limitazione può essere efficacemente compensata.

La capacità di Python di integrarsi con moduli scritti in linguaggi più efficienti dal punto di vista delle prestazioni offre un modo pratico per migliorare la velocità di esecuzione dove necessario, combinando così la facilità d'uso di Python con l'efficienza di linguaggi di più basso livello. Sfruttando la menzionata estendibilità di Python, si è affrontata la sfida delle prestazioni in operazioni matriciali complesse, come l'inversione, che risultano particolarmente onerose per dimensioni elevate (come evidenziato dai dati nella Sezione 4.2). In Python, la scelta più ovvia quando si deve operare con grandi numeri e con alta velocità, ricade spesso sulla libreria Numpy. Questa preferenza è dovuta principalmente alle prestazioni superiori di Numpy nell'elaborazione di array multidimensionali e alla sua vasta gamma di funzioni matematiche ottimizzate, che la rendono particolarmente efficiente per calcoli scientifici e numerici su larga scala. Tuttavia, Numpy utilizza tipi di dati a precisione fissa, con un massimo di 64 bit per i sistemi operativi più comuni, che possono risultare insufficienti per calcoli che richiedono una precisione estremamente elevata o che coinvolgono numeri al di fuori del range rappresentabile con 64 bit. Questa limitazione può portare a errori di arrotondamento o overflow in operazioni matematiche complesse o con numeri estremamente grandi, resta comunque possibile e utile l'uso di questo modulo, ponendo però molta attenzione al tipo di calcoli affidatogli. Per ottimizzare queste operazioni mantenendo al contempo il vantaggio della precisione arbitraria di Python, si è scelto di integrare una libreria specializzata scritta in un linguaggio ad alte prestazioni. La Fast Library for Number Theory [18] (FLINT) è stata selezionata per questo scopo critico, offrendo un equilibrio ideale tra velocità e precisione. La possibilità di integrare FLINT tramite PyPI e richiamare direttamente le sue funzioni da Python, consente di incorporare facilmente la libreria nel progetto, combinando così l'efficienza computazionale con la flessibilità e la leggibilità del codice Python. Tuttavia, a causa delle limitate funzionalità offerte dall'integrazione di FLINT e per semplificare il codice, è stato necessario utilizzare altre librerie esterne di supporto. I principali moduli e librerie utilizzati nel progetto, oltre a FLINT, sono:

- `Numpy`: Modulo esterno che, come già introdotto, si specializza in calcoli numerici e matematici con particolare attenzione all'efficienza. Il suo utilizzo può diventare rischioso a causa della sua limitata precisione a "soli" 64 bit, per questo il suo utilizzo uso va confinato a contesti verificati dove la precisione limitata non compromette l'accuratezza dei risultati finali.

- `Sympy`: Modulo esterno dedicato ai calcoli simbolici a precisione arbitraria. Utilizzato come supporto a FLINT grazie alle sue numerose funzioni già pronte e il mantenimento della precisione nei calcoli. A causa delle sue basse performance, il suo utilizzo è strettamente limitato a funzioni di bassa complessità.

- `Decimal` e `Fraction`: Moduli nativi Python per la gestione di numeri decimali e frazionari a precisione arbitraria. La loro funzione è quella di sostituire FLINT

ove esso non può fornire una soluzione diretta o dove Sympy risulta troppo lento.

- `Random`: Modulo Python nativo che si occupa di generazione pseudo-casuale di dati. Utilizzato solo ed esclusivamente per la generazione di basi e vettori randomici.

Questi moduli non rappresentano l'interezza delle librerie utilizzate nel progetto, ma solo le principali, utilizzate in vari punti del codice di ciascuna implementazione e che giocano un ruolo importante nel progetto.
Per gli algoritmi complessi e ad alta precisione, come quelli di riduzione reticolare, si è optato, ove possibile, per l'utilizzo di implementazioni efficienti e testate provenienti da librerie esterne, anziché riscriverli in Python. Nello specifico:

- L'algoritmo LLL è stato integrato direttamente tramite una sua versione presente in FLINT.
- L'algoritmo BKZ è stato incorporato mediante la libreria open-source FPLLL [19], che offre implementazioni in virgola mobile di vari algoritmi di riduzione per reticoli, tra cui anche LLL.
- Per l'algoritmo di Gram-Schmidt, non avendo trovato un'implementazione rapida e facilmente integrabile, si è proceduto a una riscrittura in Python attraverso l'uso di `Numpy`.

È importante precisare che FPLLL è disponibile solo per il sistema operativo Linux, impedendo quindi di poter funzionare su più piattaforme, punto importante del progetto. Il problema è stato superato su Windows grazie all'introduzione del Windows Subsystem for Linux (WSL). Questa tecnologia consente di eseguire un ambiente Linux virtualizzato all'interno di Windows, integrandosi con il sistema operativo e permettendo di richiamare funzionalità Linux direttamente. È infine opportuno notare che il sistema di archiviazione su file adottato, ovvero su semplici file di testo, non rappresenta la soluzione ottimale in termini di efficienza dello spazio. Come evidenziato nella Tabella 2 della Sezione 6.1, questo approccio genera file di dimensioni considerevoli per gli standard crittografici. Le chiavi e il testo cifrato potrebbero essere deserializzate attraverso l'uso del modulo `Pickle` e poi compresse usando algoritmi come bz2 o lzma, ottenendo una riduzione significativa delle dimensioni dei file. Nonostante ciò, la scelta di utilizzare il formato `txt` anziché alternative come `Pickle` è stata dettata da considerazioni di sicurezza e portabilità. Il formato `txt` garantisce una maggiore leggibilità e modificabilità dei dati, riducendo i rischi associati alla deserializzazione di oggetti potenzialmente dannosi, un problema noto con `Pickle`. Inoltre, il formato testuale assicura una migliore compatibilità tra diversità

sistemi operativi, facilitando la condivisione e il trasferimento sicuro dei dati da e verso WSL, senza la necessità di gestire la decompressione o affrontare potenziali problemi di compatibilità legati alle versioni degli algoritmi di compressione.

5.1.1 Struttura del progetto

Essendo Python il linguaggio scelto è conseguente che la struttura del progetto più corretta si configuri come un pacchetto Python. Un pacchetto Python è una struttura organizzativa che racchiude moduli e sottopacchetti correlati, presentandosi come una directory nel filesystem. Questa directory contiene file Python (.py) che fungono da moduli, un file speciale chiamato __init__.py che identifica la directory come pacchetto, e può includere altre subdirectory rappresentanti sottopacchetti. Il file __init__.py, pur potendo essere vuoto, è fondamentale per segnalare a Python che la directory deve essere trattata come un pacchetto, consentendo così un'importazione e un utilizzo strutturato dei componenti software all'interno del progetto.

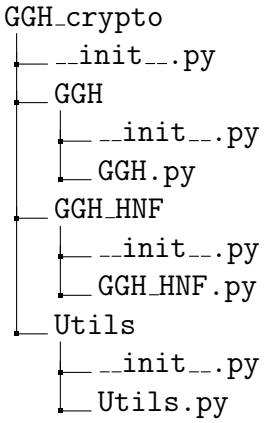


Figura 5: Struttura logica del pacchetto Python.

Come osservabile in Figura 5, il progetto è strutturato da:

- `GGH_crypto`: Rappresentante il pacchetto principale contenente tutte le implementazioni. Esso è il modulo primario dal quale tutte le funzionalità dei sottopacchetti al suo interno possono essere chiamate e usate.
- `GGH`: Sottopacchetto contenente l'implementazione di GGH.
- `GGH_HNF`: Sottopacchetto contenente l'implementazione di GGH-HNF.
- `Utils`: Sottopacchetto contenente degli algoritmi relativi ai reticoli e dei metodi in comune utilizzati dalle implementazioni dei due crittosistemi.

La scelta di organizzare il progetto in tre sottopacchetti distinti, ciascuno con il proprio file di inizializzazione, invece di utilizzare tre moduli nel pacchetto principale `GGH_crypto`, potrebbe essere considerata non ottimale. Tuttavia, questa organizzazione presenta vantaggi significativi: consente una chiara separazione logica dei componenti, offre maggiore flessibilità e semplifica la gestione delle importazioni tra i vari elementi. Di contro, è innegabile che questa configurazione comporti una maggiore complessità nella lettura del progetto a causa del numero più elevato di files.

5.1.2 Integrazione e gestione di FLINT

Come introdotto nella Sezione 5.1, FLINT è un modulo cardine nell'implementazione proposta, che si rende responsabile della gestione dei calcoli ad alta precisione ed efficienza. FLINT riesce nel suo intento grazie a dei tipi di dati specializzati, progettati appositamente per gestire numeri di precisione arbitraria. Nello specifico, tra i diversi tipi proposti da FLINT, due sono quelli utilizzati nel progetto:

- `fmpz` (Fast Multiple Precision Integers)
- `fmpq` (Fast Multiple Precision Rationals)

e le loro varianti per il calcolo matriciale: `fmpz_mat` e `fmpq_mat`. Python, essendo un linguaggio a tipizzazione dinamica, non dispone nativamente di tipi di dati statici come quelli offerti da FLINT su C. Tuttavia, per sfruttare le potenti funzionalità di FLINT in Python, sono stati sviluppati dei bindings ovvero delle interfacce che consentono l'interoperabilità tra i due linguaggi. Grazie a questi bindings, è possibile creare oggetti FLINT direttamente in Python:

1. Tipi interi:
    - Gli oggetti `fmpz` possono essere istanziati a partire da valori `Integer`.
2. Tipi razionali:
    - Gli oggetti `fmpq` possono essere creati fornendo due valori `Integer` rappresentanti rispettivamente numeratore e denominatore.
3. Matrici:
    - `fmpz_mat` (matrici di interi) possono essere generate da una lista di liste contenenti valori `fmpz` o `Integer`.
    - `fmpq_mat` (matrici di razionali) possono essere create da da una lista di liste contenenti valori `fmpz`, `Integer`, o `fmpq`

Sebbene questo sistema sia in grado di fornire alta efficienza e precisione, esso comporta una maggiore complicazione qualora FLINT non sia in grado di fornire una soluzione integrata e diretta, come nei casi della radice quadrata e della potenza. Per risolvere il problema sono state quindi implementate delle conversioni attraverso codice statico o funzioni. Tali conversioni utilizzano funzionalità di moduli più versatili come `Sympy` o `Decimal`, ma a scapito dell'efficienza. Sebbene questi moduli offrano una maggiore flessibilità, essi non sono ottimizzati per operazioni aritmetiche a basso livello come lo è FLINT. Ad esempio, Sympy è una libreria simbolica scritta in Python, e per questo motivo risulta più lenta, poiché oltre ai calcoli deve gestire anche l'interpretazione simbolica delle espressioni.
Un esempio di conversione da un oggetto `fmpq_mat` a `Decimal` è illustrato in Figura 6. Dopo aver verificato che l'oggetto in questione sia di tipo `fmpq_mat`, si estrae il numeratore e il denominatore, li si converte in interi e infine si effettua la conversione in `Decimal`.

```
        def vector_l2_norm(row):
            if isinstance(row, fmpz_mat):
                row = fmpq_mat(row)

            getcontext().prec = 50

            return Decimal(
                sum(
                    (Decimal(int(x.numer())) /
                    Decimal(int(x.denom()))) ** 2
                    for x in row)
                ).sqrt()
```

Figura 6: Funzione del modulo `Utils` contenente una conversione da `fmpq_mat` a `Decimal`.

5.2 Modulo GGH

Il modulo GGH contenente l'implementazione del relativo crittosistema è caratterizzato da una classe `GGHCryptosystem` che al suo interno contiene tutte le funzionalità necessarie al suo corretto andamento. Istanziando la classe è possibile passare diversi parametri al fine di poter modificare a piacimento determinati valori che verranno usati dal crittosistema in fase di cifratura e decifratura. Tutti i parametri hanno un valore predefinito impostato a `None`, con l'eccezione del parametro relativo alla dimensione e dei parametri booleani. Se l'utente non specifica valori diversi per i

parametri nulli, il modulo genererà automaticamente valori casuali per ciascuno di essi. I parametri consentiti sono i seguenti:

- `dimension` (`fmpz_mat`): Dimensione delle basi e dei vettori generati dal crittosistema.
- `private_basis` e `public_basis` (`fmpz_mat`, default: `None`): Rispettivamente base privata e base pubblica con le quali il crittosistema effettuerà tutte le operazioni.
- `unimodular` (`fmpz_mat`, default: `None`): Matrice unimodulare usata per la generazione della base pubblica.
- `message` (`fmpz_mat`, default: `None`): Vettore rappresentante il messaggio che verrà cifrato dal crittosistema.
- `error` (`fmpz_mat`, default: `None`): Vettore rappresentante l'errore che verrà aggiunto in fase di cifratura.
- `sigma` (`Integer`, default: `None`): Valore di sigma con il quale verrà generato il vettore di errore.
- `integer_sigma` (`Boolean`, default: `True`): Se `True`, la classe genera un sigma di tipo `Integer` anziché `Float`.
- `debug` (`Boolean`, default: `False`): Se `True`, mostra in console gli output dettagliati di tutte le fasi del crittosistema, includendo il tempo richiesto per ciascuna fase.

Dopo l'assegnazione dei valori alle variabili interne, la classe eseguirà controlli per verificare che, se un vettore o una base sono stati forniti come input, la loro dimensione sia (`dimension`, `dimension`) nel caso delle matrici e (1, `dimension`) nel caso dei vettori. Se tale condizione dovesse risultare falsa per qualsiasi dei parametri, come risposta la classe ritornerà un'eccezione di tipo `ValueError`, con annesso un messaggio riguardante il parametro che ha causato l'errore.

5.2.1 Generazione delle chiavi

La composizione delle chiavi necessita della generazione di tre elementi: la base privata $\mathbf{R}$, σ e una matrice unimodulare $\mathbf{U}$. In [4] vennero proposte più opzioni di generazione per ciascuno di questi elementi. Nella presente implementazione, sono state selezionate alcune opzioni specifiche tra quelle proposte, che verranno discusse singolarmente nel contesto delle prestazioni offerte. Le due chiavi sono rappresentate da due tuple contenute ciascuna nel rispettivo attributo `private_key` o `public_key`.

Chiave privata

La chiave privata è rappresentata dalla sola base privata $\mathbf{R}$. Per generarla è stato deciso di seguire i passaggi descritti in Sezione 3.1.1 al punto 2. I parametri utilizzati sono quelli descritti nei risultati sperimentali riportati in [4], ovvero $k = (l\lceil\sqrt{\texttt{dimension}} + 1\rceil)$ con $l = 4$. Attraverso un ciclo, si genera quindi prima una matrice $\mathbf{R}$ di numeri casuali compresi tra $[-l, l-1]$, si ottiene la base privata $\mathbf{R} = \mathbf{R} + k\mathbf{I}$ e si verifica che essa sia invertibile, salvando in contemporanea il risultato dell'inversione. Se essa non dovesse risultare invertibile si procede, grazie al ciclo, ad una nuova generazione. L'algoritmo implementato fa uso dei moduli `Random` e `Sympy` per la rispettiva generazione della matrice casuale e della matrice identità, le quali sono poi convertite entrambe in `fmpz_mat`.

Chiave pubblica

Al contrario della chiave privata, quella pubblica è più complessa e si compone dalla tupla $(\mathbf{B}, \sigma)$. Dopo la generazione di $\mathbf{R}$, si procede subito con la derivazione di σ secondo la prima metrica, basata sulla norma L1, introdotta in 3.1.1. Per il suo calcolo è stato fatto uso di una conversione al tipo `decimal`. Il suo risultato, definito come ρ, viene infine usato per determinare σ attraverso $\sigma = 1/(2\rho)$. Grazie al parametro `integer_sigma` è possibile decidere se lasciare σ in forma di `float` o arrotondarlo per difetto all'intero più vicino. L'arrotondamento è necessario che sia per difetto al fine di non invalidare la precisione del crittosistema: la metrica basata su L1 definisce un limite a σ sotto al quale il successo della decifratura con base privata è sicuro. Ottenuto quindi σ si calcola infine la base pubblica, ottenuta dalla moltiplicazione di una matrice unimodulare $\mathbf{U}$ con la base privata $\mathbf{R}$. Per la creazione di $\mathbf{U}$ è stato scelto di usare la tecnica descritta al punto 1 della Sezione 3.1.1, in quanto i valori della matrice generati sono meno grandi di quelli ottenuti col metodo del punto 2. L'implementazione dell'algoritmo si basa esclusivamente su matrici `SymPy`, integrate con funzionalità del modulo `Random`. Questa scelta è stata fatta dopo aver condotto esperimenti comparativi che hanno evidenziato la superiorità di `SymPy` rispetto a FLINT per questo specifico caso d'uso. Analogalmente alla generazione della base privata, il risultato finale viene poi convertito in `fmpz_mat` .

5.2.2 Cifratura e decifratura

Il processo di cifratura e decifratura rappresenta la parte più delicata dell'implementazione di GGH. Le operazioni al loro interno trasformano il messaggio originale in testo cifrato e viceversa, utilizzando le chiavi generate precedentemente. L'implementazione di questi processi richiede particolare attenzione per garantire sia l'efficienza

computazionale che la sicurezza del sistema, un errore di calcolo dovuto a bassa precisione renderebbe vana la decifratura.

Cifratura

L'implementazione della cifratura nient'altro è che la computazione dell'equazione 1, la quale, fa uso di sole matrici di tipo `fmpz_mat`. La funzione di cifratura si occupa però prima della generazione del messaggio **m** e dell'errore **e**, secondo i semplici passaggi descritti nella sezione 3.1. Per creare **m**, si genera un vettore di valori casuali nell'intervallo $[-127, 128]$ utilizzando il modulo `Random`, che viene poi convertito in una matrice di razionali `fmpq_mat`. A causa di incompatibilità operative tra oggetti `fmpz_mat` e `fmpq_mat`, è obbligatorio salvare **m** come oggetto di tipo razionale. Questa conversione è necessaria perché durante la fase di cifratura si verifica un'operazione di somma tra un `fmpq_mat` e un `fmpz_mat`. Se **m** non viene convertito in formato razionale, l'operazione causa inevitabilmente un errore di tipo `TypeError` nell'ambiente FLINT. Questa conversione forzata non causa nessun effetto negativo in quanto, in seguito a esperimenti, sia velocità che precisione non sono intaccati. Per **e**, si generano valori casuali nell'intervallo $\pm\sigma$. La conversione finale di **e** dipende dal parametro `integer_sigma`: se vero, risulta in una matrice `fmpz_mat` di interi; se falso, produce una matrice `fmpq_mat` di razionali.

Decifratura

La decifratura nel crittosistema GGH si basa essenzialmente sulla risoluzione del CVP. In [4], gli autori presentano due approcci principali: il metodo del piano più vicino e la tecnica di arrotondamento di Babai. Un'analisi dei vantaggi e degli svantaggi di entrambi gli algoritmi è stata trattata nella Sezione 2.4.1. La scelta implementativa è ricaduta sulla tecnica di arrotondamento, principalmente per la sua efficienza computazionale, rinunciando però alla precisione massima ottenibile. Questa perdita di precisione però, in seguito a risultati sperimentali osservabili in Tabella 7 Sezione 6.2, non influisce in maniera drastica sulle probabilità di successo nella decifratura. L'implementazione della decifratura segue fedelmente i passaggi illustrati nell'esempio 3.1.2, mentre l'algoritmo di Babai applicato è stato dettagliatamente descritto nell'esempio 2.4.1. Tutti i calcoli sono gestiti completamente da FLINT eccetto per la funzione di arrotondamento all'intero più vicino, nativa di Python. La funzione di decifratura ritorna infine il messaggio decifrato sottoforma di `fmpq_mat`.

5.2.3 Caso d'uso

Un esempio d'utilizzo generico è proposto in Figura 7: una volta importata la classe `GGHCryptosystem` è possibile istanziarla. In questo caso, l'istanza viene creata senza

parametri aggiuntivi, ad eccezione della dimensione obbligatoria. Dato che tutti i parametri sono nulli, la classe genererà casualmente sia le basi che i vettori. Subito dopo l'istanza, verranno create le chiavi pubblica e privata. Gli altri attributi, come il messaggio, l'errore e il testo cifrato, verranno generati al momento della chiamata della funzione `encrypt`. Utilizzando la funzione `decrypt` invece, si potrà ottenere il testo decifrato e verificarne la correttezza con un semplice controllo.

```
from GGH_crypto import GGHCryptosystem
dimension = 100

GGH_object = GGHCryptosystem(dimension = dimension)
GGH_object.encrypt()

message = GGH_object.message
decrypted_message = GGH_object.decrypt()

print(decrypted_message == message)
```

Figura 7: Esempio di funzionamento della classe `GGHCryptosystem`.

5.3 Modulo GGH-HNF

Poiché GGH-HNF è una variante di GGH, la sua implementazione mantiene la stessa struttura e utilizza alcuni dei parametri e dei meccanismi precedentemente descritti nella scorsa sezione. Il modulo è anch'esso costituito da una classe principale, `GGHHNFCryptosystem`, che, come nel caso del modulo fratello GGH, serve da contenitore per l'accesso a tutte le funzioni. I parametri ereditati e mantenuti da GGH sono: `dimension`, `private_basis`, `public_basis`, `error` e `debug`, che mantengono le stesse proprietà e sono soggetti agli stessi controlli. I nuovi parametri invece sono i seguenti:

- `lattice_point` (`fmpz_mat`, default: `None`) : Vettore rappresentante un punto del reticolo, il quale, una volta moltiplicato con la base pubblica, verrà sottratto ad `error` in fase di cifratura.
- `alpha` (`Float`, default: 0.75): Valore decimale usato come fattore moltiplicativo per ρ in fase di generazione dell'errore causale. La sua lughezza sarà garantita essere strettamente minore di $\alpha\rho$.
- `GGH_private` (`Boolean`, default: `False`): Se `True`, utilizza la tecnica di generazione della base privata di GGH invece di quella proposta da Micciancio.

5.3.1 Generazione delle chiavi

Diversamente da GGH, lo schema GGH-HNF richiede la generazione di solo due elementi per le chiavi: la base privata e ρ. In [7], Micciancio propone un metodo alternativo per la generazione della base privata, pur riconoscendo che l'approccio utilizzato in GGH e descritto nell'implementazione della Sezione 5.2.1 fosse già adeguatamente efficiente. Dato che ambedue le tecniche generano basi con proprietà diverse, è stato scelto di adottarle entrambe dando scelta all'utente di decidere quale usare attraverso il parametro `GGH_private`. Di default la scelta ricade sulla tecnica proposta da Micciancio.

Chiave privata

Come per GGH, la chiave privata è rappresentata dalla sola base privata $\mathbf{R}$. La funzione che si occupa della sua generazione si articola in due fasi indipendenti, determinate dal parametro `GGH_private`. Se `GGH_private` è settato a `True`, viene utilizzato l'algoritmo impiegato nell'implementazione di GGH. In caso contrario, il programma avvia un ciclo in cui genera matrici di numeri casuali nell'intervallo $[-\texttt{dimension}, \texttt{dimension}]$, le riduce tramite l'algoritmo LLL e poi le inverte. Se l'inversione fallisce, il ciclo riparte fino a ottenere una matrice LLL-ridotta invertibile. Come per il primo metodo, anche il secondo sfrutta il modulo `Random` per la generazione della matrice.

Chiave pubblica

Anche la chiave pubblica, come quella privata, presenta delle somiglianze con la sua controparte nel caso di GGH, con la sola eccezione del parametro σ. In GGH-HNF, infatti, σ non è presente ed è sostituito da ρ. Pertanto, la chiave pubblica è costituita dalla tupla $(\mathbf{B}, \rho)$. Come spiegato in 4.1, la base pubblica è ottenibile con il semplice calcolo della forma normale di Hermite, operazione direttamente integrata in FLINT. Al contrario invece il calcolo del ρ è più oneroso, poiché richiede l'ortogonalizzazione tramite il metodo di Gram-Schmidt, descritto in Algoritmo 1. Questo algoritmo è stato implementato nel modulo `Utils` con l'ausilio di `Numpy`, una sua discussione più approfondita può essere trovata nella sottosezione dedicata in Sezione 5.4. Dopo aver ortogonalizzato la base privata, non resta che trovare la norma minima euclidea ed effettuare il calcolo come definito in Equazione 7. Per determinare la norma, è stata utilizzata uitilizzata la funzione illustrata in Figura 6, che sfrutta il modulo `Decimal`.

5.3.2 Cifratura e decifratura

I processi di cifratura e decifratura di GGH-HNF mantengono la stessa struttura generale di quelli presenti nell'implementazione di GGH, con alcune differenze fondamentali. La decifratura segue fedelmente l'algoritmo già presente in GGH, mantenendo gli stessi vantaggi e svantaggi nella risoluzione del CVP. Al contrario, la cifratura subisce una modifica significativa, poiché utilizza un approccio diverso per la generazione e gestione del messaggio e dell'errore.

Cifratura

Come precedentemente introdotto, la cifratura differisce completamente adottando l'Equazione 6. Data $\mathbf{B}$, due sono gli elementi mancanti: un punto $\mathbf{x}$ del reticolo e l'errore $\mathbf{e}$. Il primo valore viene calcolato direttamente usando l'Equazione 5 che sfrutta interamente le matrici FLINT e l'operazione arrotondamento per difetto nativa di Python. La generazione dell'errore utilizza una funzione che crea un vettore di errore casuale attraverso il modulo `Random`. Il vettore di errore viene generato attraverso un processo iterativo che inizia con la creazione di un vettore casuale nell'intervallo [`-dimension`, `dimension`]. Viene calcolata poi la lunghezza del vettore che deve essere strettamente inferiore a `alpha` $\times \rho$. Se così non fosse, il vettore viene rigenerato con valori casuali diminuendo il parametro `dimension` dal range iniziale finché non si ottiene un errore che soddisfa il criterio desiderato. Questo metodo assicura che l'errore generato sia sempre entro i limiti accettabili definiti dal ρ, dando la possibilità all'utente di poter controllare la probabilità di decifratura con base privata attraverso il parametro `alpha`.

Decifratura

La decifratura, come introdotto, segue gli stessi passaggi di GGH, con l'unica differenza nel valore di ritorno. Dato che il messaggio da recuperare è codificato in $\mathbf{e}$ è necessario ritornare il testo cifrato a cui viene sottratto il risultato della tecnica di arrotondamento di Babai, come mostrato più dettagliatamente nell'Esempio 4.1.1. La scelta di questo algoritmo per la risoluzione del CVP porta a un'ulteriore conseguenza: il vettore di errore $\mathbf{e}$ non basta che sia semplicemente minore di ρ, ma è necessario che sia metodicamente minore a causa della minore precisione dell'algoritmo. Risultati sperimentali, osservabili in Tabella 7 Sezione 6.2, hanno dimostrato che è sufficiente un $\mathbf{e} < 0.8\rho$ per assicurare un successo con probabilità stimata del 97.92%. Inoltre, nel caso in cui il parametro `GGH_private` sia impostato su `True`, gli stessi risultati indicano che un valore di $\mathbf{e} < \rho$ garantisce una probabilità di decifratura pari al 100%.

5.3.3 Caso d'uso

Come osservabile in Figura 8, il caso d'uso della classe `GGHHNFCryptosystem` risulta quasi identico a quello del modulo `GGH` in quanto, come già discusso, entrambi i sistemi seguono la stessa struttura. L'unica differenza sostanziale sta nel fatto che l'attributo `message` non è più presente e al suo posto viene utilizzato `error`.

```
from GGH_crypto import GGHHNFCryptosystem
dimension = 100

GGHHNF_object = GGHHNFCryptosystem(dimension = dimension)
GGHHNF_object.encrypt()

message = GGHHNF_object.error
decrypted_message = GGHHNF_object.decrypt()

print(f"message: {message}")
```

Figura 8: Esempio di funzionamento della classe `GGHHNFCryptosystem`.

5.4 Modulo Utils

Questo terzo e ultimo modulo completa il pacchetto `GGH_crypto`, offrendo algoritmi e strumenti generali e utili per i due crittosistemi descritti e implementati nelle sezioni precedenti. Anche in questo caso è caratterizzato da una classe `Utils` che, sebbene priva di un costruttore, serve esclusivamente come contenitore per i metodi richiesti dai crittosistemi o per metodi indipendenti utili nella crittografia basata sui reticoli. Nelle prossime sottosezioni verranno esaminate e discusse le singole funzioni, organizzate per categoria.

5.4.1 Conversione e norme

La prima categoria di funzioni trattate riguarda quelle relative alla conversione e alle norme. Per quanto riguarda la conversione, sono presenti solo due funzioni specifiche: `npsp_to_fmpz_mat` e `npsp_to_fmpq_mat`, illustrate rispettivamente in Figura 9 e Figura 10. Come suggerisce il nome, queste funzioni eseguono la trasformazione di oggetti `Numpy` o `Sympy` in `fmpz_mat` e `fmpq_mat`. La conversione inversa non è necessaria, poiché `Numpy` e `Sympy` sono in grado di interpretare istanze FLINT convertite in liste tramite il metodo `tolist` integrato in quest'ultimo. Ulteriori conversioni non necessitano di funzioni proprie in quanto sono strettamente specifiche al contesto in cui si trovano.

```
def npsp_to_fmpz_mat(basis):
    return fmpz_mat([[int(item) for item in sublist]
            for sublist in basis.tolist()])
```

Figura 9: Funzione del modulo `Utils` che converte oggetti `Numpy` o `Sympy` in `fmpz_mat`.

```
def npsp_to_fmpq_mat(basis):
    fractions = [[Fraction(item) for item in row]
                            for row in basis.tolist()]
    return fmpq_mat([[fmpq(f.numerator, f.denominator)
                        for f in row] for row in fractions])
```

Figura 10: Funzione del modulo `Utils` che converte oggetti `Numpy` o `Sympy` in `fmpq_mat`.

Le norme contenute in `Utils` includono la L1 ed L2. La norma L2, implementata tramite la funzione `vector_l2_norm`, è osservabile in Figura 6 Sezione 5.1.2, mentre la norma L1 segue una struttura simile, differenziandosi principalmente per un calcolo leggermente più complesso, poiché le due norme misurano distanze in modi diversi. Quest'ultima norma è implementata tramite la funzione `vector_l1_norm`. Entrambe fanno uso di conversioni da `fmpq_flint` a `Decimal` con una precisione dei calcoli settata a 50.

5.4.2 Scrittura e lettura su file

La seconda categoria di funzioni concerne quelle riguardanti la scrittura e la lettura, su file di testo, di matrici FLINT. Due sono le funzioni appartenenti a questa categoria:

- `write_matrix_to_file(matrix, filename)`: Questa funzione consente di scrivere una matrice FLINT su un file di testo. Il parametro `matrix` rappresenta l'oggetto matrice da salvare, che può essere sia di tipo `fmpz_mat` che `fmpq_mat`. Il parametro `filename` specifica il nome del file in cui salvare la matrice. La funzione costruisce automaticamente il percorso completo del file utilizzando la posizione dello script attualmente in esecuzione. Il contenuto della matrice viene scritto in un formato simile a una lista di liste Python, con ogni riga della matrice rappresentata come una lista interna. Gli elementi sono separati da spazi all'interno di ogni riga, e le righe sono separate dal carattere newline, ovvero `\n`. Per le matrici `fmpq_mat`, i numeri razionali vengono rappresentati nella loro forma frazionaria esatta.

- `load_matrix_from_file(filename, matrix_type='fmpq')`: Questa funzione permette di leggere una qualsiasi matrice FLINT da un file di testo precedentemente scritto con `write_matrix_to_file`. Il parametro `filename` specifica il percorso del file da cui leggere la matrice, mentre `matrix_type` determina il tipo di matrice da caricare ('fmpq' per matrici razionali o 'fmpz' per matrici intere, con 'fmpq' come valore predefinito). La funzione gestisce automaticamente la conversione del contenuto del file nel formato appropriato, utilizzando il modulo `ast` per le matrici intere e un parsing personalizzato per preservare con precisione i valori razionali nelle matrici `fmpq_mat`. Il parsing utilizza espressioni regolari con il modulo `re` per isolare le frazioni, estraendo numeratore e denominatore. Questi valori vengono utilizzati per creare oggetti `Fraction`, che vengono inseriti in una lista e successivamente convertiti in una matrice `fmpq_mat`, la quale viene infine restituita dalla funzione.

Anche se queste due funzioni non vengono impiegate direttamente dai crittosistemi, possono rivelarsi molto utili per il salvataggio di basi e vettori, facilitando così il loro riutilizzo in un secondo momento. Inoltre, entrambe sono utilizzate dal modulo `Utils` durante la fase di riduzione tramite BKZ usando FPLLL. Questo approccio è particolarmente vantaggioso perché la gestione dei dati tramite file rappresenta uno dei metodi più efficaci e affidabili, soprattutto nella comunicazione tra programmi in esecuzione su Windows e su WSL. Utilizzare lo standard input e output potrebbe non essere sufficiente, considerata la grande quantità di dati e le elevate dimensioni coinvolte.

5.4.3 Visualizzazione grafica

```
from GGH_crypto import Utils
from flint import fmpz_mat

R = fmpz_mat([[1, 2],[3, 0]])
B = fmpz_mat([[5, 4],[-6, -6]])
T = fmpz_mat([[5, 3]])
w1 = Utils.babai_rounding(R, V)

w2 = Utils.babai_rounding(B, V)

Utils.visualize_lattice(R, w1, T, B, w2,
                "Babai Visualization Example", limit=5)
```

Figura 11: Esempio di caso d'uso della funzione `visualize_lattice`.

La terza categoria è relativa alla visualizzazione grafica dei dati grazie la libreria `Matplotlib`, integrata nel modulo attraverso una funzione chiamata `visualize_lattice`. I parametri in input accettabili dalla funzione sono:

- `basis_1` (`fmpz_mat`): Base dalla quale il reticolo viene generato e poi visualizzato, parametro solitamente usato per rappresentare la base privata.
- `basis_1_cvp` (`fmpz_mat`): Vettore indicante il punto più vicino a un punto dato, ottenibile con gli algoritmi dedicati usando il parametro `basis_1`.
- `point` (`fmpz_mat`) Vettore indicante un punto generalmente non appartenente al reticolo.
- `basis_2` (`fmpz_mat`, default: `None`): Base secondaria anch'essa generante il reticolo, parametro solitamente usato per rappresentare la base pubblica.
- `basis_2_cvp` (`fmpz_mat`, default: `None`): Vettore indicante il punto più vicino a un punto dato, ottenibile con gli algoritmi dedicati usando il parametro `basis_2`.
- `title` (`String`, default: 'Lattice Plot'): Stringa per impostare il titolo del grafico.
- `limit` (`Integer`, default: 5): Parametro intero per limitare la quantità di punti del reticolo nel grafico.

Questa funzione è progettata per visualizzare un reticolo generato da una o due basi, insieme a punti di interesse specifici, generalmente pensata per la visualizzazione di dati relativi al CVP. Per motivi di performance e complessità dell'output, le basi e i vettori passati come parametri devono avere una dimensione di massimo 2. La funzione inizialmente converte le basi e i vettori in array del modulo `Numpy`, in quanto, `Matplotlib` ci si interfaccia nativamente. Viene successivamente generata una griglia di coordinate intere (meshgrid) moltiplicata poi per `basis_1`, dando origine quindi ai punti del reticolo. La funzione visualizza il reticolo risultante, includendo con diversi colori frecce per le basi, punti di interesse e annotazioni. Le dimensioni della visualizzazione vengono infine regolate automaticamente in base ai punti visualizzati.

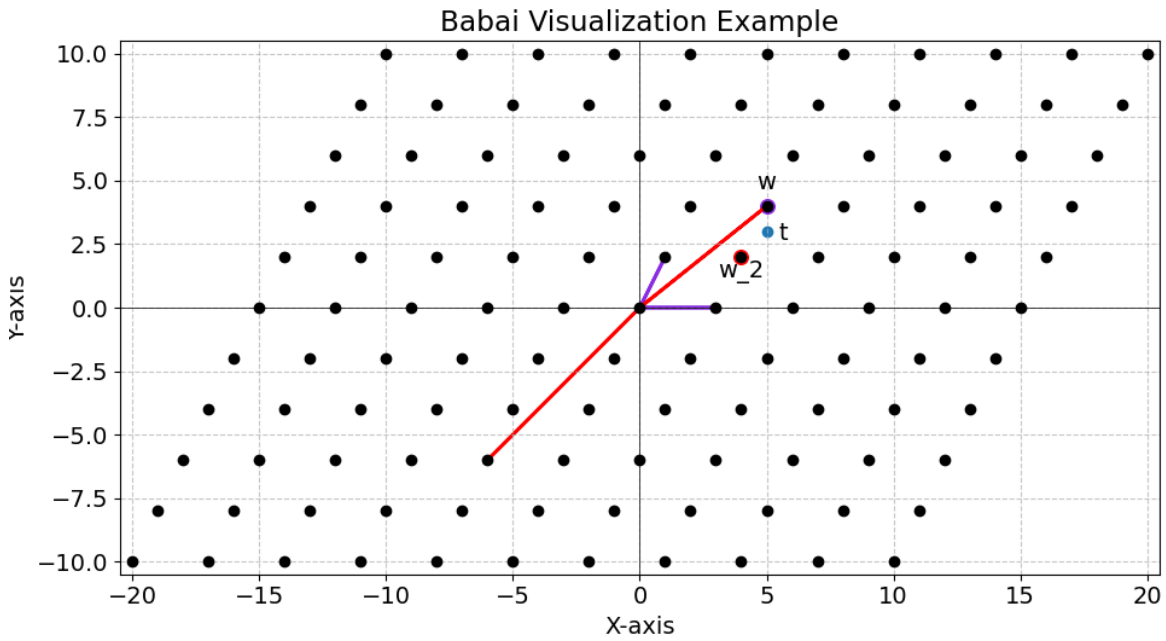


Figura 12: Esempio di visualizzazione della tecnica di arrotondamento di Babai con l'ausilio della funzione `visualize_lattice`.

Tale funzione è direttamente integrata negli algoritmi per la risoluzione del CVP implementati in `Utils` e spiegati nella prossima selezione. È possibile osservare un esempio d'uso in Figura 11 nel quale sono stati usati i dati dell'Esempio 2.4.1. Il risultato è osservabile in Figura 12: in viola è rappresentata la base privata $\mathbf{R}$, insieme al risultato della tecnica di arrotondamento di Babai $\mathbf{w} \in \mathcal{L}(\mathbf{R})$ applicata a tale base. In rosso sono illustrati la base pubblica $\mathbf{B}$ e il punto corrispondente $\mathbf{w\_2} \in \mathcal{L}(\mathbf{B})$ ottenuto utilizzando la stessa tecnica, ma con la base pubblica. In blu è invece evidenziato il punto $\mathbf{t} \notin \mathcal{L}(\mathbf{R})$, per il quale si richiede l'individuazione del punto più vicino appartenente al reticolo.

5.4.4 Algoritmi di risoluzione del CVP

La penultima categoria di funzioni si concentra sugli algoritmi per la risoluzione del CVP. Considerando le scelte di implementazione discusse nelle sezioni 5.2 e 5.3, e seguendo la metodologia proposta da Nguyen nella crittoanalisi presentata in [11], gli unici due algoritmi implementati nel modulo `Utils` sono la tecnica di arrotondamento di Babai e la tecnica di incorporamento. Il primo algoritmo, implementato attraverso la funzione `babai_rounding`, è nettamente il meno complesso dei due in quanto si compone dei soli seguenti parametri:

- `basis` (`fmpz_mat`): Base con la quale l'algoritmo procederà alla risoluzione del CVP.
- `point` (`fmpz_mat`): Punto non appartenente al reticolo generato da `basis` con il quale l'algoritmo procederà alla risoluzione del CVP.
- `visualize` (`Boolean`, default: `False`): Se `True`, attiva la visualizzazione del reticolo chiamando la funzione `visualize_lattice` con i parametri `basis`, `point` e il CVP trovato.

```
    def babai_rounding(basis, point, visualize=False):

        x = point * basis.inv()

        for i in range(x.nrows()):
            for j in range(x.ncols()):
                x[i,j] = round(x[i,j])

        closest_vector = x * basis

        if visualize:
            if basis.nrows() != 2:
                raise ValueError("Dimension Error")
            Utils.visualize_lattice(basis, closest_vector,
                point, title="Babai rounding technique")

        return closest_vector
```

Figura 13: Implementazione della tecnica di arrotondamento di Babai nel modulo `Utils`.

L'implementazione mostrata in Figura 13 è molto semplice, poichè i passaggi da eseguire sono pochi e non richiedono una particolare complessità. Tutti i calcoli sono gestiti da FLINT, fatta eccezione per la funzione `round()`, integrata nativamente in Python. Questa funzione viene direttamente chiamata da entrambe le implementazioni dei due crittosistemi, che poi gestiranno in maniera indipendente il risultato ritornato al fine da ottenere il messaggio originale. Si può anche osservare che la visualizzazione grafica, gestita dal parametro visualize e realizzata tramite la funzione `visualize_lattice`, viene eseguita solo dopo aver verificato che la base sia bidimensionale. È opportuno specificare che il controllo sul parametro `point` risulterebbe inutile in quanto, se fosse di dimensione diversa dalla base, l'algoritmo ritornerebbe un errore a causa dell'impossibilità di moltiplicare matrici e basi di dimensioni diverse. La tecnica di incorporamento invece, implementata attraverso la funzione

`embedding_technique`, è sia teoricamente che implementativamente più complessa. Si compone dei seguenti parametri in input:

- `basis` (`fmpz_mat`): Base reticolare utilizzata nella fase di incorporamento.
- `ciphertext` (`fmpz_mat`): Testo cifrato che verrà incorporato insieme alla matrice `basis` e al quale verrà sottratto il CVP ottenuto al termine dei calcoli.
- `visualize` (`Boolean`, default: `False`) Flag booleana che, se impostata a `True`, inoltrerà `basis`, `ciphertext` e il CVP calcolato, alla funzione `visualize_lattice` per ottenere un risultato grafico.
- `GGH` (`Boolean`, default: `False`): Flag che, se impostata a `True`, attiva una modalità specifica di ricerca del vettore più corto, considerando solo i vettori con l'ultimo elemento pari a 1.
- `BKZ` (`Boolean`, default: `False`): Se `True`, applica l'algoritmo BKZ invece di LLL per la riduzione del reticolo.
- `block` (`Integer`, default: 20): Dimensione del blocco da utilizzare nell'algoritmo BKZ, se attivato.
- `pruned` (`Boolean`, default: `False`): Se `True`, attiva la modalità di pruning nell'algoritmo BKZ. Questa modalità utilizza una strategia default per migliorare le performance dell'algoritmo rinunciando però a parte della precisione nel calcolo della soluzione ottimale.
- `precision` (`Integer`, default: 100): Precisione da utilizzare nei calcoli dell'algoritmo BKZ.
- `bkzautoabort` (`Boolean`, default: `True`): Se `True`, permette all'algoritmo BKZ di interrompersi automaticamente quando non si ottengono ulteriori miglioramenti.
- `bkzmaxloops` (`Boolean`, default: `False`): Se impostato a un valore intero, limita il numero massimo di iterazioni dell'algoritmo BKZ.
- `nolll` (`Boolean`, default: `False`): Se `True` non verrà eseguita una riduzione LLL prima di procedere con la riduzione BKZ. La riduzione LLL non verrebbe effettuata da FLINT, ma bensì direttamente da FPLLL.

L'implementazione di tale tecnica si compone inizialmente dalla costruzione della matrice come descritto in Sezione 2.4.2 e mostrato nell'Esempoio 3.2.1. La costruzione fa

uso unicamente di oggetti FLINT e list comprehension, metodi concisi per la creazione di liste più o meno complesse in Python. A seconda del parametro `BKZ` poi, viene deciso come effettuare la riduzione della matrice costruita: se tale parametro è impostato a `True`, allora verrà fatta una chiamata alla funzione di riduzione BKZ presente in `Utils` con il passaggio dei relativi parametri. In caso contrario l'opzione default è una riduzione LLL tramite FLINT. Dopo la riduzione, l'algoritmo cerca il vettore più corto nella matrice ridotta. Il procedimento itera su tutte le righe, calcolando la norma L2 di ciascun vettore attraverso la funzione integrata `vector_l2_norm` e tracciando quello con la norma minore. Se il parametro `GGH` è `True`, il processo considera solo vettori con l'ultimo elemento uguale a 1, altrimenti considera tutti i vettori. Se l'algoritmo non trova vettori validi, viene eseguita una seconda ricerca considerando tutti i vettori disponibili. Questo approccio garantisce sempre la restituzione di un risultato, anche se potrebbe non essere la soluzione corretta al problema. Il CVP infine viene calcolato sottraendo il vettore trovato dal testo cifrato originale. Anche questa funzione consente una visualizzazione grafica del risultato attraverso il medesimo parametro `visualize` e gli stessi controlli implementati in `babai_rounding`.

5.4.5 Riduzione e qualità di una base

L'ultima categoria di funzioni presenti nel modulo `Utils` è quella relativa alla misurazione della qualità di una base e alla sua riduzione. Per la prima tipologia è stata introdotta un'unica funzione, chiamata `get_hadamard_ratio`, responsabile del calcolo del rapporto di Hadamard, discusso in Sezione 2.3.1. La sua implementazione, osservabile in Figura 14, si caratterizza dall'uso del modulo `Decimal` invece che FLINT. Questa scelta è stata forzata dal fatto che FLINT non dispone nativamente dell'operazione di elevazione alla potenza, che in questo caso deve essere fatta attraverso un numero frazionario. Al fine di calcolare il determinante è stato comunque usato FLINT, mentre per il calcolo della norma è stata usata la funzione `vector_l2_norm`. La funzione `get_hadamard_ratio` accetta solo due parametri, ovvero:

- `basis` (`fmpz_mat` o `fmpq_mat`): Base reticolare della quale viene calcolato il rapporto di Hadamard.
- `precision` (`Integer`, default: 10): Precisione con la quale verranno effettuati i calcoli dal modulo `Decimal`. Questo parametro specifica inoltre di quante cifre decimali sarà composto il risultato formattato.

```
        def get_hadamard_ratio(basis=None, precision=10):
            norms = []
            dimension = matrix.nrows()

            getcontext().prec = precision

            for i in range(matrix.nrows()):
                row = fmpz_mat([[matrix[i, j]
                                 for j in range(matrix.ncols())]])
                norm = Utils.vector_l2_norm(row)
                norms.append(Decimal(str(norm)))

            log_denominator = sum(norm.ln() for norm in norms)
            log_numerator = abs(Decimal(matrix.det().str())).ln()

            log_result = (log_numerator - log_denominator) /
                                                Decimal(dimension)
            result = log_result.exp()

            return result, f"{result:.{precision}f}"
```

Figura 14: Funzione del modulo `Utils` per il calcolo del rapporto di Hadamard.

Un aspetto rilevante dell'implementazione è l'ampio impiego di operazioni logaritmiche. Operando nello spazio logaritmico, si prevengono problemi di stabilità numerica che potrebbero verificarsi manipolando direttamente numeri di scale molto diverse. Nel caso del rapporto di Hadamard il problema può verificarsi nella moltiplicazione di norme vettoriali, che nel caso di matrici a grandi dimensioni, può causare un overflow. La funzione ritorna infine due valori: il risultato sottoforma di oggetto `Decimal` e una sua versione formattata in stringa e limitata a `precision` cifre dopo la virgola.
La seconda tipologia, relativa alla riduzione di una base reticolare, è invece composta dalle due funzioni `gram_schmidt` e `BKZ_reduction`. La prima, nient'altro è, che un'implementazione in Python dell'Algoritmo 2, attraverso l'uso del modulo `Numpy`. La costruzione inizialmente utilizzava esclusivamente oggetti FLINT e costrutti Python, ma dopo vari esperimenti è stata scartata a causa dei tempi di calcolo eccessivi: per ortogonalizzare una matrice di 300 dimensioni era necessaria oltre un'ora di elaborazione. La nuova implementazione, osservabile in Figura 15 è invece in grado di ottenere i medesimi risultati riducendo però ad una manciata di secondi il tempo di computazione richiesto a parità di input. La base in input necessita una conversione da `fmpz_mat` ad array `Numpy` mentre, viceversa, l'output riceve il procedimento inverso con l'uso di `npsp_to_fmpq_mat`.

```
def gram_schmidt(basis):
    B = basis[0:1,:].copy()
    for i in range(1, basis.shape[0]):
        proj = np.diag((basis[i,:].dot(B.T)
                /np.linalg.norm(B,axis=1)**2).flat).dot(B)

        B = np.vstack((B, basis[i,:] - proj.sum(0)))
    return B
```

Figura 15: Funzione del modulo `Utils` per l'ortogonalizzazione Gram-Schmidt.

La seconda ed ultima funzione `BKZ_reduction` invece esegue una riduzione BKZ alla base passata come input. Gli altri parametri di questa funzione sono gli stessi descritti precedentemente nella sezione che illustra la funzione `embedding_technique`. Tutti questi parametri vengono direttamente utilizzati nella costruzione del comando `fplll`. Tale comando è proposto direttamente dalla libreria e, per una migliore comprensione dei relativi parametri, si rimanda alla sezione 6.3. Come inizialmente introdotto in Sezione 5.1, tutti i passaggi di dati tra il modulo `Utils` e `FPLLL` avvengono attraverso l'uso di files, quindi con l'ausilio delle funzioni `write_matrix_to_file` e `load_matrix_from_file`. Successivamente al salvataggio su file `input.txt` della base in input, inizia il processo di costruzione del comando. Questo ha una struttura base che include parametri predefiniti, che vengono poi personalizzati in base agli input della funzione. Questa personalizzazione avviene sostituendo dei segnaposto o aggiungendo ulteriori parametri alla fine del comando. Il sistema effettua un rilevamento automatico del sistema operativo prima di procedere e, se viene identificato Windows, l'istruzione finale sarà preceduta dal prefisso `wsl`. Questo consente l'esecuzione del comando nell'ambiente Linux integrato in Windows. Su sistemi Linux nativi, invece, il comando viene eseguito direttamente senza alcun prefisso, poiché l'ambiente Unix-like è già disponibile. Nelle figure 16 e 17 è possibile osservare due esempi di comandi che sono stati impiegati per attaccare GGH attraverso il metodo di Nguyen in Sezione 6.3.

```
wsl fplll input.txt -a bkz -b 20 -p 100
-f mpfr -m wrapper -bkzautoabort > out.txt
```

Figura 16: Comando Windows FPLLL per una riduzione BKZ-20 con auto-abort.

Dopo che il comando è stato generato, viene eseguito utilizzando il modulo `subprocess` con la funzione `Popen`. L'output e gli errori del processo vengono catturati tramite `stdout` e `stderr`, rispettivamente, e successivamente decodificati in stringhe di testo. Se il comando termina in maniera controllata e senza nessun errore fatale, il risultato viene salvato in un file `out.txt` e poi caricato in un oggetto attraverso

`load_matrix_from_file`. Infine entrambi i file di input e output vengono cancellati e la matrice caricata viene ritornata.

```
fplll input.txt -a bkz -b 60 -p 100 -bkzmaxloops 5
-s default.json -f mpfr -m wrapper -nolll > out.txt
```

Figura 17: Comando Linux FPLLL per BKZ-60 potato con auto-abort a 5 iterazioni, senza riduzione LLL preliminare.

Capitolo 6

Risultati sperimentali

In questo capitolo vengono presentati e discussi i risultati sperimentali ottenuti attraverso una serie di test condotti sul progetto. Il focus principale degli esperimenti è stato la valutazione dell'efficienza e della sicurezza delle due implementazioni dei crittosistemi discusse nelle sezioni precedenti. È importante sottolineare che questi test non si sono limitati ai singoli moduli crittografici, ma hanno considerato il progetto nella sua interezza, includendo tutte le sue componenti interconnesse attraverso il terzo modulo di supporto `Utils`. Per fornire una valutazione completa, i risultati ottenuti sono stati confrontati con dati di test passati eseguiti da altri autori, ove disponibili. Questa comparazione ha permesso di esaminare i vantaggi e le limitazioni delle nuove implementazioni rispetto a soluzioni già note e testate. Tutte le misurazioni sono state effettuate su una macchina con 16 GB di RAM e un processore Intel Core i5-11400F, con una frequenza operativa compresa tra 1.60 GHz e 4.40 GHz. Per una maggiore chiarezza, in questo capitolo, il termine "GGH-HNF" farà sempre riferimento alla versione con `GGH_private` impostato su `False`, salvo diversa indicazione.

6.1 Generazione delle chiavi

Per confrontare le prestazioni delle due implementazioni in termini di tempi di generazione e dimensioni delle chiavi, si è proceduto con la generazione di 10 coppie di chiavi (pubbliche e private) per ogni dimensione, partendo da 100 e aumentando di 100 unità alla volta fino a raggiungere 800. Tutti i parametri dei crittosistemi sono sono rimasti impostati sul default, fatta eccezione per `dimension` e `debug`, il primo usato per dettare la dimensione ad ogni iterazione e il secondo per effettuare il calcolo dei tempi di ciascuno step. I tempi di generazione delle chiavi sono dati dalle seguenti formule:

- Chiave privata:

  - Per entrambi i crittosistemi è stato sufficiente calcolare il tempo di generazione della base casuale (nel caso di GGH-HNF con riduzione LLL). Il tempo finale comprende anche il calcolo del determinante della matrice risultante per verificarne l'invertibilità. Se una matrice generata non risulta essere invertibile, l'iterazione viene rieseguita per non inquinare i risultati finali.

- Chiave pubblica:

  - GGH: Il tempo di generazione della chiave pubblica è dato:

$$\texttt{pubkey\_time} = \texttt{unim\_time} + \texttt{sigma\_time} + \texttt{pub\_time}$$

    dove: `unim_time` è il tempo di generazione della matrice unimodulare, `sigma_time` è il tempo di generazione del sigma e `pub_time` è il tempo per effettuare il calcolo dell'Equazione 1.

  - GGH-HNF: Il tempo di generazione della chiave pubblica è dato:

$$\texttt{pubkey\_time} = \texttt{rho\_time} + \texttt{pub\_time}$$

    dove: `rho_time` è il tempo di generazione del ρ e `pub_time` è il tempo per effettuare il calcolo dell'Equazione 6.

Dimensione	Tempo chiave privata (s)		Tempo chiave pubblica (s)	
	GGH	GGH-HNF	GGH	GGH-HNF
100	0.009	0.030	0.491	0.063
200	0.046	0.187	3.626	0.213
300	0.111	0.618	12.991	0.618
400	0.234	0.927	34.056	1.444
500	0.411	1.676	74.576	2.728
600	0.930	2.747	184.469	6.065
700	1.107	5.484	251.482	14.525
800	1.546	5.942	368.591	22.277

Tabella 1: Tempi medi di generazione delle chiavi di GGH e GGH-HNF.

Analizzando i risultati complessivi riportati nelle Tabelle 1 e 2, relative ai tempi di generazione in secondi e alle dimensioni in Kilobytes delle chiavi, emergono sostanziali differenze tra GGH e GGH-HNF. Per quanto riguarda la chiave privata, GGH dimostra prestazioni leggermente superiori sia in termini di velocità di generazione che di dimensioni, con tempi e spazio di archiviazione consistentemente inferiori per tutte

Dimensione	Dimensione chiave privata (KB)		Dimensione chiave pubblica (KB)	
	GGH	GGH-HNF	GGH	GGH-HNF
100	24.862	34.615	97.656	44.611
200	98.580	156.392	677.253	195.580
300	221.042	366.645	2107.403	462.928
400	392.332	666.127	4936.998	851.883
500	612.420	1053.891	9481.981	1366.516
600	882.048	1529.102	16249.625	2009.303
700	1200.219	2093.324	25162.143	2782.517
800	1566.549	2744.888	36664.683	3688.281

Tabella 2: Dimensioni medie delle chiavi di GGH e GGH-HNF.

le dimensioni testate. Al contrario, GGH-HNF si distingue ampiamente nella gestione della chiave pubblica, superando di gran lunga GGH in entrambi gli aspetti. In particolare, i tempi di generazione della chiave pubblica per GGH-HNF sono significativamente più bassi (ad esempio, circa 22 secondi contro 368 secondi di GGH per una dimensione di 800), e le dimensioni delle chiavi pubbliche risultano notevolmente più contenute (circa 3.7 MB per GGH-HNF contro 36.7 MB per GGH, sempre per dimensione 800). I risultati confermano come la forma normale di Hermite sia nettamente migliore della controparte usata in GGH. Grazie alla sua struttura triangolare, metà della matrice è composta da zeri, il che comporta due vantaggi significativi quando viene impiegata come operatore: una riduzione delle dimensioni della matrice e una minore complessità computazionale. La generazione della chiave pubblica nel sistema GGH è significativamente rallentata dal processo di inversione della chiave privata, necessario per il calcolo di σ. Come evidenziato in Tabella 3, che illustra i tempi medi di generazione per ogni fase, questo passaggio costituisce una porzione considerevole del tempo totale richiesto per la generazione della chiave pubblica, circa il 62.53%. Sebbene sia possibile eliminare il tempo di generazione di σ impostando un valore prefissato per il parametro `sigma`, GGH-HNF rimane comunque più veloce. Come illustrato nella Tabella 4, i tempi di generazione per GGH-HNF benché condizionati dal calcolo della forma normale di Hermite, risultano sorprendentemente inferiori a quello che ci si aspettava nel rispetto dei test eseguiti in [6], rendendo la generazione complessivamente più veloce. Per quanto riguarda la chiave privata di GGH-HNF invece, è possibile affermare che il suo tempo di generazione medio è trascurabilmente superiore a quello di GGH, con una differenza di circa 4 secondi in dimensione 800. Al contrario la dimensione è maggiore di circa 1.1MB rendendola quindi di qualità inferiore rispetto alla sua controparte.

Dimensione	Tempo medio Sigma (s)	Tempo medio Matrice Unimodulare $\mathbf{U}$ (s)	Tempo medio Calcolo di $\mathbf{U} \times \mathbf{R}$ (s)
100	0.181	0.310	0.001
200	1.845	1.768	0.013
300	7.817	5.113	0.061
400	21.459	12.460	0.137
500	49.384	24.937	0.255
600	129.887	53.991	0.592
700	185.891	64.772	0.819
800	368.591	97.709	1.372

Tabella 3: Tempi medi di generazione degli step della chiave pubblica in GGH.

Dimensione	Tempo medio Rho (s)	Tempo medio HNF (s)
100	0.043	0.021
200	0.193	0.213
300	0.471	0.713
400	0.927	1.444
500	1.676	2.728
600	2.747	6.065
700	5.484	14.525
800	5.942	16.335

Tabella 4: Tempi medi di generazione degli step della chiave pubblica in GGH-HNF.

Un'ulteriore proprietà che differenzia particolarmente i due metodi di generazione della chiave privata sta nel valore assunto da ρ, un parametro chiave che determina il successo della decifratura e la lunghezza assumibile dal vettore di errore. I dati relativi al cambiamento del ρ, in funzione della dimensione della chiave privata, riportati in Figura 18, sono stati raccolti durante un secondo esperimento, il quale sarà descritto dettagliatamente nella sezione successiva. È subito osservabile quanto il valore medio di ρ cresca in maniera significativamente più rapida nel caso in cui `GGH_private` sia impostato su `True` rispetto a `False`. In particolare, per dimensioni più grandi della chiave, come 800, si nota che il valore di ρ supera i 9600 con `GGH_private` = `True`, mentre rimane intorno ai 600 con `GGH_private` = `False`. Le conseguenze di tali differenze drastiche, discusse più in dettaglio nella sezione successiva, è notabile in fase di decifratura e potrebbe essere altrettanto importante per la crittoanalisi.

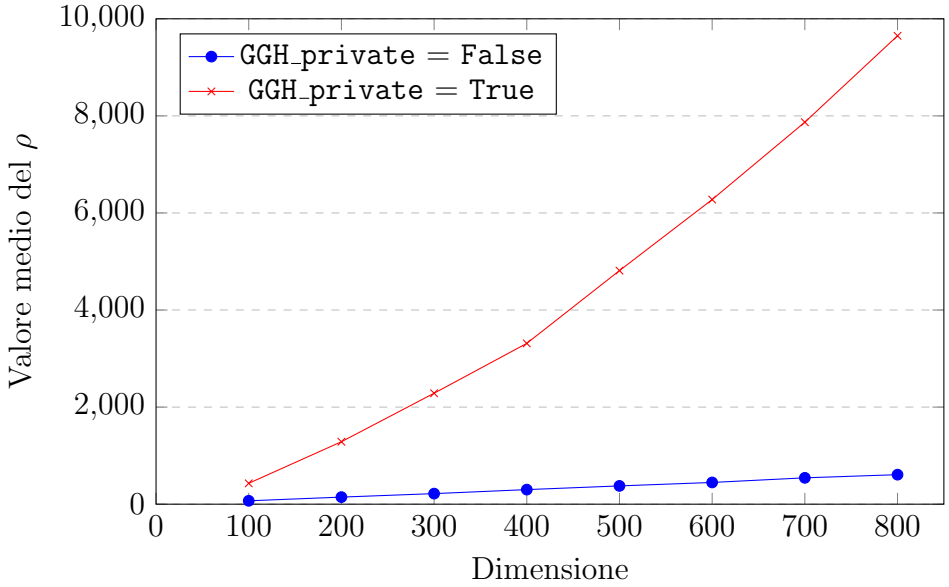


Figura 18: Confronto del valore medio di ρ con `GGH_private` = `True` o `False`.

6.2 Cifratura e decifratura

Dimensione	Tempo di cifratura (s)		Tempo di decifratura (s)	
	GGH	GGH-HNF	GGH	GGH-HNF
100	≈0	0.001	0.323	0.227
200	≈0	0.005	3.787	2.979
300	0.004	0.011	14.820	13.214
400	0.014	0.019	41.887	39.836
500	0.025	0.030	102.415	94.337
600	0.044	0.047	272.059	225.689
700	0.049	0.071	352.680	343.373
800	0.070	0.077	792.555	645.152

Tabella 5: Tempi di cifratura e decifratura medi di GGH e GGH-HNF.

L'esperimento illustrato all'inizio della sezione precedente, che ha analizzato le caratteristiche delle chiavi generate, è stato ulteriormente ampliato per includere una valutazione approfondita delle prestazioni dei processi di cifratura e decifratura. Oltre all'analisi delle chiavi, sono stati misurati i tempi medi di esecuzione, nonché calcolata la dimensione media del testo cifrato risultante. I risultati di queste misurazioni sono stati organizzati in due tabelle distinte: la Tabella 5 presenta i dati relativi

ai tempi di esecuzione, mentre la Tabella 6 riporta le informazioni sulla grandezza del testo cifrato. Entrambi i crittosistemi mostrano tempi di cifratura significativamente più bassi rispetto a quelli di decifratura. GGH presenta un lieve vantaggio in termini di velocità di cifratura, ma la differenza è minima. È da sottolineare il fatto che i tempi di cifratura rimangono contenuti, non superando 0.1 secondi anche per input di dimensioni considerevoli, il che evidenzia l'efficienza computazionale dei metodi di cifratura impiegati in entrambi gli schemi. In fase di decifratura invece si osserva una crescita non lineare dei tempi all'aumentare della dimensione dell'input, con GGH-HNF che mostra generalmente prestazioni leggermente migliori rispetto a GGH, soprattutto per dimensioni maggiori. Questa differenza è attribuibile al fatto che GGH richiede l'inversione sia della base privata che di quella pubblica, mentre GGH-HNF necessita solo dell'inversione della base privata. La notevole disparità temporale tra i processi di cifratura e decifratura è un fenomeno atteso. Questa differenza è principalmente dovuta alla necessità di invertire almeno una base durante la decifratura, un'operazione che richiede un considerevole sforzo computazionale.

Dimensione	Dimensione testo cifrato (KB)	
	GGH	GGH-HNF
100	1.412	0.450
200	4.626	0.984
300	9.830	1.549
400	16.802	2.135
500	25.629	2.739
600	36.511	3.355
700	48.587	3.982
800	62.014	4.617

Tabella 6: Dimensioni medie del testo cifrato di GGH e GGH-HNF.

Spostando l'attenzione invece sulla dimensione del testo cifrato è immediatamente evidente che GGH-HNF superi notevolmente GGH in termini di efficienza spaziale. GGH-HNF riesce a mantenere la dimensione del testo cifrato sotto i 5 KB, persino quando si lavora con una dimensione di 800. Al contrario, GGH produce un output significativamente più voluminoso, raggiungendo i 62 KB nelle stesse condizioni. Le motivazioni dietro questa diversità peculiare sta nella struttura intrinseca dei due testi cifrati. Osservando la Figura 19, che illustra i vettori rappresentanti i testi cifrati generati in dimensione 7 con parametri predefiniti, si nota che il testo cifrato di GGH-HNF, denominato $\mathbf{c}_{\text{HNF}}$, si compone della maggior parte di numeri a singola cifra, seguiti infine da un unico numero intero di grande valore. Questa struttura è il risultato diretto dell'operazione modulare di $\mathbf{e}$ con la base pubblica in forma HNF, la cui particolare configurazione genera questo output compatto. In contrasto,

il testo cifrato di GGH, $\mathbf{c}_{\text{GGH}}$, presenta una serie di numeri interi di media grandezza, sia positivi che negativi, distribuiti uniformemente nel vettore. La dimensione di questi valori è direttamente proporzionale alla grandezza delle chiavi, portando il testo cifrato ad appesantirsi sempre di più con l'aumentare della dimensione.

$$\mathbf{c}_{\text{HNF}} = \texttt{[0, 0, 0, 0, 0, 0, 2371763]}$$
$$\mathbf{c}_{\text{GGH}} = \texttt{[6074, 1396, -137, 235, 842, -4540, -4766]}$$

Figura 19: Comparazione fra le strutture dei testi cifrati di GGH e GGH-HNF.

Nell'implementazione di GGH-HNF è stata data la possibilità all'utente di poter decidere con quale metodo generare la chiave privata attraverso il parametro `GGH_private`. Come precedentemente accennato, è stato quindi svolto un secondo esperimento per valutare le conseguenze e i cambiamenti di tale scelta. Dal precedente test possiamo affermare che il metodo di GGH genera matrici meno pesanti con meno tempo rispetto alla tecnica usata da Micciancio. Ciò che non viene dimostrato è l'impatto in termini di percentuale di successo e sicurezza. Per valutare il secondo termine è stato svolto un ulteriore test specifico, descritto nella prossima sezione. Per quanto riguarda l'analisi dei successi, seguendo l'approccio del precedente esperimento, sono state generate nuovamente 10 coppie di chiavi pubbliche e private per ciascuna dimensione, partendo da 100 fino a 800, con incrementi di 100. Tuttavia, per valutare l'efficacia di ciascun metodo di generazione della chiave privata, è stato introdotto il parametro `sigma`. In questo nuovo test, per ogni dimensione e per ogni set di 10 prove, sono stati applicati diversi valori del valore ρ, modulati da un fattore moltiplicativo determinato dal parametro `sigma`. I valori di `sigma` utilizzati erano compresi tra 0.50 e 1, con incrementi di 0.10. Complessivamente, sono state generate e analizzate 480 istanze del crittosistema, risultanti dalla combinazione di 10 test per ciascuna delle 8 dimensioni considerate e per ognuno dei 6 valori di `sigma`. I risultati, osservabili in Tabella 7, sono sorprendenti. Il metodo di GGH mostra una performance perfetta in tutte le dimensioni e per tutti i valori di `sigma`, con un tasso di successo del 100% in ogni caso. Il metodo di Micciancio invece, seppur mostri generalmente buone prestazioni, soffre di alcune imperfezioni con un totale di 16 fallimenti sui 480 tentativi. Il fatto che il numero di errori aumenti con l'aumentare di `sigma` non è una sorpresa: dato l'uso della tecnica di arrotondamento di Babai per la decifratura, una scelta di lunghezza del vettore di errore troppo vicina al ρ risulta in un aumento delle probabilità di insuccesso come sottolineato in [6, Sezione 3.1]. Durante il test sono anche stati rilevati dati sul ρ medio delle chiavi private in riferimento alla dimensione, i risultati sono stati discussi nella precedente sezione e sono osservabili in Figura 18. L'alto valore di ρ delle chiavi private generate da GGH sono probabilmente la causa di questo aumento critico dei successi di decifratura, poiché un ρ più grande aumenta la distanza tra i punti del reticolo, rendendo più robusta la decifratura rispetto agli

errori introdotti dal vettore di errore. Questo maggiore "margine di errore" permette al metodo GGH di mantenere un tasso di successo del 100% anche con valori di `sigma` più elevati, dove il metodo di Micciancio inizia a mostrare alcune imperfezioni.

Dimensione	`GGH_private`	$\alpha = 0.5$	$\alpha = 0.6$	$\alpha = 0.7$	$\alpha = 0.8$	$\alpha = 0.9$	$\alpha = 1.0$
100	`False`	10/10	9/10	9/10	10/10	8/10	8/10
	`True`	10/10	10/10	10/10	10/10	10/10	10/10
200	`False`	10/10	9/10	10/10	10/10	10/10	8/10
	`True`	10/10	10/10	10/10	10/10	10/10	10/10
300	`False`	10/10	10/10	9/10	9/10	10/10	9/10
	`True`	10/10	10/10	10/10	10/10	10/10	10/10
400	`False`	10/10	10/10	10/10	10/10	10/10	9/10
	`True`	10/10	10/10	10/10	10/10	10/10	10/10
500	`False`	10/10	9/10	10/10	10/10	10/10	10/10
	`True`	10/10	10/10	10/10	10/10	10/10	10/10
600	`False`	10/10	10/10	10/10	10/10	10/10	10/10
	`True`	10/10	10/10	10/10	10/10	10/10	10/10
700	`False`	10/10	10/10	10/10	10/10	10/10	9/10
	`True`	10/10	10/10	10/10	10/10	10/10	10/10
800	`False`	10/10	10/10	10/10	10/10	10/10	9/10
	`True`	10/10	10/10	10/10	10/10	10/10	10/10

Tabella 7: Successi nella decifratura per α con `GGH_private` = `True` e `False`.

6.3 Sicurezza

L'ultimo test eseguito sui crittosistemi proposti riguarda la loro sicurezza contro gli attacchi. Per questa analisi, sono state impiegate due metodologie distinte, ciascuna specifica per il crittosistema in esame. Nonostante le differenze, entrambi gli approcci si basano sull'utilizzo del modulo FPLLL, precedentemente menzionato, con particolare attenzione agli algoritmi BKZ e LLL. Prima di approfondire le due strategie di attacco, è fondamentale comprendere il funzionamento di FPLLL, con un focus particolare sui suoi molteplici parametri. Di seguito, un elenco dei principali parametri utilizzati nei test:

- `-a bkz`: Esegue la riduzione BKZ del reticolo contenuto nel file `input.txt`. Di default viene eseguita una preliminare riduzione LLL completa prima di procedere con BKZ.
    - `-b block_size`: Imposta la dimensione del blocco per BKZ.

    - `-bkzautoabort`: Interrompe l'esecuzione quando la pendenza media dei $\log \|\mathbf{b}_i^*\|$ non diminuisce abbastanza rapidamente, ovvero quando la riduzione sta ottenendo miglioramenti talmente trascurabili da poterli ritenere accettabili come soluzione.
    - `-bkzmaxloops loops`: Imposta il numero massimo di iterazioni complete dei loop di BKZ.

- `-f mpfr`: Utilizza l'aritmetica in virgola mobile MPFR.
    - `-p precision`: Specifica la precisione dell'aritmetica in virgola mobile MPFR.
- `-nolll`: Evita la riduzione LLL preliminare.
- `-m wrapper`: Utilizza un metodo che fa uso della versione eurisitica o della versione dimostrata di BKZ a seconda della fase in cui si trova.
- `-s default.json`: Usa strategie per il preprocessing e imposta parametri di pruning da una configurazione default proposta dagli autori.

Una documentazione più dettagliata di tutti i parametri è visitabile in [19]. FPLLL offre un'implementazione all'avanguardia dell'algoritmo BKZ, incorporando tutte le ottimizzazioni sviluppate negli ultimi anni. Tuttavia, questa implementazione presenta una limitazione significativa: non è possibile personalizzare il fattore di potatura. Gli utenti sono vincolati ad utilizzare esclusivamente il file di configurazione predefinito fornito dagli sviluppatori di FPLLL. Questa restrizione ha delle implicazioni importanti per l'analisi: le strategie condotte non possono replicare esattamente le controparti di precedenti studi, come quelli riportati da Nguyen [11] e Ludwig [6]. Nonostante ciò, prendendo spunto dai loro lavori, sono stati elaborati tre approcci di attacco distinti: uno specifico per GGH e due per GGH-HNF. Per quest'ultimo la divisione in due approcci è dettata dal parametro `GGH_private` che, come già visto nella scorsa sezione, impatta notevolmente sulle probabilità di successo. L'obiettivo del test è quindi valutare la sicurezza di entrambi i crittosistemi, concentrandosi in particolare sulle implicazioni per la sicurezza di GGH-HNF a seconda che il parametro `GGH_private` sia impostato su `True` o `False`. A seguito di varie misurazioni, le strategie ottimali per ciascuno dei crittosistemi sono le seguenti:

- GGH: Una volta identificata l'istanza del CVP semplificata attraverso l'attacco di Nguyen, si procede con la tecnica di incorporamento. Alla matrice ottenuta vengono applicate poi una serie di riduzioni attraverso i comandi rispettivamente ossevabili in Figura 16 e in Figura 17. In breve vengono applicati nell'ordine una riduzione LLL, una BKZ-20 con auto-abort e una BKZ-60 prunata con auto-abort a 5 iterazioni senza riduzione LLL preliminare.

- GGH-HNF: Data la differenza di architettura di questo crittosistema rispetto all'originale, l'attacco di Nguyen non risulta più attuabile. Nonostante ciò è comunque possibile applicare la tecnica di incorporamento attraverso l'uso della base pubblica e del testo cifrato. Alla matrice risultante vengono applicate le stesse riduzioni usate con GGH, con l'unica differenza dell'utilizzo di un BKZ-20 potato invece che normale.

- GGH-HNF con `GGH_private` = `True`: La medesima strategia precedentemente introdotta viene impiegata anche quando `GGH_private` = `True`.

Tutte gli approcci sfruttano il metodo `wrapper`, che favorisce un buon bilanciamento tra precisione ed efficienza, e una precisione di 100 con il tipo `mpfr`. È opportuno segnalare che, per migliorare l'efficienza, la precisione dovrebbe essere adattata in base alle dimensioni della matrice in input. Ad esempio, per una matrice di 300 dimensioni, è sufficiente una precisione di 90, mentre per una matrice di 350 è invece necessaria una precisione di almeno 100. Occorre inoltre specificare che FPLLL non fornisce un sistema per interrompere il processo una volta individuato un determinato vettore, il che impedisce una conclusione ottimale in termini di tempo. Per aggirare il problema sono stati aggiunti i due parametri di abort, il secondo in particolare ferma il processo ogni 5 iterazioni per verificare se la soluzione è stata trovata. Nei risultati riportati nelle Tabelle 8 e 9, i minuti successivi all'iterazione che ha prodotto la soluzione sono stati esclusi dal conteggio finale; ad esempio, se il vettore è stato trovato all'iterazione 3, i minuti impiegati per le restanti 2 sono saranno scartati dal conto complessivo. Ciascun attacco è stato eseguito tre volte, impiegando istanze diverse del sistema crittografico per ogni tentativo. I tempi di esecuzione di ogni prova sono stati registrati al fine di calcolare le medie complessive, successivamente riportate nelle tabelle dei risultati.

Dimensione	GGH	
	Tempo (min)	Metodo risolutivo
100	0.136	LLL
200	5.965	BKZ-20
300	238.240	BKZ-60 (P)
350	-	Non risolto
400	-	Non risolto

Tabella 8: Tempi medi per attaccare GGH su diverse dimensioni.

Dimensione	GGH-HNF			
	`GGH_private` = `False`		`GGH_private` = `True`	
	Tempo (min)	Metodo risolutivo	Tempo (min)	Metodo risolutivo
100	0.191	LLL	0.199	LLL
200	5.005	BKZ-20 (P)	5.779	BKZ-20 (P)
300	37.018	BKZ-60 (P)	593.782	BKZ-60 (P)
350	232.412	BKZ-60 (P)	-	Non risolto
400	992.087	BKZ-60 (P)	-	Non risolto

Tabella 9: Tempi medi per attaccare GGH-HNF su diverse dimensioni.

La scelta di utilizzare BKZ-20 potato invece di quello non potato per GGH-HNF è stata determinata a seguito di test complementari che ne hanno dimostrato miglioramenti in termini di velocità. In particolare si ipotizza che questi miglioramenti siano dovuti alla alla particolare struttura della forma normale di Hermite. Dai risultati presentati è possibile osservare innanzitutto come l'algoritmo BKZ, come già introdotto in Sezione 2.3.4, abbia una complessità esponenziale. Per esempio, considerando la Tabella 9 relativa a GGH-HNF, per una dimensione di 100, l'algoritmo LLL risolve il problema in soli 11 secondi. Passando a una dimensione di 200, dove viene utilizzato BKZ-20 potato, il tempo di esecuzione sale a circa 5 minuti. Per una dimensione di 300, con l'impiego di BKZ-60 potato, il tempo di esecuzione aumenta drasticamente a circa 37 minuti. Proseguendo, per una dimensione di 350, il tempo di esecuzione cresce ulteriormente a circa 4 ore. Infine, per una dimensione di 400, il tempo di esecuzione impiega almeno 16 ore. Il tempo allocato per ciascun attacco è stato di massimo 4 giorni. Le istanze marcate come "Non risolto" nelle tabelle indicano che, anche dopo questo esteso periodo di elaborazione, non si è giunti a una soluzione. La probabile causa di questo fenomeno può essere ricercata sia nell'hardware usato, classificabile come di fascia media, e sia nel fatto che FPLLL non fornisce un meccanismo per specificare il fattore di pruning. Per quanto riguarda le performance dei singoli crittosistemi, è possibile notare come La versione originale di GGH-HNF sia generalmente meno resistente agli attacchi, con un tempo stimato di appena 37 minuti in dimensione 300. Ciò cambia però drasticamente per quanto riguarda GGH e la versione di GGH-HNF con `GGH_private` = `True`. Per questi ultimi si osserva infatti un tempo stimato di rispettivamente 238 e 593 minuti sempre in dimensione 300. È possibile concludere quindi che la versione di GGH-HNF con `GGH_private` = `True`, sia il crittosistema più resistente dei tre.

6.4 Confronti con studi precedenti

In questa sezione, i risultati degli esperimenti condotti verranno messi a confronto con studi precedenti sui crittosistemi implementati e su schemi tradizionali. Questa comparazione permetterà di contestualizzare i risultati ottenuti e di valutarne la loro coerenza con quelli ricavati da altri studi. Sfortunatamente, non sono stati trovati dati relativi all'implementazione originale dello schema GGH come per GGH-HNF. Tuttavia, è stato possibile reperire informazioni su un'implementazione realizzata con Mathematica, descritta in [13]. Questo studio offre un'analisi delle prestazioni dello schema, fornendo risultati complessivi per: la generazione delle chiavi, la cifratura di un testo e la sua successiva decifratura. L'analisi copre un range di dimensioni che spazia da 10 a 50, mantenendo costante il valore del parametro sigma a 3 e impiegando la tecnica di incorporamento. Questi dati, sebbene non direttamente correlati all'implementazione originale, costituiscono un valido punto di riferimento per la valutazione comparativa delle prestazioni dell'implementazione descritta in questa tesi. In particolare nella Tabella 1 di [13, Sezione 6] può essere osservato che il tempo richiesto in dimensione 50 è stato di circa 500 secondi. Questo risultato, considerando la dimensione relativamente modesta, suggerisce l'utilizzo di strumenti e/o hardware non ottimali o di fascia inferiore nell'implementazione descritta nello studio. Tale ipotesi trova ulteriore conferma nel confronto con l'implementazione proposta nel presente lavoro. Sorprendentemente, l'implementazione proposta, operando su una dimensione significativamente maggiore (400), richiede un tempo complessivo di circa 392.332 secondi per completare tutte le fasi del processo. Questo tempo, che rappresenta la somma delle durate di ciascuna fase operativa, risulta addirittura inferiore a quello riportato in [13] per una dimensione otto volte minore.

Caratteristica	GGH-HNF in [6]	GGH-HNF
Tempo Chiave privata	4.5 ore	5.942 secondi
Tempo Chiave pubblica	46 ore	22.277 secondi
Tempo Cifratura	0.29 secondi	0.077 secondi
Tempo Decifratura	73.7 minuti	10.752 minuti
Dimensione chiave pubblica	1 MB	3725.132 KB in txt 954.684 KB con compressione

Tabella 10: Confronto dei tempi medi d'esecuzione in dimensione 800 tra l'implementazione di GGH-HNF di Ludwig e quella proposta in questa tesi.

Per quanto riguarda GGH-HNF, è possibile ottenere informazioni sia dalla pubblicazione originale [6] per le dimensioni dei testi cifrati, sia dal report tecnico di Ludwig Christoph [6] per quanto riguarda i tempi di esecuzione e gli aspetti relativi alla

sicurezza. Un breve riassunto delle differenze tra l'implementazione originale del crittosistema nello studio di Ludwig e quella presentata in questa tesi è osservabile in Tabella 10. I risultati sono notevolmente sorprendenti: l'implementazione proposta mostra miglioramenti significativi in termini di efficienza raggiungendo diminuzioni di tempo considerevoli. In particolare, si osservano riduzioni drastiche nei tempi di generazione delle chiavi. La creazione della chiave privata passa da 4.5 ore a soli 5.942 secondi, mentre la generazione della chiave pubblica si riduce da 46 ore a 22.277 secondi. Questi miglioramenti rappresentano un incremento di oltre 2700 volte per la chiave privata e circa 7400 volte per la chiave pubblica. Anche i tempi di cifratura e decifratura mostrano ottimizzazioni rilevanti: la cifratura è circa 3.8 volte più veloce passando da 0.29 a 0.077 secondi, mentre la decifratura, pur rimanendo l'operazione più onerosa, si riduce da 73.7 minuti a 10.752 minuti, con un miglioramento di quasi 7 volte.

Formato	GGH, dimensione = 800			
	Chiave privata	Chiave pubblica	Testo cifrato	Tempo
txt	1582.093 KB	38900.546 KB	28.671 KB	-
gzip	426.187 KB	17079.172 KB	28.671 KB	4.08 s
bz2	**280.169** KB	17057.499 KB	28.599 KB	**3.489** s
lzma	376.138 KB	**16798.197** KB	**28.434** KB	9.346 s

Tabella 11: Confronto di vari algoritmi di compressione applicati a GGH.

Per quanto riguarda la dimensione della chiave pubblica, l'implementazione proposta risulta notevolmente peggiore nella sua versione originale con un aumento delle dimensioni di circa 2.7MB. Questo peggioramento è tuttavia prevedibile, poiché l'argomento era già stato discusso nella Sezione 5.1, dove sono state discusse inoltre delle considerazioni su possibili approcci per migliorarne le performance. A tal proposito è stato svolto un test per verificare quanto si potesse ridurre la grandezza delle chiavi e del testo cifrato per entrambi i crittosistemi: ogni file è stato prima deserializzato con `Pickle` e poi compresso con uno dei tre seguenti algoritmi di compressione: gzip, bz2 e lzma. I risultati nelle Tabelle 11 e 12 mostrano che le chiavi GGH rimangono di dimensioni intrattabili per livelli superiori. L'algoritmo di compressione migliore risulta essere generalmente bz2 sia per qualità di riduzione di grandezza che tempo richiesto. Questo algoritmo è riuscito a ridurre notevolmente la chiave privata da 1582.093 KB a 280.169 KB. Tuttavia, la chiave pubblica, seppur dimezzata, mantiene dimensioni considerevoli, limitandone l'applicabilità pratica. Gli stessi risultati al contempo sottolineano invece come le chiavi di GGH-HNF e il suo testo cifrato giovino particolarmente da questo trattamento. Entrambe le chiavi passano da rispettivamente da 2772.010 e 3725.132 KB a "soli" 954.684 e 1035.702 KB, allineandosi con i dati riportati in [6]. Il testo cifrato invece, a differenza di quello di

GGH che non è impattato particolarmente dalla compressione, viene ridotto da 4.662 a 1.456 KB. Concludendo invece per quanto riguarda la sicurezza dei crittosistemi in esame, è possibile comparare i dati della Tabella 8 con lo studio fatto da Nguyen in [11] e quelli della Tabella 9 con i risultati sempre svolti da Ludwig in [6]. Confrontando i dati riguardo la sicurezza di GGH con lo studio di Nguyen [11], emerge una discrepanza nei risultati. Nguyen ha risolto problemi fino a dimensione 350 usando tecniche BKZ e pruning, con tempi crescenti all'aumentare della dimensione. I risultati rilevati durante i test invece mostrano tempi inferiori per dimensioni minori, ma l'impossibilità di trovare una soluzione già dalla dimensione 350. Questa differenza è attribuibile ai diversi metodi usati per affrontare i valori razionali nella riduzione: Nguyen ha utilizzato l'approccio del punto 2 mentre nell'implementazione è stato impiegato quello del punto 1 che risulta essere meno efficace. Questo perché il vettore di errore generato contiene valori 0 e 1, producendo un vettore più corto rispetto a uno composto unicamente da valori 1. Considerando i tempi più rapidi nelle dimensioni in cui è stata trovata una soluzione, si può affermare che, se fosse stato utilizzato lo stesso metodo, l'attacco avrebbe avuto successo in un tempo inferiore rispetto a quello ottenuto da Nguyen.

Formato	GGH-HNF, dimensione = 800			
	Chiave privata	Chiave pubblica	Testo cifrato	Tempo
txt	2772.010 KB	3725.132 KB	4.662 KB	-
gzip	1418.059 KB	1053.693 KB	1.458 KB	3.064 s
bz2	**954.684** KB	**1035.702** KB	1.760 KB	**2.088** s
lzma	1171.314 KB	1044.256 KB	**1.456** KB	3.678 s

Tabella 12: Confronto di vari algoritmi di compressione applicati a GGH-HNF.

La stessa affermazione può essere fatta anche per quanto riguarda i risultati ottenuti da Ludwig. I suoi esperimenti hanno trattato attacchi a GGH-HNF per dimensioni tra 50 e 280 rilevando in particolare i seguenti tempi con un vettore di errore scontato del 30%: 50 secondi per dimensione 100, 27 minuti per 200 e 6 ore per 280. Poiché i test mostrano che una riduzione dello sconto comporta un aumento del tempo di attacco, ci si aspetta che utilizzando un valore di `alpha` = 0.75 l'implementazione corrente offra una resistenza leggermente superiore. Tuttavia, come avviene per GGH, gli attacchi eseguiti sulla versione proposta di GGH-HNF hanno rivelato tempi di esecuzione per le stesse dimensioni pari a: 11 secondi per dimensione 100, 5 minuti per dimensione 200 e 37 minuti per dimensione 300.
Per completare il quadro comparativo, è opportuno confrontare le prestazioni dell'implementazione proposta di GGH-HNF con quelle di sistemi crittografici asimmetrici ampiamente utilizzati e consolidati, come RSA ed ElGamal. Questi crittosistemi si

basano su problemi completamente diversi dai reticoli: RSA si basa sulla difficoltà di fattorizzare grandi numeri primi ed ElGamal sfrutta il problema del logaritmo discreto.

Caratteristica	RSA	ElGamal	GGH-HNF
Tempo Chiavi	24.03 secondi	7.84 secondi	28.21 secondi
Tempo Cifratura	0.39 secondi	0.28 secondi	0.077 secondi
Tempo Decifratura	0.12 secondi	0.031 secondi	10.75 minuti
Dimensione chiave pubblica	2048 bit	2048 bit	3725.132 KB in txt 954.684 KB con compressione

Tabella 13: Confronto dei tempi medi d'esecuzione con parametri sicuri tra l'implementazione proposta e sistemi tradizionali.

I dati riguardanti i sopracitati crittosistemi sono stati rilevati dallo studio presentato in [17], il quale offre risultati molto recenti e aggiornati. La Tabella 13 presenta un confronto dei tempi medi di esecuzione e delle dimensioni delle chiavi tra GGH-HNF ed gli attuali sistemi RSA ed ElGamal, utilizzando parametri considerati sicuri secondo gli standard odierni. Per GGH-HNF, è stata quindi utilizzata una dimensione di 800 seguento quanto dimostrato in [6], mentre per RSA ed ElGamal sono state impiegate chiavi di 2048 bit. Quest'ultima scelta è in linea con le raccomandazioni del National Institute of Standards and Technology (NIST), che considera tali chiavi sufficientemente sicure almeno fino al 2030. GGH-HNF risulta essere tempisticamente quasi al pari con i risultati di RSA per la generazione delle chiavi e leggermente migliore per quanto riguarda la cifratura rispetto ad entrambe le controparti. Tuttavia le rimanenti due caratteristiche soffrono di svantaggi significativi in comparazione ai dati presentati dagli altri schemi. La decifratura in GGH-HNF richiede un tempo notevolmente superiore rispetto a RSA ed ElGamal, con una durata di 10.75 minuti contro frazioni di secondo per gli altri due sistemi. La dimensione della chiave pubblica per GGH-HNF risulta essere anch'essa sostanzialmente maggiore rispetto a RSA ed ElGamal. Mentre questi ultimi utilizzano chiavi di 2048 bit (circa 256 byte), GGH-HNF richiede 3725.132 KB (circa 3.64 MB) in formato testuale, che si riduce a 932 KB con la compressione. Anche nella sua forma compressa, la chiave pubblica di GGH-HNF è quasi 4 volte più grande di quelle di RSA ed ElGamal, rendendola competitivamente fuori discussione rispetto le altre due.

Capitolo 7

Conclusioni

In questa tesi sono stati approfonditi argomenti chiave per la crittografia basata sui reticoli, con relative implementazioni pratiche per dimostrarne le effettive proprietà. Nello specifico i due soggetti chiave sono stati i crittosistemi GGH e GGH-HNF, quest'ultimo una versione migliorata del primo. Gli studi eseguiti su di essi non miravano alla sola revisitazione degli stessi in un contesto molto più moderno rispetto a quello in cui sono nati, ma anche al tentativo di proporne miglioramenti ulteriori. I risultati ottenuti hanno dimostrato come l'avanzamento tecnologico abbia incrementato drasticamente le performance di questi due crittosistemi, sia a livello hardware che software. È infatti notevole osservare come un computer di fascia media odierno, se equipaggiato con software ottimizzato, sia in grado di superare nettamente le prestazioni dei computer di fascia alta di circa vent'anni fa. È stato provato inoltre come un linguaggio di programmazione considerato "lento" come Python sia in grado invece, attraverso opportune integrazioni con librerie di basso livello, a fornire alte prestazioni aggiuntive ai suoi tanti vantaggi, tra cui: leggibilità, flessibilità e possibilità di essere eseguito su piattaforme diverse senza dover installare ulteriori dipendenze. Un'ulteriore scoperta eseguita è legata alla versione ibrida tra GGH e GGH-HNF, che utilizza la generazione della chiave privata di GGH integrata nella restante struttura di GGH-HNF. I dati di questa versione infatti spiccano tra le altre, dimostrando di essere migliore sotto tutti i punti di vista. Se però le prestazioni dei crittosistemi hanno giovato particolarmente dalle nuove tecnologie usate, stessa cosa potrebbe essere infatti detta per gli attacchi su di essi. Tale supposizione risulta parzialmente vera: gli attacchi hanno impiegato sensibilmente meno tempo per essere svolti, ma sia a causa dell'esponenzialità di BKZ che dei limiti teorici dei reticoli, essi non sono andati oltre alle dimensioni già precedentemente raggiunte. Nello specifico per GGH i dati sono rimasti coerenti con le scoperte di Nguyen in [11], con una discrepanza in dimensione 350 dovuta alle differenze nelle scelte teorico-implementative usate. Per quanto riguarda GGH-HNF invece, gli studi precedenti si sono fermati in pratica a

dimensione 280, calcolando attraverso proiezioni statistiche che una dimensione di 800 fosse necessaria per una sicurezza adeguata. I risultati sperimentali di questa tesi sono stati limitati alla dimensione 400 con un tempo richiesto di circa 16 ore, è però verosimile ammettere che, un uso di hardware migliore e di un algoritmo di riduzione costruito ad hoc, potrebbero permettere attacchi su dimensioni maggiori di essere raggiunti in un tempo accettabile. Le stesse considerazioni non possono però essere stanziate anche per la versione ibrida di GGH-HNF, la quale ha dimostrato una resistenza superiore a GGH originale, impiegandoci il doppio del tempo richiesto per rompere quest'ultimo in dimensione 300 e non andando oltre. È molto importante questo dato in quanto questa versione non è soggetta alla vulnerabilità scoperta da Nguyen, deve essere infatti attaccata usando la stessa metodologia usata per GGH-HNF. Ulteriori studi sulla sua sicurezza e le sue proprietà teoriche sono però necessari prima di affermare che essa possa risultare come un punto di svolta nel contesto di GGH.
Per concludere, una domanda che resta senza risposta è la seguente: GGH e le sue varianti potrebbero essere usate in ambito crittografico pratico? Sfortunatamente considerando una dimensione di 800, ritenuta un limite pratico dove la sicurezza è provata, i dati dimostrano come alcune proprietà dei crittosistemi restino poco competitive rispetto ad altri schemi odierni come RSA. Il processo di decifratura richiede un minimo di 10 minuti, utilizzando un metodo che non garantisce il pieno successo nel recupero della soluzione, senza considerare la versione ibrida del crittosistema che non risulta essere impattata da questo fattore. Inoltre, ben più grave, sono le dimensioni delle chiavi che non sono cambiate dai dati di circa vent'anni fa, neanche utilizzando algoritmi di compressione. Bisogna considerare però che non tutte le strategie di ottimizzazione disponibili sono state usate e che forse è possibile ridurre ulteriormente le dimensioni delle stesse. Anche se ulteriori approcci dovessero funzionare, sarebbe poco verosimile raggiungere le caratteristiche di RSA, dove le chiavi arrivano a qualche centinaio di bytes.

Bibliografia

[1] Aharonov Dorit e Regev Oded, "Lattice Problems in NP ∩ coNP", CiteSeer X (The Pennsylvania State University), 2009

[2] Babai László, "On Lovász' Lattice Reduction e the Nearest Lattice Point Problem," Combinatorica, vol. 6, no. 1, pagine 1–13, 1986

[3] Galbraith Steven, "Mathematics of Public Key Cryptography", seconda edizione, Cambridge University Press, 2018

[4] Goldreich Oded, Goldwasser Shafi e Halevi Shai, "Public-key Cryptosystems from Lattice Reduction Problems," Advances in Cryptology — CRYPTO '97, Springer, volume 1294, pagine 112–131, 1997

[5] Lenstra Arjen Klaas, Lenstra Hendrik Willem e László Lovász, "Factoring polynomials with rational coefficients", Mathematlsche Annalen, Springer, volume 261, pagine 515-534, 1982

[6] Ludwig Christoph, "The Security and Efficiency of Micciancio's Cryptosystem", Technische Universität Darmstadt Germany, 2004

[7] Micciancio Daniele, "Improving Lattice Based Cryptosystems Using the Hermite Normal Form", Lecture Notes in Computer Science, 9500 Gilman Drive La Jolla CA 92093 USA, pagine 126–145, 2001

[8] Micciancio Daniele e Regev Oded, "Lattice-based Cryptography," Post-Quantum Cryptography, pagine 147–191, 2009

[9] Micciancio Daniele e Goldwasser Shafi, "Complexity of Lattice Problems: a cryptographic perspective", The Kluwer International Series in Engineering and Computer Science, Boston, Massachusetts, Kluwer Academic Publishers, volume 671, 2002

[10] Moon Sung Lee e Sang Geun Hahn, "Cryptanalysis of the GGH Cryptosystem", Mathematics in Computer Science, volume 3, pagine 201-208, 2010

[11] Nguyen Phong, "Cryptanalysis of the Goldreich-Goldwasser-Halevi Cryptosystem from Crypto '97," Advances in Cryptology — CRYPTO' 99, 45 rue d'Ulm, 75230 Paris Cedex 05, France, pagine 288–304, 1999

[12] Nguyen Phong e Damien Stehlé, "Floating-point LLL revisited", LNCS, Springer, volume 3494, pagine 215-233, 2005

[13] Schenström Amelie, "The GGH Encryption Scheme - A Lattice-Based Cryptosystem", Matematiska Institutionen Stockholms Universitet, Stoccolma, 2016

[14] Schnorr Claus Peter e M. Euchner, "Lattice basis reduction: Improved practical algorithms and solving subset sum problems", Mathematical Programming, volume 66, pagine 181-199, 1994

[15] Schnorr Claus Peter e H. H. Hörner, "Attacking the Chor-Rivest cryptosystem by improved lattice reduction", Proc. of Eurocrypt'95, Springer-Verlag, volume 921, pagine 1-12, 1995

[16] Silverman Joseph H., Pipher Jill e Hoffstein Jeffrey, "An introduction to mathematical cryptography", seconda edizione, Springer, Undergraduate texts in mathematics, 2014

[17] S. J. Mohammed and D. B. Taha, "Performance Evaluation of RSA, ElGamal, and Paillier Partial Homomorphic Encryption Algorithms", International Conference on Computer Science and Software Engineering (CSASE), Duhok, Iraq, 2022, pagine. 89-94, 2022

[18] Fast Library for Number Theory, URL: `https://flintlib.org/`

[19] FPLLL, a lattice reduction library, URL: `https://github.com/fplll/fplll`