UNIVERSITÀ POLITECNICA DELLE MARCHE

FACOLTÀ DI INGEGNERIA

Corso di Laurea triennale in Ingegneria Elettronica

Implementazione di uno schema post-quantum di firma digitale basato su codici

Implementation of a code-based post-quantum digital signature scheme

Tesi di Laurea di:
Lei Chen

Relatore:
Prof. Franco Chiaraluce

Correlatore:
Dr. Paolo Santini

Anno Accademico 2020/2021

Indice

1 Introduzione

La sicurezza è un elemento fondamentale per qualunque sistema di comunicazione. Ad esempio, durante uno scambio di informazioni tra due utenti, è quasi sempre necessario avere certezza sull'identità del mittente (ovvero, colui che ha prodotto ed inviato uno o più messaggi). Allo stesso modo, in alcuni contesti può essere necessario avere certezza riguardo all'integrità del messaggio, ovvero, riguardo al fatto che il suo contenuto non sia stato modificato dopo l'invio. Il metodo utilizzato per garantire queste (ed altre) proprietà è la firma digitale.
Le firme digitali sfruttano la crittografia a chiave pubblica, anche detta asimmetrica, ovvero algoritmi crittografici basati su una coppia di chiavi denominate, rispettivamente, chiave segreta e chiave pubblica. La chiave segreta, come dice il nome stesso, non viene resa pubblica, e consente di generare la firma del messaggio che si vuole firmare. Chi riceve, a sua volta utilizza la chiave pubblica per decifrare e verificare la validità della firma. La coppia di chiavi viene generata dall'utente che pone le firme, e deve essere tale per cui recuperare la chiave segreta dalla chiave pubblica è computazionalmente molto difficile (ovvero, non possibile in modo efficiente).
Infatti la fiducia in questi schemi è basato proprio sul fatto che le chiavi segrete non possono essere recuperate da un attaccante, in modo che solamente il possessore della chiave segreta può generare velocemente e correttamente firme valide.

Algoritmi di firma digitale vengono oggi utilizzati in molteplici ed estremamente diffuse applicazioni. Uno degli esempi più significativi è quello dell'accesso ad un sito web tramite il browser; un sito "sicuro" dev'essere dotato di un certificato digitale (secondo lo standard X.509) che, a sua volta, contiene la firma digitale. In questo modo l'utente, verificando la firma, può avere garanzie sull'identità del sito su cui sta navigando.
In Figura 1.1 è mostrato un esempio di trasmissione e verifica di un messaggio con firma. Come si vede dalla figura, Bob tramite la verifica della firma posta da Alice può essere sicuro dell'identità di quest'ultima. Nel caso in cui il messaggio (originalmente inviato e firmato da Alice) viene modificato da un attaccante, Bob se ne può accorgere perché la firma allegata al messaggio non è più valida. In questo modo, il messaggio non verrà riconosciuto come autentico e pertanto non verrà accettato.

I primi schemi di firma digitale furono inventati negli anni 70. Ad oggi, vi è una molteplicità di schemi di firma utilizzabili, ognuno con le proprie peculiarità (ad esempio, compattezza delle chiavi o alta velocità di calcolo); tra questi, si possono citare RSA [1], Diffie-Hellman [2], El Gamal [3], ECC [4, 5], etc. Da

un punto di vista formale, la sicurezza di tali algoritmi è legata alla risoluzione di problemi matematici complessi. Ad esempio, RSA sfrutta il problema della fattorizzazione dei numeri interi: dati due numeri primi è facile calcolarne il loro prodotto mentre l'operazione inversa (ossia, trovare i fattori primi partendo da un numero intero grande) è estremamente complicata. Più precisamente, la complessità computazionale di ogni algoritmo che risolve questo compito (ovvero, di ogni attacco ad RSA) cresce come $2^{k\cdot\log_2(n)}$ dove n è il numero che si vuole fattorizzare.

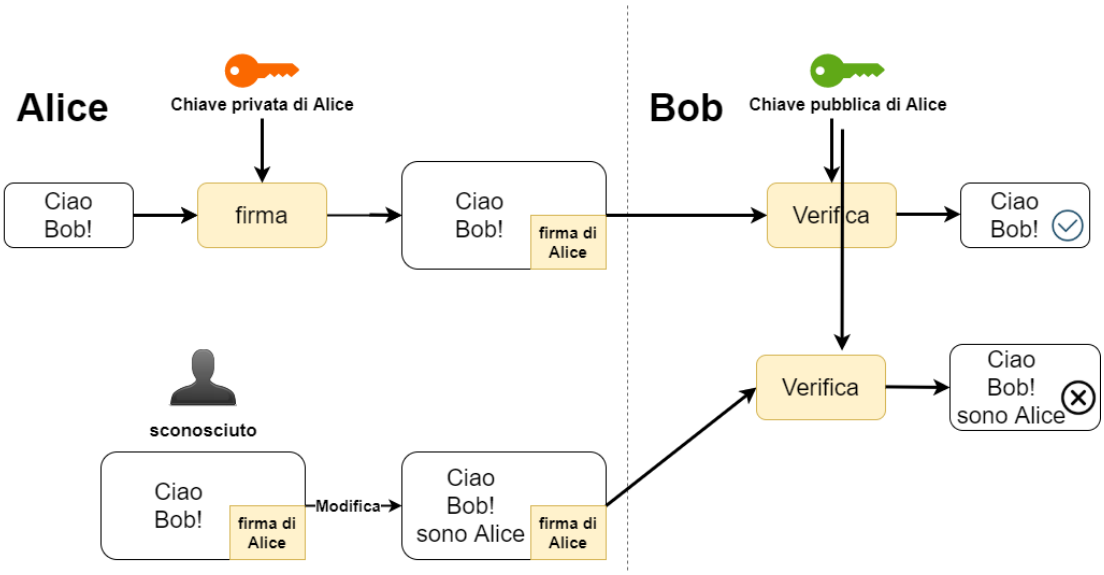

Figura 1.1 Esempio di trasmissione di un messaggio firmato

Però, con l'avvento dei computer quantistici, gli schemi attualmente più diffusi corrono seri rischi di sicurezza [6]. Ad esempio, l'algoritmo quantistico di Shor permette di risolvere il problema della fattorizzazione in un tempo polinomiale; pertanto, questo algoritmo consente di recuperare in modo efficiente la chiave segreta nello schema RSA.
Lo stesso algoritmo, con opportune modifiche, può essere utilizzato per attaccare sia la procedura di Diffie-Hellman, che è basata sul calcolo del logaritmo discreto, sia la procedura ECC che è invece basata sull'utilizzo delle curve ellittiche. Questo prova che schemi come RSA, Diffie-Hellman ed ECC potranno, in futuro, essere attaccati molto efficientemente.

Per questi motivi è stata introdotta la crittografia post-quantum, ovvero, l'area scientifica mirata alla definizione ed allo studio di schemi ed algoritmi che possano resistere ad attacchi implementati tramite computer quantistici, con particolare riferimento alla crittografia a chiave pubblica.
Infatti, mentre per schemi a chiave simmetrica proteggersi dagli attacchi è relativamente semplice (come regola generale, è normalmente sufficiente raddoppiare le dimensioni delle chiavi [7]), per gli schemi a chiave pubblica la

situazione è differente, in quanto aumentare i valori dei parametri non è sufficiente e, pertanto, diventa necessario sostituire i problemi matematici che stanno alla base dello schema. Per questo motivo, nel 2017 il NIST (National Institute of Standards and Technology) ha lanciato la competizione "Post-Quantum Cryptography Standardization" per selezionare schemi di crittografia a chiave pubblica con sicurezza post quantum [8]. Tra gli algoritmi proposti, tutti gli schemi di firma basati su codici per la correzione d'errore sono stati eliminati dalla competizione, perché sono stati tutti attaccati con successo. Questo rappresenta un'evidenza del fatto che trovare schemi di firma basati su codici è un problema aperto e, ad oggi, non esiste una soluzione efficiente.
A questo proposito, il gruppo di ricerca in crittografia dell'Università Politecnica delle Marche ha recentemente proposto uno schema di firma digitale basato su codici per la correzione d'errore di nuova concezione, avendo come obiettivo il superamento dei limiti posti dagli schemi precedenti.

Il presente lavoro di tesi consiste nell'implementazione dello schema proposto dai ricercatori UNIVPM [9], basato su codici random definiti su un campo finito non binario.
In un primo momento, lo schema è stato implementato tramite linguaggio Matlab, in modo da verificarne la correttezza procedurale. In un secondo momento, l'implementazione è stata sottoposta ad una fase di benchmark, con l'obiettivo di misurare i tempi di calcolo richiesti da ognuna delle fasi svolte dall'algoritmo.

2 Notazione e background

In questa sezione vengono introdotti alcuni concetti di base per la comprensione di quanto seguirà.

2.1 Notazione

Verranno considerati numeri definiti su un campo finito $\mathbb{F}_q$ con q elementi, che vengono rappresentati come $\{0, 1, ..., q-1\}$, dove q è un numero primo. Quando si effettua un'operazione in un campo finito, i calcoli vengono fatti riducendo modulo q. Ad esempio, se q = 5 allora $2 + 3 = 0$. Infatti $2 + 3 \bmod 5 = 5 \bmod 5 = 0$. Lo stesso sistema viene applicato ai numeri negativi, ad esempio: $-1 \bmod 5 = 4$.

Lo schema di firma considerato in questo lavoro esegue operazioni e calcoli su matrici e vettori. Matrici e vettori verranno, rispettivamente, indicati con lettere maiuscole in grassetto e lettere minuscole in grassetto, ad esempio $\mathbf{A}$ rappresenta una matrice e $\mathbf{b}$ un vettore.

2.2 Background

Come si è detto in precedenza, lo schema di firma digitale utilizza una chiave segrete e una chiave pubblica. La chiave segreta è nota solo all'autore della firma, mentre la chiave pubblica è nota a tutti gli utenti con cui esso comunica. Le chiavi svolgono compiti diversi: la chiave segreta serve a produrre la firma sull'hash del messaggio mentre, dualmente, la chiave pubblica consente di verificare la validità della firma. In particolare, se l'hash del messaggio coincide con il risultato della decifrazione attraverso la chiave pubblica, allora questo significa che la firma è valida e, quindi, che il messaggio non è stato alterato dopo il suo invio.
Una proprietà fondamentale per garantire che tutto funzioni correttamente è che la stessa chiave non può contemporaneamente cifrare e decifrare; è quindi indispensabile che l'altra chiave svolga l'operazione inversa. Non a caso si parla di crittografia asimmetrica, in contrapposizione a quella simmetrica in cui la stessa chiave può essere utilizzata sia per cifrare che per decifrare.
Nello schema che in seguito verrà trattato, le chiavi saranno rappresentate sotto forma di matrici.

Un tipico schema di firma digitale è composto da tre algoritmi:

- Un algoritmo per la generazione della chiave segreta e della chiave pubblica.
- Un algoritmo per la generazione della firma.
- Un algoritmo per la verifica della firma.

3 Schema di firma digitale

In questa sezione vengono descritti gli algoritmi alla base dello schema di firma digitale oggetto del lavoro di tesi.

3.1 Generazione della chiave

I parametri considerati di seguito per la generazione della chiave sono:

- n: lunghezza del codice;
- k: dimensione del codice;
- r: lunghezza della ridondanza;
- b: righe della matrice chiave segreta;
- w_E: peso di Hamming, il cui significato verrà spiegato in seguito;
- t_E: numero di elementi non nulli in una riga di una matrice, il cui significato verrà pure specificato in seguito.

Le operazioni da svolgere per generare chiave pubblica e chiave privata sono le seguenti.

1. Si sceglie una matrice $\mathbf{H}$ generata concatenando una matrice identità $\mathbf{I}$ di dimensione $r \times r$ con una matrice $\mathbf{P}$ generata casualmente, di dimensioni $r \times k$ (r righe e k colonne).
2. Si sceglie a caso una matrice $\mathbf{E}$ di dimensione $b \times n$, in cui ogni colonna ha w_E elementi non nulli, con valori -1 o 1, mentre gli altri elementi sono nulli. Inoltre $\mathbf{E}$ deve avere in ogni riga almeno t_E elementi non nulli. Se questa proprietà non è verificata, allora la matrice, generata casualmente, viene scartata e se ne genera un'altra, finché non si ottiene una matrice valida.
3. Si calcola $\mathbf{S} = \mathbf{H}\mathbf{E}^{\top}$, dove $\top$ indica la trasposizione.
4. La chiave segreta è $\mathbf{E}$, mentre la chiave pubblica è costituita dalle matrici $\mathbf{H}$ ed $\mathbf{S}$. In particolare, la chiave segreta $\mathbf{E}$ non può essere ricavata invertendo la precedente relazione perché la matrice $\mathbf{H}$, in quanto non quadrata, non è invertibile. e quindi non è possibile calcolare $\mathbf{E} = (\mathbf{H}^{-1}\mathbf{S})^{\top}$.

3.2 Generazione della firma

Oltre ai parametri definiti nella sezione precedente, per la generazione della firma occorre considerare:

- $\boldsymbol{m}$: il messaggio;
- γ, $\bar{\gamma}$, e w_c, il cui significato verrà spiegato successivamente.

Le operazioni da svolgere per generare la firma sono le seguenti:

1. Si sceglie a caso un vettore $\mathbf{y}$ di lunghezza n, con valori $\in\{0, \pm1, \pm2, \dots, \pm\gamma\}$.
2. Si calcola $\mathbf{s_y} = \mathbf{yH}^\top$.
3. Si calcola l'hash di m concatenato con $\mathbf{s_y}$; indichiamo questo hash con $\mathbf{c}$; $\mathbf{c}$ sarà un vettore di lunghezza b, con w_c elementi non nulli, che assumono valori 1 oppure -1, mentre gli altri elementi sono nulli.
4. Si calcola la firma come $\mathbf{z} = \mathbf{cE} + \mathbf{y}$.
5. La firma è valida solo se $\mathbf{z}$ ha solo valori $\in\{0, \pm1, \pm2, \dots, \pm\bar{\gamma}\}$; se questa proprietà non è verificata, allora si riparte dal punto 1, finché non si ottiene una $\mathbf{z}$ valida.
6. L'output è costituito dalla firma $\mathbf{z}$ e dall'hash $\mathbf{c}$; $\boldsymbol{\sigma} = \{\mathbf{z}, \mathbf{c}\}$.

3.3 Verifica della firma

La verifica della firma consente a chi riceve il messaggio di riconoscere la validità, o meno, della firma relativa al messaggio stesso.
Le operazioni per la verifica sono più semplici rispetto agli altri due algoritmi. Infatti per verificare la firma, si eseguono i seguenti passaggi:

1. Si verifica che $\mathbf{z}$ assume valori $\in\{0, \pm1, \pm2, \dots, \pm\bar{\gamma}\}$;
2. Si calcola $\mathbf{s_y} = \mathbf{zH}^\top - \mathbf{cS}$;
3. Si verifica la validità dell'hash, ovvero se $\mathbf{c}$ calcolato come $\mathbf{c} = \mathrm{Hash}(\boldsymbol{m}, \mathbf{s_y})$ coincide con $\mathbf{c}$ ricavato dal valore di $\boldsymbol{\sigma}$.
4. Se i passi precedenti hanno un riscontro positivo allora la firma è valida, altrimenti no.

Una firma “onestamente generata” è sicuramente valida; infatti le operazioni derivano dal seguente equazione:

$$\mathbf{zH}^{\top} - \mathbf{cS} = (\mathbf{cE} + \mathbf{y})\mathbf{H}^{\top} - \mathbf{cEH}^{\top} = \mathbf{yH}^{\top} = \mathbf{s}_{\mathbf{y}}$$

4 Implementazione Matlab

In questa sezione viene descritta l'implementazione Matlab dei tre algoritmi che realizzano la firma digitale, descritti nella sezione precedente.

4.1 Implementazione della generazione della chiave

Per implementare l'algoritmo di generazione delle chiavi, per prima cosa è stata scritta una funzione Matlab, *key_generation.m*, che svolge parzialmente il lavoro di generazione della chiave. Più precisamente, nella funzione viene trascurato il vincolo sulle righe della matrice **E**. *key_generation* prende in input i parametri q, n, k, b, w_E, e t_E; e restituisce le matrici **S**, **H**, ed **E**; *r* definito come variabile è uguale a $n - k$.
Per generare **H**, servono le matrici **I** e **P**; la funzione *randi* è capace di generare matrici di dimensione arbitraria, i cui elementi sono numeri interi, uniformemente distribuiti nel range predefinito. Più in particolare, con tale comando si genera **P** avente come elementi valori numerici compresi fra 0 e $q - 1$; dopodiché è stata usata la funzione *eye* per produrre la matrice identità **I**; infine **H** si ottiene concatenando **P** ed **I**.
La generazione di **E** è stata un po' più complessa; per prima cosa è stata generata una matrice $b \times n$ di zeri con la funzione *zeros*. Per soddisfare il primo vincolo posto su **E**, su ciascuna colonna devono essere presenti w_E elementi con valori -1 o 1. È stato creato un vettore colonna ausiliario *a* di lunghezza w_E e valori casuali pari a 0 od 1, usando sempre il comando *randi*; successivamente gli zeri di *a* sono stati sostituiti con -1. Poi è stato generato un altro vettore, *pos*, che è un vettore di indici. Attraverso il comando *randperm*, che ritorna un vettore contenente w_E numeri interi casuali univoci compresi tra 1 ed *b*; alle posizioni identificate da *pos* vengono quindi assegnati i valori di *a*. Ripetendo la procedura per tutte le colonne si ottiene la matrice **E** desiderata.
Come ultima fase svolta dalla funzione, è stato calcolato **S** come prodotto tra **H** ed **E** trasposto, eseguendo il calcolo tramite le operazioni *mod*: questo assicura che il risultato rimanga nel campo finito $\mathbb{F}_q$ considerato.

Per verificare il secondo vincolo su **E**, cioè che in ogni riga vi siano almeno t_E elementi non nulli, è stata definita un'altra funzione, di controllo, denominata *key_check.m*. Tale funzione prende in input la matrice **E** ed i parametri *b*, *n* e t_E, e restituisce "true" se la matrice è valida (ovvero, se soddisfa il vincolo), mentre restituisce "false" nel caso in cui la matrice non è valida.

Nella funzione sono state definite alcuni variabili locali:

- *rlttE* (rows less than t_E) contatore che salva il numero delle righe che hanno un numero di elementi non nulli minore di t_E.
- *N* per salvare il numero di elementi non nulli presenti nella riga.
- *minv* per salvare il valore minimo di *N*.

L'esecuzione del controllo è piuttosto meccanica, in quanto si effettua il controllo riga per riga. Su ogni riga il numero di elementi non nulli viene calcolato tramite il comando *nnz*, che restituisce il numero di elementi non nulli. Questo valore viene salvato nella variabile *N*, che viene poi sottoposta a due verifiche: se *N* è minore di t_E si incrementa la variabile *rlttE*, inoltre se *N* è minore di *minv* il valore di *N* viene assegnato *minv*. La matrice è valida se e solo se, alla fine del ciclo di controllo, il valore di *rlttE* risulta ancora nullo.
Le variabili *rlttE* e *minv*, inoltre, servono anche per avere informazioni dettagliate sul controllo fatto, come numero di matrici generate (in media) prima di ottenerne una valida.

Dopo l'implementazione delle due funzioni descritte in precedenza, si è proceduto a costruire la versione finale dell'algoritmo di generazione della chiave. Per questo scopo, la procedura della verifica su t_E è stata integrata nell'algoritmo di generazione della chiave (*key_generation*), in modo tale da garantire che l'output della funzione di generazione sia una matrice $\mathbf{E}$ sicuramente valida (ovvero, una chiave segreta valida). In pratica, nella funzione si ripete la generazione di $\mathbf{E}$, estraendo una nuova matrice casuale ad ogni tentativo, fino a quando il vincolo sul peso delle righe non è soddisfatto. La versione finale della funzione è stata chiamata *valid_key_generation.m*.

4.2 Implementazione della generazione della firma

Sono state implementate due funzioni per la generazione della firma, una per generare l'hash $\mathbf{c}$ partendo dal messaggio m e dal vettore $\mathbf{s_y}$, l'altra per generare la firma $\mathbf{z}$.

La prima funzione, denominata *generate_c.m*, prende in input q, w_c, b, il messaggio $\boldsymbol{m}$ e il vettore $\mathbf{s_y}$, e restituisce il vettore $\mathbf{c}$.

Il generatore di $\mathbf{c}$ prima di tutto concatena $\boldsymbol{m}$ ed $\mathbf{s_y}$ in una variabile chiamata *m_conc_sy* sotto forma di stringa; il comando *num2str* consente di convertire numeri o vettori in stringa, in questo caso viene applicato a $\boldsymbol{m}$ ed $\mathbf{s_y}$.

Successivamente calcola l'hash di *m_conc_sy* con l'ausilio di una funzione hash; per semplicità è stata assunta la funzione *DataHash* scaricabile dal sito ufficiale Matlab[1]. Il risultato viene usato come seme per un generatore pseudorandom, necessario per generare numeri casuali; in particolare, in questo lavoro è stata usata la funzione *rng*. Il generatore produce $\mathbf{c}$, utilizzando lo stesso metodo che è stato scelto per generare le colonne della matrice $\mathbf{E}$.
Infatti, $\mathbf{c}$ deve essere un vettore di lunghezza *b,* con solamente w_c elementi non nulli. Per far ciò, è stata definita una variabile ausiliaria *pos_ones*, contenente le posizioni degli elementi non nulli, generata attraverso il comando *randperm*. Poi, è stato definito un vettore **a** di lunghezza w_c, che viene riempito con valori casuali estratti dal set $\{1, -1\}$.
In questo modo, ad ognuna delle posizioni contenute in *pos_ones* viene assegnato il rispettivo valore preso da **a**; per tutte le altre posizioni (ovvero, quelle non specificate da *pos_ones*), il vettore $\boldsymbol{c}$ ha valore nullo.

La seconda funzione implementata è stata chiamata *signature_generation.m*; essa prende in input la chiave pubblica $\mathbf{H}$, la relativa chiave segreta $\mathbf{E}$, i parametri *n*, γ e $\bar{\gamma}$, ed i parametri che servono alla generazione di $\mathbf{c}$, ovvero, q, w_c, b, il messaggio $\boldsymbol{m}$ e il vettore $\mathbf{s_y}$.

L'output della funzione è dato dalla coppia di vettori $\{\mathbf{z}, \mathbf{c}\}$, che rappresenta la firma nello schema considerato.

Questa volta i calcoli da eseguire sono più semplici, ma hanno bisogno di essere ripetuti più volte. Infatti, l'algoritmo di generazione della firma necessita di una fase di rejection sampling, necessaria per garantire che la firma prodotta non sia debole (ovvero, che non riveli informazioni relative alla chiave segreta). In breve, l'effetto del rejection sampling è quello di scartare le firme che non valide.
La generazione viene ripetuta fino a che una firma valida non viene generata; ogni nuovo tentativo è completamente random ed incorrelato dagli altri.
Per avere informazioni sul numero di tentativi è stata aggiunta una variabile denominata *not_valid_counter*.

L'algoritmo di generazione della firma inizia con la generazione casuale del vettore $\mathbf{y}$, utilizzando il comando *randi*. In particolare, per garantire che i numeri ottenuti siano elementi del campo finito $\mathbb{F}_q$, è necessario specificare sia valore massimo e minimo che, in questo caso, corrispondono rispettivamente a $q - 1$ e 0. Una volta che $\boldsymbol{y}$ è stato generato, viene calcolato il vettore $\mathbf{s_y} = \mathbf{y}\mathbf{H}^{\top}$.

[1] https://it.mathworks.com/matlabcentral/answers/3314-hash-function-for-matlab-struct

Successivamente si utilizza la funzione *generate_c* per produrre l'hash $\mathbf{c}$, utilizzando come input sia il messaggio che deve essere firmato, sia il vettore $\boldsymbol{s_y}$.

A questo punto, si hanno tutti gli elementi per calcolare la firma $\mathbf{z} = \mathbf{cE} + \mathbf{y}$.
Una volta che $\boldsymbol{z}$ è stato ottenuto, si passa allo step di rejection sampling, in cui si valuta se il vettore rispetta i vincoli richiesti. Come detto più sopra, affinché la firma sia valida occorre che $\mathbf{z}$ abbia tutti i valori appartenenti al set $\{0, \pm 1, \pm 2, \dots, \pm\bar{\gamma}\}$. Per effettuare il controllo viene usata la funzione *find,* la quale riesce a trovare gli elementi di un vettore che soddisfano un determinato vincolo. In questo caso è necessario effettuare la verifica duale, ovvero se ci sono elementi di $\mathbf{z}$ che sono maggiori di $\bar{\gamma}$ e, allo stesso tempo, minori di $-\bar{\gamma} = q - \bar{\gamma}$. Se questi elementi non vengono trovati, allora la firma è valida; in caso contrario, la firma viene scartata e si riparte dall'inizio (ovvero, dalla generazione di $\mathbf{y}$). Per verificare la presenza degli elementi non desiderati, è stato utilizzato il comando *isempty*: se l'output del comando *find* è un vettore vuoto, allora questo significa che questi elementi non sono stati trovati, pertanto la firma è valida.

4.3 Implementazione della verifica della firma

Per implementare l'algoritmo di verifica è stata scritta una funzione denominata *signature_verification.m*. La funzione prende in input, oltre alla chiave pubblica $\{\mathbf{H}, \mathbf{S}\}$ ed alla firma $\boldsymbol{\sigma} = \{\mathbf{z}, \mathbf{c}\}$, anche il messaggio $\boldsymbol{m}$ ed i parametri $\bar{\gamma}$ e q. La funzione restituisce **true** o **false**, a seconda che la firma sia valida oppure no.
La funzione per prima cosa verifica la validità di $\mathbf{z}$, ovvero, se nel vettore compaiano solamente valori presi dal set $\{0, \pm 1, \pm 2, \dots, \pm\bar{\gamma}\}$. Per fare ciò è stato utilizzato lo stesso metodo di verifica introdotto per la generazione della firma (basato sulla combinazione di istruzioni *find* e isempty). Se $\mathbf{z}$ è valido, allora si procede alla verifica di $\mathbf{s_y}$, altrimenti la funzione interrompe la procedura di verifica e restituisce **false**.

Il secondo passaggio consiste nel calcolo di $\mathbf{s_y} = \mathbf{zH}^\top - \mathbf{cS}$, passando poi il risultato alla funzione *generate_c* insieme al messaggio $\boldsymbol{m}$. La funzione *generate_c* è la stessa che viene utilizzata nell'algoritmo di generazione della firma, definito in precedenza. Se quest'ultima funzione restituisce un vettore $\mathbf{c}_1$ identico a $\mathbf{c}$, allora si conclude che la firma è valida. Per comparare $\mathbf{c}$ e $\mathbf{c}_1$ è stato adoperato il comando *isequal* che restituisce **true** se tutti gli elementi sono uguali, e **false** in caso contrario. Di conseguenza, l'algoritmo di verifica restituisce direttamente il risultato ottenuto da *isequal* sui due vettori $\mathbf{c}$ e $\mathbf{c}_1$.

5 Analisi sui tempi di calcolo

Ai fini della valutazione dell'efficienza dell'algoritmo di firma digitale proposto, un parametro molto importante è ovviamente il tempo necessario per eseguire i singoli algoritmi. Come è facile intuire, questo parametro dipende dai valori numerici delle variabili in gioco. Questi ultimi sono scelti per garantire un prefissato livello di sicurezza. Nondimeno, lo stesso livello di sicurezza può essere raggiunto utilizzando valori diversi. Nel seguito, verranno prima presi in esame i valori delle variabili proposti in [9]. Successivamente l'analisi sarà estesa a scelte alternative, risultanti da un lavoro di ottimizzazione svolto, in parallelo, in un'altra tesi di laurea [10].

5.1 Analisi con parametri non ottimizzati

In questa sezione si andranno a mostrare i risultati numerici per lo schema con parametri non ottimizzati. In particolare, per questa versione dell'algoritmo, si sono considerate le seguenti scelte:

- $q = 16381$,
- $n = 400$,
- $k = 300$,
- $r = n - k = 100$,
- $b = 218$,
- $w_E = 46$,
- $t_E = 64$;
- $w_C = 67$;
- $\gamma = 3420$,
- $\bar{\gamma} = 3375$.

5.1.1 Tempo di calcolo per la generazione della chiave

Per ottenere una stima significativa, le misurazioni dei tempi sono state effettuate mediando su un totale di 1000 esecuzioni dell'algoritmo di generazione delle chiavi. Chiaramente, maggiore è il numero di prove che vengono effettuate, maggiore è l'affidabilità statistica del risultato che si ottiene.
Per la misurazione dei tempi, si è utilizzato un PC con CPU Intel quad-core 3.2 GHz.

Per misurare lo scorrere del tempo, è stata utilizzata la funzione Matlab *tic-toc* consente di misurare il tempo trascorso tra i comandi *tic* e *toc*, come un cronometro.

I risultati ottenuti per la generazione della chiave sono i seguenti:

- 3.6 secondi per 1000 esecuzioni senza la verifica su t_E; in media, quindi, 3.6 ms.
- Aggiungendo il controllo sul valore di t_E, il tempo aumenta fino ad 8 secondi per 1000 esecuzione; in media, pertanto, 8 ms.

Nella Figura 5.1 è mostrato un istogramma del tempo generazione chiave tenendo conto della verifica su t_E. La Figura 5.2 riporta la stima della probabilità di occorrenza del tempo di generazione della chiave (sui singoli valori temporali discretizzati), sulla base delle 1000 prove fatte. Come si vede dalla figura, in circa il 30% dei casi il tempo di generazione è prossimo a 4 ms, mentre in circa il 10% dei casi il tempo necessario è di circa 6,5 ms. Per la stima della probabilità di occorrenza sul tempo di generazione della chiave sono stati ricavati anche la media e la varianza, che sono rispettivamente 8 ms e 26 ms^2.
Per ottenere tali risultati e di conseguenza i diagrammi sulla statistica del tempo di generazione della chiave, ovvero le Figure 5.1 e 5.2, è stato scritto un apposito script, denominato *test_firma_istogramma.m*, il quale fornisce in uscita un vettore *time_gen* che contiene il tempo che ha richiesto ciascuna delle 1000 generazioni della chiave. Successivamente, la funzione *histogram* genera un istogramma partendo dagli elementi di *time_gen*, specificando un numero di step pari a 100 (lo stesso numero di step verrà usato per tutti gli istogrammi). La probabilità di occorrenza è stimata sullo stesso vettore: si definisce l'asse temporale in un intervallo tra il valore minimo e il valore massimo contenuto in *time_gen* suddiviso in un numero prefissato di intervalli, *num_step*, assunto pari a 100. La probabilità in ogni sottointervallo è calcolato dividendo elementi appartenenti al sottointervallo sul numero totale di test (1000). Infine attraverso la funzione *plot* viene generato il grafico relativa alla stima della probabilità di occorrenza. Ovviamente lo step size (larghezza del sottointervallo delle ascisse) usati dall'istogramma e dalla stima della probabilità di occorrenza coincidono; nei grafici valgono entrambi 0.4 ms.

5.1.2 Numero di tentativi per avere matrice E con almeno t_E elementi non nulli per riga

Un'altra informazione utile riguarda la percentuale di matrici valide. Nelle 1000 simulazioni effettuate sulla generazione della chiave senza il vincolo su t_E, 620 matrici **E** sono risultate inadeguate mentre le restanti 380 sono state accettate in quanto soddisfano i vincoli. La probabilità stimata di generare una matrice valida, sulla base di questa prova, è quindi pari al 38%.
Solo per questo test, per rendere l'analisi statistica più affidabile, il numero delle simulazioni viene portato a 100000. Le matrici **E** inadeguate sono 60620 invece le restanti 39380 sono valide. Ora la probabilità stimata di generare una matrice valida diventa pari al 39.3%.
Con 1000 simulazioni il risultato è già significativo, ma con due ordini di grandezza superiori diventa ancora più affidabile. Infatti tra la versione meno affidabile e quella più affidabile lo scostamento della percentuale è del 1.3%, ovvero un errore relativo del 3.3%. Questo dimostra anche, che sarebbe necessario salire ancora con il numero delle prove. Ma evidentemente non è stato fatto perché l'onere diventa ingestibile.

Se viene fatta ripetere 1000 volte la generazione della chiave con matrice **E** valida e, per l'analisi, avendo aggiunto nella funzione anche il ritorno del numero di tentativi effettuati per generare matrice **E** valida (è stato aggiunto un contatore che registra le ripetizioni fatte dal ciclo while), il risultato è in media di circa 2.58 tentativi per generare una matrice **E** valida durante una generazione di chiave. Come si vede nella Figura 5.3, nel 38% dei casi la prima matrice **E** generata è già valida.
In [9] è stata fornita (e giustificata) la seguente formula per il calcolo del numero medio di tentativi da effettuarsi per determinare una matrice valida:

$$\text{Media tentativi} = \left(1 - \sum_{i=0}^{t_E-1} \binom{n}{i} \left(\frac{W_E}{b}\right)^i \left(1 - \frac{W_E}{b}\right)^{n-i}\right)^{-b}$$

Questa formula, con il valore assunto per le variabili, fornisce: 2.493, valore questo molto prossimo a quello determinato attraverso le simulazioni.

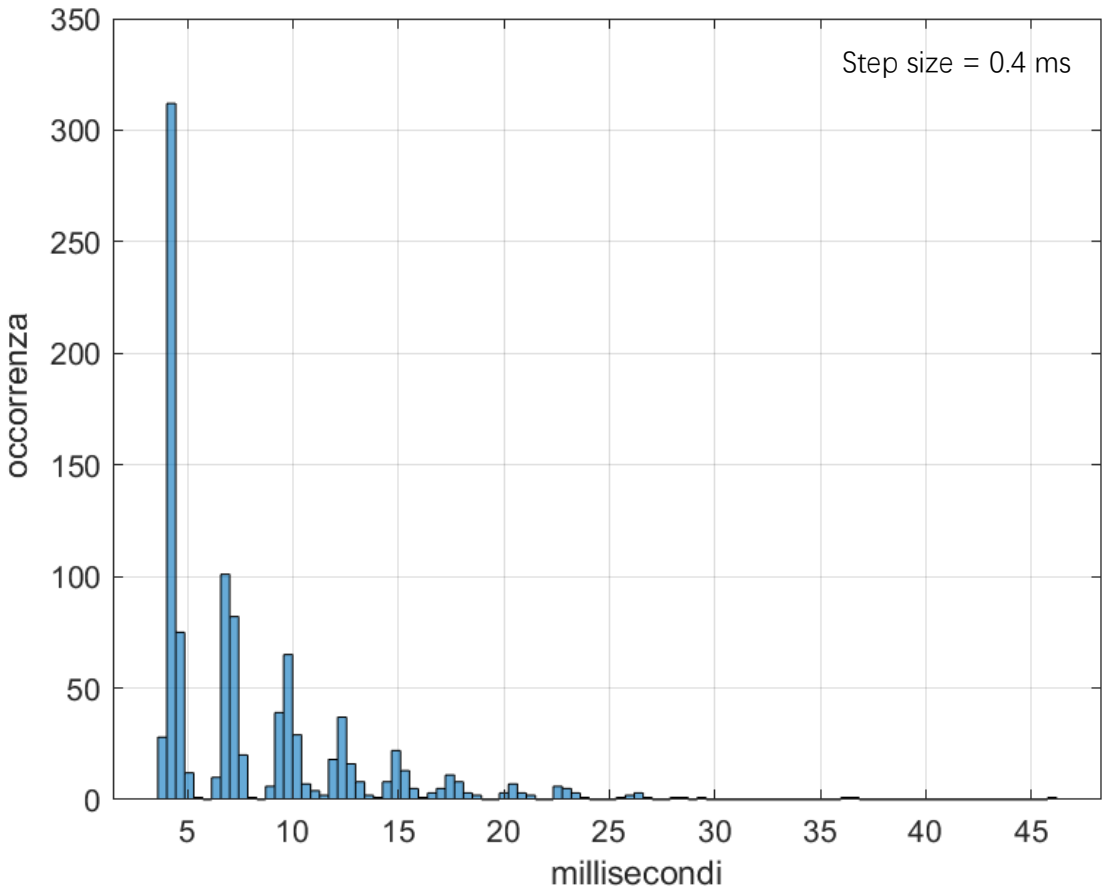

Figura 5.1: Istogramma del tempo di generazione della chiave in millisecondi

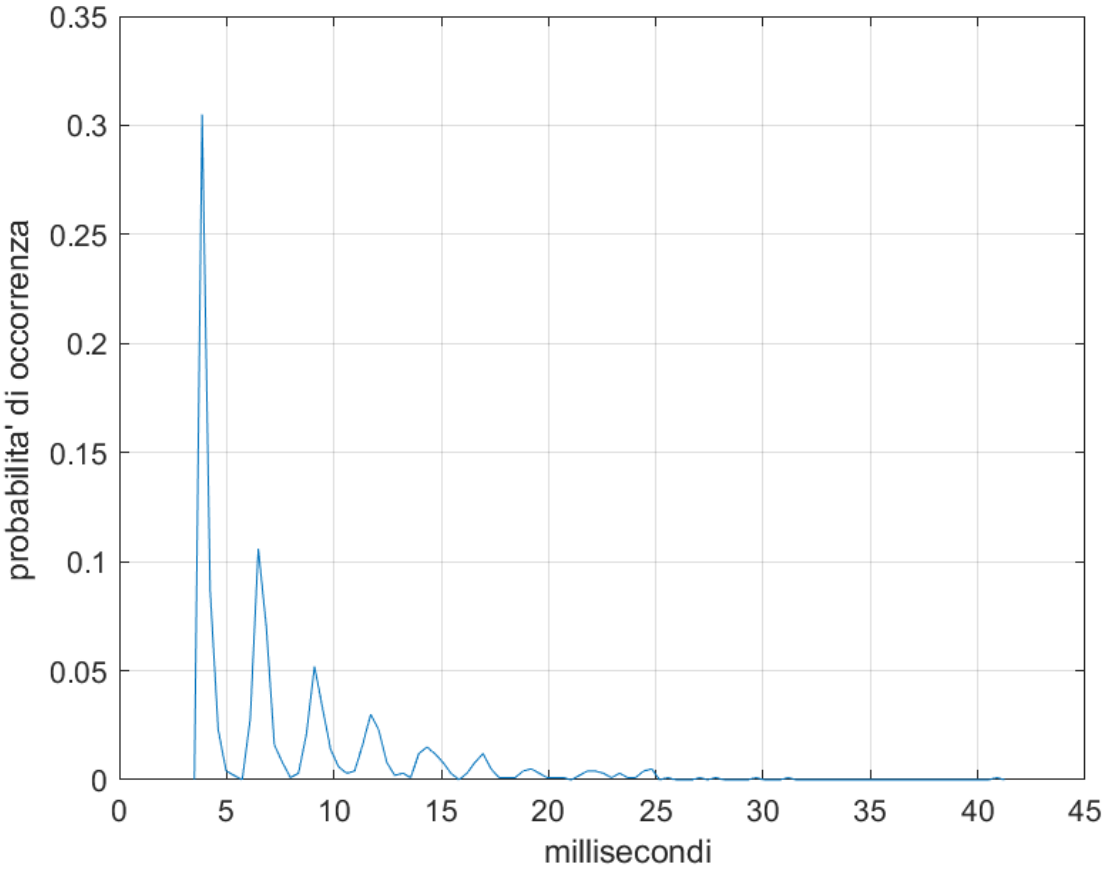

Figura 5.2: Probabilità di occorrenza stimata del tempo di generazione della chiave in millisecondi

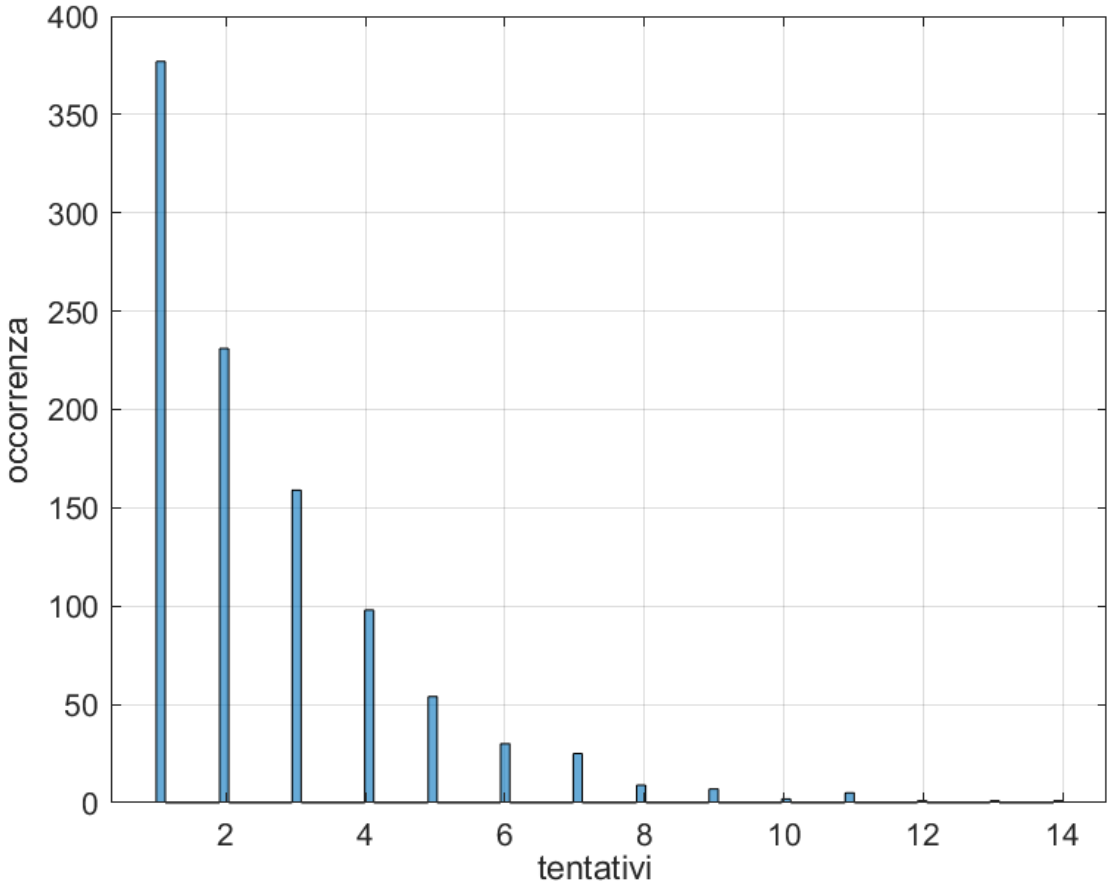

Figura 5.3: Istogramma del numero di tentativi per ottenere una matrice **E** valida

5.1.3 Tempo di calcolo per la generazione della firma

Così come per gli altri algoritmi, anche quello di *signature_generation* è stato eseguito 1000 volte, al fine di ottenere una stima del tempo medio e del numero di tentativi richiesti per ogni firma (si ricorda, infatti, che in fase di generazione di firma si utilizza il rejection sampling per scartare firme non sicure).
Per ogni iterazione, sono stati registrati la durata ed il numero di tentativi necessari per ottenere una firma valida.

I risultati ottenuti dallo script *test_signature_istogramma.m* per la stima del tempo medio sono i seguenti:

- 137 secondi per 1000 esecuzioni per generazione di una firma valida; il valore medio è quindi 137 ms;
- la varianza è risultata pari 17000 ms^2.
- la media dei tentativi per generare una firma valida è di 206.

Questi risultati evidenziano come il tempo richiesto per generare la firma sia sensibilmente più grande rispetto al tempo necessario per la generazione della chiave.
Questo risultato è conforme a quanto ci si aspettava. I due algoritmi utilizzano infatti, essenzialmente, la stessa tipologia di operazioni matriciali (trasposizioni e prodotti), ma l'algoritmo di generazione della firma invoca anche funzioni hash, che possono richiedere un tempo discretamente alto (specialmente se richiamate da Matlab).
Inoltre, per i parametri considerati, si ha che il tasso di rejection in sede di signature generation è più alto di quello che si ha in sede di key generation. In altre parole, ci si aspetta che il numero di firme scartate (prima di generarne una valida) sia, in media, più alto del numero di matrici $\boldsymbol{E}$ che vengono testate prima di ottenerne una valida.

Le Figure 5.4 e 5.5 riportano gli istogrammi relativi, rispettivamente, al tempo di generazione della firma ed al numero di tentativi fatti. Come si vede dalle figure, nel 65% delle prove si può ottenere una firma valida entro 206 tentativi, ovvero entro 137 ms, ricavando così che la proporzionalità tra loro (cioè il legame tempo-tentativi) è di 1,5 ms per tentativo. D'altro canto, nel 0,5% dei casi, esistono generazioni che richiedono più di 1000 tentativi.

5.1.4 Nota sul tempo di generazione della firma

Come già detto, Matlab non è ottimizzato per eseguire calcoli come l'hash, però è molto efficiente per eseguire calcoli sulle matrici. Nell'implementazione dell'algoritmo di generazione della firma è stata richiamata la funzione *DataHash* contenente l'esecuzione di un sottoprogramma di altro linguaggio di programmazione (Java), e questo procedimento non nativo ha determinato un aumento del tempo di esecuzione e, pertanto, una diminuzione dell'efficienza dell'intero algoritmo. Con il supporto del modulo *profile*, il quale consente di visualizzare il tempo speso da ogni funzione durante una esecuzione, è stata verificata l'occupazione nel tempo delle funzioni chiamate dall'algoritmo di generazione della firma. Come si vede nella Figura 5.6, generata direttamente dal modulo *profile*: la barra "signature generation" corrisponde al tempo totale, la barra "generate_c" ci dice quanto tempo del *signature_generation* è occupato dalla funzione *generate_c*, dove, al suo interno la quasi totalità del tempo è stato consumato dal calcolo dell'hash (*DataHash*) che sono molto costose. Inoltra si può affermare che, tolto il tempo speso su *generate_c*, solo una piccola parte del tempo del *signature_generation* è destinato alle operazioni sulle matrici; ciò a conferma dell'efficienza di Matlab nell'eseguire tali operazioni.

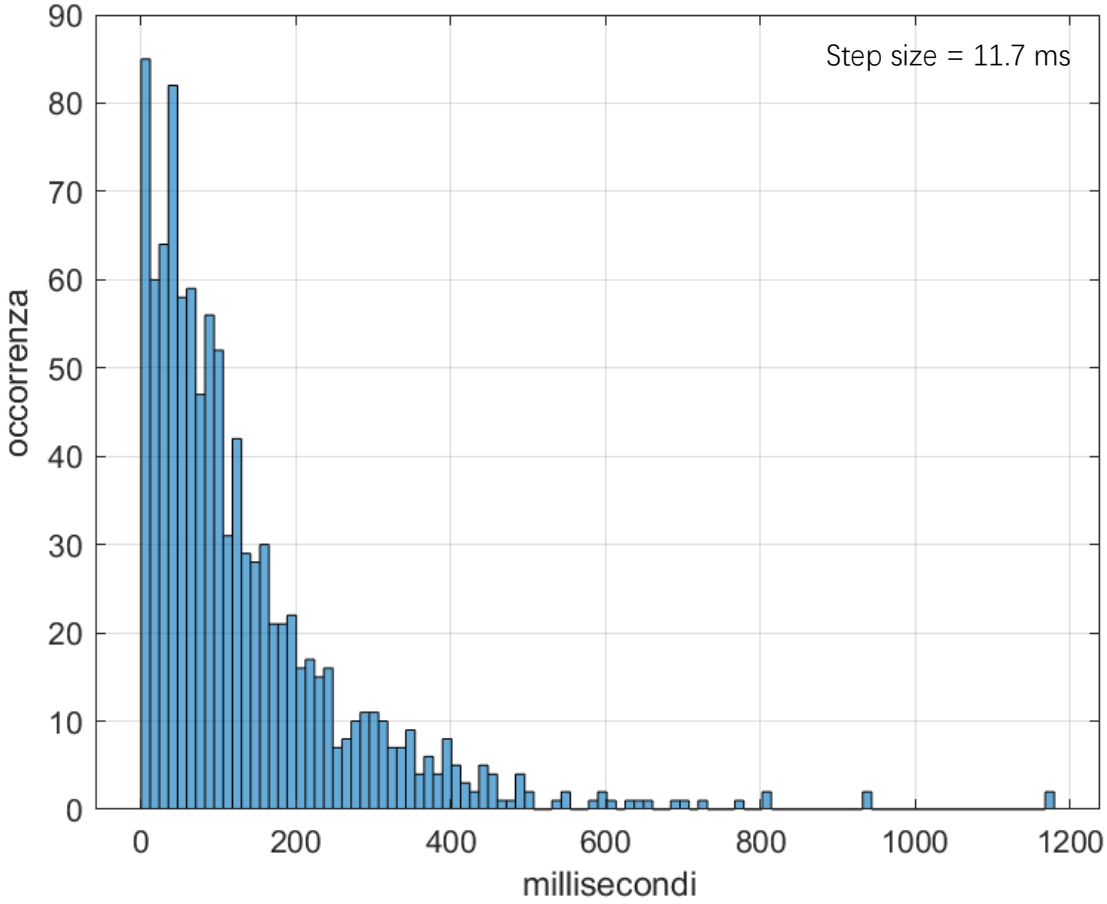

Figura 5.4: istogramma tempo generazione firma in millisecondi

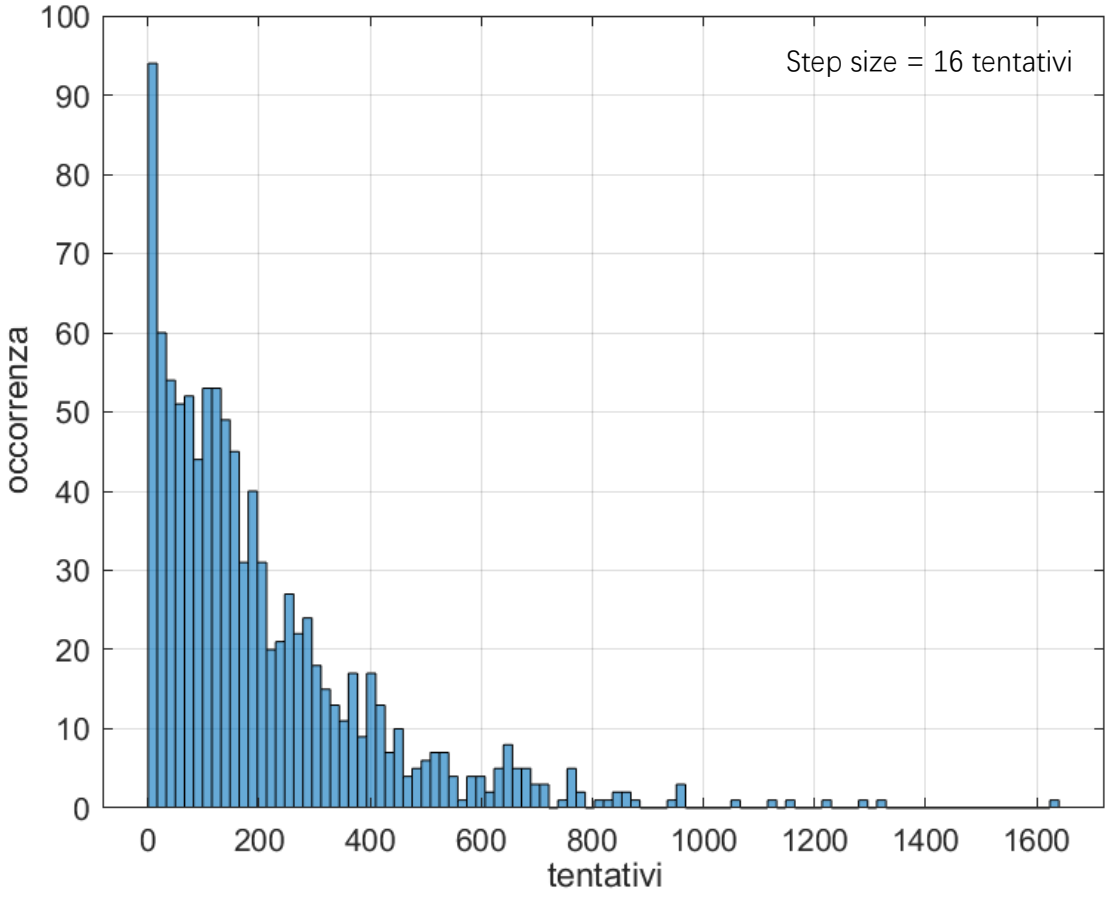

Figura 5.5: istogramma numero tentativi generazione firma

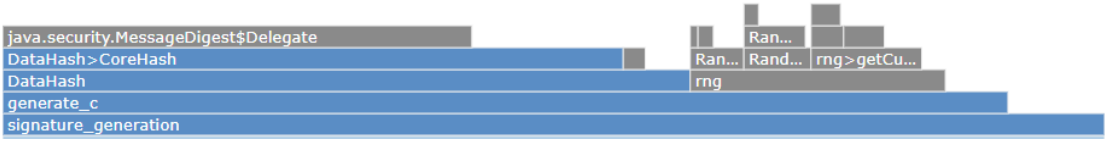

Figura 5.6: Occupazione del tempo della singola funzione utilizzata per la generazione della firma

5.1.5 Tempo di calcolo per la verifica della firma

Per provare l'algoritmo della verifica della firma, e per verificarne la funzionalità, è stato scritto un apposito script *test_verification.m*. Nello script è stata prima lanciata la generazione della chiave, quindi la generazione della firma e infine la verifica della firma utilizzando la chiave pubblica.
Per calcolare il tempo medio necessario per eseguire la verifica, è stato messo a punto uno script *test_verification_time.m* dove i tre algoritmi dello schema vengono ripetuti 1000 volte. Il cronometro *tic-toc*, già descritto in precedenza, consente di registrare il tempo di esecuzione. Inoltre lo script registra il risultato di ogni verifica (ovvero il fatto che la verifica sia andata o meno a buon fine).
I risultati di questo script sono i seguenti:

- la media del tempo di verifica della firma è di 1 ms;
- la varianza è di 0.1 ms^2
- In tutti i casi, la verifica è risultata valida, ciò conferma la funzionalità dell'algoritmo implementato.

Nella Figura 5.7 è riportato l'istogramma relativo al tempo di esecuzione della verifica della firma. Come si vede, la distribuzione è molto concentrata e raccolta intorno ai valori bassi. Questa conclusione era comunque attesa, considerando che l'algoritmo deve solo effettuare il confronto tra due vettori.

È stato calcolato anche il tempo medio di esecuzione per l'intera procedura, cioè il tempo necessario per generare la chiave, generare la firma e verificarla. Con una piccola modifica dello script *test_verification_time.m*, viene registrato il tempo dei tre algoritmi; il risultato ottenuto è 182 ms.
La figura 5.8 mostra infine l'istogramma relativa al tempo di esecuzione complessivo dei tre algoritmi. Come si vede, il grafico assomiglia alla Figura 5.4, perché la latenza del calcolo è dominata dall'algoritmo di generazione della firma.

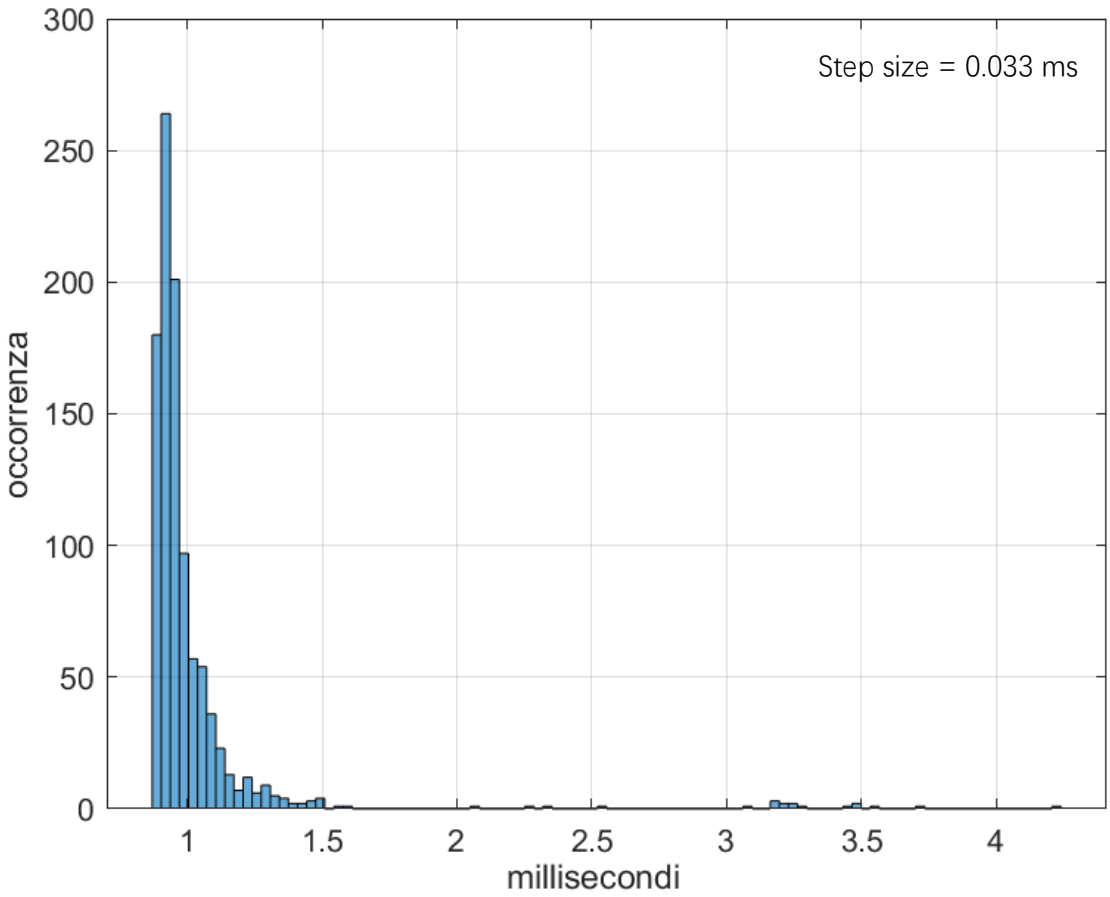

Figura 5.7: Istogramma del tempo di verifica della firma in millisecondi

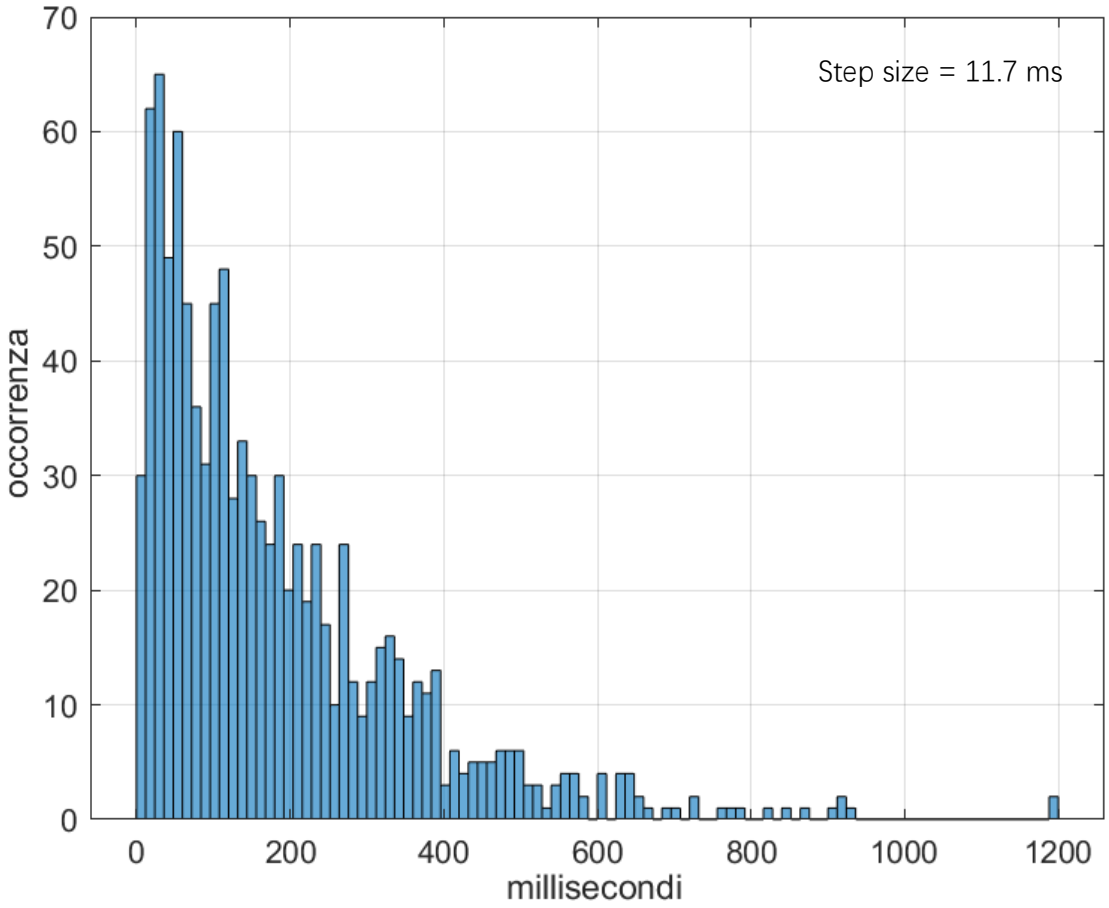

Figura 5.8: Istogramma del tempo di esecuzione complessivo dei tre algoritmi in millisecondi

5.2 Analisi con parametri ottimizzati

L'analisi della Sezione 5.1 è stata quindi ripetuta utilizzando i valori ottimi delle variabili ricavati in [10]. Anche in questo caso la sicurezza equivalente raggiunta è pari a 128 bit. I valori scelti sono i seguenti:

- $n = 190$,
- $k = 133$,
- $r = 57$,
- $q = 16381$,
- $w_C = 90$,
- $w_E = 77$,
- $t_E = 74$,
- $b = 171$,
- $\gamma = 2400$,
- $\bar{\gamma} = 2323$.

Uno degli obiettivi dell'ottimizzazione in [10] è stato quello di minimizzare la dimensione della chiave pubblica, cioè le dimensioni di $\{\mathbf{H}, \mathbf{S}\}$; si osserva che la dimensione del campo finito è stata fissata a q = 16381. Mentre con i parametri non ottimizzati le dimensioni erano 100×400 per **H** e 218×100 per **S**, corrispondenti ad un totale di 61800, ora con il nuovo set di parametri si è arrivati a $57 \times 190 + 171 \times 57 = 20577$. Pertanto, la chiave pubblica è stata ridotta di un fattore circa pari a 3.

5.2.1 Tempo di esecuzione per la generazione della chiave

I risultati ottenuti per la generazione della chiave utilizzando i parametri ottimizzati sono i seguenti:

- 1337 secondi (22 minuti e 17 secondi) per 500 esecuzioni dell'algoritmo di generazione delle chiavi valide; in media 2.7 secondi.

Nella Figura 5.9 è mostrato l'istogramma del tempo di generazione della chiave. Dalla figura si può ricavare che in circa il 68% dei casi la generazione della chiave valida viene completata entro 3 secondi, ed entro 6 secondi per il 89% dei casi. Nel caso peggiore, tra quelli simulati, occorrono 20 secondi per ottenere la chiave valida.
Oltre alla tempo medio di generazione della chiave è stata calcolata anche la varianza, che risulta 8.5 s^2.
A causa dei parametri scelti, la generazione della chiave richiede un tempo e un costo computazionale molto maggiori rispetto al caso di variabili non ottimizzate.

5.2.2 Numero di tentativi per avere una matrice E valida durante la generazione della chiave

Il numero di tentativi per la generazione della matrice **E** valida è una informazione strettamente correlata al tempo di esecuzione, infatti il tempo di calcolo della generazione della chiave cresce proporzionalmente al numero di tentativi effettuati. La Figura 5.10 mostra un istogramma di 500 campioni relativa al numero di tentativi effettuati da ogni lancio della generazione della chiave utilizzando parametri ottimizzati. La figura ha un andamento simile a quello della precedente Figura 5.9, ovvero al tempo di calcolo per la generazione della chiave. Nel caso peggiore sono necessari più di 14000 tentativi per ottenere una matrice **E** valida, corrispondente a 20 secondi di esecuzione perché i due risultati sono il massimo valore per numero di tentativi e tempo di esecuzione derivati dalla stessa esecuzione dello script.
Dai risultati della simulazione, la media dei tentativi è di circa 1900, che a differenza del caso relativo ai parametri in [9] trattato nella Sezione 5.1, risulta di tre ordini di grandezza più grande.

Il risultato ottenuto dalla funzione $\left(1-\sum_{i=0}^{t_E-1}\binom{n}{i}\left(\frac{\omega_E}{b}\right)^i\left(1-\frac{\omega_E}{b}\right)^{n-i}\right)^{-b}$ invece è di 870 tentativi.

Rispetto al mismatch della simulazione e teoria per i parametri non ottimizzati, dove erano 2.58 e 2.493 tentativi che si poteva considerare trascurabile, qui il distaccamento è più che doppio, una possibile ragione è dovuta al fatto dell'approssimazione fatta sulla formula, che ha trascurato le correlazioni fra le righe.

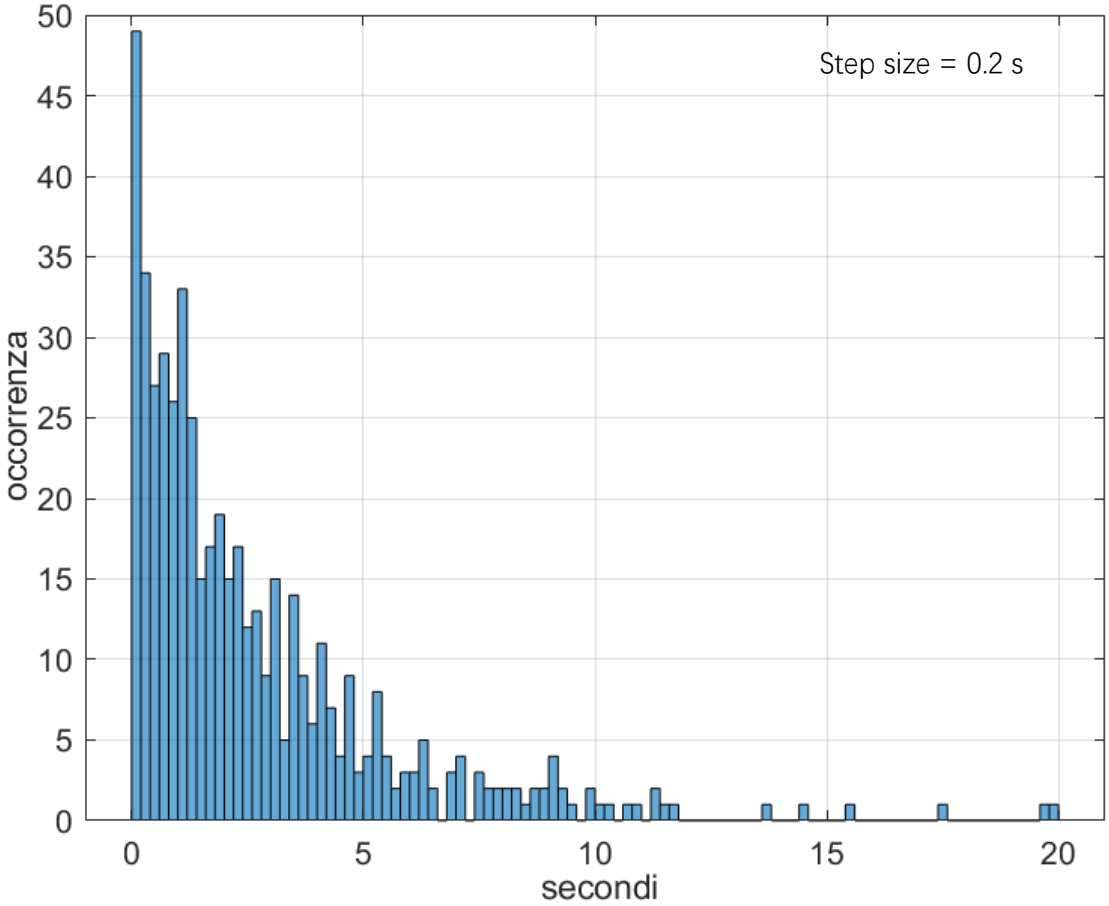

Figura 5.9: Istogramma del tempo di generazione della chiave in secondi, nel caso di parametri ottimizzati

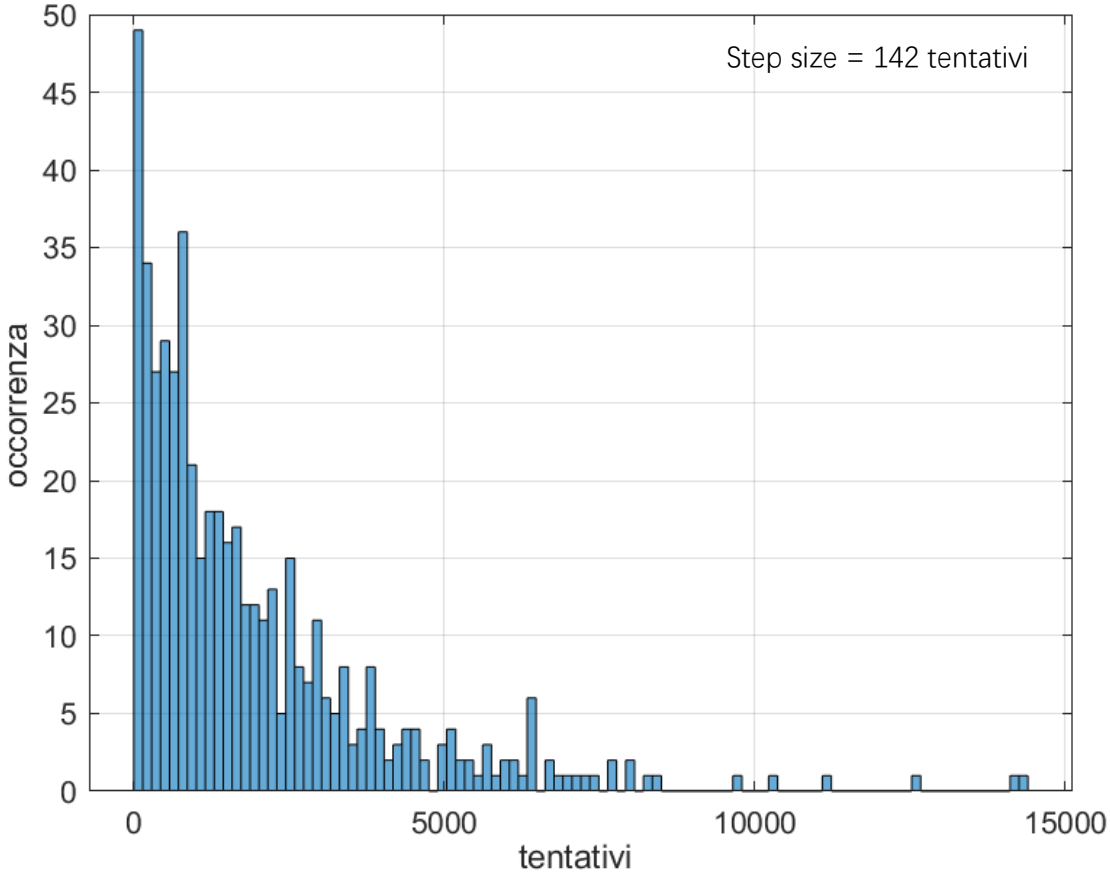

Figura 5.10: Istogramma del numero di tentativi per ottenere una matrice **E** valida, nel caso di parametri ottimizzati

5.2.3 Tempo di esecuzione per la generazione della firma

Il tempo di esecuzione per la generazione della firma utilizzando o parametri ottimizzati è risultato pari a:

- 124 secondi per 500 esecuzioni di generazione della firma; in media 248 ms;
- con una varianza di 71700 ms^2.

La Figura 5.11 mostra l'istogramma relativa al tempo di generazione della; come si vede dalla figura, nell' 87% circa dei casi la generazione della firma può essere completata entro mezzo secondo, un piccolo 10% tra mezzo secondo e un secondo, e solo una esecuzione ha richiesto più di 2 secondi.
Inoltre, nella Figura 5.12 è mostrato la versione dell'istogramma della Figura 5.11 (in blu) sovrapposto al caso per i parametri utilizzati nella Sezione 5.1 (in arancione). Come si vede, l'intervallo temporale che serve per rappresentare il blu è quasi il doppio di quella arancione ed è meno concentrato verso l'origine.

5.2.4 Numero di tentativi per ottenere una firma valida

Utilizzando i parametri ottimizzati, la media dei tentativi per ottenere una firma è di 520, cioè più del doppio rispetto al caso di utilizzo di parametri non ottimizzati.
In Figura 5.13 è presentato un istogramma del numero di tentativi per ottenere una firma valida durante una generazione della firma con i parametri ottimizzati; anche in questo caso, la maggior parte delle firme valide (86%) richiedono meno di 1000 tentativi, mentre è quasi trascurabile la quantità di firme (2%) che richiedono più di 2000 tentativi.
La Figura 5.14 mostra la sovrapposizione dell'istogramma relativo al numero di tentativi con parametri ottimizzati (in blu) con il caso per i parametri non ottimizzati (in arancione). Similmente al confronto precedente, l'istogramma blu ha un andamento meno ripido rispetto all'arancione.

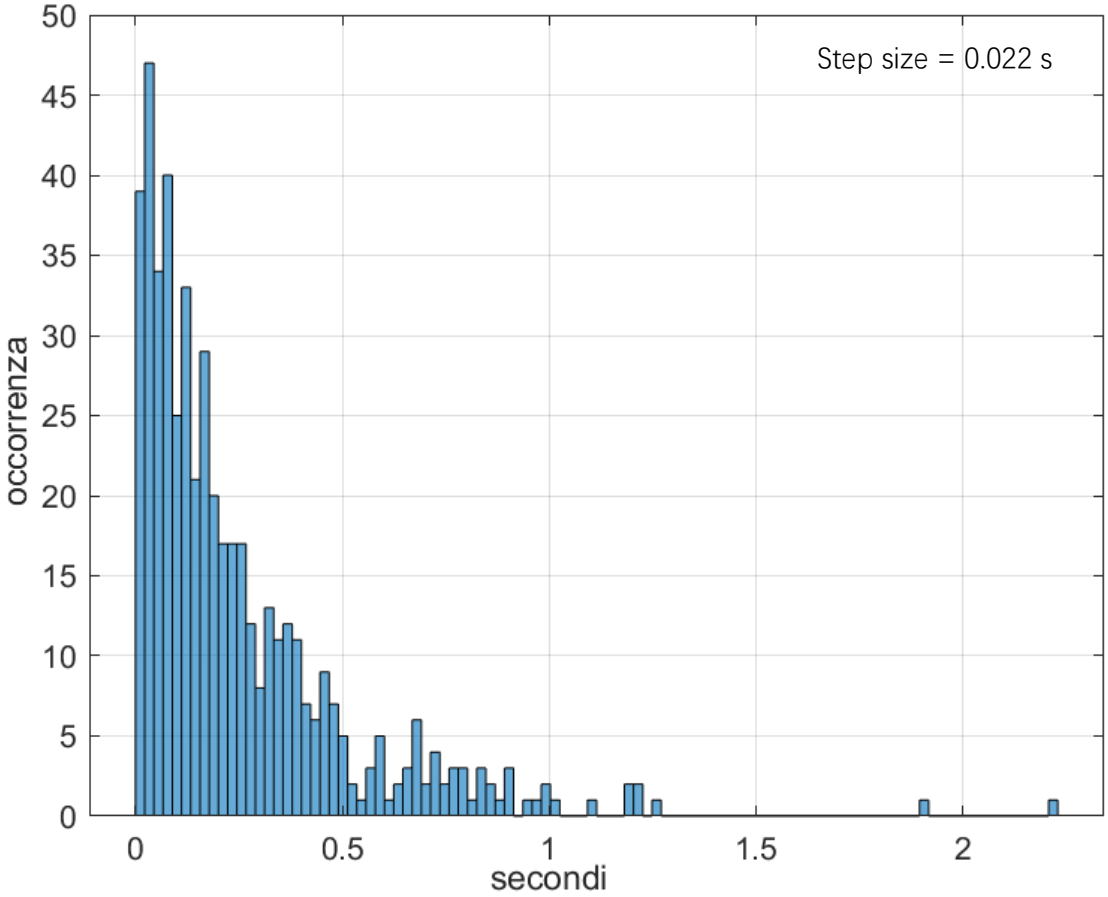

Figura 5.11: Istogramma del tempo di generazione firma in secondi con parametri ottimizzati

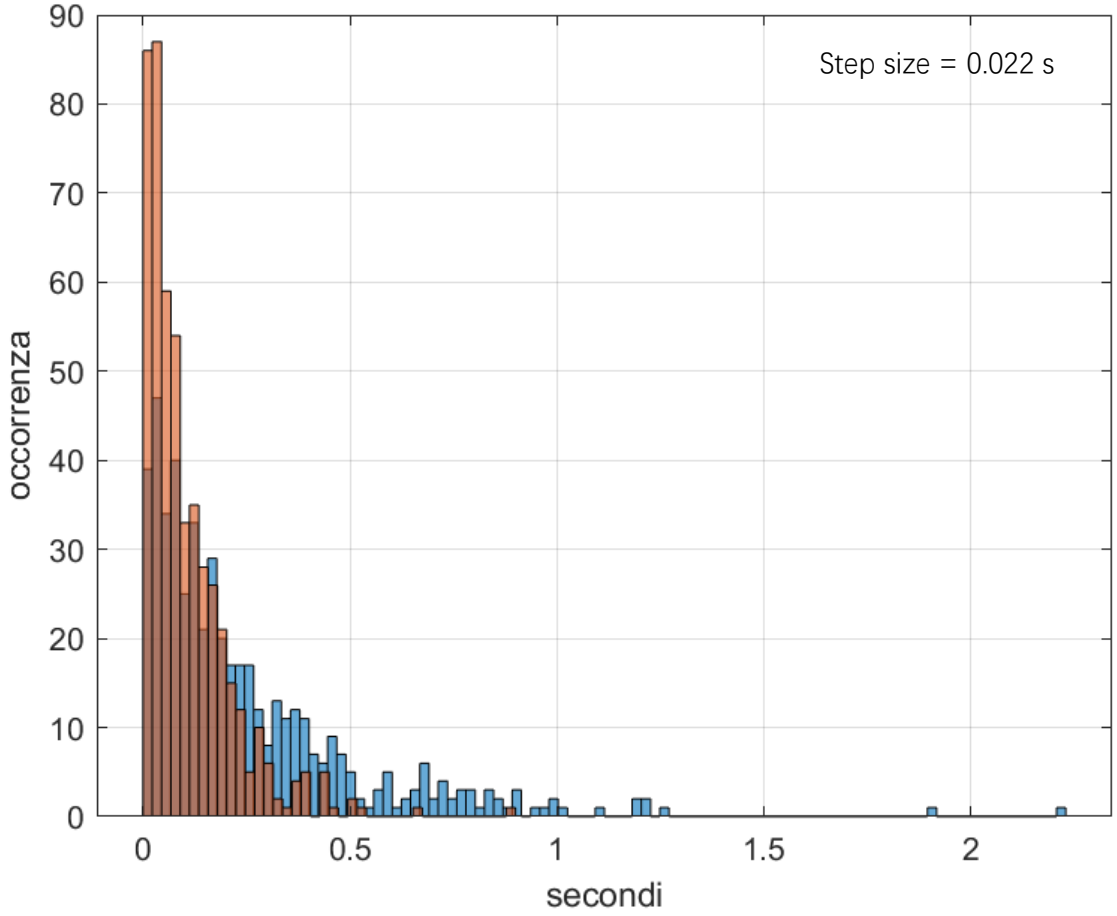

Figura 5.12: Confronto tra i tempi di generazione della firma, in secondi, utilizzando i parametri ottimizzati e quelli non ottimizzati

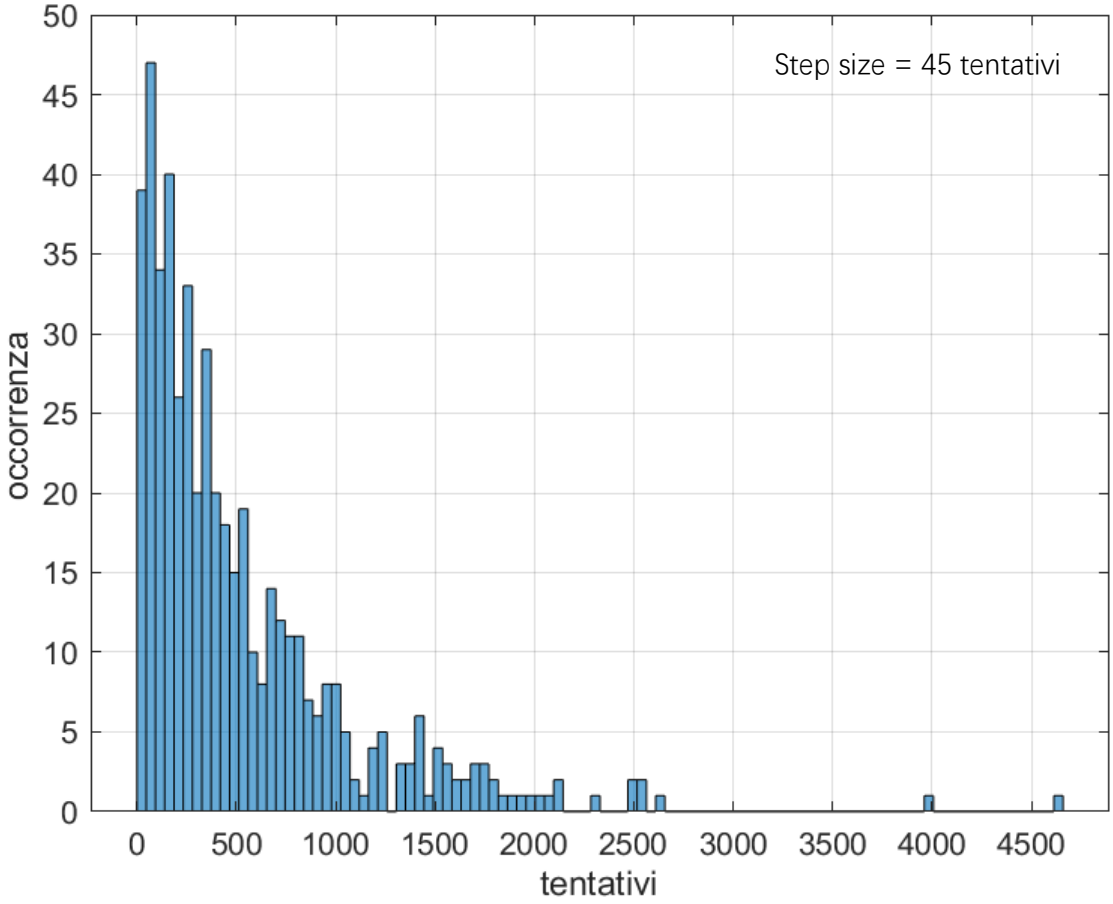

Figura 5.13; Istogramma del numero di tentativi di generazione di una firma valida con parametri ottimizzati

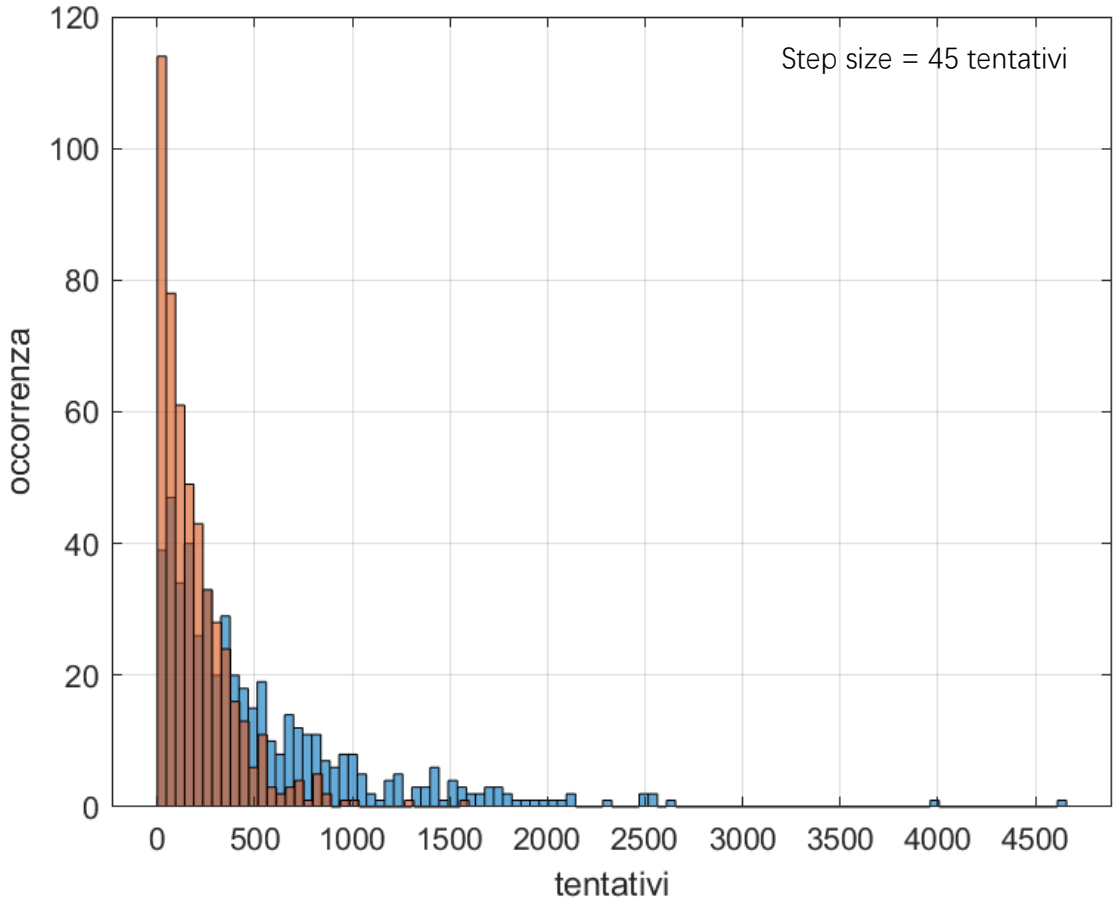

Figura 5.14: Confronto il numero di tentativi per la generazione della firma, utilizzando i parametri ottimizzati e quelli non ottimizzati

5.2.5 Tempo di esecuzione della verifica della firma

L'accorciamento della dimensione della firma, posto come obiettivo della procedura di ottimizzazione in [10], ha invece, come prevedibile, un impatto positivo nell'algoritmo di verifica della firma. In effetti, la Figura 5.15 mostra che il tempo necessario per la verifica della firma con parametri ottimizzati è quasi sempre inferiore a 500 μs, invece di 1 ms nel caso precedente (con parametri non ottimizzati).

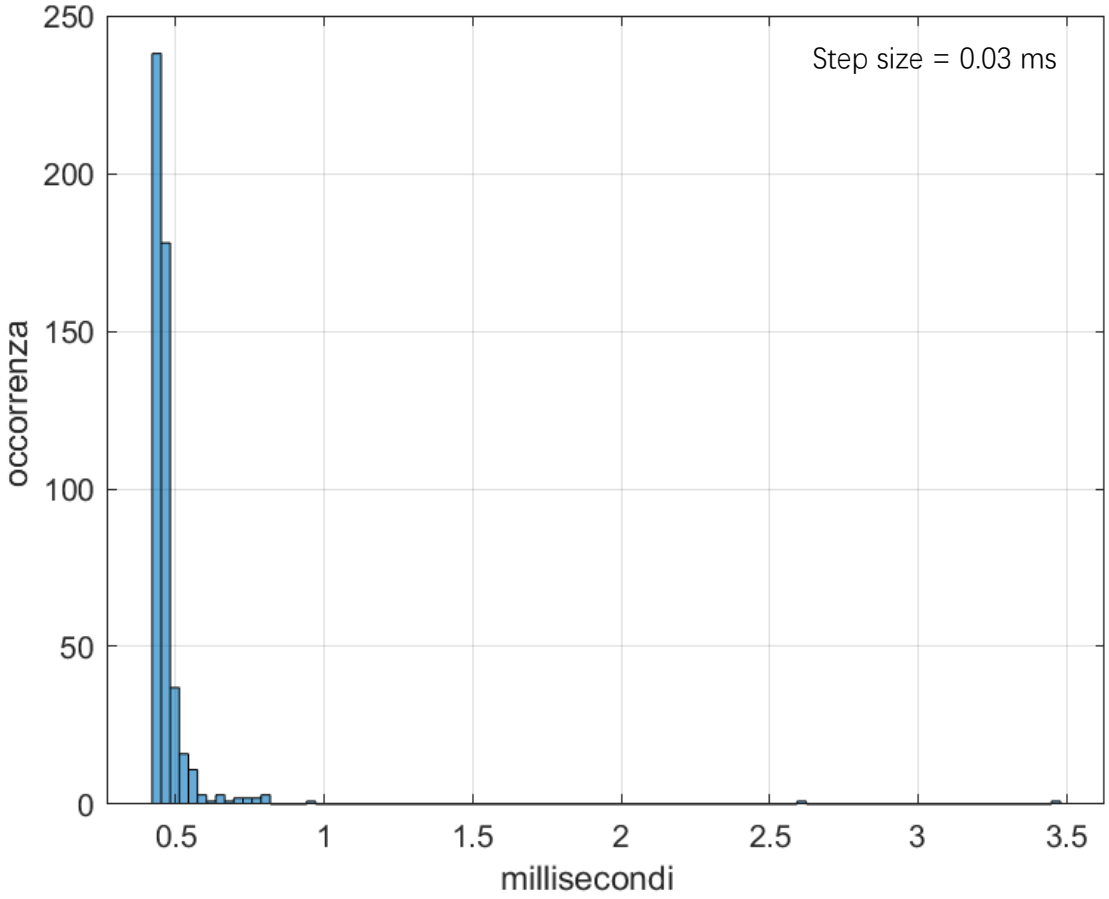

Figura 5.15: Istogramma del tempo di verifica della firma in millisecondi con parametri ottimizzati

Conclusioni

In questa tesi è stato implementato uno schema di firma digitale post-quantum, basato su codici non binari.
Il lavoro svolto è stato suddiviso in due fasi: in una prima fase, l'algoritmo è stato implementato al fine di verificane la correttezza formale e procedurale.
In una seconda fase, l'algoritmo è stato sottoposto a misurazioni intensive, con l'obiettivo di misurare importanti figure di merito come tempi di calcolo e tassi di rejection (sia in fase di firma, che in fase di generazione della firma).
In particolare, in questa seconda fase sono stati considerati due set di parametri diversi, entrambi progettati per garantire un livello di sicurezza pari a 128 bit.
Il set che è stato chiamato "ottimizzato" è in grado di raggiungere chiavi pubbliche di dimensioni ridotte, a patto però di avere un tasso di rejection in sede di signature generation elevato (infatti, in media l'algoritmo di generazione della firma richiede circa 1900 tentativi).

Per entrambi i set, si sono andati a misurare i tempi medi richiesti dai tre algoritmi caratteristici dell'algoritmo, ovvero: generazione della chiave, generazione della firma e verifica della firma.
I risultati ottenuti mostrano che i tempi dell'algoritmo sono molto interessanti: infatti, per il set ottimizzato, una esecuzione completa dell'algoritmo richiede un tempo medio pari a 2.92 s. Questo risultato è fortemente incoraggiante.
Infatti, come è già stato ribadito nel lavoro di tesi, Matlab non è il linguaggio adatto per implementazioni ottimizzate di algoritmi di crittografia.
Una prova di questo fatto è stata ottenuta tramite lo strumento profiler di Matlab, che ha mostrato come una grande percentuale del tempo necessario per produrre una firma sia occupata dalle funzioni di hash, che invece sono fortemente ottimizzate su altri linguaggi di programmazione (come C o Python).
I risultati in questa tesi sembrano quindi suggerire che un'implementazione ad-hoc in linguaggi di più basso livello possa raggiungere tempi fortemente competitivi ed in linea con quelle che, ad oggi, sono le richieste ed i requisiti tecnologici per applicazioni in cui algoritmi di firma digitale sono richiesti.

Riferimenti

[1] Rivest, Ronald L., Adi Shamir, and Leonard Adleman. "A method for obtaining digital signatures and public-key cryptosystems." *Communications of the ACM* 21.2 (1978): 120-126.

[2] Diffie, Whitfield, and Martin Hellman. "New directions in cryptography." *IEEE transactions on Information Theory* 22.6 (1976): 644-654.

[3] ElGamal, Taher. "A public key cryptosystem and a signature scheme based on discrete logarithms." *IEEE transactions on information theory* 31.4 (1985): 469-472.

[4] Koblitz, Neal. "Elliptic curve cryptosystems." *Mathematics of computation* 48.177 (1987): 203-209.

[5] Miller, Victor S. "Use of elliptic curves in cryptography." *Conference on the theory and application of cryptographic techniques*. Springer, Berlin, Heidelberg, 1985.

[6] Jurvetson, Steve. "How a quantum computer could break 2048-bit RSA encryption in 8 hours." *MIT Technology Review, May* 30 (2019): 9.

[7] Bernstein, Daniel J. "Grover vs. McEliece." *International Workshop on Post-Quantum Cryptography*. Springer, Berlin, Heidelberg, 2010.

[8] Hsu, Jeremy. "How the United States Is Developing Post-Quantum Cryptography." (2019).

[9] Marco Baldi, Franco Chiaraluce, and Paolo Santini. "Code-based signatures without trapdoors through restricted vectors." *IACR Cryptol. ePrint Arch.* 2021 (2021): 294.

[10] Giuliani Rebecca. "Dimensionamento ed ottimizzazione di uno schema post-quantum di firma digitale basato su codici." Tesi di Laurea, 2021.

Appendice

Nell'appendice viene riportato il codice che è stato utilizzato per implementare l'algoritmo di firma, e per testarne le funzionalità.

key_generation.m **(generazione delle chiavi S, H ed E, senza verifica sulla matrice E)**

```
function [S, H, E] = key_generation(q,n,k,b,wE,tE)

r = n-k;
qm1 = q-1; % rappresenta il valore -1 nel campo finito;

%% generazione della chiave pubblica H %%
    P = randi([0 q-1],[r k]); % P matrice di dimensione r*k,su ogni
posizione un numero casuale tra 0 e q-1
    H = [eye(r) P];     % concatenazione di matrice identità r*r con P

%% generazione della chiave segreta E %%
    E= zeros(b,n);  % E per il momento è una matrice di zeri di dimensione
b*n

% ciclo for applicato su tutte le colonne di E;
for i = 1:n
    pos = randperm(b,wE); % prelevo casualmente wE elementi nel [1:b] e li
uso come indice
    a = randi([0 1] , [wE 1]);
    a(a==0)=qm1; % scambio gli 0 con qm1;
    %a = mod(2*randi([0 1] , [wE 1])-1,q); alternativa;
    E(pos,i) = a; % alle posizione pos di E assegno i valori di a
end

%% generazione della chiave pubblica S %%
S = mod(E*H',q); % mod fa il modulo degli elementi di E*H' , ' indica la
trasposizione
end
```

key_check.m (verifica della matrice E)

```
function OK = key_check(E,b,n,tE)

rlttE=0; % rows less than tE, numero delle righe che hanno elementi non
nulli minori di tE
N=n; % elementi presenti in riga
minv=n; % valore minino di N
%
% contrllo su ogni riga di E
for k = 1:b
    N=nnz(E(k,:)); % nnz restituisce il numero di elementi non nulli della
k-esima riga di E
    minv=min(N,minv);
    if N<tE   % incremento rlttE se  N è minore di tE
       rlttE = rlttE+1;
    end
end

if (rlttE == 0) % la matrice è valida se rlttE risulta 0
    OK = true;
else
    OK = false;
end
```

valid_key_generation.m **(generazione della chiave con matrice E sicuramente valida)**

```
function [S, H, E] = valid_key_generation(q,n,k,b,wE,tE)

r = n-k;
qm1=q-1; % rappresenta il valore -1 nel campo finito;

%% generazione della chiave pubblica H %%
P = randi([0 q-1],[r k]);
H = [eye(r) P];

%% generazione della chiave segreta E valida%%
found_validE = false;
while found_validE==false  % finché non trova una matrice E valida
    E= zeros(b,n);

    for i = 1:n
        pos = randperm(b,wE); % prelevo casualmente wE elementi nel [1:b] e
li uso come indice
        a = randi([0 1] , [wE 1]);
        a(a==0)=qm1;
        E(pos,i) = a;
    end

    %% verifica della matrice E %%
    rlttE=false; % rows less than tE di E

    for ii = 1:b
        N=nnz(E(ii,:));     % elementi presenti in riga di E
        if N<tE
           rlttE = true;
           break
        end
    end
    if rlttE==false    %se non ci sono righe minori di tE allora la matrice
E è valida
        found_validE=true; % matrice E valida trovata
    end

end % fine while
%% generazione della chiave pubblica S %%
S = mod(E*H',q);
end
```

test_firma_istogramma.m **(script per testare generazione della chiave, con uscita un istogramma sul tempo di generazione)**

```
clear all; close all; clc;
%% default param %%
n = 400;
k = 300;
r = n-k;
q = 16381;
b = 218;
wE = 46;
tE = 64;
%% Key generation test %%
num_test = 1000;        % numero di prove
time_gen = zeros(1,num_test);  % asse temporale
tic  % partenza cronometro
for i = 1:num_test
    [S,H,E] = valid_key_generation(q,n,k,b,wE,tE);
    time_gen(i)=toc;
end
toc  % fine cronometro

time_gen(2:num_test)=time_gen(2:num_test)-time_gen(1:num_test-1);   %%
vettore dei tempi di tutte le generazioni
x=time_gen;
%%Calcolo il minimo e massimo di x, e suddivido l'intervallo in num_step
%%sottointervalli
min_val = min(x);
max_val = max(x);
num_step = 100;

asse_x = min_val:((max_val-min_val)/num_step):max_val;
valori_prob = zeros(1,length(asse_x)); %%[];
%per ogni sottointervallo, calcolo quanti campioni di x cadono in quel
%sottointervallo. Divido per la lunghezza di x e così trovo la probabilità
for i = 1:length(asse_x)-1
    min_x = asse_x(i);
    max_x = asse_x(i+1);
    a = nnz((x>=min_x)&(x<max_x));
    valori_prob(i) = a/length(x);%%[valori_prob,a/length(x)];
end
plot(asse_x,valori_prob) % generazione grafico probabilità di occorrenza
figure;
histogram(x,num_step)  % generazione istogramma
```

generate_c.m (generazione dell'hash c, utilizzata per la generazione della chiave e la verifica della chiave)

```
function c = generate_c(b,wc,q,m,sy)
qm1 = q-1;

%%concatenazione di m e sy, m_conc_sy per generare l'hash, si può
concatenarle anche senza convertirle in stringhe.
m_conc_sy = [num2str(m),num2str(sy)];
seed_hex = DataHash(m_conc_sy); %si può assegnare a seed direttamente m
concatenato sy

%conversione dell'hash in decimale, assegnato come seed per il rng
seed_dec = hex2dec(seed_hex(1:8));
rng(seed_dec); % rng: seme del generatore pseudorandom

%%generazione di c, con la stessa strategia per generare una colonna di E
c = zeros(1,b);
pos_ones = randperm(b,wc);
a = randi([0 1] , [wc 1]);
a(a==0) = qm1;
c(pos_ones) = a;
end
```

signature_generation.m **(generazione della firma, con uscita z, c, e numero di tentativi eseguiti)**

```
function [z,c,not_valid_counter] =
signature_generation(q,m,n,b,gamma,gamman,H,E,wC)
not_valid_counter=0; %% contatore tentativi
found_valid_z=false;
while found_valid_z==false % finché non trovo z valido, continuo il loop
    not_valid_counter=not_valid_counter+1;

    %% generazione del vettore y e sy %%
    y= randi([-gamma gamma], [1 n]);
    sy=mod(y * H' , q); %% sy vettore riga.

    %% generazione dell'hash c %%
    c = generate_c(b,wC,q,m,sy);

    %% generazione firma z %%
    z= mod(c*E+y,q);
    %not_valid=z(z>gamman & z<q-gamman);
    not_valid=find(z>gamman & z<q-gamman,1); %% trova 1 elemento che
soddisfa l'espressione
    if isempty(not_valid)
        found_valid_z=true;
        break; % esco dal while quando trova un z valido (not_valid vuoto)
    end
end

fprintf('times tried: %d\n',not_valid_counter) % print sulla console il
numero di tentativi provati
end
```

test_signature_istogramma.m **(script per testare la generazione della firma, con uscita un istogramma relativa al tempo generazione e numero di tentativi provati)**

```
clear all; close all; clc;
%% default param %%
n = 400;
k = 300;
r = n-k;
q = 16381;
b = 218;
wE = 46;
tE = 64;
%% parametri generazione firma %%
wC = 67;
m=123456; %% messaggio
gamma=3420;
gamman=3375;

%% generazione della chiave + firma %%
[S, H, E] = valid_key_generation(q,n,k,b,wE,tE);

num_test=1000;
time_signature_gen=zeros(1,num_test); % vettore tempo generazione
num_attempts=zeros(1,num_test); % vettore numero di tentativi fatti
tic
for i = 1:num_test
     [z,c,not_valid_counter] =
signature_generation(q,m,n,b,gamma,gamman,H,E,wC);
    time_signature_gen(i)=toc;
    num_attempts(i)=not_valid_counter;
end
toc

%% vettore dei tempi di esecuzione di tutte le generazioni %%
time_signature_gen(2:num_test)=time_signature_gen(2:num_test)-
time_signature_gen(1:num_test-1); % assegno la differenza tra i due toc

histogram(time_signature_gen,100)
figure; % nuova figura
histogram(num_attempts,100)
%% risultato
% media numero tentativi 206
% media tempo : 0.137 secondi
```

signature_verification.m (verifica della firma)

```
function OK = signature_verification(q,m,gamman,z,c,H,S)
OK=false;

not_valid=find(z>gamman & z<q-gamman,1); %% trova 1 elemento che soddisfa
l'espressione
if ~isempty(not_valid) %% se not_valid non e' vuoto allora return OK=false
    return;
end

sy = mod( z * H' - c * S , q);
wC=nnz(c); %% trovo wC per passarlo a generate_c
b=length(c); %% stessa cosa per b

c1=generate_c(b,wC,q,m,sy);

equals=isequal(c,c1); %% isequal compara elemento per elemento se sono
uguali, restituisce true se sono tutti uguali, altrimenti false
if equals
    OK=true; % OK=true se tutti i elementi di equals sono 1
end

end
```

test_verification.m **(test su tutti i 3 algoritmi, generazione della chiave, generazione della firma, e verifica della firma usando la chiave pubblica generata)**

```
clear all; close all; clc;
%% default param %%
n = 400;
k = 300;
r = n-k;
q = 16381;
b = 218;
ell = b;
wE = 46;
tE = 64;
%% parametri generazione firma %%
wC = 67;
gamma=3420;
gamman=3375;
%rng(6);
m=randi([0 q],[1 b]); %% messaggio

%% generazione della chiave + firma %%

tic
[S, H, E] = valid_key_generation(q,n,k,b,wE,tE);
toc
[z,c] = signature_generation(q,m,n,ell,gamma,gamman,H,E,wC);
toc
%% verifica della firma %%
OK=signature_verification(q,m,gamman,z,c,H,S); % risultato della verifica
toc
fprintf('result: OK is %d.\n' , OK);
```

test_verification_time.m (test per calcolare il tempo medio di verifica della firma)

```
clear all; close all; clc;
%% default param %%
n = 400;
k = 300;
r = n-k;
q = 16381;
b = 218;
ell = b;
wE = 46;
tE = 64;
%% parametri generazione firma %%
wC = 67;
gamma=3420;
gamman=3375;
%rng(6);
m=randi([0 q],[1 b]); %% messaggio

%% generazione della chiave + firma %%
num_test=1000;
OK=zeros(1,num_test); % vettore che dovrà salvare risultati della verifica
time_verification=zeros(1,num_test); % vettore tempo di verifica

for i = 1:num_test
    [S, H, E] = valid_key_generation(q,n,k,b,wE,tE);
    [z,c] = signature_generation(q,m,n,ell,gamma,gamman,H,E,wC);

    tic
    OK(i)=signature_verification(q,m,gamman,z,c,H,S);
    time_verification(i)=toc;
end
toc

histogram(time_verification,100);
```