PROGETTO DI INGEGNERIA INFORMATICA

IMPLEMENTAZIONE DI PRIMITIVE CRITTOGRAFICHE RESISTENTI A CALCOLATORI QUANTISTICI (POST QUANTUM)

Anno Accademico: 2023/2024

**Contemi Agostino Dodi Filippo**

Professore: Barenghi Alessandro

**Indice………………………………………………………………….2**

1. **Introduzione……………………………………………………………..3**
   1. Perché SHA-3 3
   2. Crittografia post quantum 4
   3. Keccak 5
2. **Ottimizzare SHA-3…………………………………………………….10**
   1. Come è possibile ottimizzarlo 10
   2. Architetture e loro differenze 10
   3. AVX 11
   4. NEON 15
3. **Risultati…………………………………………………………...….....18**
   1. Soggetti dei confronti 18
   2. Dispositivi usati 18
   3. Processo di validazione e misura dei risultati 18
   4. Analisi dei risultati 19
4. **Conclusioni……………………………………………………………..22**
   1. Cosa abbiamo imparato 22
   2. Citazioni / Bibliografia 22
5. **Introduzione**
6. **Perché SHA-3** *[1][10][12]*

Le funzioni di hash hanno varie applicazioni: vengono usate nella verifica di firme digitali e dell’autenticità dei messaggi scambiati, nella generazione di numeri pseudo-casuali e in molte altre applicazioni nel mondo della sicurezza.

Prima di SHA-3 venivano usate (e, spesso, vengono usate tuttora) altre famiglie di funzioni di hash, come MD4, MD5, SHA-0, SHA-1 e SHA-2.

La famiglia MD4, introdotta nel 1990, era veloce a calcolare i digest (cioè i risultati dell’hash dei messaggi), ma peccava in sicurezza, al punto che appena 2 anni dopo erano già noti comuni attacchi in grado di decifrare la funzione. Venne quindi introdotta MD5, più sicura e anch’essa molto veloce. Non abbastanza sicura, in realtà: ormai sono noti alcuni attacchi come Den Boer/Bosselaers oppure Dobbertin.

Nel 1993 venne introdotta SHA-0 (Secure Hash Algorithm). L’obiettivo era quello di offrire funzioni più resistenti delle precedenti pur mantenendo un buon tempo di calcolo. Venne ritenuta affidabile fino ai primi attacchi avvenuti nel 1998.

Venne migliorata nel 1995, con l’introduzione di SHA-1, che resistette fino al 2005. Due anni dopo arrivò SHA-2, non ancora attaccata.

Il NIST (National Institute of Standards and Technology), però, spingeva per una versione ancora più sicura e che, grazie ad un’architettura diversa, offrisse flessibilità, affinché vi fosse un’alternativa nel caso in cui SHA-2 venisse attaccata con successo. Così, a novembre 2007, l’istituto annunciò la competizione SHA-3, in modo simile a come avvenuto in precedenza per l’algoritmo AES. La competizione terminò nell’ottobre 2012, vinta da un team composto da Guido Bertoni, Joan Daemen e Gilles Van Assche di STMicroelectronics e Michaël Peeters di NXP Semiconductors, con un algoritmo chiamato Keccak.

La competizione fu vinta da Keccak per diversi motivi:

* Un ampio margine di sicurezza;
* Un design elegante e pulito;
* Buone prestazioni sull’hardware;
* Una logica diversa da SHA-2;

Un’altra caratteristica di Keccak è la sua versatilità. Infatti, esistono più versioni: 512, 384, 256, 224. Ciascuna di esse ha un caso d’uso specifico, poiché man mano che il numero della versione sale aumenta il tempo di calcolo in favore della sicurezza. Peraltro, il codice è semplice da implementare, grazie alla semplicità dell’algoritmo.

Attualmente, SHA-3 è usata in molti dei casi in cui è richiesta una funzione di hashing, nonostante sia ancora molto utilizzata anche SHA-2, in quanto particolarmente più veloce. Inoltre, SHA-2 non è ancora stata attaccata con successo, pertanto per molti sviluppatori non vi è una ragione evidente che giustifichi l’adozione di SHA-3. D’altronde, SHA-3 non è mai stato concepito come sostituto di SHA-2, ma come alternativa all’interno della migrazione da SHA-1 verso alternative più sicure.

1. **Crittografia post quantum**

Un problema comune degli algoritmi di hash è la collisione. Esistono due tipi di collisioni, di stato ed interne: quelle di stato avvengono quando due input differenti producono lo stesso stato interno, mentre quelle interne avvengono quando due stati differenti durante la fase di mescolamento producono lo stesso stato. Di conseguenza, a parità di stato, l’output è il medesimo. E’ quindi possibile, partendo dall’hash, ritrovare il messaggio originale, oppure sostituire un messaggio con un altro avente lo stesso hash.

Le funzioni spugna (un sottogruppo delle funzioni di hash di cui fa parte SHA-3) sono progettate per far sì che il modo più efficiente di attaccarle sia il brute-force (eliminando quindi attacchi più veloci).

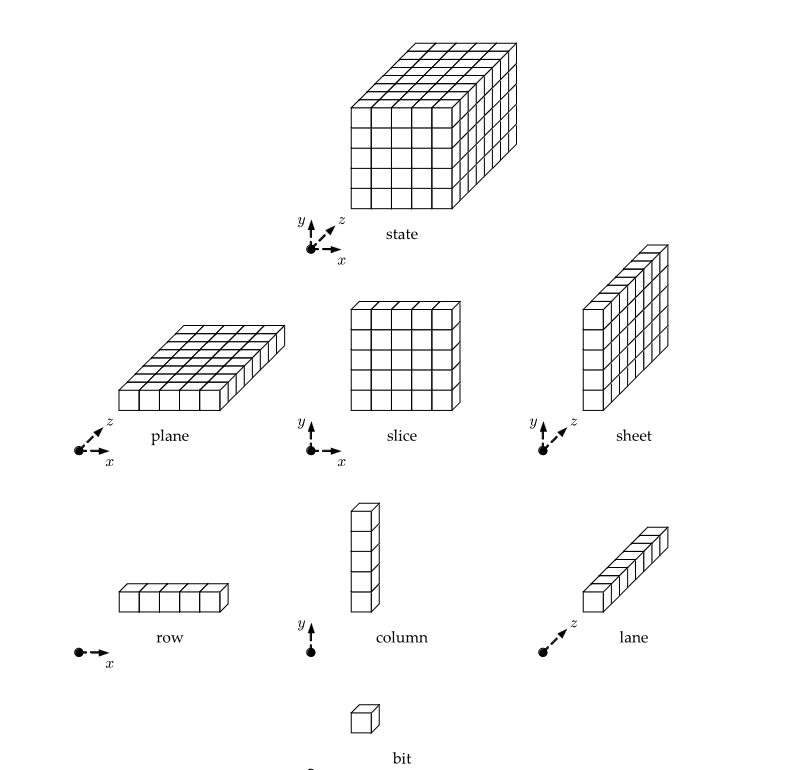
Un computer quantistico, tuttavia, grazie alla sua capacità di eseguire calcoli in parallelo, può invertire velocemente le funzioni di hash. Infatti, può utilizzare algoritmi che riducono significativamente il numero di operazioni necessarie rispetto ai computer classici (ad esempio, quello di Grover). Pertanto, dato un output di lunghezza d, per un computer classico la complessità temporale di decifrazione è mentre per un computer quantistico è .

Una soluzione, banale, a tale problema è quella di aumentare la lunghezza dell’output. Ciò però implica maggiori tempi di esecuzione dell’algoritmo, portando alla necessità di ottimizzarlo.

1. **Keccak** *[2][3][6][7][8][9]*

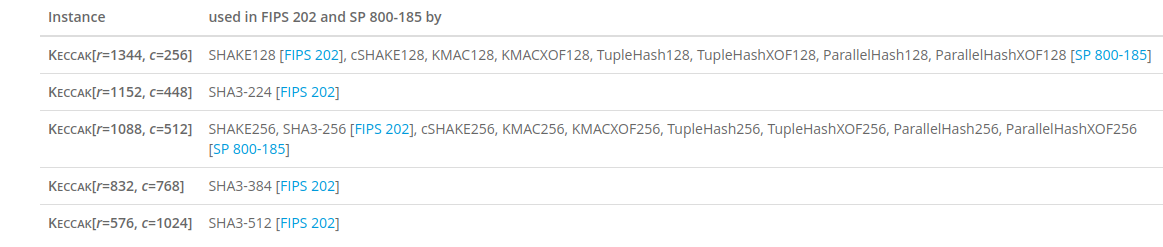
Gli algoritmi di SHA-3 utilizzano le funzioni spugna. Le funzioni spugna sono come dei generatori di numeri pseudo casuali: “assorbono” un input di lunghezza variabile nello stato, che poi può essere “spremuto” per generare un output di lunghezza variabile. Non ci sono limiti alla lunghezza dell’input e dell’output, infatti questi possono teoricamente essere infiniti.

Lo stato è una matrice 5x5 di cui ogni elemento è lungo 64 bit. In totale, quindi, è composto da 1600 bit. Viene inizializzato con tutti i bit a 0.



[Figura 1] Rappresentazione grafica dello stato in Keccak

Durante la fase di assorbimento l’input viene assorbito in blocchi, la cui grandezza è definita “rate” e varia in base alla funzione di hash scelta. L’assorbimento di un blocco avviene tramite una operazione di XOR tra il blocco e una porzione di stato, grande quanto il rate. La dimensione del resto dello stato, invece, è definita “capacità”, ed è la parte che, non essendo modificata nell’assorbimento dei singoli blocchi, è “nascosta” all’esterno. Perciò, maggiore è la capacità (e minore il rate), maggiore è la sicurezza (e minore la velocità, in quanto è necessario assorbire più blocchi).

[Figura 2] Rate e le capacità (in bit) delle versioni di SHA e varianti

Tra l’assorbimento di un blocco e l’altro avviene una permutazione di tutto lo stato, composta da 24 round di 5 passaggi consecutivi. Con ciascun bit dello stato nella riga i, colonna j e posizione nella parola k, indicato come a[i][j][k], i passaggi di un round sono:

* θ :
* ρ : , dove RC è un’opportuna costante di rotazione, che dipende da x e y
* π : , con x = y’ e y = 2x’ + 3y’
* χ :
* ι : , dove RC è la costante del round, che cambia ad ogni round

**Passaggio θ:** A ogni bit dello stato viene aggiunta la somma delle parità della colonna precedente e della colonna successiva ruotata a sinistra di un bit. Segue una rappresentazione in pseudocodice del passaggio (ROTL rappresenta la rotazione di una parola da 64 bit a sinistra):

for x in 0…4:

for x in 0…4:

for x in 0…4:

for y in 0…4:

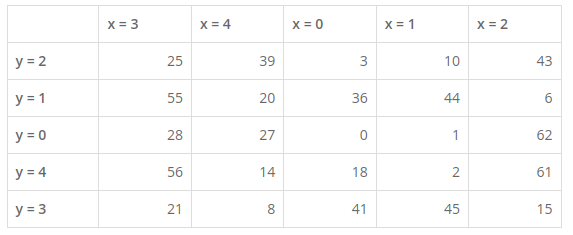
**Passaggio ρ:** Ogni parola dello stato viene ruotata a sinistra della propria opportuna costante di rotazione. Vi sono 24 costanti, una per parola. Lo pseudocodice corrispondente è:

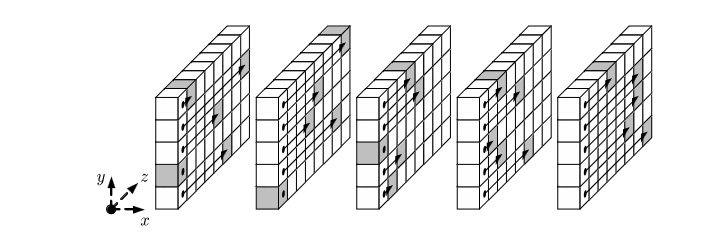
for x in 0…4:

for y in 0…4:

a[x][y] = ROTL(a[x][y], RC[x][y])

con RC presa dalla seguente tabella:

[Tabella 1] Valori delle Round Constants

[Figura 3] Una rappresentazione grafica del passaggio ρ

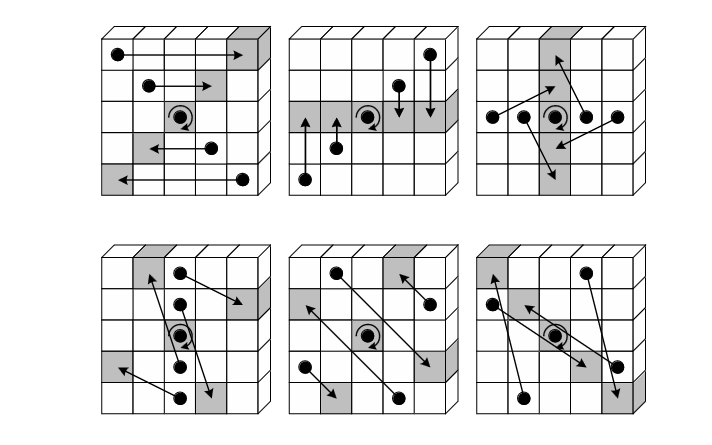
**Passaggio π:** Le parole dello stato vengono permutate tra di loro secondo la formula

Lo pseudocodice è:

for x in 0…4:

for y in 0…4:

B[y][2\*x+3\*y] = a[x][y]



[Figura 4] Una rappresentazione grafica del passaggio π

**Passaggio χ:** a ogni parola viene aggiunto il risultato di , dove è la parola corrispondente alla riga successiva e quella due righe dopo.

Lo pseudocodice è:

for x in 0…4:

for y in 0…4:

**Passaggio ι:** infine, alla prima parola dello stato viene aggiunta la costante del round attuale.

Dopo l’assorbimento dell’input, vi è la fase di squeezing, in cui si genera l’output: inizialmente avviene una permutazione, dopodiché viene mandato in output un certo numero di bit, che dipende dalla funzione utilizzata (nel caso di SHA3-512 512 bit, nel caso di SHA3-384 384 bit, etc.).

Esiste poi una variante di SHA-3, chiamata SHAKE. Questa è una XOF (extendable output function), ovvero una funzione la cui lunghezza di output è arbitraria. In questo caso, l’output viene generato a blocchi di dimensione pari al rate: per ogni blocco, prima viene permutato lo stato, poi vengono caricati sul blocco i primi bit dello stato, tanti quanti il rate.

1. **Ottimizzare SHA-3**
2. **Com’è possibile ottimizzarlo**

Ѐ possibile, in seguito ad un’analisi generale del codice, trovare sequenze di istruzioni superflue. In alcune implementazioni, infatti, si nota che lo stato viene caricato e scaricato dai registri più frequentemente del necessario. Altre situazioni di semplice ottimizzazione si verificano quando vengono lasciati risolvere al computer dei calcoli il quale valore non cambia e che potrebbero essere agevolmente precalcolati. Anche prestando attenzione all’ordine delle operazioni è possibile aumentare la velocità di esecuzione usando maggiormente i valori calcolati nei registri prima di scaricarli in memoria per usi futuri. Un’altra idea per migliorare l’algoritmo è quella di eseguire più operazioni allo stesso tempo. Infatti, se il processore è dotato di più cores, può eseguire più operazioni contemporaneamente, impiegando molto meno tempo in totale.

Facciamo un esempio semplice di parallelizzazione delle operazioni: abbiamo una lista di numeri interi e vogliamo trovare la metà di ciascun numero.

Un approccio sequenziale sarebbe del tipo:

for each element in array:

array[index of loop] = element / 2

La complessità temporale di questo algoritmo è O(n).

Parallelizzando le operazioni è possibile eseguire in O(n/k), dove k è la quantità di elementi che si possono elaborare in parallelo, avendo un algoritmo del tipo:

for each vector\_of\_element in array:

vectorized\_shift\_right\_logical(vector\_of\_element, 1)

1. **Architetture e le loro differenze**

Esistono due famiglie principali di architetture dei processori: x86\_64 e ARM. La prima è generalmente più prestante a discapito di consumi energetici maggiori, con conseguente bisogno di sistemi di raffreddamento migliori, mentre la seconda trova il suo punto di forza nell’efficienza energetica, per questo è più prevalente su dispositivi con batterie più piccole come per esempio gli smartphone.

Entrambe le architetture però consentono di eseguire istruzioni vettorizzate dette SIMD (Single Instruction Multiple Data). Infatti dispongono di appositi registri di dimensioni maggiori e unità di calcolo apposite. Per usare questi registri si usano delle speciali funzioni dette intrinsics.

**c. AVX** *[4]*

x86\_64 utilizza registri da 128 bit detti XMM e registri da 256 bit detti YMM. Si può poi scegliere quanti elementi mettere in un singolo registro (per esempio 4 da 64 o 8 da 32 per i registri YMM). Ci sono varie famiglie di istruzioni apposite, tra cui AVX, AVX2 e AVX512. Per utilizzare le istruzioni corrispondenti in C è necessario importare la libreria immintrin.h e abilitare le flag del compilatore -mavx, -mavx2 e -mavx512.

I registri da 256 bit vengono rappresentati tramite delle variabili di tipo \_\_m256 e dalle varianti con numeri interi o a precisione doppia \_\_m256i e \_\_m256d.

**Instrisics usate con una breve documentazione:**

void \_mm256\_storeu\_si256(\_\_m256i \* mem\_addr, \_\_m256i a)

Salva il valore di a in mem\_addr

\_\_m256i \_mm256\_loadu\_si256(\_\_m256i const \* mem\_addr)

Carica 256 bit da mem\_addr e ritorna il valore

\_\_m256i \_mm256\_xor\_si256(\_\_m256i a, \_\_m256i b)

Esegue lo XOR bit a bit di a e b e ritorna il valore

\_\_m256i \_mm256\_unpackhi\_epi64(\_\_m256i a, \_\_m256i b)

Ritorna un vettore costruito in questo modo: [a[0], b[0], a[2], b[2]], dove ogni elemento è grande 64 bit.

\_\_m256i \_mm256\_unpcklo\_epi64(\_\_m256i a, \_\_m256i b)

Ritorna un vettore costruito in questo modo: [a[1], b[1], a[3], b[3]], dove ogni elemento è grande 64 bit.

\_\_mm256i \_mm256\_permute4x64\_epi64(\_mm256i a, const int imm8)

Divide a in sottogruppi da 64 bit e li mescola secondo la maschera in imm8, infine ritorna il valore

\_\_mm256i \_mm256\_slli\_epi64(\_\_m256i a, int imm8)

Esegue uno shift sinistro logico dei sottogruppi da 64 bit di a di imm8 posizioni

\_\_m256i \_mm256\_srli\_epi64(\_\_m256i a, int imm8)

Esegue uno shift destro logico dei sottogruppi da 64 bit di a di imm8 posizioni

\_\_mm256i \_mm256\_broadcastq\_epi64(\_\_m128i a)

Mette i 64 bit inferiori di a in tutti i posti del valore ritornato

\_\_mm256i \_mm256\_sllv\_epi64(\_\_mm256i a, \_\_m256i count)

Esegue uno shift sinistro logico dei sottogruppi da 64 bit dei valori nei rispettivi sottogruppi di count

\_\_m256i \_mm256\_srlv\_epi64(\_\_mm256i a, \_\_m256i count)

Esegue uno shift destro logico dei sottogruppi da 64 bit dei valori nei rispettivi sottogruppi di count

\_\_m256i \_mm256\_castsi256\_si128(\_\_mm256i a)

Esegue un cast da \_\_m256i a \_\_m128i. Non genera codice in compilazione

\_\_m256i \_mm256\_blend\_epi32(\_\_m256i a, \_\_m256i b, const int imm8)

Divide a e b in sottogruppi da 32 bit e li mescola seguendo la maschera definita da imm8, infine ritorna il valore

\_\_m256i \_mm256\_permute2x128\_si256(\_\_m256i a, \_\_m256i b, const imm8)

Divide a e b in sottogruppi da 128 bit e li mescola seguendo la maschera definita da imm8, infine ritorna il valore

\_\_m256i \_mm256\_andnot\_si256(\_\_m256i a, \_\_m256i b)

Esegue il bit a bit not di a e dopo esegue il bit a bit del risultato con b, infine ritorna il valore

**Gestione dello stato:**

Lo stato viene memorizzato in questo modo:

Le prime 4 parole (64 bit ciascuna) della prima riga vengono memorizzate nella variabile \_\_m256i a0.

Le prime 4 parole (64 bit ciascuna) della seconda riga vengono memorizzate nella variabile \_\_m256i a1.

Le prime 4 parole (64 bit ciascuna) della terza riga vengono memorizzate nella variabile \_\_m256i a2.

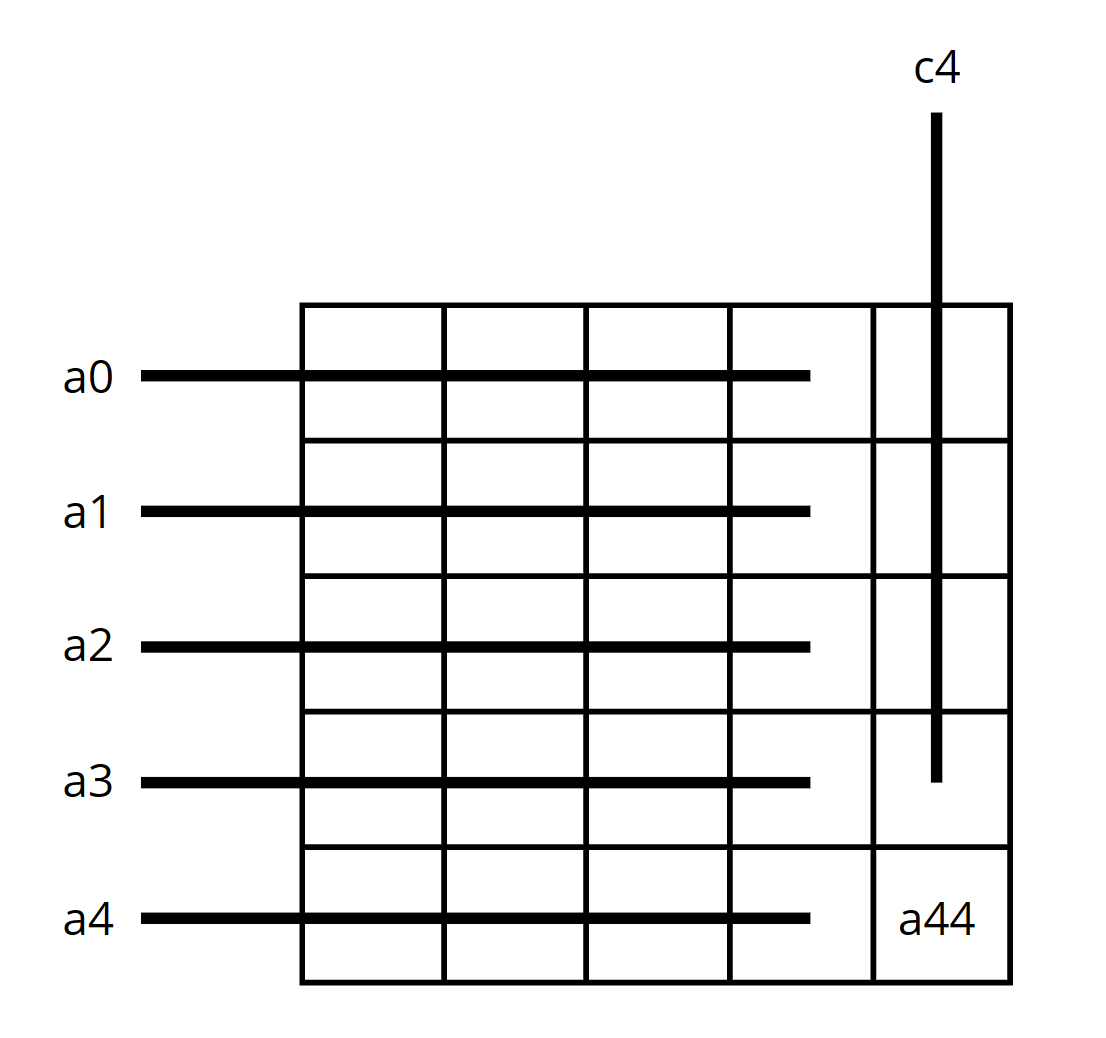
Le prime 4 parole (64 bit ciascuna) della quarta riga vengono memorizzate nella variabile \_\_m256i a3.

Le prime 4 parole (64 bit ciascuna) della quinta riga vengono memorizzate nella variabile \_\_m256i a4.

Le ultime parole delle prime quattro righe vengono memorizzate nella variabile \_\_m256i c4.

Infine l’ultimo elemento viene salvato nella variabile \_\_m256i a44.

Quest’ultima variabile da 256 bit è necessaria anche se rappresenta un valore da solo 64 bit per poter essere utilizzata in operazioni vettorizzate



[Figura 5] Rappresentazione grafica dell’organizzazione dello stato nell’implementazione AVX

**Ottimizzazioni effettuate**

**Ottimizzazione sulle costanti:**

Una prima ottimizzazione effettuata è stata quella di rendere globali le round constants, in modo da ridurre il tempo utilizzato per inizializzarle ogni volta che avvengono le permutazioni. Si sono inoltre creati degli \_\_m256i, ciascuno contenente 4 costanti di rotazione, in modo da vettorizzare istruzioni successive. Anch’essi sono stati dichiarati globali e vengono caricati nella fase di inizializzazione dello stato (evitando quindi operazioni ripetitive che si avrebbero ricaricandoli ad ogni permutazione).

**La StateXORBytes:**

E’ stata ottimizzata la funzione StateXORBytes (quella che assorbe i blocchi di input) tramite la vettorizzazione. Infatti, grazie alle funzioni \_mm256\_loadu\_si256, \_mm256\_xor\_si256 e \_mm256\_storeu\_si256 è possibile assorbire 256 bit in una singola istruzione, anziché 64.

In particolare, la funzione è stata ottimizzata grazie al seguente codice:

while (length > 32) {

\_\_m256i stateVec = \_mm256\_loadu\_si256((\_\_m256i \*) firstStateWord);

\_\_m256i dataVec = \_mm256\_loadu\_si256((\_\_m256i \*) dataWords);

stateVec = \_mm256\_xor\_si256(stateVec, dataVec);

\_mm256\_storeu\_si256((\_\_m256i \*) firstStateWord, stateVec);

firstStateWord += 4;

dataWords += 4;

length -= 32;

}

**La StateExtractBytes:**

Anche la funzione StateExtractBytes, utilizzata per estrarre lo stato nell’output, ma anche per ottenere “stateVec”, utilizzato nella StateXORBytes, è stata ottimizzata tramite vettorizzazioni. Infatti, permette di caricare nell’indirizzo di destinazione tutte le 25 parole dello stato con sole 5 \_mm256\_storeu\_si256, invece che 25 istruzioni che si avrebbero lavorando con variabili da 64 bit.

**La StatePermute:**

La funzione StatePermute, usata per permutare lo stato, è stata anch’essa ottimizzata. In particolare, all’inizio della funzione lo stato viene caricato su vari \_\_m256i, utilizzati per eseguire operazioni vettoriali in seguito.

Nel passaggio *θ*, ad esempio, vengono utilizzate le \_mm256\_xor\_si256 per calcolare le prime 4 C[x] in una volta, così come poi verranno ruotate tutte e 4 in una volta per calcolare i D[x], grazie alle intrinsics \_mm256\_slli\_epi64, \_mm256\_srli\_epi64 e \_mm256\_xor\_si256, e ancora di nuovo per aggiungere il risultato ai vettori contenenti lo stato.

Nel passaggio *ρ* devono essere eseguite le rotazioni delle parole dello stato. Tuttavia,non esistono intrinsics specifiche per ruotare le parole in un \_\_m256i, pertanto è necessario, per effettuare ROTL(a, i), eseguire \_mm256\_xor\_si256(\_mm256\_slli\_epi64(a, i), \_mm256\_srli\_epi64(a, 64 - i)), cosa che, scomposta in più operazioni e riorganizzate per ottimizzare il codice, accade nella nostra implementazione. Usare queste intrinsics permette di ruotare 4 parole alla volta.

Nel passaggio *π* utilizziamo le intrinsics \_mm256\_permute4x64\_epi64, \_mm256\_permute2x128\_si256, \_mm256\_blend\_epi32, \_mm256\_unpackhi\_epi64 e \_mm256\_broadcastq\_epi64 per effettuare le corrette permutazioni fra le parole dello stato.

Il passaggio *χ* utilizza le intrinsics \_mm256\_andnot\_si256 e \_mm256\_xor\_si256 per calcolare con solo 2 operazioni 4 valori risultanti.

**Il Ciclo Accelerato di Assorbimento:**

Un’altra ottimizzazione effettuata è all’interno della fase di assorbimento della nostra implementazione. Questa consiste in un loop accelerato di assorbimento, che consente di caricare e salvare lo stato una volta sola all’interno di tutto l’assorbimento, ottenendo così permutazioni e assorbimenti di blocchi più veloci.

**d. NEON** *[5]*

Le famiglie di intrinsics di ARM si chiamano NEON ed HELIUM. La differenza tra le due sta nella dimensione dei registri: la prima lavora con registri di dimensione massima a 128 bit, ed è la famiglia usata nella nostra implementazione.

Le variabili seguono una nomenclatura più simile a quella della libreria stdint.h.

Troviamo, ad esempio, i tipi uint64x2\_t che denotano un registro da 128 bit composto da due numeri interi senza segno. Definisce, inoltre, delle strutture come uint64x2x4\_t, ovvero un vettore da 4 elementi da 128 bit ciascuno. Si può accedere ad ogni elemento con .val[index], dove index è l’indice dell’elemento.

**Intrinsics usate con una breve documentazione:**

void vst4q\_u64(uint64\_t \* ptr, uint64x2x4\_t val)

Pone gli elementi in val nella posizione ptr in memoria. L’azione avviene interlacciando gli elementi

void vst1q\_u64(uint64\_t \* ptr, uint64x2x4\_t val)

Pone gli elementi in val nella posizione ptr in memoria. L’azione avviene senza interlacciamento

uint64x2\_t vld1q\_u64(uint64\_t const \* ptr)

Carica 128 bit dalla posizione ptr in memoria e ritorna il valore in un uint64x2\_t, senza interlacciare

uint64x2\_t veorq\_u64(uint64x2\_t a, uint64x2\_t b)

Esegue lo xor bit a bit di a e b, infine ritorna il valore

**La gestione dello stato:**

Tenendo in considerazione la forma dello stato, una matrice 5x5 di parole da 64 bit ciascuna, leggendolo come un vettore, questo viene salvato secondo la seguente formattazione:

uint64x2\_t a04 = stato[0] ## stato[4]

uint64x2\_t a15 = stato[1] ## stato[5]

uint64x2\_t a26 = stato[2] ## stato[6]

uint64x2\_t a37 = stato[3] ## stato[7]

uint64x2\_t a812 = stato[8] ## stato[12]

uint64x2\_t a913 = stato[9] ## stato[13]

uint64x2\_t a1014 = stato[10] ## stato[14]

uint64x2\_t a1115 = stato[11] ## stato[15]

uint64x2\_t a1620 = stato[16] ## stato[20]

uint64x2\_t a1721 = stato[17] ## stato[21]

uint64x2\_t a1822 = stato[18] ## stato[22]

uint64x2\_t a1923 = stato[19] ## stato[23]

uint64x2\_t a24 = stato[24] ## 0

Dove a##b indica che gli elementi a e b sono posti adiacenti in memoria

La peculiare disposizione degli elementi è necessaria al fine di mantenere una maggiore velocità tramite uno specifico interlacciamento, utilizzato dalle intrinsics ARM, che pertanto consente di eseguire in maniera più efficiente le operazioni.

Durante l’esecuzione della funzione StatePermute abbiamo notato che le implementazioni che usano le intrinsics NEON rallentano il programma. Per questo motivo su ARM la StatePermute non utilizza intrinsics.

**StateXORBytes:**

In modo simile a come avviene nell’implementazione AVX, la StateXORBytes viene ottimizzata vettorizzando il caricamento, l’esecuzione della xor e il salvataggio dello stato utilizzando degli uint64x2\_t e le istruzioni vld1q\_u64, veorq\_u64, vst1q\_u64, e sfruttando l’interlacciamento dello stato. L’ottimizzazione non è al pari di quella AVX, in quanto in questo caso si stanno utilizzando variabili di 128 e non 256 bit.

**StateExtractBytes:**

Questa funzione utilizza le intrinsics vst4q\_u64 per estrarre 4 elementi da 128 bit in una destinazione, sterlacciandoli. Le istruzioni vst4q\_u64 permettono di ottimizzare il codice di estrazione, che, in caso contrario, estrarrebbe solamente 64 bit alla volta.

Per poter utilizzare queste intrinsics e quindi vettorizzare il codice, è stato necessario salvare lo stato in modo interlacciato. La funzione StateXORBytes, inizialmente sterlaccia lo stato (chiamando la funzione StateExtractBytes). Dopodiché esegue le XOR, infine salva lo stato in modo interlacciato. Il vantaggio dell’interlacciamento consiste nell’ottimizzazione del salvataggio dello stato alla fine della StateXORByte. Diventa possibile, così, salvare 128 bit alla volta anziché 64.

1. **Risultati**
2. **Soggetti del confronto**

I risultati ottenuti dalle nostre implementazioni sono stati confrontati con altre 2 implementazioni: SUPERCOP e CROSS. La prima raccoglie implementazioni vettorizzate per x86 e ARM, mentre la seconda è una versione non ottimizzata in grado di girare su tutti i dispositivi, sviluppata dall’omonimo gruppo tra cui è presente il professor Barenghi.

Ci riferimemo alle nostre implementazioni come ARM o ARM IMPL e AVX o AVX IMPL.

Ci riferiremo alle implementazioni SUPERCOP come SUPERCOP ARM e SUPERCOP AVX.

1. **Dispositivi usati**

I risultati sono presi dalle esecuzioni sulle nostre macchine.

Per l'architettura x86 abbiamo usato un laptop dotato di i5 1135g7 Tigerlake.

Per l’architettura ARM invece abbiamo usato una Raspberry Pi 4.

In compilazione vengono sempre utilizzate le flag che permettono al compilatore di ottimizzare il più possibile, ovvero:

* Per ARM si usano -O3 e -mcpu=native
* Per AVX si usano -O3, -march=native e -mtune=native

In particolare, le flag -mpcu, -march e -mtune consentono di ottimizzare utilizzando le istruzioni SIMD.

1. **Processo di validazione e misura dei risultati**

Ci siamo assicurati della correttezza della nostra implementazione tramite delle serie di test con input di dimensione variabile. Abbiamo eseguito il nostro algoritmo con tali input e abbiamo verificato che i digest corrispondessero a quelli presenti nei file di test.

In seguito, abbiamo misurato i tempi di esecuzione della nostra implementazione con messaggi di 4kB di lunghezza. La misura dei tempi è indicata come cicli/byte, ovvero quanti cicli di clock servono per elaborare un byte di input. Per misurare i tempi, nelle architetture x86 è stata usata l’istruzione assembly rdtscp, mentre in quelle ARM è stata usata cntvct\_el0. In questo caso la specifica ARM non assicura che il clock che aggiorna il registro abbia la stessa frequenza del clock principale del processore, perciò è stato necessario riscalare i valori ottenuti per per renderli coerenti.

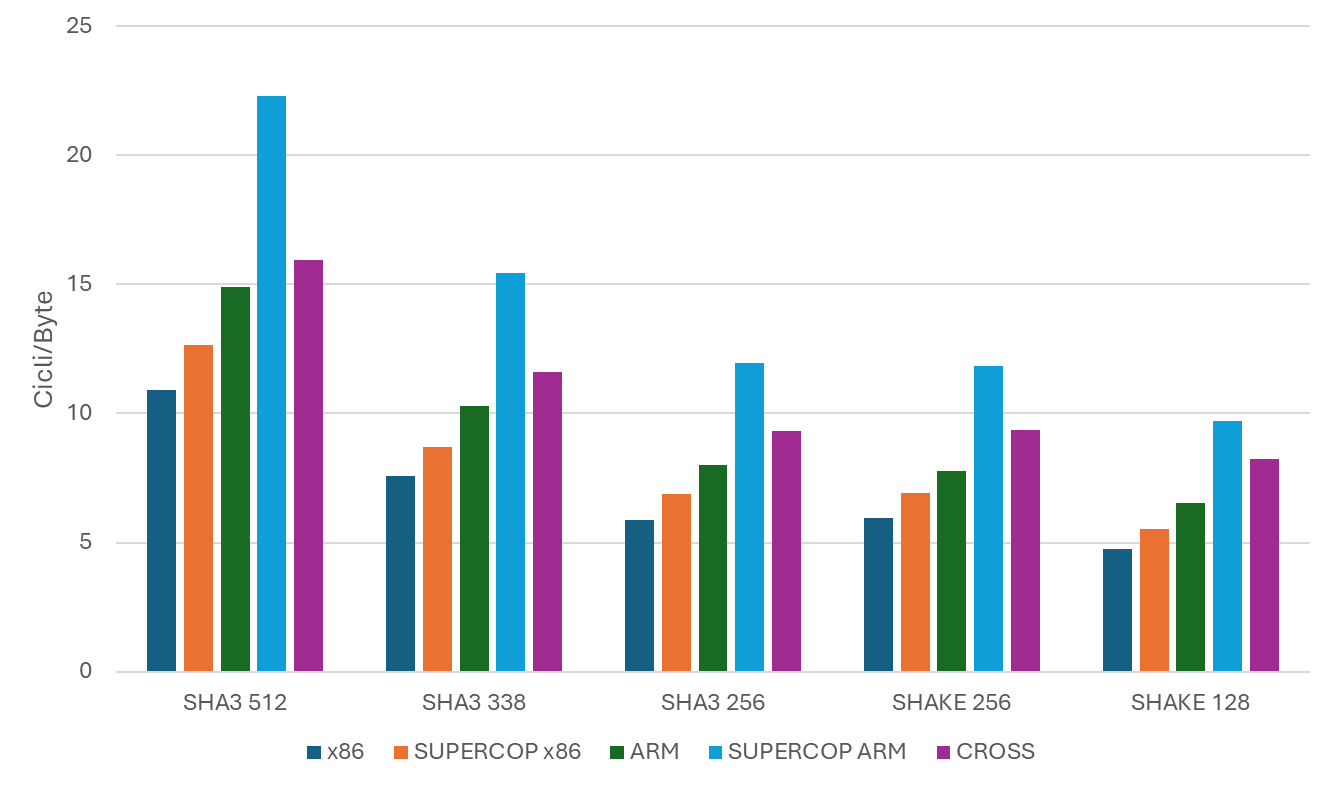
Poiché sulle macchine oltre alla implementazione girano anche altri programmi abbiamo tenuto in considerazione che i cicli di clock variano anche se si esegue la stessa funzione più volte. Per questo abbiamo usato l’algoritmo di Welford (fornitoci dal professore) per calcolare i valori medi. Abbiamo eseguito i test fino ad ottenere una varianza inferiore ad 0,02 cicli di clock.

1. **Analisi dei risultati**

Di seguito una tabella riassuntiva dei dati raccolti

|  | SHA3 512 | SHA3 384 | SHA3 256 | SHAKE 256 | SHAKE 128 |
| --- | --- | --- | --- | --- | --- |
| ARM IMPL | 14,90696 | 10,2816 | 8,0083 | 7,7896 | 6,5231 |
| SUPERCOP ARM | 22,2836 | 15,4224 | 11,9369 | 11,8342 | 9,7088 |
| AVX IMPL | 10,89 | 7,57 | 5,86 | 5,95 | 4,77 |
| SUPERCOP AVX | 12,64 | 8,72 | 6,88 | 6,91 | 5,54 |
| CROSS | 15,95 | 11,59 | 9,34 | 9,36 | 8,22 |

[Tabella 2] I numeri indicano i cicli/byte per ogni implementazione nelle varie versioni



[Figura 6] Istogramma dei dati presenti in [Tabella 2]

Dai risultati mostrati si può notare che la nostra implementazione AVX è la più veloce, essendo in media il 16% più veloce rispetto a SUPERCOP AVX, quest’ultima notevolmente migliore di CROSS.

Ciò è coerente con quanto atteso in quanto CROSS non è ottimizzato; inoltre le implementazioni AVX sono più veloci di quelle ARM, utilizzando variabili di grandezza maggiore.

Si nota anche come detto in precedenza che l’implementazione di SUPERCOP ARM è più lenta rispetto alla CROSS. Infatti utilizza istruzioni SIMD per eseguire la permutazione, cosa che abbiamo notato essere più lenta rispetto ad una più semplice implementazione.

Vogliamo portare all’attenzione del lettore la differenza di velocità tra le diverse implementazioni ARM (in percentuale), la quale non è costante. Infatti, nel caso della differenza tra SUPERCOP e CROSS questa tende a diminuire man mano che aumenta il rate Questo perché la permutazione è più lenta in SUPERCOP e con rate maggiore si hanno meno permutazioni in totale.

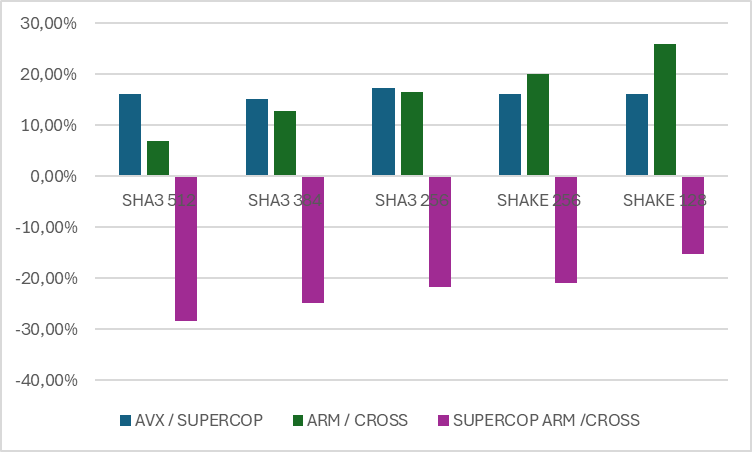
Nel caso invece della differenza tra la nostra implementazione e CROSS essa tende ad aumentare al crescere del rate, infatti si tengono costanti i tempi di permutazione in favore delle altre fasi, che diventano più predominanti sul tempo totale al crescere del rate.

Proprio per questo motivo, i tempi associati a rate maggiori sono sempre minori.

Di seguito troviamo le variazioni di velocità tra le varie implementazioni per ogni algoritmo:

|  | SHA3 512 | SHA3 384 | SHA3 256 | SHAKE 256 | SHAKE 128 |
| --- | --- | --- | --- | --- | --- |
| AVX / SUPERCOP | 16,07% | 15,19% | 17,41% | 16,13% | 16,14% |
| ARM / CROSS | 7,00% | 12,73% | 16,63% | 20,16% | 26,01% |
| SUPERCOP ARM /CROSS | -28,42% | -24,85% | -21,76% | -20,91% | -15,33% |

[Tabella 3] Variazione di velocità in percentuale tra diverse implementazioni



[Figura 7] Istogramma dei dati presenti in [Tabella 3]

1. **Conclusioni**

In sintesi, abbiamo realizzato delle implementazioni per SHA-3 particolarmente ottimizzate, in grado di superare la performance di quelle testate in SUPERCOP. Abbiamo realizzato ottimizzazioni di diversi tipi, vettorizzando operazioni come XOR, permutazioni e caricamento/salvataggio dati, ma anche ottimizzando l’ordine delle istruzioni ed eliminando sequenze superflue di operazioni. Implementazioni ottimizzate di SHA-3 permettono il calcolo con rapidità di digest che siano sicuri ad attacchi anche provenienti da calcolatori quantistici. Seppure non sia possibile tramite implementazioni software ottenere ottimizzazioni pari a quelle raggiunte da dispositivi hardware appositi (realizzati, ad esempio, su FPGA), riteniamo soddisfacente l’ottimizzazione software in grado di raggiungere tempistiche migliori a quelle ottenute con le implementazioni di SUPERCOP.

1. **Cosa abbiamo imparato**

Questo progetto ci è stato molto utile nell’apprendimento dell’ottimizzazione di codice a basso livello, fornendoci importanti competenze. In particolare, abbiamo ottenuto esperienza nell’utilizzo delle intrinsics in C al fine di migliorare notevolmente le prestazioni algoritmiche tramite la vettorizzazione. Inoltre, siamo giunti ad una conoscenza più approfondita del funzionamento del compilatore, necessaria per comprendere i meccanismi di ottimizzazione da adoperare. Infine abbiamo appreso tecniche di analisi e misurazione delle prestazioni di codice C e assembly, utili ad analizzare le implementazioni di riferimento in modo da coglierne i tratti migliori e integrarli all’interno del nostro algoritmo.

1. **Citazioni / Bibliografia**

*[1] SHA3: Past, Present and Future*, https://csrc.nist.gov/CSRC/media/Projects/Hash-Functions/documents/kelsey\_ches2013\_presentation.pdf. Accessed 10 July 2024.

*[2]* Bertoni, Guido. “Keccak Team.” *Keccak Team*, https://keccak.team/keccak.html. Accessed 10 July 2024.

*[3]* “FIPS 197, Advanced Encryption Standard (AES).” *NIST Technical Series Publications*, 26 November 2001, https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf. Accessed 10 July 2024.

*[4]* “Intel® Intrinsics Guide.” *Intel*, https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html. Accessed 10 July 2024.

*[5]* “Intrinsics – Arm Developer.” *Arm Developer*, https://developer.arm.com/architectures/instruction-sets/intrinsics/. Accessed 10 July 2024.

*[6]* “Keccak Team.” *Keccak Team*, 12 June 2017, https://keccak.team/2017/is\_sha3\_slow.html. Accessed 10 July 2024.

*[7]* “Keccak Team.” *Keccak Team*, https://keccak.team/sponge\_duplex.html. Accessed 10 July 2024.

*[8]* “Keccak Team.” *Keccak Team*, https://keccak.team/keccak\_specs\_summary.html. Accessed 10 July 2024.

*[9]* “The K reference.” *Keccak Team*, 14 January 2011, https://keccak.team/files/Keccak-reference-3.0.pdf. Accessed 10 July 2024.

*[10]* “Measurements of SHA-3 finalists.” *eBACS (ECRYPT Benchmarking of Cryptographic Systems)*, http://bench.cr.yp.to/results-sha3.html. Accessed 10 July 2024.

*[11]* “NIST Selects Winner of Secure Hash Algorithm (SHA-3) Competition | NIST.” *National Institute of Standards and Technology*, 2 October 2012, https://www.nist.gov/news-events/news/2012/10/nist-selects-winner-secure-hash-algorithm-sha-3-competition. Accessed 10 July 2024.

*[12]* “Sponge Functions.” *Keccak Team*, https://keccak.team/files/SpongeFunctions.pdf. Accessed 10 July 2024.