Università Politecnica delle Marche

Facoltà di Ingegneria

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria Informatica e dell'Automazione

Tesi di Laurea

Progetto ed analisi di varianti relative a schemi di firma digitale basati su codici con restrizioni

Design and analysis of variants for digital signature schemes based on codes with restrictions

Relatore

Prof. Marco Baldi

Candidato

Valerio Procaccioli

Correlatore

Dott. Paolo Santini

Anno accademico 2022-2023

Indice

Elenco delle figure

Elenco dei listati

Abstract

La ragione che motiva la realizzazione di questo studio è intrinseca alla costante evoluzione delle minacce informatiche ed alla necessità imperante di adeguare le attuali tecniche crittografiche per affrontare le sfide in modo efficace. La ricerca di soluzioni più efficienti, oltre a rispondere alla sempre crescente complessità delle minacce, contribuisce significativamente al progresso generale nella progettazione di algoritmi più robusti e scalabili. Questo lavoro, pertanto, si configura non solo come uno sforzo teso al miglioramento delle prestazioni dello schema di firma digitale CROSS, attualmente in gara nella competizione NIST Post-Quantum, ma si inserisce anche in un contesto più ampio di adattamento e innovazione.

La riflessione critica e l'esplorazione di nuove prospettive costituiscono elementi fondamentali di un approccio dinamico alla sicurezza informatica, orientato verso la continua ricerca di soluzioni più avanzate ed efficaci. Il seguente studio di CROSS mira ad esplorare miglioramenti sensibili in termini di prestazioni, concentrandosi in particolare su due aspetti chiave: la dimensione della firma e la velocità di esecuzione dell'algoritmo.

Inizialmente, è stata condotta un'analisi relativa alle basi matematiche che sottendono la robustezza della procedura. Tuttavia, l'attenzione non si è limitata alla mera comprensione teorica; al contrario, è stata adottata una nuova prospettiva di modifica, concentrata sulla rimozione di una specifica sequenza di dati dal modello originale. Attraverso dimostrazioni matematiche solide ed una progettazione schematica rigorosa, la validità della revisione proposta è stata confermata, portando a miglioramenti tangibili nell'efficienza complessiva dello schema.

L'implementazione Python degli algoritmi ha svolto un ruolo essenziale nella completezza del progetto CROSS. Quest'ultima non solo ha fornito uno strumento pratico per sostenere l'approccio teorico, ma ha anche contribuito ad una migliore comprensione del contesto crittografico di riferimento. Parallelamente, è stata condotta un'analisi parametrica volta a identificare istanze valide e performanti per la nuova alternativa. Tale attività è finalizzata al rilevamento della versione più efficiente di CROSS, attraverso implementazioni di alto livello in combinazione con parametri adatti ai cambiamenti riscontrati.

Infine, è stata posta particolare enfasi sul confronto tra le versioni realizzate rispetto allo stato dell'arte attuale. Questa valutazione comparativa permette di mostrare in modo dettagliato gli aspetti cruciali della soluzione proposta, evidenziandone contemporaneamente eventuali criticità.

Nel dettaglio, la struttura della tesi è delineata come segue:

- **Capitolo 1: Introduzione** - Fornisce un contesto iniziale sulla firma digitale, presentando la notazione matematica ed i requisiti minimi necessari per una comprensione approfondita del lavoro.

- **Capitolo 2: CROSS** - Approfondisce lo schema di firma digitale CROSS, illustrandone il funzionamento, le basi concettuali ed il processo di analisi.

- **Capitolo 3: CROSS Single Commitment** - Presenta le proposte di miglioramento emerse durante l'analisi dello schema originale. Viene discussa nel dettaglio l'evoluzione teorica, seguita dall'implementazione completa in linguaggio Python di entrambe le versioni.

- **Capitolo 4: Profilazione del codice e risultati** - Esamina i parametri chiave per entrambe le versioni dello schema, identificandone i valori che conducono alle migliori prestazioni. La profilazione del codice, forte dei nuovi parametri, completa la sezione consegnando i risultati del lavoro svolto.

- **Conclusioni** - Rivede criticamente il lavoro svolto, evidenziando: punti di forza, opportunità, criticità e minacce; oltre a proporre idee innovative per futuri sviluppi nell'ambito delle firme digitale.

1

Introduzione

Nel primo capitolo viene esplorato il contesto generale relativo alle firme digitali, ponendo l'enfasi sulla rilevanza delle competizioni crittografiche indette dal NIST. Di seguito vengono ampiamente discussi: gli strumenti, le tecniche, la teoria ed i problemi alla base della ricerca condotta.

1.1 Contestualizzazione

La firma è storicamente un mezzo cruciale per attestare l'autenticità e l'approvazione di documenti, risentendo però di ampie vulnerabilità nel contemporaneo. Difatti, nel mondo digitale, la firma non rappresenta l'equivalente elettronico della firma autografa, bensì svolge un ruolo fondamentale nell'ambito della sicurezza e della legalità relative alle interazioni fra gli utenti.

La pratica della firma autografa, risalente a secoli di tradizione, caratterizza l'atto fisico di apporre una firma personale su un documento cartaceo per confermarne l'autenticità. Essa viene spesso associata all'identità unica del firmatario ed alla sua approvazione formale del contenuto di un documento. Nonostante ciò, con l'avvento delle nuove tecnologie, l'esigenza di adattare questo concetto tradizionale alla sfera informatica ha dato vita, inizialmente, alla firma biometrica (o grafometrica) e, di seguito, alla firma digitale.

Quest'ultima rappresenta un avanzamento tecnologico importante, che consente di generare in modo sicuro un dato identificativo, calcolato tipicamente attraverso l'uso di una primitiva crittografica asimmetrica (Fig.1.1). Essa è contraddistinta da un funzionamento basato su coppie di chiavi asimmetriche, dunque differenti (privata, pubblica), per ciascuna entità che volesse autenticarsi. Attraverso l'utilizzo della chiave privata è possibile apporre sul documento un sigillo che attesta l'univocità dell'azione eseguita; allo stesso modo la chiave pubblica dell'utente, fruibile da chiunque, permette di verificare la validità della marcatura apposta. Quindi, contrariamente alla firma cartacea, quella digitale si basa su algoritmi crittografici per garantire l'autenticità e l'innegabilità del firmatario. La sua evoluzione nell'era contemporanea, nonostante la moltitudine di variabili presenti, mantiene comunque un legame indissolubile con la firma autografa. Entrambe mirano a fornire un mezzo sicuro per identificare il firmatario, congiuntamente ad una sua volontà. Tuttavia, la firma digitale va oltre, offrendo vantaggi unici come la capacità di verificare l'origine e l'inte-

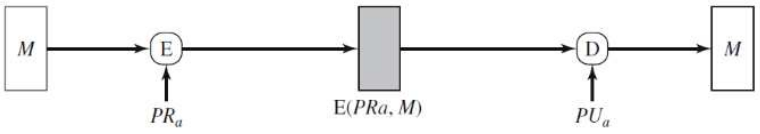

Figura 1.1: Architettura di una primitiva crittografica asimmetrica per la firma digitale

grità del documento, oltre a consentire la firma di documenti senza necessità di presenziare fisicamente.

1.1.1 Competizioni Crittografiche e Standard del NIST

Il National Institute of Standards and Technology (NIST) svolge un ruolo centrale nell'evoluzione e la definizione degli standard crittografici alla base della cybersecurity. Attraverso delle competizioni crittografiche, il NIST seleziona e promuove algoritmi robusti per affrontare sfide emergenti e garantire la sicurezza delle comunicazioni digitali.

Uno dei contributi più significativi del NIST è rappresentato dalla competizione per l'Advanced Encryption Standard (AES) nel 1997. Questa competizione ha portato all'adozione di un algoritmo simmetrico di crittografia che è diventato uno standard ampiamente utilizzato per garantire la confidenzialità dei dati.

Il NIST ha anche guidato la competizione per lo sviluppo del Secure Hash Algorithm 3 (SHA-3). Questo processo ha portato alla selezione dell'algoritmo Keccak, riconosciuto come standard crittografico per gli algoritmi di hash.

Inoltre, la crescente minaccia dei computer quantistici alle attuali tecniche di crittografia ha spinto il NIST ad intraprendere un'iniziativa ambiziosa per identificare algoritmi Post-Quantum. Attualmente in corso, questa competizione mira a stabilire nuovi standard crittografici resistenti ai futuri sviluppi nella computazione quantistica.

Rilevante per la ricerca, l'attenzione del NIST nei confronti delle competizioni crittografiche riflette l'importanza di adottare algoritmi sicuri e innovativi per garantire la validità e la sicurezza delle firme digitali.

L'algoritmo Codes and Restricted Objects Signature Scheme (CROSS), descritto ed approfondito in questo lavoro di tesi, è attualmente in gara nella competizione NIST relativa alle firma digitali Post-Quantum.

1.1.2 Notazione Matematica

Si utilizza $[a; b]$ per denotare l'insieme di tutti i reali $x \in \mathbb{R}$ tali che $a \leq x \leq b$. Per un insieme finito $A \subset \{1, \ldots, n\}$, l'espressione $a \xleftarrow{\$} A$ sta a significare che a è scelto in maniera uniformemente casuale da A. In aggiunta, si rappresenta con $|A|$ la cardinalità di A, con $A^C = \{1, \ldots, n\} \backslash A$ il suo complementare e con $A_0 = A \cup \{0\}$ l'insieme completo del valore zero. Considerando m un intero positivo, si indicherà con $\mathbb{Z}_m = \mathbb{Z}/m\mathbb{Z}$ l'anello di interi modulo m. Sia p un numero primo: si rappresenterà con $\mathbb{F}_p$ il campo finito di ordine p e con F_p^* il suo gruppo moltiplicativo. Si denota con $ord(g)$ l'ordine moltiplicativo di un elemento $g \in \mathbb{F}_p^*$. L'utilizzo delle maiuscole servirà per definire le matrici, mentre le minuscole per i vettori. Preso in esame l'insieme J, si considererà $\mathbf{A}_J$ come la matrice formata dalle

colonne di $\mathbf{A}$, indicizzate da J; una notazione similare è utilizzata anche per i vettori. La matrice identità di dimensione m sarà $\mathbf{I}_m$. Si userà $\mathbf{0}$ per definire sia la matrice nulla che il vettore, a seconda del contesto e senza specificarne la dimensione. Infine, si definisce h_p come la funzione di entropia p-aria.

1.1.3 Notazione Crittografica

L'argomento trattato in questa tesi richiede l'utilizzo di notazioni crittografiche convenzionali, come λ come parametro di sicurezza, a valori $\lambda \in \{128,\ 192,\ 256\}$. Di seguito, sono mostrate tutte le strutture crittografiche utilizzate:

- $\mathsf{Hash}{:}\{0,1\}^* \rightarrow \{0,1\}^{2\lambda}$, ossia una funzione hash crittografica sicura, che riceve un input di dimensione arbitraria e produce un digest di lunghezza fissa, pari a 2λ;
- MerkleTree, costruito a partire da t elementi $(a^{(1)},\ \ldots,a^{(t)})$ costituenti le foglie dell'albero, indicato come $\mathsf{T}{=}\mathsf{MerkleTree}(a^{(1)},\ \ldots,a^{(t)})$. La radice dell'albero è estratta attraverso il metodo $\mathsf{T.Root()}$. La funzione per comporre la Merkle Proof, partendo dalle foglie indicate dall'insieme J è $\mathsf{T.MerkleProof}(J)$. La ricostruzione della radice, partendo dalle foglie $\{a^{(i)}\}_{i\in J}$ e dalla `MerkleProof`, è realizzata attraverso $\mathsf{VerifyMerkleRoot}(\{c_1^{(i)}\}_{i\notin J}, \texttt{MerkleProof})$. La funzione hash impiegata per la costruzione dell'albero produce digest di lunghezza 2λ dato che, all'intero della struttura, ogni foglia è una stringa binaria di lunghezza appunto pari a 2λ;
- SeedTree, realizzato a partire dalla radice `Root`. La funzione che compone le t foglie $(\texttt{Seed}^{(1)},\ \ldots,\texttt{Seed}^{(t)})$ è $\mathsf{SeedTree}(\texttt{Root})$. Tutti i seed generati hanno lunghezza λ. Per costruire la struttura dati `SeedPath`, la quale serve per rigenerare i seed indicizzati dall'insieme J con $\mathsf{GetSeeds}(\texttt{SeedPath}, J)$, si usa la funzione $\mathsf{SeedPath}(\texttt{Root}, J)$. Inoltre, è possibile includere una stringa di `Salt`, di lunghezza 2λ, per favorire una maggiore complessità e sicurezza nella struttura dati.

Infine, la notazione $a \xleftarrow{\text{Seed}} A$ indica che a è campionato attraverso un generatore deterministico casuale crittograficamente sicuro, il quale produce in output elementi scegliendoli uniformemente da A, ed è inizializzato in input da `Seed`.

1.2 Protocolli di Identificazione ZK e Schemi di Firma

Per completezza viene trattata, in maniera sintetica, la teoria che descrive il passaggio dagli schemi interattivi di identificazione a quelli non interattivi di firma digitale.

Uno schema di identificazione Zero Knowledge, anche chiamato Proof of Knowledge, è un protocollo interattivo nel quale un *prover* P ha l'obiettivo di dimostrare ad un *verifier* V la conoscenza di un segreto che verifica delle assunzioni precedenti, senza rivelarlo o dare informazioni su di esso. In questo caso, si esamineranno solamente i protocolli che prevedono l'invio di cinque messaggi, con il prover che invia sempre il primo e l'ultimo.

Questa tipologia di protocolli (descritta in Fig.1.2) è detta a 5-step ed ogni messaggio scambiato ha un nome specifico. Il primo messaggio del prover è chiamato *commitment*, ossia impegno verso una certa dichiarazione. I successivi due messaggi del prover sono chiamati *responses*, in quanto sono risposte alle interazioni del verifier. Esso, invece, invia due messaggi chiamati *challenges*; quest'ultimi sono determinati in maniera uniformemente casuale

attraverso due insiemi di valori, nominati rispettivamente C_1 e C_2. Dunque, una singola esecuzione del protocollo d'identificazione, mantenendo invariato l'ordine delle interazioni, può essere riassunta come:

$$T = (\texttt{Com}, \texttt{Ch}_1, \texttt{Rsp}_1, \texttt{Ch}_2, \texttt{Rsp}_2).$$

Si indicherà con T un *transcript* del protocollo, ossia una singola esecuzione corretta e completa della procedura. Al termine di ogni transcript, il verifier produce in output un valore $\texttt{Out} \in \{0, 1\}$ che indica l'autenticazione del prover (valore 1), o meno (valore 0).

Di seguito, invece, sono descritte le proprietà che un protocollo ZKID deve garantire:

- *Completeness*: un prover onesto deve essere sempre accettato. Questo significa che un'esecuzione avviata da un prover che conosce il segreto termina sempre con il risultato del verifier pari a 1.

- *Zero Knowledge*: le interazioni tra prover e verifier non devono rivelare informazioni sul segreto. Questo implica che conoscere i valori delle challenge a priori permette ad un prover malevolo di produrre transcript validi ed indistinguibili da quelli di un prover onesto.

- *Soundness*: quando il verifier è onesto, dunque le challenge sono campionate attraverso distribuzioni uniformi su C_1 e C_2, un prover malevolo (che non conosce il segreto) può convincere il verifier con una probabilità $\varepsilon < 1$. La quantità ε è chiamata *soundness error* e corrisponde alla probabilità che un prover malevolo riesca ad indovinare correttamente quale sottoinsieme di challenge sarà scelto dal verifier (in aggiunta ad una quantità trascurabile, relativa alla probabilità che il prover riesca a risolvere dei problemi difficili).

Considerando t esecuzioni parallele di un protocollo a 5-step, caratterizzato da un soundness error ε, si ottiene un nuovo schema con soundness error ε^t. Al fine di distinguere i dati dei diversi round, si usa la notazione $^{(i)}$ per indicare l'elemento dell'i-esimo round.

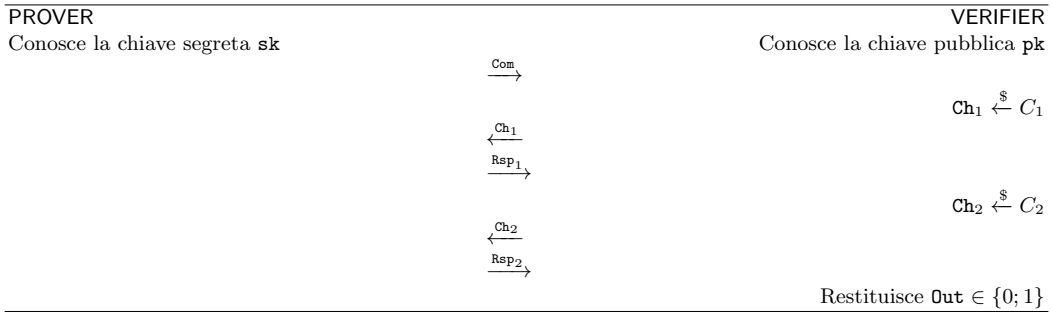

Figura 1.2: Architettura di un protocollo di identificazione Zero Knowledge 5-step

1.2.1 Trasformazione di Fiat Shamir

La trasformazione di Fiat-Shamir è una tecnica che consente di ottenere uno schema di firma a partire da un protocollo d'identificazione ZK interattivo. Tale operazione elimina l'interattività, facendo si che il prover simuli le operazioni del verifier in autonomia (challenge, output).

Il messaggio da firmare `Msg` passa come input di una funzione hash, congiuntamente ad altri dati, al fine di legarlo al transcript. Le operazioni relative alla trasformata di Fiat-Shamir, su t esecuzioni della procedura, sono:

1. Generazione dei commitment
$$\mathtt{Com} = (\mathtt{Com}^{(1)}, \mathtt{Com}^{(2)}, \cdots, \mathtt{Com}^{(t)});$$
2. Generazione della prima challenge
$$\mathtt{Ch}_1 = (\mathtt{Ch}_1^{(1)}, \mathtt{Ch}_1^{(2)}, \cdots, \mathtt{Ch}_1^{(t)}) = \mathtt{Hash}(\mathtt{Msg}, \mathtt{Com});$$
3. Composizione della prima risposta
$$\mathtt{Rsp}_1 = (\mathtt{Rsp}_1^{(1)}, \mathtt{Rsp}_1^{(2)}, \cdots, \mathtt{Rsp}_1^{(t)})$$
4. Generazione della seconda challenge
$$\mathtt{Ch}_2 = (\mathtt{Ch}_2^{(1)}, \mathtt{Ch}_2^{(2)} \cdots, \mathtt{Ch}_2^{(t)}) = \mathtt{Hash}(\mathtt{Msg}, \mathtt{Com}, \mathtt{Ch}_1, \mathtt{Rsp}_1);$$
5. Composizione della seconda risposta
$$\mathtt{Rsp}_2 = (\mathtt{Rsp}_2^{(1)}, \mathtt{Rsp}_2^{(2)}, \cdots, \mathtt{Rsp}_2^{(t)}).$$

Un'esecuzione parallela di t istanze del protocollo ci consegna, come risultato, una firma digitale. In altre parole, l'algoritmo di firma consiste in un prover che non interagisce più con un verifier, bensì lo simula generando le challenge localmente. L'algoritmo di verifica, invece, simula i controlli effettuati dal verifier sui dati ricevuti dal prover. A seconda del transcript, l'output potrà essere 1 (se la firma viene accettata) oppure 0 altrimenti.

Intuitivamente la firma ottenuta è sicura, dato che ogni challenge è generata attraverso una funzione one-way pseudo casuale, la quale riceve in input i precedenti messaggi scambiati. Inoltre, non è possibile cambiare il commitment $\texttt{Com}$ dopo la generazione della prima challenge $\texttt{Ch}_1$, a meno che l'attaccante non rilevi delle collisioni nella funzione hash. Tipicamente, una leggera modifica a $\texttt{Com}$ porta ad un cambiamento imprevedibile della challenge $\texttt{Ch}_1$.

Per ridurre la dimensione della firma, le challenge di norma sono omesse, in quanto possono essere rigenerate durante la verifica. In conclusione, la firma sarà caratterizzata dalla seguente forma:

$$\texttt{Sign} = (\texttt{Salt}, \texttt{Com}, \texttt{Rsp}_1, \texttt{Rsp}_2).$$

1.2.2 Commitment, Seed e Salt

I commitment rappresentano un impegno che il prover consegna, al fine di garantire la propria identificazione e l'integrità dei messaggi che invia. Essi sono tipicamente implementati attraverso delle funzioni hash: se il valore di riferimento è x, il prover comporrà ed invierà $\texttt{Com} = \texttt{Hash}(x)$.

Le funzioni in questione, però, dovranno offrire le seguenti caratteristiche per essere considerate valide:

- *Hiding*, se dato l'output $\texttt{Com}$ non si possono ricavare informazioni sull'input x;
- *Binding*, risulta impossibile trovare due messaggi distinti $x \neq x'$ che corrispondono allo stesso output $\texttt{Com}$. In altre parole, il prover non può modificare x dopo che l'ha scelto come riferimento.

Inoltre, il valore di riferimento per il prover è tipicamente un seme di lunghezza λ. Se non si adottano determinate precauzioni, è possibile trovare collisioni nei commitment in tempo $O(2^{\frac{\lambda}{2}})$. In questo caso un attaccante può trovare valori coerenti, senza che il prover li propaghi. Per evitare questo basta campionare, per ogni nuova firma generata, un nuovo $\texttt{Salt}$ di dimensione 2λ, il quale sarà utilizzato come input aggiuntivo della funzione hash con il seguente criterio:

$$\texttt{Com} = \texttt{Hash}(x, \texttt{Salt}).$$

1.2.3 Assunzioni di Sicurezza

L'impiego della trasformata di Fiat-Shamir ad un protocollo di identificazione ZK, con soundness error pari a ε, porta ad ottenere uno schema di firma che ammette attacchi di falsificazione in tempo $O(\varepsilon^{-1})$. Di norma un prover malevolo può simulare ripetutamente il protocollo, cercando di indovinare i valori delle challenge e preparando commitment con risposte ad esse coerenti. I valori delle challenge saranno legati ai tentativi del prover malevolo con probabilità pari a ε; ciò vuol dire che in media i tentativi da effettuare per compromettere l'esecuzione devono essere ε^{-1}.

Se si prende in considerazione uno schema di firma con soundness error ε, caratterizzato da t esecuzioni parallele, il costo risultante per un attacco di falsificazione è pari a $O(\varepsilon^{-t})$. Se si segue la nota *euristica-*ε^{-t}, è sufficiente scegliere t tale per cui $\varepsilon^{-t} > 2^{\lambda}$. L'unica

problematica è legata al fatto che, nel caso di schemi 5-step, alcune istanze del protocollo possono fallire; dunque, non si riesce a garantire sempre un livello di complessità pari a 2^λ.

Nel documento [1] vengono approfondite le formule per caratterizzare i costi degli attacchi di falsificazione. Sulla base delle analisi condotte, tale costo dipende solamente da t e dalla dimensione degli spazi delle challenge, ossia $|C_1|$ e $|C_2|$. Dato che C_1 e C_2 sono essenzialmente fissati, in base al protocollo ZK sul quel si appoggia lo schema, il valore di t va scelto di modo che il costo sia sopra a 2^λ. Nella pratica, se ci si basa sugli attacchi definiti nella documentazione, il valore effettivo di t è più grande di quello dato dall'*euristica-*ε^{-t}.

Ulteriori attacchi si basano, invece, sull'idea di acquisire informazioni e sfruttando la non interattività dello schema per produrre transcript validi, con una probabilità di successo maggiore di ε^t.

Di seguito, è possibile dimostrare come lo schema di firma ottenuto dal paradigma Fiat-Shamir raggiunga sicurezza EUF-CMA (Existential Unforgeability Under Chosen Message Attack), attraverso un protocollo di identificazione Zero Knowledge con specifiche proprietà.

1.2.4 Schemi di identificazione *q*2

Un protocollo ZK $q2$ è classificato come tale, dal momento in cui garantisce tre proprietà:

1. $|C_1| = q$;
2. $|C_2| = 2$;
3. La probabilità che il `Com` assuma un determinato valore rilevabile è una quantità trascurabile considerata nel parametro di sicurezza λ.

Si può dimostrare come l'esecuzione di t istanze in parallelo di un protocollo ZK $q2$, trasformato con la tecnica di Fiat-Shamir, conduca ad uno schema di firma con sicurezza EUF-CMA [2], necessaria a validare lo schema di firma digitale rispetto al Quantum Random Oracle Model. La dimostrazione di quanto affermato si basa sull'esistenza di un risolutore q2, ossia un algoritmo probabilistico polinomiale ξ che calcola, con probabilità non trascurabile di successo pari a $1 - \varepsilon$, il segreto `sk`, dati quattro transcript validi del tipo:

$$T = (\mathtt{Com}, \mathtt{Ch}_1, \mathtt{Rsp}_1, \mathtt{Ch}_2, \mathtt{Rsp}_2),$$

$$T' = (\mathtt{Com}, \mathtt{Ch}'_1, \mathtt{Rsp}'_1, \mathtt{Ch}'_2, \mathtt{Rsp}'_2),$$

$$T'' = (\mathtt{Com}, \mathtt{Ch}''_1, \mathtt{Rsp}''_1, \mathtt{Ch}''_2, \mathtt{Rsp}''_2),$$

$$T''' = (\mathtt{Com}, \mathtt{Ch}'''_1, \mathtt{Rsp}'''_1, \mathtt{Ch}'''_2, \mathtt{Rsp}'''_2),$$

con

$$\mathtt{Ch}_1 = \mathtt{Ch}''_1 \neq \mathtt{Ch}'_1 = \mathtt{Ch}'''_1,$$

$$\mathtt{Ch}_2 = \mathtt{Ch}''_2 \neq \mathtt{Ch}'_2 = \mathtt{Ch}'''_2.$$

Dunque, per quanto concerne gli schemi di identificazione q2, provare l'esistenza di un risolutore q2 risulta identico a mostrare che il protocollo è nella forma (2,2)-out-of-(q,2) special sound, il che implica soundness error pari a:

$$\varepsilon = 1 - \Big(1 - \frac{1}{q}\Big)\Big(1 - \frac{1}{2}\Big) = \frac{q+1}{2q}.$$

Sia, infine, noto come la scelta di un valore adatto per le t ripetizioni parallele è di fondamentale importanza e richiede di considerare a fondo il costo degli attacchi di falsificazione descritti in [**?**].

1.3 Teoria dei Codici

Un *codice lineare* C su un campo finito $\mathbb{F}_q$, con lunghezza n e dimensione $k \leq n$, è un sottospazio lineare k-dimensionale su $\mathbb{F}_q^n$. Una prima rappresentazione compatta del codice si basa su una *generator matrix* $G \in \mathbb{F}_q^{k\times n}$, tale che $C = \{\mathbf{u}G \mid \mathbf{u} \in \mathbb{F}_q^k\}$. Un codice di lunghezza n e dimensione k ha un *tasso* $R = \frac{k}{n}$ e *ridondanza* $r = n - k$.

Allo stesso modo, è possibile rappresentare il codice attraverso una *parity-check-matrix* $\mathbf{H} \in \mathbb{F}_q^{r\times n}$, tale che $C = \{\mathbf{c} \in \mathbb{F}_q^n \mid \mathbf{c}\mathbf{H}^\top = \mathbf{0}\}$. La *sindrome* di qualche $\mathbf{x} \in \mathbb{F}_q^n$ è un vettore di lunghezza r, ottenuto da $\mathbf{s} = \mathbf{x}\mathbf{H}^\top$.

Un insieme $J \subseteq \{1, \ldots, n\}$ di dimensione k, viene chiamato *information set* per C se $|C_J| = q^k$, dove $C_J = \{\mathbf{c}_J \mid c \in C\}$. Intuitivamente, G_J e H_{J^C} sono matrici invertibili. Inoltre, la *generator matrix* e la *parity-check matrix* sono rispettivamente in forma sistematica (rispetto all'*information set* J) se $G_J = I_k$ e $H_{J^C} = I_{n-k}$. Allo spazio vettoriale $\mathbb{F}_q^n$ si integrano, poi, le metriche di *Hamming*: dato $\mathbf{x} \in \mathbb{F}_q^n$, il suo *peso di Hamming* $wt(\mathbf{x})$ è caratterizzato da tutte le sue componenti diverse da zero. La *distanza minima* di un codice lineare è data da $d = min\{wt(\mathbf{c}) \mid \mathbf{c} \in C, \mathbf{c} \neq \mathbf{0}\}$. Facendo successivamente riferimento al limite di Gilbert-Varshamov, si ottiene che $R \geq 1 - h_q(\frac{d}{n})$; di conseguenza, un codice di sufficiente lunghezza (pari a n) che rispetta tale condizione, permette di ricavare $\delta = \frac{d}{n} = h_q^{-1(1-R)}$.

Infine, è necessario evidenziare come la crittografia basata su codici linari faccia spesso riferimento a problemi difficili [3, 4].

1.3.1 Syndrome Decoding Problem (SDP)

Dati $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$, $t \in \mathbb{N}$ ed $\mathbf{s} \in \mathbb{F}_q^{n-k}$, verificare se esiste un $\mathbf{e} \in \mathbb{F}_q^n$ tale che $wt(\mathbf{e}) \leq t$ e $\mathbf{e}\mathbf{H}^\top = \mathbf{s}$.

Si assume che l'istanza di SDP sia scelta in maniera uniformemente casuale, così che il codice con parity-check matrix $\mathbf{H}$ rispetti il limite di Gilbert-Varshamov. Se il peso desiderato t è minore della minima distanza δn del limite di GV, ci si aspetta di ottenere in media un'unica soluzione (se esistente), dato che (in media) il numero di soluzioni è dato da $p^{n(h_q(\delta)-1+R)} \leq 1$.

1.3.2 Restricted Syndrome Decoding Problem (R-SDP)

Dato $g \in \mathbb{F}_q^*$ di ordine z, $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$, $\mathbf{s} \in \mathbb{F}_q^{n-k}$, ed $\mathbb{E} = \{g^i \mid i \in \{1, ..., z\}\} \subset \mathbb{F}_q^*$, verificare che esista $\mathbf{e} \in \mathbb{E}^n$ tale che $\mathbf{e}\mathbf{H}^\top = \mathbf{s}$.

Il problema R-SDP è strettamente legato ad altri problemi difficili ben noti, tra cui lo stesso SDP; inoltre, se $z = 1$, R-SDP risulta simile al problema di Subset Sum (SSP) su campi finiti. Infine, si osserva come R-SDP risulti NP-completo per qualsivoglia scelta di $\mathbb{E}$ [5].

Caratteristiche di R-SDP: Unicità della soluzione

Le istanze di R-SDP sono scelte sempre in maniera uniformemente casuale. Allo stesso modo, sia la parity-check matrix $\mathbf{H}$ sia il vettore di errore ristretto $\mathbf{e}$ vengono generati attraverso un campionamento uniforme casuale, rispettivamente da $\mathbb{F}_q^{(n-k)\times n}$ ed $\mathbb{E}^n$. Inoltre, il numero di soluzioni atteso in media risulta essere:

$$\frac{z^n}{q^{n-k}} = 2^{n(\log_2(z)-(1-R)\log_2(q))}.$$

Dove z e R sono tali che $\log_2(z) \leq (1-R)\log_2(q)$, dunque la soluzione attesa è, al massimo, unica.

Caratteristiche di R-SDP: Vettori ristretti

Sia noto come $(\mathbb{E}^n, \star)$ sia un gruppo commutativo, transitivo ed isomorfo a $(\mathbb{F}_z^n, +)$. L'isomorfismo è dato da:

$$\ell : \mathbb{E}^n \to \mathbb{F}_z^n,$$
$$\mathbf{x} = (g^{\ell_1}, ..., g^{\ell_n}) \mapsto \ell(\mathbf{x}) = (\ell_1, ..., \ell_n).$$

Questa rappresentazione di vettori in $\mathbb{E}^n$ come vettori in $\mathbb{F}_z^n$ è fondamentale, al fine di ridurre la dimensione degli oggetti. Per quanto riguarda, invece, la direzione opposta dell'isomorfismo, la notazione usata è la seguente:

$$\mathbf{a} = g^{\ell(\mathbf{a})} = (g^{\ell(\mathbf{a})_1}, ..., g^{\ell(\mathbf{a})_n}),$$

per qualche $\ell(\mathbf{a}) = (\ell(\mathbf{a})_1, ..., \ell(\mathbf{a})_n) \in \mathbb{F}_z^n$.

Dato che ogni vettore ristretto $\mathbf{a} \in \mathbb{E}^n$ può essere rappresentato, in modo compatto, come un vettore $\ell(\mathbf{a}) \in \mathbb{F}_z^n$, allora per rappresentare un vettore ristretto sono richiesti solamente $n\log_2(z)$ bit.

Caratteristiche di R-SDP: Trasformazioni ristrette

Una mappa linare $\psi : \mathbb{E}^n \to \mathbb{E}^n$, che agisce transitivamente su $\mathbb{E}^n$, è data da una moltiplicazione componente per componente del tipo $\psi(\mathbf{b}) = \mathbf{a} \star \mathbf{b}$, per qualche $\mathbf{a} \in \mathbb{E}^n$. Infatti, per ogni $\mathbf{a} = g^{\ell(\mathbf{a})}$ e ogni $\mathbf{b} = g^{\ell(\mathbf{b})}$, è sempre possibile trovare $\mathbf{c} = g^{\ell(\mathbf{a})-\ell(\mathbf{b})}$, tale che $\mathbf{a} = \mathbf{c} \star \mathbf{b}$.

Si consideri la mappa ψ come la moltiplicazione componente per componente con $\mathbf{a} \in \mathbb{E}^n$. Successivamente, è possibile rappresentare in maniera compatta ψ, attraverso il vettore $\ell(\mathbf{a}) \in \mathbb{F}_z^n$. In aggiunta, il calcolo di $\psi(\mathbf{b}) = \mathbf{a} \star \mathbf{b}$ è dato dall'addizione in $\mathbb{F}_z^n$, ossia $\ell(\mathbf{a}) + \ell(\mathbf{b})$. Così, per rappresentare le trasformazioni ristrette, si impiegano sempre $n\log_2(z)$ bit.

1.3.3 Restricted Decoding Problem in a Subgroup G (R-SDP$\langle G \rangle$)

Per ultimo, è possibile ragionare su R-SDP considerando il sottogruppo $(G, \star) \leq (\mathbb{E}^n, \star)$ come:

$$G = \langle \mathbf{a}_1, ..., \mathbf{a}_m \rangle = \Big\{ \prod_{i=1}^{m} \mathbf{a}_i^{u_i} \mid u_i \in \{1, ..., z\} \Big\},$$

per qualche $m < n$. In questo modo è possibile aggiornare R-SDP alla versione R-SDP(G). Sia $G = \langle \mathbf{a}_1, ..., \mathbf{a}_m \rangle$, con $\mathbf{a}_i \in \mathbb{E}^n$, $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$ ed $\mathbf{s} \in \mathbb{F}_q^{n-k}$. Esiste un vettore $\mathbf{e} \in G$ tale che $\mathbf{e}\mathbf{H}^\top = \mathbf{s}$?

RSDP(G) risulta essere un problema NP-hard, dato che R-SDP è di per se NP-hard. Dunque, se esistesse un risolutore del problema per ogni G (in tempo polinomiale), allora potrebbe risolverlo anche per $G = \mathbb{E}^n$.

Caratteristiche di R-SDP(G): Vettori ristretti nel sottogruppo

Per costruire elementi $\mathbf{e} \in G$ è possibile salvare tutti gli esponenti dei generatori $\mathbf{a}_i$ all'interno di una matrice. In questo modo, si definisce la matrice $\mathbf{M}_G \in \mathbb{F}_z^{m\times n}$ come:

$$\mathbf{M}_G = \begin{pmatrix} \ell(\mathbf{a}_1)_1 & \dots & \ell(\mathbf{a}_1)_n \\ . & . & . \\ . & . & . \\ . & . & . \\ \ell(\mathbf{a}_m)_1 & \dots & \ell(\mathbf{a}_m)_n \end{pmatrix} = \begin{pmatrix} \ell(\mathbf{a}_1) \\ . \\ . \\ . \\ \ell(\mathbf{a}_m) \end{pmatrix}.$$

Per verificare se $|G| = z^m$, basta verificare che $rank(\mathbf{M}_G) = m$. Al fine di costruire un elemento $\mathbf{e} \in G$, possiamo poi scegliere un qualsiasi vettore $\mathbf{u} \in \mathbb{F}_z^m$ e comporre $\ell(\mathbf{e}) = \mathbf{u}\mathbf{M}_G \in \mathbb{F}_z^n$, dato che $\mathbf{e} = g^{\ell(\mathbf{e})}$. Inoltre, si introduce l'omomorfismo di gruppo ℓ_G:

$$\ell_G : G \to \mathbb{F}_z^m,$$

$$\mathbf{e} = \mathbf{a}_1^{u_1} \star \dots \star \mathbf{a}_m^{u_m} \mapsto \ell_G(\mathbf{e}) = (u_1, ..., u_m).$$

Quindi, per campionare $\mathbf{e} \in G$, basta scegliere $\ell_G(\mathbf{e}) \in \mathbb{F}_z^m$. Per rappresentare vettori ristretti nel sottogruppo G, sono richiesti solo $m \log_2(z)$ bit.

Caratteristiche di R-SDP(G): Trasformazioni ristrette in un sottogruppo

Le mappe lineari $\psi : G \to G$, che agiscono transitivamente su G, sono ancora date da moltiplicazioni componente per componente con un altro elemento in G, cioè per $\mathbf{e} \in G$ si ha $\psi(\mathbf{e}) = \mathbf{e}' \star \mathbf{e}$. Dunque, è possibile rappresentare nuovamente la mappa con un vettore in $\mathbb{F}_z^m$. Per rappresentare le trasformazioni ristrette in un sottogruppo, sono richiesti ancora $m \log_2(z)$ bit.

2

CROSS: Codes and Restricted Objects Signature Scheme

In questo capitolo viene intrapreso lo studio di CROSS, derivando lo schema di firma a partire da un protocollo di identificazione Zero Knowledge. Si porrà l'accento sulle proprietà e le caratteristiche, congiuntamente alla trasformazione di Fiat-Shamir che permette di ottenere la procedura ultimata.

2.1 CROSS-ID

Lo schema di firma CROSS è basato sul protocollo di identificazione Zero Knowledge CROSS-ID, il quale si appoggia alla logica di identificazione CVE trattata in [6]. É possibile dimostrare come questo schema garantisca le proprietà fondamentali alla sicurezza descritte nel capitolo precedente. Allo stesso modo, l'impiego della trasformazione di Fiat-Shamir garantisce, per t esecuzioni parallele di CROSS-ID ($q2$), l'ottenimento di uno schema di firma con Existential Unforgeability under Chosen Message Attack (EUF-CMA).

Successivamente, il focus si sposta sulle ottimizzazioni derivanti dal calcolo parallelo, con l'obiettivo di ridurre i costi di comunicazione e la dimensione della firma. Il protocollo sarà descritto nella versione basata sul problema R-SDP(G), che considera R-SDP con $G = \mathbb{E}^n$.

2.1.1 Struttura del Protocollo

Il protocollo ZK alla base di CROSS è descritto in Fig.2.1. Ciò che differenzia questo schema dal CVE classico riguarda il prover, il quale campiona per primo un vettore di errore trasformato $\mathbf{e}' \in G$ ed un vettore casuale $\mathbf{u}' \in \mathbb{F}_q^n$. Solo in seguito viene calcolata la trasformazione $\sigma \in G$ tale che $\mathbf{e} = \sigma(\mathbf{e}')$. Sia noto che $\sigma : G \mapsto G$ è una biiezione tale che σ è uniformemente casuale su G, questo garantisce che e' sia campionato in maniera casuale e uniforme da G. Inoltre, dato che $\mathbf{u}$' è uniformemente casuale su $\mathbb{F}_q^n$, il vettore di risposta $\mathbf{y} = \mathbf{u'} + \beta\mathbf{e'}$ segue la distribuzione uniforme su $\mathbb{F}_q^n$. Nel dettaglio, $\beta \in \mathbb{F}_q^*$ rappresenta il vettore della prima challenge. Un'altra differenza con lo schema CVE è la prima risposta, che in CROSS-ID risulta essere $h = \mathsf{Hash}(\mathbf{y})$. Il vettore della seconda challenge è indicato con b, a valori binari. Quando $b = 1$, il prover deve comunicare il seed che ha utilizzato per campionare sia $\mathbf{u}$' sia $\mathbf{e}$': ciò mostra come $\mathbf{y}$ sia generato come la somma di un vettore mascherato ed di un vettore ristretto, moltiplicato da β. Questa strategia garantisce un risparmio in termini di

costi di comunicazione, in quanto inviare h richiede meno bit rispetto ad $\mathbf{y}$. Quando $b = 0$, il prover rivela $\mathbf{y}$, assieme a σ (che non essendo una trasformazione casuale, non può essere compressa usando i seed).

Private Key $\mathbf{e} \in G$
Public Key $G \subseteq \mathbb{E}^n$, $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$, $\mathbf{s} = \mathbf{e}\mathbf{H}^\top \in \mathbb{F}_q^{n-k}$

PROVER		VERIFIER
Sample $\texttt{Seed} \xleftarrow{\$} \{0;1\}^\lambda$		
Sample $(\texttt{Seed}^{(\texttt{u'})}, \texttt{Seed}^{(\texttt{e'})}) \xleftarrow{\texttt{Seed}} \{0;1\}^{2\lambda}$		
Sample $\texttt{u}' \xleftarrow{\texttt{Seed}^{(\texttt{u'})}} \mathbb{F}_q^n$		
Sample $\texttt{e}' \xleftarrow{\texttt{Seed}^{(\texttt{e'})}} G$		
Sample $\sigma \in G$ such that $\sigma(\texttt{e'}) = \texttt{e}$		
Set $\texttt{u} = \sigma(\texttt{u'})$		
Compute $\tilde{\texttt{s}} = \texttt{u}\texttt{H}^\top$		
Set $c_0 = \texttt{Hash}(\tilde{\texttt{s}}, \sigma)$		
Set $c_1 = \texttt{Hash}(\texttt{u'}, \texttt{e'})$		
	$\xrightarrow{(c_0,c_1)}$	
		Sample $\beta \xleftarrow{\$} \mathbb{F}_p^*$
	$\xleftarrow{\beta}$	
Compute $\texttt{y} = \texttt{u'} + \beta\texttt{e'}$		
Set $h = \texttt{Hash(y)}$		
	$\xrightarrow{h}$	
		Sample $b \xleftarrow{\$} \{0;1\}$
	$\xleftarrow{b}$	
If $b = 0$, set $f := (\texttt{y}, \sigma)$		
If $b = 1$, set $f := \texttt{Seed}$		
	$\xrightarrow{f}$	
		If $b = 0$:
		Compute $\tilde{\texttt{y}} = \sigma(\texttt{y})$ e $\tilde{\texttt{s}} = \tilde{\texttt{y}}\texttt{H}^\top - \beta\texttt{s}$
		Accept if:
		1) $\texttt{Hash}(\texttt{y}) = h$
		2) $\texttt{Hash}(\tilde{\texttt{s}}, \sigma) = c_0$
		3) $\sigma \in G$
		If $b = 1$:
		Sample $\texttt{Seed}^{(\texttt{u'})}, \texttt{Seed}^{(\texttt{e'})} \xleftarrow{\texttt{Seed}} \{0;1\}^{2\lambda}$
		Set $\texttt{y} = \texttt{u'} + \beta\texttt{e'}$
		Accept if:
		1) $\texttt{Hash}(\texttt{y}) = h$
		2) $\texttt{Hash}(\texttt{u'}, \texttt{e'}) = c_1$

Figura 2.1: Protocollo di identificazione Zero Knowledge CROSS-ID

Di seguito, sono approfondite le operazioni del protocollo che possono garantire migliore efficienza implementativa:

- Il gruppo G è rappresentato attraverso la base $\mathbf{M}_G \in \mathbb{F}_z^{m \times n}$. Quest'ultima può essere facilmente ottenuta attraverso $\mathbf{M}_G = (\mathbf{I}_m, \mathbf{W})$, con $\mathbf{W} \in \mathbb{F}_z^{m \times (n-m)}$. Per campionare in modo uniformemente casuale in G, è sufficiente seguire questi passaggi:
  1. campionare $\mathbf{a} \xleftarrow{\$} \mathbb{F}_z^m$,
  2. ricavare l'esponente $\mathbf{x} = \mathbf{a}\mathbf{M}_G \in \mathbb{F}_z^n$,
  3. ricavare il vettore ristretto/isometria come $(g^{x_1}, \ \ldots, g^{x_n}) \in F_q^n$.

  Si noti che $\mathbf{a}$ è la rappresentazione ℓ_G dell'oggetto ristretto che è stato campionato. Usando $\mathbf{M}_G$ nella forma sistematica, si ottiene un vantaggio computazionale in quanto $\mathbf{x} = (\mathbf{a}, \mathbf{a}\mathbf{W})$ ed inoltre, il calcolo di $\mathbf{a}\mathbf{W}$ richiede solamente $O(m(n-m))$ operazioni sul campo $\mathbb{F}_z$.
- La rappresentazione ℓ_G di σ può essere calcolata facilmente come $\ell_G(\sigma) = (a_1, \ \ldots, a_m) = \ell_G(\mathbf{e}) - \ell_G(\mathbf{e}')$. Il verifier dovrà solamente verificare che $\ell_G(\sigma) \in \mathbb{F}_z^m$ e ricalcolerà locamente σ passando prima per i coefficienti $\ell_G(\sigma)\mathbf{M}_G \in \mathbb{F}_z^n$, per poi utilizzarli come esponenti per g.
- La matrice $\mathbf{W}$, similmente alla parity-check matrix $\mathbf{H}$, può essere campionata a partire da un seed univoco $\mathtt{Seed}_{pk} \in \{0;1\}^\lambda$, il quale è incluso all'interno della chiave pubblica. In questo modo, quest'ultima sarà composta dalla coppia $\{\mathbf{s}, \mathtt{Seed}_{pk}\}$ ed avrà dimensione pari a $(n-k)\log_2(q) + \lambda$. In questo caso, si utilizzerà un $\mathtt{Seed}_{pk}$ di λ bit.
- Se ci si riferisce al problema R-SDP, non c'è bisogno di usare G. Il seed $\mathtt{Seed}_{pk}$ viene utilizzato solo per campionare $\mathbf{H}$. Gli esponenti di σ possono essere ottenuti da $\ell(\mathbf{e}) - \ell(\mathbf{e}')$.

2.1.2 Proprietà di CROSS-ID

Sulla base di quanto introdotto precedentemente, c'è la necessità di dimostrare come il protocollo CROSS-ID garantisca le caratteristiche di: zero knowledge, completeness e (2,2)-out-of-($p-1$,2) special soundness. In questo modo sarà possibile affermare il raggiungimento della forma q2, con $q = p - 1$ e soundness error pari a $\frac{p}{2(p-1)}$.

Proposizione 1. (Completeness)

Il protocollo in Fig.2.1 è completo.

Dimostrazione. Bisogna dimostrare come un prover onesto venga sempre accettato. Quando $b = 0$, si avrà:

$$\tilde{\mathbf{y}} = \sigma(\mathbf{y}) = \sigma(\mathbf{u}') + \beta\sigma(\mathbf{e}') = \mathbf{u} + \beta\mathbf{e}.$$

Da ciò, si osserva che:

$$\tilde{\mathbf{y}}\mathbf{H}^\top - \beta\mathbf{s} = \mathbf{u}\mathbf{H}^\top + \beta\mathbf{e}\mathbf{H}^\top - \beta\mathbf{s} = \tilde{\mathbf{s}} + \beta\mathbf{s} - \beta\mathbf{s} = \mathbf{u}\mathbf{H}^\top.$$

Questo risultato corrisponde alla sindrome utilizzata per generare il commitment c_0, considerando $\sigma \in G$. Quando $b = 1$, il prover invia solamente i seed e, dato che i PRNG sono deterministici, il verifier otterrà sempre le stesse identiche quantità che sono state utilizzate per generare i commitment.

Proposizione 2. (Zero Knowledge)

Il protocollo in Fig.2.1 garantisce Zero-Knowledge.

Dimostrazione. Deve essere dimostrato che un simulatore S, il quale è a conoscenza delle challenge, riesca facilmente a simulare le interazioni tra il prover e il verifier. Formalmente, bisogna mostrare come S debba produrre un transcript T^* che sia indistinguibile da un transcript T, risultante dall'interazione $\langle P, V \rangle$. A partire da qui, è possibile definire due strategie per S, le quali dipenderanno unicamente dai valori di b (vettore della seconda challenge):

- *Strategia per $b = 0$.* Il simulatore S cerca, mediante calcoli di algebra lineare, un vettore $\mathbf{e}^\star \in \mathbb{F}_q^n$ con sindrome $\mathbf{s}$ tale che $\mathbf{e}^\star \mathbf{H}^\top = \mathbf{s}$. Poi, S sceglie un $\sigma^\star \in G$ ed un vettore $\mathbf{u}^\star \in \mathbb{F}_q^n$ per poi comporre $\mathbf{u}^{\star'} = \sigma^{\star -1}(\mathbf{e}^\star)$. Infine, esso calcola $\tilde{\mathbf{s}^\star} = \mathbf{u}^\star \mathbf{H}^\top$ e il commitment $c_0 = \texttt{Hash}(\tilde{\mathbf{s}^\star}, \sigma^\star)$. Poi, S otterrà il vettore di risposta $\mathbf{y}^\star = \mathbf{u}^{\star'} + \beta \mathbf{e}^{\star'}$. Risulta, dunque, facile da verificare come il trascript prodotto da S risulti la medesima distribuzione statistica di un transcript prodotto da un prover onesto (considerando i valori di $\mathbf{y}^\star$e di $\sigma^\star$). Di certo, durante un'esecuzione onesta, $\mathbf{y}$ risulta uniformemente casuale rispetto al campo $\mathbb{F}_q^*$, in quanto $\mathbf{u}'$ è uniformemente casuale rispetto a $\mathbb{F}_q^n$. Ciò garantisce che $\mathbf{u}' + \beta \mathbf{e}'$ è uniformemente casuale su $\mathbb{F}_q^n$ e lo stesso ragionamento è applicabile alla trasformazione σ. Dunque, in un'esecuzione onesta del protocollo, risulta che σ è uniformemente distribuito su G. Si osserva cioè che, per ogni $\mathbf{e}' \in G$, esiste un unico $\sigma \in G$ tale che $\sigma(\mathbf{e}') = \mathbf{e}$. Se $\mathbf{e}'$ è uniformemente casuale su G, allora σ segue la medesima distribuzione. Il commitment, il quale non viene verificato, può essere scelto come una stringa binaria di lunghezza 2λ. Sotto l'ipotesi di Random Oracle Model, si avrà la stessa distribuzione statistica di un commitment c_1 calcolato onestamente.

- *Strategia per $b = 1$.* In questo caso, il simulatore deve semplicemente eseguire il protocollo campionando i seed e calcolando c_0, analogamente a quello che farebbe un prover onesto. Per quanto concerne l'altro commitment, è sufficiente utilizzare una stringa binaria casuale come nel caso precedente.

Proposizione 3. (Soundness)

Il protocollo in Fig.2.1 garantisce soundness, con errore $\varepsilon = \frac{p}{2(p-1)}$.

Dimostrazione. Si prende in considerazione un avversario A che prova ad imitare il prover. Il suo obiettivo è di replicare correttamente le risposte, in relazione alle challenge del verifier. Verranno mostrate due strategie di attacco che raggiungono una probabilità di successo pari a $\varepsilon = \frac{p}{2(p-1)}$. Successivamente si osserverà il motivo per il quale queste strategie risultano ad oggi ottimali, ossia che la probabilità di successo è massima e corrisponde all'errore di soundness. Per poter dimostrare queste affermazioni, bisognerà osservare che il protocollo garantisce la caratteristica (2,2)-out-of-$(p-1$,2) special sound, per poi calcolare l'errore attraverso le formule definite in [7, 8].

Strategia 0: L'avversario A mira a rispondere correttamente sempre, per il caso $b = 0$, ma cerca comunque di indovinare i valori di $\beta^\star$ quando $b = 1$. Perciò, l'avversario decide in primis il valore per $\beta^\star \in \mathbb{F}_p^*$ e un seed `Seed`, il quale sarà utilizzato per campionare $\mathbf{u}' \in \mathbb{F}_p^n$

ed $\mathbf{e}' \in G$. L'avversario, inoltre, sceglie un $\sigma \in G$ casuale e compone $\mathbf{y}^\star = \mathbf{u}' + \beta^\star \mathbf{e}'$. Il commitment c_1 si ottiene mediante $c_1 = \texttt{Hash}(\mathbf{u}', \mathbf{e}')$. Successivamente, l'avversario compone $\tilde{\mathbf{s}} = \sigma(\mathbf{y}^\star)\mathbf{H}^\top$ ed imposta $c_0 = \texttt{Hash}(\tilde{\mathbf{s}} - \beta^\star \mathbf{s}, \sigma)$. Infine, l'avversario sceglie un vettore $\tilde{\mathbf{e}} \in \mathbb{F}_p^n$ tale che $\tilde{\mathbf{e}}\mathbf{H}^\top = \mathbf{s}$ e un vettore $\tilde{\mathbf{u}} \in \mathbb{F}_p^n$ tale che $\tilde{\mathbf{u}}\mathbf{H}^\top = \sigma(\mathbf{y}^\star)\mathbf{H}^\top - \beta^\star \mathbf{s}$. L'avversario invia c_0 e c_1 al verifier e riceve β. Se $\beta = \beta^\star$, l'avversario risponde con $h = \texttt{Hash}(\mathbf{y}^\star) = h$ con $\sigma \in G$ e

$$c_0 = \texttt{Hash}(\sigma(\mathbf{y}^\star\mathbf{H}^\top - \beta\mathbf{s}), \sigma) = \texttt{Hash}(\tilde{\mathbf{s}} - \beta^\star \mathbf{s}, \sigma).$$

Se $b = 1$, l'avversario risponde con un seed per comporre $\mathbf{e}$' e $\mathbf{u}$'. Inoltre, l'avversario viene accettato come $h = \texttt{Hash}(\mathbf{u}' + \beta^\star \mathbf{e}')$ e $c_1 = \texttt{Hash}(\mathbf{u}', \mathbf{e}')$. Se $\beta \neq \beta^\star$, l'avversario invia un h diverso. Formalmente, l'avversario compone $\mathbf{y} = \sigma^{-1}(\tilde{\mathbf{u}}) + \beta\sigma^{-1}(\tilde{\mathbf{e}})$ e invia $h = \texttt{Hash}(\mathbf{y})$. Se il verifier domanda $b = 0$, l'avversario invia la coppia $(\mathbf{y}, \sigma)$ e viene accettato in quanto $h = \texttt{Hash}(\mathbf{y})$, $\sigma \in G$, e

$$c_0 = \texttt{Hash}(\sigma(\mathbf{y})\mathbf{H}^\top - \beta\mathbf{s}, \sigma) = \texttt{Hash}(\tilde{\mathbf{u}}\mathbf{H}^\top, \sigma) = \texttt{Hash}(\sigma(\mathbf{y}^\star)\mathbf{H}^\top - \beta^\star \mathbf{s}, \sigma).$$

Se il verifier invia $b = 1$, l'avversario non ha modo di essere accettato. Di conseguenza, tale strategia ha una probabilità di successo pari a

$$Pr[b = 0] + Pr[(b = 1) \wedge (\beta = \beta^\star)] = \frac{1}{2} + \frac{1}{2(p-1)} = \frac{p}{2(p-1)}.$$

Strategia 1: L'avversario spera di ricevere la challenge $b = 1$ ma, nuovamente, si prepara a rispondere anche se ottiene $b = 0$, indovinando il valore di β. L'avversario inizialemente sceglie un valore $\beta^\star \in \mathbb{F}_p^*$. L'avversario seleziona un seed dal quale vengono generati $\mathbf{u}' \in \mathbb{F}_p^n$ e $\mathbf{e}' \in G$. L'avversario sceglie anche $\sigma \in G$ e calcola $\mathbf{u} = \sigma(\mathbf{u}')$, $\tilde{\mathbf{e}} = \sigma(\mathbf{e}') \in G$. L'avversario calcola $\tilde{\mathbf{s}} = \mathbf{u}\mathbf{H}^\top + \beta^\star \tilde{\mathbf{e}}\mathbf{H}^\top - \beta^\star \mathbf{s}$. L'avversario invierà i commitment $c_0 = \texttt{Hash}(\tilde{\mathbf{s}}, \sigma)$ e $c_1 = \texttt{Hash}(\mathbf{u}' + \mathbf{e}')$. Quando l'avversario riceve $\beta \in \mathbb{F}_p^*$, calcola $\mathbf{y} = \mathbf{u}' + \mathbf{e}'\beta$ ed invia l'Hash h.

Se l'avversario riceve $b = 1$, invia i seed per calcolare $\mathtt{u}$' e $\mathtt{e}$', e sarà accettato sicuramente. Questo poichè il verifier usa i seed per ricostruire $\mathtt{u}$' e $\mathtt{e}$', i queli servono per calcolare e controllare i valori di $h = \texttt{Hash}(\mathbf{u}' + \beta\mathbf{e}')$ e $c_1 = \texttt{Hash}(\mathbf{u}', \mathbf{e}')$.

Comunque, se l'avversario riceve la challenge $b = 0$, allora invierà la coppia $(\mathbf{y}, \sigma)$ e verrà accettato solamente se $\beta = \beta^\star$, dato che

$$\sigma(\mathbf{y})\mathbf{H}^\top - \beta\mathbf{s} = \mathbf{u}\mathbf{H}^\top + \beta\tilde{\mathbf{e}}\mathbf{H}^\top - \beta\mathbf{s} = \tilde{\mathbf{s}}.$$

Così, nel caso in cui $c_0 = \texttt{Hash}(\sigma(\mathbf{y})\mathbf{H}^\top) - \beta\mathbf{s}, \sigma)$ e $h = \texttt{Hash}(\mathbf{y})$, con $\sigma \in G$. Quindi, questa strategia ha una probabilità di successo

$$Pr[b = 1] + Pr[(b = 0) \wedge (\beta = \beta^\star)] = \frac{1}{2} + \frac{1}{2(p-1)} = \frac{p}{2(p-1)}.$$

(2,2)-out-of-(p-1,2) special soundness

Si considerano quattro transcript T_1, T_2, T_3, T_4, tutti associati alle stesse coppie di commitments c_0, c_1. Il commitment c_0 permette di fissare $(\tilde{\mathbf{s}}, \sigma)$, mentre il commitment c_1 fissa la coppia $(\mathbf{u}', \mathbf{e}')$. Di seguito si identificano i transcript sulla base dei valori delle challenge, che sono caratterizzati come $(\beta, 0)$, $(\beta, 1)$, $(\beta^\star, 0)$ e $(\beta^\star, 1)$. Prendendo a riferimento le risposte del prover, le strutture dei transcript sono le seguenti:

$$T_1 : (c_0, c_1, \beta, h, \mathbf{y}, \sigma);$$

$$T_2 : (c_0, c_1, \beta, h, \mathbf{Seed});$$

$$T_3 : (c_0, c_1, \beta^\star, h^\star, \mathbf{y}^\star, \sigma^\star);$$

$$T_4 : (c_0, c_1, \beta^\star, h^\star, \mathtt{Seed}^\star).$$

Successivamente si mostra, sulla base della conoscenza di quattro transcript, una soluzione per un'istanza di R-SDP(G) con $\{\mathbf{s}, \mathbf{H}\}$ che può essere facilmente calcolata in tempo polinomiale. Ci si concentra inizialmente su T_2 e T_4. Siano $\mathbf{u}'$, $\mathbf{e}'$ i vettori generati a partire da $\mathtt{Seed}$ e siano $\mathbf{u}^{\star'}$, $\mathbf{e}^{\star'}$ generati invece da $\mathtt{Seed}^\star$. Dato che c_1 viene verificato in entrambi i casi, bisogna porre attenzione verso le collisioni delle funzioni $\mathtt{Hash}(\mathbf{u}', \mathbf{e}') = \mathtt{Hash}(\mathbf{u}^{\star'}, \mathbf{e}^{\star'})$ con $\mathbf{u}' \neq \mathbf{u}^{\star'}$ ed $\mathbf{e}' \neq \mathbf{e}^{\star'}$, oppure $\mathbf{u}' = \mathbf{u}^{\star'}$ ed $\mathbf{e}' = \mathbf{e}^{\star'}$. Dato che h e $h^\star$ sono controllati e che non si rilevano collisioni nelle funzioni hash, si ottiene $h = \mathtt{Hash}(\mathbf{y})$, con $\mathbf{y} = \mathbf{u}' + \beta\mathbf{e}'$ ed $h^\star = \mathbf{Hash}(\mathbf{y}^\star)$, con $\mathbf{y}^\star = \mathbf{u}^{\star'} + \beta^\star\mathbf{e}^{\star'} = \mathbf{u}' + \beta^\star\mathbf{e}'$. Questo implica che $\mathbf{y} - \mathbf{y}^\star = \mathbf{e}'(\beta - \beta^\star)$.

Infine, ci si concentra sulla coppia di transcript T_1 e T_3. Se non vengono rilevate collisioni, considerando $\sigma = \sigma^\star$, si ottiene:

$$\sigma(\mathbf{y})\mathbf{H}^\top - \beta\mathbf{s} = \tilde{\mathbf{s}}, e\sigma(\mathbf{y}^\star)\mathbf{H}^\top - \beta^\star\mathbf{s} = \tilde{\mathbf{s}},$$

dal quale segue che

$$\sigma(\mathbf{y} - \mathbf{y}^\star)\mathbf{H}^\top = (\beta - \beta^\star)\mathbf{s}.$$

Analizzando le relazioni che derivano dalla coppia (T_2, T_4), si calcola $\mathbf{y} - \mathbf{y}^\star = (\beta - \beta^\star)\mathbf{e}'$, dove $\mathbf{e}'$ è un vettore ristretto tale che

$$(\beta - \beta^\star)\sigma(\mathbf{e}')\mathbf{H}^\top = (\beta - \beta^\star)\mathbf{s} \Rightarrow \sigma(\mathbf{e}')\mathbf{H}^\top = \mathbf{s}.$$

Dato che σ ed $\mathbf{e}'$ sono stati verificati, in quanto $\sigma, \mathbf{e}' \in G$, allora $\sigma(\mathbf{e}') \in G$. Questo risultato permette di affermare che $\sigma(\mathbf{e}')$ risolve il problema R-SDP(G) per l'istanza $\{\mathbf{s}, \mathbf{H}\}$.

2.1.3 Trasformazione di Fiat-Shamir

Dato che il protocollo in Fig.2.1 è classificato $q2$, come specificato nella documentazione [2], se vengono prese in considerazione t esecuzioni parallele dell'algoritmo applicando la trasformazione di Fiat-Shamir, ciò che si ricava è uno schema di firma che garantisce sicurezza EUF-CMA.

Teorema

CROSS, lo schema di firma risultante dall'applicazione della trasformata di Fiat-Shamir su t esecuzioni parallele del protocollo di identificazione Zero Knowledge q2 CROSS-ID, garantisce sicurezza EUF-CMA.

2.1.4 Ottimizzazioni del Protocollo

Tutti i messaggi scambiati nell' i-esimo round saranno indicati con gli apici $^{(i)}$. Per ottenere una notazione chiara e compatta, si raggruppano i messaggi scambiati nella seguente rappresentazione:

	Round 1	**Round 2**	$\cdots$	**Round t**
$Commitment =$	$c_0^{(1)}\ c_1^{(1)}$	$c_0^{(2)}\ c_1^{(2)}$	$\cdots$	$c_0^{(t)}\ c_1^{(t)}$
$First\ Challenge =$	$\beta^{(1)}$	$\beta^{(2)}$	$\cdots$	$\beta^{(t)}$
$First\ Response =$	$h^{(1)}$	$h^{(2)}$	$\cdots$	$h^{(t)}$
$Second\ Challenge =$	$b^{(1)}$	$b^{(2)}$	$\cdots$	$b^{(t)}$
$Second\ Response =$	$f^{(1)}$	$f^{(2)}$	$\cdots$	$f^{(t)}$

Per quanto concerne invece un'ottica di sicurezza, al fine di prevenire attacchi basati sulle collisioni delle funzioni hash, vengono introdotti 2λ bit di `Salt`, come suggerito nella documentazione [9]. Successivamente, ci si concentra sull'approfondimento delle configurazioni impiegate al fine di garantire migliorie allo schema, sia in termini di efficienza che di sicurezza.

Ottimizzazione: Peso Fisso nella Seconda Challenge

Considerando la seconda challenge, composta da $(b^{(1)},\cdots,b^{(t)})$, essa avrà sempre peso fissato pari a w. Tale valore rappresenta il numero di round nei quali il verifier richiede i valori associati a $b = 1$, mentre $t - w$ sono i round nei quali $b = 0$.

Nel primo caso, il prover è tenuto ad inviare solamente un seed di lunghezza λ e non deve rivelare $\mathbf{y}$ (dato che il verifier può ricalcolarlo autonomamente). Una scelta valida risulta essere basata su un valore di w vicino a quello di t. Dunque, l'obiettivo è garantire un minor costo di comunicazione possibile, per il maggior numero di round. Tale intervento, però, modifica il costo degli attacchi di falsificazione, in quanto un avversario può trarre vantaggio dalla predominanza dei round a valore $b = 1$.

Ottimizzazione: Impiego di Seed Tree

Per ogni esecuzione dell'algorimto di firma, vengono generati t seed $\texttt{Seed}^{(1)},\cdots,\texttt{Seed}^{(t)}$, i quali saranno impiegati per campionare, ad ogni round, $\mathbf{u'}^{(i)}$ e $\mathbf{e'}^{(i)}$. Tali seed sono ottenuti attraverso una struttura dati ad albero, composta a partire dalla radice `MSeed||Salt`, con $\texttt{MSeed} \xleftarrow{\$} \{0;1\}^{\lambda}$. Questo elenco di seed $\texttt{Seed}^{(1)},\cdots,\texttt{Seed}^{(t)}$ rappresenta le foglie dell'albero, in quanto ne caratterizza il livello più basso.

Sia noto che al prover è richiesto un seed in w round, mentre nei restanti $t-w$ è obbligato a rivelare tutti i dati. In conlusione, il massimo numero di nodi da rivelare è pari a $(t - w)\log_2(\frac{t}{t-w})$. Dunque, inviare tutti i seed ha un costo di comunicazione pari a

$$|\texttt{SeedPath}| = \lambda(t - w)\log_2(\frac{t}{t - w}).$$

Ottimizzazione: Posticipare la Verifica della Prima Risposta

La verifica della prima challenge può essere rimandata alla fine dell'algorimto di controllo.

Difatti, ad ogni round, il verifier entra in possesso di $\mathbf{y}^{(i)}$ (sia che l'abbia ricevuto dal prover, sia che l'abbia ricalcolato localmente a partire dal $\texttt{Seed}^{(i)}$).

Invece di rispondere con $(h^{(1)},\cdots,h^{(t)})$, il prover convenientemente può inviare

$$h = \texttt{Hash}(\mathbf{y}^{(1)},\mathbf{y}^{(2)},\cdots,\mathbf{y}^{(t)}).$$

Per generare la seconda challenge, il prover utilizza h (in quanto questo valore è incluso nella firma). Dopo aver eseguito tutti i round, il verifier può ricomporre localmente h. Se il ricalcolo del verifier coincide con il valore dato di h, allora è stata trovata una collisione nella funzione hash oppure i t vettori $\mathbf{y}^{(1)},\mathbf{y}^{(2)},\cdots,\mathbf{y}^{(t)}$ sono con certezza validi.

Ottimizzazione: Riduzione della dimensione dei Commitment

In ogni round il verifier può sempre ricomporre localmente uno dei due commitment. Dato che il vettore della seconda challenge ha peso fisso w vicino a t, il verifier ricalcolerà la maggior parte dei commitment $c_1^{(i)}$ e solamente pochi commitment $c_0^{(i)}$. Per quanto riguarda i commitment $c_1^{(i)}$, il prover può convenientemente associarsi ad un unico hash digest

$$c_1 = \texttt{Hash}(c_1^{(i)},\cdots,c_1^{(t)}).$$

Sia $J \subseteq \{1,\cdots,t\}$ il supporto di $(b^{(1)},\cdots,b^{(t)})$ (ossia l'insieme di indici i tali per cui $b^{(j)}=1$): il verifier conoscerà tutti i $c_1^{(i)}$ con $i \in J$ e non conoscerà quelli per cui $i \notin J$. Per ognuno di questi ultimi indici, il prover dovrà includere $c_1^{(i)}$ nella seconda risposta. In questo modo, il costo complessivo associato ai commitment $c_1^{(i)},\cdots,c_1^{(t)}$ sarà

$$|\texttt{Com(1)}| = \underbrace{2\lambda}_{c_1} + \underbrace{2(t-w)\lambda}_{c_1^{(i)},\ i\notin J} = 2\lambda(t-w-1).$$

Per i commitment $c_0^{(i)}$, il prover può realizzare un albero di Merkle T, usando $c_0^{(1)},\cdots,c_0^{(t)}$ come foglie, con d_0 come radice. Il verifier potrà ricomporre tutti i $c_0^{(i)}$ con $i \notin J$; inoltre, per certificare che il prover si è legato a valori corretti, il verifier richiederà una Merkle Proof per tutti i restanti $c_0^{(i)}$.

Sia noto che $t-w$ round sono caratterizzati da una seconda challenge a valore zero. Di norma, inviare tutte le prove richiederebbe $(t-w)\log_2(t)$ hash digest (essendoci $\log_2(t)$ digest per ognuna delle $t-w$ foglie per le quali è richiesta la prova). In maniera più conveniente, è possibile considerare come queste prove hanno dei path in comune: il numero di hash differenti che sono necessari non è maggiore di $(t-w)\log_2(\frac{t}{t-w})$.

Così facendo, il costo complessivo associato ai commitment $c_0^{(1)},\cdots,c_0^{(t)}$ sarà limitato superiormente da

$$|\texttt{Com(0)}| = \underbrace{2\lambda}_{c_0} + \underbrace{2\lambda\log_2(\frac{t}{t-w})}_{Proof\ di\ c_0^{(i)},\ i\notin J} = 2\lambda(1+(t-w)\log_2(\frac{t}{t-w})).$$

2.2 Schema di firma CROSS

Dopo aver attentamente valutato tutte le ottimizzazioni precedentemente suggerite, lo schema di firma effettivo si articola nei seguenti passaggi (Fig. 2.2, 2.3).

Signing:

1. campiona un $\texttt{Salt} \xleftarrow{\$} \{0;1\}^{2\lambda}$;
2. scampiona un $\texttt{MSeed} \xleftarrow{\$} \{0;1\}^{\lambda}$ e crea un albero di seed, il quale ha come t foglie gli elementi $\texttt{Seed}^{(1)},\cdots,\texttt{Seed}^{(t)}$. Il singolo $\texttt{Seed}^{(i)}$ è necessario per campionare $\mathbf{u'}^{(i)}$ e $\mathbf{e'}^{(i)}$, impiegati di fatto nel round i;
3. per i round $i = 1,\cdots,t$ calcola la trasformazione ristretta $\sigma^{(i)}$ ed i commitment $c_0^{(i)}$ e $c_1^{(i)}$, come definito in CROSS-ID. Inoltre, utilizza il $\texttt{Salt}$ e l'indice di round i all'interno delle funzioni hash per aggiungere sicurezza;
4. costruisce l'albero di Merkle T con i commitment $c_0^{(1)},\cdots,c_0^{(t)}$;
5. genera il vettore della prima challenge $(\beta^{(1)},\cdots,\beta^{(t)})$ utilizzando: il messaggio, il $\texttt{Salt}$ ed i commitment;
6. calcola $(\mathbf{y}^{(1)},\cdots,\mathbf{y}^{(t)})$ come descritto in CROSS-ID, per poi generare h mediante hash;
7. genera il vettore della seconda challenge $\mathbf{b} = (b^{(1)},\cdots,b^{(t)}) \in \{0;1\}^t$ a partire dall'hash di: messaggio, $\texttt{Salt}$, commitment, risposte ed h. Tale vettore ha peso di Hamming fisso, pari a w. Inoltre, viene definito l'insieme J come supporto di b, caratterizzante gli indici i tali per cui $b^{(i)} = 1$;
8. calcola $\texttt{SeedPath}$ come l'insieme dei nodi intermedi dell'albero di seed, i quali sono necessari a ricomporre i $\texttt{Seed}^{(i)}$, per $i \in J$;
9. configura $\texttt{MerkleProof}$ come una struttura contentente le prove per le foglie $\{c_0^{(i)}\}_{i \notin J}$, al fine di ricalcolare la radice dell'albero di Merkle;
10. la firma ottenuta sarà

$$\texttt{Signature} = \left\{\texttt{Salt}, c_0, c_1, h, \texttt{SeedPath}, \texttt{MerkleProof}(T_0), \{\mathbf{y}^{(i)}, \sigma^{(i)}, c_1^{(i)}\}_{i \notin J}\right\}.$$

Private Key $\mathbf{e} \in G$
Public Key $G \subseteq \mathbb{E}^n$, $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$, $\mathbf{s} = \mathbf{e}\mathbf{H}^\top \in \mathbb{F}_q^{n-k}$

Input Message $\mathtt{Msg}$
Output Signature $\mathtt{Signature}$

SIGNER | VERIFIER

Sample $\mathtt{MSeed} \xleftarrow{\$} \{0;1\}^\lambda$, $\mathtt{Salt} \xleftarrow{\$} \{0;1\}^{2\lambda}$
Generate $(\mathtt{Seed}^{(1)},\cdots,\mathtt{Seed}^{(t)}) = \mathtt{SeedTree(MSeed,Salt)}$

For $i = 1,\cdots,t$:
  Sample $(\mathtt{Seed}^{(u')},\mathtt{Seed}^{(e')}) \xleftarrow{\mathtt{Seed}^{(i)}} \{0;1\}^{2\lambda}$
  Sample $\mathbf{u}'^{(i)} \xleftarrow{\mathtt{Seed}^{(u)'}} \mathbb{F}_q^n$, $\mathbf{e}'^{(i)} \xleftarrow{\mathtt{Seed}^{(e')}} G$
  Compute $\sigma^{(i)} \in G$ such that $\sigma^{(i)}(\mathbf{e}'^{(i)}) = \mathbf{e}$
  Set $\mathbf{u}^{(i)} = \sigma^{(i)}(\mathbf{u}'^{(i)})$
  Compute $\tilde{\mathbf{s}}^{(i)} = \mathbf{u}^{(i)}\mathbf{H}^\top$
  Set $c_0^{(i)} = \mathtt{Hash}(\tilde{\mathbf{s}}^{(i)}, \sigma^{(i)}, \mathtt{Salt}, i)$
  Set $c_1^{(i)} = \mathtt{Hash}(\mathbf{u}'^{(i)}, \mathbf{e}'^{(i)}, \mathtt{Salt}, i)$

Set $T = \mathtt{MerkleTreee}(c_o^{(1)},\cdots,c_o^{(t)})$
Compute $c_o = T.\mathtt{Root}()$
Compute $c_1 = \mathtt{Hash}(c_1^{(1)},\cdots,c_1^{(t)})c$
Generate $(\beta^{(1)},\cdots,\beta^{(t)}) = \mathtt{GenCh}_1(c_o, c_1, \mathtt{Msg}, \mathtt{Salt})$

For $i = 1,\cdots,t$:
  Compute $\mathbf{y}^{(i)} = \mathbf{u}'^{(i)} + \beta^{(i)}\mathbf{e}'^{(i)}$
  Compute $h^{(i)} = \mathtt{Hash}(\mathbf{y}^{(i)})$

Compute $h = \mathtt{Hash}(h^{(1)},\cdots,h^{(t)})$
Generate $(b^{(1)},\cdots,b^{(t)}) = \mathtt{GenCh}_2(c_o, c_1, \beta^{(1)},\cdots,\beta^{(t)}, h, \mathtt{Msg}, \mathtt{Salt})$
Set $J = \{i | b^{(i)} = 1\}$
Set $\mathtt{SeedPath} = \mathtt{SeedPath(MSeed,\ Salt,\ J)}$

For $i \notin J$:
  Set $f^{(i)} := (\mathbf{y}^{(i)}, \sigma^{(i)}, c_1^{(i)})$

Compute $\mathtt{MerkleProofs} = T.\mathtt{Proofs}(\{1,\cdots,t\} \setminus J)$
Set $\mathtt{Signature} = \{\mathtt{Salt}, c_0, c_1, h, \mathtt{SeedPath}, \mathtt{MerkleProofs}, \{f^{(i)}\}_{i\notin J}\}$

$\xrightarrow{\mathtt{Signature}}$

Figura 2.2: Lo schema di firma CROSS: generazione della firma

Verification:

1. genera il vettore della prima challenge $(\beta^{(1)},\cdots,\beta^{(t)})$ a partire da $\mathtt{Msg}$, $\mathtt{Salt}$, c_0 e c_1;
2. genera il vettore della seconda challenge $(b^{(1)},\cdots,b^{(t)})$ a partire da $\mathtt{Msg}$, $\mathtt{Salt}$, c_0, c_1, $\beta^{(1)},\cdots,\beta^{(t)}$ e h;
3. usando $\mathtt{SeedPath}$, genera i seed $\{\mathtt{Seed}^{(i)}\}_{i\in J}$;
4. per $i \in J$, ricalcola $c_1^{(i)}$, $\mathbf{y}^{(1)}$ e $h^{(1)}$;

5. per $i \notin J$, calcola $h^{(i)} = \texttt{Hash}(\mathbf{y}^{(i)})$ e $c_1^{(i)}$;
6. utilizza la `MerkleProof`, assieme ai $\{c_0^{(i)}\}_{i \notin J}$, per ricalcolare e verificare la radice c_0;
7. verifica $c_1 = \texttt{Hash}(c_1^{(1)}, \cdots, c_1^{(t)})$;
8. verifica $h = \texttt{Hash}(h^{(1)}, \cdots, h^{(t)})$.

Inoltre, le funzioni specifiche per la realizzazione delle struttre dati complesse, le quali per ora sono implicite, saranno esplicitate nella sezione successiva.

Private Key $\mathbf{e} \in G$
Public Key $G \subseteq \mathbb{E}^n$, $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$, $\mathbf{s} = \mathbf{e}\mathbf{H}^\top \in \mathbb{F}_q^{n-k}$

Input *Message* `Msg`
Output *Signature* `Signature`

PROVER		VERIFIER
	$\xrightarrow{\texttt{Signature}}$	
		Generate $(\beta^{(1)}, \cdots, \beta^{(t)}) = \texttt{GenCh}_1(c_0, c_1, \texttt{Msg}, \texttt{Salt})$
		Generate $(b^{(1)}, \cdots, b^{(t)}) = \texttt{GenCh}_2(c_0, c_1, \beta^{(1)}, \beta^{(t)}, h, \texttt{Msg}, \texttt{Salt})$
		Set $J = \{i \mid b^{(i)} = 1\}$
		Generate $\{\texttt{Seed}_{i\in J}^{(i)} = \texttt{GetSeeds(SeedPath, Salt)}\}$
		For $i \in J$:
		Compute $(\texttt{Seed}^{(\mathbf{u}')}, \texttt{Seed}^{(\mathbf{e}')}) \xleftarrow{\texttt{Seed}^{(i)}} \{0;1\}^{2\lambda}$
		Sample $\mathbf{u}'^{(i)} \xleftarrow{\texttt{Seed}^{(\mathbf{u}')}} \mathbb{F}_q^n$, $\mathbf{e}'^{(i)} \xleftarrow{\texttt{Seed}^{(\mathbf{e})'}} G$
		Set $c_1^{(i)} = \texttt{Hash}(\mathbf{u}'^{(i)}, \mathbf{e}'^{(i)}, \texttt{Salt}, i)$
		Compute $\mathbf{y}^{(i)} = \mathbf{u}'^{(i)} + \beta^{(i)}\mathbf{e}'^{(i)}$
		Compute $h^{(i)} = \texttt{Hash}(\mathbf{y}^{(i)})$
		For $i \notin J$:
		Set $h^{(i)} = \texttt{Hash}(\mathbf{y}^{(i)})$
		Compute $\tilde{\mathbf{s}}^{(i)} = \sigma^{(i)}(\mathbf{y}^{(i)})\mathbf{H}^\top - \beta^{(i)}\mathbf{s}$
		Set $c_1^{(i)} = \texttt{Hash}(h^{(1)}, \cdots, h^{(t)})$
		Verify $h = \texttt{Hash}(h^{(1)}, \cdots, h^{(t)})$
		Verify $c_0 = \texttt{VerifyMerkleRoot}(\{c_0^{(i)}\}_{i\notin J}, \texttt{MerkleProof})$
		Verify $c_1 = \texttt{Hash}(c_1^{(1)}, \cdots, c_1^{(t)})$

Figura 2.3: Lo schema di firma CROSS: verifica della firma

2.2.1 Dimensioni notevoli

Lo schema appena descritto è caratterizzato da una chiave pubblica `pk` a dimensione

$$|\texttt{pk}| = \underbrace{(n-k)\lceil \log_2(q) \rceil}_{\mathbf{s}} + \underbrace{\lambda}_{\texttt{Seed}_{pk}} .$$

Allo stesso modo, la dimensione della firma è

$$|\texttt{Signature}| = \underbrace{8\lambda}_{h,c_0,c_1,\texttt{Salt}} + \underbrace{\lambda(t-w)\log_2\left(\frac{t}{t-w}\right)}_{\texttt{SeedPath}} + \underbrace{2\lambda\left(1+(t-w)\log_2\left(\frac{t}{t-w}\right)\right)}_{\texttt{MerkleProof}} + \underbrace{(t-w)\Big(\underbrace{2\lambda}_{c_1^{(i)}} + \underbrace{n\lceil\log_2(q)\rceil}_{\mathbf{y}^{(i)}} + \underbrace{m\lceil\log_2(z)\rceil}_{\sigma^{(i)}}\Big)}_{f^{(i)},\ i\notin J}.$$

2.2.2 Descrizione procedurale

Di seguito viene mostrata una descrizione delle primitive che contraddistinguono lo schema di firma CROSS: Keygen, Sign e Verify. Keygen corrisponde all'insieme di operazioni necessarie a generare la coppia di chiavi per lo schema; le restanti due, invece, caratterizzano l'effettivo funzionamento di CROSS, in generazione e verifica della firma. Queste primitive presentano comandi ed operazioni differenti, a seconda del problema di riferimento (in $\mathbb{E}^n$ per CROSS R-SDP ed in G sottogruppo di $\mathbb{E}^n$ per CROSS R-SDP(G)).

La rappresentazione algoritmica segue la convenzione per cui le parti di colore nero sono condivise tra le due varianti, invece quelle in azzurro sono riferite a CROSS R-SDP, mentre quelle in arancione sono uniche per CROSS R-SDP(G).

Inoltre, per garantire efficienza nella computazione si rappresenteranno, quando possibile, elementi di $\mathbb{E}^n$ come vettori di lunghezza n sul campo finito $\mathbb{F}_z$ e saranno identificabili attraverso le lettere greche minuscole come η. Tale scelta serve ad evitare ambiguità di natura lessicale. La stessa rappresentazione può essere impiegata per le trasformazioni ristrette sugli elementi di $\mathbb{E}^n$, come σ, per i quali si adotterà la medesima notazione; ciò è dato dal fatto che lavorare con σ permette di ottenere $\mathbf{e} \in \mathbb{E}^n$ ma, allo stesso tempo, corrisponde ad effettuare una moltiplicazione componente per componente su $\mathbb{F}_z$. Perciò, si userà σ per denotare l'elemento $\ell_G(\sigma)$.

Nel caso R-SDP(G), i vettori di lunghezza m nel campo dei valori $\mathbb{F}_z$ saranno rappresentati con le lettere greche minuscole, come $\ell_G(\mathbf{e}'^{(i)}) = \zeta$.

Per evitare ambiguità relative ai round, è possibile passare dagli apici ai pedici, nella forma ($\texttt{Seed}^{(i)} \mapsto \texttt{Seed[i]}$). Saranno anche descritte tutte le variabili temporanee che sono state impiegate.

Inoltre, le stringhe di bit saranno caratterizzate dal font `monoface`, mentre le strutture dati e gli array seguiranno un font sans serif. Infine, per indicare un elemento di un vettore, si utilizzeranno le parentesi quadre $\mathsf{v}[i]$. La tabella 2.1 rissume le assunzioni fatte in materia di notazione tra protocollo e pseudocodice.

Tabella 2.1: Tabella degli elementi da protocollo a pseudocodice

Protocollo	**Pseudocodice**	**Semantica**
$\ell_G(\mathbf{e})$	ζ	vettore in $\mathbb{F}_z^m$ per $\mathbf{e}$
$\ell(\mathbf{e})$	η	$\zeta\mathbf{M}_G = \ell_G(\mathbf{e})\mathbf{M}_G \in \mathbb{F}_z^n$ per $\mathbf{e}$
$\ell_G(\sigma^{(i)})$	δ_i	vettore in $\mathbb{F}_z^m$ for $\sigma^{(i)}$
$\ell(\mathbf{e}'^{(i)})$	$\tilde{\eta}_i$	vettore in $\mathbb{F}_z^n$ per $\mathbf{e}'^{(i)}$
$\ell(\sigma^{(i)})$	σ_i	vettore in $\mathbb{F}_z^n$ per $\sigma^{(i)}$
$\sigma^{(i)}$	$\mathbf{v}$	trasf. su $\mathbb{E}^n$, risp. G
$\mathbf{u}^{(i)}$	$\mathbf{u}$	trasf. $\mathbf{u}'^{(i)}$
$\tilde{\mathbf{s}}^{(i)}$	$\tilde{\mathbf{s}}$	sindrome di $\mathbf{u}^{(i)}$
$c_0^{(i)}$	$\mathtt{cmt}_0[i]$	Commitment 0, round i
$c_1^{(i)}$	$\mathtt{cmt}_1[i]$	Commitment 1, round i
$c_0 = \mathcal{T}.\mathtt{root}()$	d_0	radice del Merkle tree $\mathcal{T}$ del commitment 0
c_1	d_1	Hash del commitment 1
	d_{01}	Hash del d_0, d_1
	d_m	Hash del messaggio
	d_β	Hash di d_m,d_{01}, $\mathtt{Salt}$
$(\beta^{(1)}, \ldots, \beta^{(t)})$	$\mathtt{beta}$	Prima challenge
$\mathbf{e}'^{(i)}$	$\tilde{\mathbf{e}}$	trasf. $\mathbf{e}$
$\mathbf{y}^{(i)}$	$\mathbf{y}_i$	Risposta alla prima challenge
h	d_b	Hash di $y^{(1)}, \ldots, y^{(t)}$
$b^{(1)}, \ldots, b^{(t)}$	$\mathbf{b}$	Seconda challenge
$f^{(i)}$	$\mathtt{rsp}_0, \mathtt{rsp}_1$	Risposta in round i

Generazione delle Chiavi

La primitiva KeyGen ha il compito di generare in maniera uniformemente casuale una chiave segreta, basata su logica TRNG, ed una chiave pubblica, la quale è caratterizzata da un vettore ristretto $\mathbf{e}$ appartenente ad un sottogruppo ristretto $G \subseteq \mathbb{E}^n$ ed una parity-check matrix $\mathbf{H}$, per ottenere una sindrome $\mathbf{s} = \mathbf{e}\mathbf{H}^\top$. Nella versione CROSS-R-SDP(G), è presente una matrice aggiuntiva $\mathbf{M}_G \in \mathbb{F}_z^{m \times n}$, contenente le informazioni sul sottogruppo G, necessarie alla generazione di $\mathbf{e}$. Seguendo questa procedura si riducono i dati immagazzinati, passando da una chiave privata ed una pubblica ad un seed $\mathtt{Seed}_{sk}$ e un seed $\mathtt{Seed}_{pk}$ più la sindrome $\mathbf{s}$.

L'algoritmo di KeyGen inizia pescando un seed di λ bit attraverso un TRNG; questo dato risulterà essere $\mathtt{Seed}_{sk}$, ossia la chiave segreta. Impiegando questo seed all'interno di un CSPRNG, otteniamo un'espansione deterministica da λ a 2λ bit, dai quali origineranno gli elementi della chiave pubblica: $\mathtt{Seed}_e$ e $\mathtt{Seed}_{pk}$. $\mathtt{Seed}_{pk}$ viene impiegato per generare, attraverso un'altra espansione CSPRNG, la matrice $\mathbf{H}$ (nel caso R-SDP(G) per calcolare la matrice $\mathbf{M}_G$). Per evitare il sovraccarico computazionale sui CSPRNG, si generano solamente le componenti non sistematiche di $\mathbf{H}$ o $\mathbf{M}_G$. Successivamente, l'algoritmo calcola il vettore di errore ristretto $\mathbf{e}$ nel sottogruppo G, attraverso un'espansione CSPRNG con input $\mathtt{Seed}_e$. Per la versione CROSS-R-SDP, si segue un'espansione basata sulla relazione $\eta = \ell(\mathbf{e})$, a partire da $\mathtt{Seed}_e$. Per CROSS-R-SDP(G), basta campionare un vettore casuale $\zeta = \ell(\mathbf{e}) \in \mathbb{F}_z^m$ e comporre $\eta = \ell(\mathbf{e}) = \ell_G(\mathbf{e})\mathbf{M}_G \in \mathbb{F}_z^n$. Di seguito, il vettore d'errore ristretto

Algorithm 1: KeyGen()

Data: λ: security parameter,
$g \in \mathbb{F}_p^*$: generator of a subgroup $\mathbb{E}$ of $\mathbb{F}_p^*$ with cardinality z
$\mathbb{E}^n$: restricted subgroup
M_G: $m \times n$ matrix of $\mathbb{F}_z$ elements, employed to generate vectors $\boldsymbol{\eta} \in G \subset \mathbb{E}^n$

Input: None

Output: $\mathtt{pri} : \mathtt{Seed}_{\mathtt{sk}}$: private key seed;
$\mathtt{pub} : (\mathtt{Seed}_{\mathtt{pk}}, \mathrm{s})$ public key: $\mathtt{Seed}_{\mathtt{pk}}$ is a λ bit seed, to derive the non-systematic portion of a random parity-check matrix $\mathbf{H}$ and the $m \times n$ matrix of $\mathbb{F}_z$ elements M_G;
$\mathrm{s} \in \mathbb{F}_p^{n-k}$ is the syndrome of $\mathbf{e}$ through $\mathbf{H}$

1 $\mathtt{Seed}_{\mathtt{sk}} \xleftarrow{\$} \{0,1\}^\lambda$
2 $(\mathtt{Seede}, \mathtt{Seed}_{\mathtt{pk}}) \leftarrow \mathrm{CSPRNG}\left(\mathtt{Seed}_{\mathtt{sk}}, \{0,1\}^\lambda \times \{0,1\}^\lambda\right)$
3 $(\mathtt{SeedV}, \mathtt{SeedW}) \leftarrow \mathrm{CSPRNG}\left(\mathtt{Seed}_{\mathtt{pk}}, \{0,1\}^\lambda \times \{0,1\}^\lambda\right)$
$\mathrm{V} \leftarrow \mathrm{CSPRNG}\left(\mathtt{SeedV}, \mathbb{F}_p^{(n-k)\times k}\right)$
$\mathrm{V} \leftarrow \mathrm{CSPRNG}\left(\mathtt{Seed}_{\mathtt{pk}}, \mathbb{F}_p^{(n-k)\times k}\right)$
4 $\mathbf{H} \leftarrow [\mathbf{I}_{n-k} \mid \mathbf{V}]$
5 $\mathrm{W} \leftarrow \mathrm{CSPRNG}\left(\mathtt{SeedW}, \mathbb{F}_z^{m\times(n-m)}\right)$
$\mathrm{M}_G \leftarrow [\mathrm{I}_m \mid \mathrm{W}]$
$\boldsymbol{\zeta} \leftarrow \mathrm{CSPRNG}\left(\mathtt{Seede}, \mathbb{F}_z^m\right)$
$\boldsymbol{\eta} \leftarrow \boldsymbol{\zeta}\mathrm{M}_G$
$\boldsymbol{\eta} \leftarrow \mathrm{CSPRNG}\left(\mathtt{Seede}, \mathbb{F}_z^n\right)$
6 **for** $j \leftarrow 0$ **to** $n-1$ **do**
7 $\quad \mathbf{e}[j] \leftarrow g^{\boldsymbol{\eta}[j]}$
8 **end**
9 $\mathrm{s} \leftarrow \mathbf{e}\mathbf{H}^\top$
10 $\mathtt{pri} \leftarrow \mathtt{Seed}_{\mathtt{sk}}$
11 $\mathtt{pub} \leftarrow (\mathtt{Seed}_{\mathtt{pk}}, \mathrm{s})$
12 **return** (pri,pub);

Figura 2.4: Primitiva KeyGen di CROSS

$\mathbf{e}$ nel sottogruppo G si ottiene attraverso

$$\mathbf{e} = (g^{\eta[0]}, \cdots, g^{\eta[n-1]}) = (g^{\ell(\mathbf{e})_1}, \cdots, g^{\ell(\mathbf{e})_n}) \in G.$$

Infine, una volta ottenuto il valore di $\mathbf{e}$, si calcola la sindrome $\mathbf{s}$ attraverso $\mathbf{e}\mathbf{H}^\top$. La chiave pubblica è quindi costituita da seed $\mathtt{Seed}_{pk}$, richiesto per calcolare le parti non sistematiche $\mathbf{V}$ e $\mathbf{W}$, e dalla sindrome $\mathbf{s}$. La chiave privata consiste solamente nel seed segreto $\mathtt{Seed}_{sk}$, dal quale è possibile derivare tutti gli altri elementi.

Generazione della Firma

L'algoritmo di firma realizza le operazioni descritte in Fig.2.5, ricevendo in input la chiave privata `pri` ed il messaggio da firmare `Msg`, come stringhe di bit.

Il primo passo nella procedura è di espandere, partendo da $\mathtt{Seed}_{sk}$, sia il vettore di errore ristretto $\mathbf{e}$, sia la parity-check-matrix $\mathbf{H}$; mentre, nel caso R-SDP(G), solamente la matrice $\mathbf{M}_G$ che descrive G. Queste operazioni sono le medesime della primitiva di KeyGen. Una volta che le componenti relative alle chiavi sonno state ottenute, l'algoritmo pesca da un TRNG un `MSeed` di λ bit e un `Salt` di 2λ bit. Questi elementi ricoprono un ruolo cruciale nella procedura di `SeedTreeLeaves`, che compone una sequenza di t seed (`Seed[0]`,$\cdots$,`Seed[t-1]`) (linea 4). Al suo interno la funzione costruisce un albero binario di nodi, ognuno dei quali composto

Algorithm 2: SIGN(pri, Msg)

Data: λ: security parameter,
$g \in \mathbb{F}_p^*$: generator of a subgroup $\mathbb{E}$ of $\mathbb{F}_p^*$ with cardinality z
$\mathbb{E}^n$: restricted subgroup
M_G: $m \times n$ matrix of $\mathbb{Z}_z$ elements, employed to generate vectors $\boldsymbol{\eta} \in G \subset \mathbb{E}^n$
t: number of iterations of the ZKID protocol
$\mathcal{B}_w^t$: set of all binary strings with length w and Hamming weight t
Input: pri: private key constituted of $\mathtt{Seed}_{\mathtt{sk}} \in \{0,1\}^\lambda$
Msg: message to be signed $\mathtt{Msg} \in \{0,1\}^*$
Output: Signature Signature

Begin

// Key material expansion

$\boldsymbol{\zeta}, \mathrm{H}, \mathrm{M}_G \leftarrow$ EXPANDPRIVATESEED($\mathtt{Seed}_{\mathtt{sk}}$) | $\boldsymbol{\eta}, \mathrm{H} \leftarrow$ EXPANDPRIVATESEED($\mathtt{Seed}_{\mathtt{sk}}$)

// Computation of commitments

$\mathtt{Mseed} \xleftarrow{\$} \{0,1\}^\lambda$, $\mathtt{Salt} \xleftarrow{\$} \{0,1\}^{2\lambda}$

$(\mathtt{Seed}[0], \ldots, \mathtt{Seed}[t-1]) \leftarrow$ SEEDTREELEAVES(Mseed, Salt)

for $i \leftarrow 0$ **to** $t-1$ **do**

$(\mathtt{Seed}_{\boldsymbol{u}'}, \mathtt{Seed}_{\boldsymbol{e}'}) \leftarrow \mathrm{CSPRNG}\left(\mathtt{Seed}[i], \{0,1\}^\lambda \times \{0,1\}^\lambda\right)$

$\boldsymbol{\zeta}' \leftarrow \mathrm{CSPRNG}\left(\mathtt{Seed}_{\boldsymbol{e}'}, \mathbb{F}_z^m\right)$
$\boldsymbol{\delta}_i \leftarrow \boldsymbol{\zeta} - \boldsymbol{\zeta}'$
$\boldsymbol{\eta}_i' \leftarrow \boldsymbol{\zeta}' \mathrm{M}_G$
| $\boldsymbol{\eta}_i' \leftarrow \mathrm{CSPRNG}\left(\mathtt{Seed}_{\boldsymbol{e}'}, \mathbb{F}_z^n\right)$

$\boldsymbol{\sigma}_i \leftarrow \boldsymbol{\eta} - \boldsymbol{\eta}_i'$

for $j \leftarrow 0$ **to** $n-1$ **do**

$\boldsymbol{v}[j] \leftarrow g^{\boldsymbol{\sigma}_i[j]}$

end

$\mathbf{u}_i' \leftarrow \mathrm{CSPRNG}\left(\mathtt{Seed}_{\boldsymbol{u}'}, \mathbb{F}_p^n\right)$

$\mathbf{u} \leftarrow \mathbf{v} \star \mathbf{u}_i'$ // $\star$ is component-wise product

$\tilde{\mathbf{s}} \leftarrow \mathbf{u}\mathrm{H}^\top$

$\mathtt{cmt}_0[i] \leftarrow$ HASH($\tilde{\mathbf{s}} || \boldsymbol{\delta}_i || \mathtt{Salt} || i$) | $\mathtt{cmt}_0[i] \leftarrow$ HASH($\tilde{\mathbf{s}} || \boldsymbol{\sigma}_i || \mathtt{Salt} || i$)

$\mathtt{cmt}_1[i] \leftarrow$ HASH($\mathtt{Seed}[i] || \mathtt{Salt} || i$)

end

$\mathtt{d}_0 \leftarrow$ MERKLEROOT($\mathtt{cmt}_0[0], \ldots, \mathtt{cmt}_0[t-1]$)

$\mathtt{d}_1 \leftarrow$ HASH($\mathtt{cmt}_1[0] \;||\; \ldots \;||\; \mathtt{cmt}_1[t-1]$)

$\mathtt{d}_{01} \leftarrow$ HASH($\mathtt{d}_0 \;||\; \mathtt{d}_1$)

// First challenge vector extraction

$\mathtt{d}_m \leftarrow$ HASH(m)

$\mathtt{d}_\beta \leftarrow$ HASH($\mathtt{d}_m || \mathtt{d}_{01} || \mathtt{Salt}$)

$\mathtt{beta} \leftarrow \mathrm{CSPRNG}\left(\mathtt{d}_\beta, (\mathbb{F}_p^\star)^t\right)$

// Computation of first round of responses

for $i \leftarrow 0$ **to** $t-1$ **do**

for $j \leftarrow 0$ **to** $n-1$ **do**

$\mathbf{e}'[j] \leftarrow g^{\boldsymbol{\eta}_i'[j]}$

end

$\mathbf{y}_i \leftarrow \mathbf{u}_i' + \mathtt{beta}[i]\mathbf{e}_i'$

end

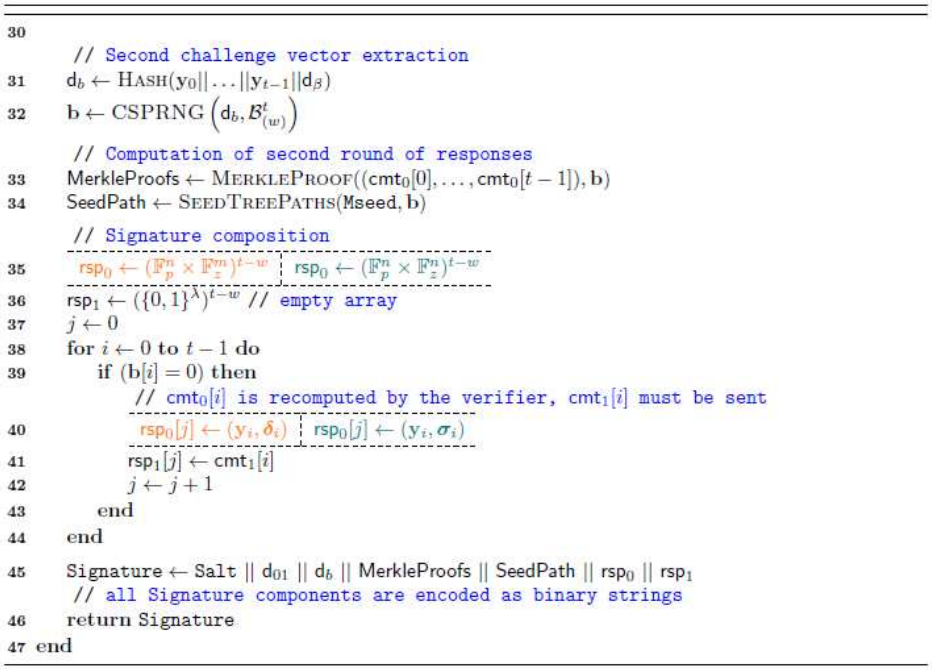

Figura 2.5: Primitiva Sign di CROSS

da λ bit pseudo-casuali, concatenati con un indice intero di posizione, calcolato tramite enumerazione a livelli dei nodi. La radice dell'albero si ottiene attraverso la concatenazione di: `MSeed`, `Salt` e l'indice di posizione 0. Mediante logica CSPRNG, ogni figlio è ottenuto dall'output della funzione, partendo da un input formato con: la stringa binaria del padre, il `Salt` e l'indice di posizione (concatenati). Le prime t foglie generate sono restituite da SeedTreeLeaves come la sequenza richiesta di (`Seed[0]`,$\cdots$,`Seed[t-1]`).

La Sign prosegue nel calcolo dei commitment per i t round del protocollo, utilizzando il `Seed[i]` generato. L'obiettivo dei cicli for (linee 5-17) è la composizione degli elementi relativi ai commitment $\mathtt{cmt}_0$ e $\mathtt{cmt}_1$, divisi nei valori $c_0^{(j)}$ e $c_1^{(j)}$ per $1 \leq j \leq t$. Per fare ciò, bisogna che l'i-esima iterazione del ciclo espanda `Seed[i]` in due componenti $\mathtt{Seed}_{u'}$ e $\mathtt{Seed}_{e'}$. $\mathtt{Seed}_{e'}$ è usato per calcolare il vettore di errore ristretto, rappresentato come un vettore di n elementi relativo agli esponenti del sottogruppo $\mathbb{E}$ e denotato come $\eta'_i \in \mathbb{F}_z^n$ (linea 7). Questo elemento è usato assieme al errore segreto η, per ottenere la trasformazione $\sigma_i \in \mathbb{F}_z^n$, tale che $\sigma_i + \eta'_i = \eta$ (corrispondente a $\ell_G(\sigma_i) + \ell_G(\mathbf{e}'_i) = \ell_G(\mathbf{e})$ nella notazione di protocollo). Queste operazioni corrispondono ad una sottrazione componente per componente in $\mathbb{F}_z^n$, in quanto tutti gli elementi sono rappresentati come vettori in $\mathbb{F}_z^n$ (linea 8). Per applicare la trasformazione ritretta σ, la Sign necessita di convertire la rappresentazione dei vettori di n elementi a valori nel campo finito $\mathbb{F}_z$, a valori in $\mathbb{F}_q$. Questo è possibile attraverso un calcolo componente per componente, pari a $g^{\sigma_i[j]}$, per tutti i $0 \leq j < n$; il che permette di appoggiarsi ai valori temporanei v. La trasformazione, adesso rappresentata come un vettore valorizzato in $\mathbb{F}_q$, è applicata ad un vettore casuale $\mathtt{u}'_i$ a valori in $\mathbb{F}_q$ (ottenuto mediante un'espansione CSPRNG con input $\mathtt{Seed}_{u'}$ linea 12). L'uso della trasformazione avviene me-

diante un prodotto componente per componente su $\mathbb{F}_q$ (linea 13). Una volta completato, la sindrome $\tilde{\mathbf{s}}$ si ottiene calcolando $\mathbf{u} = \mathbf{v} \star \mathbf{u'}$ e recuperando $\mathbf{H}$ (linea 14). I commitment relativi all'i-esimo round sono calcolati nelle linee 15 e 16. `cmt0[i]` è ottenuto come hash digest di: sindrome $\tilde{\mathbf{s}}$, trasformazione, `Salt` e numero di round i (in byte). Inoltre, sempre per quanto concerne `cmt0[i]`, si considera una rappresentazione compatta per la trasformazione σ_i nel caso R-SDP(G). Sia noto come, grazie alla linearità della moltiplicazione per $\mathbf{M}_G$, è possibile calcolare il valore δ_i sulla base della relazione $\delta_i \mathbf{M}_G = \sigma_i$, sostituendo σ_i con δ_i nel calcolo di $\mathtt{cmt}_0[i]$. Tutto ciòè realizzabile successivamente alla generazione di η'_i, nel caso R-SDP(G) (linea 7), quando i valori ζ e ζ' (tali per cui $\zeta_i \mathbf{M}_G = \eta_i$ e $\zeta'_i \mathbf{M}_G = \eta'_i$) sono disponibili. Sostituire σ_i con δ_i consente di ridurre la quantità di dati da fornire alla funzione hash, oltre che a velocizzare il calcolo di $\mathtt{cmt}_0[i]$ e $\mathtt{cmt}_1[i]$.

Una volta che tutti i commitment sono pronti, l'algoritmo procede a comporre il digest per ottenere il primo vettore di challenge `beta`. La `Sign` ottimizza la procedura classica (basata su Fiat-Shamir), attraverso due considerazioni. La prima è che il vettore della seconda challenge deve avere peso fisso elevato, quindi gli elementi di `cmt0`, da rivelare se la challenge fosse a valore 0, vengono scambiati raramente. Ciò, è di fondamentale importanza in quanto permette di realizzare una struttura di hash legati all'albero di Merkle, i quali rivelano una quantità di dati meno che linare (rispetto a t). La procedura di `Sign`, poi, calcola la radice dell'albero di Merkle $\mathtt{d}_0$, a partire dalle foglie, che risultano essere gli elementi di cmt_0. Successivamente, attraverso un hash di tutti gli elementi di cmt_1, viene generato il digest $\mathtt{d}_1$. Nell'approccio classico di Fiat-Shamir sarebbero da includere entrambi i digest nella firma, per garantire il controllo da parte del verifier. In questo caso, invece, si realizza un digest successivo $\mathtt{d}_{01}$, composto da un input di $\mathtt{d}_0$ e $\mathtt{d}_1$ (linea 20) ed includendo solamente questo all'intero della `Signature`. Il secondo aspetto notevole è relativo all'inclusione del messaggio come input della funzione hash che genera il seed, espanso successivamente in `beta`, richiedendo però che l'intero messaggio sia scambiato durante la firma. Nel caso ottimizzato, d'altra parte, si include solamente un digest del messaggio $\mathtt{d}_m$, al fine di permettere esecuzioni parallele nel caso in cui più componenti siano simultaneamente disponibili. Di seguito, si compone l'input della funzione hash che genera il digest $\mathtt{d}_\beta$, concatenando $\mathtt{d}_m$, $\mathtt{d}_{01}$ e il `Salt` (linea 22). Espandendo $\mathtt{d}_\beta$ usando un CSPRNG, si ottiene il primo vettore di challenge $\mathtt{beta} \in (F_q^*)^t$ (linea 23). Una volta generato `beta`, la procedura compone le risposte $\mathbf{y}_i$, $0 \leq i < t$, basandosi sulla rappresentazione η'_i sul campo $\mathbb{F}_q$ (linea 25-27) e calcolando $\mathbf{u}'_i + \mathbf{beta}[i]\mathbf{e}'_i$.

Per ogni round, la risposta corrispondente $\mathbf{y}_i$ è calcolata a partire dalla challenge $\mathbf{beta}[i]$, dall'errore ristretto $\mathbf{e}'_i$ e da $\mathbf{u}'_i$.

Il digest $\mathtt{d}_b$, ottenuto dall'hash di tutte le $\mathbf{y}_i$ e da $\mathtt{d}_\beta$, rappresenterà la prima risposta e sarà incluso nella firma; successivamente, la sua espansione mediante CSPRNG permetterà di ottenere il secondo vettore di challenge $\mathbf{b}$. Tale vettore deve essere campionato a partire da un insieme di stringhe binarie di lunghezza t e peso w. Si può osservare che, fin quando il digest $\mathtt{d}_b$ è pubblico, non c'è necessità di implementare un processo di campionamento a peso costante. Ciò riduce la difficoltà di ottenere un algoritmo sicuro e corretto. Dato che il numero di zeri in $\mathbf{b}$ è molto minore rispetto al numero di uni, è possibile popolare le posizioni degli zeri nella stringa, passando per un CSPRNG che ha come input i numeri di $\mathtt{d}_b$ nel range $0, n-1$. Se una posizione pescata ha già il valore zero, si scarta il numero campionato e si passa ad uno nuovo.

Una volta che il vettore di challenge $\mathbf{b}$ è disponibile, è possibile calcolare quali nodi dell'albero di Merkle dovrebbero essere inclusi nella firma, così che il verifier possa ricomporre

la radice $\mathtt{d}_0$ partendo da essi e dalle risposte alle challenge 0. Ciò è possibile grazie alla procedura MerkleProof, che riceve in input il vettore a peso fisso $\mathbf{b}$, assieme alla sequenza di commitment $\mathtt{cmt}_0$. Questa funzione include, all'interno della struttura dati `MerkleProof`, tutti i nodi dell'albero di Merkle che sono radici dei sottoalberi più elevati, tali che non contengono nodi ricalcolabili autonomamente dal verifier.

La procedura di Sign calcola poi il secondo set di risposte. L'approccio ottimizzato rappresenta esplicitamente le risposte ai bit di challenge 0, mantenendole nelle sequenze $\mathtt{rsp}_0$ ed $\mathtt{rsp}_1$. In contrasto, dato che le risposte ai bit di challenge 1 possono essere esposte in forma compatta impiegando un seed come nelle linee 5-17, allora si rappresentano tutte quante come radici di sottoalberi binari della struttura dati ad albero di seed. La procedura SeedTreePaths prende in input `MSeed`, permettendo di generare l'albero di seed assieme ad una sequenza di bit che evidenzia quali foglie devono essere divulgate al verifier. Questa funzione determina i nodi da includere nella struttura dati `SeedPaths`, opzionando tutte le radici dei sottoalberi dell'albero binario tali che i loro figli sono nodi unici da divulgare (il fratello non deve essere divulgato).

La Sign procede popolando le sequenze $\mathtt{rsp}_0$ ed $\mathtt{rsp}_1$, aventi entrambe lunghezza $t-w$. In particolare, il ciclo descritto nelle linee 38-44 lavora per ripetizioni del protocollo (indicizzate da i) e, per ogni round in cui la challenge ha valore zero, si include la coppia $(\mathbf{y}_i, \sigma_i)$ nel caso R-SDP, o quella $(\mathbf{y}_i, \delta_i)$ nel caso R-SDP(G), nella sequenza di $\mathtt{rsp}_0$; mentre memorizza in $\mathtt{rsp}_1$ i corrispondenti commitment provenienti da cmt_1.

Infine, la firma `Signature` si compone concatenando e codificando, con logica Pack Unpack [10], i seguenti elementi: un `Salt` di 2λ bit, il digest dei commitment $\mathtt{d}_{01}$, il digest delle prime risposte $\mathtt{d}_b$, la proof relativa all'albero di Merkle `MerkleProofs` (degli elementi $\mathtt{cmt}_0[i]$ associati ai bit di challenge $\mathbf{b}[i] = 0$), il percorso dell'albero dei seed `SeedPath` (per i round in cui si ha $\mathbf{b}[i] = 1$) ed i vettori di risposta $\mathtt{rsp}_0$ ed $\mathtt{rsp}_1$.

Verifica della firma

La procedura di verifica implementa le operazioni descritte in Fig.2.6. L'algoritmo prende in input la chiave pubblica `pub`, il messaggio `Msg` (sul quale andrebbe verificata la firma) e la stessa firma `Signature`. L'output della funzione è un valore booleano `True` o `False` a seconda della validità, o meno, della firma.

Il primo passaggio della procedura si basa sull'espansione del seed contenuto nella chiave pubblica, ossia $\mathtt{Seed}_{pk}$, nella parity-check matrix $\mathbf{H}$ nel caso R-SDP, e nella matrice $\mathbf{M}_G$ nel R-SDP(G) (linea 2). Una volta che questi dati sono disponibili, la Verify si concentra sul ricalcolo dei due vettori di challenge `beta` e $\mathbf{b}$. Partendo dal secondo, esso viene ricomposto usando la medesima logica CSPRNG impiegata durante la fase di generazione della firma, fornendo in input il digest $\mathtt{d}_b$. Per generare $\mathtt{d}_\beta$ mediante CSPRNG è necessario fornire in input $\mathtt{d}_m$, $\mathtt{d}_{01}$ ed il `Salt`. Successivamente, ricalcoliamo `beta` a partire da $\mathtt{d}_\beta$, a valori in $(F_q^*)^t$ e $\mathbf{b}$ utilizzando $\mathtt{d}_b$, con la medesima logica della Sign basata su peso di Hamming fisso.

L'algoritmo, poi, rigenera i seed richiesti per ricomporre i valori di $\mathtt{cmt}_1[i]$, per tutte le iterazioni i del protocollo con valore di challenge $\mathbf{b} = 1$. Queste operazioni sono realizzate da RebuildSeedTreeLeaves, che riceve come input la struttura dati `SeedPath`, il vettore di challenge $\mathbf{b}$ ed il `Salt`. La procedura calcola, a partire dalle informazioni contenute in $\mathbf{b}$, le radici dei sottoalberi più elevati che contengono solo nodi da rigenerare, per poi completare con i nodi di `SeedPath`. Si riesce poi, con logica top-down, a recuperare tutte le foglie richieste. Dopo aver generato la sequenza `Seed[0]`,$\cdots$,`Seed[t-1]`, per la quale solamente i

Algorithm 3: CROSS-VERIFY(pub, Msg, Signature)

Data: λ: security parameter,
$g \in \mathbb{F}_p^*$: generator of a subgroup $\mathbb{E}$ of $\mathbb{F}_p^*$ with cardinality z
$\mathbb{E}^n$: restricted subgroup
M_G: $m \times n$ matrix of $\mathbb{F}_z$ elements, employed to generate vectors $\boldsymbol{\eta} \in G \subset \mathbb{E}^n$
t: number of iterations of the ZKID protocol
$\mathcal{B}_w^t$: set of all binary strings with length w and Hamming weight t
Input: pub: $(\mathsf{Seed}_{\mathsf{pk}}, \mathrm{s})$ public key: $\mathsf{Seed}_{\mathsf{pk}}$ is a λ bit seed to derive the non-systematic portion of a random parity-check matrix $\mathbf{H}$ and the $m \times n$ matrix of $\mathbb{F}_z$ elements M_G
Msg: message to verify the signature on; $\mathtt{Msg} \in \{0,1\}^*$
Signature: signature obtained encoding as binary the tuple
$(\mathsf{Salt}, \mathsf{d}_{01}, \mathsf{d}_b, \mathsf{MerkleProofs}, \mathsf{SeedPath}, \mathsf{rsp}_0, \mathsf{rsp}_1)$
Output: a Boolean value, {True, False}, indicating if the signature is verified or not

1 **Begin**
// Key material expansion
2 $(\mathsf{SeedV}, \mathsf{SeedW}) \leftarrow \mathrm{CSPRNG}\left(\mathsf{Seed}_{\mathsf{pk}}, \{0,1\}^\lambda \times \{0,1\}^\lambda\right)$; $\mathbf{V} \leftarrow \mathrm{CSPRNG}\left(\mathsf{Seed}_{\mathsf{pk}}, \mathbb{F}_p^{(n-k)\times k}\right)$
$\mathbf{V} \leftarrow \mathrm{CSPRNG}\left(\mathsf{SeedV}, \mathbb{F}_p^{(n-k)\times k}\right)$
3 $\mathbf{H} \leftarrow [\mathbf{I}_{n-k} \mid \mathbf{V}]$
4 $\mathbf{W} \leftarrow \mathrm{CSPRNG}\left(\mathsf{SeedW}, \mathbb{F}_z^{m\times(n-m)}\right)$
5 $\mathrm{M}_G \leftarrow [\mathrm{I}_m \mid \mathrm{W}]$
// Challenge recomputation
6 $\mathsf{d}_m \leftarrow \mathrm{HASH}(m)$
7 $\mathsf{d}_\beta \leftarrow \mathrm{HASH}(\mathsf{d}_m || \mathsf{d}_{01} || \mathsf{Salt})$
8 $\mathsf{beta} \leftarrow \mathrm{CSPRNG}\left(\mathsf{d}_\beta, (\mathbb{F}_p^*)^t\right)$
9 $\mathbf{b} \leftarrow \mathrm{CSPRNG}\left(\mathsf{d}_b, \mathcal{B}_w^t\right)$
10 $(\mathsf{Seed}[0], \ldots, \mathsf{Seed}[t-1]) \leftarrow \mathrm{REBUILDSEEDTREELEAVES}(\mathsf{SeedPath}, \mathbf{b}, \mathsf{Salt})$
11 $j \leftarrow 0$
12 **for** $i \leftarrow 0$ **to** $t-1$ **do**
13 **if** $(\mathbf{b}[i] = 1)$ **then**
14 $\mathsf{cmt}_1[i] \leftarrow \mathrm{HASH}(\mathsf{Seed}[i] || \mathsf{Salt} || i)$
15 $(\mathsf{Seed}_{\boldsymbol{u}'}, \mathsf{Seed}_{\boldsymbol{e}'}) \leftarrow \mathrm{CSPRNG}\left(\mathsf{Seed}[i], \{0,1\}^\lambda \times \{0,1\}^\lambda\right)$
16 $\boldsymbol{\zeta}' \leftarrow \mathrm{CSPRNG}\left(\mathsf{Seed}_{\boldsymbol{e}'}, \mathbb{F}_z^m\right)$; $\boldsymbol{\eta}' \leftarrow \mathrm{CSPRNG}\left(\mathsf{Seed}_{\boldsymbol{e}'}, \mathbb{F}_z^n\right)$
$\boldsymbol{\eta}' \leftarrow \boldsymbol{\zeta}' \mathrm{M}_G$
17 **for** $j \leftarrow 0$ **to** $n-1$ **do**
18 $\mathbf{e}'[j] \leftarrow g^{\boldsymbol{\eta}'[j]}$
19 **end**
20 $\mathbf{u}' \leftarrow \mathrm{CSPRNG}\left(\mathsf{Seed}_{\boldsymbol{u}'}, \mathbb{F}_p^n\right)$
21 $\mathbf{y}_i \leftarrow \mathbf{u}' + \mathsf{beta}[i]\mathbf{e}'$
22 **end**
23 **else**
24 $(\mathbf{y}_i, \boldsymbol{\delta}_i) \leftarrow \mathsf{rsp}_0[j]$; $(\mathbf{y}_i, \boldsymbol{\sigma}_i) \leftarrow \mathsf{rsp}_0[j]$
verify $\boldsymbol{\delta}_i \in G$; verify $\boldsymbol{\sigma}_i \in G$
$\boldsymbol{\sigma}_i \leftarrow \boldsymbol{\delta} \mathrm{M}_G$
25 **for** $j \leftarrow 0$ **to** $n-1$ **do**
26 $\mathbf{v}[j] \leftarrow g^{\boldsymbol{\sigma}[j]}$
27 **end**
28 $\mathbf{y}' \leftarrow \mathbf{v} \star \mathbf{y}_i$
29 $\tilde{\mathrm{s}} \leftarrow \mathbf{y}'\mathbf{H}^\top - \mathsf{beta}[i]\mathrm{s}$
30 $\mathsf{cmt}_0[i] \leftarrow \mathrm{HASH}(\tilde{\mathrm{s}} || \boldsymbol{\delta}_i || \mathsf{Salt} || i)$; $\mathsf{cmt}_0[i] \leftarrow \mathrm{HASH}(\tilde{\mathrm{s}} || \boldsymbol{\sigma}_i || \mathsf{Salt} || i)$
31 $\mathsf{cmt}_1[i] \leftarrow \mathsf{rsp}_1[j]$
32 $j \leftarrow j + 1$
33 **end**
34 **end**

```
35
36    d'_0 ← RecomputeMerkleRoot(cmt_0, MerkleProofs, b)
37    d'_1 ← Hash(cmt_1[0] || ... || cmt_1[t − 1])
38    d'_01 ← Hash(d'_0 || d'_1)
39    d'_b ← Hash(y_0||...||y_{t−1})
40    if (d_01 = d'_01 and d_b = d'_b) then
41        return True
42    end
43    return False
44 end
```

Figura 2.6: Primitiva Verify di CROSS

nodi rivelati hanno un valore effettivo, l'algorimto compone i valori all'interno delle sequenze $\texttt{cmt}_0$ e $\texttt{cmt}_1$, cosi come i valori delle prime risposte $\mathbf{y}_i$, le quali dipendono dal bit di challenge **b[i]** (linee 12-34).

Se $\mathbf{b}[i] = 1$ (linee 13-22), la Verify ricalcola il valore di $\texttt{cmt}_1[i]$ effettuando l'hash di `Seed[i]` congiuntamente al `Salt` (linea 14). Di seguito, la procedura espande `Seed[i]`, ottenuto come foglia dell'albero di seed, nella coppia $(\texttt{Seed}_{u'}, \texttt{Seed}_{e'})$ (linea 15). Partendo da questi valori, la procedura permette di ricalcolare il vettore $\mathbf{e}'$ realizzando la stessa espansione, attraverso un CSPRNG, della Sign (linea 16), a seconda della versione di CROSS. Si nota poi, come il valore di $\mathbf{u}'$ venga recuperato da un'espansione CSPRNG di $\texttt{Seed}_{u'}$. Di seguito, la Verify calcola i valore delle prime risposte $\mathbf{y}_i$, usando le prime challenge `beta[i]` assieme ad $\mathbf{e}'$ e $\mathbf{u}'$ (linea 21).

Se $\mathbf{b[i]} = 0$, la procedura di verifica ricomputa invece il valore di $\texttt{cmt}_0[i]$. Poi, si verifica che la trasformazione contenuta in $\texttt{rsp}_0$, rappresentata da n elementi nel campo finito $\mathbb{F}_z$ (dentro σ per R-SDP) e da m elementi (dentro δ per R-SDP(G)), è un elemento valido del sottogruppo ristretto G, verificando se tutti i valori del vettore codificato (o σ o δ) sono in un intervallo appropriato $\{0,\cdots,z-1\}$ (linea 24) e, se necessario, ricostruendo σ a partire da δ mediante una moltiplicazione per $\mathbf{M}_G$. Quindi, la trasformazione è applicata all' $\mathbf{y}_i$ fornito, che viene successivamente impiegato per calcolare la sindrome $\tilde{\mathbf{s}}$ (linee 25-29). Svolgere l'hash di: sindrome, trasformazione (σ o δ) e `Salt`, conduce al digest di $\texttt{cmt}_0[i]$. Il corrispondente valore del digest di $\texttt{cmt}_1[i]$ viene recuperato a partire dalla sequenza $\texttt{rsp}_1$ (linea 31).

La Verify può adesso ricomporre sia $\texttt{d}_0$ sia $\texttt{d}_1$, i valori ricostruiti saranno chiamati $\texttt{d}'_0$ e $\texttt{d}'_1$. Il digest $\texttt{d}'_0$ è ottenuto attraverso RecomputeMerkleRoot, che riceve una sequenza di commitment $\texttt{cmt}_0$, dove una parte è ricalcolata ed una parte è recuperata consultando la struttura dati `MerkleProofs`, presente nella `Signature`. La procedura effettua poi un hash gerarchico dei commitment, permettendo di generare $\texttt{d}'_0$ (linea 36). Il digest $\texttt{d}'_1$ è ottenuto performando l'hash della sequenza $\texttt{cmt}_1$ (linea 37). Di seguito, questi due digest sono impiegati per generare mediante hash il digest $\texttt{d}'_{01}$ (linea 38), il quale dovrebbe coincidere con il $\texttt{d}_{01}$ presente nella `Signature` (se essa è valida). L'ultimo calcolo effettuato dalla procedura è la ricostruzione del digest $\texttt{d}'_b$, ottenuto mediante hash delle risposte $\mathbf{y}_i$, seguite da $\texttt{d}'_\beta$. In conclusione, l'algoritmo determina se entrambi i digest $\texttt{d}'_{01}$ e $\texttt{d}'_b$ coincidono con i corrispettivi recuperati dalla `Signature` (linea 40-41): se la firma è valida Verify restituisce `True`, altrimenti restituisce `False`.

2.2.3 Logica progettuale

Di seguito sono elencate, in maniera riassuntiva, tutte le scelte intraprese per la realizzazione dello schema di firma CROSS.

Codici lineari

Dall'avvio del processo di standardizzazione Quantum Resistant del NIST, già tre schemi di firma digitale sono stati considerati validi: CRYSTALS-Dilithium, FALCON e SPHINCS+. Questi sono schemi basati prevalentemente su reticoli, ragione per cui ha senso presentare un progetto che affondi le sue radici su altri principi. Nota, inoltre, l'esistenza di attacchi validi verso gli schemi basati su crittografia a logica multivariata e su isogenie, è prudente appoggiarsi a diverse primitive. Problemi difficili (dalla teoria dei codici), come SDP e sue varianti, hanno una difficoltà di risoluzione worst-case garantita, grazie alle prove di NP-completezza mostrate in [3],[4]. In aggiunta, non ci sono risolutori subesponenziali, ne classici o quantistici per questi problemi, nonostante le continue ricerche [11]. Il problema di decodifica per vettori di errore ristretti è già stato identificato come NP-hard [5], dove i migliori decodificatori mostrano una complessità computazionale maggiore rispetto ai problemi classici.

Protocollo ZK

Impiegare una trasformazione di Fiat-Shamir su un protocollo di identificazione Zero Knowledge garantisce di ottenere uno schema di firma ad elevata sicurezza. Si può garantire sicurezza EUF-CMA attraverso un'esecuzione in parallelo di più istanze.

Errori ristretti

I protocolli ZK basati su codici si legano alla difficoltà del problema basato su metriche di Hamming. Comunque, questi protocolli soffrono di costi elevati a causa di molte permutazioni. L'introduzione di errori ristretti nei protocolli ZK consente di gestire i costi elevati legati alle permutazioni, mantenendo la difficoltà del problema alla base [5]. In aggiunta, la ristrettezza garantisce la possibilità di rappresentare vettori e trasformazioni in maniera compatta.

Scelta di $\mathbb{E}$

Si possono utilizzare anche differenti restrizioni per i vettori di errore. Nonostante ciò, per ridurre i costi di comunicazione di un protocollo ZK (e di conseguenza anche la dimensione della firma), è di fondamentale importanza la scelta di $\mathbb{E}$. Noto che ogni valore del vettore ristretto è dato da g^{ℓ}, allora è richiesto di inviare solamente ℓ. In altre parole, utilizzando l'isometria tra gruppi $(\mathbb{E}^n, \star)$ in $(\mathbb{F}_z^n, +)$, se il gruppo $(\mathbb{E}^n, \star)$ è transitivo, è possibile rappresentare in maniera compatta le trasformazioni; difatto, questa logica richiede solamente $n \log_2(z)$ bit.

Scelta di G

Dato che le equazioni relative alla sindrome sono lineari rispetto all'addizione, avere un sottogruppo moltiplicativo di $\mathbb{E}^n$ non arreca danno alla sicurezza del problema. Considerare il sottogruppo $(G, \star) \subset (\mathbb{E}^n, \star)$ permette di ridurre la dimensione della firma, oltre al fatto che ogni vettore $\mathbf{e}$ (assieme alle trasformazioni associate) può essere rappresentato usando solo $m \log_2(z)$ bit.

Scelta di CROSS-ID

Ci sono molti protocolli ZK basati su codice, anche se si rifericono e basano su SDP con metriche di Hamming. Lavorando invece con Restricted-SDP, si ottiene una riduzione significativa della dimensione della firma. Usando il SDP con metrica di Hamming in GPS [12], è richiesta una firma di almeno 24 kB per avere una sicurezza di 128 bit. La versione corrispondente basata su R-SDP richiede solo 14 kB in dimensione della firma; mentre R-SDP(G) richiede solo 12 kB, ossia la meta della dimensione per la medesima sicurezza. Sono stati considerati molti protocolli ZK e analizzate le performance delle varianti R-SDP in termini di dimensione della firma; quest'ultima risulta il maggiore collo di bottiglia per questi schemi. Dunque, CROSS ID sembra essere il protocollo con migliori prestazioni; ciò è dato dal largo numero di hash svolti negli altri protocolli, il che conduce a vari problemi di natura computazionale.

Scelta dello spazio

R-SDP ed i protocolli ZK possono essere considerati su campi finiti $\mathbb{F}_q$, con q numero primo, in quanto si evitano vulnerabilità legate ad attacchi a sottocampi e sottospazi basati su numeri primi [13].

3

CROSS Single Commitment

Il terzo capitolo introduce le idee, le intuizioni e le operazioni che contraddistinguono l'innovazione dello schema CROSS. Basandosi sulla teoria ampiamente discussa, il protocollo viene rielaborato al fine di consegnare una proposta ad hoc per il contesto di riferimento.

3.1 Proposta di modifica

Di seguito saranno descritti tutti i passaggi che hanno condotto alla proposta di modifica dello schema di CROSS, partendo dall'idea di fondo fino all'implementazione del codice in Python.

Attraverso un approfondito esame dell'architettura di CROSS, la quale incorpora il protocollo di identificazione omonimo, e mediante l'utilizzo della trasformazione di Fiat-Shamir, si ottiene un nuovo schema di firma che mantiene sicurezza EUF-CMA per t esecuzioni parallele. Analizzando lo schema in questione, a partire dal modello teorico di base noto come CVE [6], si è evidenziata la possibilità di eliminare la seconda sequenza di commitment.

Inoltre, è possibile osservare come due dei tre elementi utilizzati nella fase di verifica sono caratterizzati dalle medesime componenti. In particolare, nel caso in cui $\mathbf{b} = 1$, il verifier ricostruisce la prima risposta $\mathbf{y}$, facendo riferimento ai termini $\mathbf{u}'$ ed $\mathbf{e}'$. Questa sorta di ridondanza deriva dall'utilizzo del secondo commitment c_1, il quale si appoggia identicamente ad $\mathbf{u}'$ ed $\mathbf{e}'$ (che originano dallo stesso seme di round $\texttt{Seed}^{(i)}$). Di conseguenza, aver notato come vengano controllati dei dati derivanti dalla stessa fonte, ha portato alla luce l'idea di modificare lo schema. Tuttavia, con l'avvento di questa nuova versione, è necessario riconsiderare tutti gli aspetti di robustezza relativi a CROSS. Si dimostrerà la conservazione di un adeguato livello di sicurezza, oltre al mantenimento delle proprietà fondamentali, con particolare attenzione verso la Soundness ed il relativo errore.

La struttura dello schema rimane 5-step, con uno scambio di messaggi ordinato e ben caratterizzato (commitment, prima challenge, prima risposta, seconda challenge, seconda risposta), passando per CROSS-ID, per poi essere trasformato in uno schema di firma non interattivo attraverso la logica Fiat-Shamir. All'interno di questa versione di CROSS le funzioni hash conservano le caratteristiche di robustezza hiding e binding; inoltre, sono mantenute le ottimizzazioni del caso a due commitment. Bisognerà dimostrare che il protocollo sul quale si appoggia il nuovo schema, è ancora di tipologia $q2$. L'aspetto degno di maggior

interesse è quello relativo al grado di sicurezza e di robustezza, garantiti dalla nuova base matematica.

A livello procedurale, è noto come questa modifica interessi precisamente le primitive di Sign e Verify dello schema precedente, con i singoli elementi $\texttt{cmt}_1[i]$ che erano generati attraverso l'hash di: $\texttt{Seed}[i]$, Salt e l'indice di round i. Successivamente, essi venivano concatenati ed utilizzati come input per un ulteriore hash che consegnava il digest $\texttt{d}_1$, il quale (nell'idea di partenza) ricopriva un ruolo cruciale sia per la generazione delle challenge (essendo uno degli elementi necessari all'ottenimento di $\texttt{d}_{01}$), sia per le operazioni di ricalcolo nella primitiva Verify. Inoltre lo stesso $\texttt{cmt}_1$, con i suoi elementi, popolava il vettore delle seconde risposte $\texttt{rsp}_1$, relative ai round con peso $\mathbf{b} = 0$.

Dunque, attuare un'eliminazione di commitment nello schema, può portare indicativamente ed oggettivamente a migliorie in termini di velocità d'esecuzione e di dimensione della firma a parità di parametri (proprio perchè si riduce il numero di elementi nella Signature, assieme alle operazioni svolte). D'altra parte, è necessario concentrarsi a fondo sull'aspetto teorico, al fine di manipolare il problema sul quale si appoggia lo schema (dovendo mantenere una certa difficoltà), passando per la modifica effettiva dello schema.

3.1.1 Base matematica

In primo luogo, rispetto al protocollo descritto nel capitolo precedente a Fig.2.1, è possibile introdurre una prima modifica parziale. Essa consiste nel comporre il commitment c_1 a partire direttamente dal Seed, invece che dalla coppia $\mathbf{u}'$ ed $\mathbf{e}'$. Tuttavia, questa prima modifica implica un'ulteriore assunzione di sicurezza. Di conseguenza, segue la nuova formulazione del problema, accompagnata da una dimostrazione relativa alla proprietà di soundness, passando dalla versione a due commitment a quella ad uno solo.

Problema

Sia $\ell \in \mathbb{N}$ ed $\mathbb{O}$ un Oracolo Casuale che, sulla base di input come $\texttt{Seed} \in \{0;1\}^\ell$, restituisce (in maniera uniformemente casuale) un vettore $\mathbf{u}' \in \mathbb{F}_q^n$ ed un vettore ristretto $\mathbf{e}' \in \mathbb{E}^n$. Dati $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$ ed $\tilde{\mathbf{s}} \in \mathbb{F}_q^{n-k}$, si rilevino due input $\texttt{Seed}, \texttt{Seed}^* \in \{0;1\}^\ell$, un vettore ristretto $\mathbf{d}$ e due scalari $\beta, \beta' \in \mathbb{F}_q^*$ tali che

1. $\mathbb{O}(\texttt{Seed}) = (\mathbf{u}', \mathbf{e}')$;
2. $\mathbb{O}(\texttt{Seed}^*) = (\mathbf{u}'^*, \mathbf{e}'^*)$;
3. $(\mathbf{d} \star (\mathbf{u}' - \mathbf{u}'^* + \beta\mathbf{e}' - \beta^*\mathbf{e}'^*))\mathbf{H}^\top = (\beta - \beta^*)\mathbf{s}$, dove $\star$ rappresenta la moltiplicazione componente per componente.

La presunta difficoltà del problema descritto è strumentale e propedeutica al funzionamento dello schema.

Difficoltà del problema

Di seguito, si affronta l'ambito relativo alla difficoltà di risolvere il problema descritto. Inizialmente si osserva come il problema possa essere considerato banale se si confronta ad altri problemi difficili. Ad esempio, considerando la metrica di Hamming (non potendo usare in questo caso l'operatore $\star$) , un modo semplice di risolvere il problema sarebbe:

1. scegliere casualmente due seed ed ottenere $\mathbf{u}'$ $\mathbf{u}'^*$, $\mathbf{e}'$ e $\mathbf{e}'^*$;
2. scegliere β e β^* casuali;
3. impostare $\mathbf{x} = \mathbf{u}' - \mathbf{u}'^* + \beta\mathbf{e}' - \beta^*\mathbf{e}'^*$ e comporlo con peso di Hamming w;
4. trovare un vettore $\tilde{\mathbf{x}} \in \mathbb{F}_q^n$ con peso di Hamming w tale che $\tilde{\mathbf{x}}\mathbf{H}^\top = (\beta - \beta^*)\mathbf{s}$;
5. impostare σ come una trasformazione monomiale che trasforma $\mathbf{x}$ in $\tilde{\mathbf{x}}$.

Questa procedura è banale se w non è sufficientemente piccolo. Allo stesso modo, si nota come $\mathbf{x}$ ha un peso di Hamming stimato $n(1-\frac{1}{q})$. Tale valore corrisponde a quello ottenibile risolvendo lo step 4 con semplice algebra lineare (indovinando casualmente gli elementi di $\tilde{\mathbf{x}}$ attraverso un insieme di dati). Dunque, è possibile risolvere il problema con poche chiamate verso l'oracolo e attraverso un numero basso di tentativi per trovare $\tilde{\mathbf{x}}$.

Intuizione sulla difficoltà

Con l'introduzione dei vettori ristretti, invece, il problema risulta meno banale. Sia $\mathbf{x} = \mathbf{u}' - \mathbf{u}^* + \beta\mathbf{u}' - \beta^*\mathbf{e}'^*$ e considerando che $\sigma(\mathbf{x})$ può essere scritto come $\mathbf{d} \star \mathbf{x}$, dove $\mathbf{d}$ è un rappresentazione vettoriale di σ, si nota come

$$\sigma(\mathbf{x})\mathbf{H}^\top = \mathbf{s} \mapsto \mathbf{d}\tilde{\mathbf{H}}^\top = \mathbf{s}, \tag{3.1}$$

dove $\tilde{\mathbf{H}} = \mathbf{H} \star \mathbf{x}$ è una matrice ottenuta moltiplicando ogni colonna di $\mathbf{H}$ con il corrispondente scalare dentro $\mathbf{x}$. Si può discutere sul fatto che, per qualsiasi scelta di `Seed` e `Seed`*, $\mathbf{x}$ non è molto differente da un vettore casuale. Sia noto come $\mathbf{u}'$, $\mathbf{u}'^*$, $\mathbf{e}'$ ed $\mathbf{e}'^*$ sono campionati da un oracolo (modellato come un PRNG). Si assuma di mantenere fisso `Seed` e provare valori differenti di `Seed`*: ogni volta si ottiene una differente coppia ($\mathbf{u}'^*$, $\mathbf{e}'^*$), la quale, quando sommata con altri termini, garantisce la randomizzazione di $\mathbf{x}$. A meno che l'avversario non abbia qualche maniera di rompere o attaccare il PRNG, non si nota la differenza tra $\mathbf{x}$ ed un vettore causale.

Osservando la formula precedente 3.1, i termini cruciali appaiono come un'istanza casuale di R-SDP. Inoltre, $\mathbf{H}$ è una parity-check matrix casuale, $\mathbf{s}$ è una sindrome e $\mathbf{d}$ è un vettore ristretto. Si ottiene, di conseguenza, $\tilde{\mathbf{H}} = \mathbf{H} \star \mathbf{x}$ con $\mathbf{x}$ casuale. Quindi, risolvere 3.1 sembrerebbe difficile come un'istanza causale di R-SDP, catturando la difficoltà del problema a cui si fa riferimento con la modifica.

Di seguito, si studia la special soundness del possibile schema modificato ad un singolo commitment.

Proposizione 1. (Special Soundness)

Successivamente, si elabora la medesima dimostrazione nel caso in cui viene considerata la possibiltà di rimuovere il secondo commitment dalla procedura. Sotto l'assunzione che il precedente problema sia difficile, il nuovo protocollo garantisce (2,2)-out-of-(q,2) special sound.

Dimostrazione. Si considerano quattro transcript $T_1,\ T_2,\ T_3,\ T_4$ i quali, in questo caso, sono strutturati nella seguente maniera:

$$T_1 : (c_0, c_1, \beta, h, \mathbf{y}, \sigma);$$

$$T_2 : (c_0, c_1, \beta, h, \texttt{Seed});$$
$$T_3 : (c_0, c_1, \beta^*, h^*, \mathbf{y}^*, \sigma^*);$$
$$T_4 : (c_0, c_1, \beta^*, h^*, \texttt{Seed}^*).$$

Siano $\mathbf{u}'$ ed $\mathbf{e}'$ i vettori generati a partire da $\texttt{Seed}$, mentre $\mathbf{u}'^*$ ed $\mathbf{e}'^*$ quelli generati da $\texttt{Seed}^*$. A partire da T_2 e T_4, è possibile ottenere $\texttt{Hash(Seed)} = c_1 = \texttt{Hash}(\texttt{Seed}^*)$, il che implica (a meno di collisioni)

$$\texttt{Seed} = \texttt{Seed}^*.$$

Adesso, $\texttt{Seed}$ è usato per generare $\mathbf{u}'$ ed $\mathbf{e}'$; notando che $h = \texttt{Hash}(\mathbf{y})$, con $\mathbf{y} = \mathbf{u}' + \mathbf{e}'$. Sia $\mathbf{x} = \mathbf{u}' + \beta\mathbf{e}'$: sia T_2 un transcript accettato/valido, si avrà che $h = \texttt{Hash}(\mathbf{x})$. Allo stesso modo se T_1 è un transcript valido e ciò sottolinea come $\mathbf{y}$ e $\mathbf{x}$ sono differenti e formano una collisione hash, oppure sono uguali e dunque $\mathbf{y} = \mathbf{u}' + \beta'\mathbf{e}'$. Secondo un analogo ragionamento, otteniamo $\mathbf{y}^* = \mathbf{u}'^* + \beta^*\mathbf{e}'^*$. Si nota anche, a differenza della prova relativa al protocollo precedente, che rimuovendo c_1 non si riesce a garantire che $\mathbf{u}' = \mathbf{u}'^*$ ed $\mathbf{e}' = \mathbf{e}'^*$; quindi, si considereranno queste coppie come distinte. Adesso, si considerano T_1 e T_3. Inizialmente, dato che c_0 è derivato legandosi a σ e σ^*, si può garantire come $\sigma = \sigma^*$. Di seguito, si ottengono queste coppie di relazioni:

$$\sigma(\mathbf{y})\mathbf{H}^\top - \beta\mathbf{s} = \sigma(\mathbf{u}' + \beta\mathbf{e}')\mathbf{H}^\top - \beta\mathbf{s} = \tilde{\mathbf{s}},$$

$$\sigma(\mathbf{y}^*)\mathbf{H}^\top - \beta^*\mathbf{s} = \sigma(\mathbf{u}'^* + \beta^*\mathbf{e}'^*)\mathbf{H}^\top - \beta^*\mathbf{s} = \tilde{\mathbf{s}},$$

dalle quali origina

$$\sigma(\mathbf{u}' - \mathbf{u}'^* + \beta\mathbf{e}' - \beta^*\mathbf{e}'^*)\mathbf{H}^\top = (\beta - \beta*)\mathbf{s}.$$

Sotto l'assunzione che $\mathbf{u}' = \mathbf{u}'^*$ ed $\mathbf{e}' = \mathbf{e}'^*$, si conclude con una prova o dimostrazione simile se non identica a quella proposta per il protocollo precedente. Dunque, bisogna osservare che o vale quanto detto, oppure l'avversario è capace di risolvere un problema difficile. Si tenga a mente come σ debba essere un'isometria ristretta: questa proprietà garantisce la difficoltà del problema.

Proposizione 2. (Special Soundness)

Sotto l'assunzione che il precedente problema sia difficile, il protocollo in Fig.3.1 garantisce (2,2)-out-of-(q,2) special sound.

Dimostrazione. Si considerano quattro transcript $T_1,\ T_2,\ T_3,\ T_4$ i quali, in questo caso, sono strutturati nella seguente maniera:

$$T_1 : (c, \beta, h, \mathbf{y}, \sigma);$$
$$T_2 : (c, \beta, h, \texttt{Seed});$$
$$T_3 : (c, \beta^*, h^*, \mathbf{y}^*, \sigma^*);$$
$$T_4 : (c, \beta^*, h^*, \texttt{Seed}^*).$$

Siano $\mathbf{u}'$ ed $\mathbf{e}'$ i vettori generati a partire da $\texttt{Seed}$, mentre $\mathbf{u}'^*$ ed $\mathbf{e}'^*$ quelli generati da $\texttt{Seed}^*$. Sia $\mathbf{x} = \mathbf{u}' + \beta\mathbf{e}'$: dato che T_2 è un transcript accettato/valido, si avrà che $h = \texttt{Hash}(\mathbf{x})$. Allo stesso modo T_1 è un transcript valido e ciò sottolinea come $\mathbf{y}$ e $\mathbf{x}$ sono differenti e formano una

collisione hash, oppure sono uguali e dunque $\mathbf{y} = \mathbf{u}' + \beta'\mathbf{e}'$. Secondo un analogo ragionamento, otteniamo $\mathbf{y}^* = \mathbf{u}'^* + \beta^*\mathbf{e}'^*$. Si nota anche, a differenza della prova relativa al protocollo precedente, che rimuovendo c_1 non si riesce a garantire che $\mathbf{u}' = \mathbf{u}'^*$ ed $\mathbf{e}' = \mathbf{e}'^*$; quindi, si considereranno queste coppie come distinte. Adesso, si considerano T_1 e T_3. Inizialmente, dato che c è derivato legandosi a σ e σ^*, si può garantire come $\sigma = \sigma^*$. Di seguito, si ottengono queste coppie di relazioni:

$$\sigma(\mathbf{y})\mathbf{H}^\top - \beta\mathbf{s} = \sigma(\mathbf{u}' + \beta\mathbf{e}')\mathbf{H}^\top - \beta\mathbf{s} = \tilde{\mathbf{s}},$$

$$\sigma(\mathbf{y}^*)\mathbf{H}^\top - \beta^*\mathbf{s} = \sigma(\mathbf{u}'^* + \beta * \mathbf{e}'^*)\mathbf{H}^\top - \beta^*\mathbf{s} = \tilde{\mathbf{s}},$$

dalle quali origina

$$\sigma(\mathbf{u}' - \mathbf{u}'^* + \beta\mathbf{e}' - \beta^*\mathbf{e}'^*)\mathbf{H}^\top = (\beta - \beta*)\mathbf{s}.$$

Sotto l'assunzione, infatti, che $\mathbf{u}' = \mathbf{u}'^*$ ed $\mathbf{e}' = \mathbf{e}'^*$, si conclude con una prova o dimostrazione simile se non identica a quella proposta per il protocollo precedente. Dunque, bisogna osservare che o vale quanto detto, oppure l'avversario è capace di risolvere un problema difficile. Si tenga a mente come σ debba essere un'isometria ristretta: questa proprietà garantisce la difficoltà del problema, anche per la versione ad un commitment.

Assunto che il problema introdotto sia sufficientemente difficile, senza variare di molto i parametri (dei quali si discuterà nel capitolo successivo), le performance aumentano e tutto ciò offre un miglioramento significativo allo schema.

3.1.2 Confronto tra CROSS e CROSS Single Commitment

Dimensione della firma

Come già descritto precedentemente, il principale collo di bottiglia prestazionale dello schema CROSS concerne la dimensione della firma digitale; di conseguenza, è di primaria importanza verificare se, difatto, la nuova versione dell'algoritmo garantisce migliorie rispetto all'originale. Il passaggio da SDP all'alternativa ristretta di per se garantisce una miglioria da 24 kB a 14 kB, per avere una sicurezza di 128 bit; allo stesso modo, una riduzione ulteriore si ottiene con R-SDP(G) con 12 kB.

L'attuale versione di CROSS che ottimizza il tempo d'esecuzione, trascurando in parte la dimensione della firma, non fa riferimento a strutture dati ad albero come Merkle e Seed. La dimensione delle firma relativa alla versione di CROSS SPEED è pari a:

$$|\texttt{Signature}| = \underbrace{8\lambda}_{\texttt{Salt},c_0,c_1,h} + w \cdot \underbrace{3\lambda}_{\texttt{Risposte per b[i]=1}} + (t-w)\underbrace{(2\lambda + n\lceil\log_2(q)\rceil + m\lceil\log_2(z)\rceil)}_{\texttt{Risposte per b[i]=0}} \tag{3.2}$$

Impiegando CROSS Single Commmitment la nuova dimensione sarà:

$$|\texttt{Signature}| = \underbrace{6\lambda}_{\texttt{Salt},c,h} + w \cdot \underbrace{3\lambda}_{\texttt{Risposte per b[i]=1}} + (t-w)\underbrace{(n\lceil\log_2(q)\rceil + m\lceil\log_2(z)\rceil)}_{\texttt{Risposte per b[i]=0}} \tag{3.3}$$

Sia noto come la differenza è data dal fatto che con $b[i] = 0$ non c'è più necessità di inviare il commitment c_1.

La versione di CROSS che ottimizza la dimensione della firma, anche chiamata SIG_SIZE, utilizza strutture dati ad albero e rinuncia ad un'esecuzione rapida al fine di ottenere un output assai ridotto. La dimensione nell'originale è:

$$|\texttt{Signature}| = \underbrace{8\lambda}_{h,\ c_0,\ c_1,\ \texttt{Salt}} + \underbrace{\lambda(t-w)\log_2\left(\frac{t}{t-w}\right)}_{\texttt{SeedPath}} + \underbrace{2\lambda\left(1+(t-w)\log_2\left(\frac{t}{t-w}\right)\right)}_{\texttt{MerkleProof}} + \underbrace{(t-w)\left(\underbrace{2\lambda}_{c_1^{(i)}} + \underbrace{n\lceil\log_2(q)\rceil}_{\mathbf{y}^{(i)}} + \underbrace{m\lceil\log_2(z)\rceil}_{\sigma^{(i)}}\right)}_{f^{(i)},\ i \notin J}. \tag{3.4}$$

Usando il nuovo protocollo, la firma diventa pari a:

$$|\texttt{Signature}| = \underbrace{6\lambda}_{h,\ c,\ \texttt{Salt}} + \underbrace{\lambda(t-w)\log_2\left(\frac{t}{t-w}\right)}_{\texttt{SeedPath}} + \underbrace{2\lambda\left(1+(t-w)\log_2\left(\frac{t}{t-w}\right)\right)}_{\texttt{MerkleProof}} + \underbrace{(t-w)\left(\underbrace{n\lceil\log_2(q)\rceil}_{\mathbf{y}^{(i)}} + \underbrace{m\lceil\log_2(z)\rceil}_{\sigma^{(i)}}\right)}_{f^{(i)},\ i \notin J}. \tag{3.5}$$

Dunque, il guadagno della firma è identico in entrambe le versioni, in quanto interessa solamente i termini $2\lambda + (t-w)2\lambda$. Di conseguenza, questa modifica impatta in percentuali differenti sui due output. Considerando un parametro di sicurezza pari a $\lambda = 128, t = 243, w = 206, q = 509, m = 24$ e $z = 127$ l'effettivo guadagno, derivante dalla rimozione del secondo commitment, risulta pari a 9728 bit complessivi. Nel caso SPEED, a parametri fissi, da una firma complessiva di 109802 bit si ottiene una riduzione a 100074, pari al 8,86%; d'altra parte, la versione SIG_SIZE passa da 69528 bit a 59800, ossia il 13,99%. La quantità di bit rimossa è fissa per entrambe le versioni, il che ha un impatto percentuale differente sulle due ottimizzazioni.

Tempo d'esecuzione

Sul tempo d'esecuzione non è possibile concludere nulla in materia di riduzioni. Nonostante ciò, il tempo d'esecuzione per le versioni SIG_SIZE e SPEED a parità di parametri impiegati dovrà risultare ridotto o al massimo mantenersi identico al caso precedente, in quanto le operazioni da svolgere sono ridotte. Di conseguenza, il discorso sul tempo d'esecuzione è da rimandare al capitolo successivo, dove ci si concentrerà sulla profilazione del codice C. Per quanto concerne la teoria, il tempo d'esecuzione è strettamente legato sia al parametro t sia al peso di Hamming w; nello specifico, a rigor di logica, più t è piccolo e meno esecuzioni parallele del protocollo vengono realizzate. D'altra parte, gioca un ruolo cruciale anche il peso, il quale identifica i round della procedura nei quali il prover è tenuto a scambiare solamente un seed oppure tutte le informazioni relative alla trasformazione e alle risposte; quindi, più grande è l'intervallo $t-w$ e maggiore è l'onerosità di ogni round. Bisogna, dunque, tenere conto di entrambe queste grandezze per lo studio delle prestazioni dell'algoritmo, congiuntamente alla valutazione delle istanze più promettenti per le versioni di CROSS.

3.1.3 Considerazioni crittografiche

Quindi, la proposta di modifica allo schema CROSS garantisce teoricamente una miglioria in termini di dimensione della firma, sulla base dei calcoli condotti precedentemente; inoltre, offrire anche un tempo d'esecuzione migliore o al massimo uguale (quest'ultimo aspetto dipenderà molto dalla scelta dei parametri utilizzati, come spiegato prima) evidenzierebbe ulteriormente questa alternativa come valida. Il nuovo schema si appoggia ad un problema riconducibile ad un'istanza di R-SDP, problema che risulta NP-hard e, allo stesso tempo, è basato su uno protocollo di identificazione $q2$ che rispetta le proprietà di Zero Knowledge, Completeness e Soundness (stessa dimostrazione del Capitolo 2). Applicando, come ampiamente discusso, la trasformazione di Fiat-Shamir su t esecuzioni parallele della nuova procedura si ottiene ancora una volta uno schema di firma con sicurezza EUF-CMA; il che rafforza l'ipotesi relativa alla robustezza.

Private Key	restricted vector $\mathbf{e} \in \mathbb{F}_q^n$
Public Key	$\mathbb{E}, \mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}, \mathbf{s} = \mathbf{e}\mathbf{H}^\top \in \mathbb{F}_q^{n-k}$

PROVER		VERIFIER
Sample $\mathtt{Seed} \xleftarrow{\$} \{0;1\}^\lambda$		
Sample $\left(\mathtt{Seed}^{(\mathbf{u}')}, \mathtt{Seed}^{(\mathbf{e}')}\right) \xleftarrow{\mathtt{Seed}} \{0;1\}^{2\lambda}$		
Sample $\mathbf{u}' \xleftarrow{\mathtt{Seed}^{(\mathbf{u}')}} \mathbb{F}_q^n$		
Sample $\mathbf{e}' \xleftarrow{\mathtt{Seed}^{(\mathbf{e}')}} \mathbb{E}^n$		
Compute $\sigma \in \mathbb{E}^n$ such that $\sigma(\mathbf{e}') = \mathbf{e}$		
Set $\mathbf{u} = \sigma(\mathbf{u}')$		
Compute $\widetilde{\mathbf{s}} = \mathbf{u}\mathbf{H}^\top$		
Set $c = \mathsf{Hash}\left(\widetilde{\mathbf{s}}, \sigma\right)$		
	$\xrightarrow{c}$	
	$\xleftarrow{\beta}$	Sample $\beta \xleftarrow{\$} \mathbb{F}_q^*$
Compute $\mathbf{y} = \mathbf{u}' + \beta\mathbf{e}'$		
Set $h = \mathsf{Hash}(\mathbf{y})$		
	$\xrightarrow{h}$	
		Sample $b \xleftarrow{\$} \{0,1\}$
	$\xleftarrow{b}$	
If $b = 0$, set $f := \left(\mathbf{y}, \sigma\right)$		
If $b = 1$, set $f := \mathtt{Seed}$	$\xrightarrow{f}$	
		If $b = 0$:
		Compute $\widetilde{\mathbf{y}} = \sigma(\mathbf{y})$ and $\widetilde{\mathbf{s}} = \widetilde{\mathbf{y}}\mathbf{H}^\top - \beta\mathbf{s}$
		Accept if:
		1) $\mathsf{Hash}(\mathbf{y}) = h$
		2) $\mathsf{Hash}\left(\widetilde{\mathbf{s}}, \sigma\right) = c$
		3) $\sigma \in \mathbb{E}^n$
		If $b = 1$:
		Sample $\left(\mathtt{Seed}^{(\mathbf{u}')}, \mathtt{Seed}^{(\mathbf{e}')} \xleftarrow{\mathtt{Seed}} \{0;1\}^{2\lambda}\right)$
		Set $\mathbf{y} = \mathbf{u}' + \beta\mathbf{e}'$
		Accept if:
		$\mathsf{Hash}(\mathbf{y}) = h$

Figura 3.1: Schema di CROSS-ID ottimizzato ad un singolo commitment (R-SDP)

3.2 Fiat Shamir e lo schema di firma CROSS SC

Facendo riferimento alle proposizioni ed alle dimostrazioni delle sezioni precedenti è possibile confermare come, mediante la trasformazione di Fiat-Shamir sul nuovo protocollo di identificazione CROSS-ID, per t esecuzioni parallele, si otteniene uno schema di firma EUF-CMA [2].

Di netto, nonostante le modifiche introdotte, c'è ancora la necessità di discutere le principali differenze che interessano il nuovo schema rispetto a quello originale.

Tali divergenze sono le seguenti:

- **Eliminazione del secondo commitment**, eliminando la ridondanza di cui si è discusso;

- **Riduzione delle operazioni di firma**, evitando l'hash del seed di round, del `Salt` e dell'indice i, che compone la sequenza $\texttt{cmt}_1$;
- **Riduzione della firma**, all'interno della `Signature` non è più presente la seconda risposta $\texttt{rsp}_1$.

D'altra parte, si mantengono molte assunzioni relative alla versione precedente, come l'utilizzo di un `Salt` di 2λ bit per evitare collisioni hash. Inoltre, le ottimizzazioni volte a ridurre i costi complessivi di comunicazione sono ancora adottati nel nuovo schema:

- **Seconda challenge a peso di Hamming fisso**, volta a monitorare i costi e la sicurezza dello schema;
- **Impiego di Seed e Merkle tree**, finalizzati a realizzare sequenze più compatte;
- **Posticipare la Verifica della Prima Risposta**, per evitare eventuali colli di bottiglia nel centro dello schema;
- **Riduzione della Dimensione dei Commitment**, in ogni round il verifier ricompone il commitment solamente se garantisce la presenza di un $\mathbf{b} = 0$ all'interno del vettore, per aver accesso all'elemento $c_0^{(i)}$. Non si hanno più i $c_1^{(i)}$, motivo per il quale non si ha la caratteristica per la quale ad ogni round il verifier ottiene o ricalcola sicuramente uno dei due commitment. Successivamente, si prepara ancora il Merkle Tree T usando come foglie $c_0^{(1)}, \cdots, c_0^{(t)}$. L'elemento di fondamentale interesse è ancora c_0, ossia la radice di T. Il verifier, attraverso gli elementi noti quando $\mathbf{b} = 0$ ed i restanti ottenuti mediante struttura dati `Merkle Proof`, ricompone l'albero e la radice. Il limite superiore del costo totale associato al commitment si mantiene identico al caso precedente.

3.2.1 Schema di firma risultante

Lo schema, comprensivo delle primitive di generazione e verifica della firma, subisce alcune modifiche conseguenti all'eliminazione del commitment. Considerate tutte le ottimizzazioni proposte, lo schema finale segue le Fig.3.2 e 3.3.

Signing:

1. campiona un $\texttt{Salt} \xleftarrow{\$} \{0;1\}^{2\lambda}$;

2. campiona un $\texttt{MSeed} \xleftarrow{\$} \{0;1\}^{\lambda}$ e crea un albero di seed che ha come t foglie gli elementi $\texttt{Seed}^{(1)}, \cdots, \texttt{Seed}^{(t)}$. Il singolo $\texttt{Seed}^{(i)}$ è necessario per comporre $\mathbf{u}'^{(i)}$ e $\mathbf{e}'^{(i)}$, i quali sono impiegati nel round i;

3. per i round $i = 1, \cdots, t$ calcola la trasformazione ristretta $\sigma^{(i)}$ ed i commitment $c_0^{(i)}$, come definiti precedentemente. Inoltre, utilizza il `Salt` e l'indice di round i all'interno delle funzioni hash;

4. costruisce l'albero di Merkle T con i commitment $c_0^{(1)}, \cdots, c_0^{(t)}$;

5. genera il vettore della prima challenge $(\beta^{(1)}, \cdots, \beta^{(t)})$ usando il messaggio, il salt ed i commitment;

6. calcola $(\mathbf{y}^{(1)},\cdots,\mathbf{y}^{(t)})$ come descritto in CROSS-ID, per poi generare h attraverso il loro hash:

7. genera il vettore della seconda challenge $\mathbf{b} = (b^{(1)},\cdots,b^{(t)}) \in \{0;1\}^t$ a partire dall'hash di messaggio, `Salt`, commitment, risposte ed h. Questo vettore ha un peso di Hamming w fisso. Inoltre, definisce l'insieme J come il supporto di b, che caratterizza gli indici i tali per cui $b^{(i)} = 1$;

8. calcola `SeedPath` come l'insieme dei nodi intermedi dell'albero di seed, necessari per ricomporre tutti i $\texttt{Seed}^{(i)}$, per $i \in J$;

9. imposta `MerkleProof` come la prova per le foglie $\{c_0^{(i)}\}_{i\notin J}$, al fine di ricalcolare la radice dell'albero di Merkle;

10. la firma ottenuta è

$$\texttt{Signature} = \big\{\texttt{Salt}, c_0, h, \texttt{SeedPath}, \texttt{MerkleProof}(T_0), \{\mathbf{y}^{(i)}, \sigma^{(i)}\}_{i\notin J}\big\}.$$

Private Key $\mathbf{e} \in G$
Public Key $G \subseteq \mathbb{E}^n$, $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$, $\mathbf{s} = \mathbf{e}\mathbf{H}^\top \in \mathbb{F}_q^{n-k}$

Input *Message* `Msg`
Output *Signature* `Signature`

SIGNER	VERIFIER
Sample $\texttt{MSeed} \xleftarrow{\$} \{0;1\}^\lambda$, $\texttt{Salt} \xleftarrow{\$} \{0;1\}^{2\lambda}$	
Generate $(\texttt{Seed}^{(1)},\cdots,\texttt{Seed}^{(t)}) = \texttt{SeedTree(MSeed,Salt)}$	
For $i = 1,\cdots,t$:	
Sample $(\texttt{Seed}^{(u')},\texttt{Seed}^{(e')}) \xleftarrow{\texttt{Seed}^{(i)}} \{0;1\}^{2\lambda}$	
Sample $\mathbf{u}'^{(i)} \xleftarrow{\texttt{Seed}^{(u')}} \mathbb{F}_q^n$, $\mathbf{e}'^{(i)} \xleftarrow{\texttt{Seed}^{(e')}} G$	
Compute $\sigma^{(i)} \in G$ tale che $\sigma^{(i)}(\mathbf{e}'^{(i)}) = \mathbf{e}$	
Set $\mathbf{u}^{(i)} = \sigma^{(i)}(\mathbf{u}'^{(i)})$	
Compute $\tilde{\mathbf{s}}^{(i)} = \mathbf{u}^{(i)}\mathbf{H}^\top$	
Set $c_0^{(i)} = \texttt{Hash}(\tilde{\mathbf{s}}^{(i)}, \sigma^{(i)}, \texttt{Salt}, i)$	
Set $T = \texttt{MerkleTreee}(c_o^{(1)},\cdots,c_o^{(t)})$	
Compute $c_o = T.\texttt{Root}()$	
Generate $(\beta^{(1)},\cdots,\beta^{(t)}) = \texttt{GenCh}_1(c_o, \texttt{Msg}, \texttt{Salt})$	
For $i = 1,\cdots,t$:	
Compute $\mathbf{y}^{(i)} = \mathbf{u}'^{(i)} + \beta^{(i)}\mathbf{e}'^{(i)}$	
Compute $h^{(i)} = \texttt{Hash}(\mathbf{y}^{(i)})$	
Compute $h = \texttt{Hash}(h^{(1)},\cdots,h^{(t)})$	
Generate $(\mathbf{b}^{(1)},\cdots,\mathbf{b}^{(t)}) = \texttt{GenCh}_2(c_o, \beta^{(1)},\cdots,\beta^{(t)}, h, \texttt{Msg}, \texttt{Salt})$	
Set $J = \{i \mid \mathbf{b}^{(i)} = 1\}$	
Set $\texttt{SeedPath} = \texttt{SeedPath(MSeed, Salt, J)}$	
For $i \notin J$:	
Set $f^{(i)} := (\mathbf{y}^{(i)}, \sigma^{(i)})$	
Compute $\texttt{MerkleProofs} = T.\texttt{Proofs}(\{1,\cdots,t\} \setminus J)$	
Set $\texttt{Signature} = \{\texttt{Salt}, c_0, h, \texttt{SeedPath}, \texttt{MerkleProofs}, \{f^{(i)}\}_{i\notin J}\}$	
	$\xrightarrow{\texttt{Signature}}$

Figura 3.2: Lo schema di firma CROSS SC: generazione della firma

Verification:

1. genera il vettore della prima challenge $(\beta^{(1)},\cdots,\beta^{(t)})$ a partire da `Msg`, `Salt` e c_0;

2. genera il vettore della seconda challenge $(b^{(1)},\cdots,b^{(t)})$ a partire da `Msg`, `Salt`, c_0, $\beta^{(1)},\cdots,\beta^{(t)}$ e h;

3. usando `SeedPath`, genera i seed $\{\texttt{Seed}^{(i)}\}_{i\in J}$;

4. per $i \in J$, ricalcola $\mathbf{y}^{(1)}$ e $h^{(1)}$;

5. per $i \notin J$, calcola $h^{(i)} = \texttt{Hash}(\mathbf{y}^{(i)})$;

6. utilizza la `MerkleProof` e $\{c_0^{(i)}\}_{i \notin J}$ per ricalcolare e verificare la radice c_0;

7. verifica $h = \texttt{Hash}(h^{(1)}, \cdots, h^{(t)})$.

Private Key $\mathbf{e} \in G$
Public Key $G \subseteq \mathbb{E}^n$, $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$, $\mathbf{s} = \mathbf{e}\mathbf{H}^\top \in \mathbb{F}_q^{n-k}$
Input *Message* `Msg`
Output *Signature* `Signature`

PROVER		VERIFIER
	$\xrightarrow{\texttt{Signature}}$	Generate $(\beta^{(1)}, \cdots, \beta^{(t)}) = \texttt{GenCh}_1(c_0, \texttt{Msg}, \texttt{Salt})$
		Generate $(b^{(1)}, \cdots, b^{(t)}) = \texttt{GenCh}_2(c_0, \beta^{(1)}, \beta^{(t)}, h, \texttt{Msg}, \texttt{Salt})$
		Set $J = \{i \mid b^{(i)} = 1\}$ Generate $\{\texttt{Seed}_{i\in J}^{(i)} = \texttt{GetSeeds(SeedPath, Salt)}\}$
		For $i \in J$: Set $(\texttt{Seed}^{(\mathbf{u'})}, \texttt{Seed}^{(\mathbf{e'})}) \xleftarrow{\texttt{Seed}^{(i)}} \{0;1\}^{2\lambda}$ Compute $\mathbf{u'}^{(i)} \xleftarrow{\texttt{Seed}^{(\mathbf{u'})}} \mathbb{F}_q^n$, $\mathbf{e'}^{(i)} \xleftarrow{\texttt{Seed}^{(\mathbf{e'})}} G$ Compute $\mathbf{y}^{(i)} = \mathbf{u'}^{(i)} + \beta^{(i)}\mathbf{e'}^{(i)}$ Compute $h^{(i)} = \texttt{Hash}(\mathbf{y}^{(i)})$
		For $i \notin J$: Set $h^{(i)} = \texttt{Hash}(\mathbf{y}^{(i)})$ Compute $\tilde{\mathbf{s}}^{(i)} = \sigma^{(i)}(\mathbf{y}^{(i)})\mathbf{H}^\top - \beta^{(i)}\mathbf{s}$
		Verify $h = \texttt{Hash}(h^{(1)}, \cdots, h^{(t)})$ Verify $c_0 = \texttt{Hash}(c_0^{(1)}, \cdots, c_0^{(t)})$

Figura 3.3: Lo schema di firma CROSS SC: verifica della firma

Inoltre, le funzioni per la costruzioni degli alberi e la generazione delle challenge sono, come nel caso precedente, implicite e saranno approfondite nella sezione implementativa. La dimensione della chiave pubblica `pk` è

$$|\texttt{pk}| = (n-k)\lceil \log_2(q) \rceil + \lambda.$$

La dimensione della firma è già stata calcolata e stimata nell'equazione 3.5.

3.2.2 Descrizione procedurale di CROSS SC

La descrizione del flusso di operazioni svolte dal protocollo si suddivide ancora una volta nelle tre primitive, viste nel caso classico: KeyGen, Sign e Verify. Anche in questo caso è

Tabella 3.1: Tabella degli elementi da protocollo a pseudocodice

Protocollo	**Pseudocodice**	**Semantica**
$\ell_G(\mathbf{e})$	ζ	Vettore in $\mathbb{F}_z^m$ per $\mathbf{e}$
$\ell(\mathbf{e})$	η	$\zeta\mathbf{M}_G = \ell_G(\mathbf{e})\mathbf{M}_G \in \mathbb{F}_z^n$ per $\mathbf{e}$
$\ell_G(\sigma^{(i)})$	δ_i	Vettore in $\mathbb{F}_z^m$ for $\sigma^{(i)}$
$\ell(\mathbf{e}'^{(i)})$	$\tilde{\eta}_i$	Vettore in $\mathbb{F}_z^n$ per $\mathbf{e}'^{(i)}$
$\ell(\sigma^{(i)})$	σ_i	Vettore in $\mathbb{F}_z^n$ per $\sigma^{(i)}$
$\sigma^{(i)}$	$\mathbf{v}$	Trasf. su $\mathbb{E}^n$, risp. G
$\mathbf{u}^{(i)}$	$\mathbf{u}$	Trasf. $\mathbf{u}'^{(i)}$
$\tilde{\mathbf{s}}^{(i)}$	$\tilde{\mathbf{s}}$	Sindrome di $\mathbf{u}^{(i)}$
$c_0^{(i)}$	$\mathtt{cmt}_0[i]$	Commitment 0 del round i
$c_0 = \mathcal{T}.\mathtt{root}()$	d_0	Radice del Merkle tree $\mathcal{T}$ del commitment 0
	d_m	Hash del messaggio
	d_β	Hash di d_m,d_0, $\mathtt{Salt}$
$(\beta^{(1)}, \ldots, \beta^{(t)})$	$\mathtt{beta}$	Prima challenge
$\mathbf{e}'^{(i)}$	$\tilde{\mathbf{e}}$	Trasf. $\mathbf{e}$
$\mathbf{y}^{(i)}$	$\mathbf{y}_i$	Risposta alla prima challenge
h	d_b	Hash di $y^{(1)}, \ldots, y^{(t)}$
$b^{(1)}, \ldots, b^{(t)}$	$\mathbf{b}$	Seconda challenge
$f^{(i)}$	$\mathtt{rsp}_0$	Risposta del round i

possibile analizzare il problema sia nel caso R-SDP (a vettori ristretti in $\mathbb{E}^n$) sia nel caso R-SDP(G) (a vettori ristretti nel sottogruppo G). Viene mantenuta la medesima notazione del capitolo precedente, relativa all'utilizzo dei caratteri.

Generazione delle chiavi

La primitiva di KeyGen sfrutta le medesime operazioni del caso classico. Non è presente alcun tipo di modifica.

Algorithm 1: KeyGen()

Data: λ: security parameter,
$g \in \mathbb{F}_p^*$: generator of a subgroup $\mathbb{E}$ of $\mathbb{F}_p^*$ with cardinality z
$\mathbb{E}^n$: restricted subgroup
M_G: $m \times n$ matrix of $\mathbb{F}_z$ elements, employed to generate vectors $\boldsymbol{\eta} \in G \subset \mathbb{E}^n$

Input: None

Output: $\mathtt{pri} : \mathtt{Seed}_{\mathtt{sk}}$: private key seed;
$\mathtt{pub} : (\mathtt{Seed}_{\mathtt{pk}}, \mathrm{s})$ public key: $\mathtt{Seed}_{\mathtt{pk}}$ is a λ bit seed, to derive the non-systematic portion of a random parity-check matrix $\mathbf{H}$ and the $m \times n$ matrix of $\mathbb{F}_z$ elements M_G;
$\mathrm{s} \in \mathbb{F}_p^{n-k}$ is the syndrome of e through $\mathbf{H}$

1 $\mathtt{Seed}_{\mathtt{sk}} \xleftarrow{\$} \{0,1\}^\lambda$
2 $(\mathtt{Seede}, \mathtt{Seed}_{\mathtt{pk}}) \leftarrow \mathrm{CSPRNG}\left(\mathtt{Seed}_{\mathtt{sk}}, \{0,1\}^\lambda \times \{0,1\}^\lambda\right)$
3
$(\mathtt{SeedV}, \mathtt{SeedW}) \leftarrow \mathrm{CSPRNG}\left(\mathtt{Seed}_{\mathtt{pk}}, \{0,1\}^\lambda \times \{0,1\}^\lambda\right)$ $\mathrm{V} \leftarrow \mathrm{CSPRNG}\left(\mathtt{SeedV}, \mathbb{F}_p^{(n-k)\times k}\right)$	$\mathrm{V} \leftarrow \mathrm{CSPRNG}\left(\mathtt{Seed}_{\mathtt{pk}}, \mathbb{F}_p^{(n-k)\times k}\right)$

4 $\mathbf{H} \leftarrow [\mathbf{I}_{n-k} \mid \mathbf{V}]$
5
$\mathrm{W} \leftarrow \mathrm{CSPRNG}\left(\mathtt{SeedW}, \mathbb{F}_z^{m\times(n-m)}\right)$ $\mathrm{M}_G \leftarrow [\mathbf{I}_m \mid \mathrm{W}]$ $\boldsymbol{\zeta} \leftarrow \mathrm{CSPRNG}\left(\mathtt{Seede}, \mathbb{F}_z^m\right)$ $\boldsymbol{\eta} \leftarrow \boldsymbol{\zeta}\mathrm{M}_G$	$\boldsymbol{\eta} \leftarrow \mathrm{CSPRNG}\left(\mathtt{Seede}, \mathbb{F}_z^n\right)$

6 **for** $j \leftarrow 0$ **to** $n-1$ **do**
7 $\quad \mathrm{e}[j] \leftarrow g^{\boldsymbol{\eta}[j]}$
8 **end**
9 $\mathrm{s} \leftarrow \mathbf{e}\mathbf{H}^\top$
10 $\mathtt{pri} \leftarrow \mathtt{Seed}_{\mathtt{sk}}$
11 $\mathtt{pub} \leftarrow (\mathtt{Seed}_{\mathtt{pk}}, \mathrm{s})$
12 **return** (pri,pub);

Figura 3.4: Primitiva KeyGen di CROSS SC

Generazione della firma

La struttura risulta simile al caso classico. L'eliminazione del secondo commitment riduce il numero di chiamate alla funzione hash, evitando di eseguire l'hashing di $\mathtt{Seed}[i]$, $\mathtt{Salt}$ e l'indice di round i. Successivamente, non si realizza il calcolo di $\mathtt{d}_1$ per generare $\mathtt{d}_{01}$ e di seguito $\mathtt{d}_\beta$, ma solamente $\mathtt{d}_0$. L'ultima differenza, da questo punto di vista, sta nella rimozione del vettore $\mathtt{rsp}_1$ relativo alla seconda risposta, il quale non deve essere utilizzato ne definito; da questo concetto origina il principale risultato in termini di dimensione della firma.

Algorithm 2: SIGN(pri, Msg)

Data: λ: security parameter,
$g \in \mathbb{F}_p^*$: generator of a subgroup $\mathbb{E}$ of $\mathbb{F}_p^*$ with cardinality z
$\mathbb{E}^n$: restricted subgroup
M_G: $m \times n$ matrix of $\mathbb{Z}_z$ elements, employed to generate vectors $\boldsymbol{\eta} \in G \subset \mathbb{E}^n$
t: number of iterations of the ZKID protocol
$\mathcal{B}_w^t$: set of all binary strings with length w and Hamming weight t
Input: pri: private key constituted of $\mathtt{Seed}_{\mathtt{sk}} \in \{0,1\}^\lambda$
Msg: message to be signed $\mathtt{Msg} \in \{0,1\}^*$
Output: Signature Signature

Begin
// Key material expansion
$\boldsymbol{\zeta}, \mathrm{H}, \mathrm{M}_G \leftarrow$ EXPANDPRIVATESEED($\mathtt{Seed}_{\mathtt{sk}}$) | $\boldsymbol{\eta}, \mathrm{H} \leftarrow$ EXPANDPRIVATESEED($\mathtt{Seed}_{\mathtt{sk}}$)
// Computation of commitments
$\mathtt{Mseed} \xleftarrow{\$} \{0,1\}^\lambda$, $\mathtt{Salt} \xleftarrow{\$} \{0,1\}^{2\lambda}$
$(\mathtt{Seed}[0], \ldots, \mathtt{Seed}[t-1]) \leftarrow$ SEEDTREELEAVES(Mseed, Salt)
for $i \leftarrow 0$ **to** $t-1$ **do**
  $(\mathtt{Seed}_{\boldsymbol{u}'}, \mathtt{Seed}_{\boldsymbol{e}'}) \leftarrow \mathrm{CSPRNG}\left(\mathtt{Seed}[i], \{0,1\}^\lambda \times \{0,1\}^\lambda\right)$
  $\boldsymbol{\zeta}' \leftarrow \mathrm{CSPRNG}\left(\mathtt{Seed}_{\boldsymbol{e}'}, \mathbb{F}_z^m\right)$ | $\boldsymbol{\eta}_i' \leftarrow \mathrm{CSPRNG}\left(\mathtt{Seed}_{\boldsymbol{e}'}, \mathbb{F}_z^n\right)$
  $\boldsymbol{\delta}_i \leftarrow \boldsymbol{\zeta} - \boldsymbol{\zeta}'$
  $\boldsymbol{\eta}_i' \leftarrow \boldsymbol{\zeta}' \mathrm{M}_G$
  $\boldsymbol{\sigma}_i \leftarrow \boldsymbol{\eta} - \boldsymbol{\eta}_i'$
  for $j \leftarrow 0$ **to** $n-1$ **do**
    $\boldsymbol{v}[j] \leftarrow g^{\boldsymbol{\sigma}_i[j]}$
  end
  $\mathbf{u}_i' \leftarrow \mathrm{CSPRNG}\left(\mathtt{Seed}_{\boldsymbol{u}'}, \mathbb{F}_p^n\right)$
  $\mathbf{u} \leftarrow \mathbf{v} \star \mathbf{u}_i'$ // $\star$ is component-wise product
  $\tilde{\mathbf{s}} \leftarrow \mathbf{u}\mathrm{H}^\top$
  $\mathsf{cmt}_0[i] \leftarrow$ HASH($\tilde{\mathbf{s}}||\boldsymbol{\delta}_i||\mathtt{Salt}||i$) | $\mathsf{cmt}_0[i] \leftarrow$ HASH($\tilde{\mathbf{s}}||\boldsymbol{\sigma}_i||\mathtt{Salt}||i$)
end
$\mathsf{d}_0 \leftarrow$ MERKLEROOT($\mathsf{cmt}_0[0], \ldots, \mathsf{cmt}_0[t-1]$)
// First challenge vector extraction
$\mathsf{d}_m \leftarrow$ HASH(m)
$\mathsf{d}_\beta \leftarrow$ HASH($\mathsf{d}_m||\mathsf{d}_0||\mathtt{Salt}$)
$\mathsf{beta} \leftarrow \mathrm{CSPRNG}\left(\mathsf{d}_\beta, (\mathbb{F}_p^*)^t\right)$
// Computation of first round of responses
for $i \leftarrow 0$ **to** $t-1$ **do**
  for $j \leftarrow 0$ **to** $n-1$ **do**
    $\mathbf{e}'[j] \leftarrow g^{\boldsymbol{\eta}_i'[j]}$
  end
  $\mathbf{y}_i \leftarrow \mathbf{u}_i' + \mathsf{beta}[i]\mathbf{e}_i'$
end

Verifica della firma

Anche la primitiva di verifica segue la maggioranza delle operazioni già viste. Le modifiche fondamentali sono conseguenze della rimozione del commitment. Non viene rieseguito l'hash di: $\mathtt{Seed}[i]$, Salt e l'indice di round i. Non si recuperano i valori dal vettore di risposta per i $t-w$ round con $b[i]=0$, non avendo più il commitment $\mathtt{cmt}_1$. In aggiunta si ricalcola d_0', evitando d_1' e d_{01}'. Il controllo finale della verifica avviene sui digest d_0' e d_b', i quali devono essere entrambi uguali per garantire una corretta autenticazione della firma.

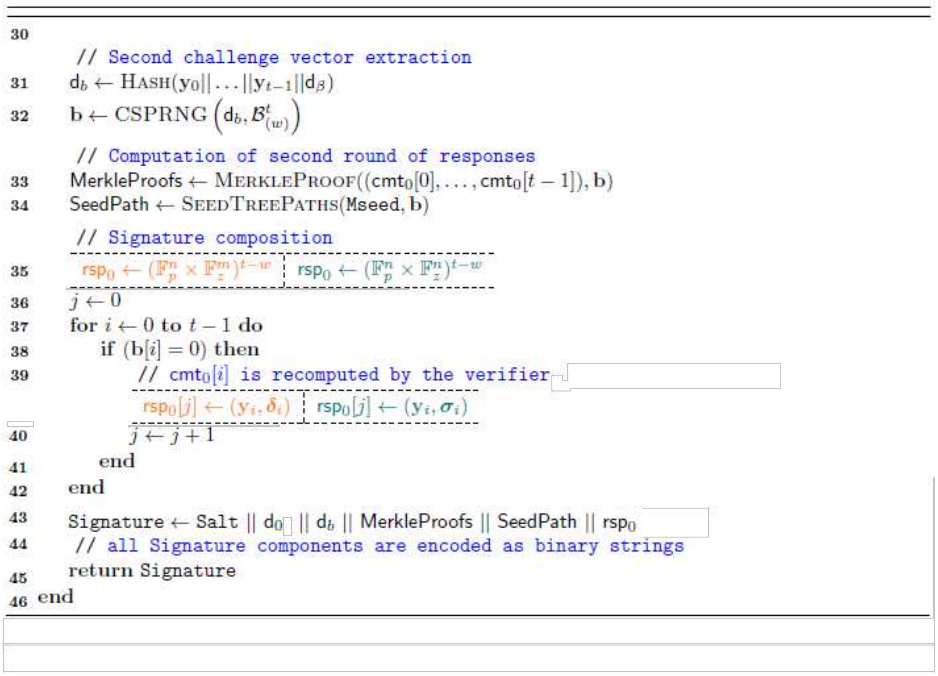

Figura 3.5: Primitiva Sign di CROSS SC

3.3 Implementazione Python

Dopo aver proposto la modifica dello schema, è fondamentale procedere con la sua realizzazione. All'inizio è stato necessario associare un'implementazione Python di riferimento ad una già esistente in linguaggio C. Questa nuova versione è stata progettata per riflettere la struttura della controparte, migliorandone così la fruibilità e la comprensibilità dal punto di vista tecnico.

I risultati e le componenti di interesse sono approfonditi per consentire l'analisi delle caratteristiche e per verificare la coerenza dell'implementazione rispetto alle logiche concepite. Basandosi sulla teoria esaminata nella sezione precedente, relativa al protocollo di identificazione $q2$ fino al raggiungimento dello schema di firma con sicurezza EUF-CMA, è stata sviluppata una versione in Python tramite Jupyter Notebook. L'implementazione è relativa sia alla versione classica che a quella con un solo commitment.

Dato che le differenze tra le due varianti sono ridotte, verranno prima esaminate le funzioni comuni, per poi concentrarsi sulle modifiche e le differenze specifiche.

3.3.1 Packages e Struttura

Per lo sviluppo è stato impiegato l'IDE Visual Studio Code, nel quale è stato necessario installare delle estensioni. In primo luogo sono state aggiunte le versioni aggiornate di Python, SageMath e Jupyter. Successivamente, dal terminale di VSCODE è stato possibile installare le estensioni di Python, Jupiter e quelle relative ai pacchetti crittografici (come pycryptodome).

Algorithm 3: CROSS-VERIFY(pub, Msg, Signature)

Data: λ: security parameter,
$g \in \mathbb{F}_p^*$: generator of a subgroup $\mathbb{E}$ of $\mathbb{F}_p^*$ with cardinality z
$\mathbb{E}^n$: restricted subgroup
M_G: $m \times n$ matrix of $\mathbb{F}_z$ elements, employed to generate vectors $\boldsymbol{\eta} \in G \subset \mathbb{E}^n$
t: number of iterations of the ZKID protocol
$\mathcal{B}_w^t$: set of all binary strings with length w and Hamming weight t

Input: pub: $(\mathsf{Seed}_{\mathsf{pk}}, \mathrm{s})$ public key: $\mathsf{Seed}_{\mathsf{pk}}$ is a λ bit seed to derive the non-systematic portion of a random parity-check matrix $\mathbf{H}$ and the $m \times n$ matrix of $\mathbb{F}_z$ elements M_G
Msg: message to verify the signature on; $\mathtt{Msg} \in \{0,1\}^*$
Signature: signature obtained encoding as binary the tuple $(\mathsf{Salt}, \mathsf{d}_{01}, \mathsf{d}_b, \mathsf{MerkleProofs}, \mathsf{SeedPath}, \mathsf{rsp}_0)$

Output: a Boolean value, {True, False}, indicating if the signature is verified or not

1 **Begin**
// Key material expansion
2 $(\mathsf{SeedV}, \mathsf{SeedW}) \leftarrow \mathrm{CSPRNG}\left(\mathsf{Seed}_{\mathsf{pk}}, \{0,1\}^\lambda \times \{0,1\}^\lambda\right)$; $\mathbf{V} \leftarrow \mathrm{CSPRNG}\left(\mathsf{SeedV}, \mathbb{F}_p^{(n-k)\times k}\right)$ | $\mathbf{V} \leftarrow \mathrm{CSPRNG}\left(\mathsf{Seed}_{\mathsf{pk}}, \mathbb{F}_p^{(n-k)\times k}\right)$
3 $\mathbf{H} \leftarrow [\mathbf{I}_{n-k} \mid \mathbf{V}]$
4 $\mathbf{W} \leftarrow \mathrm{CSPRNG}\left(\mathsf{SeedW}, \mathbb{F}_z^{m\times(n-m)}\right)$
5 $\mathrm{M}_G \leftarrow [\mathbf{I}_m \mid \mathbf{W}]$
// Challenge recomputation
6 $\mathsf{d}_m \leftarrow \mathrm{HASH}(m)$
7 $\mathsf{d}_\beta \leftarrow \mathrm{HASH}(\mathsf{d}_m \| \mathsf{d}_0 \| \mathsf{Salt})$
8 $\mathsf{beta} \leftarrow \mathrm{CSPRNG}\left(\mathsf{d}_\beta, (\mathbb{F}_p^*)^t\right)$
9 $\mathbf{b} \leftarrow \mathrm{CSPRNG}\left(\mathsf{d}_b, \mathcal{B}_w^t\right)$
10 $(\mathsf{Seed}[0], \ldots, \mathsf{Seed}[t-1]) \leftarrow \mathrm{REBUILDSEEDTREELEAVES}(\mathsf{SeedPath}, \mathbf{b}, \mathsf{Salt})$
11 $j \leftarrow 0$
12 **for** $i \leftarrow 0$ **to** $t-1$ **do**
13 **if** $(\mathbf{b}[i] = 1)$ **then**
14 $(\mathsf{Seed}_{\boldsymbol{u}'}, \mathsf{Seed}_{\boldsymbol{e}'}) \leftarrow \mathrm{CSPRNG}\left(\mathsf{Seed}[i], \{0,1\}^\lambda \times \{0,1\}^\lambda\right)$
15 $\boldsymbol{\zeta}' \leftarrow \mathrm{CSPRNG}\left(\mathsf{Seed}_{\boldsymbol{e}'}, \mathbb{F}_z^m\right)$; $\boldsymbol{\eta}' \leftarrow \mathrm{CSPRNG}\left(\mathsf{Seed}_{\boldsymbol{e}'}, \mathbb{F}_z^n\right)$
16 $\boldsymbol{\eta}' \leftarrow \boldsymbol{\zeta}' \mathrm{M}_G$
for $j \leftarrow 0$ **to** $n-1$ **do**
17 $\mathbf{e}'[j] \leftarrow g^{\boldsymbol{\eta}'[j]}$
end
18 $\mathbf{u}' \leftarrow \mathrm{CSPRNG}\left(\mathsf{Seed}_{\boldsymbol{u}'}, \mathbb{F}_p^n\right)$
19 $\mathbf{y}_i \leftarrow \mathbf{u}' + \mathsf{beta}[i]\mathbf{e}'$
20 **end**
21 **else**
22 $(\mathbf{y}_i, \boldsymbol{\delta}_i) \leftarrow \mathsf{rsp}_0[j]$; verify $\boldsymbol{\delta}_i \in G$; $\boldsymbol{\sigma}_i \leftarrow \boldsymbol{\delta}\mathrm{M}_G$ | $(\mathbf{y}_i, \boldsymbol{\sigma}_i) \leftarrow \mathsf{rsp}_0[j]$; verify $\boldsymbol{\sigma}_i \in G$
for $j \leftarrow 0$ **to** $n-1$ **do**
23 $\mathbf{v}[j] \leftarrow g^{\boldsymbol{\sigma}[j]}$
end
24 $\mathbf{y}' \leftarrow \mathbf{v} \star \mathbf{y}_i$
25 $\tilde{\mathbf{s}} \leftarrow \mathbf{y}'\mathbf{H}^\top - \mathsf{beta}[i]\mathbf{s}$
26 $\mathsf{cmt}_0[i] \leftarrow \mathrm{HASH}(\tilde{\mathbf{s}} \| \boldsymbol{\delta}_i \| \mathsf{Salt} \| i)$ | $\mathsf{cmt}_0[i] \leftarrow \mathrm{HASH}(\tilde{\mathbf{s}} \| \boldsymbol{\sigma}_i \| \mathsf{Salt} \| i)$
27 $j \leftarrow j + 1$
28 **end**
29 **end**

```
35
36    d'_0 ← RecomputeMerkleRoot(cmt_0, MerkleProofs, b)
37    d'_b ← Hash(y_0||...||y_{t-1})
38
39    if (d_0 = d'_0 and d_b = d'_b) then
          return True
40    end
41    return False
42 end
```

Figura 3.6: Primitiva Verify di CROSS SC

Per costruire un progetto valido e comprensibile c'è, inoltre, la necessità di avviare un notebook Sagemath a cui collegare un kernel Jupyter attraverso indirizzo IPv4. Una volta effettuata la connessione, bisogna caricare i file ed i moduli ausiliari (da includere nel codice principale .ipynb) sul server Jupyter. Infine, è sufficiente lanciare il codice per verificarne il corretto funzionamento.

La struttura del programma è la seguente:

- **CROSS e CROSS Single Commitment**: sono i file che fanno da riferimento, in maniera molto fedele agli schemi teorici, al codice principale;
- **parameters.json**: è una stringa di codice JSON contenente i parametri valorizzati in base agli studi condotti ed allo standard NIST;
- **CSPRNG**: è una cartella contenente tutti i file relativi alle funzioni CSPRNG impiegate, in questo caso le primitive crittografiche adoperate si basano su AES-CTR e SHAKE128, 256 (a seconda della categoria);
- **HASH**: è la cartella caratterizzante le funzioni hash crittografiche, nello specifico le versioni SHA3 256, 384 e 512 (dipendente nuovamente dalla categoria);
- **UTILS**: è la cartella più varia, in quanto contiene i riferimenti alle fuzioni per la costruzione degli alberi (Seed e Merkle Tree), nonchè le funzioni di utilità generica.

3.3.2 CROSS.ipynb

Per quanto riguarda il codice principale, in prima istanza vanno aggiunte le librerie per le inclusioni:

- **json**, per la lettura dei file, tra cui quello dei parametri;
- **os**, per includere comandi legati al sistema operativo, che permettono anche di implementare logiche TRNG;
- **pack & unpack**, per la gestione della firma in termini di scambio tra prover e verifier;
- **math**, in relazione ai campi finiti e le relative logiche di calcolo;
- **file**, per l'importazione di moduli e file esterni che aggiungano funzionalità al progetto.

Inizialmente si ha la sottosezione di configurazione, dove vengono recuperati i valori dei parametri dal file **parameters.json** e sono definite le categorie relative allo schema (problema R-SDP o R-SDPG, categoria delle variabili NIST 1 3 5 e livello di ottimizzazione FIRMA o TEMPO).

Le funzioni di utilità fondamentali non sono state incluse in un modulo esterno, in quanto sono immediate e vengono richiamate molteplici volte dentro le primitive; oltre al fatto che tale inclusione può essere sempre effettuata in un secondo momento. Questa procedura serve ad integrare le logiche TRNG e CSPRNG per i seed dello schema (nel dettaglio per la generazione e l'espansione); nonchè le operazioni di addizione, sottrazione e moltiplicazione componente per componente su campi finiti e sui sottogruppi moltiplicativi. Infine, sono anche considerate e relizzate funzioni che implementano il prodotto vettore-matrice ed il controllo dell'appartenenza dei valori ad un certo insieme. [1]

Primitiva: KeyGen

La primitiva di generazione delle chiavi integra le operazioni descritte nello schema teorico. Essa si concentra sulla generazione di $\texttt{Seed}_{sk}$ e $\texttt{Seed}_{pk}$, rispettivamente attraverso l'espansione tramite TRNG e CSPRNG. A seconda della tipologia di problema, tra R-SDP ed R-SDP(G), avviene il calcolo della componente non sistematica $\mathbf{V}$, della matrice $\mathbf{H}$. Successivamente l'enfasi si sposta sulla generazione di η, i quali valori sono usati come esponenti del generatore g, al fine di popolare il vettore di errore ristretto $\mathbf{e}$. Si passa infine per il calcolo della sindrome $\mathbf{s}$ mediante prodotto vettore-matrice, per concludere restituendo la coppia di chiavi `pri`, `pub` generata.

Primitiva: Sign

La primitiva di firma integra una prima parte relativa al ricalcolo delle strutture dati d'interesse, ossia ζ, $\mathbf{H}$ e $\mathbf{M}_G$ per R-SDP(G) e solamente η e $\mathbf{H}$ per R-SDP. Il calcolo dei commitment si basa sulla generazione del Master Seed `MSeed`, a partire dal TRNG; conseguentemente con la funzione `SeedTreeLeaves`, che riceve in input `MSeed` ed il `Salt`, genera la lista di t seed $\texttt{Seed}[0],\cdots,\texttt{Seed}[t-1]$. Segue lo sviluppo di un ciclo for, in entrambi i problemi, relativo alla creazione degli elementi i-esimi di η'; inoltre, con una sottrazione componente per componente si ottengono i valori di σ'. Questi ultimi saranno gli esponenti del generatore g, necessari a comporre il vettore temporaneo v, che occorre per realizzare la moltiplicazione componente per componente finalizzata ad ottenere u e calcolare la sindrome $\tilde{\mathbf{s}}$. Il ciclo si conclude con il calcolo degli elementi delle sequenze relative ai commitment $\texttt{cmt}_0$ e $\texttt{cmt}_1$, mediante hash di: sindrome, δ_i o σ_i a seconda del problema, `Salt` ed indice i di round.

Di seguito ci si concentra, con i dati ottenuti dal ciclo, sull'ottenimento dei digest per lo sviluppo delle challenge. Dai valori di cmt_0, usati come foglie, si realizza l'albero di Merkle, con radice $\texttt{d}_0$ ottenuta da hash consecutivi. $\texttt{d}_1$ è invece un digest ottenuto dall'hash dei valori di cmt_1 concatenati. Un successivo hash di $\texttt{d}_0$ e $\texttt{d}_1$ congunti consegna $\texttt{d}_{01}$. Infine, l'ultimo digest cruciale è quello relativo all'hash del messaggio da firmare $\texttt{d}_m$, al fine di ridurre il costo di comunicazione.

Il primo vettore di challenge **beta** è ottenuto a partire dal digest $\texttt{d}_\beta$ (come hash di $\texttt{d}_m$, $\texttt{d}_{01}$ e del `Salt`), a valori in $\mathbb{F}_q^*$.

[1] Tutto il codice relativo alle due diverse implementazioni è consultabile nell'Appendice.

La generazione della prima risposta avviene mediante ricostruzione del nuovo vettore di errore ristretto e', usando gli elementi di η' come esponenti del generatore g. Poi, mediante addizione componente per componente di $\mathbf{u}'$ e del prodotto scalare $\mathbf{beta}e'$ si ottengono le risposte in y.

Si passa all'estrazione della seconda challenge, a partire dal digest $\mathtt{d}_b$ (come hash delle risposte concatenato a $\mathtt{d}_\beta$) e il successivo calcolo del vettore di challenge $\mathbf{b}$ da un CSPRNG binario a peso di Hamming w fissato, che riceve in input $\mathtt{d}_b$ come seed.

Le seconde risposte $\mathtt{rsp}_0$ ed $\mathtt{rsp}_1$ sono relative ai $t - w$ round nei quali $\mathbf{b}$ assume valore zero. In primis ci si concentra sul calcolo della `MerkleProof`, necessaria al verifier per la ricostruzione di $\mathtt{d}_0$, mediante la funzione MerkleProof che riceve in input gli elementi di $\mathtt{cmt}_0$ ed il vettore $\mathbf{b}$. Attraverso `MSeed` e $\mathbf{b}$, nella funzione SeedTreePaths, viene generato il `SeedPath`, per far si che il verifier ricomponga la lista di seed. Infine, mediante ciclo for, si popolano le strutture $\mathtt{rsp}_0$ ed $\mathtt{rsp}_1$, fondamentali per comporre la firma `Signature` la quale sarà caratterizzata da: `Salt`, $\mathtt{d}_{01}$, d_b, `MerkleProof`, `SeedPath`, $\mathtt{rsp}_0$ ed $\mathtt{rsp}_1$.

Primitiva: Verify

La primitiva di verifica si concentra (ricevendo in input la coppia pubblica `pub`, il messaggio da verificare `Msg` e la firma da controllare `Signature`), a seconda del problema, sulla ricostruzione della componente non sistematica $\mathbf{V}$, della matrice $\mathbf{H}$. Nel caso R-SDP(G), invece, c'è necessità di ricalcolare $\mathbf{W}$ e $\mathbf{M}_G$. Il ricalcolo delle challenge si basa sulla ricostruzione dei digest $\mathtt{d}_m$, $\mathtt{d}_\beta$, `beta` e $\mathbf{b}$ della primitiva di generazione della firma.

Di seguito, mediante la funzione RebuildSeedTreeLeaves con input `SeedPath`, $\mathbf{b}$ e `Salt`, si ricostruisce la lista dei seed $Seed[0],\cdots,Seed[t-1]$.

Il ciclo for scandisce la casistica $b[i] = 1$ da quella $b[i] = 0$. Nel primo caso si ricostruiscono gli elementi di $\mathtt{cmt}_1$ e la coppia di seed $\mathtt{Seed}_{e'}$ ed $\mathtt{Seed}_{u'}$ (per ricalcolare il vettore di prime risposte $\mathbf{y}$). Il procedimento che ricalcola i valori di η' e li usa come esponenti di g è identico a quello della primitiva di firma. La differenza cruciale sta nel caso $b[i] = 0$, dove si fa riferimento alle sequenze $\mathtt{rsp}_0$ ed $\mathtt{rsp}_1$, recuperandone gli elementi e verificandone l'appartenenza al sottogruppo G.

Segue il ricalcolo della sindrome $\tilde{\mathbf{s}}$, attraverso una sottrazione componente per componente, la quale comprende l'uso del vettore di risposte $\mathbf{y}'$ (ottenuto dal prodotto componente per componente tra il vettore temporaneo $\mathbf{v}$ e le risposte già note $\mathbf{y}_i$), la matrice $\mathbf{H}$, il vettore di challenge $\mathbf{beta}$ e la sindrome pubblica $\mathbf{s}$. Successivamente, si ricalcolano gli elementi di $\mathtt{cmt}_0$, i quali sono recuperabili mediante hash di: sindrome $\tilde{\mathbf{s}}$, δ_i o σ_i a seconda del problema, `Salt` ed indice i; poi si recuperano da $\mathtt{rsp}_1$ gli elementi di $\mathtt{cmt}_1$.

Conseguentemente, è possibile riottenere il digest $\mathtt{d}'_0$ attraverso la funzione RecomputeMerkleRoot che riceve come input $\mathtt{cmt}_0$, `MerkleProof` e $\mathbf{b}$. Eseguendo un hash degli elementi di $\mathtt{cmt}_1$ si ottiene il digest $\mathtt{d}'_1$; inoltre, il successivo hash produrrà $\mathtt{d}'_{01}$. Componendo un ulteriore hash delle risposte $\mathbf{y}$ concatenate con $\mathtt{d}_\beta$ si ottiene $\mathtt{d}'_b$.

Infine, il controllo avviene su $\mathtt{d}_{01} = \mathtt{d}'_{01}$ e su $\mathtt{d}_b = \mathtt{d}'_b$. Se entrambe le verifiche sono positive, allora l'algoritmo restituisce `True` e dunque la firma è autentica. Nel caso opposto, invece, la firma è compromessa e non valida.

Differenze tra CROSS e CROSS_ONE

Nel caso della versione ad un singolo commitment sia noto, come già affermato, che molte operazioni si mantengono identiche, mentre altre subiscono delle modifiche che sono sostan-

ziali non tanto in termini di struttura ma di risultati ottenuti (sia velocità sia dimensione della firma). La primitiva di generazione della coppia di chiavi è identica e produce ancora la coppia `pub`, `pri`.

Nella primitiva di generazione della firma si evita l'hash che produce $\texttt{cmt}_1$, nonchè il calcolo del digest $\texttt{d}_1$ e di $\texttt{d}_{01}$. Il digest per la generazione di **beta** utilizza $\texttt{d}_0$ al posto di $\texttt{d}_{01}$. Infine, non si costruisce la struttura dati $\texttt{rsp}_1$ che è completamente omessa, garantendo velocizzazione dell'algoritmo e riduzione della dimensione della firma.

La primitiva di verifica ricalcola alla stessa maniera i digest d'interesse, evita il ricalcolo di $\texttt{cmt}_1$ mediante hash e recuperando i valori da $\texttt{rsp}_1$. Ovviamente non si compone mediante hash $\texttt{d}'_1$ e tanto meno $\texttt{d}'_{01}$. Il controllo finale avviene sempre su una coppia, ma questa volta risulta essere $\texttt{d}_0 = \texttt{d}'_0$ e $\texttt{d}_b = \texttt{d}'_b$, per verificare se la firma è valida o meno.

3.3.3 CSPRGN

Il modulo esterno relativo ai CSPRNG, integrati direttamente nel file principale mediante importazione, contiene tutte le funzioni relative alle primitive crittografiche di generazione di numeri casuali. La libreria a cui si fa riferimento è pycryptodome, che fornisce classi preconfigurati relative alle primitive. Vengono istanziate le funzioni relative ad AES-CTR ed alle versioni SHAKE128, 256. Di quest'ultime, sono state implementate versioni che generano sequenze di numeri in un sottogruppo moltiplicativo, ossia in assenza di zero, o alternative che producono stringhe casuali binaria a peso di Hamming fissato.

3.3.4 HASH

Nel modulo relativo alle funzioni HASH sono integrate le informazioni di pycryptodome relative a questa tipologia di primitiva. Nel progetto si è fatto riferimento solamente alle tre versioni di SHA3 256, 384 e 512, a seconda della categoria di riferimento del NIST.

3.3.5 UTILS

Risulta il modulo maggiormente popolato in quanto contiene i file relativi alla costruzione e maniopolazione degli alberi (Seed e Merkle), nonchè funzioni di utilità generiche secondarie.

merkle.py

Questo file contiene le funzioni per lo sviluppo dell'albero di Merkle, a partire dai t commitment $\texttt{cmt}[0],\cdots,\texttt{cmt}[t-1]$ come foglie. La funzione costruisce l'albero come una lista di liste, dove ogni lista rappresenta un livello della struttura ed il singolo elemento è un nodo. Inizialmente vengono lette le foglie a coppie e viene eseguito l'hash; il risultato dell'operazione viene salvato nella casella relativa al padre della lista del livello più alto. Il tutto viene ripercorso attraverso ciclo fintanto che ci sono coppie da considerare. Nel caso in cui il numero di elementi da considerare è dispari, l'ultimo di essi viene semplicemente copiato al livello successivo. Ciò che si ottiene al termine della funzione è una struttura dati albero completa come lista di liste ed un singolo elemento $\texttt{d}_0$ che rappresenta la radice dell'albero di Merkle. All'interno della funzione sono anche presenti micro estensioni che permettono, ad esempio, di contare i nodi ad ogni livello per verificare la coerenza strutturale.

La seconda funzione dìinteresse cruciale è quella che realizza la prove di Merkle per i nodi che il verifier deve ricostruire. Questa funzione elabora, sulla base del vettore di challenge b i nodi che non vengono passati durante l'esecuzione del protocollo; ragione per la quale, al fine di garantire uno schema corretto, c'è necessità di fornire all'entità che verifica, una struttura dati che contiene tutti questi nodi. La funzione analizza a coppie i nodi di ogni livello (a partire dalle foglie) e verifica, sulla base di una struttura dati temporanea artificiosa, che i due nodi non abbiano valore associato pari a 1 o che abbiano due valori opposti. Il livello delle foglie della nuova struttura viene valorizzato sulla base dei valori del vettore b all'opposto (il valore di 1 con 0 e viceversa); a partire da ciò, se due nodi in coppia hanno entrambi valore 1 allora il valore del padre sarà impostato ad 1. Dal momento in cui una coppia ha valore discordante, si imposta il valore del padre da 1 e si inserisce nella prova di Merkle il nodo associato al figlio con valore 1. Nel caso in cui una coppia manifesti entrambi i valori nulli, si passa alla coppia successiva. La radice non viene mai raggiunta, in quanto non avrebbe senso costruire una struttura nella quale viene salvato il dato da ricostruire in fase di verifica; dunque, esso viene impostato a 0. La struttura dati comprendente i nodi di interesse che non saranno ricomposti viene prodotta in output e sarà integrata alla `Signature`.

Infine, l'ultima funzione d'interesse realizza la ricostruzione dell'albero di Merkle, durante la fase di verifica. Essa fa riferimento ai commitment $\texttt{cmt}[i]$, per i round in cui $b = 1$, ed i restanti nodi di interesse dentro la struttura `MerkleProof`. Mediante interfacciamento a queste due macro componenti, la funzione ricostruisce (recuperando coerentemente i dati mancanti dalla Proof) l'albero di Merkle e garantisce l'ottenimento della radice di Merkle ricalcolata. Mediante logica basata su contatori, viene letto l'albero (e quindi i nodi) ed ogni volta che la coppia appoggiata ad una struttura dati provvisoria artificiosa a valori binari ha delle caratteristiche specifiche, allora viene ricalcolato l'albero. Se la coppia di nodi ha entrambi valori 1, allora vuol dire che il verifier li possiede già entrambi, ragione per la quale può ricalcolare il valore del padre facilmente mediante hash. Nel caso in cui uno dei due valori della coppia è 0, c'è la necessità di recuperare quello nodo dalla prova, per effettuare poi l'hash dei valori. Il tutto avviene iterativamente fino a ricomporre la radice dell'albero.

seed_tree.py

Questo file contiene invece tutte le funzioni relative all'albero di seed. In primis la funzione che costruisce i t seed a partire dal `MSeed` e dal `Salt` come radice. La logica è opposta a quella dell'albero di Merkle, in quanto si ragiona top-down, a differenza del precedente bottom-up. A partire dalla radice, si realizza una logica basata su CSPRNG che consegna un albero di seed come lsita di liste. Ogni nodo padre genera i figli mediante espansione CSPRNG di 2λ bit, separati poi nella metà sinistra e quella destra (a λ bit ciascuno). Il tutto viene realizzato fino a che non si ottiene una struttura che all'ultimo livello, ossia quello delle foglie, non ha t nodi popolati, che saranno poi utilizzati per le operazioni successive.

La funzione che costruisce la struttura `SeedPath` serve, analogamente al caso di Merkle, a comporre una sequenza per i dati che non possono essere ricalcolati dal verifier e dunque c'è la necessità di integrarli nella `Signature`. Si fa sempre riferimento al secondo vettore di challenge b che detta le condizioni per le quali un dato è da salvare o meno. Viene implementata una struttura dati ausiliaria che lavora in maniera complementare a quella realizzata per Merkle, leggendo l'albero nell'ordine opposto. Seguendo le medesime considerazioni sulle coppie di nodi, relazionando questa volta i nodi padre ai figli e non viceversa, si riesce a popolare questa lista di elementi.

L'ultima funzione, invece, è necessaria a ricalcolare i Seed di round, che occorrono al verifier per le operazioni di composizione della sindrome e delle risposte. Questa logica parte dalla struttura `SeedPath`, dal secondo vettore di challenge b e dal numero di round t. Alla base di ciò ripopola la struttura dati artificiale vista precedentemente e recupera, con logica complementare a quella usata in Merkle, i Seed relativi ai round del protocollo.

utils.py

Questo file contiene invece tutte le funzionalità per creare e definire matrici casuali, generatori per il problema, l'inizializzazione di alcune strutture dati; nonchè alcune operazioni nei campi finiti e la misurazione del peso delle strutture dati di interesse.

4

Profilazione del codice e risultati

Nel seguente capitolo l'attenzione sarà rivolta verso l'analisi del codice C relativo allo schema CROSS, caratterizzata dal confronto tra le versioni, lo studio dei parametri e la profilazione. Successivamente seguirà l'esposizione dei risultati ottenuti, al fine di mostrare i vantaggi e le eventuali criticità legate alla proposta di modifica.

4.1 Analisi parametrica

In primo luogo l'attenzione viene focalizzata sull'analisi dei parametri, ricercando nuove possibili istanze per la proposta di modifica. Bisogna inoltre tenere conto del fatto che, tra le due versioni di CROSS, potrebbero presentarsi alternative potenzialmente simili, come altre diametralmente opposte.

Questo processo di ricerca è reso possibile attraverso l'impiego di script Python, appositamente realizzati per esplorare il campo dei possibili valori. In questo modo, quindi, saranno evidenziate alcune istanze che risultano particolarmente interessanti (attraverso specifiche funzioni di ottimo).

L'analisi parametrica è di cruciale importanza per comprendere i concetti di ottimizzazione ed efficienza dello schema. I principali dati caratterizzanti dello schema vengono recuperati per completezza:

- **n**, la lunghezza del codice lineare;
- **k**, la dimensione del codice;
- **q**, lo spazio dei valori della prima challenge;
- **t**, il numero di round o il numero di esecuzioni parallele (volte a garantire sicurezza EUF-CMA);
- **w**, il peso fissato di Hamming per la seconda challenge, a valori binari;
- **m**, il parametro relativo al numero di generatori **a** del problema ristretto, nel sottogruppo G;
- λ, il parametro di sicurezza dello schema;
- **seed length**, il numero in byte delle principali strutture dello schema, tra cui seed e digest.

L'obiettivo è quello di rilevare, sulla base dei parametri fissati mediante ipotesi, valori differenti che offrano (se possibile) migliorie in termini di prestazioni. Il risultato atteso è quello

di offrire, con la proposta di modifica e con le analisi condotte, possibilità di miglioramento in termini di dimensione della firma e velocità d'esecuzione.

Per quanto riguarda un'ulteriore caratterizzazione dei parametri d'interesse, sono identificate tre discriminanti nel protocollo relative ai seguenti ambiti:

- **Versione del problema**, R-SDP o R-SDP(G);
- **Categoria delle variabili basate sul NIST**, 1, 3 o 5;
- **Ottimizzazione dello schema**, dimensione della firma (SIG_SIZE) o velocità di esecuzione (SPEED).

4.1.1 Script per i parametri

L'hardware sul quale sono state lanciate le versioni di CROSS è stato uniforme e caratterizzato dalle seguenti componenti d'interesse:

- Processore, **i7-10700K**;
- RAM, **32GB DDR4**;
- Memoria, **SSD 1TB**.

Lo script e le relative funzioni sono indirizzate verso l'analisi dello spazio dei possibili valori per determinati parametri d'interesse. Specificando, nella chiamata a terminale, il file desiderato di destinazione dell'output, ciò che si ottiene è un file .csv contenente una lista di possibili istanze.

Nel caso specifico, i parametri sui quali si focalizza l'ottimizzazione sono il numero di round t ed il peso di Hamming w della seconda challenge. La ragione legata alla scelta è basata sulla loro crucialità, ossia l'impatto che essi hanno sulle prestazioni dello schema, rispetto ai restanti. Una spiegazione intuitiva a questo fenomeno è identificabile dal fatto che tali grandezze definiscono il numero di esecuzioni parallele del protocollo da realizzare, nonchè la manipolazione dei dati scambiati nel protocollo.

Sulla base delle analisi teoriche condotte nei capitoli precedenti, è già noto come i migliori valori di w siano quelli relativamente vicini a t i quali, variando, influenzano l'interezza del protocollo.

Gli algoritmi di analisi, in termini pratici, si caratterizzano in due script Python:

- **CROSS_fixed_weight_finder_trange.py**, il quale ha l'obiettivo di scandagliare uno o più intervalli di t (tra un massimo ed un minimo definiti), al fine di rilevare una coppia (t, w) valida in termini di dimensione della firma e velocità di esecuzione;
- **CROSS_code_round_parameters_combiner.py**, il quale effettua un accoppiamento adeguato della coppia rilevata precedentemente, rispetto ai restanti parametri fissati.

Dunque, la coppia di script garantisce di rilevare, a seconda delle specifiche ed al termine dell'esecuzione, istanze valide che permettano un'esecuzione molto efficiente di CROSS.

CROSS_fixed_weight_finder_trange.py

Nel dettaglio, questo script è caratterizzato inizialmente dall'importazione delle librerie necessarie al funzionamento, tra cui le estensioni: per gestire file CSV, per l'esecuzione multiprocessore in parallelo e per la definizione del numero di core disponibili al processamento parallelo. Le funzioni dello script sono le seguenti:

- **find_number_of_random_chall_rounds(sec_parameter, q)**, ha l'obiettivo di calcolare il numero equivalente di round per raggiungere una certa sicurezza specificata rispetto agli attacchi d'interesse;
- **soundness_error_2(param_list)**, effettua i calcoli relativi all'errore di soundness (ampiamente discussi nella sezione teorica);
- **__main__**, il quale verifica che il livello di sicurezza fornito è valido; successivamente, fornendo il file .CSV di destinazione assieme alla lista di parametri da ottimizzare (comprensivi dell'intervallo di t), viene richiamata la funzione **find_fixed_weight_options**;
- **find_fixed_weight_options**, è il cuore del processo di ottimizzazione, il quale itera sull'intervallo t alla ricerca dei parametri ottimali (passando per la funzione **soundness_error_2**); inoltre, nei suoi calcoli, tiene conto dei valori di peso e dell'uso degli alberi di Merkle/Seed.

CROSS_code_round_parameters_combiner.py

Il secondo script è caratterizzato dall'analisi, in termini di dimensione delle chiavi e della firma, del protocollo CROSS, in relazione alle versioni del problema.

Inizialmente, vi è l'importazione delle librerie matematiche e di lettura dei file CSV. In seguito, le funzioni cruciali risultano essere:

- **worst_case_tree_nodes(num_rounds_t, seeds_to_hide)**, che calcola il numero massimo di nodi per un albero di Merkle, considerando una sfida a peso costante (quindi con determinati seed nascosti);
- **bitpacked_vector_size_B(vect_len, max_to_repr)**, che calcola la dimensione in byte di un vettore contenente elementi d'interesse;
- **public_key_size_B(n, k, q, prng_seed_size)**, calcola la dimensione della chiave pubblica in byte;
- **private_key_size_B(n, k, q, prng_seed_size)**, calcola la dimensione della chiave privata in byte;
- **signature_size_B(q, z, n, t, m, seeds_to_hide, prng_seed_size_b, hash_digest_size_b)**, calcola la dimensione della firma digitale in byte;
- **__main__**, come funzione che legge i parametri d'interesse dai due file di riferimento (l'output del precedente script ed un altro file preimpostato). Successivamente, vengono calcolate le dimensioni relative alle chiavi ed alle firme, per tutte le combinazioni di parametri. Infine, vengono memorizzati su un nuovo file di output i valori adeguati risultanti l'esecuzione.

4.1.2 Valori Parametrici per CROSS

Di seguito sono mostrati i risultati relativi all'esecuzione dei due script, sulla base di un set di parametri predefinito a seconda della categoria del NIST. Nello specifico, il set da ottimizzare è il seguente:

Parametri Base		
CAT_1	**CAT_3**	**CAT_5**
$[\lambda, q, n, z, m, t\_low, t\_high]$	$[\lambda, q, n, z, m, t\_low, t\_high]$	$[\lambda, q, n, z, m, t\_low, t\_high]$
$[128, 509, 42, 127, 24, 230, 1000]$	$[256, 509, 63, 127, 36, 230, 1000]$	$[512, 509, 87, 127, 48, 230, 1000]$

Tabella 4.1: Set di parametri degli script

I risultati derivanti dall'esecuzione del primo script, sulla base delle tre categorie del NIST, offrono i seguenti valori, i quali sono stati utilizzati per studiare l'andamento della versione originale di CROSS (Tab. 4.2):

R-SDP		
$q = 127$	$z = 7$	$g = 2$
CAT_1	**CAT_3**	**CAT_5**
seed_len = 16	seed_len = 24	seed_len = 32
$n = 127$	$n = 187$	$n = 251$
$k = 76$	$k = 111$	$k = 150$
SIG_SIZE		
$t = 871$	$t = 1024$	$t = 1024$
$w = 848$	$w = 987$	$w = 968$
SPEED		
$t = 256$	$t = 256$	$t = 512$
$w = 216$	$w = 160$	$w = 432$

R-SDP(G)		
$q = 509$	$z = 127$	$g = 16$
CAT_1	**CAT_3**	**CAT_5**
seed_len = 16	seed_len = 24	seed_len = 32
$n = 42$	$n = 63$	$n = 87$
$k = 23$	$k = 35$	$k = 47$
$m = 24$	$m = 36$	$m = 48$
SIG_SIZE		
$t = 871$	$t = 949$	$t = 949$
$w = 850$	$w = 914$	$w = 897$
SPEED		
$t = 243$	$t = 255$	$t = 356$
$w = 206$	$w = 176$	$w = 257$

Tabella 4.2: Parametri attuali di CROSS

4.1.3 Valori Parametrici per CROSS SC

Dopo aver focalizzato l'attenzione sulla versione classica di CROSS, l'enfasi è stata posta sull'analisi e l'ottimizzazione dei parametri per CROSS Single Commitment. Dunque, è stato necessario intervenire per modificare le componenti dello script relative alla caratterizzazione della firma digitale (la quale manca del vettore $\texttt{rsp}_1$) e, conseguentemente, dello schema. Quest'ultimo non dovrà più generare il vettore di commitment $\texttt{cmt}_1$, ne tantomeno effettuare l'hash per ottenere il digest $\texttt{d}_1$.

Tenendo in considerazione queste modifiche, che interessano prevalentemente lo script **CROSS_code_round_parameters_combiner.py** (nella funzione **signature_size_B(q, z, n, t, m, seeds_to_hide, prng_seed_size_b, hash_digest_size_b)**), è valutare possibili nuovi set di parametri per CROSS Single Commitment.

La funzione che ha senso mostrare, in relazione alle differenze tra le versioni, è quella che concerne la firma digitale. Essa semplicemente integra e definisce tutte le componenti d'interesse; inoltre, pone l'attenzione sugli aspetti critici del protocollo, per i quali si ottengono istanze valide.

```
1  def signature_size_B(q,z,n,t,m,seeds_to_hide,prng_seed_size_b,
       hash_digest_size_b):
2      # salt + d_01 + d_b
3      cost = prng_seed_size_b/8 + 2*hash_digest_size_b/8
4
5      # Seed tree cost where t-w (b=1) seeds are revealed, w are
           not (b=0)
6      cost += (prng_seed_size_b * worst_case_tree_nodes(t,
           seeds_to_hide))/8
7      # Cost of revealing the missing w = seeds_to_hide commitment
           hashes
8      cost += (hash_digest_size_b * worst_case_tree_nodes(t,
           seeds_to_hide))/8 # Merkle tree non revealed
9      cost += (seeds_to_hide * hash_digest_size_b)/8 # random
           reveals leaves as raw hashes
10     if (m == 0): # RSDP case
11         cost += seeds_to_hide * (bitpacked_vector_size_B(n,q-1)+
               bitpacked_vector_size_B(n,z-1)) # size of actual
               reveals (y,sigma)
12     else :
13         cost += seeds_to_hide * (bitpacked_vector_size_B(n,q-1)+
               bitpacked_vector_size_B(m,z-1)) # size of actual
               reveals (y,sigma)
14     return cost
```

Listato 4.1: Codice relativo al calcolo della firma in CROSS CS

Per quanto riguarda i risultati che sono stati evidenziati rispetto alla versione classica (prendendo in considerazione, per semplicità, solamente il caso R-SDP(G)), le istanze più interessanti che sono state ottenute a seguito dell'esecuzione degli algoritmi, sono le seguenti:

R-SDP(G)		
$q = 509$	$z = 127$	$g = 16$
CAT_1		
seed_len = 16		
$n = 42$	$k = 23$	$m = 24$
SIG_SIZE		
$t = 1026$	$w = 1006$	$t - w = 20$
SPEED		
$t = 225$	$w = 186$	$t - w = 39$

Tabella 4.3: Parametri Alternativi per CROSS Single Commitment

Prima di passare alla profilazione vera e propria del codice, impiegando i parametri precedentemente identificati, è necessario discutere su come quest'ultimi possano influenzare gli aspetti di ottimizzazione sui quali si è focalizzata l'attenzione. Per quanto concerne il tempo d'esecuzione, si cerca una combinazione di valori t e w piccola, dove il peso di Hamming deve essere abbastanza vicino al numero di round; così facendo, la maggior parte delle esecuzioni saranno caratterizzate da $b = 1$ e la quantità di dati scambiati sarà ridotta. L'istanza alternativa a quella classica, per la versione di CROSS Single Commitment, ha valori che non si discostano molto dai precedenti per le ragioni descritte precedentemente. D'altra parte, se avessimo un t grande, il conseguente algoritmo risulterebbe pesante dal punto di vista computazionale; se si lavorasse invece con un w piccolo, si avrebbero molte esecuzioni con costo elevato nello scambio di informazione.

Nel contesto relativo alla riduzione della dimensione della firma, l'interesse si pone sulla differenza $t - w$, ma con una differente chiave di lettura. C'è bisogno che t sia abbastanza grande da garantire allo schema di elaborare efficientemente la firma e, allo stesso tempo, che tale differenza sia più piccola possibile entro un certo limite di sicurezza, per offrire una marcatura compatta e robusta.

Dunque, sulla base di quanto è stato osservato e proposto, sarà necessario confrontare le nuove istanze rilevate rispetto alla nuova versione, tenendo conto del fatto di valutare anche l'algoritmo a parità di parametri. In questo modo, sarà possibile confrontare e valutare se questo adattamento consegna migliorie relative agli aspetti d'interesse evidenziati, oppure non raggiunge un risultato sperato, a causa di una non conformità nelle scelte fatte (tenendo in considerazione che la modifica proposta non rivoluziona totalmente lo schema, ma riduce le operazioni ed i calcoli eseguiti). [1]

4.2 Profilazione degli schemi

L'ultimo argomento trattato concerne l'analisi delle prestazioni relative alle implementazioni di basso livello (codice C). Si farà riferimento al progetto realizzato in [10], il quale rappresenta una base solida sulla quale si è appoggiato lo studio condotto, nonchè il successivo

[1] Il codice completo relativo agli script di ottimizzazione è allegato in Appendice.

sviluppo in Python. Modificando le operazioni per introdurre la modifica, si ottengono due versioni che, se confrontate tra loro, risulta come abbiano una struttura molto simile ma vengano ridotte le operazioni. Un futuro sviluppo potrà invece riguardare la realizzazione delle versioni ottimizzate in linguaggio C, con l'impiego di logiche AVX AVX2. L'obiettivo è quello di valutare il comportamento e l'efficienza del codice relativo alla versione classica di CROSS e monitorare come le modifiche effettuate abbiano portato a migliorie o peggioramenti, per poi considerare le nuove istanze parametriche.

L'iter di sviluppo di CROSS (parallelo agli studi condotti in questo lavoro di tesi) ha portato, a partire dalla versione originale che è stata considerata, ad una versione aggiornata di CROSS a due commitment, nella quale sono state ridotte sensibilmente le chiamate ai CSPRNG e le cui prestazioni saranno discusse in seguito. Successivamente, gli studi hanno fatto emergere la possibilità di lavorare ad un'alternativa a singolo commitment al fine di valutare, al variare dei parametri, l'efficienza della nuova versione rispetto alle precedenti; avendo, dunque, a disposizione tre alternative da studiare.

Per la profilazione sono state prese in considerazione tutte le alternative di CROSS, concentrando l'interesse sul problema R-SDP(G), con categoria di variabili CAT_1, in entrambe le ottimizzazioni SIG_SIZE e SPEED.

4.2.1 Struttura del progetto C

Più nel dettaglio, la struttura del progetto si divide nelle seguenti directory:

- **Reference Implementation**, contenente tutti i file .c ed i relativi header .h, specifici dell'implementazione classica, anche nota come Proof of Concept;
- **Optimized Implementation**, contenente invece la versione ottimizzata con logica AVX, AVX2, al fine di garantire una versione ad elevate prestazioni (che non è stata presa in analisi per questo lavoro);
- **Additional Implementation**, contenente le logiche di compilazione delle varie versioni del codice, divise per Known Answer Tests e Benchmarking;
- **Supporting Implementation**, con i dati relativi alle proprietà intellettuali;
- **CMake**, sottocartella dell'Additional, contenente tutte le informazioni di compilazione, di debug e di profilazione.

Le componenti di maggior interesse sono quelle relative all'implementazione di riferimento, congiuntamente alle specifiche addizionali per testare le versioni.

4.2.2 Profilazione Gprof

Tale struttura di progetto garantisce una configurazione basata su determinate specifiche. Ogni volta che, attraverso i comandi descritti nel README si svolge la compilazione dei KAT o del Benchmarking, vengono prodotti tutti gli eseguibili relativi alle possibili combinazioni di:

- **R-SDP** o **R-SDP(G)**;
- **CAT_1**, **CAT_3** e **CAT_5**;
- **SIG_SPEED** o **SPEED**.

Ne derivano, dunque, dodici eseguibili .ELF sui quali è possibile lavorare. Oltre a ciò, è necessario sottolineare una caratterizzazione del file CMakeLists, nel quale è stato possibile modificare una riga di codice per poter definire: il grado di ottimizzazione del codice basato sulla macchina (01 grado minimo 02 grado intermedio e 03 grado massimo), l'inserimento di dati di debug negli eseguibili (g0 per non includere, g per includerli) e l'impostazione per creare informazioni per la profilazione del codice mediante il software Gprof (attraverso il comando pg).

```
1 set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS}-Wall-pedantic-Wuninitialized-
      march=native-O3-g0-pg")
```

Listato 4.2: Codice C per la configurazione delle specifiche di compilazione

In seguito alla compilazione, è sufficiente eseguire i singoli file, per generare un output composto da:

- **File d'esecuzione**, relativi ai dati forniti in input allo schema ed i dati prodotti dall'esecuzione effettiva, come la firma digitale ed i seed;
- **gmon.out**, contenente i dati di profilazione in formato acerbo.

Mediante un comando specifico è possibile convertire i file di profilazione gmon in un formato .txt di facile interpretazione e lettura.

```
1 gprof nome_eseguibile gmon.out > report.txt
```

Listato 4.3: Codice Shell Linux per la conversione dei file di profilazione

In questa maniera è possibile analizzare i risultati ottenuti in maniera chiara. Da ciò è stato possibile realizzare grafici immediati per valutare quanto fatto.

Nel dettaglio della profilazione saranno investigate le misure ed i valori che, tenendo conto della rilevanza statistica, danno informazioni sul valore della modifica proposta. Si terrà conto principalmente di: dimensione della firma, dimensione delle chiavi ed il tempo d'esecuzione delle fasi dello schema.

4.2.3 Script per il Benchmarking

Mediante uno script realizzato nel progetto CROSS in linguaggio C, sarà possibile monitorare gli aspetti d'interesse sopracitati, utilizzando i cicli come unità di misura del tempo d'esecuzione (in valor medio e deviazione standard) ed i byte per le sequenze di dati d'interesse. Attraverso l'uso dei cicli si ottiene il primario vantaggio di trascurare, in parte, l'hardware sul quali si eseguono le analisi; difatto, nota la frequenza di clock del processore utilizzato, questa logica consente una sorta di normalizzazione della misura. Nel dettaglio, l'algoritmo considera il tempo d'esecuzione relativo alle primitive di: generazione delle chiavi, firmatura e verifica. Dunque si hanno informazioni complete sulla versione, basandosi su un numero medio di esecuzioni che dia rilevanza statistica.

4.3 Risultati e Confronti

Le analisi sono state condotte sulla versione R-SDP(G) con CAT_1 del NIST, relativamente ai valori dei parametri utilizzati, considerando le due ottimizzazioni basate sulla dimensione della firma SIG_SIZE e sulla velocità SPEED. Per garantire una significatività statistica dei risultati, è stata valutata una media di mille esecuzioni del medesimo algoritmo.

Prestazioni di CROSS

La versione originale di CROSS, basata su una logica a due commitment (con istanza SPEED $t = 243$, $w = 206$ e SIG_SIZE $t = 871$, $w = 850$), ha le seguenti caratteristiche e prestazioni:

Parametri (q, z, n, k, m)	Versione	Sign.Size (kB)	**KeyGen** MCycles	**Sign** MCycles	**Verify** MCycles
509, 127, 42, 23, 24	SPEED	8.665	0.03	3.08	2.11
	SIG_SIZE	7.625	0.03	11.04	7.81

Figura 4.1: Tabella relativa al tempo d'esecuzione di CROSS

Ciò che si aspetta di ottenere con la nuova versione, sulla base delle modifiche effettuate, è una riduzione importante relativa alla dimensione della firma e una che concerne il tempo d'esecuzione relativo alle primitive dell'algoritmo.

Prestazioni di CROSS Single Commitment

Mantenendo inizialmente gli stessi parametri della versione originale, i valori ottenuti dal benchmark della nuova versione sono i seguenti: Di conseguenza, il miglioramento è osser-

Parametri (q, z, n, k, m)	Versione	Sign.Size (kB)	**KeyGen** MCycles	**Sign** MCycles	**Verify** MCycles
509, 127, 42, 23, 24	SPEED	7.497	0.07	2.10	1.43
	SIG_SIZE	6.969	0.05	7.71	5.47

Figura 4.2: Tabella relativa alle prestazioni di CROSS SC a parametri invariati

vabile in entrambe le direzioni, in quanto, si ha una firma ridotta di circa 1.168kB per la versione SPEED ed una riduzione di 0.656kB per SIG_SIZE. Allo stesso modo, si ha un oggettiva miglioria anche relativamente al tempo d'esecuzione. C'è un minimo peggioramento riguardo alla generazione della firma, giustificabile dai piccoli valori in gioco che la rendono una variazione trascurabile; d'altra parte sia la primitiva di firma sia di verifica, in entrambe le configurazioni, vengono ridotte di molto. In SPEED, si passa da una fase di firma a 3.08 MCycles ad una a 2.10 MCycles, mentre la verifica decresce da 2.11 ad 1.43 MCycles. Per SIG_SIZE il miglioramento nella firma va da 11.04 a 7.71 MCycles e per la verifica da 7.81 a 5.47 MCycles.

Come si può intuire, laddove il tempo d'esecuzione è minore generalmente, le migliorie sono minori appunto perchè c'è l'avvicinamento verso un valore sempre più ottimale. Nella SIG_SIZE le riduzioni di tempo sono molto più sensibili rispetto a quelle della SPEED per questa ragione.

Osservazione Parametrica

Quindi, mantenendo fissi i parametri, si ottengono oggettive migliorie negli ambiti di interesse, che risultano anche come i principali colli di bottiglia dello schema (dimensione della firma e tempo d'esecuzione).

Successivamente, è stata testata la nuova proposta CROSS rispetto alle nuove istanze parametrice, rilevate attraverso gli script Python. Ciò che è stato identificato è come queste nuove alternative garantiscano migliorie negli aspetti d'interesse, ma includano anche peggioramenti importanti sul restante contesto; per esempio, la nuova istanza per SPEED offre una diminuzione del tempo d'esecuzione per le primitive dello schema, ma d'altra parte peggiora abbastanza la dimensione della firma. Con una logica simile, la versione SIG_SIZE, mediante i nuovi parametri, consegna una firma di dimensione ridotta ma con tempi d'esecuzione più lunghi.

Di conseguenza, è possibile concludere, come non valga la pena impiegare queste nuove configurazioni allo schema in quanto, mantenendo i medesimi della versione originale, si ottengono migliorie valide e sensibili che sono sufficienti per il nostro studio.

4.3.1 Confronti con lo stato dell'arte

Dunque, il primis ci si concentra sul confronto diretto tra le due versioni di CROSS, relativamente agli aspetti più interessanti. La dimensione della coppia di chiavi (privata e pubblica) si mantiene identica in entrambe, in quanto non vi sono modifiche che interessano tale fase del protocollo. D'altra parte, l'eliminazione del secondo commitment, con tutte le relative conseguenze, conduce ad una diminuzione della dimensione della firma (in media) del 8,6% per SIG_SIZE e del 13,5% per SPEED. Il tempo d'esecuzione complessivo, che dunque comprende le tre fasi di generazione chiavi, firma e verifica, subisce una riduzione del 29,9% per SIG_SIZE e del 31,03% per SPEED. Risultati che quindi mostrano, su una media di mille esecuzioni, un valore effettivo della proposta rispetto alla precedente.

Il confronto, sia sul tempo d'esecuzione delle singole fasi dell'algoritmo e sia nel complesso, mostrano un miglioramento netto nelle prestazioni di CROSS adottando la versione ad un commitment.

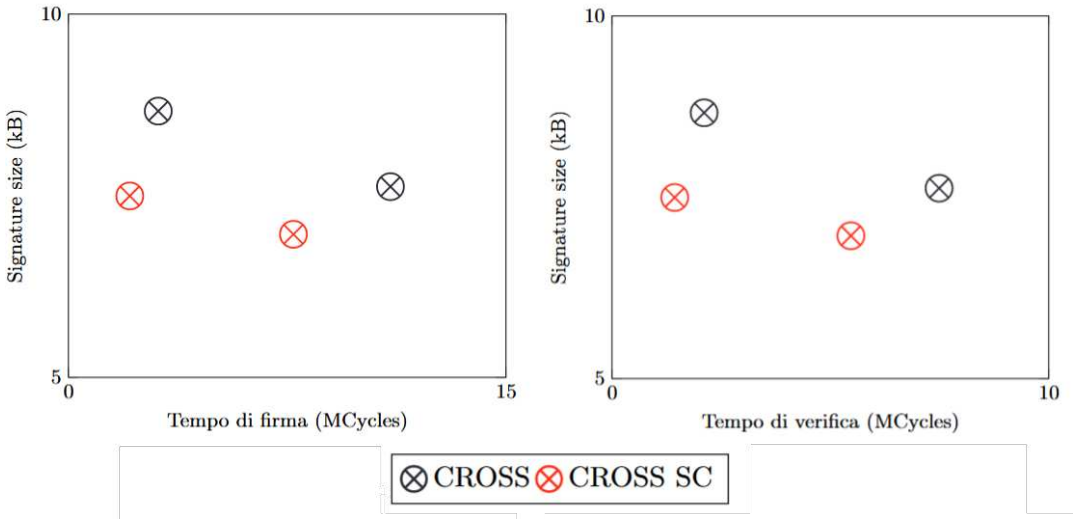

Figura 4.3: Confronto tra le versioni di CROSS basato su firma e tempo d'esecuzione delle fasi

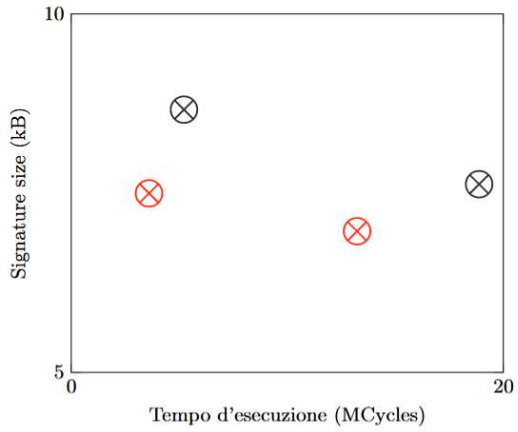

Figura 4.4: Confronto tra le versioni di CROSS basato su firma e tempo d'esecuzione complessivo

Confronto con schemi di firma basati su codici

Per quanto concerne, invece, il panorama complessivo degli schemi di firma digitale della competizione Post-Quantum, è possibile confrontare quanto prodotto rispetto agli altri schemi basati su codici lineari, così da rimarcare il valore di CROSS. Successivamente, è possibile confrontare CROSS con tutti i restanti schemi di firma, legati a primitive e logiche differenti, per offrire una valutazione completa di quanto realizzato.

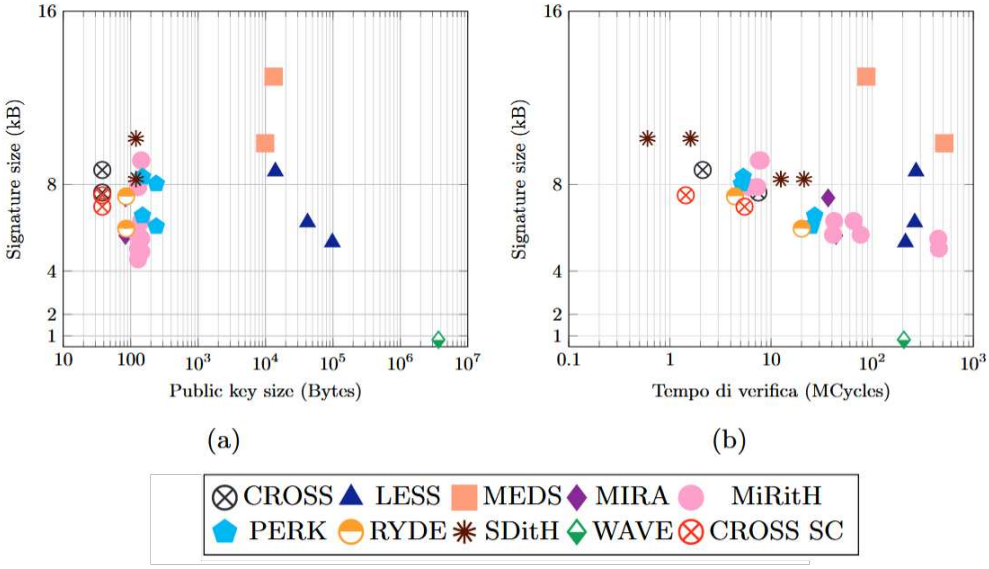

Figura 4.5: Confronto tra CROSS e lo stato dell'arte relativo agli schemi basati su codici

Relativamente al panorama degli schemi basati su codici lineari, CROSS e la sua versione SC si dimostrano schemi molto performanti dal punto di vista della dimensione della chiave pubblica, nonchè nel tempo d'esecuzione, osservando dei tempi di verifica (in entrambe le versioni) molto validi e migliori rispetto alla restante popolazione. Quindi, si può osservare come CROSS e CROSS SC possono essere considerati come due degli schemi più performanti e validi nel panorama dei codici lineari, grazie ad una firma con dimensione nella media, che è stata ulteriormente ridotta con l'ultima modifica effettuata, ed una velocità d'esecuzione tra le migliori.

Confronto con i restanti schemi di firma

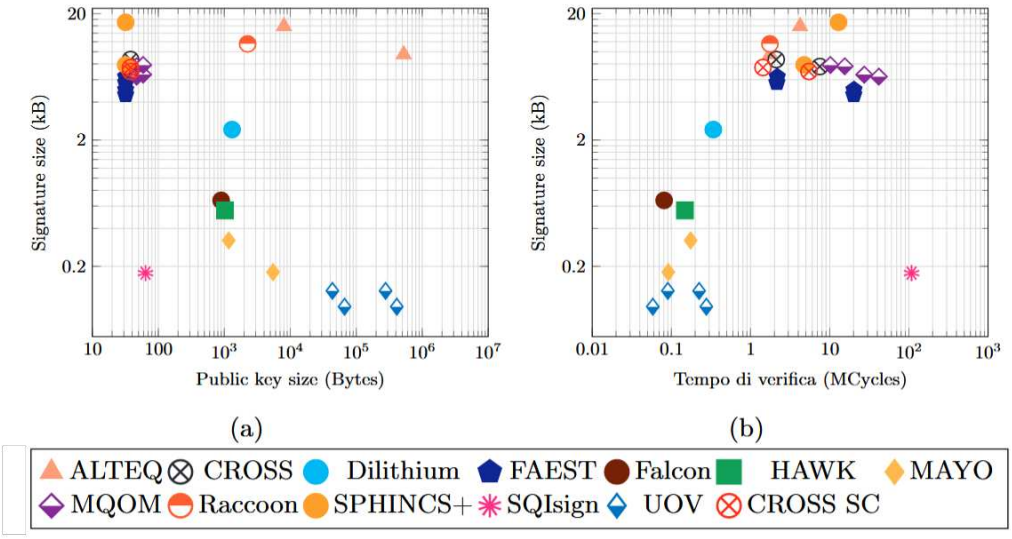

Figura 4.6: Confronto tra CROSS e lo stato dell'arte relativo al resto degli schemi di firma

Differente è il discorso rispetto al panorama complessivo degli schemi di firma. Sia noto come sono presenti schemi che hanno già vinto i primi round della competizione NIST, ossia Falcon, Dilithium e SPHINCS+. Si può notare come questi ultimi risultino vincenti in quanto garantiscono tempi di verifica eccellenti, legati ad una chiave pubblica piccola ed una firma che ha dimensione molto ridotta, ad eccezione di SPHINCS+, il quale ha vinto la competizione seppur offrendo firme di dimensione elevata e tempi di verifica peggiori della media. Questo discorso, che colloca CROSS nel contesto come uno schema con una chiave pubblica ottima, una firma di dimensione relativamente elevata ed un tempo d'esecuzione nella media, come uno schema meritevole di attenzione.

4.3.2 Analisi delle funzioni di CROSS

Un'ulteriore analisi che viene realizzata si lega allo studio delle singole funzioni comprese all'interno dello schema, al fine di definire i metodi più onerosi e che possono condurre a fenomeni di collo di bottiglia. Nel confronto saranno prese in considerazioni le due versioni di CROSS, tenendo conto delle due possibili ottimizzazioni.

Versione 1. CROSS

La versione originale di CROSS, sulla base delle analisi di profilazione mediante Gprof, mostra i seguenti contributi delle funzioni:

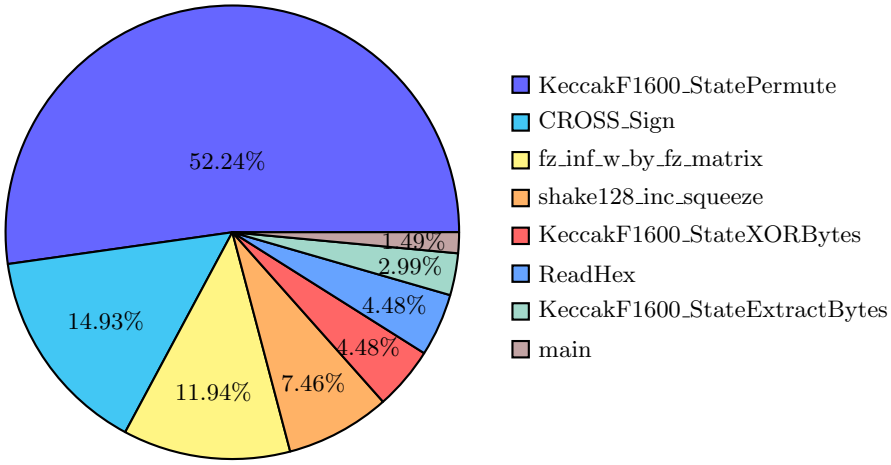

Figura 4.7: Contributo percentuale delle funzioni di CROSS SIG_SIZE

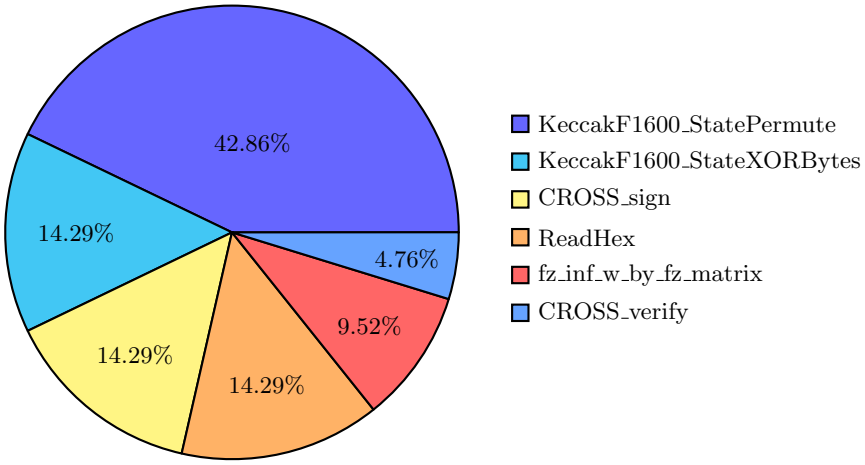

Figura 4.8: Contributo percentuale delle funzioni di CROSS SPEED

Si nota, quindi, come entrambe le ottimizzazioni coinvolgano più metodi, che offrono contributi differenti. Nel caso dell'ottimizzazione SIG_SIZE (Fig.4.7), le permutazioni di Keccak occupano oltre la metà del tempo complessivo di esecuzione, seguite dalla funzione di generazione della firma e dalle conversioni di matrice nei campi finiti. Nella versione SPEED (Fig.4.8) si osserva una riduzione del collo di bottiglia sulle permutazioni, caratterizzata da una dimunizione percentuale circa del 10%. Oltre a ciò, ad occupare la maggior parte del tempo risulta essere la funzione XOR di Keccak, seguita dalla generazione della firma e dalla lettura e traduzione dell'esadecimale.

Versione 2. CROSS SC

Di seguito si prende in considerazione come il nuovo schema, ottenuto mediante rimozione del secondo commitment, offra migliorie anche dal punto di vista della distribuzione del contributo delle funzioni.

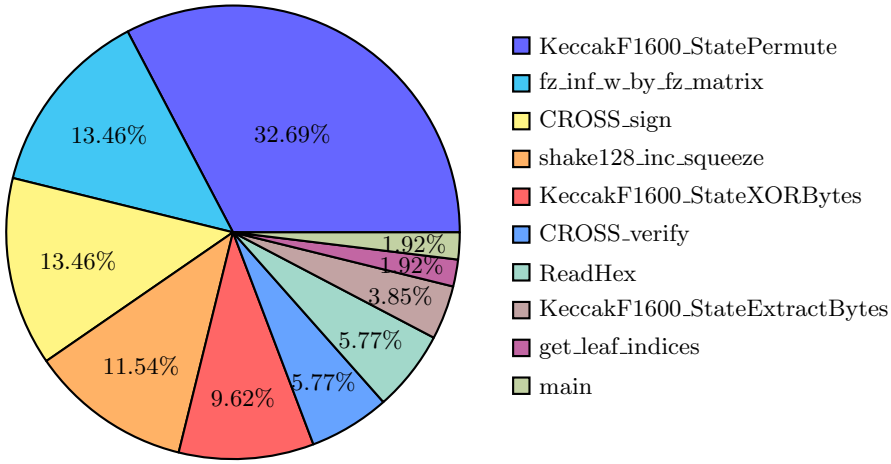

Figura 4.9: Contributo percentuale delle funzioni di CROSS SC SIG_SIZE

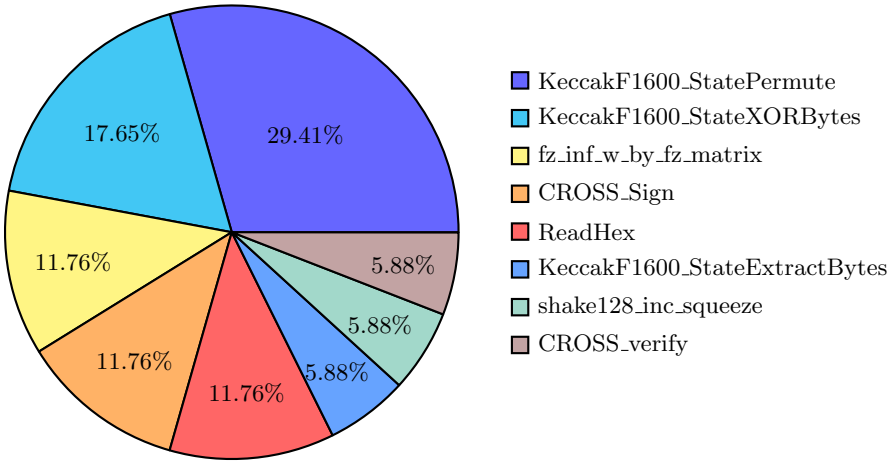

Figura 4.10: Contributo percentuale delle funzioni di CROSS SC SPEED

Si nota la modifica relativa alla distribuzione delle funzioni, dove il principale collo di bottiglia dello schema (le permutazioni) riducono il loro contributo percentuale del 19,55% per la versione SIG_SIZE e del 13,45% per SPEED. Ciò che si ottiene è una procedura con funzioni molto più omogenee, nonchè meno vulnerabile, in quanto non ci sono metodi che spiccano come maggiormente onerosi. Avere uno schema con contributi ben distribuiti garantisce molta più sicurezza di uno schema con un certo sbilanciamento da questo punto di vista.

Dai risultati ottenuti, quindi, si osserva come la modifica infici sensibilmente sulle prestazioni dell'algoritmo, limitando le operazioni complesse come le permutazioni. Quindi, come ci si aspettava, si ottiene una riduzione sia in percentuale sia in tempo effettivo di esecuzione per le funzioni interessate dal CSPRNG (Keccak); il quale, difatto, è tra le primitive più impattanti dello schema.

Conclusioni

In conclusione, è stato condotto uno studio approfondito e completo dello schema di firma digitale CROSS, comprensivo di una dettagliata analisi teorica e prestazionale. In aggiunta, sono state confrontate le proprietà di sicurezza del nuovo approccio rispetto all'originale, mantenendo un ottimo livello di validità. Nonostante ciò, l'obiettivo della ricerca è stato anche focalizzato verso il miglioramento di quanto precedentemente realizzato, riducendone al minimo le vulnerabilità e massimizzandone il potenziale.

Dunque, è stata consegnata una proposta di modifica per CROSS che garantisce sia le proprietà fondamentali di uno schema di firma digitale, sia le caratteristiche di robustezza e difficoltà necessarie a validarlo nell'ambito Post-Quantum. L'implementazione Python delle versioni di CROSS è stata ampiamente sviluppata, testata e confrontata con il riferimento, consegnando uno strumento utile per la comprensione teorica e necessario per la completezza del progetto.

L'analisi parametrica, condotta per rilevare possibili istanze alternative, ha evidenziato la significatività dei parametri classici di CROSS, rispetto alle nuove opzioni. Dunque, l'innovativo algoritmo CROSS SC, attraverso un'approfondita profilazione del codice, offre miglioramenti sensibili sia nella dimensione della firma sia nel tempo d'esecuzione complessivo; inoltre, posiziona il progetto CROSS in una posizione di vantaggio rispetto allo stato dell'arte relativo agli schemi basati su codici lineari. Parimenti, la distribuzione dei contributi nei metodi della procedura risulta maggiormente omogenea nel nuovo protocollo, il che contribuisce a minimizzare i colli di bottiglia. Quest'ultimi sono frequenti indicatori di debolezze o vulnerabilità ed il risultato ottenuto dà ancor più importanza e centralità all'alternativa proposta. Questo equilibrio favorisce robustezza, mitigando potenziali punti critici e sottolineando l'impiego di modus operandi orientato alla sicurezza, nella modifica di CROSS. I dati non solo confermano l'efficacia di quanto presentato, ma aprono anche ulteriori prospettive di ricerca e sviluppo.

Guardando al futuro, l'approccio di modifica realizzato potrebbe essere preso in considerazione come un modello per l'esplorazione di miglioramenti in altri schemi, basati tuttavia su logiche crittografiche differenti. Nonostante ciò, è importante osservare come l'introduzione di nuove versioni richieda sempre e comunque una valutazione attenta e completa legata alle debolezze, tenendo conto di tutti i possibili nuovi attacchi che possono presentarsi.

Pertanto, questa tesi ha contribuito ad ampliare lo spettro delle versioni di CROSS, oltre ad elevarne le prestazioni tecniche, aprendo nuovi scenari relativi all'evoluzione e all'otti-

mizzazione della firma digitale. Gli esiti di questa ricerca, quindi, promuovono con forza la necessità di sicurezza ed efficienza nelle nuove tecnologie dell'attuale contesto crittografico.

Appendice

CROSS.ipynb

Importazione Moduli

```
# LIB IMPORT

#If necessary %pip install pycryptodome
#%pip install tkinterdnd2
from sage.all import *
import json
import os
import hashlib
import random
import sys
import math
import pickle
import struct
import numpy as np
from CSPRNG import csprngs
from UTILS import utils
from UTILS import seed_tree
from UTILS import merkle
from HASH import hash
```

Listato 4.4: Codice relativo all'importazione delle librerie in CROSS

Configurazione e Parametri

```
# CONFIGURATION

# File
```

```
file\_path = 'parameters.json'
with open(file\_path, "r") as file:
    parameters = json.load(file)

global n, q, k, z, g, seed_byte_length, selected_cat,
    selected_opt, selected_rsdp

# Problem Data
selected_rsdp = "RSDPG"
selected_cat = "CAT_1"
selected_opt = "SPEED"

# Message to Sign
msg = b'customize'

# Parameters
n = parameters[selected_rsdp][selected_cat][selected_opt]["n"]
q = parameters[selected_rsdp][selected_cat][selected_opt]["q"]
k = parameters[selected_rsdp][selected_cat][selected_opt]["k"]
z = parameters[selected_rsdp][selected_cat][selected_opt]["z"]
g = parameters[selected_rsdp][selected_cat][selected_opt]["g"]
seed_byte_length = parameters[selected_rsdp][selected_cat][
    selected_opt]["seed_byte_len"]
t = parameters[selected_rsdp][selected_cat][selected_opt]["t"]
w = parameters[selected_rsdp][selected_cat][selected_opt]["w"]

print("CROSS␣configuration␣for␣execution:\n")
print("Problem␣:", selected_rsdp, "\n")
print("NIST␣Category␣:", selected_cat, "\n")
print("CROSS␣Optimization␣:", selected_opt, "\n")

if selected_rsdp=="RSDPG":
    m = parameters[selected_rsdp][selected_cat][selected_opt]["m"
        ]
    print("m␣:", m)

print("t␣:", t)
print("w␣:", w)
print("z␣:", z)
print("q␣:", q)
Fz = GF(z)
Fq = GF(q)
print("seed_byte_len:", seed_byte_length)
```

Listato 4.5: Codice relativo alla configurazione di CROSS e dei suoi parametri

Funzioni di utilità

```
# SEED UTILS

def expand_private_seed():
    seed_sk = os.urandom(seed_byte_length)
    return seed_sk

def get_salt():
    salt = os.urandom(2 * seed_byte_length)
    return salt

def expand_public_seed(seed):
    if selected_cat=="CAT_1":
        seed_l = csprngs.SHAKE_128(seed, 2*seed_byte_length)
        seed_a = seed_l[:seed_byte_length]
        seed_b = seed_l[seed_byte_length:]
    else:
        seed_l = csprngs.SHAKE_256(seed, 2*seed_byte_length)
        seed_a = seed_l[:seed_byte_length]
        seed_b = seed_l[seed_byte_length:]
    return seed_a, seed_b

def vector_matrix_product(vector, matrix, field_module):
    if len(vector) != matrix.nrows():
        raise ValueError("Vector and Matrix dimensions doesn't
            match")
    result = [0] * matrix.ncols()
    for i in range(matrix.ncols()):
        result[i] = sum((vector[j] * matrix[j, i]) % field_module
             for j in range(len(vector))) % field_module
    return result

def vector_matrix_product_star(vector, matrix, field_module):
    if len(vector) != matrix.nrows():
        raise ValueError("Vector and Matrix dimensions doesn't
            match")
    result = [0] * matrix.ncols()
    for i in range(matrix.ncols()):
        result[i] = sum((vector[j] * matrix[j, i]) % field_module
             for j in range(len(vector))) % field_module
    return result

def expand_seed(seed):
    seed_e, seed_pk = expand_public_seed(seed)
    if selected_rsdp=="RSDPG":
        seed_v, seed_w = expand_public_seed(seed_pk)
```

```
        V = utils.create_random_matrix_old(Fq, n-k, k, seed_v)
        W = utils.create_random_matrix_old(Fz, m, n-m, seed_w)
        Im = Matrix.identity(m)
        Mg = Im.augment(W, subdivide=True)
        if selected_cat == "CAT_1":
            zi = csprngs.SHAKE_128_vec(seed_e, m, z)
        else:
            zi = csprngs.SHAKE_256_vec(seed_e, m, z)
        Id = Matrix.identity(n-k)
        H = Id.augment(V, subdivide=True)
        return zi, H, Mg

    elif selected_rsdp=="RSDP":
        V = utils.create_random_matrix_old(Fq, n-k, k, seed_pk)
        if selected_cat == "CAT_1":
            eta = csprngs.SHAKE_128_vec(seed_e, n, z)
        else:
            eta = csprngs.SHAKE_256_vec(seed_e, n, z)
        Id = Matrix.identity(n-k)
        H = Id.augment(V, subdivide=True)
        return eta, H

    else:
        print("Problem not defined")
        sys.exit(1)

def component_wise_multiply(v1, v2, field_module):
    if len(v1) != len(v2):
        raise ValueError("Vectors length doesn't match!")
    result = [0] * len(v1)
    for i in range(len(v1)):
        result[i] = v1[i] * v2[i] % field_module
    return result

def component_wise_addition(v1, v2, field_module):
    if len(v1) != len(v2):
        raise ValueError("Vectors length doesn't match!")
    result = [0] * len(v1)
    for i in range(len(v1)):
        result[i] = (v1[i] + v2[i]) % field_module
    return result

def component_wise_subtraction(v1, v2, field_module):
    if len(v1) != len(v2):
        raise ValueError("Vectors lenght doesn't match")
    subtraction = [0] * len(v1)
    for i in range(len(v1)):
```

```
        subtraction[i] = (v1[i] - v2[i])
        if subtraction[i] < 0:
            subtraction[i] = subtraction[i] + field_module
        else:
            subtraction[i] = subtraction[i] % field_module

    return subtraction

def scalar_vector_product(scalar, vector, field_module):
    result = [(scalar * x) % field_module for x in vector]
    return result

# temporary seed tree leaves
def seed_tree_leaves(master_seed, salt):
    if selected_cat == "CAT_1":
        seed_list = [csprngs.SHAKE_128(master_seed + salt,
            seed_byte_length) for _ in range(t)]
    else:
        seed_list = [csprngs.SHAKE_256(master_seed + salt,
            seed_byte_length) for _ in range(t)]

    return seed_list

def check_vector(vector, field_module):
    is_in_field= all(element % field_module == element for
        element in vector)
    if is_in_field:
        return True
    else:
        return False
```

Listato 4.6: Codice relativo alle funzioni di utilità generiche per CROSS

Generazione delle chiavi

```
# KEY GENERATION ALGORITHM

def CROSS_keygen():
    seed_sk = expand_private_seed()
    seed_e, seed_pk= expand_public_seed(seed_sk)

    if selected_rsdp=="RSDPG":
        seed_v, seed_w = expand_public_seed(seed_pk)
        V = utils.create_random_matrix_old(Fq, n-k, k, seed_v)
        W = utils.create_random_matrix_old(Fz, m, n-m, seed_w)
        Im = Matrix.identity(m)
```

```
        Mg = Im.augment(W, subdivide=True)

        if selected_cat == "CAT_1":
            zi = csprngs.SHAKE_128_vec(seed_e, m, z)
        elif selected_cat == "CAT_3" or selected_cat == "CAT_5":
            zi = csprngs.SHAKE_256_vec(seed_e, m, z)
        else:
            raise ValueError("Category error!")
        zi_vec = vector(zi)
        eta = vector_matrix_product(zi_vec, Mg, z)


    elif selected_rsdp=="RSDP":
        V = utils.create_random_matrix_old(Fq, n-k, k, seed_pk)

        if selected_cat == "CAT_1":
            eta = csprngs.SHAKE_128_vec(seed_e, n, z)
        elif selected_cat == "CAT_3" or selected_cat == "CAT_5":
            eta = csprngs.SHAKE_256_vec(seed_e, n, z)
        else:
            raise ValueError("Category error!")

    else:
        print("Problem not defined")
        sys.exit(1)

    Id = Matrix.identity(n-k)
    H = Id.augment(V, subdivide=True)
    e = [1]*n

    for j in range(n):
        e[j] = (g**eta[j]) % q
        if e[j] == 0:
            raise ValueError("e[j] value is 0!")

    e_vec = vector(e)
    s = vector_matrix_product(e_vec, H.transpose(), q)

    return seed_sk, seed_pk, s
```

Listato 4.7: Codice relativo alla primitiva di generazione delle chiavi di CROSS

Generazione della firma

```
# SIGNATURE GENERATION ALGORITHM

```

```
def CROSS_sign(pri, message):

    v = [0]*n
    cmt0 = [0]*t
    cmt1 = [0]*t
    if selected_rsdp == "RSDPG":
        delta = [0*m]*t
    sigma = [0*n]*t
    eta_prime = [0*n]*t
    u_prime = [0*n]*t
    e_prime = [0]*n
    y = [0*n]*t

    # Key material expansion
    if selected_rsdp == "RSDPG":
        zi, H, Mg = expand_seed(pri)
        zi_vec = vector(zi)
        eta = zi_vec * Mg
    elif selected_rsdp == "RSDP":
        eta, H = expand_seed(pri)
    else:
        print("Problem not defined")
        sys.exit(1)

    # Computation of commitments
    M_seed = expand_private_seed()
    salt = get_salt()

    seed_tree_struct = seed_tree.SeedTree_x(M_seed, salt, t,
        selected_cat, seed_byte_length)
    seed_list = seed_tree_struct[-1][:t]

    for i in range(t):
        seed_u_prime, seed_e_prime = expand_public_seed(seed_list
            [i])
        if selected_rsdp == "RSDPG":
            if selected_cat == "CAT_1":
                zi_prime = csprngs.SHAKE_128_vec(seed_e_prime, m,
                    z)
            elif selected_cat == "CAT_3" or selected_cat == "CAT_5
                ":
                zi_prime = csprngs.SHAKE_256_vec(seed_e_prime, m,
                    z)
            else:
                raise ValueError("Category error!")

            zi_vec = vector(zi)
```

```
45              zi_prime_vec = vector(zi_prime)
46              delta[i] = component_wise_subtraction(zi_vec,
                    zi_prime_vec, z)
47              zi_prime_vec = vector(zi_prime)
48              eta_prime[i] = vector_matrix_product(zi_prime_vec, Mg,
                     z)
49
50          elif selected_rsdp == "RSDP":
51              if selected_cat == "CAT_1":
52                  eta_prime[i] = csprngs.SHAKE_128_vec(seed_e_prime,
                        n, z)
53              elif selected_cat == "CAT_3" or selected_cat == "CAT_5
                    ":
54                  eta_prime[i] = csprngs.SHAKE_256_vec(seed_e_prime,
                        n, z)
55              else:
56                  raise ValueError("Category error!")
57
58          else:
59              raise ValueError("Problem error!")
60
61          eta_vec = vector(eta)
62          eta_prime_vec = vector(eta_prime[i])
63          sigma[i] = component_wise_subtraction(eta_vec,
                eta_prime_vec, z)
64
65          for j in range(n):
66              v[j] = (g**sigma[i][j]) % q
67          v_vec = vector(v)
68
69          if selected_cat == "CAT_1":
70              u_prime[i] = csprngs.SHAKE_128_vec(seed_u_prime, n, q)
71          elif selected_cat == "CAT_3" or selected_cat == "CAT_5":
72              u_prime[i] = csprngs.SHAKE_256_vec(seed_u_prime, n, q)
73          else:
74              raise ValueError("Category error!")
75          u_prime_vec = vector(u_prime[i])
76          u = component_wise_multiply(v_vec, u_prime_vec, q)
77
78          u_vec = vector(u)
79
80          s_tilde = vector_matrix_product_star(u_vec, H.transpose()
                , q)
81          for x in range(len(s_tilde)):
82              if s_tilde[x] == 0:
83                  s_tilde[x] = 1
84
```

```
85          # Conversion to Hash
86          s_tilde_str = ''.join(map(str, s_tilde))
87          if selected_rsdp == "RSDPG":
88              delta_i_str = ''.join(map(str, delta[i]))
89          sigma_i_str = ''.join(map(str, sigma[i]))
90          salt_str = ''.join(map(str, salt))
91          i_str = str(i)
92          seed_list_i_str = ''.join(map(str, seed_list[i]))
93
94          # Commitments Generation
95          if selected_rsdp == "RSDPG":
96              data_to_hash_cmt0 = s_tilde_str + delta_i_str +
                    salt_str + i_str
97              if selected_cat == "CAT_1":
98                  cmt0[i] = hash.SHA3_256_HASH(data_to_hash_cmt0.
                        encode())
99              elif selected_cat == "CAT_3":
100                 cmt0[i] = hash.SHA3_384_HASH(data_to_hash_cmt0.
                        encode())
101             elif selected_cat == "CAT_5":
102                 cmt0[i] = hash.SHA3_512_HASH(data_to_hash_cmt0.
                        encode())
103             else:
104                 raise ValueError("Category␣error!")
105
106         elif selected_rsdp == "RSDP":
107             data_to_hash_cmt0 = s_tilde_str + sigma_i_str +
                    salt_str + i_str
108             if selected_cat == "CAT_1":
109                 cmt0[i] = hash.SHA3_256_HASH(data_to_hash_cmt0.
                        encode())
110             elif selected_cat == "CAT_3":
111                 cmt0[i] = hash.SHA3_384_HASH(data_to_hash_cmt0.
                        encode())
112             elif selected_cat == "CAT_5":
113                 cmt0[i] = hash.SHA3_512_HASH(data_to_hash_cmt0.
                        encode())
114             else:
115                 raise ValueError("Category␣error!")
116         else:
117             raise ValueError("Problem␣error!")
118
119         data_to_hash_cmt1 = seed_list_i_str + salt_str + i_str
120         if selected_cat == "CAT_1":
121             cmt1[i] = hash.SHA3_256_HASH(data_to_hash_cmt1.encode
                    ())
122         elif selected_cat == "CAT_3":
```

```
            cmt1[i] = hash.SHA3_384_HASH(data_to_hash_cmt1.encode
                ())
        elif selected_cat == "CAT_5":
            cmt1[i] = hash.SHA3_512_HASH(data_to_hash_cmt1.encode
                ())
        else:
            raise ValueError("Category error!")

    # Merkle Root and Commitment Hashing
    d0, tree = merkle.merkle_root_lnh(cmt0, selected_cat)
    tree = tree[::-1]

    # First Challenge vector extraction
    cmt1_str = ''.join(map(str, cmt1))

    if selected_cat == "CAT_1":
        d1 = hash.SHA3_256_HASH(cmt1_str.encode())
        d01 = hash.SHA3_256_HASH(d0 + d1)
        dm = hash.SHA3_256_HASH(message)
        dbeta = hash.SHA3_256_HASH(dm + d01 + salt)
        beta = csprngs.SHAKE_128_vec_star_(dbeta, t, q)
    elif selected_cat == "CAT_3":
        d1 = hash.SHA3_384_HASH(cmt1_str.encode())
        d01 = hash.SHA3_384_HASH(d0 + d1)
        dm = hash.SHA3_384_HASH(message)
        dbeta = hash.SHA3_384_HASH(dm + d01 + salt)
        beta = csprngs.SHAKE_256_vec_star_(dbeta, t, q)
    elif selected_cat == "CAT_5":
        d1 = hash.SHA3_512_HASH(cmt1_str.encode())
        d01 = hash.SHA3_512_HASH(d0 + d1)
        dm = hash.SHA3_512_HASH(message)
        dbeta = hash.SHA3_512_HASH(dm + d01 + salt)
        beta = csprngs.SHAKE_256_vec_star_(dbeta, t, q)
    else:
        raise ValueError("Category Error!")

    # Computation of First Responses
    j= 0

    for i in range(t):
        for j in range(n):
            e_prime[j] = (g**eta_prime[i][j]) % q
        e_prime_vec = vector(e_prime)
        temp_i = scalar_vector_product(beta[i], e_prime_vec, q)
        temp_vec = vector(temp_i)
        u_prime_vec = vector(u_prime[i])
        y[i] = component_wise_addition(u_prime_vec, temp_vec, q)
```

```
168
169     # Second Challenge vector extraction
170     y_str = ''.join(map(str, y))
171     dbeta_str = ''.join(map(str, dbeta))
172     input = y_str + dbeta_str
173
174     if selected_cat == "CAT_1":
175         db = hash.SHA3_256_HASH(input.encode())
176         b = csprngs.SHAKE_128_vec_weighted(db, t, 2, w)
177     elif selected_cat == "CAT_3":
178         db = hash.SHA3_384_HASH(input.encode())
179         b = csprngs.SHAKE_256_vec_weighted(db, t, 2, w)
180     elif selected_cat == "CAT_5":
181         db = hash.SHA3_512_HASH(input.encode())
182         b = csprngs.SHAKE_256_vec_weighted(db, t, 2, w)
183     else:
184         raise ValueError("Category␣error!")
185
186     # Computation of Second Responses
187     proof = merkle.merkle_proof(tree, cmt0, b)
188     seed_path = seed_tree.SeedPaths(seed_tree_struct, t, b)
189
190     # Signature Composition
191     if selected_rsdp == "RSDPG":
192         rsp0 = utils.initialize_couple_vector(t-w, n, m, q, z)
193     elif selected_rsdp == "RSDP":
194         rsp0 = utils.initialize_couple_vector(t-w, n, n, q, z)
195     else:
196         raise ValueError("Problem␣error!")
197
198     rsp1 = utils.initialize_vector(t-w, seed_byte_length, 2)
199     j = 0
200     for i in range(t):
201         if b[i] == 0:
202             if selected_rsdp == "RSDPG":
203                 rsp0[j] = (y[i], delta[i])
204             elif selected_rsdp == "RSDP":
205                 rsp0[j] = (y[i], sigma[i])
206             else:
207                 raise ValueError("Problem␣error!")
208             rsp1[j] = cmt1[i]
209             j += 1
210
211     bytes_proof = pickle.dumps(proof)
212     bytes_seed_path = pickle.dumps(seed_path)
213     bytes_rsp0 = pickle.dumps(rsp0)
214     bytes_rsp1 = pickle.dumps(rsp1)
```

```
215
216     proof_len = len(bytes_proof)
217     seed_path_len = len(bytes_seed_path)
218     rsp0_len = len(bytes_rsp0)
219     rsp1_len = len(bytes_rsp1)
220
221     tuple_signature = (salt, d01, db, bytes_proof,
            bytes_seed_path, bytes_rsp0, bytes_rsp1)
222
223     if selected_cat == "CAT_1":
224         format = f'{2*seed_byte_length}s 32s 32s {proof_len}s {
                seed_path_len}s {rsp0_len}s {rsp1_len}s'
225     elif selected_cat == "CAT_3":
226         format = f'{2*seed_byte_length}s 48s 48s {proof_len}s {
                seed_path_len}s {rsp0_len}s {rsp1_len}s'
227     elif selected_cat == "CAT_5":
228         format = f'{2*seed_byte_length}s 64s 64s {proof_len}s {
                seed_path_len}s {rsp0_len}s {rsp1_len}s'
229     else:
230         raise ValueError("Category error!")
231
232     signature = struct.pack(format, *tuple_signature)
233     return signature, format
```

Listato 4.8: Codice relativo alla primitiva di generazione della firma di CROSS

Verifica della firma

```
1  # SIGNATURE VERIFICATION ALGORITHM
2
3  def CROSS_verify(seed_pk, s, msg, signature, f):
4
5      # Key material expansion
6      if selected_rsdp == "RSDPG":
7          seed_v, seed_w = expand_public_seed(seed_pk)
8          V = utils.create_random_matrix_old(Fq, n-k, k, seed_v)
9          W = utils.create_random_matrix_old(Fz, m, n-m, seed_w)
10         Im = Matrix.identity(m)
11         Mg = Im.augment(W, subdivide=True)
12
13     elif selected_rsdp == "RSDP":
14         V = utils.create_random_matrix_old(Fq, n-k, k, seed_pk)
15     else:
16         raise ValueError("Problem selection wrong")
17
18     Id = Matrix.identity(n-k)
```

```
    H = Id.augment(V, subdivide=True)
    decoded = struct.unpack(f, signature)

    salt = decoded[0]
    d01 = decoded[1]
    db = decoded[2]
    bytes_proof = decoded[3]
    bytes_seed_path = decoded[4]
    bytes_rsp0= decoded[5]
    bytes_rsp1 = decoded[6]

    proof = pickle.loads(bytes_proof)
    seed_path = pickle.loads(bytes_seed_path)
    rsp0 = pickle.loads(bytes_rsp0)
    rsp1 = pickle.loads(bytes_rsp1)

    # Challenge recomputation
    if selected_cat == "CAT_1":
        dm = hash.SHA3_256_HASH(msg)
        dbeta = hash.SHA3_256_HASH(dm + d01 + salt)
        beta = csprngs.SHAKE_128_vec_star_(dbeta, t, q)
        b = csprngs.SHAKE_128_vec_weighted(db, t, 2, w)
    elif selected_cat == "CAT_3":
        dm = hash.SHA3_384_HASH(msg)
        dbeta = hash.SHA3_384_HASH(dm + d01 + salt)
        beta = csprngs.SHAKE_256_vec_star_(dbeta, t, q)
        b = csprngs.SHAKE_256_vec_weighted(db, t, 2, w)
    elif selected_cat == "CAT_5":
        dm = hash.SHA3_512_HASH(msg)
        dbeta = hash.SHA3_512_HASH(dm + d01 + salt)
        beta = csprngs.SHAKE_256_vec_star_(dbeta, t, q)
        b = csprngs.SHAKE_256_vec_weighted(db, t, 2, w)
    else:
        raise ValueError("Category␣Error!")

    seed_list = seed_tree.Recompute_Leaves(seed_path, b, salt,
        selected_cat, seed_byte_length)

    cmt1 = [0] * t
    cmt0 = [0] * t
    j = 0
    j_rsp = 0
    e_prime = [0]*n
    y = [0*n]*t
    if selected_rsdp == "RSDPG":
        delta = [0*m]*t
    sigma = [0*n]*t
```

```
    v = [0]*n
    eta_prime = [0*n]*t
    u_prime = [0*n]*t

    for i in range(t):
        if b[i] == 1:
            i_str = str(i)
            seed_list_i_str = ''.join(map(str, seed_list[i]))
            salt_str = ''.join(map(str, salt))

            data_to_encode1 = seed_list_i_str + salt_str + i_str

            if selected_cat == "CAT_1":
                cmt1[i] = hash.SHA3_256_HASH(data_to_encode1.
                    encode())
            elif selected_cat == "CAT_3":
                cmt1[i] = hash.SHA3_384_HASH(data_to_encode1.
                    encode())
            elif selected_cat == "CAT_5":
                cmt1[i] = hash.SHA3_512_HASH(data_to_encode1.
                    encode())
            else:
                raise ValueError("Category error!")

            seed_u_prime, seed_e_prime = expand_public_seed(
                seed_list[i])

            if selected_rsdp == "RSDPG":
                if selected_cat == "CAT_1":
                    zi_prime = csprngs.SHAKE_128_vec(seed_e_prime,
                        m, z)
                elif selected_cat == "CAT_3" or selected_cat == "
                    CAT_5":
                    zi_prime = csprngs.SHAKE_256_vec(seed_e_prime,
                        m, z)
                else:
                    raise ValueError("Category error!")

                zi_i_vec = vector(zi_prime)
                eta_prime[i] = vector_matrix_product(zi_i_vec, Mg,
                     z)

            elif selected_rsdp == "RSDP":
                if selected_cat == "CAT_1":
                    eta_prime[i] = csprngs.SHAKE_128_vec(
                        seed_e_prime, n, z)
```

```
102                 elif selected_cat == "CAT_3" or selected_cat == "
                        CAT_5":
103                     eta_prime[i] = csprngs.SHAKE_256_vec(
                            seed_e_prime, n, z)
104                 else:
105                     raise ValueError("Category␣error!")
106             else:
107                 continue
108 
109             for j in range(n):
110                 e_prime[j] = (g**eta_prime[i][j]) % q
111             if selected_cat == "CAT_1":
112                 u_prime[i] = csprngs.SHAKE_128_vec(seed_u_prime, n
                        , q)
113             elif selected_cat == "CAT_3" or selected_cat == "CAT_5
                    ":
114                 u_prime[i] = csprngs.SHAKE_256_vec(seed_u_prime, n
                        , q)
115             else:
116                 raise ValueError("Category␣error!")
117 
118             e_prime_vec = vector(e_prime)
119             temp_i = scalar_vector_product(beta[i], e_prime_vec, q
                    )
120             temp_vec = vector(temp_i)
121             u_prime_vec = vector(u_prime[i])
122             y[i] = component_wise_addition(u_prime_vec, temp_vec,
                    q) #
123 
124         elif b[i] == 0:
125             if selected_rsdp == "RSDPG":
126                 y[i] = rsp0[j_rsp][0]
127                 delta[i] = rsp0[j_rsp][1]
128                 boolean = check_vector(delta[i], z)
129                 assert boolean == True, "boolean␣no␣true"
130                 delta_vec = vector(delta[i])
131                 sigma[i] = vector_matrix_product(delta_vec, Mg, z)
132 
133                 for j in range(n):
134 
135                     v[j] = (g**sigma[i][j]) % q
136 
137                 v_vec = vector(v)
138                 y_i_vec = vector(y[i])
139                 y_prime = component_wise_multiply(v_vec, y_i_vec,
                        q)
140                 y_prime_vec = vector(y_prime)
```

```
                temp1 = vector_matrix_product_star(y_prime_vec, H.
                    transpose(), q)
                s_vec = vector(s)
                temp2 = scalar_vector_product(beta[i], s_vec, q)
                temp1_vec = vector(temp1)
                temp2_vec = vector(temp2)

                s_tilde = component_wise_subtraction(temp1_vec,
                    temp2_vec, q)
                for x in range(len(s_tilde)):
                    if s_tilde[x] == 0:
                        s_tilde[x] = 1

                s_tilde_str = ''.join(map(str, s_tilde))
                delta_i_str = ''.join(map(str, delta[i]))
                sigma_i_str = ''.join(map(str, sigma[i]))
                salt_str = ''.join(map(str, salt))
                i_str = str(i)

                data_to_encode0 = s_tilde_str + delta_i_str +
                    salt_str + i_str

                if selected_cat == "CAT_1":
                    cmt0[i] = hash.SHA3_256_HASH(data_to_encode0.
                        encode())
                elif selected_cat == "CAT_3":
                    cmt0[i] = hash.SHA3_384_HASH(data_to_encode0.
                        encode())
                elif selected_cat == "CAT_5":
                    cmt0[i] = hash.SHA3_512_HASH(data_to_encode0.
                        encode())
                else:
                    raise ValueError("Category error!")

            elif selected_rsdp == "RSDP":
                y[i] = rsp0[j_rsp][0]
                sigma[i] = rsp0[j_rsp][1]
                boolean = check_vector(sigma[i], z)
                assert boolean == True, "boolean no true"

                for j in range(n):
                    v[j] = (g**sigma[i][j]) % q

                v_vec = vector(v)
                y_i_vec = vector(y[i])
                y_prime = component_wise_multiply(v_vec, y_i_vec,
                    q)
```

```
181                 y_prime_vec = vector(y_prime)
182
183                 temp1 = vector_matrix_product_star(y_prime_vec, H.
                        transpose(), q)
184
185                 s_vec = vector(s)
186                 temp2 = scalar_vector_product(beta[i], s_vec, q)
187
188                 temp1_vec = vector(temp1)
189                 temp2_vec = vector(temp2)
190                 s_tilde = component_wise_subtraction(temp1_vec,
                        temp2_vec, q)
191                 for x in range(len(s_tilde)):
192                     if s_tilde[x] == 0:
193                         s_tilde[x] = 1
194
195                 s_tilde_str = ''.join(map(str, s_tilde))
196
197                 if selected_rsdp == "RSDPG":
198                     delta_i_str = ''.join(map(str, delta[i]))
199                 sigma_i_str = ''.join(map(str, sigma[i]))
200                 salt_str = ''.join(map(str, salt))
201                 i_str = str(i)
202
203                 if selected_cat == "CAT_1":
204                     data_to_encode0 = s_tilde_str + sigma_i_str +
                            salt_str + i_str
205                     cmt0[i] = hash.SHA3_256_HASH(data_to_encode0.
                            encode())
206                 elif selected_cat == "CAT_3":
207                     data_to_encode0 = s_tilde_str + sigma_i_str +
                            salt_str + i_str
208                     cmt0[i] = hash.SHA3_384_HASH(data_to_encode0.
                            encode())
209                 elif selected_cat == "CAT_5":
210                     data_to_encode0 = s_tilde_str + sigma_i_str +
                            salt_str + i_str
211                     cmt0[i] = hash.SHA3_512_HASH(data_to_encode0.
                            encode())
212                 else:
213                     raise ValueError("Category␣error!")
214             else:
215                 continue
216
217             cmt1[i] = rsp1[j_rsp]
218             j_rsp += 1
219         else:
```

```
            continue

    d0_prime = merkle.recompute_root(cmt0, proof, b, selected_cat
        , seed_byte_length)
    cmt1_str = ''.join(map(str, cmt1))

    y_str = ''.join(map(str, y))
    dbeta_str = ''.join(map(str, dbeta))
    input = y_str + dbeta_str

    if selected_cat == "CAT_1":
        d1_prime = hash.SHA3_256_HASH(cmt1_str.encode())
        d01_prime = hash.SHA3_256_HASH(d0_prime + d1_prime)
        db_prime = hash.SHA3_256_HASH(input.encode())
    elif selected_cat == "CAT_3":
        d1_prime = hash.SHA3_384_HASH(cmt1_str.encode())
        d01_prime = hash.SHA3_384_HASH(d0_prime + d1_prime)
        db_prime = hash.SHA3_384_HASH(input.encode())
    elif selected_cat == "CAT_5":
        d1_prime = hash.SHA3_512_HASH(cmt1_str.encode())
        d01_prime = hash.SHA3_512_HASH(d0_prime + d1_prime)
        db_prime = hash.SHA3_512_HASH(input.encode())
    else:
        raise ValueError("Category error!")

    if d01 == d01_prime and db == db_prime:
        return True
    else:
        if not d01 == d01_prime:
            print("d01 different from d01 prime")
        if not db == db_prime:
            print("db different from db prime")
        return False
```

Listato 4.9: Codice relativo alla primitiva di verifica della firma di CROSS

Main

```
#CROSS ONE Main

if __name__ == "__main__":
    seed_sk, seed_pk, s = CROSS_keygen()
    sign, f = CROSS_sign_one(seed_sk, msg)
    bool = CROSS_verify_one(seed_pk, s, msg, sign, f)
    print(bool)
```

Listato 4.10: Codice relativo al main di CROSS SC

CROSS_SC.ipynb

Importazione moduli

```
# LIB IMPORT

#If necessary %pip install pycryptodome
#%pip install tkinterdnd2
from sage.all import *
import json
import os
import hashlib
import random
import sys
import math
import pickle
import struct
import numpy as np
from CSPRNG import csprngs
from UTILS import utils
from UTILS import seed_tree
from UTILS import merkle
from HASH import hash
```

Listato 4.11: Codice relativo all'importazione delle librerie di CROSS SC

Configurazione e Parametri

```
# CONFIGURATION

# File
file_path = 'parameters.json'
with open(file_path, "r") as file:
    parameters = json.load(file)

global n, q, k, z, g, seed_byte_length, selected_cat,
    selected_opt, selected_rsdp

# Problem Data
selected_rsdp = "RSDP"
selected_cat = "CAT_1"
```

```
selected_opt = "SPEED"

# Message to Sign
msg = b'customize'

# Parameters
n = parameters[selected_rsdp][selected_cat][selected_opt]["n"]
q = parameters[selected_rsdp][selected_cat][selected_opt]["q"]
k = parameters[selected_rsdp][selected_cat][selected_opt]["k"]
z = parameters[selected_rsdp][selected_cat][selected_opt]["z"]
g = parameters[selected_rsdp][selected_cat][selected_opt]["g"]
seed_byte_length = parameters[selected_rsdp][selected_cat][
    selected_opt]["seed_byte_len"]
t = parameters[selected_rsdp][selected_cat][selected_opt]["t"]
w = parameters[selected_rsdp][selected_cat][selected_opt]["w"]

print("CROSS␣SINGLE␣COMMIT␣configuration␣for␣execution:\n")
print("Problem␣:", selected_rsdp, "\n")
print("NIST␣Category␣:", selected_cat, "\n")
print("CROSS␣Optimization␣:", selected_opt, "\n")

if selected_rsdp=="RSDPG":
    m = parameters[selected_rsdp][selected_cat][selected_opt]["m"
        ]
    print("m␣:", m)

print("t␣:", t)
print("w␣:", w)
print("z␣:", z)
print("q␣:", q)
Fz = GF(z)
Fq = GF(q)
print("seed_byte_len:", seed_byte_length)
```

Listato 4.12: Codice relativo alla configurazione di CROSS SC e dei parametri

Funzionalità di utilità

```
# SEED UTILS
def expand_private_seed():
    seed_sk = os.urandom(seed_byte_length)
    return seed_sk

def get_salt():
    salt = os.urandom(2 * seed_byte_length)
    return salt
```

```
9
10 def expand_public_seed(seed):
11     if selected_cat=="CAT_1":
12         seed_l = csprngs.SHAKE_128(seed, 2*seed_byte_length)
13         seed_a = seed_l[:seed_byte_length]
14         seed_b = seed_l[seed_byte_length:]
15     else:
16         seed_l = csprngs.SHAKE_256(seed, 2*seed_byte_length)
17         seed_a = seed_l[:seed_byte_length]
18         seed_b = seed_l[seed_byte_length:]
19     return seed_a, seed_b
20
21 def vector_matrix_product(vector, matrix, field_module):
22     if len(vector) != matrix.nrows():
23         raise ValueError("Vector␣and␣Matrix␣dimensions␣doesn't␣
               match")
24     result = [0] * matrix.ncols()
25     for i in range(matrix.ncols()):
26         result[i] = sum((vector[j] * matrix[j, i]) % field_module
               for j in range(len(vector))) % field_module
27     return result
28
29 def vector_matrix_product_star(vector, matrix, field_module):
30     if len(vector) != matrix.nrows():
31         raise ValueError("Vector␣and␣Matrix␣dimensions␣doesn't␣
               match")
32     result = [0] * matrix.ncols()
33     for i in range(matrix.ncols()):
34         result[i] = sum((vector[j] * matrix[j, i]) % field_module
               for j in range(len(vector))) % field_module
35
36     return result
37
38 def expand_seed(seed):
39     seed_e, seed_pk = expand_public_seed(seed)
40     if selected_rsdp=="RSDPG":
41         seed_v, seed_w = expand_public_seed(seed_pk)
42         V = utils.create_random_matrix_old(Fq, n-k, k, seed_v)
43         W = utils.create_random_matrix_old(Fz, m, n-m, seed_w)
44         Im = Matrix.identity(m)
45         Mg = Im.augment(W, subdivide=True)
46         if selected_cat == "CAT_1":
47             zi = csprngs.SHAKE_128_vec(seed_e, m, z)
48         else:
49             zi = csprngs.SHAKE_256_vec(seed_e, m, z)
50         Id = Matrix.identity(n-k)
51         H = Id.augment(V, subdivide=True)
```

```
        return zi, H, Mg

    elif selected_rsdp=="RSDP":
        V = utils.create_random_matrix_old(Fq, n-k, k, seed_pk)
        if selected_cat == "CAT_1":
            eta = csprngs.SHAKE_128_vec(seed_e, n, z)
        else:
            eta = csprngs.SHAKE_256_vec(seed_e, n, z)
        Id = Matrix.identity(n-k)
        H = Id.augment(V, subdivide=True)
        return eta, H

    else:
        print("Problem not defined")
        sys.exit(1)

def component_wise_multiply(v1, v2, field_module):
    if len(v1) != len(v2):
        raise ValueError("Vectors length doesn't match!")
    result = [0] * len(v1)
    for i in range(len(v1)):
        result[i] = v1[i] * v2[i] % field_module
    return result

def component_wise_addition(v1, v2, field_module):
    if len(v1) != len(v2):
        raise ValueError("Vectors length doesn't match!")
    result = [0] * len(v1)
    for i in range(len(v1)):
        result[i] = (v1[i] + v2[i]) % field_module
    return result

def component_wise_subtraction(v1, v2, field_module):
    if len(v1) != len(v2):
        raise ValueError("Vectors lenght doesn't match")
    subtraction = [0] * len(v1)
    for i in range(len(v1)):
        subtraction[i] = (v1[i] - v2[i])
        if subtraction[i] < 0:
            subtraction[i] = subtraction[i] + field_module
        else:
            subtraction[i] = subtraction[i] % field_module

    return subtraction

def scalar_vector_product(scalar, vector, field_module):
    result = [(scalar * x) % field_module for x in vector]
```

```
    return result

def seed_tree_leaves(master_seed, salt):
    if selected_cat == "CAT_1":
        seed_list = [csprngs.SHAKE_128(master_seed + salt,
            seed_byte_length) for _ in range(t)]
    else:
        seed_list = [csprngs.SHAKE_256(master_seed + salt,
            seed_byte_length) for _ in range(t)]

    return seed_list

def check_vector(vector, field_module):
    is_in_field= all(element % field_module == element for
        element in vector)

    if is_in_field:
        return True
    else:
        return False
```

Listato 4.13: Codice relativo alle funzioni di utilità di CROSS SC

Generazione delle chiavi

```
# KEY GENERATION ALGORITHM

def CROSS_keygen():
    seed_sk = expand_private_seed()
    seed_e, seed_pk= expand_public_seed(seed_sk)
    if selected_rsdp=="RSDPG":
        seed_v, seed_w = expand_public_seed(seed_pk)
        V = utils.create_random_matrix_old(Fq, n-k, k, seed_v)
        W = utils.create_random_matrix_old(Fz, m, n-m, seed_w)
        Im = Matrix.identity(m)
        Mg = Im.augment(W, subdivide=True)
        if selected_cat == "CAT_1":
            zi = csprngs.SHAKE_128_vec(seed_e, m, z)
        elif selected_cat == "CAT_3" or selected_cat == "CAT_5":
            zi = csprngs.SHAKE_256_vec(seed_e, m, z)
        else:
            raise ValueError("Category error!")
        zi_vec = vector(zi)

        eta = vector_matrix_product(zi_vec, Mg, z)

```

```
    elif selected_rsdp=="RSDP":
        V = utils.create_random_matrix_old(Fq, n-k, k, seed_pk)
        if selected_cat == "CAT_1":
            eta = csprngs.SHAKE_128_vec(seed_e, n, z)
        elif selected_cat == "CAT_3" or selected_cat == "CAT_5":
            eta = csprngs.SHAKE_256_vec(seed_e, n, z)
        else:
            raise ValueError("Category error!")

    else:
        print("Problem not defined")
        sys.exit(1)

    Id = Matrix.identity(n-k)
    H = Id.augment(V, subdivide=True)
    e = [1]*n

    for j in range(n):
        e[j] = (g**eta[j]) % q

        if e[j] == 0:
            raise ValueError("e[j] value is 0!")

    e_vec = vector(e)
    s = vector_matrix_product(e_vec, H.transpose(), q)

    return seed_sk, seed_pk, s
```

Listato 4.14: Codice relativo alla generazione delle chiavi di CROSS SC

Generazione della firma

```
# SIGNATURE GENERATION ALGORITHM

def CROSS_sign_one(pri, message):

    v = [0]*n
    cmt0 = [0]*t
    if selected_rsdp == "RSDPG":
        delta = [0*m]*t
    sigma = [0*n]*t
    eta_prime = [0*n]*t
    u_prime = [0*n]*t
    e_prime = [0]*n
    y = [0*n]*t

```

```
    # Key material expansion
    if selected_rsdp == "RSDPG":
        zi, H, Mg = expand_seed(pri)
        zi_vec = vector(zi)
        eta = zi_vec * Mg
    elif selected_rsdp == "RSDP":
        eta, H = expand_seed(pri)
    else:
        print("Problem not defined")
        sys.exit(1)

    # Computation of commitments
    M_seed = expand_private_seed()
    salt = get_salt()

    seed_tree_struct = seed_tree.SeedTree_x(M_seed, salt, t,
        selected_cat, seed_byte_length)
    seed_list = seed_tree_struct[-1][:t]

    for i in range(t):
        seed_u_prime, seed_e_prime = expand_public_seed(seed_list
            [i])
        if selected_rsdp == "RSDPG":
            if selected_cat == "CAT_1":
                zi_prime = csprngs.SHAKE_128_vec(seed_e_prime, m,
                    z)
            elif selected_cat == "CAT_3" or selected_cat == "CAT_5
                ":
                zi_prime = csprngs.SHAKE_256_vec(seed_e_prime, m,
                    z)
            else:
                raise ValueError("Category error!")

            zi_vec = vector(zi)
            zi_prime_vec = vector(zi_prime)
            delta[i] = component_wise_subtraction(zi_vec,
                zi_prime_vec, z)

            zi_prime_vec = vector(zi_prime)

            eta_prime[i] = vector_matrix_product(zi_prime_vec, Mg,
                 z)


        elif selected_rsdp == "RSDP":
            if selected_cat == "CAT_1":
```

```
54                eta_prime[i] = csprngs.SHAKE_128_vec(seed_e_prime,
                      n, z)
55            elif selected_cat == "CAT_3" or selected_cat == "CAT_5
                  ":
56                eta_prime[i] = csprngs.SHAKE_256_vec(seed_e_prime,
                      n, z)
57            else:
58                raise ValueError("Category error!")
59
60        else:
61            raise ValueError("Problem error!")
62
63        eta_vec = vector(eta)
64        eta_prime_vec = vector(eta_prime[i])
65        sigma[i] = component_wise_subtraction(eta_vec,
              eta_prime_vec, z)
66
67        for j in range(n):
68            v[j] = (g**sigma[i][j]) % q
69
70        v_vec = vector(v)
71
72        if selected_cat == "CAT_1":
73            u_prime[i] = csprngs.SHAKE_128_vec(seed_u_prime, n, q)
74        elif selected_cat == "CAT_3" or selected_cat == "CAT_5":
75            u_prime[i] = csprngs.SHAKE_256_vec(seed_u_prime, n, q)
76        else:
77            raise ValueError("Category error!")
78        u_prime_vec = vector(u_prime[i])
79        u = component_wise_multiply(v_vec, u_prime_vec, q) #
              length
80
81        u_vec = vector(u)
82
83        s_tilde = vector_matrix_product_star(u_vec, H.transpose()
              , q)
84
85        for x in range(len(s_tilde)):
86            if s_tilde[x] == 0:
87                s_tilde[x] = 1
88
89        # Conversion to Hash
90        s_tilde_str = ''.join(map(str, s_tilde))
91        if selected_rsdp == "RSDPG":
92            delta_i_str = ''.join(map(str, delta[i]))
93        sigma_i_str = ''.join(map(str, sigma[i]))
94        salt_str = ''.join(map(str, salt))
```

```
95          i_str = str(i)
96
97          # Commitments Generation
98          if selected_rsdp == "RSDPG":
99              data_to_hash_cmt0 = s_tilde_str + delta_i_str +
                    salt_str + i_str
100             if selected_cat == "CAT_1":
101                 cmt0[i] = hash.SHA3_256_HASH(data_to_hash_cmt0.
                        encode())
102             elif selected_cat == "CAT_3":
103                 cmt0[i] = hash.SHA3_384_HASH(data_to_hash_cmt0.
                        encode())
104             elif selected_cat == "CAT_5":
105                 cmt0[i] = hash.SHA3_512_HASH(data_to_hash_cmt0.
                        encode())
106             else:
107                 raise ValueError("Category␣error!")
108
109         elif selected_rsdp == "RSDP":
110             data_to_hash_cmt0 = s_tilde_str + sigma_i_str +
                    salt_str + i_str
111             if selected_cat == "CAT_1":
112                 cmt0[i] = hash.SHA3_256_HASH(data_to_hash_cmt0.
                        encode())
113             elif selected_cat == "CAT_3":
114                 cmt0[i] = hash.SHA3_384_HASH(data_to_hash_cmt0.
                        encode())
115             elif selected_cat == "CAT_5":
116                 cmt0[i] = hash.SHA3_512_HASH(data_to_hash_cmt0.
                        encode())
117             else:
118                 raise ValueError("Category␣error!")
119         else:
120             raise ValueError("Problem␣error!")
121
122
123     # Merkle Root and Commitment Hashing
124     d0, tree = merkle.merkle_root_lnh(cmt0, selected_cat)
125     tree = tree[::-1]
126
127     # First Challenge vector extraction
128
129     if selected_cat == "CAT_1":
130         dm = hash.SHA3_256_HASH(message)
131         dbeta = hash.SHA3_256_HASH(dm + d0 + salt)
132         beta = csprngs.SHAKE_128_vec_star_(dbeta, t, q)
133     elif selected_cat == "CAT_3":
```

```
        dm = hash.SHA3_384_HASH(message)
        dbeta = hash.SHA3_384_HASH(dm + d0 + salt)
        beta = csprngs.SHAKE_256_vec_star_(dbeta, t, q)
    elif selected_cat == "CAT_5":
        dm = hash.SHA3_512_HASH(message)
        dbeta = hash.SHA3_512_HASH(dm + d0 + salt)
        beta = csprngs.SHAKE_256_vec_star_(dbeta, t, q)
    else:
        raise ValueError("Category␣Error!")

    # Computation of First Responses

    j= 0

    for i in range(t):
        for j in range(n):
            e_prime[j] = (g**eta_prime[i][j]) % q

        e_prime_vec = vector(e_prime)
        temp_i = scalar_vector_product(beta[i], e_prime_vec, q)
        temp_vec = vector(temp_i)
        u_prime_vec = vector(u_prime[i])

        y[i] = component_wise_addition(u_prime_vec, temp_vec, q)

    # Second Challenge vector extraction
    y_str = ''.join(map(str, y))
    dbeta_str = ''.join(map(str, dbeta))
    input = y_str + dbeta_str

    if selected_cat == "CAT_1":
        db = hash.SHA3_256_HASH(input.encode())
        b = csprngs.SHAKE_128_vec_weighted(db, t, 2, w)
    elif selected_cat == "CAT_3":
        db = hash.SHA3_384_HASH(input.encode())
        b = csprngs.SHAKE_256_vec_weighted(db, t, 2, w)
    elif selected_cat == "CAT_5":
        db = hash.SHA3_512_HASH(input.encode())
        b = csprngs.SHAKE_256_vec_weighted(db, t, 2, w)
    else:
        raise ValueError("Category␣error!")

    # Computation of Second Responses
    proof = merkle.merkle_proof(tree, cmt0, b)

    seed_path = seed_tree.SeedPaths(seed_tree_struct, t, b)

```

```
    # Signature composition

    if selected_rsdp == "RSDPG":
        rsp0 = utils.initialize_couple_vector(t-w, n, m, q, z)
    elif selected_rsdp == "RSDP":
        rsp0 = utils.initialize_couple_vector(t-w, n, n, q, z)
    else:
        raise ValueError("Problem error!")

    j = 0

    for i in range(t):
        if b[i] == 0:
            if selected_rsdp == "RSDPG":
                rsp0[j] = (y[i], delta[i])
            elif selected_rsdp == "RSDP":
                rsp0[j] = (y[i], sigma[i])
            else:
                raise ValueError("Problem error!")
            j += 1

    bytes_proof = pickle.dumps(proof)
    bytes_seed_path = pickle.dumps(seed_path)
    bytes_rsp0 = pickle.dumps(rsp0)

    proof_len = len(bytes_proof)
    seed_path_len = len(bytes_seed_path)
    rsp0_len = len(bytes_rsp0)

    tuple_signature = (salt, d0, db, bytes_proof, bytes_seed_path
        , bytes_rsp0)

    if selected_cat == "CAT_1":
        format = f'{2*seed_byte_length}s 32s 32s {proof_len}s {
            seed_path_len}s {rsp0_len}s'
    elif selected_cat == "CAT_3":
        format = f'{2*seed_byte_length}s 48s 48s {proof_len}s {
            seed_path_len}s {rsp0_len}s'
    elif selected_cat == "CAT_5":
        format = f'{2*seed_byte_length}s 64s 64s {proof_len}s {
            seed_path_len}s {rsp0_len}s'
    else:
        raise ValueError("Category error!")

    signature = struct.pack(format, *tuple_signature)

    return signature, format
```

Listato 4.15: Codice relativo alla generazione della firma di CROSS SC

Verifica della firma

```
# SIGNATURE VERIFICATION ALGORITHM

def CROSS_verify_one(seed_pk, s, msg, signature, f):
    # Key material expansion
    if selected_rsdp == "RSDPG":
        seed_v, seed_w = expand_public_seed(seed_pk)
        V = utils.create_random_matrix_old(Fq, n-k, k, seed_v)
        W = utils.create_random_matrix_old(Fz, m, n-m, seed_w)
        Im = Matrix.identity(m)
        Mg = Im.augment(W, subdivide=True)

    elif selected_rsdp == "RSDP":
        V = utils.create_random_matrix_old(Fq, n-k, k, seed_pk)
    else:
        raise ValueError("Problem selection wrong")

    Id = Matrix.identity(n-k)
    H = Id.augment(V, subdivide=True)
    decoded = struct.unpack(f, signature)

    salt = decoded[0]
    d0 = decoded[1]
    db = decoded[2]
    bytes_proof = decoded[3]
    bytes_seed_path = decoded[4]
    bytes_rsp0= decoded[5]

    proof = pickle.loads(bytes_proof)
    seed_path = pickle.loads(bytes_seed_path)
    rsp0 = pickle.loads(bytes_rsp0)

    # Challenge recomputation
    if selected_cat == "CAT_1":
        dm = hash.SHA3_256_HASH(msg)
        dbeta = hash.SHA3_256_HASH(dm + d0 + salt)
        beta = csprngs.SHAKE_128_vec_star_(dbeta, t, q)
        b = csprngs.SHAKE_128_vec_weighted(db, t, 2, w)
    elif selected_cat == "CAT_3":
        dm = hash.SHA3_384_HASH(msg)
        dbeta = hash.SHA3_384_HASH(dm + d0 + salt)
```

```
41          beta = csprngs.SHAKE_256_vec_star_(dbeta, t, q)
42          b = csprngs.SHAKE_256_vec_weighted(db, t, 2, w)
43      elif selected_cat == "CAT_5":
44          dm = hash.SHA3_512_HASH(msg)
45          dbeta = hash.SHA3_512_HASH(dm + d0 + salt)
46          beta = csprngs.SHAKE_256_vec_star_(dbeta, t, q)
47          b = csprngs.SHAKE_256_vec_weighted(db, t, 2, w)
48      else:
49          raise ValueError("Category␣Error!")
50
51      seed_list = seed_tree.Recompute_Leaves(seed_path, b, salt,
            selected_cat, seed_byte_length)
52
53      cmt0 = [0] * t
54      j = 0
55      j_rsp = 0
56      e_prime = [0]*n
57      y = [0*n]*t
58      if selected_rsdp == "RSDPG":
59          delta = [0*m]*t
60      sigma = [0*n]*t
61      v = [0]*n
62      eta_prime = [0*n]*t
63      u_prime = [0*n]*t
64
65      for i in range(t):
66          if b[i] == 1:
67              i_str = str(i)
68              salt_str = ''.join(map(str, salt))
69              seed_u_prime, seed_e_prime = expand_public_seed(
                    seed_list[i])
70
71              if selected_rsdp == "RSDPG":
72                  if selected_cat == "CAT_1":
73                      zi_prime = csprngs.SHAKE_128_vec(seed_e_prime,
                            m, z)
74                  elif selected_cat == "CAT_3" or selected_cat == "
                        CAT_5":
75                      zi_prime = csprngs.SHAKE_256_vec(seed_e_prime,
                            m, z)
76                  else:
77                      raise ValueError("Category␣error!")
78
79                  zi_i_vec = vector(zi_prime)
80                  eta_prime[i] = vector_matrix_product(zi_i_vec, Mg,
                         z)
81
```

```
            elif selected_rsdp == "RSDP":
                if selected_cat == "CAT_1":
                    eta_prime[i] = csprngs.SHAKE_128_vec(
                        seed_e_prime, n, z)
                elif selected_cat == "CAT_3" or selected_cat == "
                    CAT_5":
                    eta_prime[i] = csprngs.SHAKE_256_vec(
                        seed_e_prime, n, z)
                else:
                    raise ValueError("Category error!")
            else:
                continue

            for j in range(n):
                e_prime[j] = (g**eta_prime[i][j]) % q

            if selected_cat == "CAT_1":
                u_prime[i] = csprngs.SHAKE_128_vec(seed_u_prime, n
                    , q)
            elif selected_cat == "CAT_3" or selected_cat == "CAT_5
                ":
                u_prime[i] = csprngs.SHAKE_256_vec(seed_u_prime, n
                    , q)
            else:
                raise ValueError("Category error!")

            e_prime_vec = vector(e_prime)
            temp_i = scalar_vector_product(beta[i], e_prime_vec, q
                )
            temp_vec = vector(temp_i)
            u_prime_vec = vector(u_prime[i])
            y[i] = component_wise_addition(u_prime_vec, temp_vec,
                q) # if the leaves are correctly recomputed y[i]s
                must be identical for b[i]=1

        elif b[i] == 0:
            if selected_rsdp == "RSDPG":
                y[i] = rsp0[j_rsp][0]
                delta[i] = rsp0[j_rsp][1]
                boolean = check_vector(delta[i], z)
                assert boolean == True, "boolean no true"
                delta_vec = vector(delta[i])
                sigma[i] = vector_matrix_product(delta_vec, Mg, z)

                for j in range(n):
                    v[j] = (g**sigma[i][j]) % q

```

```
120                 v_vec = vector(v)
121                 y_i_vec = vector(y[i])
122
123                 y_prime = component_wise_multiply(v_vec, y_i_vec,
                        q)
124
125                 y_prime_vec = vector(y_prime)
126                 temp1 = vector_matrix_product_star(y_prime_vec, H.
                        transpose(), q)
127                 s_vec = vector(s)
128                 temp2 = scalar_vector_product(beta[i], s_vec, q)
129                 temp1_vec = vector(temp1)
130                 temp2_vec = vector(temp2)
131
132                 s_tilde = component_wise_subtraction(temp1_vec,
                        temp2_vec, q)
133                 for x in range(len(s_tilde)):
134                     if s_tilde[x] == 0:
135                         s_tilde[x] = 1
136
137                 s_tilde_str = ''.join(map(str, s_tilde))
138                 delta_i_str = ''.join(map(str, delta[i]))
139                 salt_str = ''.join(map(str, salt))
140                 i_str = str(i)
141
142                 data_to_encode0 = s_tilde_str + delta_i_str +
                        salt_str + i_str
143
144                 if selected_cat == "CAT_1":
145                     cmt0[i] = hash.SHA3_256_HASH(data_to_encode0.
                            encode())
146                 elif selected_cat == "CAT_3":
147                     cmt0[i] = hash.SHA3_384_HASH(data_to_encode0.
                            encode())
148                 elif selected_cat == "CAT_5":
149                     cmt0[i] = hash.SHA3_512_HASH(data_to_encode0.
                            encode())
150                 else:
151                     raise ValueError("Category␣error!")
152
153             elif selected_rsdp == "RSDP": #RSDP
154                 y[i] = rsp0[j_rsp][0]
155                 sigma[i] = rsp0[j_rsp][1]
156                 boolean = check_vector(sigma[i], z)
157                 assert boolean == True, "boolean␣no␣true"
158
159                 for j in range(n):
```

```
                    v[j] = (g**sigma[i][j]) % q

                v_vec = vector(v)
                y_i_vec = vector(y[i])
                y_prime = component_wise_multiply(v_vec, y_i_vec,
                    q)

                y_prime_vec = vector(y_prime)

                temp1 = vector_matrix_product_star(y_prime_vec, H.
                    transpose(), q)

                s_vec = vector(s)
                temp2 = scalar_vector_product(beta[i], s_vec, q)

                temp1_vec = vector(temp1)
                temp2_vec = vector(temp2)

                s_tilde = component_wise_subtraction(temp1_vec,
                    temp2_vec, q)
                for x in range(len(s_tilde)):
                    if s_tilde[x] == 0:
                        s_tilde[x] = 1

                s_tilde_str = ''.join(map(str, s_tilde))

                sigma_i_str = ''.join(map(str, sigma[i]))
                salt_str = ''.join(map(str, salt))
                i_str = str(i)

                if selected_cat == "CAT_1":
                    data_to_encode0 = s_tilde_str + sigma_i_str +
                        salt_str + i_str
                    cmt0[i] = hash.SHA3_256_HASH(data_to_encode0.
                        encode())
                elif selected_cat == "CAT_3":
                    data_to_encode0 = s_tilde_str + sigma_i_str +
                        salt_str + i_str
                    cmt0[i] = hash.SHA3_384_HASH(data_to_encode0.
                        encode())
                elif selected_cat == "CAT_5":
                    data_to_encode0 = s_tilde_str + sigma_i_str +
                        salt_str + i_str
                    cmt0[i] = hash.SHA3_512_HASH(data_to_encode0.
                        encode())
                else:
                    raise ValueError("Category␣error!")
```

```
            else:
                continue

            j_rsp += 1
        else:
            continue

    d0_prime = merkle.recompute_root(cmt0, proof, b, selected_cat
        , seed_byte_length)

    y_str = ''.join(map(str, y))
    dbeta_str = ''.join(map(str, dbeta))
    input = y_str + dbeta_str

    if selected_cat == "CAT_1":
        db_prime = hash.SHA3_256_HASH(input.encode())
    elif selected_cat == "CAT_3":
        db_prime = hash.SHA3_384_HASH(input.encode())
    elif selected_cat == "CAT_5":
        db_prime = hash.SHA3_512_HASH(input.encode())
    else:
        raise ValueError("Category error!")

    if d0 == d0_prime and db == db_prime:
        return True
    else:
        if not d0 == d0_prime:
            print("d0 different from d0 prime")
        if not db == db_prime:
            print("db different from db prime")
        return False
```

Listato 4.16: Codice relativo alla verifica della firma di CROSS SC

Main

```
#CROSS SC Main

if __name__ == "__main__":
    seed_sk, seed_pk, s = CROSS_keygen()
    sign, f = CROSS_sign_one(seed_sk, msg)
    bool = CROSS_verify_one(seed_pk, s, msg, sign, f)
    print(bool)
```

Listato 4.17: Codice relativo al main di CROSS SC

CSPRNG

csprngs.py

```
from base64 import b64encode
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
from Crypto.Hash import SHAKE128
from Crypto.Hash import SHAKE256
import random
import math

def AES_CTR(data, len):
    key = get_random_bytes(len)
    cipher = AES.new(key, AES.MODE_CTR)
    ct_bytes = cipher.encrypt(data)
    nonce = b64encode(cipher.nonce).decode('utf-8')
    ct = b64encode(ct_bytes).decode('utf-8')
    return ct

def AES_CTR_CSPRNG(seed, len, dim, field_module):
    key = get_random_bytes(len)
    cipher = AES.new(key, AES.MODE_CTR)
    ct_bytes = cipher.encrypt(seed * dim)
    #nonce = b64encode(cipher.nonce).decode('utf-8')
    #ct = b64encode(ct_bytes).decode('utf-8')
    vector = [int.from_bytes(ct_bytes[i:i+1], byteorder='big') %
        field_module for i in range(dim)]
    return vector

def SHAKE_128(data, len):
    shake = SHAKE128.new()
    shake.update(data)
    l = int(len)
    output = shake.read(l) #can be .hex()
    return output

def SHAKE_256(data, len):
    shake = SHAKE256.new()
    shake.update(data)
    l = int(len)
    output = shake.read(l) # can be .hex()
    return output

def SHAKE_128_vec(seed, dim, field_module):
    shake = SHAKE128.new()
    shake.update(seed)
```

```
    temp = shake.read(dim)
    vector = [int(temp[i]) % field_module for i in range(dim)]
    return vector

def SHAKE_256_vec(seed, dim, field_module):
    shake = SHAKE256.new()
    shake.update(seed)
    temp = shake.read(dim)
    vector = [int(temp[i]) % field_module for i in range(dim)]
    return vector

def SHAKE_128_vec_star(seed, dim, field_module):
    shake = SHAKE128.new()
    shake.update(seed)
    temp = shake.read(dim)
    vector = [int(temp[i]) % field_module for i in range(dim)]
    for i in range(dim):
        if vector[i] == 0:
            vector[i] = int(temp[i+1]) % field_module
    return vector

def SHAKE_128_vec_star_(seed, dim, field_module):
    shake = SHAKE128.new()
    shake.update(seed)
    vector = [0]*dim
    base = math.ceil((math.log(field_module, 2)))
    for i in range(dim):
        temp1 = shake.read(2**base)
        temp = int.from_bytes(temp1, byteorder='big')
        vector[i] = temp % field_module
        while vector[i] == 0:
            temp1 = shake.read(2**base)
            temp = int.from_bytes(temp1, byteorder='big')
            vector[i] = temp % field_module
    return vector

def SHAKE_256_vec_star_(seed, dim, field_module):
    shake = SHAKE256.new()
    shake.update(seed)
    vector = [0]*dim
    base = math.ceil((math.log(field_module, 2)))
    for i in range(dim):
        temp1 = shake.read(2**base)
        temp = int.from_bytes(temp1, byteorder='big')
        vector[i] = temp % field_module
        while vector[i] == 0:
            temp1 = shake.read(2**base)
```

```
            temp = int.from_bytes(temp1, byteorder='big')
            vector[i] = temp % field_module
    return vector

def SHAKE_256_vec_star(seed, dim, field_module, q):
    shake = SHAKE256.new(q)
    shake.update(seed)
    temp = shake.read(2*dim)
    vector = [int(temp[i]) % field_module for i in range(dim)]
    for i in range(dim):
        if vector[i] == 0:
            vector[i] = int(temp[i+1]) % field_module
    return vector

def SHAKE_128_vec_weighted(seed, dim, field_module, w):
    shake = SHAKE128.new()
    shake.update(seed)
    temp = shake.read(dim)
    vector = [int(temp[i]) % field_module for i in range(dim)]

    if w == 0:
        vector = [0] * dim
        return vector
    elif w == dim:
        vector = [1] * dim
        return vector
    elif w < 0 or w > dim:
        raise ValueError("Not␣valid␣Hamming␣Weight")

    current_weight = sum(vector)

    if current_weight == w:
        return vector
    elif current_weight < w:
        missing_bits = w - current_weight
        for i in range(dim):
            if missing_bits == 0:
                break
            if vector[i] == 0:
                vector[i] = 1
                missing_bits -= 1
    else:
        extra_bits = current_weight - w
        while extra_bits > 0:
            index_to_change = random.choice([i for i, bit in
                enumerate(vector) if bit == 1])
            vector[index_to_change] = 0
```

```
            extra_bits -= 1

    random.seed(seed)
    random.shuffle(vector)
    return vector

def SHAKE_256_vec_weighted(seed, dim, field_module, w):
    shake = SHAKE256.new()
    shake.update(seed)
    temp = shake.read(dim)
    vector = [int(temp[i]) % field_module for i in range(dim)]

    if w == 0:
        vector = [0] * dim
        return vector
    elif w == dim:
        vector = [1] * dim
        return vector
    elif w < 0 or w > dim:
        raise ValueError("Not valid Hamming Weight")

    current_weight = sum(vector)

    if current_weight == w:
        return vector
    elif current_weight < w:
        missing_bits = w - current_weight
        for i in range(dim):
            if missing_bits == 0:
                break
            if vector[i] == 0:
                vector[i] = 1
                missing_bits -= 1
    else:
        extra_bits = current_weight - w
        while extra_bits > 0:
            index_to_change = random.choice([i for i, bit in
                enumerate(vector) if bit == 1])
            vector[index_to_change] = 0
            extra_bits -= 1

    random.seed(seed)
    random.shuffle(vector)
    return vector
```

Listato 4.18: Funzioni CSPRNG

HASH

hash.py

```python
from Crypto.Hash import SHA3_256
from Crypto.Hash import SHA3_384
from Crypto.Hash import SHA3_512

def SHA3_256_HASH(data):
    hasher = SHA3_256.new()
    hasher.update(data)
    output = hasher.digest()
    return output

def SHA3_384_HASH(data):
    hasher = SHA3_384.new()
    hasher.update(data)
    output = hasher.digest()
    return output

def SHA3_512_HASH(data):
    hasher = SHA3_512.new()
    hasher.update(data)
    output = hasher.digest()
    return output
```

Listato 4.19: Funzioni HASH

UTILS

merkle.py

```python
from HASH import hash
import math
import secrets

def calculate_hash(leaf, selected_cat):
        if selected_cat == "CAT_1":
            return hash.SHA3_256_HASH(leaf)
        elif selected_cat == "CAT_3":
            return hash.SHA3_384_HASH(leaf)
        elif selected_cat == "CAT_5":
            return hash.SHA3_512_HASH(leaf)

def merkle_root(leaves, selected_cat):
    if not leaves:
```

```
15          return None, []
16
17      hashed_leaves = list(map(lambda leaf: calculate_hash(leaf,
            selected_cat), leaves))
18
19      #hashed_leaves = list(map(calculate_hash(selected_cat),
            leaves))
20      merkle_tree = [hashed_leaves]
21
22      while len(hashed_leaves) > 1:
23          next_level = []
24          for i in range(0, len(hashed_leaves), 2):
25              #print("\n\n" , " i fa sempre comodo root:", i)
26              if i + 1 < len(hashed_leaves):
27                  combined_hash = calculate_hash(hashed_leaves[i] +
                        hashed_leaves[i + 1], selected_cat)
28                  next_level.append(combined_hash)
29              else: #se ho una foglia isolata la ricopio sul padre
30                  next_level.append(hashed_leaves[i])
31
32          merkle_tree.append(next_level)
33          hashed_leaves = next_level
34
35      return hashed_leaves[0], merkle_tree
36
37  def merkle_root_lnh(leaves, selected_cat): #leaves not hashed
38      if not leaves:
39          return None, []
40
41      merkle_tree = [leaves]
42
43      #print("merkol tree :", merkle_tree)
44      #print("len livs :", len(leaves))
45      #print("livs :", leaves)
46      #print("tipo foglia :", type(leaves[0]))
47
48      while len(leaves) > 1:
49          next_level = []
50          if len(leaves) % 2 == 1:
51              for i in range(0, len(leaves)-2, 2):
52                  #print("i fa sempre comodo root:", i)
53                  combined_hash = calculate_hash(leaves[i]+leaves[i
                        +1], selected_cat)
54                  #print("I generated a father hash from:\n", leaves
                        [i], "\n\n", leaves[i + 1], "\n\n AND IT IS:",
                         combined_hash, "\n\n")
55                  next_level.append(combined_hash)
```

```
            #print("Last element copied cause single :", leaves[-1])
            next_level.append(leaves[-1])
        else:
            for i in range(0, len(leaves)-1, 2):
                #print("i fa sempre comodo root:", i)
                combined_hash = calculate_hash(leaves[i] + leaves[i + 1], selected_cat)
                #print("I generated a father hash from:\n", leaves[i], "\n\n", leaves[i + 1], "\n\n AND IT IS:", combined_hash, "\n\n")
                next_level.append(combined_hash)

        merkle_tree.append(next_level)
        leaves = next_level

    return leaves[0], merkle_tree

def count_nodes_at_each_level(tree, level = 0, level_counts=None):
    if level_counts is None:
        level_counts = [0] * len(tree)

    for i in tree:
        vector = i
        level_counts[level] = len(vector)
        level += 1

    return level_counts

def merkle_proof(T, cmt0, b):
    t = len(cmt0)
    num_nodes = 2*t - 1
    level_num = 0
    level = int(math.ceil(math.log2(t) + 1))
    level_nodes = count_nodes_at_each_level(T, level_num)

    proof = []

    T_prime = [0] * len(T)

    for i in range(level):
        T_prime[i] = [0] * level_nodes[i]

    # leaves initialization
    for node in level_nodes: # from root to leaf
        #print("node:", node)
```

```
98          if node == 1: # se ho 1 elemento a quel livello
99              T_prime[0][0] = 0 #root
100         elif node == t: # se ho 243 elementi
101             for j in range(node):
102                 if b[j] == 0:
103                     T_prime[-1][j] = 1
104         else:
105             for j in range(node):
106                 T_prime[level_nodes.index(node)][j] = 0
107
108     # Traversing tree
109     for node in reversed(level_nodes): # from leaf to root
110         if node % 2 == 1:
111             if T_prime[level_nodes.index(node)][-1] == 1:
112                 T_prime[level_nodes.index(node)-1][-1] = 1
113             elif T_prime[level_nodes.index(node)][-1] == 0:
114                 T_prime[level_nodes.index(node)-1][-1] = 0
115             else:
116                 continue
117
118             for i in range(0, node-2, 2): # from left to right
119                 if T_prime[level_nodes.index(node)][i] == 1 and
                        T_prime[level_nodes.index(node)][i+1] == 1: #
                        if both COMPUTED
120                     T_prime[level_nodes.index(node)-1][int(i/2)] =
                            1
121
122                 elif T_prime[level_nodes.index(node)][i] == 1 and
                        T_prime[level_nodes.index(node)][i+1] == 0:
123                     T_prime[level_nodes.index(node)-1][int(i/2)] =
                            1 # the father is set as COMPUTED
124                     proof.insert(0, T[level_nodes.index(node)][i
                            +1])
125
126                 elif T_prime[level_nodes.index(node)][i] == 0 and
                        T_prime[level_nodes.index(node)][i+1] == 1:
127                     T_prime[level_nodes.index(node)-1][int(i/2)] =
                            1 # the father is set as COMPUTED
128                     proof.insert(0, T[level_nodes.index(node)][i])
129
130                 else: # if both are UNCOMPUTED
131                     continue
132         else:
133
134             for i in range(0, node-1, 2): # from left to right
135                 if T_prime[level_nodes.index(node)][i] == 1 and
                        T_prime[level_nodes.index(node)][i+1] == 1: #
```

```
                    if both COMPUTED
                    T_prime[level_nodes.index(node)-1][int(i/2)] =
                        1

                elif T_prime[level_nodes.index(node)][i] == 1 and
                    T_prime[level_nodes.index(node)][i+1] == 0:
                    T_prime[level_nodes.index(node)-1][int(i/2)] =
                        1 # the father is set as COMPUTED
                    proof.insert(0, T[level_nodes.index(node)][i
                        +1])
                    #print("node inserted in proof:", T[level_nodes
                        .index(node)][i+1])

                elif T_prime[level_nodes.index(node)][i] == 0 and
                    T_prime[level_nodes.index(node)][i+1] == 1:
                    T_prime[level_nodes.index(node)-1][int(i/2)] =
                        1 # the father is set as COMPUTED
                    proof.insert(0, T[level_nodes.index(node)][i])
                    #print("node inserted in proof:", T[level_nodes
                        .index(node)][i])

                else: # if both are UNCOMPUTED
                    continue
    return proof

def initialize_tree(t):
    height = math.ceil(math.log(t, 2)+1)
    tree = [0] * height
    ctr_ = 0

    for i in range(height-1, 0, -1):
        if i == height-1:
            tree[height-1] = [0] * t
            ctr_ = math.floor((t/2) + 1)
            tree[height-2] = [0] * ctr_
        elif i == 0:
            tree[i] = [0]
        else:
            if ctr_ % 2 == 1:
                ctr_ = math.floor((ctr_/2) + 1)
                tree[i-1] = [0] * ctr_
            else:
                ctr_ = int(ctr_/2)
                tree[i-1] = [0] * ctr_

    return tree

```

```
def initialize_tree_bytes(t, leng):
    height = math.ceil(math.log(t, 2)+1)
    tree = [0] * height
    ctr_ = 0

    for i in range(height-1, 0, -1):
        if i == height-1:
            x = secrets.token_bytes(leng)
            tree[height-1] = [x] * t
            ctr_ = math.floor((t/2) + 1)
            tree[height-2] = [x] * ctr_
        elif i == 0:
            tree[i] = [0]
        else:
            if ctr_ % 2 == 1:
                ctr_ = math.floor((ctr_/2) + 1)
                tree[i-1] = [x] * ctr_
            else:
                ctr_ = int(ctr_/2)
                tree[i-1] = [x] * ctr_

    return tree

def recompute_root(cmt0, proof, b, selected_cat, seed_byte_len):
    t = len(cmt0)
    level_num = 0
    height = math.ceil(math.log(t, 2)+1)

    T = initialize_tree_bytes(t, seed_byte_len)
    T_prime = initialize_tree(t)

    level_nodes = count_nodes_at_each_level(T_prime, level_num)

    for i in range(t):
        if b[i] == 0:
            T_prime[-1][i] = 1
            T[-1][i] = cmt0[i]
        else:
            T[-1][i] = []

    ctr = len(proof)

    # Recomputing T_prime and T
    for node in reversed(level_nodes): # from leaf to root

        if node % 2 == 1:
            if T_prime[level_nodes.index(node)][-1] == 1:
```

```
                T_prime[level_nodes.index(node)-1][-1] = 1
                T[level_nodes.index(node)-1][-1] = T[level_nodes.
                    index(node)][-1]
            elif T_prime[level_nodes.index(node)][-1] == 0:
                T_prime[level_nodes.index(node)-1][-1] = 0
            else:
                continue

            for i in range(0, node-2, 2): # from left to right
                if T_prime[level_nodes.index(node)][i] == 1 and
                    T_prime[level_nodes.index(node)][i+1] == 1: #
                    if both COMPUTED
                    T_prime[level_nodes.index(node)-1][int(i/2)] =
                        1
                    T[level_nodes.index(node)-1][int(i/2)] =
                        calculate_hash(T[level_nodes.index(node)][i
                        ] + T[level_nodes.index(node)][i+1],
                        selected_cat)

                elif T_prime[level_nodes.index(node)][i] == 1 and
                    T_prime[level_nodes.index(node)][i+1] == 0:
                    T_prime[level_nodes.index(node)-1][int(i/2)] =
                        1 # the father is set as COMPUTED
                    T[level_nodes.index(node)][i+1] = proof[ctr-1]
                    ctr -= 1
                    T[level_nodes.index(node)-1][int(i/2)] =
                        calculate_hash(T[level_nodes.index(node)][i
                        ] + T[level_nodes.index(node)][i+1],
                        selected_cat)

                elif T_prime[level_nodes.index(node)][i] == 0 and
                    T_prime[level_nodes.index(node)][i+1] == 1:
                    T_prime[level_nodes.index(node)-1][int(i/2)] =
                        1 # the father is set as COMPUTED
                    T[level_nodes.index(node)][i] = proof[ctr-1]
                    ctr -= 1
                    T[level_nodes.index(node)-1][int(i/2)] =
                        calculate_hash(T[level_nodes.index(node)][i
                        ] + T[level_nodes.index(node)][i+1],
                        selected_cat)

                else: # if both are UNCOMPUTED
                    continue

        else:
            for i in range(0, node-1, 2): # from left to right
```

```
250                 if T_prime[level_nodes.index(node)][i] == 1 and
                        T_prime[level_nodes.index(node)][i+1] == 1: #
                        if both COMPUTED
251                     T_prime[level_nodes.index(node)-1][int(i/2)] =
                            1
252                     T[level_nodes.index(node)-1][int(i/2)] =
                            calculate_hash(T[level_nodes.index(node)][i
                            ] + T[level_nodes.index(node)][i+1],
                            selected_cat)
253
254
255                 elif T_prime[level_nodes.index(node)][i] == 1 and
                        T_prime[level_nodes.index(node)][i+1] == 0:
256                     T_prime[level_nodes.index(node)-1][int(i/2)] =
                            1 # the father is set as COMPUTED
257
258                     T[level_nodes.index(node)][i+1] = proof[ctr-1]
259                     ctr -= 1
260                     T[level_nodes.index(node)-1][int(i/2)] =
                            calculate_hash(T[level_nodes.index(node)][i
                            ] + T[level_nodes.index(node)][i+1],
                            selected_cat)
261
262                 elif T_prime[level_nodes.index(node)][i] == 0 and
                        T_prime[level_nodes.index(node)][i+1] == 1:
263                     T_prime[level_nodes.index(node)-1][int(i/2)] =
                            1 # the father is set as COMPUTED
264                     T[level_nodes.index(node)][i] = proof[ctr-1]
265                     ctr -= 1
266                     T[level_nodes.index(node)-1][int(i/2)] =
                            calculate_hash(T[level_nodes.index(node)][i
                            ] + T[level_nodes.index(node)][i+1],
                            selected_cat)
267
268                 else: # if both are UNCOMPUTED
269                     continue
270
271     leaves = T[-1]
272     root = T[0][0]
273
274     return root
```

Listato 4.20: Funzioni per l'albero di Merkle

seed_tree.py

```
from CSPRNG import csprngs
from UTILS import merkle
import os
import math

def SeedTree(Master_Seed, Salt, t, selected_cat, seed_byte_len):
    tree = []
    leaves = []

    root = Master_Seed + Salt
    tree.append([root])
    i=0
    while len(leaves) < t:
        current_level = tree[-1]
        next_level = []
        for node in enumerate(current_level):
            if selected_cat == "CAT_1":
                child1 = csprngs.SHAKE_128(node[1] + Salt + ((i*2)
                    +1).to_bytes(4, byteorder='big'),
                    seed_byte_len)
                child2 = csprngs.SHAKE_128(node[1] + Salt + ((i*2)
                    +2).to_bytes(4, byteorder='big'),
                    seed_byte_len)
            else:
                child1 = csprngs.SHAKE_256(node[1] + Salt + ((i*2)
                    +1).to_bytes(4, byteorder='big'),
                    seed_byte_len)
                child2 = csprngs.SHAKE_256(node[1] + Salt + ((i*2)
                    +2).to_bytes(4, byteorder='big'),
                    seed_byte_len)

            next_level.extend([child1, child2])
            leaves.extend([child1, child2])
            i += 2

        tree.append(next_level)

        if len(next_level) > t:
            break

        leaves = next_level[:]

    level_num = 0
    level_nodes = merkle.count_nodes_at_each_level(tree,
        level_num)

```

```
38      return tree
39
40  def SeedTree_x(Master_Seed, Salt, t, selected_cat, seed_byte_len)
        :
41      tree = []
42      leaves = []
43
44      root = Master_Seed + Salt
45      tree.append([root])
46      i = 0
47      while len(leaves) < t:
48          current_level = tree[-1]
49          next_level = []
50          for node in current_level:
51              if selected_cat == "CAT_1":
52                  childs = csprngs.SHAKE_128(node + Salt, 2*
                        seed_byte_len) #you can add "+ i.to_bytes(4,
                        byteorder='big')"
53                  child1 = childs[:seed_byte_len]
54                  child2 = childs[seed_byte_len:]
55
56              else:
57                  childs = csprngs.SHAKE_256(node + Salt, 2*
                        seed_byte_len)
58                  child1 = childs[:seed_byte_len]
59                  child2 = childs[seed_byte_len:]
60
61              next_level.extend([child1, child2])
62              i += 1
63
64          tree.append(next_level)
65
66          if len(next_level) > t:
67              break
68
69          leaves = next_level[:]
70
71      level_num = 0
72      level_nodes = merkle.count_nodes_at_each_level(tree,
            level_num)
73
74      return tree
75
76  def convert_to_tree_structure(tree):
77      tree_structure = []
78      level_size = 1
79      current_level = []
```

```
80
81      for node in tree:
82          if node is not None:
83              current_level.append(node)
84
85          if len(current_level) == level_size:
86              tree_structure.append(current_level)
87              current_level = []
88              level_size *= 2
89
90      if current_level:
91          tree_structure.append(current_level)
92
93      return tree_structure
94
95  def SeedPaths(Seed_Tree, t, b):
96      seed_path = []
97      node_to_publish = [0]*len(Seed_Tree)
98      level_num = 0
99      level = int(math.ceil(math.log2(t) + 1))
100     level_nodes = merkle.count_nodes_at_each_level(Seed_Tree,
            level_num)
101
102     for i in range(level):
103         node_to_publish[i] = [0] * level_nodes[i]
104
105     # Leaves configuration
106     for i in range(t): #make sense to go from 0 to t-1
107         if b[i] == 0:
108             node_to_publish[-1][i] = 0
109         elif b[i] == 1:
110             node_to_publish[-1][i] = 1
111
112     # Nodes configuration and publication
113     for node in reversed(level_nodes):
114         for i in range(0, node-1, 2):
115
116             if node_to_publish[level_nodes.index(node)][i] == 1
                    and node_to_publish[level_nodes.index(node)][i+1]
                    == 1:
117                 node_to_publish[level_nodes.index(node)-1][int(i
                        /2)] = 1 # set father to 1
118
119             elif node_to_publish[level_nodes.index(node)][i] == 1
                    and node_to_publish[level_nodes.index(node)][i+1]
                    == 0:
```

```
120                 seed_path.insert(0, Seed_Tree[level_nodes.index(
                        node)][i])
121
122             elif node_to_publish[level_nodes.index(node)][i] == 0
                    and node_to_publish[level_nodes.index(node)][i+1]
                    == 1:
123                 seed_path.insert(0, Seed_Tree[level_nodes.index(
                        node)][i+1])
124
125             else:
126                 continue
127
128     return seed_path
129
130 def SeedPaths_list_test(Seed_Tree, t, b):
131
132     seed_path = []
133
134     level = int(math.ceil(math.log2(t) + 1))
135
136     # retrieve leaves number
137     leaves_num = len(Seed_Tree[-1])
138
139     # conversion from tree to list
140     Seed_Tree_List = []
141     for sub_vector in Seed_Tree:
142         Seed_Tree_List.extend(sub_vector)
143
144     node_to_publish = [0]*len(Seed_Tree_List)
145
146     two_pow = 2 ** math.ceil(math.log2(t))
147
148     x = two_pow - t
149
150     first_cut = node_to_publish[-x:]
151
152     node_to_publish = node_to_publish[:-x]
153
154     second_cut = node_to_publish[-t:]
155
156     node_to_publish = node_to_publish[:-t]
157
158     for i in range(len(second_cut)):
159         if not len(b) == len(second_cut):
160             print("length␣doesn't␣match")
161         else:
162             if b[i] == 1:
```

```
                second_cut[i] = 1
            else:
                continue

    node_to_publish = node_to_publish + second_cut
    node_to_publish = node_to_publish + first_cut

    # Nodes configuration
    for j in (range(len(node_to_publish)-1, 2, -2)): # da 510 a 2
         perche tolgo root
        if node_to_publish[j] == 1 and node_to_publish[j-1] == 1:
            node_to_publish[int((j/2)-1)] = 1
        elif node_to_publish[j] == 1 and node_to_publish[j-1] ==
            0:
            seed_path = [Seed_Tree_List[j]] + seed_path[:]
        elif node_to_publish[j] == 0 and node_to_publish[j-1] ==
            1:
            seed_path = [Seed_Tree_List[j-1]] + seed_path[:]
        else:
            continue

    return seed_path

def SeedPaths_new(Seed_Tree, t, b):
    seed_path = []
    node_to_publish = [0]*len(Seed_Tree)
    level_num = 0
    level = int(math.ceil(math.log2(t) + 1))
    level_nodes = merkle.count_nodes_at_each_level(Seed_Tree,
        level_num)

    for i in range(level):
        node_to_publish[i] = [0] * level_nodes[i]

    # Leaves configuration
    for i in range(0, t-1):
        if b[i] == 1:
            node_to_publish[-1][i] = 1
        else:
            continue

    # Nodes configuration
    for lev in reversed(range(len(node_to_publish))):
        for i in range(0, len(node_to_publish[lev])-1, 2):
            if node_to_publish[lev][i] == 1 or node_to_publish[lev
                ][i+1] == 1:
```

```
                node_to_publish[lev-1][int(i/2)] == 1 # set father
                    to 1

    # Nodes publication
    for lev in range(len(node_to_publish)):
        for i in range(0, len(node_to_publish[lev])-1):

            if lev[-1] == level: #leaves level
                if node_to_publish[lev][i] == 1:
                    seed_path.append(Seed_Tree[lev][i])
            else:
                if node_to_publish[lev][i] == 1:
                    seed_path.append(Seed_Tree[lev][i])
                    node_to_publish[lev+1][i*2] = 0
                    node_to_publish[lev+1][(i*2)+1] = 0
                else:
                    continue

    return seed_path

def SeedPaths_list(Seed_Tree, t, b):
    seed_path = []

    leaves_num = len(Seed_Tree[-1])

    # conversion from tree to list
    Seed_Tree_List = []
    for sub_vector in Seed_Tree:
        Seed_Tree_List.extend(sub_vector)

    seed_tree_list_cutted_leaves = Seed_Tree_List[:-(leaves_num -
        t)]

    temp_leaves = Seed_Tree_List[-leaves_num:]
    leaves_from_list = temp_leaves[:t]

    node_to_publish = [0]*len(seed_tree_list_cutted_leaves)
    level = int(math.ceil(math.log2(t) + 1))

    # Leaves configuration
    last_elems = node_to_publish[-t:]

    for i in range(len(last_elems)):
        if not len(b) == len(last_elems):
            print("length␣doesn't␣match")
        else:
            if b[i] == 1:
```

```
                last_elems[i] = 1
            else:
                continue

    node_to_publish[-t:] = last_elems

    x = (len(seed_tree_list_cutted_leaves)-1)-t

    # Nodes configuration
    if node_to_publish[-1] == 1:
        seed_path = [seed_tree_list_cutted_leaves[-1]] +
            seed_path[:]

    for j in reversed(range(2, len(seed_tree_list_cutted_leaves)
        -1, 2)):
        if node_to_publish[j] == 1 and node_to_publish[j-1] == 1:
            node_to_publish[(j-1)//2] = 1
        elif node_to_publish[j] == 1 and node_to_publish[j-1] ==
            0:
            seed_path = [seed_tree_list_cutted_leaves[j]] +
                seed_path[:]
        elif node_to_publish[j] == 0 and node_to_publish[j-1] ==
            1:
            seed_path = [seed_tree_list_cutted_leaves[j-1]] +
                seed_path[:]
        else:
            continue

    if b == last_elems:
        print("b␣and␣last␣elements␣match!")

    return seed_path

def two_pow(num):
        two_pow = 2 ** math.ceil(math.log2(num))
        return two_pow

def initialize_tree_bytes(t, leng):
    height = math.ceil(math.log(t, 2) + 1)
    tree = [0] * height
    power = two_pow(t)

    for i in range(height-1, -1, -1):
        if i == 0: # root
            power = 0
            tree[0] = [b'0'*leng]
        else:
```

```
290             tree[i] = [b'0'*leng] * power
291             power = power // 2
292 
293     return tree
294 
295 def initialize_tree(t):
296     height = math.ceil(math.log(t, 2) + 1) # 9
297     tree = [0] * height
298     power = two_pow(t)
299 
300     for i in range(height-1, -1, -1): # da 8 a 0
301         if i == 0: # root
302             power = 0
303             tree[0] = [0]
304         else:
305             tree[i] = [0] * power
306             power = power // 2
307 
308     return tree
309 
310 def Recompute_Leaves(path, b, salt, selected_cat, seed_byte_len):
311     ctr = 0 # for path
312     t = len(b)
313     seed_list = []*t
314     height = int(math.ceil(math.log2(t) + 1))
315     level_num = 0
316 
317     node_to_publish = initialize_tree(t)
318 
319     tree = initialize_tree_bytes(t, seed_byte_len)
320 
321     level_nodes = merkle.count_nodes_at_each_level(tree,
            level_num)
322 
323     level_nodes2 = merkle.count_nodes_at_each_level(
            node_to_publish, level_num)
324 
325     # Leaves configuration
326     for i in range(t):
327         if b[i] == 1:
328             node_to_publish[-1][i] = 1
329 
330     # Nodes configuration and publication
331     for node in reversed(level_nodes): # from leaves to root
332         for i in range(0, node-1, 2):
333             if node_to_publish[level_nodes.index(node)][i] == 1
                    and node_to_publish[level_nodes.index(node)][i+1]
```

```
                == 1:
334             node_to_publish[level_nodes.index(node)-1][int(i
                    /2)] = 1 # set father to 1
335         else:
336             continue
337
338 for node in(level_nodes): # from root to leaves
339     if node == 1: # root level
340         continue
341     elif level_nodes.index(node) == height-1: # leaves level
342         for i in reversed(range(node)): # from 256 to 1
343             if node_to_publish[level_nodes.index(node)][i] ==
                    0:
344                 seed_list.append(0)
345             elif node_to_publish[level_nodes.index(node)][i]
                    == 1:
346                 if node_to_publish[level_nodes.index(node)-1][
                        math.floor(i/2)] == 0: # is a leaf in path
347                     seed_list.append(path[ctr])
348                     ctr += 1
349                 elif node_to_publish[level_nodes.index(node)
                        -1][math.floor(i/2)] == 1: # is a leaf
                        generated from parent node
350                     seed_list.append(tree[level_nodes.index(
                            node)][i])
351             else:
352                 continue
353     else:
354         for i in range(node-1, -1, -1):
355             if node_to_publish[level_nodes.index(node)][i] ==
                    1 and node_to_publish[level_nodes.index(node)
                    -1][math.floor(i/2)] == 0: # node from path
356                 if selected_cat == "CAT_1":
357                     childs = csprngs.SHAKE_128(path[ctr] + salt
                            , 2*seed_byte_len) #you can add "+ i.
                            to_bytes(4, byteorder='big')"
358                     ctr += 1
359                     tree[level_nodes.index(node)+1][i*2] =
                            childs[:seed_byte_len]
360                     tree[level_nodes.index(node)+1][(i*2)+1] =
                            childs[seed_byte_len:]
361                 else:
362                     childs = csprngs.SHAKE_256(path[ctr] + salt
                            , 2*seed_byte_len)
363                     ctr += 1
364                     tree[level_nodes.index(node)+1][i*2] =
                            childs[:seed_byte_len]
```

```
365                         tree[level_nodes.index(node)+1][(i*2)+1] =
                                childs[seed_byte_len:]
366
367                 elif node_to_publish[level_nodes.index(node)][i]
                        == 1 and node_to_publish[level_nodes.index(
                        node)-1][math.floor(i/2)] == 1: # node from
                        parent
368
369                     if selected_cat == "CAT_1":
370                         childs = csprngs.SHAKE_128(tree[level_nodes
                                .index(node)][i] + salt, 2*
                                seed_byte_len) #you can add "+ i.
                                to_bytes(4, byteorder='big')"
371                         tree[level_nodes.index(node)+1][i*2] =
                                childs[:seed_byte_len]
372                         tree[level_nodes.index(node)+1][(i*2)+1] =
                                childs[seed_byte_len:]
373                     else:
374                         childs = csprngs.SHAKE_256(tree[level_nodes
                                .index(node)][i] + salt, 2*
                                seed_byte_len)
375                         tree[level_nodes.index(node)+1][i*2] =
                                childs[:seed_byte_len]
376                         tree[level_nodes.index(node)+1][(i*2)+1] =
                                childs[seed_byte_len:]
377
378                 else:
379                     continue
380
381     seed_list.reverse()
382     seed_list = seed_list[:t]
383
384     return seed_list
```

Listato 4.21: Funzioni per l'albero di Seed

utils.py

```
1  from sage.all import *
2  import numpy as np
3  import hashlib
4  import random
5  import sys
6
7  def create_random_matrix(seed, field_module, rows, cols):
8      seed_hash = hashlib.sha256(seed).digest()
```

```
    seed_int = int.from_bytes(seed_hash, byteorder='big') %
        (2**32 - 1)
    np.random.seed(seed_int)
    matrix_temp = np.random.randint(0, field_module, size=(rows,
        cols))
    matrix = np.asmatrix(matrix_temp)
    return matrix

def create_random_matrix_old(field, rows, col, seed):
    random.seed(bytes(seed))
    M = Matrix(field, rows, col)
    for i in range(rows):
        for j in range(col):
            M[i,j] = random.randint(0, field.order() - 1)
    return M

def get_generator(q, z, Fz):
    assert is_prime(q),"q is not prime!"
    assert is_prime(z),"z is not prime!"
    assert (q-1)%z == 0, "z does not divide q-1!"
    alpha = Fz.primitive_element()
    print("alpha is :", alpha)
    exp = (q-1)/z
    g = alpha**exp
    g_out = Integer(g)
    return g_out

def add_in_field(element1, element2, prime):
    return (element1 + element2) % prime

def multiply_in_field(element1, element2, prime):
    return (element1 * element2) % prime

def initialize_couple_vector(vec_len, elem1_dim, elem2_dim,
    module1, module2):
    vector = []
    for _ in range(vec_len):
        component_q = [random.randint(0, module1 - 1) for _ in
            range(elem1_dim)]
        component_z = [random.randint(0, module2 - 1) for _ in
            range(elem2_dim)]
        vector.append((component_q, component_z))

    return vector

def initialize_vector(vec_len, elem_dim, field_module):
    vector = []
```

```
    for _ in range(vec_len):
        component = [random.randint(0, field_module - 1) for _ in
            range(elem_dim)]
        vector.append(component)
    return vector

def data_weight(data):
    data_size_bytes = sys.getsizeof(data)
    data_size_kb = data_size_bytes // 1024
    return data_size_kb
```

Listato 4.22: Funzioni generiche di utilità

Script dei Parametri

CROSS_fixed_weight_finder_trange.py

```
#!/usr/bin/python3
# This script computes a list of (round_number,number of
    commitments to reveal)
# pairs which provide 2^-lambda cheating probability for the
    attacker, considering
# the CROSS-ID protocol run with codes over F_q
# The parameter sets are hardcoded in the __main__ function: they
     define
# the security margin lambda, the value q and an interval of
    values for the
# number of rounds t, [t_high,t_low], to reduce the computational
     load

from math import comb,log2,ceil
import csv
from sys import argv,exit
from multiprocessing import Pool

num_cores = 128
####################################
# returns the exact number of bits to represent x
def bits_to_represent(x):
    return ceil(log2(x+1))

####################################
# function determining the equivalent number of rounds to achieve
# 2^-sec_parameter cheating probability, assuming a q/2(q-1) one
    for a single
# round
```

```
def find_number_of_random_chall_rounds(sec_parameter,q):
    return ceil(sec_parameter / -log2(q/(2*q-1)))


##################################
def soundness_error_2(param_list):
    t,w,p,sec_parameter = param_list
    min_cost_1 = 10**30
    min_cost_2 = 10**30
    min_cost = 10**30

    best_t_star = 0

    for t_star in range(1,100): #100 may need to be increased, but it seems to work
        P_beta = sum([ comb(t,j) * ((1/(p-1))**j) * (1-1/(p-1))**(t-j) for j in range(t_star,t+1)])
        if (P_beta <= 0):
            continue
        if -log2(P_beta)<sec_parameter: #continue only if cost of phase 1 is not above than SL
            #Now doing estimate for second phase
            P_b = 0
            for j in range(t_star, t_star+10): #same here, t_star+10 may be too small
                pr_j = ( comb(t,j)*((1/(p-1))**j)*(1-1/(p-1))**(t-j) )/P_beta
                pr_w_star = sum( [ (comb(j,w_star)**2) * comb(t-j, w-w_star) for w_star in range(0,min(j,w)+1)])
#               pr_w_star = sum( [ (comb(j,w_star)**2) * comb(t-j, w-w_star) for w_star in range(0,j+1)])
                P_b += (pr_j*pr_w_star)

            # P_b = P_b/(comb(t,w)**2)
            log2_P_b = log2(P_b)- log2(comb(t,w)**2)
            P_b = 2**log2_P_b
            # Compute cost of full attack
            cost = log2(1/P_beta+1/P_b)

            if cost < min_cost:
                min_cost = cost
                best_t_star = t_star
                min_cost_1 = -log2(P_beta)
                min_cost_2 = -log2(P_b)
    return param_list+[best_t_star, min_cost_1, min_cost_2, min_cost]

```

```
63 ########################################
64 # creates a list of pairs (length, weight) such that the fixed-
       weight strings
65 # with num_nonzero_values are at least as many as 2**
       sec_parameter
66 # i.e., comb(length, weight)*num_nonzero_values**weight ~ 2**
       sec_parameter
67
68 #if tree_flag = 0, we do not consider the use of Merkle and seed
       trees; this is used only for the fast version
69 def find_fixed_weight_options(sec_parameter, q,t_low,t_high,
       tree_flag, n, z, m):
70     possible_fw_options = []
71     # the same value of omega may be ok for different values of t
72     good_num_zeroes = 0
73     num_zeroes_step = num_cores
74     for t in range(t_low, t_high+1):
75         # batch-compute for large weight batches stop as soon as
               a batch
76         # finds a reasonable weight
77         # the soundness_error_2(t,w,p,sec_parameter) takes as w
               the weight of the
78         # challenge, therefore the number of rounds where we send
                a seed
79         valid_num_zeroes = 3
80         not_found_t_for_w = True
81         while(not_found_t_for_w and (valid_num_zeroes < (t//2 +
               num_zeroes_step))):
82             num_zeroes_pool = [i for i in range(valid_num_zeroes,
                   min(valid_num_zeroes+num_zeroes_step,t // 2+1))]
83             pooled_param_list = [ [t,t-w,q,sec_parameter] for w in
                    num_zeroes_pool ]
84             #print(pooled_param_list)
85             print(f"running␣t={t},␣wmin:{valid_num_zeroes}␣wmax:{
                   min(valid_num_zeroes+num_zeroes_step,t␣//␣2+1)}")
86             with Pool(num_cores) as p:
87                 concat_results = p.map( soundness_error_2,
                       pooled_param_list )
88
89             for i in concat_results:
90                 # i contains t,w,p,sec_parameter,best_t_star,
                       min_cost_1, min_cost_2, min_cost
91                 #if the t-w pair is ok for security
92                 if (i[7] > sec_parameter):
93                     # if we're still looking for a string with that
                           many zeroes
94                     this_num_zeros = i[0]-i[1]
```

```
                    if (good_num_zeroes != this_num_zeros):
                        is_it_best_option = True
                        #option contains sec_parameter,q,t,
                            num_zeroes,secvalue
                        for option in possible_fw_options:
                            option_num_zeros = option[3]
                            if (option[1] == i[2]) and (
                                option_num_zeros < this_num_zeros):
                                is_it_best_option = False
                        if (is_it_best_option):

                            num_rounds = i[0]
                            val_w = i[1]
                            #compute signature (in kB)
                            if tree_flag == 0:
                                sign_size = 6*sec_parameter +val_w
                                    *3*sec_parameter+(num_rounds-
                                    val_w)*(n*ceil(log2(q))+m*ceil(
                                    log2(z)))
                            else:
                                sign_size = 6*sec_parameter +(
                                    num_rounds-val_w)*sec_parameter*
                                    log2(num_rounds/(num_rounds-
                                    val_w))+2*sec_parameter*(1+(
                                    num_rounds-val_w)*log2(
                                    num_rounds/(num_rounds-val_w)))
                                    +(num_rounds-val_w)*(n*ceil(log2
                                    (q))+m*ceil(log2(z)))
                            possible_fw_options.append( [
                                sec_parameter,q,i[0],i[0]-i[1],i[7],
                                 sign_size] )
                            good_num_zeroes = valid_num_zeroes
                            not_found_t_for_w = False
            valid_num_zeroes = valid_num_zeroes + num_zeroes_step
    return possible_fw_options

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

if __name__=="__main__":
    # check that a security level is provided
    if (len(argv)< 2):
        print("Parallel round number optimizer for CROSS")
        print(f"Usage:{argv[0]} <output_file>")
        exit()

    # open file with code parameters
    try:
```

```
128         results_file = open(argv[1],"w+")
129         csvwriter = csv.writer(results_file)
130     except:
131         print(f"parameter␣file␣{argv[1]}␣not␣found")
132 
133     csvwriter.writerow(["lambda","q","t","num_zeroes","sec_margin
        ","signature_size"])
134     # The optimization is performed over the following list of
        parameter
135     # sets representing lambda, q, n, z, m, t_low,t_high
136     # for R-SDP(G), set m = n
137     sets_to_opt = [
138                    [128, 509, 55, 127, 25, 150, 170]] #here, write
                           the parameters you want to consider
139 #                   [192,509,100,1000],
140 #                   [256,509,100,1000],
141 #                   [128,127,190,270],
142 #                   [128,127,860,895],
143 #                   [192,127,200,350],
144 #                   [192,127,930,952],
145 #                   [256,127,430,455],
146 #                   [256,127,980,1020],
147 #                   [128,509,208,248],
148 #                   [192,509,232,272],
149 #                   [128,509,818,888],
150 #                   [192,509,892,952],
151 #                   [256,509,316,356]
152 #                   [256,509,916,956],
153 
154     is_tree_used = 1 #change this if you want to use trees
155 
156     for code_params in sets_to_opt:
157         security_level= code_params[0]
158         q = int(code_params[1])
159         n = int(code_params[2])
160         z = int(code_params[3])
161         m = int(code_params[4])
162         t_low = int(code_params[5])
163         t_high = int(code_params[6])
164         list_fixed_weight_options = find_fixed_weight_options(
                security_level,q,t_low,t_high, is_tree_used, n, z, m)
165         for fwopt in list_fixed_weight_options:
166             csvwriter.writerow(fwopt)
167         results_file.flush()
168     results_file.close()
```

Listato 4.23: Script relativo al calcolo della coppia t e w negli intervalli desiderati

CROSS_code_round_parameters_combiner.py

```
    #!/usr/bin/python3
# This script takes as input the code parameters for which the ISD complexity
# is high enough to provide the desired security margin.
# The parameters are expected to be in the ./final_code_parameter_sets.csv
# CSV file with header type,lambda,q,z,n,k,m
# Type can either be RSDP or RSDPG
import csv
from math import comb,log2,ceil

def bits_to_represent_value(x):
    return ceil(log2(x+1))

def worst_case_tree_nodes(num_rounds_t,seeds_to_hide):
    num_set_bits_w = num_rounds_t-seeds_to_hide
    if (num_set_bits_w < num_rounds_t//2):
        print("Warning, the constant weight challenge string is too sparse, no gain.")
        return num_rounds_t-num_set_bits_w
    diff = num_rounds_t-num_set_bits_w
    return ceil(diff*log2(num_rounds_t/diff))

################################
# This function computes the size, in bytes of a bitpacked vector of
# vect_len elements, each one able to model any number in [0; max_to_repr]
# This size is obtained considering taking the quotient of the vector
# size in bits (vect_len*bits_to_represent(max_to_repr)) by 8 and rounding up

def bitpacked_vector_size_B(vect_len,max_to_repr):
    res = vect_len*bits_to_represent_value(max_to_repr)/8
    return ceil(res)

def public_key_size_B(n,k,q,prng_seed_size):
    # a seed to expand H, and a syndrome of n-k elements in Z_q
    return prng_seed_size/8 + bitpacked_vector_size_B(n-k,q-1)

def private_key_size_B(n,m,z,prng_seed_size):
    # uncompressed, m elements in [0,z-1]
    if m!=0:
        return bitpacked_vector_size_B(m,z-1)
```

```
39      return bitpacked_vector_size_B(n,z-1)
40
41  def signature_size_B(q,z,n,t,m,seeds_to_hide,prng_seed_size_b,
        hash_digest_size_b):
42      # salt + c + h (bytes)
43      cost = prng_seed_size_b/8 + 2*hash_digest_size_b/8 # 80 bytes
44
45      # Seed tree cost where t-w (b=1) seeds are revealed, w are
            not (b=0)
46      cost += (prng_seed_size_b * worst_case_tree_nodes(t,
            seeds_to_hide))/8
47      # Cost of revealing the missing w = seeds_to_hide commitment
            hashes
48      cost += (hash_digest_size_b * worst_case_tree_nodes(t,
            seeds_to_hide))/8 # Merkle tree non revealed
49      cost += (seeds_to_hide * hash_digest_size_b)/8 # random
            reveals leaves as raw hashes
50      if (m == 0): # RSDP case
51          cost += seeds_to_hide * (bitpacked_vector_size_B(n,q-1)+
                bitpacked_vector_size_B(n,z-1)) # size of actual
                reveals (y,sigma)
52      else :
53          cost += seeds_to_hide * (bitpacked_vector_size_B(n,q-1)+
                bitpacked_vector_size_B(m,z-1)) # size of actual
                reveals (y,sigma)
54      return cost
55
56
57  if __name__=="__main__":
58      # read and materialize the list of code parameters (q,z,n,k,
            and m for RSDPG)
59      # for all security levels. For RSDP, the value m is set to
            zero in the CSV
60      # to distinguish the RDSP case in the script
61      all_code_params = []
62      with open("./final_code_parameter_sets.csv","r") as csvfile:
63          paramreader = csv.reader(csvfile)
64          for row in paramreader:
65              #type,lambda,q,z,n,k,m
66              if row[0] == "type":
67                  continue
68              for i in range(1,7):
69                  row[i] = int(row[i])
70              all_code_params.append(row)
71      # print(all_code_params)
72
```

```
    # glob all the possible t/w pairs providing security against
        Zaverucha's attack
    protocol_params_t_num_zeroes = []
    with open("./fine_tuning_full_range.csv","r") as csvfile:
        paramreader = csv.reader(csvfile)
        for row in paramreader:
            # lambda,q,t,num_zeroes,sec_margin
            if row[0] == "lambda":
                continue
            for i in range(4):
                row[i] = int(row[i])
            row[4] = float(row[4])
            protocol_params_t_num_zeroes.append(row)

    # Compute the sizes for all the possible pairs of code and
        protocol parameters
    for rsdp_par in all_code_params:
        RSDP_type,sec_level_code_param,q,z,n,k,m = rsdp_par
        print(f"combining {rsdp_par}")
        with open(f"{RSDP_type}_{sec_level_code_param}_values.csv
            ", 'a') as csvfile:
            writer = csv.writer(csvfile)
            writer.writerow(["q","z","n","k","m","t","num_zeroes",
                "sec_margin","pk_size_B","uncomp_sk_size_B","
                sig_size_B"])
            for el in protocol_params_t_num_zeroes :
                sec_level_protocol = el[0]
                t,num_zeroes,secmargin = el[2:]
                if sec_level_code_param == sec_level_protocol:
                   #          q,n,z,k,m  + t,num_zeroes,secmargin
                   to_write = [q,n,z,k,m] + [t,num_zeroes,
                       secmargin]
                   pub_size_B = public_key_size_B(n,k,q,
                       sec_level_protocol)
                   pri_size_B = private_key_size_B(n,m,z,
                       sec_level_protocol)

                   sig_size_B = signature_size_B(q,z,n,t,m,
                                          num_zeroes,
                                          sec_level_protocol,
                                          2*sec_level_protocol)
                   to_write = to_write + [pub_size_B,pri_size_B,
                       sig_size_B]
                   writer.writerow(to_write)
```

Listato 4.24: Script relativo alla definizione del set di parametri per CROSS

Riferimenti bibliografici

1. Daniel Kales and Greg Zaverucha. An attack on some signature schemes constructed from five-pass identification schemes. Cryptology ePrint Archive, Paper 2020/837, 2020. `https://eprint.iacr.org/2020/837`.
2. Andreas Hulsing, Joost Rijneveld, Simona Samardjiska, and Peter Schwabe. From 5-pass mq-based identification to mq-based signatures. *IACR Cryptol. ePrint Arch*, 2016:708, 2016.
3. Elwyn Berlekamp, Robert McEliece, and Henk Van Tilborg. On the inherent intractability of certain coding problems. *IEEE Transactions on Information Theory*, 24(3):384–386, 1978.
4. S Barg. Some new NP-complete coding problems. *Problemy Peredachi Informatsii*, 30(3):23–28, 1994.
5. Violetta Weger, Karan Khathuria, Anna-Lena Horlemann, Massimo Battaglioni, Paolo Santini, and Edoardo Persichetti. On the hardness of the Lee syndrome decoding problem. *Advances in Mathematics of Communications*, 2022.
6. Pierre-Louis Cayrel, Pascal Véron, and Sidi Mohamed El Yousfi Alaoui. A zero-knowledge identification scheme based on the q-ary syndrome decoding problem. In *International Workshop on Selected Areas in Cryptography*, pages 171–186. Springer, 2010.
7. Thomas Attema, Ronald Cramer, and Lisa Kohl. A compressed σ-protocol theory for lattices. In *Advances in Cryptology–CRYPTO 2021: 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16–20, 2021, Proceedings, Part II*, pages 549–579. Springer, 2021.
8. Thomas Attema, Serge Fehr, and Michael Klooß. Fiat-Shamir transformation of multi-round interactive proofs. In *Theory of Cryptography: 20th International Conference, TCC 2022, Chicago, IL, USA, November 7–10, 2022, Proceedings, Part I*, pages 113–142. Springer, 2022.
9. Andre Chailloux. On the (in) security of optimized stern-like signature schemes. In *In Proceedings of WCC 2022: The Twelfth International Workshop on Coding and Cryptography*, March 7-11,2022.
10. Marco Baldi, Alessandro Barenghi, Sebastian Bitzer, Patrick Karl, Felice Manganiello, Alessio Pavoni, Gerardo Pelosi, Santini Paolo, Jonas Schupp, Freeman Slaughter, Antonia Wachter-Zeh, and Violetta Weger. Cross codes and restricted objects signature scheme. In *Submission to the NIST Post-Quantum Cryptography Standardization Process*, 2023.
11. The Classic McEliece team. Listing of Information Set Decoding related papers. `https://classic.mceliece.org/papers.html`.
12. Shay Gueron, Edoardo Persichetti, and Paolo Santini. Designing a practical code-based signature scheme from zero-knowledge proofs with trusted setup. *Cryptography*, 6(1):5, 2022.
13. Felice Manganiello and Freeman Slaughter. Generic error SDP and generic error CVE. In Andre Esser and Paolo Santini, editors, *Code-Based Cryptography*, pages 125–143, Cham, 2023. Springer Nature Switzerland.