UNIVERSITA' POLITECNICA DELLE MARCHE

FACOLTA' DI INGEGNERIA

Corso di Laurea triennale Ingegneria Elettronica

"Dimensionamento ed ottimizzazione di uno schema post-quantum di firma digitale basato su codici"

"Design and optimization of a code-based post-quantum digital signature scheme"

Relatore: Chiar.mo
Prof. Franco Chiaraluce

Tesi di Laurea di:
Rebecca Giuliani

Correlatore:
Dr. Paolo Santini

A.A. 2020 / 2021

INDICE

1. INTRODUZIONE

Con il passare degli anni si stanno sviluppando nuove tipologie di computer, denominati “quantum computer”. Questa nuova tecnologia sfrutta fenomeni fisici, come lo spin dell'elettrone ed il principio di sovrapposizione degli stati, per raggiungere elevatissime capacità di calcolo, notevolmente più grandi di quelle che si possono avere con calcolatori classici.

Attualmente, dimensioni e costi di questi nuovi computer sono talmente elevati da non essere minimamente paragonabili con quelli dei computer in commercio. Infatti proprio a causa della loro scarsa praticità i quantum computer attualmente in uso sono davvero pochi.

Nonostante ciò i giganti dell’informatica, come Google, Microsoft e IBM, stanno investendo cifre esorbitanti per sviluppare dei computer quantistici, che siano di scala comparabile con quelli classici attuali. Alcuni hanno ipotizzato che questa nuova tipologia di computer sarà stata sviluppata e sarà utilizzabile già dal 2030.
Si prevede che i computer quantistici permetteranno di fare importanti passi avanti nello studio di molti problemi, che originano da svariati campi (da quello farmaceutico a quello chimico a quello dell’ingegneria dei materiali), problemi che non sarebbe possibile risolvere in modo efficiente con i computer odierni.

Ovviamente questa innovazione oltre agli svariati aspetti positivi avrà anche dei risvolti negativi. È infatti dimostrato che, con i computer quantistici, è possibile implementare degli algoritmi di natura diversa (chiamati appunto quantistici), tramite i quali è possibile attaccare in modo efficiente la quasi totalità degli schemi crittografici a chiave pubblica che sono in uso oggi.

Verso la metà degli anni ’90 Peter Shor propose un algoritmo quantistico, che permette la fattorizzazione di un numero intero in tempi polinomiali. Questo algoritmo è diventato molto famoso poiché permette di risolvere il problema matematico che sta

alla base di molti algoritmi di crittografia asimmetrica, come ad esempio RSA o Diffie-Hellman, rendendone del tutto insicuro l'utilizzo. Quindi con l'avvento dei calcolatori quantistici, a causa dell'algoritmo di Shor, questi schemi di crittografia diventerebbero obsoleti. Ad oggi, sono numerose le applicazioni che si basano sull'utilizzo di algoritmi a crittografia asimmetrica: basti pensare alle firme digitali, ai certificati di firma, alla cifratura con algoritmi asimmetrici, alla navigazione in Internet, in pratica alla grande maggioranza degli utilizzi quotidiani della crittografia.

Vista quindi l'evidente necessità di disporre di algoritmi che resistano agli attacchi dei computer quantistici e, in particolare, che sostituiscano gli algoritmi che risultano attaccabili con l'algoritmo di Shor, si sta attualmente assistendo allo sviluppo di una nuova generazione della crittografia, la "crittografia post-quantum".

Per "crittografia post-quantum" s'intende quella crittografia su cui si basano nuovi protocolli di firma, cifratura e scambio di "key material" immuni agli attacchi crittografici lanciati da un computer quantistico, e che sarà fondamentale per sostenere la sicurezza di Internet, dei dati e di tutte quelle tecnologie basate sulla crittografia classica.

Attualmente tra gli schemi più studiati e, quindi, ritenuti più affidabili vi sono quelli basati sui codici (code-based) a blocco lineari, in cui vengono aggiunti degli errori per aumentare la sicurezza dello schema.

Il panorama di firma code-based è molto diverso rispetto a quello degli schemi di cifratura. Infatti questo è privo di soluzioni sicure ed, allo stesso tempo, efficienti.
Uno dei primi schemi proposti è stato il "CFS" [6], che è basato su codici Goppa binari. Lo schema ancora non è stato attaccato, ma è molto lento ed ha chiavi pubbliche molto grandi.
Altri schemi recenti sono ad esempio [7] e [8], ma questi sono stati già rotti dagli attacchi riportati negli articoli [9] e [10]. Tali attacchi si basano sul fatto che ciascuna

firma rivela un'informazione rispetto alla chiave segreta, perciò dopo un po' di firme raccolte, l'attaccante, facendo un analisi statistica, riesce a scoprire la chiave segreta.
Qualcuno ha pensato di usare schemi di tipo "*one-time"*, cioè, una volta fatta la firma, la chiave viene cambiata. In questo modo si eviterebbero gli attacchi statistici, perché la stessa chiave non verrebbe mai usata due volte. Si può trovare un esempio di questo schema in [11], che però è stato attaccato in [12].
Al momento, uno dei pochi schemi rimasti non rotti è "WAVE" [13], che, come lo schema proposto in [1], è basato su codici non binari. Tuttavia non può essere considerato efficiente poiché ha tempi di firma piuttosto lenti e, soprattutto, chiavi molto grandi (dell'ordine del MB).

La letteratura sugli schemi di firma che sono stati attaccati è molto abbondante, a riprova del fatto che progettare schemi di firma sicuri non è un compito banale.
Per questo motivo la ricerca di uno schema di firma che sia efficiente ed in grado di resistere ad algoritmi quantistici rappresenta, ad oggi, una delle principali linee di ricerca.

Il presente lavoro di tesi prende spunto dal documento [1], in cui è stato svolto uno studio finalizzato all'ottimizzazione delle prestazioni dello schema "BCS" (acronimo derivante dalle iniziali degli autori).
In breve, lo schema BCS è un nuovo schema di firma, che utilizza codici non binari e la cui sicurezza è basata sul cosiddetto "Restricted Syndrome Decoding Problem" (R-SDP).
In particolare, lo schema prevede una fase di rejection sampling nell'algoritmo di generazione delle firme.
In estrema sintesi durante questa fase ogni firma che viene generata viene sottoposta ad una fase di test (per verificare che essa abbia determinate proprietà); se la firma non è valida, viene scartata immediatamente. Inoltre, anche in caso di firma valida, vi è una probabilità non nulla che la firma venga scartata, perché per essere accettabile

deve permettere di verificare anche la validità dell'hash. Questa procedura (la cui motivazione matematica è piuttosto complessa), essenzialmente fa sì che l'algoritmo di firma venga ripetuto per numerose volte, prima di ottenere una firma valida (per ulteriori chiarimenti vedi paragrafo "Descrizione dello Schema").
Pertanto, il tasso di rejection è un parametro che va fortemente tenuto sotto controllo, perché di fatto da esso dipende l'efficienza computazionale dello schema BCS.
Inoltre, altri aspetti fondamentali per schemi di questo tipo sono le dimensioni della chiave pubblica e della firma.

In questa tesi è stato svolto un lavoro di progettazione ed ottimizzazione dello schema BCS, cercando di tenere conto di tutti questi aspetti per calcolare set di parametri che possano garantire livelli di sicurezza adeguati e, al tempo stesso, buone prestazioni in termini di dimensioni di chiavi e firme e di efficienza computazionale.
Per realizzare questo scopo è stata prima di tutto definita una procedura di dimensionamento, con l'obiettivo di esplorare esaustivamente range di parametri e trovare il set ritenuto ottimo, secondo le metriche definite e secondo diversi livelli di sicurezza (pari a 128 e 192 bit) e diversi valori per il rate del codice.

Lo studio delle prestazioni dell'algoritmo ad un livello di sicurezza diverso da 128 e con più valori del rate del codice (di seguito indicato con code_rate") indicano le prime differenze da quanto era stato fatto in [1]: si è osservato che più il code_rate aumenta più si hanno dei miglioramenti. Infatti, per security level 128, le dimensioni delle chiavi pubbliche si sono ridotte rispetto ad [1] di più del 55%.

2. NOTAZIONE E BACKGROUND

In questo capitolo verrà descritta la notazione matematica che verrà adottata per tutta la tesi, insieme ai concetti che stanno alla base del funzionamento di uno schema di firma.

2.1 Notazione matematica

In questo lavoro di questa tesi si è lavorato con codici definiti su un campo finito non binario.

Per comprendere cosa sia un campo finito non binario, si può partire dal concetto di campo finito binario, che indichiamo con $\mathbb{F}_2$. Questo campo contiene due elementi: 0 ed 1. I calcoli vengono fatti modulo 2, ad esempio:

$$1 + 1 = 2 \mod 2 = 0.$$

Quando si utilizza il campo finito non binario, che indichiamo con $\mathbb{F}_q$, se q è primo si considera che:

1. il campo contiene q elementi, che sono $\{0,1, \dots, q-1\}$;
2. i calcoli vengono fatti riducendo modulo q.

Per fare un esempio pratico, si scelga $q = 5$: il campo corrispondente $\mathbb{F}_5$ contiene solamente i numeri $\{0,1,2,3,4\}$. Ad esempio, si ha $1 + 4 = 0$, $4 + 4 = 3$, $3 \cdot 4 = 2$, $3 \cdot 2 = 1$. Infatti:

$$\begin{aligned} 1 + 4 \mod 5 = 5 \quad &\mod 5 = 0, \\ 4 + 4 \mod 5 = 8 \quad &\mod 5 = 3, \\ 3 \cdot 4 \mod 5 = 12 \quad &\mod 5 = 2, \\ 3 \cdot 2 \mod 5 = 6 \quad &\mod 5 = 1. \end{aligned}$$

Si lavora con vettori e matrici utilizzando esattamente lo stesso ragionamento. Ad esempio, ragionando nel campo $\mathbb{F}_5$, dato il vettore $[1 \quad 4 \quad 0]$ e la matrice $\begin{bmatrix} 3 & 1 & 4 & 2 & 2 \\ 1 & 0 & 2 & 3 & 4 \\ 2 & 0 & 3 & 1 & 1 \end{bmatrix}$, il loro prodotto equivale a:

$$[1 \quad 4 \quad 0] \cdot \begin{bmatrix} 3 & 1 & 4 & 2 & 2 \\ 1 & 0 & 2 & 3 & 4 \\ 2 & 0 & 3 & 1 & 1 \end{bmatrix} = [2 \quad 1 \quad 2 \quad 4 \quad 3].$$

Per quanto riguarda la rappresentazione dei numeri negativi, ove risultassero dal calcolo, essa è basata sulla seguente corrispondenza:

$$-1 \mapsto q - 1,$$
$$-2 \mapsto q - 2,$$
$$-3 \mapsto q - 3,$$
$$.$$
$$.$$
$$.$$

Ad esempio, lavorando sempre con $q = 5$, si ha:

$-1 \mod 5 = 4$, infatti

$$4 - (-1) = 4 + 1 = 5;$$

$-2 \mod 5 = 3$, infatti

$$3 - (-2) = 3 + 2 = 5.$$

Per comprendere meglio come vengono fatti i calcoli in un campo finito, si può pensare alle lancette dell'orologio. I calcoli, nell'orologio, vengono fatti modulo 12. Ad esempio quando sono le 11 di sera, e aggiungiamo un'ora, l'orologio segna l'ora 0. L'ora successiva viene segnata come 1, eccetera.

$$11 + 1 \mod 12 = 12 \mod 12 = 0,$$

$$11 + 2 \mod 12 = 13 \mod 12 = 1.$$

2.2 Peso di Hamming

Il peso di Hamming di una stringa di lunghezza k è la sua distanza di Hamming dalla stringa costituita da k zeri, quindi il peso di Hamming è il numero di elementi diversi da zero di una stringa. Ad esempio, data la stringa [1 1 0 0 0 1 1] , il suo peso di Hamming è 4 .

Per una stringa non binaria, il peso di Hamming viene generalizzato considerando il numero di elementi diversi da zero. Ad esempio, data la stringa [0 1 3 7 0 1], il peso di Hamming è 4.

2.3 Funzioni di hash

Le funzioni di hash hanno un ruolo cruciale nella crittografia moderna. Una funzione crittografica di hash è un tipo di "*one-way function*" che prende un input arbitrario costituito da un numero qualsiasi di bit (teoricamente infinito) e restituisce un numero fisso di bit. Per poter considerare buona una funzione hash, questa deve godere delle seguenti proprietà:

1. deve essere computazionalmente molto difficile recuperare, a partire dal messaggio in uscita, il messaggio in input, ovvero, la funzione deve essere difficilmente invertibile (da cui, l'appellativo "*one-way*");
2. deve essere valida la proprietà di resistenza alla collisione tipica di una funzione hash, ovvero, dato un valore di hash, deve essere molto difficile (al limite, impossibile) risalire ad una coppia di input (x, x'), tali che $x \neq x'$;
3. dati in input due messaggi poco diversi tra loro, la funzione di hash deve dare in output due valori significativamente diversi e incorrelati tra loro. Questo significa che, in media, gli hash di due input poco diversi tra loro devono essere differenti nel 50% dei loro elementi.

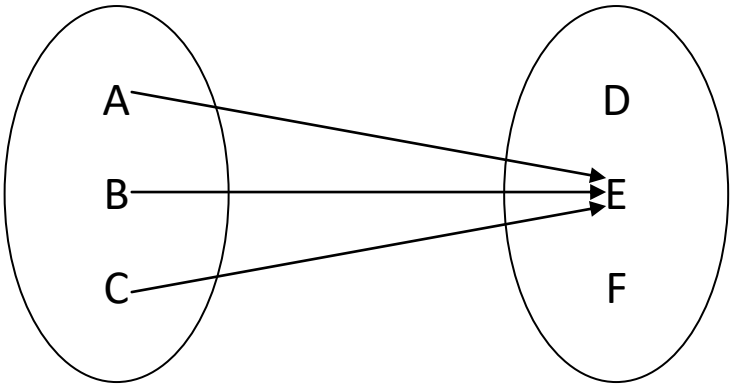

Figura 2.1

Una semplice rappresentazione della proprietà di mapping "molti a uno" da dominio a codominio di una funzione hash.

La Figura 2.1 mostra una rappresentazione della proprietà "molti ad uno" caratteristica delle funzioni di hash, ovvero dato un input di dimensioni arbitrarie sarà dato in output un hash di dimensioni fisse, più piccole di quelle in ingresso. Le funzioni di hash crittografiche al loro livello più alto possono essere classificate in due categorie:

- le funzioni di hash dette "*keyed*", ossia quelle funzioni i cui output sono dati da due parametri in input (un messaggio e la corrispondente chiave segreta);

- le funzioni di hash dette "*unkeyed*", ossia quelle funzioni il cui output è definito da un singolo parametro in input (un messaggio).

Queste due categorie sono rappresentate schematicamente nella Figura 2.2.

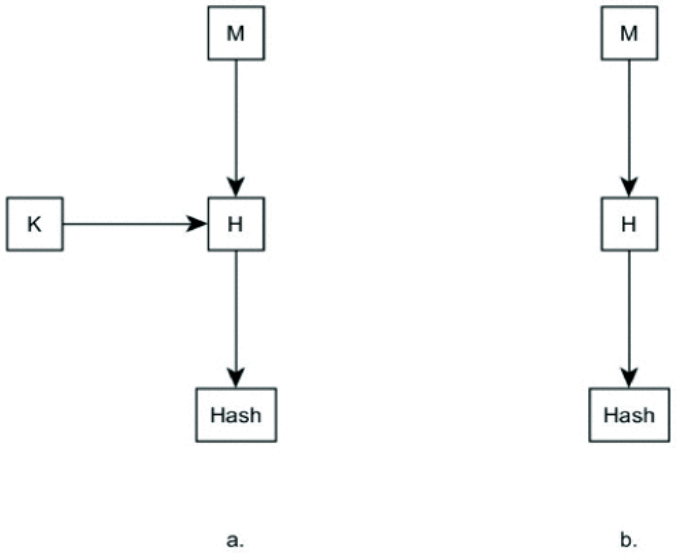

Figura 2.2

a) Una funzione di hash "*keyed*"
b) Una funzione di hash "*unkeyed*"

Le moderne funzioni crittografiche di hash "*keyed*" hanno anche le proprietà di diffusione e confusione. Ciò significa, rispettivamente, che l'algoritmo è in grado di distribuire le correlazioni statistiche del testo lungo tutto l'alfabeto utilizzato dall'algoritmo di cifratura, rendendo quanto più difficile possibile un attacco statistico (proprietà di diffusione); e che la relazione tra la chiave e il testo cifrato sarà quanto più complessa e incorrelata possibile, in modo tale che non si possa risalire alla chiave partendo dal testo cifrato (proprietà di confusione). [2]

Per semplicità in questa tesi ci si è concentrati sulle funzioni di tipo "*unkeyed*", ma si sottolinea che lo schema di firma considerato potrebbe anche essere pensato per utilizzare funzioni hash di tipo "*keyed*".

2.4 Firma digitale

La firma digitale è un corpo di informazione che si basa su dei dati firmati, sulla chiave privata, che viene utilizzata dal firmatario, e sulla chiave pubblica, che invece può essere utilizzata da chiunque per validare la firma.

Attualmente le firme digitali sono ampiamente utilizzate in vari ambiti, a partire dai certificati X.509, che vengono scaricati automaticamente quando ci si connette ad un sito, fino alla blockchain, all'e-government, all'e-commerce, ai sistemi di telecomunicazione e alle reti di computer. Nonostante ciò la ricerca e lo sviluppo di nuovi schemi di firma digitale per migliorare le prestazioni e la sicurezza degli algoritmi esistenti è ancora in corso. Nell'ultimo periodo sempre più ricercatori si stanno concentrando sullo studio degli schemi di firma post-quantum, poiché al momento il problema principale è che, a causa del post-quantum, le firme RSA non possono più essere ritenute sicure. [3, 4]

2.5 Codici lineari a blocco

Un codice a blocco è un meccanismo di codifica in cui sequenze di informazione di lunghezza fissata vengono codificate in parole di codice, anch'esse di lunghezza fissata e maggiore di quella iniziale.

La lunghezza delle parole di codice è un parametro definito *lunghezza* del codice (e normalmente indicato con n), mentre la lunghezza delle sequenze di informazione è detta *dimensione* del codice (ed è indicata con k).

Il rapporto tra k ed n viene chiamato *rate* del codice ed indicato con R, mentre il numero di cifre introdotte è normalmente chiamato *ridondanza*, e viene calcolato come $r = n - k$; siccome $k < n$ (il codice introduce ridondanza), allora si ha che $0 < R < 1$.

Se un codice a blocco è lineare, allora questo significa che la somma tra due parole di codice corrisponde sempre ad una parola di codice. È facile dimostrare che un codice a blocco lineare con valori in $\mathbb{F}_q$, lunghezza n e dimensione k può essere rappresentato, in modo compatto, da una matrice $\boldsymbol{G} \in \mathbb{F}_q^{k \times n}$ (ovvero, con k righe ed n colonne), che viene chiamata *matrice generatrice*.

In termini più espliciti, sia $\mathbb{F}_q^k$ il set di tutte le sequenze di lunghezza k e siano i valori definiti nel campo finito $\mathbb{F}_q$: allora, il codice generato da $\boldsymbol{G}$ è definito come

$$\{\boldsymbol{uG} \mid \boldsymbol{u} \in \ \mathbb{F}_q^k\},$$

ovvero, come l'insieme di tutte le possibili combinazioni lineari delle righe della matrice $\boldsymbol{G}$.

Lo stesso codice può essere rappresentato tramite la *matrice di parità*, ovvero, tramite una matrice $\boldsymbol{H} \in \mathbb{F}_q^{r \times n}$ (ovvero, con r righe ed n colonne), rango pieno e tale per cui, per ogni parola $\boldsymbol{c}$ di codice, vale $\boldsymbol{Hc}^{\mathsf{T}} = 0$, dove T indica la trasposizione.

Grazie alla matrice di parità è possibile capire se un vettore appartiene (o non appartiene) al codice, tramite il calcolo di *sindrome*. Dato un vettore $\boldsymbol{x} \in \mathbb{F}_q^n$, la sua sindrome è definita come $\boldsymbol{s} = \boldsymbol{Hx}^{\mathsf{T}}$: se $\boldsymbol{s}$ non è nullo, allora questo significa che $\boldsymbol{x}$ non è una parola di codice.

La *distanza minima* d di un codice è definita come la minima distanza di Hamming tra due parole di codice diverse o, equivalentemente, come il minimo peso di Hamming delle parole di codice (esclusa la parola nulla).

2.6 Decodifica di codici lineari

I codici lineari a blocco vengono normalmente utilizzati per la loro capacità di correzione degli errori, ovvero la possibilità di risalire alla parola di codice che era stata inizialmente trasmessa, nel caso in cui il canale di trasmissione introduce degli errori (ovvero, modifica in modo imprevedibile alcuni dei simboli della parola). Per studiare questa situazione, si supponga di voler inviare la sequenza di informazione $\boldsymbol{u}$, di lunghezza k. Utilizzando un codice a blocco con dimensione k e lunghezza n, la

sequenza viene codificata in una parola di codice $\boldsymbol{c} \in \mathbb{F}_q^n$, che viene poi trasmessa tramite il canale. Per semplicità, si considera un canale additivo, ovvero, un canale tale per cui, dato in input $\boldsymbol{c}$, restituisce in output una sequenza

$$\boldsymbol{x} = \boldsymbol{c} + \boldsymbol{e} \ ,$$

dove $\boldsymbol{e} \in \mathbb{F}_q^n$ e viene definito *vettore d'errore.*

Quando $\boldsymbol{e}$ è non nullo (ovvero, ha componenti non nulle) allora la sequenza ricevuta dal canale è diversa da quella che era stata inizialmente trasmessa. In questi casi, il codice viene utilizzato per correggere l'azione del canale, ovvero, per stimare il vettore $\boldsymbol{e}$ ed ottenere $\boldsymbol{c}$ come $\boldsymbol{x} - \boldsymbol{e}$.

Questo processo viene chiamato *decodifica* e, ad esempio, viene effettuato tramite il calcolo della sindrome. Infatti, dopo aver ricevuto $\boldsymbol{x}$, si può calcolare

$$\boldsymbol{s} = \boldsymbol{H}\boldsymbol{x}^\top = \boldsymbol{H}(\boldsymbol{c} + \boldsymbol{e})^\top = \boldsymbol{H}\boldsymbol{c}^\top + \boldsymbol{H}\boldsymbol{e}^\top = \boldsymbol{0} + \boldsymbol{H}\boldsymbol{e}^\top = \boldsymbol{H}\boldsymbol{e}^\top.$$

Si nota come la sindrome $\boldsymbol{s}$ dipende solamente dal vettore d'errore. Inoltre, quando il peso di Hamming del vettore d'errore è inferiore a $\lfloor d/2 \rfloor$ (dove d indica la distanza minima del codice), allora si può dimostrare che esiste un solo vettore d'errore con sindrome $\boldsymbol{s}$.

Pertanto, codici correttori d'errore cercano di recuperare il vettore d'errore partendo dalla sua sindrome. Esistono numerose famiglie di codici per cui questo processo può essere svolto in maniera molto efficientemente, ad esempio: codici LDPC (Low Density Parity Check, la capacità correttiva di questi codici non è garantita), codici GRS, codici di Goppa, eccetera.

Invece, per codici random, ovvero codici la cui matrice generatrice è scelta completamente a caso, si può dimostrare che il processo di decodifica è molto

complesso e non esiste alcuna procedura che possa restituire il vettore d'errore in breve tempo.

Formalmente, il problema della decodifica per un codice random viene definito come segue:

Syndrome Decoding Problem (SDP): *Sia* $\boldsymbol{H}$ *una matrice di parità scelta in modo casuale, a valori in* $\mathbb{F}_q$*, con ridondanza* r *e lunghezza* n*. Sia* $\boldsymbol{s} \in \mathbb{F}_q^r$ *e sia* t *un intero. Trovare un vettore* $\boldsymbol{e} \in \mathbb{F}_q^n$*, con peso di Hamming* $\leq t$*, tale che* $\boldsymbol{s} = \boldsymbol{H}\boldsymbol{e}^\top$.

Come già detto, è possibile dimostrare che SDP non può essere risolto in maniera efficiente da alcun algoritmo. In particolare, i migliori algoritmi per risolvere SDP sono noti come Information Set Decoding (ISD), ed hanno una complessità computazionale [14] che può essere approssimata come

$$2^{-t\log_2\left(1-\frac{k}{n}\right)}.$$

3. DESCRIZIONE DELLO SCHEMA

In questo paragrafo verranno descritti i passaggi necessari per la generazione delle chiavi pubbliche e segrete e della firma sulla base dello schema implementato, che utilizza codici random definiti su un campo finito non binario.

3.1 Key Generation

Sia q un numero primo, si lavora in un campo finito con q elementi. Per la generazione delle chiavi vengono effettuate le seguenti operazioni:

1. si sceglie una matrice di parità $\boldsymbol{H}$ con r righe ed n colonne (con $r = n - k$);
2. si genera in modo random una matrice $\boldsymbol{E}$ con ℓ righe e n colonne, in cui ogni colonna ha w_E elementi non nulli, e con valori che sono 1 oppure -1, mentre gli altri elementi sono nulli. Inoltre, $\boldsymbol{E}$ deve avere in ogni riga almeno t_E elementi non nulli e, se questa caratteristica non è rispettata, la matrice viene scartata e ne viene generata una nuova, finché non se ne trova una adeguata;
3. si calcola $\boldsymbol{S} = \boldsymbol{H}\boldsymbol{E}^{\mathsf{T}}$;
4. la chiave segreta è $\boldsymbol{E}$, mentre la chiave pubblica è costituita dalle matrici $\boldsymbol{S}$ ed $\boldsymbol{H}$.

3.2 Signature Generation

Dato un messaggio da firmare m, per la generazione della firma si utilizzerà una funzione hash che, preso in ingresso il messaggio, restituirà in uscita un vettore di lunghezza ℓ, con tutti elementi nulli meno w_C elementi che assumono solamente valore 1 oppure -1. Per generare la firma di un messaggio m sono necessari anche altri due parametri (γ e $\bar{\gamma}$), che indicano il modulo del valore massimo che delimita l'insieme dei valori che i vettori y e z possono assumere rispettivamente. Le operazioni da eseguire sono le seguenti:

1. si sceglie a caso un vettore **y** di lunghezza n, con valori scelti da $\{0, \pm1, \pm2, \pm3, \dots, \pm\gamma\}$;

2. si calcola $s_y = \boldsymbol{H}\boldsymbol{y}^\top$
3. si calcola l'hash di m concatenato con s_y; indichiamo questo hash con **c**;
4. si calcola la firma come

$$\boldsymbol{z} = \boldsymbol{c}\boldsymbol{E} + \boldsymbol{y}$$

5. la firma è valida solo se $\boldsymbol{z}$ ha i valori nell'intervallo $\{0, \pm 1, \dots, \pm\bar{\gamma}\}$;
6. si effettua il Rejection Sampling: le firme vengono scartate con una probabilità fissata a priori, questo comprende anche il "rigetto" di firme teoricamente valide sulla base del punto precedente. Viene applicato questo metodo per ridurre il rischio di avere una firma debole e quindi il rischio di essere attaccati con successo.

3.3 Signature Verification

I passaggi per verificare la firma sono i seguenti:

1. si verifica che $\boldsymbol{z}$ assuma valori in $\{0, \pm 1, \dots, \pm\bar{\gamma}\}$;
2. si calcola

$$s_y = \boldsymbol{H}\boldsymbol{z}^\top - \boldsymbol{S}\boldsymbol{c}^\top$$

3. si verifica la validità dell'hash.

Per i nostri scopi, per verificare l'hash, è sufficiente vedere se s_y è uguale a quello calcolato nella fase di calcolo della firma (ovvero, se è uguale a $\boldsymbol{H}\boldsymbol{y}^\top$).

3.4 Esempio Numerico

Come prima cosa si generano le chiavi. Scegliamo i parametri $q = 7$, $n = 5$, $r = 3$, $\ell = 4$, $w_E = 2$ e $t_E = 2$. Osserviamo che, in $\mathbb{F}_7$, si ha che -1 è uguale a 6. Supponiamo di avere le seguenti matrici $\boldsymbol{H}$ ed $\boldsymbol{E}$:

$$\boldsymbol{H} = \begin{bmatrix} 5 & 2 & 0 & 0 & 3 \\ 6 & 1 & 4 & 4 & 5 \\ 2 & 0 & 2 & 0 & 1 \end{bmatrix},$$

$$\boldsymbol{E} = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 \\ 1 & 6 & 0 & 0 & 0 \\ 6 & 0 & 6 & 0 & 0 \\ 0 & 1 & 0 & 1 & 6 \end{bmatrix}.$$

Si calcola $\boldsymbol{S}$ come

$\boldsymbol{S} = \boldsymbol{H}\boldsymbol{E}^{\top} =$

$$= \begin{bmatrix} 5 & 2 & 0 & 0 & 3 \\ 6 & 1 & 4 & 4 & 5 \\ 2 & 0 & 2 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 & 0 & 1 & 1 & 1 \\ 1 & 6 & 0 & 0 & 0 \\ 6 & 0 & 6 & 0 & 0 \\ 0 & 1 & 0 & 1 & 6 \end{bmatrix}^{\top} = \begin{bmatrix} 5 & 2 & 0 & 0 & 3 \\ 6 & 1 & 4 & 4 & 5 \\ 2 & 0 & 2 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 & 6 & 1 \\ 0 & 6 & 0 & 1 \\ 1 & 0 & 6 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 6 \end{bmatrix} =$$

$$\begin{bmatrix} 3 & 3 & 2 & 6 \\ 6 & 5 & 4 & 0 \\ 3 & 2 & 3 & 6 \end{bmatrix}.$$

Generate le chiavi, si genera la firma.

Scegliamo $\gamma = 2$, per cui gli elementi di $\boldsymbol{y}$ possono solo assumere valori in $\{0,1,2,5,6\}$. Scegliamo invece $\bar{\gamma} = 1$ e quindi gli elementi di $\boldsymbol{z}$ possono assumere solo valori in $\{0,1,6\}$. Scegliamo

$$\boldsymbol{y} = [2,5,1,6,1].$$

Calcoliamo s_y come segue

$$s_y = \boldsymbol{H}\boldsymbol{y}^{\top} = \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix}.$$

Supponiamo che l'hash sia $\boldsymbol{c} = [0,6,0,1]$, si ottiene

$$\boldsymbol{z} = \boldsymbol{c}\boldsymbol{E} + \boldsymbol{y} = [1,0,1,0,0].$$

La firma è valida, perché $\boldsymbol{z}$ assume valori in $\{0,1,6\}$, ora è necessario verificarne la correttezza.

Calcoliamo s_y come segue

$$s_y = \boldsymbol{H}\boldsymbol{z}^\top - \boldsymbol{S}\boldsymbol{c}^\top = \begin{bmatrix} 5 & 2 & 0 & 0 & 3 \\ 6 & 1 & 4 & 4 & 5 \\ 2 & 0 & 2 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} - \begin{bmatrix} 3 & 3 & 2 & 6 \\ 6 & 5 & 4 & 0 \\ 3 & 2 & 3 & 6 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 6 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix}.$$

Il valore che si ottiene è identico a quello che era stata calcolato per generare la firma, quindi la firma è corretta e viene accettata.

4. SICUREZZA E PRESTAZIONI DELLO SCHEMA

In questa sezione si andranno a valutare alcune prestazioni dello schema, come dimensioni delle chiavi e delle firme e numero di chiavi e firme scartate (in media) ad ogni esecuzione dell'algoritmo di signature generation.
Il calcolo del numero delle chiavi e delle firme scartate non va ad alterare il livello di sicurezza dello schema, ma è utile per definire l'efficienza dello schema.
Inoltre, si andranno a definire le complessità degli attacchi, in modo da fornire criteri tramite i quali parametri sicuri possono essere selezionati.

4.1 Dimensioni di chiavi e firme

La chiave pubblica nello schema è rappresentata dalla matrice $\boldsymbol{S}$, a valori in $\mathbb{F}_q$ e con $n-k$ righe e ℓ colonne. Siccome ogni elemento di $\mathbb{F}_q$ può essere rappresentato con $\lceil \log_2 q \rceil$ bit, questo significa che la dimensione della chiave pubblica (in bit) è data da

$$\ell(n-k)\lceil \log_2 q \rceil.$$

La firma è composta dal vettore $\boldsymbol{z}$ e dall'hash $\boldsymbol{c}$: entrambi questi vettori possono essere rappresentati in maniera efficiente, in modo da risparmiare dei bit e ridurre così la dimensione della firma.
Infatti $\boldsymbol{z}$ è un vettore i cui valori appartengono all'intervallo $\{0, \pm 1, \dots, \pm\bar{\gamma}\}$, pertanto esistono in un sottoinsieme del campo finito $\mathbb{F}_q$. In totale ogni elemento di $\boldsymbol{z}$ può avere $2\bar{\gamma}+1$ valori, perciò per ognuno dei suoi elementi si può utilizzare una rappresentazione che sfrutta non più di $\lceil \log_2(2\bar{\gamma}+1) \rceil$ bit. Poiché $\boldsymbol{z}$ ha lunghezza n, in totale per la sua rappresentazione si può utilizzare un numero di bit pari a

$$n\lceil \log_2(2\bar{\gamma}+1) \rceil.$$

Per quanto riguarda $\boldsymbol{c}$, si deve considerare che è un vettore con lunghezza ℓ ed i cui elementi possono avere solamente tre valori, ovvero, 0 oppure ± 1. Seguendo il ragionamento precedente, si potrebbero utilizzare $\ell\lceil \log_2(3) \rceil$ bit per la sua rappresentazione. In realtà, in questo caso si possono guadagnare altri bit, considerando che $\boldsymbol{c}$ è anche sparso, ovvero, contiene un elevato numero di zeri e solamente $w_c \ll \ell$ elementi non nulli. Pertanto, una sua rappresentazione efficiente potrebbe essere ottenuta descrivendo solamente gli elementi non nulli: per tutti gli altri elementi, si sa già che essi sono uguali a 0.

Per descrivere ognuno di questi w_C elementi non nulli, è sufficiente indicare in quale posizione si trova il generico elemento non nullo e se assume valore $+1$ o -1. La posizione è data da un intero nel range $[0\,;\ell - 1]$, pertanto servono $\lceil \log_2(\ell) \rceil$ bit per rappresentarla, mentre per capire se il valore è 1 oppure -1, basta un ulteriore bit aggiuntivo. Siccome ci sono w_c elementi non nulli, alla fine si ottiene che $\boldsymbol{c}$ può essere rappresentato con un numero di bit dato da

$$w_C(1 + \lceil \log_2(\ell) \rceil).$$

Alla fine, pertanto, si ottiene che la dimensione della firma (in bit) è data da

$$n\lceil \log_2(2\bar{\gamma} + 1) \rceil + w_C(1 + \lceil \log_2(\ell) \rceil).$$

4.2 Numero di chiavi scartate

Come già detto, nell'algoritmo di key generation viene eseguito un controllo sulla chiave segreta per garantire che essa possegga determinate proprietà (ovvero, che ciascuna riga abbia un numero di elementi non nulli non più basso di un prefissato t_E).

Nel caso in cui il controllo non fosse soddisfatto, la matrice $\boldsymbol{E}$ verrebbe scartata e ne verrebbe generata un'altra in modo casuale: il processo si ripete finché una matrice valida non viene ottenuta.

In particolare, il numero di matrici scartate in media ad ogni esecuzione dell'algoritmo di key generation può essere ottenuto come:

$$N_{key} = \left(1 - \sum_{i=0}^{t_E-1} \binom{n}{i} \left(\frac{w_E}{\ell}\right)^i \left(1 - \frac{w_E}{\ell}\right)^{n-i}\right)^{-\ell} = \frac{1}{\epsilon^\ell} \ .$$

La formula può essere spiegata nel seguente modo. Sia ϵ la probabilità che una riga di $\boldsymbol{E}$ contenga almeno t_E elementi non nulli. Siccome la matrice $\boldsymbol{E}$ contiene ℓ righe, e si vuole che il criterio sia soddisfatto per tutte queste righe, allora si può approssimare con ϵ^ℓ la probabilità che ogni riga della matrice soddisfi questo vincolo. Pertanto, l'inverso di questa probabilità corrisponde al numero medio di chiavi che devono essere generate, prima di ottenerne una valida.

4.3 Numero di firme scartate

Così come per l'algoritmo di key generation, anche in quello di signature generation viene applicata una tecnica di tipo rejection sampling, per garantire che la firma rispetti alcune proprietà statistiche (necessarie per garantirne la sicurezza rispetto ad alcuni attacchi statistici). Senza entrare nel dettaglio di come questa operazione viene eseguita, è sufficiente dire ciascuna firma viene scartata con una probabilità che dipende dagli elementi di $\boldsymbol{z}$. In particolare, la probabilità con cui in media un vettore $\boldsymbol{z}$ viene scartato è ottenuta come

$$\delta = \frac{1}{M^n} \prod_{i=0}^{n-1} \frac{f(z_i)}{g(z_i)}$$

dove:

- z_i indica la componente di $\boldsymbol{z}$ in posizione i;
- la funzione f è definita come segue

$$f(x) = \begin{cases} \dfrac{1}{2\bar{\gamma}+1} & se\ x \in \{0, \pm 1, \dots, \pm\bar{\gamma}\} \\ 0 & se\ x \notin \{0, \pm 1, \dots, \pm\bar{\gamma}\} \end{cases}$$

- la funzione g è calcolata come segue

$$g(x) = \frac{\sum_{i=-\gamma}^{\gamma} u(x-i)}{2\gamma+1}$$

con u che a sua volta è un'altra funzione, definita nel seguente modo. In input x, essa dapprima calcola $x_q = \min\{x\ ,\ q-x\}$, e poi esegue il seguente calcolo:

$$u(x) = \sum_{\substack{v=x_q \\ v\ e\ x_q\ hanno\ la\ stessa\ parità}}^{\min\{w_E, w_C\}} 2^{-v} \frac{\binom{v}{\frac{v+x_q}{2}} \binom{w_E}{v} \binom{\ell - w_E}{w_C - v}}{\binom{\ell}{w_C}}$$

(dire che v e x_q hanno la stessa parità, equivale a dire che se x_qè pari, allora v è pari. Se invece è dispari, allora anche v è dispari);

- il valore di M è definito come

$$M = \max_{x \in \mathbb{F}_q} \left\{ \frac{f(x)}{g(x)} \right\}.$$

L'inverso della probabilità δ corrisponde al numero di firme che, in media, vengono scartate ad ogni esecuzione dell'algoritmo di signature generation. Pertanto, il numero di firme che in media vengono scartate, prima di ottenerne una valida, è ottenuto come

$$N_{sig} = \frac{1}{\delta}.$$

4.4 Attacchi

In questa sezione verranno brevemente descritti gli attacchi allo schema; in particolare, verranno presentate le formule per la loro complessità computazionale, che dipendono dai vari parametri dello schema. Volendo garantire un livello di sicurezza pari a λ bit, si deve garantire che la complessità di ogni attacco sia almeno 2^{λ}. In particolare, essendo uno schema di firma, ci sono diverse tipologie di attacco, a seconda dell'obiettivo dell'attaccante (ad esempio, rubare la chiave segreta o falsificare una firma).

In questa tesi, sono stati considerati i seguenti attacchi:

- **replicazione della firma:** in questo tipo di attacco, un attaccante parte da una firma $(\boldsymbol{z}, \boldsymbol{c})$ valida per un messaggio m. A questo punto, cerca di trovare un altro messaggio $m' \neq m$, tale per cui l'hash di $m' || \boldsymbol{s_y}$ è uguale a $\boldsymbol{c}$. Per un messaggio di questo tipo, infatti, si avrebbe la stessa firma $(\boldsymbol{z}, \boldsymbol{c})$: pertanto, l'attaccante sarebbe in grado di fornire una firma valida per m' anche senza

conoscere la chiave segreta. Per evitare attacchi di questo tipo, si deve garantire che la funzione di hash sia resistente a procedure che trovano collisioni: volendo garantire un livello di sicurezza pari a λ, serve che il numero di possibili output della funzione di hash sia non più piccolo di $2^{2\lambda}$. Nel caso in esame la funzione di hash produce vettori di lunghezza ℓ, peso w_C ed elementi non nulli con solamente due valori possibili (ovvero ± 1). Pertanto il numero di possibili output della funzione di hash è dato da $\binom{\ell}{w_C} 2^{w_C}$. Volendo garantire una sicurezza pari a λ bit, è pertanto necessario che sia soddisfatta la seguente relazione

$$\binom{\ell}{w_C} 2^{w_C} > 2^{2\lambda};$$

- **recupero della chiave segreta:** data la chiave pubblica, un attaccante potrebbe provare a recuperare la chiave segreta, ovvero la matrice $\boldsymbol{E}$. Considerando che ogni riga di $\boldsymbol{E}$ è un vettore di basso peso di Hamming, e che ogni colonna della chiave pubblica $\boldsymbol{S}$ è la sindrome di una riga di $\boldsymbol{E}$, alla fine si ha che per recuperare righe della chiave segreta si deve andare a risolvere un problema di decodifica. In particolare, il peso del vettore d'errore che si deve determinare è non più basso di t_E ed il codice che si sta cercando di decodificare è quello avente $\boldsymbol{H}$ come matrice di parità, quindi ha lunghezza n e dimensione k. Perciò, per la complessità di questo attacco, si può considerare un lower bound alla complessità pari a $2^{-t_E \log\left(1-\frac{k}{n}\right)}$. Per garantire un livello di sicurezza pari a λ bit, deve pertanto essere

$$2^{-t_E \log\left(1-\frac{k}{n}\right)} > 2^{\lambda};$$

- **contraffazione di una firma:** partendo dal messaggio m che si vuole firmare, un attaccante potrebbe provare a generare una firma $(\boldsymbol{z}, \boldsymbol{c})$ valida, pur senza conoscere la chiave segreta. Per poterlo fare, è sufficiente trovare un vettore $\boldsymbol{z}$ che soddisfi la relazione su $\boldsymbol{s_y}$ e $\boldsymbol{c}$ (basata sulla funzione di hash), con componenti a valori in $\{0, \pm 1, \dots, \pm\bar{\gamma}\}$. La complessità di un attacco di questo tipo è ancora oggetto di studio ma, come prima stima grossolana, si può utilizzare $(2\bar{\gamma})^{n/2}$. Pertanto, per proteggersi anche da questo attacco, deve essere

$$(2\bar{\gamma})^{n/2} > 2^{\lambda}.$$

5. FUNZIONI E SCRIPT

Dopo un attento studio sul funzionamento dell'ambiente di programmazione SageMath (vedi [5]), sono state implementate le funzioni e gli script seguenti per effettuare la verifica della correttezza dei dati preesistenti (ossia i risultati del lavoro descritto in [1]) e successivamente l'ottimizzazione dello schema.

Nella tabella 5.1 sono riportate le denominazioni utilizzate negli script per i parametri introdotti nel testo.

Denominazione nel testo	**Denominazione nello script**
n	n
k	k
w_C	wc
w_E	we
t_E	te
γ	gamma
$\bar{\gamma}$	gamma_barra
ℓ	ell

Tabella 5.1

Denominazioni dei parametri introdotti nel testo con le rispettive denominazioni utilizzate negli script

Di seguito sono descritti i tre script realizzati in questo lavoro di tesi.

1. Si è implementato uno script SageMath nel quale è stata definita una funzione per ognuna delle tre categorie di attacco allo schema.

```
def log2(x):
    return log(x*1.)/log(2.);

### attacco 1: replicare la firma

def is_secure(ell,wc):
    ell = binomial(ell,wc)*2^wc;
    max = 2^256;
    if ell>=max:
        return "Attacco 1 secure!";
    if ell<max:
        return "Attacco 1 not secure!";

### attacco 2: recuperare la chiave segreta

def is_secure2(te,k,n):
     esp = te*log2(1-k/n);
     l = 2^(-esp);
     if l>=(2^128):
         return ("Attacco 2 secure!");
     if l<(2^128):
              return("Attacco 2 not secure!");

### attacco 3: falsificare la firma

def is_secure3(gamma_barra,n):
     l = (n/2)*log2(2*gamma_barra);
     if l>= 128:
         return "Attacco 3 secure!";
     if l< 128 :
         return "Attacco 3 not secure!";

print(is_secure(240,63));
print(is_secure2(81,400,600));
print(is_secure3(5329,600));
```

Script SageMath in cui, solo per riportarne un esempio, sono stati dati in ingresso alle funzioni i valori della quinta riga della tabella

2. Si è implementato uno script SageMath, in cui sono state definite due funzioni, una per il calcolo della dimensione della firma digitale e una per il calcolo della dimensione della chiave pubblica.

```
def log2(x):
    return log(x*1.)/log(2.);

def pub_key(ell,n,k,q):
    r = n-k;
    return ((ceil(ell*r*log2(q)))/8000)*1.;

def firma(gamma_barra,n,wc):
    return
((ceil(n*log2(1+2*gamma_barra))*1.+wc+wc*ceil(log2(n))*1.)/8000)*1
.;

print("The public key dimension is ");
print(pub_key(218,400,300,16381));
print("The digital firm dimension is ");
print(firma(3375,400,67));
```

Script per calcolare le dimensioni della chiave pubblica e della firma (solo per riportarne un esempio alle funzioni sono stati dati in ingresso i valori della prima riga della tabella)

3. Si è implementato uno script SageMath che stampa a video la dimensione della chiave pubblica e della firma sulla base dei valori ottimali di n, k, w_C, w_E, t_E, ℓ, γ, $\bar{\gamma}$ presi nel range di valori che viene definito nello script.

   Prima di illustrare la procedura utilizzata per il dimensionamento dello schema, è bene introdurre alcune quantità che saranno richiamate in avanti:

   - *code_rate* : rate del codice avente come matrice di parità $\boldsymbol{H}$, ovvero, rapporto tra k ed n;
   - *num_rows_ratio* : rapporto tra numero di righe della chiave segreta e numero di colonne, ovvero, ℓ/n;
   - *max_key_samples* : numero massimo dei campioni della chiave pubblica prodotti;
   - *max_sig_samples* : numero massimo dei campioni della firma prodotti.

Il funzionamento di questo script è stato descritto tramite un diagramma di flusso riportato in Figura 5.1:

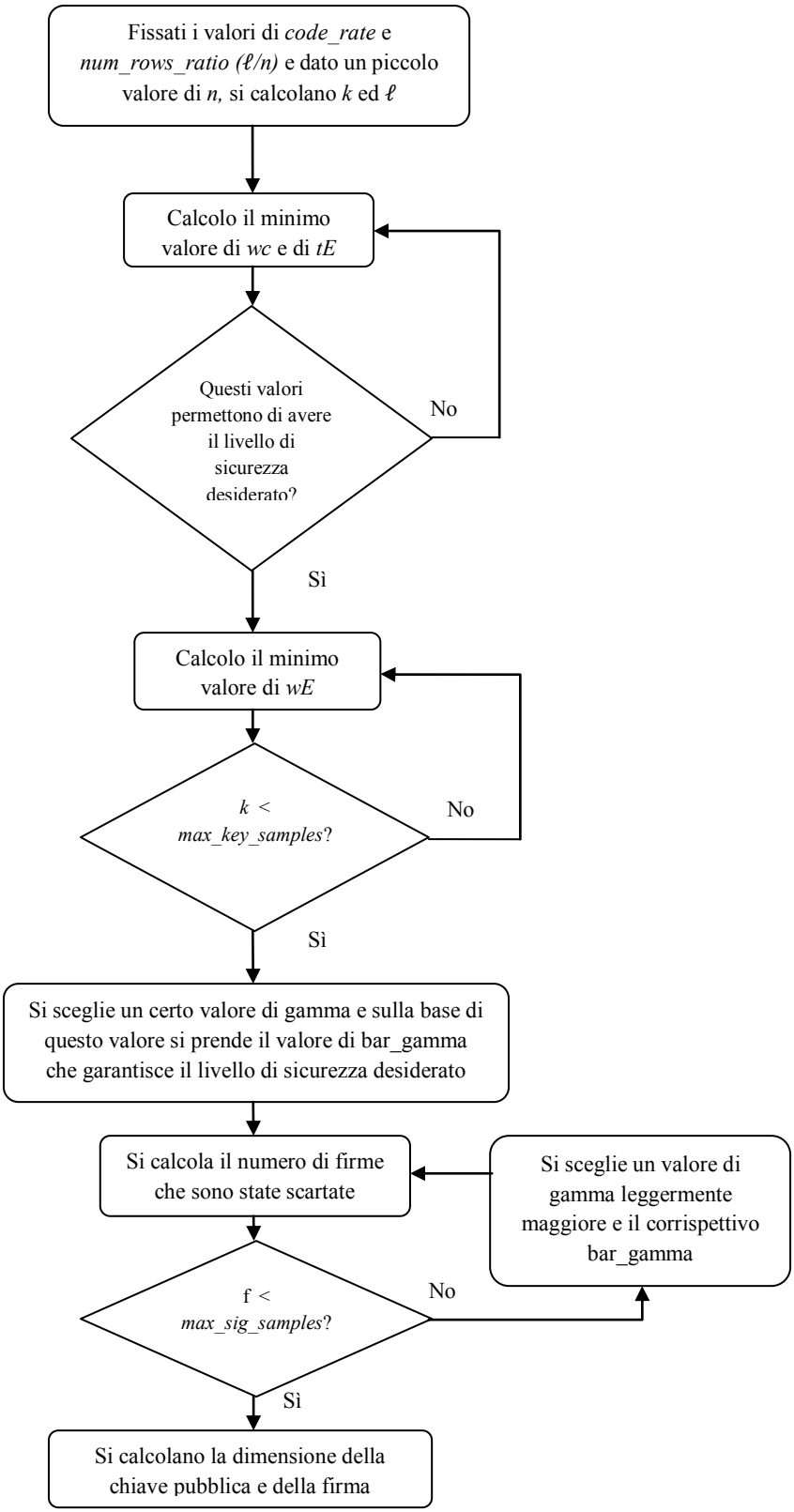

Figura 5.1
Descrizione grafica del funzionamento dello Script 3

a) Dopo aver fissato *code_rate* e *num_rows_ratio* (ℓ/n) e aver scelto un piccolo valore di n, vengono calcolati i valori di k e ℓ.
b) Dopo di che si calcolano i minimi valori di w_C e t_E in modo tale che sia garantito il livello di sicurezza desiderato e il minimo valore di w_E che permetta di scartare in media un numero di chiavi (indicato con k nel diagramma di flusso) inferiore al numero massimo inizialmente fissato (*max_key_samples*).
c) Fissato un certo γ, sulla base di questo si definisce il valore di $\bar{\gamma}$ che garantisce il livello di sicurezza desiderato.
d) Si procede calcolando il numero di firme scartate (indicato nel diagramma di flusso con *f*): se questo è inferiore al numero massimo di firme definito inizialmente (*max_sig_samples*) allora vengono calcolate le dimensioni della chiave pubblica e della firma digitale. Se questo non si verifica viene fissato un nuovo valore di γ e si ripetono i punti *c* e *d* fintanto che non si ottiene un numero di firme scartate adeguato.

Lo script completo è riportato nell'Appendice A.

6. VERIFICA CORRETTEZZA RISULTATI PRECEDENTI

Dopo aver letto e analizzato i contenuti dell'articolo [1], vista l'eccellenza dei risultati ottenuti in [1] (vedi Figura 6.1) si è ritenuto necessario verificarne la correttezza e la ragionevolezza sulla base del livello di sicurezza atteso (128 bit), così da avere una riconferma della loro validità.

Per effettuare questa verifica è stata di fondamentale importanza l'implementazione di due script in SageMath (vedi script 1 e 2 nella sezione "Funzioni e Script"), poiché hanno permesso di rendere più efficiente la valutazione dei dati e quindi, ridotto l'onere computazionale, di procedere ad un'analisi parametrica, che, seppur in termini non assoluti, ha consentito di individuare soluzioni ottime.

(n, k, q)	b	w_E	w_c	t_E	γ	$\bar{\gamma}$	Avg rejections	$\lvert\sigma\rvert$ in kB	$\lvert\mathsf{pk}\rvert$ in kB
$(400, 300, 16381)$	218	46	67	64	3420	3375	199.80	0.72	38.15
$(500, 375, 16381)$	250	42	61	64	3890	3849	199.78	0.88	54.69
$(400, 320, 16381)$	240	45	63	56	3460	3417	148.64	0.72	33.60
$(500, 375, 32749)$	260	44	60	64	4600	4559	87.88	0.90	65.99
$(600, 400, 32749)$	240	42	63	81	5370	5329	99.30	1.09	89.95

Figura 6.1

Tabella 1 importata dall'articolo [1]

Innanzitutto il sistema con i dati specificati in tabella è stato sottoposto a tre diverse categorie di attacchi: gli attacchi che cercano di recuperare la chiave segreta, gli attacchi che cercano di falsificare la firma e gli attacchi che cercano di replicare la firma per un altro messaggio.

Per fare ciò è stato caricato per cinque volte lo script 1 (vedi sezione "Funzioni e Script") dando in ingresso alle funzioni di volta in volta i valori presenti su una delle righe della Tabella riportata in Figura 6.1. È risultato che tutti valori riportati nella tabella permettevano di raggiungere un livello di sicurezza maggiore o uguale di quello atteso.

Successivamente è stata verificata la correttezza delle dimensioni della chiave pubblica e della firma digitale riportate sulla tabella. In questo caso è stato caricato per cinque volte lo script 2 (vedi sezione "Funzioni e Script") dando in ingresso alle funzioni i valori presenti su ognuna delle cinque righe della tabella.

Si è osservato che tutti i valori delle firme digitali erano corretti, mentre tra le cinque dimensioni delle chiavi pubbliche sono risultate corrette solo quelle riportate nella prima, nella seconda e nella terza riga della tabella.

Il valore riportato nella quarta riga è risultato essere più grande del valore effettivo e questo è positivo, poiché si ricerca la minore dimensione possibile per le chiavi pubbliche. Invece per quanto riguarda il valore riportato nella quinta riga è stata verificata la presenza di un errore di battitura, questo, infatti, risulta essere più piccolo di 0.045 del valore effettivo. In ogni caso, i risultati ottenuti sono conformi con quelli riportati in [1], pertanto ciò conferma la validità di essi e allo stesso tempo la validità delle formule che sono state implementate in SageMath.

7. OTTIMIZZAZIONE DELLO SCHEMA

Per questa fase del lavoro è stato caricato più volte lo script, che è descritto nel punto 3 della sezione “Funzioni e Script”.

Prima di tutto si è puntato all’ottimizzazione del numero dei campioni della firma e della chiave pubblica (*max_sig_samples* e *max_key_samples*). Il valore di queste due variabili è stato fatto variare da 100 a 1000 con un passo di 100 per osservare le variazioni di *gamma*, *bar_gamma* e delle dimensioni della firma e della chiave pubblica.

In seguito per motivi di efficienza algoritmica le variabili *max_sig_samples* e *max_key_samples* sono state fissate rispettivamente a 500 e a 1000, infatti la key generation viene eseguita una sola volta, al contrario della signature generation che viene eseguita parecchie volte; si pensi che in media l'algoritmo di generazione della firma viene ripetuto esattamente un numero di volte pari a *max_sig_samples*.

Visto che il numero di firme scartate è direttamente proporzionale al tempo di calcolo della firma, se *max_sig_samples* venisse preso troppo grande lo schema sarebbe troppo lento. Con *max_sig_samples* uguale a 500, ci si aspetta di avere un trade-off ragionevole tra prestazioni dello schema (dimensioni delle chiavi e firme) ed efficienza di calcolo.

Per studiare l’efficienza dello schema si è osservato il comportamento di quest’ultimo sia ad un livello di sicurezza pari a 128 che ad un livello di sicurezza pari a 192.

7.1 Livello di Sicurezza 128

Fissati *max_sig_samples* e *max_key_samples* rispettivamente a 500 e a 1000, il parametro *num_rows_ratio* è stato fatto variare tra da 0.3 e 0.9 ponendo *code_rate* rispettivamente a uguale a 0.7, 0.8, 0.9 e 0.95.

Da questa analisi è emerso che al crescere di *num_rows_ratio* le dimensioni della chiave pubblica e della firma diminuiscono, inoltre queste raggiungono il loro minimo

quando *code_rate* è fissato a 0.95. Al contrario i valori di *gamma* e *bar_gamma* restano più o meno costanti.

Nelle Tabelle 7.1, 7.2, 7.3, 7.4 sono riportati i valori trovati con queste prove (calcolati in kB) per un livello di sicurezza pari a 128 bit e nelle Figure 7.1, 7.2, 7.3, 7.4, 7.5 sono rappresentati i grafici che descrivono la variazione di questi parametri.

num_rows_ratio	pk_size	signature	(n,k,q)	(w_C,w_E,t_E)	ℓ	γ	$\bar{\gamma}$
0,3	47,642875	0,980375	(550,385,16381)	(102,27,74)	165	2410	2383
0,4	35,300375	0,762000	(410,287,16381)	(109,36,74)	164	2400	2364
0,5	28,585750	0,629875	(330,231,16381)	(102,44,74)	165	2360	2316
0,6	24,695625	0,544875	(280,196,16381)	(94,53,74)	168	2420	2367
0,7	21,167625	0,471375	(240,168,16381)	(94,61,74)	168	2390	2329
0,8	18,521750	0,425250	(210,147,16381)	(94,69,74)	168	2370	2301
0,9	17,057000	0,390625	(190,133,16381)	(90,77,74)	171	2400	2323

Tabella 7.1

Tabella con i dati ottenuti caricando lo script con *code_rate* fisso a 0.7 e facendo variare *num_rows_ratio* tra 0.3 e 0.9

num_rows_ratio	pk_size	signature	(n,k,q)	(w_C,w_E,t_E)	ℓ	γ	$\bar{\gamma}$
0,3	31,762000	0,955250	(550,440,16381)	(102,21,56)	165	1870	1849
0,4	23,533625	0,743500	(410,328,16381)	(109,28,56)	164	1870	1842
0,5	19,057250	0,616375	(330,264,16381)	(102,35,56)	165	1880	1845
0,6	16,463750	0,531875	(280,224,16381)	(94,41,56)	168	1870	1829
0,7	14,111750	0,461000	(240,192,16381)	(94,48,56)	168	1880	1832
0,8	12,347875	0,416000	(210,168,16381)	(94,54,56)	168	1860	1806
0,9	11,371375	0,382625	(190,152,16381)	(90,61,56)	171	1900	1839

Tabella 7.2

Tabella con i dati ottenuti caricando lo script con *code_rate* fisso a 0.8 e facendo variare *num_rows_ratio* tra 0.3 e 0.9

num_rows_ratio	pk_size	signature	(n,k,q)	(w_C,w_E,t_E)	ℓ	γ	$\bar{\gamma}$
0,3	15,881000	0,922125	(550,495,16381)	(102,15,39)	165	1340	1325
0,4	11,766875	0,718250	(410,369,16381)	(109,20,39)	164	1330	1310
0,5	9,528625	0,596250	(330,290,16381)	(102,25,39)	165	1340	1315
0,6	8,231875	0,516125	(280,252,16381)	(94,30,39)	168	1370	1340
0,7	7,055875	0,447250	(240,216,16381)	(94,35,39)	168	1370	1335
0,8	6,174000	0,404750	(210,189,16381)	(94,40,39)	168	1380	1340
0,9	5,685750	0,372125	(190,171,16381)	(90,45,39)	171	1400	1355

Tabella 7.3

Tabella con i dati ottenuti caricando lo script con *code_rate* fisso a 0.9 e facendo variare *num_rows_ratio* tra 0.3 e 0.9

num_rows_ratio	pk_size	signature	(n,k,q)	(w_C,w_E,t_E)	ℓ	γ	$\bar{\gamma}$
0,3	7,796125	0,899875	(550,523,16381)	(102,12,39)	165	1070	1058
0,4	5,740000	0,702250	(410,390,16381)	(109,16,39)	164	1070	1054
0,5	4,620000	0,583375	(330,314,16381)	(102,20,39)	165	1080	1060
0,6	4,116000	0,505125	(280,266,16381)	(94,24,39)	168	1100	1076
0,7	3,528000	0,437875	(240,228,16381)	(94,28,39)	168	1100	1072
0,8	2,940000	0,396125	(210,200,16381)	(94,32,39)	168	1100	1068
0,9	2,693250	0,364500	(190,181,16381)	(90,36,39)	171	1120	1084

Tabella 7.4

Tabella con i dati ottenuti caricando lo script con *code_rate* fisso a 0.95 e facendo variare *num_rows_ratio* tra 0.3 e 0.9

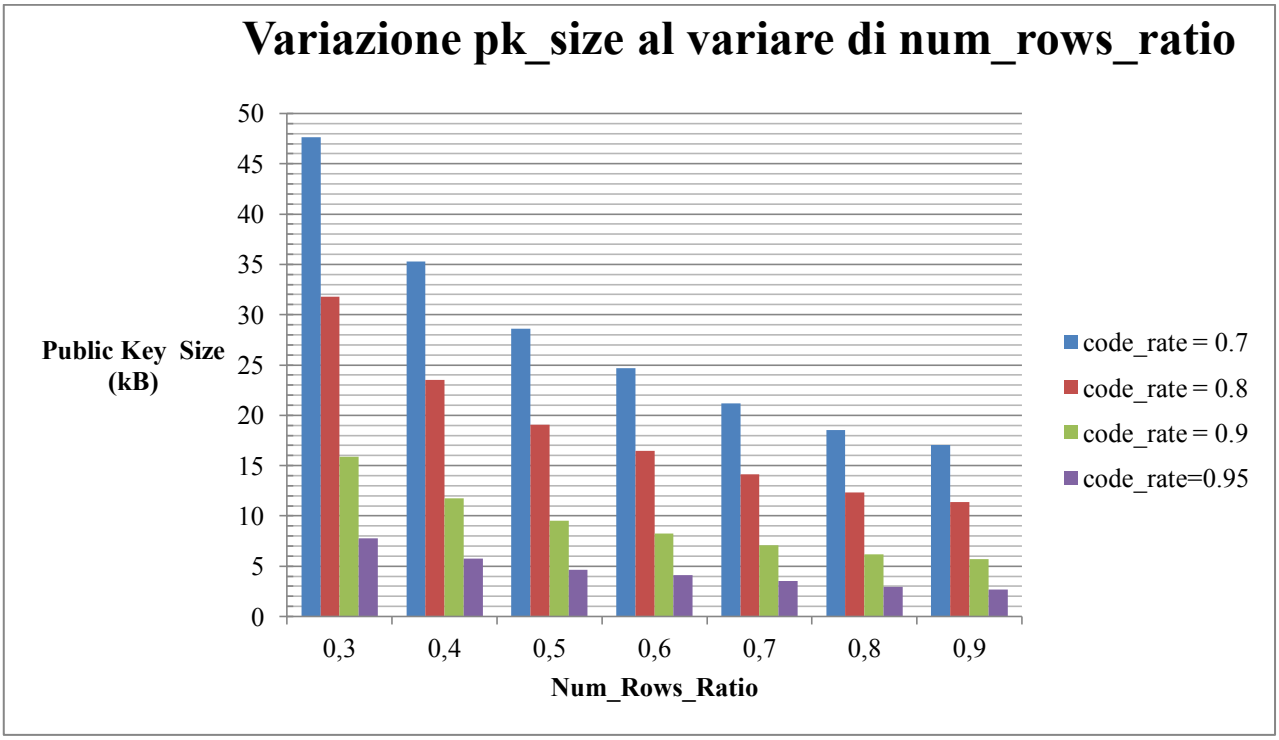

Figura 7.1

Variazione della dimensione della chiave pubblica (*pk_size*) al variare di *code_rate* e di *num_rows_ratio*

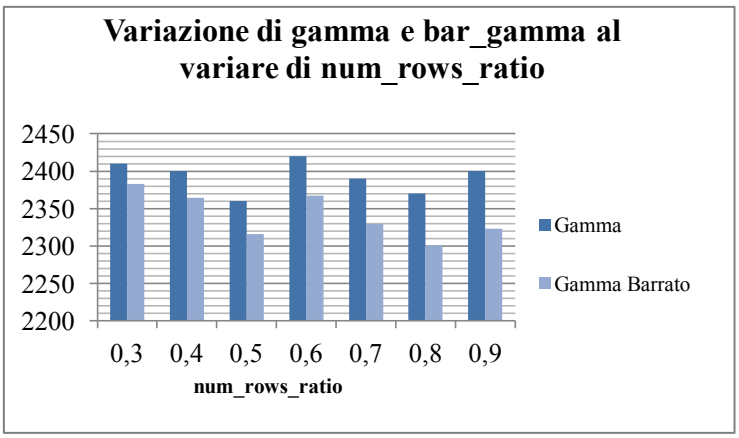

Figura 7.2

Variazione di *gamma* e *bar_gamma* al variare di *num_rows_ratio* con *code_rate* = 0.7

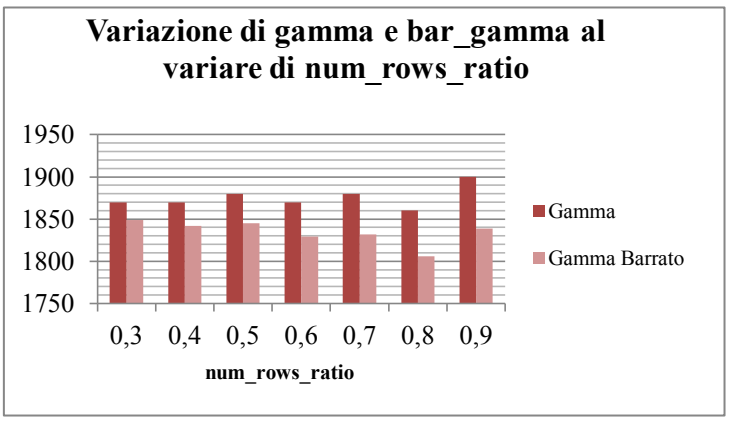

Figura 7.3

Variazione di *gamma* e *bar_gamma* al variare di *num_rows_ratio* con *code_rate* = 0.8

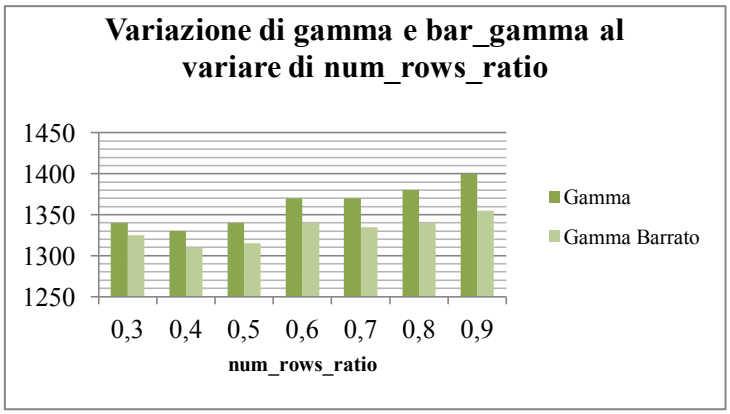

Figura 7.4

Variazione di *gamma* e *bar_gamma* al variare di *num_rows_ratio* con *code_rate* = 0.9

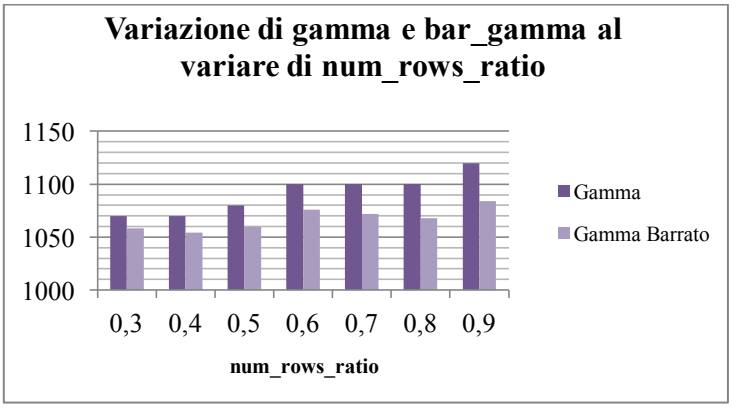

Figura 7.5

Variazione di *gamma* e *bar_gamma* al variare di *num_rows_ratio* con *code_rate* = 0.95

Dalla Figura 7.1 e dai valori riportati nelle Tabelle 7.1, 7.2, 7.3, 7.4 si può osservare che le chiavi più piccole sono state ottenute con i valori di code_rate più elevati, mentre le dimensioni delle firme restano pressoché costanti al variare del rate. Inoltre la firma è sempre molto più piccola della chiave, pertanto il parametro più importante da guardare è proprio la chiave pubblica.
Dalle Figure 7.2, 7.3, 7.4, 7.5 risulta invece evidente che fissato il rate i valori di *gamma* e *gamma barrato* rimangono più o meno costanti, mentre al crescere del rate tali valori si riducono.

7.2 Livello di Sicurezza 192

Applicando lo stesso ragionamento utilizzato per il livello di sicurezza più basso (ossia fissando *max_sig_samples* e *max_key_samples* rispettivamente a 500 e a 1000, *code_rate* a 0.7, 0.8, 0.9 e 0.95 e facendo variare *num_rows_ratio* da 0.3 a 0.9) sono state fatte delle prove con un valore di sicurezza pari a 192 bit per vedere come variavano le dimensioni di chiave pubblica e firma.
Si è osservato che per *code_rate* pari a 0.7 e per *code_rate* pari a 0.8 (quando il parametro *num_rows_ratio* era superiore a 0.5) lo script non produce alcuna uscita, per cui si può concludere che, con i vincoli assegnati, non si hanno soluzioni.

Nelle tabelle 7.5, 7.6, 7.7 sono riportati i valori dei parametri ottenuti a questo livello di sicurezza (calcolati sempre in kB) e nelle figure 7.6, 7.7, 7.8 sono rappresentati i grafici che descrivono l'andamento della variazione dei parametri.

num_rows_ratio	pk_size	signature	(n,k,q)	(w_C,w_E,t_E)	ℓ	γ	$\bar{\gamma}$
0,3	17,650250	1,462125	(820,779,16381)	(153,18,45)	246	2390	2372
0,4	13,453750	1,147875	(620,589,16381)	(146,24,45)	248	2410	2386
0,5	10,289875	0,945875	(490,466,16381)	(161,29,45)	245	2310	2281
0,6	8,609875	0,814750	(410,390,16381)	(153,35,45)	153	2330	2295
0,7	7,288625	0,732000	(350,333,16381)	(161,40,44)	245	2280	2240
0,8	6,510000	0,652250	(310,295,16381)	(146,45,44)	248	2270	2225
0,9	6,174000	0,598875	(280,266,16381)	(138,52,45)	252	2370	2318

Tabella 7.5

Tabella con i dati ottenuti caricando lo script con *code_rate* fisso a 0.95 e facendo variare *num_rows_ratio* tra 0.3 e 0.9

num_rows_ratio	pk_size	signature	(n,k,q)	(w_C,w_E,t_E)	ℓ	γ	$\bar{\gamma}$
0,3	42,524250	1,587750	(900,810,16381)	(121,22,58)	270	3200	3178
0,4	34,299375	1,271125	(700,630,16381)	(115,30,58)	280	3400	3370
0,5	21,874625	0,960875	(500,450,16381)	(142,37,58)	250	3000	2963
0,6	26,249625	0,933625	(500,450,16381)	(107,44,58)	300	3600	3556
0,7	19,599750	0,779250	(400,360,16381)	(115,51,58)	280	3400	3349
0,8	22,399625	0,771625	(400,360,16381)	(101,59,58)	320	3900	3841
0,9	14,174750	0,624375	(300,270,16381)	(121,65,58)	270	3200	3135

Tabella 7.6

Tabella con i dati ottenuti caricando lo script con *code_rate* fisso a 0.9 e facendo variare *num_rows_ratio* tra 0.3 e 0.9

num_rows_ratio	pk_size	signature	(n,k,q)	(w_C,w_E,t_E)	ℓ	γ	$\bar{\gamma}$
0,3	85,048500	1,646750	(900,720,16381)	(121,31,83)	270	4600	4569
0,4	68,598750	1,312000	(700,560,16381)	(115,41,83)	280	4700	4659
0,5	43,749250	0,989000	(500,400,16381)	(142,50,83)	250	4100	4050

Tabella 7.7

Tabella con i dati ottenuti caricando lo script con *code_rate* fisso a 0.8 e facendo variare *num_rows_ratio* tra 0.3 e 0.9

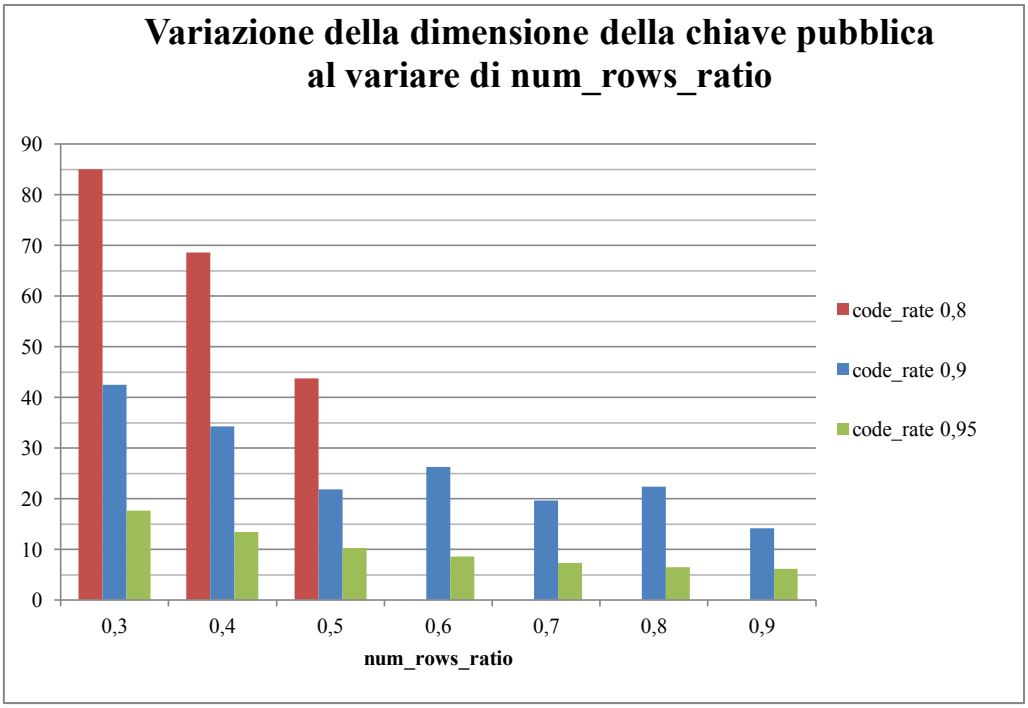

Figura 7.6

Variazione della dimensione della chiave pubblica (*pk_size*) al variare di *code_rate* e di *num_rows_ratio*

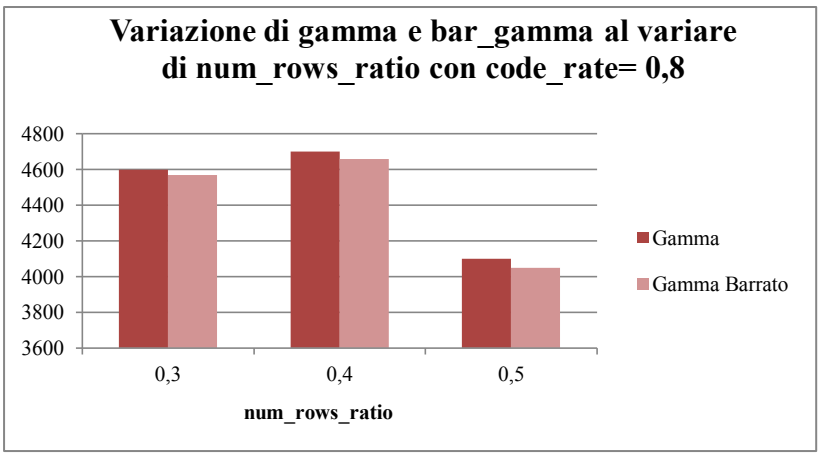

Figura 7.7

Variazione di *gamma* e *bar_gamma* al variare di *num_rows_ratio* con *code_rate* = 0.8

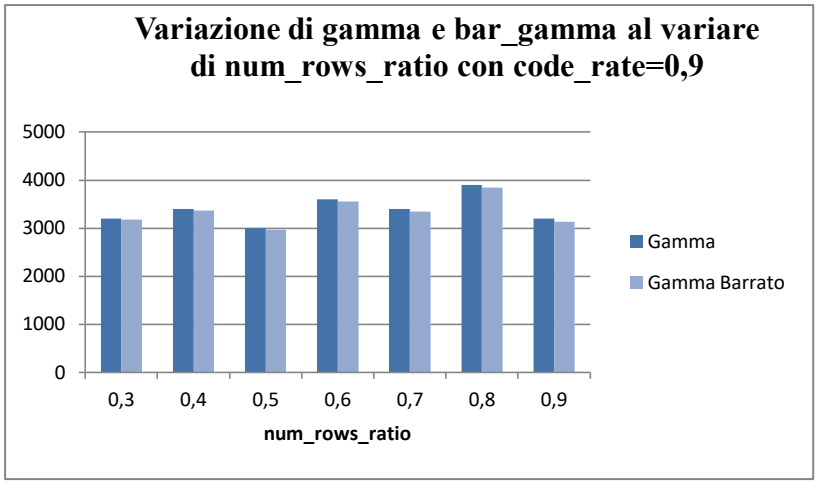

Figura 7.8

Variazione di *gamma* e *bar_gamma* al variare di *num_rows_ratio* con *code_rate* = 0.9

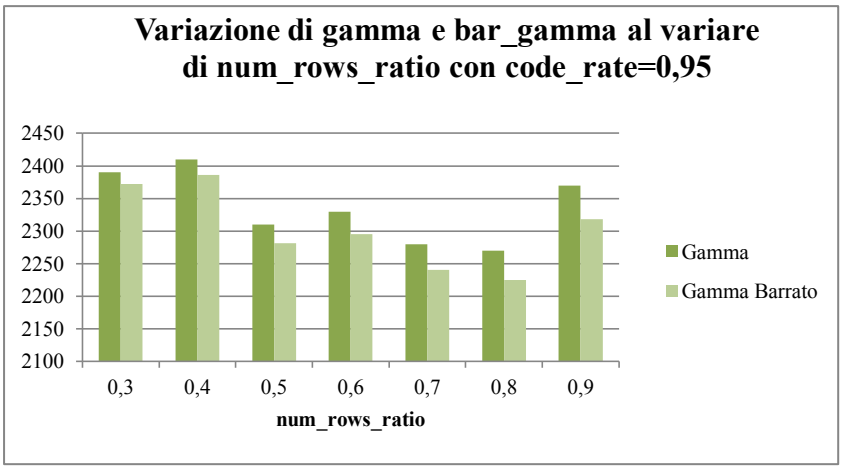

Figura 7.9

Variazione di *gamma* e *bar_gamma* al variare di *num_rows_ratio* con *code_rate* = 0.95

Dalle Figure 7.6, 7.7, 7.8, 7.9 e dai valori riportati nelle Tabelle 7.5, 7.6, 7.7 risulta evidente che, nonostante le dimensioni di chiavi pubbliche e firme e i valori di gamma e gamma barrato siano più elevati, a parità di parametri, il loro andamento è coerente con quello osservato a security level 128.

8. CONCLUSIONI

In questo lavoro è stato eseguito un dimensionamento dei parametri, cercando di ottimizzare figure di merito come dimensioni della chiave pubblica e delle firme. In particolare, questa ottimizzazione è stata eseguita cercando di guardare anche all'efficienza computazionale dello schema: per farlo, il tasso di rejection è stato tenuto limitato sia nella generazione della chiave che in quella della firma.

Per effettuare il dimensionamento dei parametri è stata utilizzata una procedura algoritmica, che punta a minimizzare l'occupazione in memoria (ovvero, dimensioni di chiavi e firme) cercando al tempo stesso di garantire il livello di sicurezza richiesto. Per determinare i valori ottimi per ogni parametro è stato esplorato un ampio range di possibilità

I risultati ottenuti mostrano che, rispetto ai set di parametri presentati in [1], vi è un ampio margine di miglioramento. Infatti, a parità di security level e con code rate pressoché identici (0.66 in [1], 0.7 in questo lavoro), è possibile passare dai 38.15 kB della chiave pubblica ai 17.057 kB ottenuti in questo lavoro, con una riduzione del 55%.

Inoltre, diversamente da [1], in questo lavoro sono stati considerati diversi valori di code rate. In seguito ad un'esplorazione in un ampio range (da 0.7 a 0.95), si è visto che aumentando il rate del codice si possono ottenere notevoli riduzioni nelle dimensioni delle chiavi pubbliche.

Ad esempio, sempre per un security level pari a 128, scegliendo *code_rate* pari a 0.95 è possibile portare la chiave pubblica a 2.693 kB, ottenendo così una riduzione quasi dell’93% rispetto alla minore dimensione della chiave pubblica tra quelle riportate in [1].

È importante osservare anche che queste riduzioni nella chiave pubblica corrispondono a firme altrettanto compatte. Ad esempio per code rate = 0.95 la dimensione della firma può scendere fino a 0.36 kB.

Inoltre, diversamente da [1], in questo lavoro sono stati calcolati parametri per security level più alti (192 bit) e dai risultati ottenuti si può osservare che lo schema continua ad avere chiavi e firme compatte, anche se, come ovvio, maggiori rispetto ad un security level più basso. Ad esempio, per *code_rate* = 0.95 si può arrivare ad avere una chiave pubblica di 6.174 kB.

Questi risultati mostrano che lo schema BCS è in grado di utilizzare chiavi relativamente compatte, anche per livelli di sicurezza elevati. Tuttavia si è osservato che, aumentando il security level e fissando il numero massimo di rejection, il range di valori ammissibili per alcuni parametri (come il *code_rate*) si riduce. Questo fenomeno può essere dovuto al fatto che per livelli di sicurezza maggiori diventa necessario accettare tassi di rejection molto più alti. Questo potrebbe rappresentare un problema, perché andrebbe ad ridurre notevolmente l'efficienza computazionale dello schema.

9. BIBLIOGRAFIA

1. Baldi, Marco, Franco Chiaraluce, and Paolo Santini. "Code-based signatures without trapdoors through restricted vectors." IACR Cryptol. ePrint Arch. 2021 (2021): 294.
2. Rajeshwaran, Kartik, and Kakelli Anil Kumar. "Cellular Automata Based Hashing Algorithm (CABHA) for Strong Cryptographic Hash Function." 2019 IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT). IEEE, 2019.
3. Zhu, Likun, and Lizhe Zhu. "Electronic signature based on digital signature and digital watermarking." 2012 5th International Congress on Image and Signal Processing. IEEE, 2012.
4. Van, Luu Xuan, and Luu Hong Dung. "Constructing a Digital Signature Algorithm Based on the Difficulty of Some Expanded Root Problems." 2019 6th NAFOSTED Conference on Information and Computer Science (NICS). IEEE, 2019.
5. https://www.sagemath.org/
6. Courtois, Nicolas T., Matthieu Finiasz, and Nicolas Sendrier. "How to achieve a McEliece-based digital signature scheme." International Conference on the Theory and Application of Cryptology and Information Security. Springer, Berlin, Heidelberg, 2001.
7. Song, Yongcheng, et al. "A code-based signature scheme from the Lyubashevsky framework." Theoretical Computer Science 835 (2020): 15-30.
8. Li, Zhe, Chaoping Xing, and Sze Ling Yeo. "A New Code Based Signature Scheme without Trapdoors." IACR Cryptol. ePrint Arch. 2020 (2020): 1250.
9. Aragon, Nicolas, et al. "Cryptanalysis of a code-based full-time signature." *Designs, Codes and Cryptography* 89.9 (2021): 2097-2112.
10. Baldi, Marco, et al. "Cryptanalysis of a Code-Based Signature Scheme Based on the Schnorr-Lyubashevsky Framework." IEEE Communications Letters 25.9 (2021): 2829-2833.
11. Persichetti, Edoardo. "Efficient one-time signatures from quasi-cyclic codes: A full treatment." Cryptography 2.4 (2018): 30.
12. Santini, Paolo, Marco Baldi, and Franco Chiaraluce. "Cryptanalysis of a one-time code-based digital signature scheme." 2019 IEEE International Symposium on Information Theory (ISIT). IEEE, 2019.
13. Debris-Alazard, Thomas, Nicolas Sendrier, and Jean-Pierre Tillich. "Wave: A new code-based signature scheme." (2018).
14. Coffey, John T., and Rodney M. Goodman. "The complexity of information set decoding." IEEE Transactions on Information Theory 36.5 (1990): 1031-1037.

10. APPENDICE

Appendice A – Script SageMath per l'Ottimizzazione dello Schema

Si riporta qui di seguito l'intero script SageMath implementato per l'ottimizzazione dello schema.

***

```
reset();

###################################
def log2(x):
    return log(x*1.)/log(2.);

###################################

##Funzione per verificare che il numero di hash sia sufficientemente
alto.
##Ritorna 1 se è sicuro, 0 in caso contrario
##In input prende:
# - ell: lunghezza hash
# - wc: numero di 1 e -1 nell'hash
# - SL: security level in bit
def hash_is_secure(ell,wc,SL):
    num_hash = log2(binomial(ell,wc)*2^wc);
    min_hash = 2*SL;
    if num_hash>min_hash:
        return 1
    else:
        return 0

#########################

##Funzione per verificare che il numero di uni e meno uni nella chiave
segreta siano sufficientemente alti
##Ritorna 1 se è sicuro, 0 in caso contrario
##In input prende:
# - te: numero di elementi non nulli (1 oppure -1) in ogni riga della
chiave segreta E
# - k: dimensione del codice
# - n: lunghezza del codice
# - SL: security level in bit
def key_is_secure(te,k,n,SL):
    esp_attack = -te*log2(1-k/n)
    if esp_attack>=SL:
        return 1
```

```
    else:
        return 0


##########################

##Funzione per verificare che la firma è sicura
##Ritorna 1 se è sicuro, 0 in caso contrario
##In input prende:
# - gamma_barra: parametro di restrizione del campo finito
# - n: lunghezza del codice
# - SL: security level in bit
def sign_is_secure(gamma_barra,n,SL):
    cost_attack = (n/2)*log2(2*gamma_barra)
    if cost_attack>= SL:
        return 1
    else:
        return 0


##########################################

##dimensione chiave pubblica (in kB)
def pub_key_size(b,n,k,q):
    r = n-k;
    return ((ceil(b*r*log2(q)))/8000)*1.;

##dimensione firma (in kB)
def signature_size(gamma_barra,n,wc):
    return ((ceil(n*log2(1+2*gamma_barra))+wc+wc*ceil(log2(n)))/8000)*1.;

##########################################################################
########################

##FUNZIONI PER REJECTION SAMPLING
##################################################

def ce_distribution(wE,wc,num_rows,q):

    th_pr = vector(RR,q);

    y = 0;
    for v in range(0,min(wE,wc)+1,2):
        y += 2^(-v)*binomial(v,v/2)*binomial(wE,v)*binomial(num_rows-
wE,wc-v)/binomial(num_rows,wc);
    th_pr[0] = y;

    for mbeta in range(1,min(wE,wc)):
```

```
        y = 0;
        for v in range(mbeta,min(wE,wc)+1,2):
            y += 2^(-
v)*binomial(v,(v+mbeta)/2)*binomial(wE,v)*binomial(num_rows-wE,wc-
v)/binomial(num_rows,wc)*1.;

        th_pr[mbeta] = y;
        th_pr[q-mbeta] = y;
    return th_pr;
##################################################

def ce_plus_y_distribution(wE,wc,num_rows,q,gamma_val, ce_pr):

    max_index = min(wE,wc);
    th_pr = vector(RR,q);
    for i in range(0,max_index+1):
        x = ce_pr[i];
        th_pr[i]+=x;
        if i>0:
            th_pr[q-i]+=x;
        for j in range(1,gamma_val+1):
            np = (i+j)%q;
            th_pr[np]+=x;
            np = (i-j)%q;
            th_pr[np]+=x;
            if i>0:
                np = (-i+j)%q;
                th_pr[np]+=x;
                np = (-i-j)%q;
                th_pr[np]+=x;


    return th_pr/(2*gamma_val+1);
##################################################################


def
faster_ce_plus_y_distribution(wE,wc,num_rows,q,new_gamma_val,old_gamma_va
l, old_th_pr, ce_pr):
    th_pr = old_th_pr*(2*old_gamma_val+1);
    for i in range(old_gamma_val+1,new_gamma_val+1):
        print(i);
        for j in range(q):
            np = (i+j)%q;
            th_pr[j]+= ce_pr[np];

    for i in range(-new_gamma_val, -old_gamma_val+1):
        for j in range(q):
```

```
            np = (i+j)%q;
            th_pr[j]+= ce_pr[np];

    return th_pr/(2*new_gamma_val+1);
########################################################################

def find_optimum(SL,max_sig_samples, max_key_samples,n,k,ell,q,gamma_min,
gamma_max, gamma_step, best_pk_size):

    wc = 0;
    while (hash_is_secure(ell,wc,SL)==0)&(wc<=ell):
        wc += 1;

    #Se wc è troppo alto, significa che il valore di ell è troppo basso
per garantire la sicurezza dello schema
    if wc>=ell:
        return 0,best_pk_size;

    #Calcolo il minimo valore di tE per avere sicurezza.
    tE = 0;
    while (key_is_secure(tE,k,n,SL)==0)&(tE<=n):
        tE += 1;

    #Se tE è troppo alto, significa che i valori di k ed n sono troppo
basso per garantire la sicurezza dello schema
    if tE>=n:
        return 0,best_pk_size;

    #Calcolo il minimo valore di wE per garantire che una chiave valida
viene generata dopo non più di num_key_samples in media
    wE = 1;
    num_key_samples = max_key_samples+1;
    while (num_key_samples>=max_key_samples)&(wE<=ell):
        wE += 1;
        pr = 0;
        for i in range(tE):
            pr+=(binomial(n,i)*((wE/ell)^i)*(1-wE/ell)^(n-i));
        pr = (1-pr)^ell;
        num_key_samples = (pr^-1);

    #Vado avanti solo se wE è sufficientemente basso:

    if wE>=ell:
        return 0,best_pk_size;

    #Distribuzione di c*E
    ce_x = ce_distribution(wE,wc,ell,q);
```

```
    print("wc = "+str(wc)+", we = "+str(wE)+", tE = "+str(tE));

 #Provo i valori di gamma. Per ogni gamma, calcolo il gamma barra più
grande possibile per avere sicurezza, e vedo se il rejection sampling è
abbastanza efficiente. Per farlo, si vuole che il numero di firme
scartate (in media) sia non maggiore di max_sig_samples
    for gamma_val in range(gamma_min, gamma_max, gamma_step):

        #Distribuzione di c*E
        g_x = ce_plus_y_distribution(wE,wc,ell,q,gamma_val,ce_x);

        #Calcolo gamma_barra imponendo che binomial(ell,gamma-
gamma_barra)*2^(gamma-gamma_barra) sia maggiore di 2^SL )
       # bar_gamma_val = gamma_val;
       # val = -SL+1;
       # while (val>(-SL))&((gamma_val-bar_gamma_val)>min(wc,wE)):
       #     bar_gamma_val = bar_gamma_val-1;
       #     val = ce_x[0];
       #     for i in range(1,gamma_val-bar_gamma_val+1):
       #         val+=2*ce_x[i];
#      #      print("bar_gamma = "+str(bar_gamma_val)+", val
="+str(val*1.));
       #     val = log2(1-val^n);
            #print("val = "+str(val));

       # print("gamma = "+str(gamma_val)+", bar_gamma =
"+str(bar_gamma_val));
        #vado avanti solo se bar_gamma_val non è negativo
        bar_gamma_val = gamma_val-min(wc,wE);
        #Verifico che lo schema sia sicuro nei confronti di attacchi per
falsificare le firme
        if sign_is_secure(bar_gamma_val,n,SL):

            #Imposto f_x come la distribuzione uniforme tra -bar_gamma e
bar_gamma
            f_x = vector(RR,q);
            for i in range(bar_gamma_val+1):
                f_x[i] = 1/(2*bar_gamma_val+1);
                if i>0:
                    f_x[q-i] = 1/(2*bar_gamma_val+1);

            #Calcolo di M, e del numero di firme generate in media come
M^n
            M = 0;
            for i in range(bar_gamma_val+1):
                this_M = f_x[i]/g_x[i];
                if this_M > M:
                    M = this_M;
```

```
            #Calcolo numero di firme scartate
            num_sig_samples = M^n;
          #  print("M = "+str(M));
          #  print("num_key_samples = "+str(num_key_samples));

            #Se il numero di firm scartate è ok, allora calcolo la chiave
pubblica
            if num_sig_samples < max_sig_samples:
                pk_size = pub_key_size(ell,n,k,q);
                if pk_size < best_pk_size:
                    best_pk_size = pk_size;
                    print("n = "+str(n));
                    print("k = "+str(k));
                    print("q = "+str(q));
                    print("wc = "+str(wc));
                    print("wE = "+str(wE));
                    print("ell = "+str(ell));
                    print("gamma_val = "+str(gamma_val));
                    print("bar_gamma_val = "+str(bar_gamma_val));
                    print("signature (kB) =
"+str(signature_size(bar_gamma_val,n,wc)));
                    print("pk size (kB) = "+str(pk_size));
                    print("key samples = "+str(num_key_samples*1.));
                    print("sig samples = "+str(num_sig_samples*1.));
                    print("---------------------------------");

    return 1, best_pk_size;

#########################################################################
#Inizio ricerca parametri per trovare l'ottimo

SL = 128; #livello di sicurezza
max_sig_samples = 500; #massimo numero di firme scartate (in media) per
generare una firma valida
max_key_samples = 1000; #massimo numero di chiavi scartate (in media) per
generare una chiave valida
code_rate = 0.9; #rate del codice (calcolato come k/n)
num_rows_ratio = 0.3; #rapporto tra ell ed n (ovvero, ell/n)

q = 16381;  #dimensione del campo finito

#Il codice esplora il range di valori di n da n_min ad n_max, a passi di
n_step
n_min = 40;
n_max = 600;
n_step = 10;
```

```
#Il codice esplora il range di valori di gamma da gamma_min a gamma_max,
a passi di gamma_step
gamma_min = 1000;
gamma_max = 2000;
gamma_step = 10;


best_pk_size = 1000000000000000; #minima dimensione per la chiave
pubblica


for n in range(n_min,n_max,n_step):  #ciclo for per esplorare i diversi
valori di n

    k = round(n*code_rate); #calcolo dimensione del codice (valore di k)
    ell = round(n*num_rows_ratio); #calcolo nuumero di righe di E (valore
di ell)

    ok, best_pk_size = find_optimum(SL,max_sig_samples,
max_key_samples,n,k,ell,q,gamma_min, gamma_max, gamma_step,
best_pk_size);
```

***