Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

ENERGY AND MEMORY CONSUMPTION OF SHA256 ALGORITHM

Relatore:
Prof.
Ivan Lanese

Presentata da:
Nicolò Dentale

Sessione
2021/2022

Ai miei familiari,
Ai miei affetti,
Ai miei amici e
Ai miei colleghi
Che mi sono stati vicini in questo percorso

Abstract

L'efficienza dei computer ha un limite inferiore dettato dal principio di Landauer. La distruzione di una qualsiasi informazione ha un costo energetico per la dissipazione dei bit che la formavano. L'unico modo per aggirare il principio di Landauer è attraverso la reversibilità. Questa tecnica di computazione ci permette di eseguire un programma senza dover distruggere informazioni e quindi, senza dissipare bit. Molti algoritmi ai giorni nostri hanno un grande impatto energetico sul mondo, ed uno di questi è SHA256, l'algoritmo usato nella block-chain di Bitcoin. Questa tesi si pone l'obbiettivo di analizzare il consumo energetico e di memoria di una implementazione di SHA256 reversibile ideale.

Introduzione

Secondo alcuni studi recenti l'efficienza energetica delle CPU, dal 1946 al 2009, è raddoppiata ogni 1,57 anni [1]. Questo incremento, tuttavia, è destinato ad arrestarsi a causa di leggi della fisica. Secondo il principio di Landauer, infatti, dissipare un bit di informazione richiede $kT \ln 2$ volt di energia, dove K è la costante di Boltzmann e T è la temperatura ambiente [5]. Dissipare 1 bit a temperatura ambiente di 20 gradi dovrebbe costare circa $2.8 * 10^{-21} joules$ o $7.8 * 10^{-28}$ kilowattora. La reversibilità è l'unico modo per aggirare il limite di Landauer. La reversibilità è un modello di computazione dove il processo in atto è invertibile nel tempo. Una funzione reversibile, preso un suo output, ci permette di ricostruirne sempre l'input. Grazie all'aumento dell'efficienza energetica il costo delle singole operazioni è sempre andato calando. Secondo il principio di Landauer si arriverà ad un momento nel quale il costo della dissipazione di informazione sarà una buona parte del costo dell'esecuzione di una computazione, creando così un limite all'efficienza energetica delle operazioni calcolabili. In un futuro prossimo una macchina reversibile, con la diminuizione dei costi energetici per singola operazione, potrebbe portare ad un risparmio di energia fino al 24% sull'esecuzione di un programma [1].

Nell'ultimo decennio i repentini cambiamenti scanditi dallo sviluppo tecnologico stanno portando un ulteriore contributo al processo di digitalizzazione anche nel campo della finanza. Il bitcoin è una criptovaluta, ossia una valuta virtuale che costituisce una rappresentazione digitale di valore ed è utilizzata come mezzo di scambio o detenuta a scopo di investimento. Creata nel 2009 da un anonimo cono-

sciuto con lo pseudonimo di Satoshi Nakamoto, il bitcoin ha un grande problema poiché per convalidare le transazioni deve essere spesa una quantità spropositata di energia [6]. Per convalidare le transazioni eseguite è necessaria la risoluzione di un problema matematico svolto dall'esecuzione di un algoritmo chiamato SHA256. SH256 è un algoritmo di HASH che presa in input una stringa ci restituisce in output un messaggio di lunghezza arbitraria. La risoluzione del problema consiste nel trovare un input che restituisce un determinato output deciso dal sistema. La tesi è suddivisa in 3 capitoli. Nel primo capitolo vengono introdotti i concetti essenziali per procedere con l'analisi. Tali concetti sono:

- la reversibilità
- il sistema alla base di Bitcoin
- il funzionamento di SHA256

Nel capitolo 2 viene effettuata l'analisi di una possibile implementazione di SHA256 reversibile.

Nel capitolo 3 vengono analizzati i dati raccolti e possibili implementazioni di SHA256.

Indice

Elenco delle tabelle

Capitolo 1

Stato dell'arte

In questa sezione verrà trattata più nello specifico la reversibilità, enunciandone la storia, le scoperte ed il funzionamento.

1.1 Reversibilità

1.1.1 Principio di Landauer

Il «principio di Landauer» è attribuito a Rolf Landauer, fisico di origine europea che lavorò per lungo tempo alla IBM negli Stati Uniti. Nel 1961, in un articolo pubblicato su «IBM Journal of Research and Development» [5], Landauer aveva sottolineato come l'applicazione delle leggi della fisica al funzionamento dei computer impone limiti sull'energia che la macchina deve usare durante il suo funzionamento. Ogni volta che in un processo computazionale viene distrutta dell'informazione, e quindi due situazioni precedentemente distinte diventano indistinguibili, è necessaria una dissipazione di energia sotto forma di calore.

Il «principio di Landauer» afferma che "qualsiasi manipolazione logica irreversibile delle informazioni, come la cancellazione di un bit o la fusione di due percorsi computazionali, è accompagnata da un aumento dell'entropia nei gradi di libertà non informativi del dispositivo di elaborazione delle informazioni o del suo ambiente"

[5]. Ad ogni bit da dover dissipare è accompagnata un'emanazione di calore, ossia l'energia prima contenuta nel bit che si trasforma. Ipotizzando una temperatura di 20 gradi l'energia richiesta per la dissipazione di un bit corrisponde a $2.8 * 10^{-21}$ joules oppure $7.8 * 10^{-28}$ kilowattora. L'architettura ed il modo di intendere l'informatica al giorno d'oggi non è pensabile senza la cancellazione di informazioni. Si prenda com esempio una porta logica AND. Quest'ultima prende in input 2 bit e ne restituisce in output 1. 1 in meno di quelli presi in ingresso, bit che verrà poi trasformato in calore. Un attuale CPU Intel ad una configurazione a 4 processori riesce a raggiungere le $1.2 * 10^{12}$ istruzioni al secondo a 620 watts per un rapporto di $7,4 * 10^{15}$ istruzioni per kilowattora. Se ognuno dei 4,3*10 9 transistor scarta un bit, il prodotto $3.2 * 10^{25}$ è solo 3 ordini di grandezza più piccolo del costo di dissipazione di un bit [1]. Proseguendo in questa direzione, duplicando l'efficienza energetica ogni 1,57 anni, si arriverà ad un momento, secondo il principio di Landauer, in cui il costo di una singola operazione diventerà uguale al costo per smaltire un bit se non superiore, ponendo così un limite fisico allo sviluppo tecnologico.

1.1.2 Storia della Reversibilità

Come visto in [3]: la computazione reversibile è un'idea molto vecchia, é stata dimostrata la sua possibile esistenza già nel 1963 da Lecerf e nel 1973 da Bennett. Alcuni studi hanno dimostrato che qualsiasi computazione può essere eseguita reversibilmente, ma con un overhead quadratico in spazio o un overhead esponenziale in tempo. Risultati ancora più recenti hanno mostrato un trade-off con spazio subquadratico e tempo subesponenziale. Dopo la scoperta della reversibilità, verso la fine degli anni '70, Tommaso Toffoli, Ed Fredkin ed altri membri del MIT proposero come idea quella di una computazione balistica ed idearono delle porte logiche reversibili: la porta logica di Toffoli e di Fredkin. Purtroppo non si riuscì mai a costruire un modello che potesse descrivere la computazione balistica da loro ideata. L'idea di un computer balistico è stata importata alla Caltech con l'aiuto del fisico Richard Feynman. Feynman per dimostrare la possibile esistenza di un computer balistico sviluppò su carta un intero computer reversibile, mettendo anche le radici

del campo dei computer quantistici. Verso la metà degli anni '80 Charles Seitz ed alcuni suoi colleghi svilupparono una nuova tecnica per implementare la computazione reversibile, avvalendosi dell'uso di grandi induttori privi di chip. Sempre in questo periodo nacquero i primi linguaggi di alto livello reversibili quali R e Janus, sviluppati rispettivamente da John Chambers e colleghi e da Christopher Lutz e Howard Derby. Verso l'inizio degli anni '90 queste tecniche vennero approfondite da vari gruppi dell'ISI, rendendo possibili controlli logici reversibili. Dall'inizio degli anni 2000 fino ad oggi non sono state effettuate scoperte eclatanti ma il campo di studio è molto ampio e la ricerca continua imperterrita.

1.1.3 Modelli energetici

Come visto in [1]: il costo di energia deriva dalla dissipazione di informazioni. Per comprendere al meglio ciò useremo dei modelli energetici per creare una stima della perdita di energia tratta dal principio di Landauer. Il primo modello, anche quello più semplice dei 3, si basa sulle porte logiche e serve per riuscire a comprendere cosa accade a livello macchina. I restanti 2 modelli serviranno per analizzare i costi energetici di algoritmi visti da un linguaggio di alto livello.

Modello Circuito Energetico

Consideriamo al livello più basso le porte logiche. Ogni porta logica è una funzione $x \xrightarrow{f} y$. L'energia spesa a questo livello corrisponde al logaritmo delle possibili combinazioni dell'input x fratto le possibili combinazioni dell'output y. In una funzione Y = f(X) quindi sappiamo che l'energia spesa corrisponde a $lg(\|X\|/\|Y\|)$. L'output in una porta logica non può essere negativo in quanto la dimensione dell'output non può superare la dimensione dell'input. Il costo energetico di un circuito equivale alla somma del costo energetico delle sue porte logiche. Il costo energetico risulta 0, in una porta logica e di conseguenza in un circuito, solamente se la dimensione dell'input corrisponde alla dimensione dell'output. Nel caso in cui il costo energetico risulti 0 allora la porta logica, o il circuito, è reversibile. Un circuito reversibile deve computare per forza una funzione biettiva. Ci rendiamo conto che non tutte le porte logiche sono reversibili. La AND e la OR prendono

in input 2 bit ritornandone 1; per cui il loro costo energetico non equivale a 0. Ciò significa che non sono reversibili e che quindi non si può costruire il loro input a partire dall'output. Dalla porta NOT si può ricavare l'input partendo dal suo output (se abbiamo in output 0 sappiamo benissimo che l'input è 1) e pertanto è reversibile. Essendo reversibile allora il suo costo energetico risulta essere 0.

Modello Energetico ad Accesso Casuale (Word RAM)

Il modello energetico ad accesso casuale permette l'accesso a locazioni di memoria w in tempo costante ed una serie di operazioni in tempo O(1), assumendo che l'accesso in memoria sia possibile in maniera reversibile (questo argomento non verrà trattato ma è dimostrato essere possibile) [1]. Ci restringiamo ad usare operazioni di linguaggi di alto livello con i loro analoghi reversibili. Il costo di ogni operazione risulta essere quello calcolato nel primo modello, ossia il logaritmo della dimensione dell'input fratto la dimensione dell'output. Questo modello, inoltre. deve ritornare alla fine della computazione la macchina al suo stato originale, con l'eccezione di una copia dell'output salvata in un registro specifico della macchina. In questo modello il nostro pseudocodice di livello basso sarà simile ad assembly. A questo livello esplicitiamo ogni linea di codice con un numero del program counter che verrà incrementato all'esecuzione di ogni operazione. Quest'ultimo potrà però essere anche manipolato grazie a dei salti.

Modello energetico Transdicotomico RAM

Questo modello è il più potente e flessibile. Come il modello precedente l'accesso in memoria di dimensione w è costante. Assumiamo che la dimensione delle locazioni in memoria w sia uguale a $\Omega(\lg n)$ dove n è la dimensione dell'input per far si che sia possibile memorizzare in una parola l'intero input. Il costo di ogni operazione come nel primo modello è il logaritmo della dimensione dell'input fratto la dimensione dell'output. Come il secondo modello alla fine di una computazione vogliamo tornare allo stato iniziale ma con una copia dell'output salvata in una cella di memoria apposita. Tutte le assegnazioni a variabili saranno nella forma di TUPLE = TUPLE. Entrambe le tuple dovranno avere lo stesso numero di elementi

ed il numero di elementi dovrà essere della grandezze O(1). Nella tupla sinistra vediamo i blocchi di memoria che verranno usati, mentre nella tupla destra i valori che verranno messi al loro interno espresso in espressioni. Anche le espressioni devono contenere O(1) elementi. Per evidenziare se una variabile non è inizializzata vengono usate lettere sottolineate. (a,b,c) = (a,b,a + b) infatti ha costo energetico w perché andremo a cancellare tutto ciò che era dentro c. Invece (a,b,$\underline{c}$) = (a,b, a + b) avrà costo 0 perché non dovremo cancellare nessun bit. È un modello molto comodo e relativamente facile da usare per poter calcolare il costo energetico di molte operazioni. Anche se non sarà detto esplicitamente per l'analisi ci si baserà su modello simile a quest'ultimo. Come verrà detto anche dopo l'obbiettivo dell'algoritmo reversibile sarà quello di ritornare allo stato iniziale della macchina con l'output salvato in una cella apposita.

1.1.4 Pseudocodice di alto livello

Ora si analizzerà lo pseudocodice che utilizzeremo per il calcolo della complessità nel resto del progetto. Il linguaggio risulta simile a quello usato nel modello Trandicotomio RAM ma con una serie di accortezze per renderlo più simile ad un linguaggio come Python. Le linee di codice non corrisponderanno più a TUPLA = TUPLA ma nel caso una variabile non dovesse cambiare questa verrà tolta, diventando VARIABILI = ESPRESSIONI. (a, b, c) = (a + 1, b , a + b) diventa (a, c) = (a + 1, a + b). Potrebbe essere utile ai fini dell'analisi usare alcune volte un linguaggio di basso livello simile ad assembly per spiegare nel dettaglio cosa succede a livello macchina. Il linguaggio preso in considerazione risulta simile a quello visto nel livello RAM. Le funzioni di sistema che faranno uso di hardware dedicato avranno lo stesso nome sia nel codice di basso livello che in quello di alto livello. Ad ogni frammento di codice di alto livello corrisponderanno 2 frammenti di codice di basso livello. Il primo frammento corrisponderà all'esecuzione del comando, mentre il secondo all'inversione delle istruzioni date in precedenza. L'inversione viene quindi nascosta dal linguaggio di alto livello.

1.1.5 Logging ed Unrolling

In un computer normale alla fine della sua esecuzione tutte le variabili verrebbero eliminate, creando così un grande costo energetico. Per evitare tutto questo basterà eseguire reversibilmente parti del codice per riarrivare allo stato iniziale del calcolatore. Questo processo di invertire parti di codice viene definito come "Unrolling". Per decidere con certezza quali parti del codice invertire useremo una delle keyword del linguaggio di alto livello detta log. Tutto ciò che è scritto dentro un log verrà eseguito in maniera inversa quando verrà raggiunta un'altra keyword del linguaggio di alto livello, unroll.

Esempio di esecuzione di log ed unroll ad alto livello

```
1  log x = 5
2  log x = 10
3  unroll
```

Esempio di esecuzione di log ed unroll a basso livello

```
1  x += 5
2  mem[lp] += x
3  x -= mem[lp]
4  lp += 1
5  x += 10
6  x -= 10
7  lp -= 1
8  x += mem[lp]
9  mem[lp] -= x
10 x -= 5
```

In questo esempio vediamo come del codice di alto livello sia esprimibile in un codice di basso livello. Alla riga 1 viene assegnato ad x un valore di 5 dentro un log. Questo vuol dire che questa operazione di assegnamento dovrà poi essere resa reversibile. Questa operazione avviene solamente a riga 3 di basso livello. Per invertire questa funzione basterà semplicemente sottrarre 5. A riga 2 però viene aggiornato il valore di x da 5 a 10. Questa operazione avviene in codice di basso

livello da riga 2 a riga 5. Per poter far ritornare x al valore precedente a quello del nuovo assegnamento dobbiamo salvare nella memoria il valore attuale (riga 2 basso livello). Come abbiamo detto prima le assegnazioni costano memoria mentre gli incrementi ed i decrementi no. Per aggiornare il valore di x si riduce prima il suo valore a 0 (riga 3 basso livello) e poi si incrementa la variabile col nuovo valore (riga 5 basso livello). In tutto questo si incrementa anche il puntatore in memoria di 1 così che ci indichi uno spazio di memoria libero (riga 4 basso livello). L'unroll occupa tutto il resto del codice di basso livello (riga 6-10 basso livello) perché deve invertire tutte le operazioni fatte in precedenza. Da riga 6 a riga 9 viene invertito il codice che ad alto livello si trovava a riga 2, riportando allo stato che si aveva a riga 1. Si vuole quindi che il valore di x ritorni a 5. Prima di tutto viene sottratto l'attuale valore di x portandolo di nuovo a 0 (riga 6) e poi viene incrementato il valore che abbiamo salvato prima (riga 8), ossia proprio 5. A riga 107 viene decrementato il puntatore alla memoria perché prima puntava ad un blocco di memoria vuoto e decrementandolo ritorna a puntare ad uno spazio di memoria occupato. Anche lo stack di memoria deve ritornare al suo stadio originale ed infatti a riga 109 viene decrementato il valore che conteneva prima quello spazio di memoria, ora contenuto in x, riportando il valore della cella a 0, il suo valore originale. Infine viene inverto il codice di alto livello a riga 1 decrementando il valore assegnatogli inizialmente (riga 10), e riportando la macchina al suo stato iniziale.

1.2 BITCOIN

Come visto in [4]: il bitcoin è una criptovaluta, ossia una valuta virtuale che costituisce una rappresentazione digitale di valore ed è utilizzata come mezzo di scambio o detenuta a scopo di investimento, creata nel 2009 da un anonimo conosciuto con lo pseudonimo di Satoshi Nakamoto. La più importante feature di questo sistema è il fatto che sia decentralizzato. Non vi sono autorità che lo gestiscono, nessuna banca che conii moneta e nessun intermediario finanziario per convalidare le transazioni. La banca è costituita infatti da un sistema peer-to-peer al quale tutti possono accedere. Secondo il sito `block-chain.com` sono circa 19 milioni i bitcoin attualmente in circolazione e sono circa 700 mila i blocchi minati. Il valore in data 28 agosto 2022 di un bitcoin sul mercato è di circa 20 mila dollari.

1.2.1 Block-Chain

La Block-Chain è l'idea che si trova alla base di Bitcoin. Sfrutta le caratteristiche di una rete informatica di nodi e consente di gestire e aggiornare un registro che contiene dati ed informazioni, come le transazioni, senza la necessità di un'unità centrale di controllo e verifica. Le transazioni nel sistema di Bitcoin non vengono salvate in una singola lista ma in una lista di blocchi. Ogni blocco contiene un insieme di transazione che puntano ad un blocco successivo. Percorrendo tutta la lista è quindi possibile riuscire a vedere tutte le transazioni che sono state convalidate fino a quel momento. È credenza comune che il sistema Bitcoin consenta anonimato ma ciò è falso poichè la lista è visitabile da chiunque lo voglia fare e le transazioni sono dunque accessibili a tutti. Ogni blocco è composto da 2 parti: l'header e la lista di transazioni del blocco. L'header contiene varie informazioni riguardanti il blocco, fra le più importanti abbiamo l'hash identificativo del blocco stesso, di quello precedente, il nonce, usato nel processo di mining ed il numero del blocco. Ogni persona che partecipa al meccanismo peer-to-peer possiede un elenco di transazioni non ancora assimilate alla block-chain. Quando una nuova transazione viene effettuata la persona che la ha attivata la aggiunge alla propria lista e invia una copia della transazione a tutte le altre persone della rete

affinché la aggiungano alla loro lista. La tecnologia dietro a Bitcoin rende possibile scambiarsi valori attraverso internet nello stesso modo in cui vengono scambiate le informazioni. Una transazione tuttavia non è considerata valida fino a che non viene aggiunta alla block-chain. Grazie al meccanismo di mining è possibile aggiungere un blocco alla block-chain. Il mining è un'attività che ogni persona può compiere grazie alla quale le nuove transizioni vengono confermate e viene erogata nuova moneta. Questa attività può essere svolta da ogni persona che installi il client Bitcoin sul proprio calcolatore. Per riuscire a collegare un blocco con il resto della Block-Chain bisogna risolvere un problema matematico che richiede molti tentativi e quindi anche molto tempo per essere risolto. Il problema è volutamente complesso per rendere difficile il collegamento di nuovi blocchi. Essendo arduo collegare un nuovo blocco è molto difficile che una persona malintenzionata riesca a collegare più blocchi di seguito scrivendo su varie tabelle di transazioni il falso. Essendo il mining eseguito da varie persone contemporaneamente può succedere che ad un blocco ne vengano collegati 2 creando una biforcazione. In questi casi le due biforcazioni continueranno a crescere in parallelo fino a quando una delle due non diventerà abbastanza più lunga dell'altra facendo cadere quest'ultima in disuso.

1.2.2 Proof of Work

Per collegare un blocco con il resto della block-chain è necessario utilizzare un algoritmo di HASH per trovare un determinato output deciso dal sistema, l'algoritmo di HASH in questione è SHA256. Una funzione di HASH è una funzione che può prendere in input una qualsiasi stringa e ritorna in output una stringa con una lunghezza prefissata. La principale caratteristica di queste funzioni è che al minimo variare dell'input l'output viene completamente stravolto. Queste funzioni sono molto usate nel campo della cyber security in quanto possono essere impiegate in varie maniere. Per ottenere il digest (ossia l'output della funzione di HASH) di un blocco vengono prese in input dalla funzione il blocco stesso ed un nonce (valore casuale che cambia di volta in volta) fino ad ottenere in output una stringa con abbastanza bit 0 iniziali richiesti dal sistema. La possibilità di riuscire

a trovare un digest con 30 bit 0 iniziali prendendo in input il blocco ed un once è di 1 su un miliardo, il che rende molto difficile trovare un output adatto. Ogni blocco è collegato al blocco precedente, contenendo il suo HASH. Se si volesse volesse modificare un blocco qualsiasi il suo digest cambierebbe, ed a sua volta cambierebbero anche quelli dei blocchi successivi e così via. Ciò rende quasi impossibile o comunque molto difficile la modifica di un qualsiasi blocco, perché cambiando un blocco si dovrebbe cambiare l'intera lista riuscendo a trovare il nuovo digest di ogni singolo blocco. Volendo cambiare un blocco si è costretti a dover cambiare tutta la block-chain. Quindi si dovrebbe rifare tutto il lavoro di collegamento che l'intera comunità di Bitcoin ha svolto dal lancio di quest'ultimo.

1.2.3 Mining

L'attività di cercare un digest accettabile viene definita Mining e coloro che la praticano si chiamano miners. Ci si potrebbe chiedere, perché praticare un'attività tanto dispendiosa per convalidare le transazioni di qualcun altro? Il sistema Bitcoin premia coloro che praticano Mining. Appena trovato un digest adatto per collegare un nuovo blocco alla lista di transizioni verrà emessa una transazione che premia il miner con dei bitcoin. Oltre a questo guadagno si viene anche ricompensati in percentuale alle transazioni che sono state eseguite nel blocco. Il numero di bitcoin ottenuti come ricompensa per il collegamento dopo ogni blocco però si va sempre riducendo. Infatti il numero massimo di bitcoin è tarato a 21 milioni. Raggiunta quella cifra i miners guadagneranno solo dalla commissione per aver collegato il blocco e non otterranno più bitcoin "extra".

1.3 SHA256

Come accennato in precedenza SHA256 è un algoritmo di HASH usato per il proof of work dei bitcoin. Ora si analizzerà quest'ultimo nel dettaglio per capirne il funzionamento.

1.3.1 Operazioni e Funzioni Ausiliarie

L'algoritmo di SHA256 fa utilizzo di varie funzioni che ora andremo ad elencare

Operazioni

Fra le vari operazioni non basilari che vengono utilizzate nell'algoritmo troviamo:

- RotR(A, n) che denota la rotazione circolare verso destra di A di n bit.
- ShR(A, n) che denota lo shift verso destra di A di n bit.

Operazioni

Le funzioni che vengono utilizzate sono:

- $Ch(X, Y, Z) = (X \wedge Y) \oplus (\overline{X} \wedge Z)$ indicata nel codice come ch(x,y,z)
- $Maj(X, Y, Z) = (X \wedge Y) \oplus (X \wedge Z) \oplus (Y \wedge Z)$ indicata nel codice come maj(x,y,z)
- $\Sigma_0(X) = RotR(X, 2) \oplus RotR(X, 13) \oplus RotR(X, 22)$ indicata nel codice come capsigma_(x)
- $\Sigma_1(X) = RotR(X, 6) \oplus RotR(X, 11) \oplus RotR(X, 25)$ indicata nel codice come capsigma_1(X)
- $\alpha_0(X) = RotR(X, 7) \oplus RotR(X, 18) \oplus ShR(X, 3)$ indicata nel codice come sigma_0(X)
- $\alpha_1(X) = RotR(X, 17) \oplus RotR(X, 19) \oplus ShR(X, 10)$ indicata nel codice come sigma_1(X)

1.3.2 Funzionamento

Il codice che si vedrà ora funziona prendendo in input una stringa rappresentata da n Byte. Avendo questa particolarità molte operazioni saranno implementate con byte invece che con bit anche se l'algoritmo SHA256 prevede un utilizzo partendo da singoli bit.

L'algoritmo riesce a prendere in input una qualsiasi stringa che abbia meno di 2^{64} bit. L'input ricevuto in questa versione dell'algoritmo è una stringa. Questa stringa è a sua volta rappresentata da un numero finito di byte. Dopo che si è ricevuto l'input di lunghezza l bisogna creare un messaggio in questa maniera:

- Aggiungendo un bit 1 (riga 8). In questo caso viene aggiunto un byte con il valore di 128 che corrisponde ad un insieme di 8 bit di cui il primo sarà uguale ad 1.
- Aggiungendo k bit 0 tali che k + l + 1 = 448 mod 512 (riga 11-12). Come il caso precedente non vengono aggiunti bit ma byte, il risultato finale sarà lo stesso.
- Aggiungere la lunghezza del messaggio iniziale in base 2 usando 64 bit, ossia 8 byte.

Alla fine di questo processo otterremo un messaggio lungo k + l + 1 + 64 bit, e quindi divisibile in macroblocchi da 512 bit (riga 19-22).

Da ognuno dei macroblocchi ottenuti verranno costruiti 64 blocchi da 32 bit l'uno. I primi 16 sono ottenuti dalla suddivisione del macroblocco in appunto 16 parti (riga 42-46), i restanti 48 invece sono ottenuti dalla seguente formula:

$$W_i = a_1(W_{i-2}) + W_{i-7} + a_0(W_{i-15}) + W_{i-16}$$

per i compreso tra 17 e 64 (riga 36-56).

Vengono inizializzate poi le variabili h0,...h7 che serviranno a ricavare l'output finale (riga 59-66). Il messaggio che verrà ritornato dall'algoritmo sarà la concate-

nazione dei valori di queste ultime variabili. Ognuna di queste è un intero composto da 32 bit. Concatenando tutte le 8 variabili otteniamo infine un messaggio da 256 bit, da qui il nome SHA256.

Dopo questa inizializzazione viene eseguito un for che itera un numero di volte pari al numero di macroblocchi generati. All'inizio di questo ciclo verranno inizializzate delle variabili, ossia a,...,h (59-66). Queste variabili verranno modificate a loro volta dentro un altro for che itera 64 volte (69-82).

Dentro al for sono anche presenti 2 variabili ausiliarie (70-73) alle quali vengono assegnati dei valori usando la funzioni viste in precedenza. Sempre dentro a quest'ultimo verranno eseguite una serie di permutazioni, ossia uno scambio di valori fra variabili. Finito il for che itera 64 volte vengono aggiornate le variabili h0,...,h7, incrementate dei valori di a,...,h (riga 84-92). Alla conclusione del for che itera per ogni macroblocco viene ritornato come risultato dell'algoritmo la concatenazione dei valori di h0,...,h7 (riga 95-98).

```
1  # Padding
2   #determiniamo il numero di bit
3  length = len(message) * 8
4   #aggiungiamo il bit necessario
5  message.append(0x80)
6  #aggiungiamo bit 0 fino a quando la lunghezza non diventa 448
7  while (len(message) * 8 + 64) % 512 != 0:
8      message.append(0x00)
9
10 #aggiungiamo la lunghezza del messaggio in 64 bit
11 message += length.to_bytes(8, 'big') # pad to 8 bytes or 64
       bits
12
13
14 # Dividiamo il messaggio in macroblocchi da 512 bit
15 blocks = [] # contains 512-bit chunks of message
16 for i in range(0, len(message), 64): # 64 bytes is 512 bits
17     blocks.append(message[i:i+64])
18
19 #inizializiamo le veriabili
20 h0 = 0x6a09e667
21 h1 = 0xbb67ae85
22 h2 = 0x3c6ef372
23 h3 = 0xa54ff53a
24 h5 = 0x9b05688c
25 h4 = 0x510e527f
26 h6 = 0x1f83d9ab
27 h7 = 0x5be0cd19
28
29 # costruiamo i blocchi che useremo per calcolare l'hash finale
30 # for per ogni blocco ottenuto
31 for message_block in blocks:
32     # Prepare message schedule
```

```
33      message_schedule = []
34      #per ogni blocco dobbiamo ricavare 64 blocchi
35     for t in range(0, 64): #for reversibile
36          #i 16 blocchi corrispondono ad una suddivisione del
               macroblocco
37          if t <= 15: # if anche esso reversibile
38                  # adds the t'th 32 bit word of the block,
39                  # starting from leftmost word
40                  # 4 bytes at a time
41                 message_schedule.append(bytes
42                                 (message_block[t*4:(t*4)+4]))
43              else:
44                  #gli altri blocchi sono ricavati da questa
                       formula
45
46                  term1 = sigma_1(int.
47                  from_bytes(message_schedule[t-2], 'big'))
48                  term2 = int.
49                  from_bytes(message_schedule[t-7], 'big')
50                  term3 = sigma_0(int.
51                          from_bytes(message_schedule[t-15], 'big
                               '))
52                  term4 = int.
53                          from_bytes(message_schedule[t-16], 'big
                               ')
54
55                  schedule = ((term1 + term2 + term3 + term4) %
                       2**32)
56                           .to_bytes(4, 'big')
57                  message_schedule.append(schedule)
58
59  # Initializiamo nuove variabili
60      a = h0
```

```
61      b = h1
62      c = h2
63      d = h3
64      e = h4
65      f = h5
66      g = h6
67      h = h7
68
69      # iteriamo 64 volte per ogni blocco
70      for t in range(64):
71          t1 = ((h + capsigma:1(e) + ch(e, f, g) + K[t] +
72                  int.from_bytes(message_schedule[t], 'big')) %
                       2**32)
73
74          t2 = (capsigma_0(a) + maj(a, b, c)) % 2**32
75
76          h = g
77          g = f
78          f = e
79          e = (d + t1) % 2**32
80          d = c
81          c = b
82          b = a
83          a = (t1 + t2) % 2**32
84
85      h0 = (h0 + a) % 2**32
86      h1 = (h1 + b) % 2**32
87      h2 = (h2 + c) % 2**32
88      h3 = (h3 + d) % 2**32
89      h4 = (h4 + e) % 2**32
90      h5 = (h5 + f) % 2**32
91      h6 = (h6 + g) % 2**32
92      h7 = (h7 + h) % 2**32
```

```
93
94
95      return ((h0).to_bytes(4, 'big') + (h1).to_bytes(4, 'big') +
96              (h2).to_bytes(4, 'big') + (h3).to_bytes(4, 'big') +
97              (h4).to_bytes(4, 'big') + (h5).to_bytes(4, 'big') +
98            (h6).to_bytes(4, 'big') + (h7).to_bytes(4, 'big'))
```

Come possiamo vedere SHA256 è un algoritmo semplice e con una complessità lineare. Il numero di iterazioni che esegue dipende esclusivamente dal numero di macroblocchi che si ricavano dal messaggio iniziale, quindi dalla lunghezza del messaggio e non dal valore che può avere.

Capitolo 2

SHA256 Reversibile

Questo studio vuole analizzare il costo energetico e quello in termini di spazio di un'ipotetica implementazione di un algoritmo SHA256 reversibile rispetto alla sua controparte originale.

2.1 Nozioni di Reversibilità

Prima di iniziare l'analisi vanno enunciate alcune premesse al fine di comprendere al meglio ciò che si tratterà dopo.

2.1.1 Operazioni e Costrutti Reversibili

Come detto in precedenza una funzione reversibile è una funzione che ci permette di ricostruire l'output partendo dall'input. Tratto da [1]: l'incremento ed il decremento sono operazioni reversibili. Gli assegnamenti a variabili con dei valori diversi da 0 invece non lo sono, perché per essere effettuati necessitano la cancellazione dell'informazione che si trovava all'interno della variabile.

AND, OR Reversibili

AND ed OR sono delle porte logiche non reversibili, infatti partendo dal loro output, come detto in precedenza, di nessuna di queste porte è possibile ricostruire l'input. L'unica maniera per poter ricostruire il loro input è quello di costruire delle porte logiche apposite. Per definizione una porta logica reversibile è una porta dal quale output è ricostruibile l'input. Prendiamo come esempio la porta AND. Una classica porta AND prende 2 input e restituisce un output. Se l'output è 1 possiamo ricostruire l'input con certezza, infatti sappiamo essere (1,1). In caso l'output fosse 0 non sappiamo con certezza quale possa essere stato il suo input ma abbiamo 3 possibili soluzioni: (1,0), (0,1) e (0,0). Prendiamo un'altra porta logica come esempio, che prende in input 2 bit e ne restituisce altri 2. Nella prima porta di output restituirà il bit inserito nella prima porta di input e nella seconda porta di output restituisce l'AND logico fra il primo input ed il secondo. Quindi inserendo come input 1 e 0 questa porta dovrebbe restituirci in output 1 e 0. Questa porta però non è reversibile nel caso in cui il primo input fosse 0 perché se avessimo come input 0 e 1 avremmo come output 0 e 0, e noi non possiamo dedurre che il secondo input fosse 0. L'unico modo per rendere reversibile una porta AND è quello di realizzare una porta logica come 3 porte di input e 3 porte di output dove i primi 2 input sono i bit che vogliamo confrontare ed il terzo un bit inizializzato a 0 dove vogliamo salvare il risultato dell'AND. In output quindi le prime 2 porte ritorneranno gli input e la terza il risultato dell'AND. Lo studio di una porta logica OR è analogo a quello di una porta logica AND e per ridondanza non verrà trattato. Possiamo notare come sia possibile ottimizzare il nostro hardware per risparmiare spazio sui singoli bit.

XOR Reversibile

La porta logica XOR è simile alla porta AND e OR ma per ottenere l'input a partire dall'output possiamo risparmiare spazio salvando una sola variabile dell'input. Se sappiamo che l'output è 1 e conosciamo 1 dei 2 input possiamo ricostruire anche l'altro sapendo che se sono diversi restituiscono 1 e 0 se sono uguali. Allo stesso

modo se sappiamo che l'output è 0 e conosciamo 1 dei 2 input sapremo che l'altro sarà uguale a quest'ultimo.

Durante l'esecuzione di un processo è spesso necessario l'utilizzo di costrutti logici come l' IF ed il FOR.

IF reversibile

Per eseguire un IF reversibile è necessario salvare dentro lo stack di memoria un singolo bit per poi, quando è arrivato il momento dell'unroll, capire quale frammento di codice è stato eseguito. Nel codice che si vedrà più avanti per salvare queste informazioni non si salverà un singolo bit ma direttamente una parola contenente il valore 1 o 0. Se il valore salvato dentro lo stack sarà 1 allora si saprà di essere entrati nel costrutto in precedenza e quindi di dover eseguire reversibilmente il codice contenuto nello stesso. In caso il valore dentro lo stack sarà uguale a 0 allora non si dovrà eseguire il codice inverso dentro il costrutto, poiché durante l'esecuzione iniziale questo codice non è stato eseguito. Si potrebbe realizzare un meccanismo per salvare singoli bit ma per semplicità e per salvare tutte le informazioni in un solo stack non useremo questo approccio. Tuttavia è possibile, invece di salvare un bit per controllare successivamente se si è entrati nel costrutto, controllare direttamente se la condizione dell'IF viene rispettata. Tuttavia questa soluzione è utilizzabile solamente se l'IF è protetto, ossia se le variabili usate nella condizione di entrata dell'IF non vengono cambiate dentro il costrutto.

FOR reversibile

Per eseguire un FOR reversibile abbiamo bisogno di una parola di memoria aggiuntiva. Questa parola serve per salvare il numero di iterazioni effettuate durante l'esecuzione per poi sapere quante volte andrà eseguito il codice inverso all'interno del FOR.

2.1.2 Esecuzione su hardware non reversibile

Nell'informatica odierna non è possibile purtroppo eseguire istruzioni totalmente reversibili poiché i computer non sono progettati per eseguirle. Per esempio: se in un programma si eseguisse un'assegnazione ad una variabile inizializzata a 0 questa operazione dovrebbe essere reversibile. Nonostante questo i computer odierni sono progettati per fare un utilizzo di registri interni dei quali noi non possiamo controllare il valore. L'unico modo per creare un processo totalmente reversibile è quindi quello di progettare anche un hardware completamente reversibile. In questa analisi si vedrà più avanti la realizzazione di un algoritmo di SHA256 scritto in C. Nonostante siano usate solo funzioni ed operazioni reversibili l'esecuzione del programma non lo è, in quanto per esserlo dovrebbe appoggiarsi su un hardware reversibile.

2.1.3 Costi delle funzioni

Nella reversibilità non solo il linguaggio utilizzato ma anche l'hardware ha una grande importanza. Infatti il costo in energia si calcola proprio dalle porte logiche. Come visto prima in Python, SHA256 presenta delle funzioni più complesse del semplice assegnamento o addizione. Le funzioni da analizzare sono le stesse viste in sezione 1.3.1.

Il costo delle funzioni stesso potrebbe inficiare considevorevolmente sul costo totale in energia dell'algoritmo. Per ottimizzare questo algoritmo assumiamo d'ora in avanti di avere un hardware che possa svolgere queste funzioni. Per chiarezza si svolgerà un'analisi su come potrebbero essere implementata ognuna di queste funzioni. Prima però si deve analizzare il costo di ognuna delle operazioni base che compongono le precedenti funzioni.

Shift reversibile

Lo shift verrà eseguito da un hardware dedicato per risparmiare spazio durante l'inversione. Se costruiamo delle componenti dedicate infatti non dobbiamo preoccuparci di costi in energia, o memoria nel caso di uno shift reversibile, aggiuntivi.

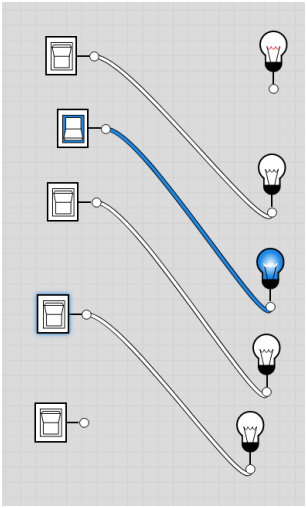

Figura 2.1: Realizzazione di shift che esegue 1 passo verso destra

Durante il codice dovremo eseguire vari shift e con passi diversi. Avendo un circuito per ogni shift da eseguire, non dobbiamo impiegare memoria per tener conto del numero di passi, né tanto meno della direzione dello shift. L'unica cosa da salvare in memoria saranno i bit che verranno tolti dalla variabile, per via di questa operazione.

Nel nostro codice tuttavia, facendo un circuito apposito, gli shift non costeranno mai energia poiché non verrà mai cambiata la variabile sulla quale è eseguito lo shift. Pertanto non ci ritroveremo mai a dover salvare singoli bit. Comunque è da ricordare che se ad una variabile già inizializzata viene assegnato un nuovo valore allora questo ci costerà energia in caso non decidessimo di immagazzinare il valore prima della nuova assegnazione. Analizzando l'algoritmo di SHA256 si sa che gli shift eseguiti sono solo 2: lo shift a destra di 3 passi e lo shift a destra di 10 passi. Si devono costruire 2 circuiti che ci permettano di eseguire questa operazione.

Rotate reversibile

L'operazione di rotazione è molto simile a quella di shift ma a differenza dello shift se fatto con un hardware apposito non avrà alcun costo dal punto di vista dell'energia. Infatti la funzione Rotate è facilmente invertibile e se sappiamo di

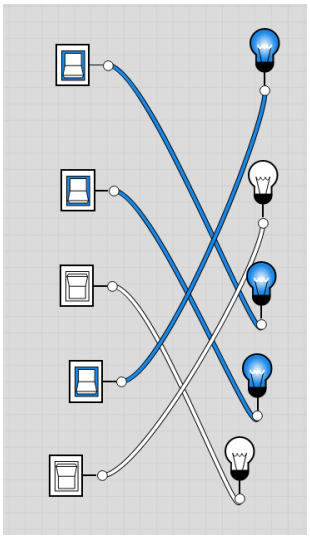

Figura 2.2: Realizzazione rotate che esegue una rotazione di 3 passi verso sinistra

quanto deve essere la rotazione, senza doverlo passare in input, il costo di questa funzione arriva a 0. Nello shift vi erano dei bit di informazione che potevano essere persi ma con rotate vengono semplicemente spostati. Siccome nell'algoritmo SHA256 le rotazioni sono sempre dello stesso numero possiamo evitare di passare in input il valore di rotazione ma assumendo che vi sia un hardware apposito, risparmiando così sia energia che spazio in memoria. Analizzando l'algoritmo di SHA256 si sa che i rotate eseguiti sono solo 10. Questi rotate sono tutti verso destre e di passi: 2, 13, 22, 6, 11, 25, 7, 18, 17, 19. Si devono costruire 10 circuiti che ci permettano di eseguire questa operazione.

Si può quindi affermare che nessuna delle funzioni viste in precedenza abbia un costo di energia per essere eseguita e che se si salvano i risultati in una variabile diversa da quelle utilizzate (come viene fatto nell'algoritmo SHA256) le funzioni sono reversibili.

2.2 Tecniche di reversibilità

Come descritto in [1] la reversibilità è un campo ancora da scoprire e pertanto non sono state definite tecniche univoche per rendere reversibile un processo. In questo

ambito dobbiamo sempre operare un trade-off fra memoria, energia e prestazioni. Sta infatti a noi stabilire se vogliamo rendere tutto l'algoritmo reversibile o solo alcune parti ed anche in che modo renderlo tale. Una maniera per rendere un algoritmo reversibile è quello di fare un complete logging. Questo significa decidere di immagazzinare ogni singola informazione al minimo cambiamento e di riassegnare il valore in memoria alla variabile quando il processo verrà invertito. Nonostante la semplicità, questa tecnica ha un costo dal punto di vista dello spazio pari al numero di volte in cui viene assegnato un nuovo valore ad una variabile. SHA256 ha una crescita lineare per quanto riguarda l'input e quindi è una strada che si potrebbe decidere di percorrere e vedremo anche in avanti che alcune sezioni si potrebbero risolvere in questa maniera.

2.3 Cambiamenti rispetto all'algoritmo originale

Rispetto al codice analizzato prima andranno effettuati alcuni cambiamenti necessari. Uno dei cambiamenti più importanti è l'assegnamento dei valori alle variabili. L'assegnamento infatti spesso ha un suo costo in energia se viene fatto ad una variabile contenente già delle informazioni. Anche quando si vuole "svuotare una variabile" questo costerà energia. Gli incrementi ed i decrementi però non costano informazioni e questo potrebbe permettere di ridurre il costo a 0. Tutti gli assegnamenti o quasi verranno sostituiti da incrementi e decrementi per ridurre i costi energetici.

2.4 Analisi

Cosa importante da sottolineare è cosa si vuole ottenere: si vuole massimizzare la reversibilità dell'algoritmo, così da risparmiare la maggior quantità di energia. Ma cosa vuol dire ottenere un algoritmo reversibile? Si vuole salvare l'output in un registro apposito del nostro calcolatore, alla fine dell'esecuzione, per poi eseguire l'algoritmo all'incontrario e ritornare allo stato iniziale della macchina, come visto nel modello energetico transdicotomico. Questa operazione deve essere

effettuata perché ritornando allo stato iniziale della macchina non si avranno celle di memoria con informazioni dentro da dover ripulire successivamente, non dovendo così distruggere informazioni. Per ogni frammento dell'algoritmo saranno mostrati:

- codice di alto livello
- codice di basso livello
- implementazione in c++

Successivamente verrà mostrato anche in basso livello ed in c++ come i frammenti di codice visti in precedenza saranno invertiti. Tutti i codici presentati in questa analisi sono già reversibili, e quindi eseguibili in maniera inversa.

Il codice in c++ che vediamo è tratto da una possibile implementazione di un algoritmo di SHA256 presentata in [10]. Precedentemente all'implementazione in c++ si era tentato di usare Janus [8] per rendere reversibile questo algoritmo. Purtroppo il linguaggio è molto carente dal punto di vista delle funzionalità e non è stato possibile eseguire le bit operation richieste. Nonostante tutto è stato possibile implementare un piccolo spezzone di codice, ossia il for che itera 64 volte per ogni macroblocco, in questo linguaggio. Resomi conto dell'impossibile implementazione in Janus sono passato alla modifica del codice sul quale ci si è basati per l'apprendimento del funzionamento dell'algoritmo, il codice visto in precedenza scritto in Python. É stato possibile,nonostante non fosse totalmente reversibile, eseguire alcuni test per constatare quanto spazio in più occupasse questo algoritmo rispetto a quello originale. Purtroppo Python è un linguaggio di "alto livello" e per questo molte informazioni sull'uso della memoria sono tenute nascoste al programmatore. Per avere un'idea più chiara sull'uso della memoria ho quindi optato per un implementazione in c++.

Lo studio di questo algoritmo verrà suddiviso in varie sezioni, analizzando il costo energetico ed il costo di spazio aggiuntivo per rendere un algoritmo reversibile.

Si assume di avere uno stack, lp, con celle di memoria inizializzate tutte a 0. Questo stack sarà utilizzato per salvare le informazioni durante l'esecuzione.

2.4.1 Operazioni su Stack

In questa analisi si vuole usare uno stack ma per farlo bisogna tenere conto delle operazioni da usare per potervi accedere. Si definiscono 2 funzioni: "push" e "pop". La prima funzione prende in input una variabile ed azzerandola mette il suo valore all'interno dello stack, poi aumentando il del puntatore alla cima dello stack. La seconda funzione prende in input una variabile e vi salva l'ultimo valore salvato nello stack, decrementandolo e decrementando anche il valore del puntatore.

Codice "push"	
1	`push(a):`
2	`    mem[lp] +=a`
3	`    a -= mem[lp]`
4	`    lp++`
Codice "pop"	
1	`pop(a):`
2	`    lp--`
3	`    a += mem[lp]`
4	`    mem[lp] -= a`

Tabella 2.1: Confronto "push" e "pop"

Queste stesse funzioni devono essere implementate anche in c++.

Codice "push" in C++

```
1
2 void SHA256::push(unsigned int* a)
3 {
4
5     mem[lp] += *a;
6     *a -= mem[lp];
7     lp++;
8
9 }
```

Codice "pop" in C++

```
1
2 void SHA256::pop(unsigned int* a)
3 {
4     lp--;
5     *a += mem[lp];
6     mem[lp] -= *a;
7
8 }
```

Tabella 2.2: Confronto "push" e "pop" i C++

2.4.2 Analisi creazione messaggio reversibile

Iniziamo l'analisi della creazione del messaggio dell'algoritmo iniziale. Questa operazione è a sua volta divisibile in 3 sezioni: l'aggiunta del bit 1, l'aggiunta di k bit 0 e l'aggiunta di 64 bit che esprimono la lunghezza del messaggio totale. Come prima cosa si deve salvare il messaggio iniziale in una variabile nuova da poter modificare successivamente, questa variabile sarà "newMessage". Partiamo

con l'aggiunta del bit 1.

Implementazione alto livello	1
	2
	3
		`newMessage = message`
		
		`newMessage.addBit(1)`

Implementazione basso livello	1
	2
	3
		`newMessage += message`
		
		`newMessage.addBit(1)`

Implementazione in c++	1	`m_block[len] += 0x80;`

Tabella 2.3: Confronto del codice dell'aggiunta bit 1

L'aggiunta di un singolo bit ad un messaggio risulta un'operazione molto semplice come possiamo vedere. Occupa una sola riga in ognuno dei 3 linguaggi. Nell'implementazione in c++ il messaggio iniziale è costituito da un array di byte. Per questo non verrà aggiunto un singolo bit al messaggio ma un byte col valore di 128. Poiché il valore in binario di 128 corrisponde a 10000000 e sapendo che il messaggio è rappresentato in byte, aggiungere 128 equivale ad aggiungere un bit 1 e 7 bit 0 alla fine del messaggio. Invertire questo codice richiede anch'esso poco codice. Basta togliere un bit nel caso del linguaggio di alto livello e sottrarre il valore aggiunto prima, ossia 128, al byte indicato del messaggio in c++.

Implementazione basso livello	`message.removeBit()`
Implementazione in c++	`m_block[len] -= 0x80;`

Tabella 2.4: Confronto del codice dell'inversione dell'aggiunta bit 1

Questa operazione è reversibile di suo e pertanto non costa memoria aggiuntiva ed è eseguibile senza costi energetici per dissipare bit.

Successivamente al messaggio devono essere aggiunti k bit tali che (n + k + 1)mod512 = 448.

Implementazione alto livello

```
1
2 int i = (lenBit(message) + 1) % 512
3 while ( i < 447):
4       newMessage.addBit(0)
5       i++
```

Implementazione basso livello

```
1 i += (lenBit(message) + 1) % 512
2 For(i < 448, i+=1):
3     newMessage.addBit(0)
4     goto beginFor
5 endFor
```

Implementazione in c++	
1	`for (i += len + 1 ; i % SHA224_256_BLOCK_SIZE != SHA224_256_BLOCK_SIZE - 9; i++) {`
2	
3	`  m_block[i] += 0x00;`
4	
5	`}`
6	
7	`push(&(i));`

Tabella 2.5: Confronto del codice dell'aggiunta k bit 0

Per aggiungere k bit 0 dobbiamo eseguire un ciclo for che itera per ogni bit 0 da aggiungere (riga 1-5 basso livello). Decidendo di usare i come una variabile globale (riga 1 basso livello) non dobbiamo preoccuparci di salvare questo valore in memoria e quando andremo ad invertire potremo riprendere semplicemente quest'ultimo. Nella versione scritta in c++ si può decidere di non apportare modifiche all'array in quanto, se lo stack è inizializzato a 0, si hanno già i valori che si volevano avere, ossia 0. Per comprendere meglio il funzionamento dell'algoritmo si è deciso comunque di implementare un for che renda l'idea dell'aggiunta di bit con valore 0. Per rendere reversibile questa operazione si può invertire il for salvando il valore di (riga 7 implementazione in c++) oppure, siccome le variabili dell'array sono inizializzate a 0, scegliere di non agire. L'aggiunta di un bit al nostro messaggio (riga 3 basso livello), come l'aggiunta di un bit 1, è di sua natura reversibile. L'unica cosa non reversibile in questo estratto di codice è il for che, a meno che non si imposti la variabile di iterazione come una variabile globale, avrà bisogno di una parola di memoria per salvare questa variabile.

Implementazione basso livello

```
1 #invertiamo il for togliendo i bit 0
2 For(i > (lenBit(message) + 1), i-=1)
3     removeBit(newMessage)
4 endFor
5 i -= (lenBit(message) + 1) / 512
6 removeBit(newMessage) #rimuoviamo 1 bit dal
    messaggio
```

Implementazione in c++

```
1 lp--;
2
3 for (i += mem[lp], mem[lp] -=i ; i >len; i--) {
4
5     m_block[i] -= 0x00;
6
7 }
```

Tabella 2.6: Contronto dell'inversione del codice dell'aggiunta k bit 0

Per invertire il for visto in precedenza basta aver salvato la variabile i e fare un for dove la variabile di iterazione deve riraggiungere il suo valore iniziale. Di solito questa variabile è settata a 0, ma essendo partiti da len(newMessage), sappiamo che la condizione di arresto sarà quella vista a riga 2. Dentro il for deve essere invertito l'aggiunta di un bit. Questa azione viene eseguita reversibilmente semplicemente con la rimozione di quest'ultimo (riga 3 basso livello reversibile). Per poi riportare la variabile i al valore 0 basterà eseguire un decremento del valore dell'inizializzazione (riga 5 implementazione reversibile). Lo stesso procedimento deve essere effettuato nel codice in C++ per essere reso reversibile. Infine devono

essere aggiunti al messaggio gli ultimi 64 bit (o 8 byte) contenenti la lunghezza del messaggio.

Implementazione alto livello	1	`newMessage.add64Bits(lenBit(message))`
Implementazione basso livello	1	`newMessage.add64Bits(lenBit(message))`
Implementazione in c++	1	`REV_SHA2_UNPACK32(len << 3, m_block + fm_len- 4);`

Tabella 2.7: Confronto di codice dell'aggiunta di 64 bit

L'aggiunta di 64 bit può essere paragonata come azione all'aggiunta di un solo bit. Questa azione è eseguita nel nostro codice da un componente hardware. Nel codice scritto in c++ invece possiamo vedere questa azione compiuta da una MACRO appositamente scritta.

Implementazione basso livello	1	`newMessage.remove64Bits(lenBit(message))`
Implementazione in c++	1	`SHA2_UNPACK32(len << 3, m_block + fm_len- 4);`

Tabella 2.8: Confronto di codice dell'inversione dell'aggiunta di 64 bit

Per invertire tale operazione ci basterà decrementare gli ultimi 64 bit del valore della lunghezza del messaggio iniziale. Questa operazione è eseguibile senza costi in spazio aggiunti a patto che abbiamo mantenuto in memoria o la lunghezza del messaggio iniziale o il messaggio iniziale stesso. In caso questa condizione fosse soddisfatta l'operazione sarà reversibile senza costi in spazio aggiuntivi. Nel codice in c++ possiamo vedere come l'operazione sia invertibile sempre grazie ad una macro che prende in input, in questo ordine, la lunghezza del messaggio iniziale e l'indirizzo in memoria del primo byte dove vengono salvati i 64 bit.

MACRO SHA2_UNPACK3

```
1 #define SHA2_UNPACK32(x, str)                \
2 {                                            \
3     *((str) + 3) += (uint8) ((x));           \
4     *((str) + 2) += (uint8) ((x)>>8);        \
5     *((str) + 1) += (uint8) ((x)>>16);       \
6     *((str) + 0) += (uint8) ((x)>>24);       \
7 }
```

MACRO REV_SHA2_UNPACK32

```
1 #define REV_SHA2_UNPACK32(x, str)            \
2 {                                            \
3     *((str) + 3) -= (uint8) ((x));           \
4     *((str) + 2) -= (uint8) ((x)>>8);        \
5     *((str) + 1) -= (uint8) ((x)>>16);       \
6     *((str) + 0) -= (uint8) ((x)>>24);       \
7 }
```

Tabella 2.9: Confronto MACRO 1

Calcolando il costo aggiuntivo in spazio dell'aggiunta di 1 bit, ossia 0 parole di memoria aggiuntive, il costo dell'aggiunta di k bit 0, ossia 1 parola di memoria

aggiuntiva, e l'aggiunta di 64 bit che esprimono la lunghezza del messaggio, arriviamo alla conclusione che questa prima parte di codice di SHA256 è eseguibile reversibilmente con un consumo di memoria addizionale di 1 parola.

2.4.3 Analisi creazioni blocchi reversibile

L'analisi prosegue con la suddivisione in blocchi del messaggio iniziale.

Implementazione alto livello

```
1  macroBLock = [lenBit(newMessage) / 512]
2  block = [lenBit(newMessage) / 512][64]
3  for i in range (len(newMessage / 512)):
4        macroBlock[i] = part(i, newMessage)
5          for j in range(64):
6                if (j <= 15):
7                    block[i][j] += takePart(j,
                         macroBlock[i])
8              else:
9                    block[i][j] += W(j, block[i][])
```

Implementazione basso livello

```
1  i = 0
2  macroBLock = [len(newMessage) / 512]
3  block = [lenBit(newMessage) / 512][64]
4  For( i < len(newMessage) / 512, i+=1):
5      macroBlock[i] = part(i, newMessage)
6      j = 0
7      For( j < 64, j+=1):
8          if(j < 16):
9
10             block[i][j] += takePart(j,
                   macroBlock[i])
11
12         else:
13
14             block[i][j] += W(j, block[i][])
15
16     endFor
17     push(j)
18 endFor
19 push(i)
```

Implementazione in c++

```
1  unsigned char *sub_block;
2  uint32 i;
3  uint32 j;
4  for (i = 0; i < (int) block_nb; i++) {
5
6      pmem[pp] = sub_block;
7      sub_block = 0;
8      pp++;
9      sub_block = message + (i << 6);
10
11     for (j = 0; j < 16; j++) {
12
13         push(&(w[j]));
14         SHA2_PACK32(&sub_block[j << 2], &w[j]);
15
16     }
17     for (j = 16; j < 64; j++) {
18
19         push(&(w[j]));
20         w[j] = SHA256_F4(w[j - 2]) + w[j -
               7] + SHA256_F3(w[j - 15]) + w[j -
               16];
21
22     }
23
24     push(&(j));
```

Tabella 2.10: Confronto di codice della creazione di macroblocchi e blocchi

SHA2_PACK32

```
1 #define SHA2_PACK32(str, x)                          \
2 {                                                    \
3     *(x) =   ((uint32) *((str) + 3)      )           \
4            | ((uint32) *((str) + 2) <<  8)           \
5            | ((uint32) *((str) + 1) << 16)           \
6            | ((uint32) *((str) + 0) << 24);          \
7 }
```

SHA256_F3

```
1 #define SHA256_F2(x) (SHA2_ROTR(x,  6) ^
    SHA2_ROTR(x, 11) ^ SHA2_ROTR(x, 25))
```

SHA256_F3

```
1 #define SHA256_F3(x) (SHA2_ROTR(x,  7) ^
    SHA2_ROTR(x, 18) ^ SHA2_SHFR(x,  3))
```

Tabella 2.11: Codici MACRO C++

Il messaggio deve essere suddiviso in n macroblocchi, dove n = len(newMessage) /512, e da ognuno di questi macroblocchi si devono ricavare 64 blocchi. La divisione in macroblocchi può essere ottenuta eseguendo un for che itera un numero di volte pari alla lunghezza del messaggio in bit diviso 512 (riga 1-9 alto livello, riga 1-19 basso livello). Eseguire questo for costerà una parola aggiunta in memoria per essere reso reversibile. Il numero di iterazioni viene salvato solamente alla fine dell'esecuzione del for (riga 19 basso livello, riga 25 C++). Ogni blocco del messaggio viene salvato in una variabile globale grazie ad una primitiva di nome part (riga 4 alto livello, riga 5 basso livello). Questa primitiva prende in input un indice a ed una variabile e restituisce in output l'i-esimo frammento costituito da 512 bit

della variabile. Per ogni macroblocco devono essere inizializzati 64 blocchi. Per effettuare questa operazione si può eseguire dentro al for principale un altro for che itera 64 volte (riga 5-9 alto livello, riga 4-17 basso livello). Ad ogni iterazione, se la variabile di iterazione è minore di 16, il blocco viene inizializzato come un frammento del suo macroblocco, operazione svolta dalla funzione primitiva takePart che, preso in input un indice i ed un variabile di 512 bit, restituisce l'i-esimo frammento di 32 bit della variabile (riga 7 alto livello, riga 10 basso livello). Alternativamente viene inizializzato con la funzione W(). L'inizializzazione dei blocchi non ha un costo energetico poiché le variabili vengono inizializzate in quel preciso momento e poiché l'esecuzione di W() non deve dissipare bit. Per poter rendere reversibile questo for innestato c'è bisogno di una parola di memoria che tenga conto delle iterazioni per ogni volta che viene chiamato. Questo for viene chiamato n volte, dove n è il numero di macroblocchi nel messaggio iniziale. Per rendere reversibile questo for, quindi, saranno necessarie n celle di memoria dove salvare le variabili di iterazione.

Il codice in c++ da noi reso reversibile funziona in maniera differente da come illustrato da questo studio. Le variabili dove sono salvati i blocchi vengono ad ogni iterazione sovrascritte dai blocchi successivi. Ciò ci obbliga a salvare in memoria il valore delle variabili prima che queste ultime vengano aggiornate (riga 14/15, 20/21 in C++). L'idea dietro questa implementazione differente è quella di risparmiare memoria utilizzata. Se nel nostro codice dobbiamo usare n*64 celle per salvare i valori dei vari blocchi, in questo codice ne vengono utilizzate solo 64. Purtroppo certe tecniche in un computer reversibile non possono essere applicate ed il numero di celle usate è lo stesso. Se ad ogni iterazione di n sappiamo di dover salvare 64 celle in memoria alla fine avremo bisogno dello stesso numero di celle come nel nostro codice ideale.

Implementazione basso livello

```
1  pop(i)
2  For( i > 0, i-=1):
3      pop(j)
4      For( j > 0, i-=1):
5          if(j < 16, secondIf):
6              block[i][j] -= W(j, block[i][])
7          else:
8              block[i][j] -= takePart(macroBlock[
                   j])
9      endFor
10 endFor
```

Implementazione in c++

```
1  lp--;
2  for (i += mem[lp], mem[lp] -= i; i > 0; i--) {
3      lp--;
4      for (j += mem[lp], mem[lp] -= j; j > 16; j
           --) {
5          w[j - 1] -= SHA256_F4(w[j - 2 - 1]) +
               w[j - 7 - 1] + SHA256_F3(w[j - 15
               - 1]) + w[j - 16 - 1];
6          pop(&(w[j - 1]));
7      }
8      for (; j > 0; j--) {
9
10         REV_SHA2_PACK32(&sub_block[j - 1 << 2],
               &w[j - 1]);
11         pop(&(w[j - 1]));
12     }
13     pp--;
14     sub_block = 0;
15     sub_block = pmem[pp];
16 }
```

Tabella 2.12: Confronto dell'inversione di codice della creazione di macroblocchi e blocchi

Per invertire l'assegnamento dei blocchi si deve prendere dallo stack il valore di iterazione ed iterare il for fino a quando i non sarà uguale a 0 (riga 1-10 reversibile basso livello). Stesso procedimento deve essere svolto nel for indentato (riga 3-9 reversibile basso livello). Per riportare i blocchi al valore precedente basterà invertire il procedimento fatto in precedenza, sottraendo il valore prima assegnato (riga 6/8 reversibile basso livello). Nel codice scritto in c++, visto che usiamo

un array per contenere tutti i blocchi, dovremo oltre a sottrarre il valore attuale, ricaricare il valore precedente nella variabile prendendolo dallo stack (riga 5-6, 10-11 reversibile C++).

Nella versione ottimale dell'algoritmo, salvando tutti i blocchi in memoria, possiamo dedurre che, per eseguire questo frammento di codice di SHA256 reversibilmente, abbiamo bisogno di n + 1 parole di memoria aggiuntive, per salvare le variabili di iterazione. Non abbiamo bisogno di dissipare bit in questa variante e quindi possiamo affermare che il consumo di energia equivale a 0.

MACRO SHA2_PACK3	1	`#define SHA2_PACK32(str, x) \`
	2	`{ \`
	3	`*(x) += ((uint32) *((str) + 3) ) \`
	4	`| ((uint32) *((str) + 2) << 8) \`
	5	`| ((uint32) *((str) + 1) << 16) \`
	6	`| ((uint32) *((str) + 0) << 24); \`
	7	`}`
MACRO REV_SHA2 _PACK32	1	`#define REV_SHA2_PACK32(str, x) \`
	2	`{ \`
	3	`*(x) -= ((uint32) *((str) + 3) ) \`
	4	`| ((uint32) *((str) + 2) << 8) \`
	5	`| ((uint32) *((str) + 1) << 16) \`
	6	`| ((uint32) *((str) + 0) << 24); \`
	7	`}`

Tabella 2.13: Confronto MACRO 2

Come detto prima la versione scritta in c++ differisce vagamente dalla versione scritta nel nostro pseudolinguaggio, in quanto ha bisogno di $64 * n + n + 1$ parole in memoria per essere resa reversibile.

2.4.4 Analisi creazione output reversibile

Dopo la creazione dei blocchi bisogna iniziare ad "assemblare" il messaggio finale. Questa è la parte più costosa dell'algoritmo, sia in termini di tempo che in termini di spazio.

```
1
2  for (i < len(newMessage) / 512, i+=1):
3
4     a = h0
5     b = h1
6     c = h2
7     d = h3
8     e = h4
9     f = h5
10    g = h6
11    h = h7
12
13    for (t < 64, t+=1):
14        t1 = h + sum1(e) + Ch(e, f, g) + K[i] + macro[i]
15        t2 = (_capsigma0(a) + _maj(a, b, c))
16        h = g
17        g = f
18        f = e
19        e = d + t1
20        d = c
21        c = b
22        b = a
23        a = t1 + t2
24    endfor
```

```
25
26      h0 += a
27      h1 += b
28      h2 += c
29      h3 += d
30      h4 += e
31      h5 += f
32      h6 += g
33      h7 += h
34
35  endfor
```

Il messaggio finale di SHA256 sarà la concatenazione delle variabili h0,...,h7. In questa parte dell'algoritmo si dovrà aggiornare queste variabili, prima modificando delle variabili ausiliarie, a,...,h, e poi andando ad incrementare h0,...,h7 con queste ultime.

Implementazione alto livello

```
1
2   for (i < len(newMessage) / 512, i+=1):
3
4     a = h0
5     b = h1
6     c = h2
7     d = h3
8     e = h4
9     f = h5
10    g = h6
11    h = h7
12
13  endfor
```

Implementazione basso livello

```
1
2  i = 0
3  blockNumber = len(macroBlocks)
4
5  for( i <= blockNumber , i+=1):
6
7      push(a)
8      a += h0
9
10     push(b)
11     b += h1
12
13     push(c)
14     c += h2
15
16     push(d)
17     d += h3
18
19     push(e)
20     e += h4
21
22     push(f)
23     f += h5
24
25     push(g)
26     g += h6
27
28     push(h)
29     h += h7
```

Implementazione in c++	
1	`for (j = 0; j < 8; j++) {`
2	`push(&(wv[j]));`
3	`wv[j] += m_h[j];`
4	`}`
5	`push(&j);`

Tabella 2.14: Confronto di codice dell'assegnamento delle variabili a,...,h

Come prima cosa vanno inizializzate le variabili a,...,h. Questo passaggio potrebbe essere risolto reversibilmente con un decremento ed un incremento in quanto le variabili vengono inizializzate in quel momento. Nonostante questo sappiamo che questa inizializzazione avverrà n volte, dove n è il numero totale di macroblocchi, trovandoci dentro un for, costringendoci quindi a salvare ogni volta il valore contenuto in queste variabili prima dell'assegnamento di un nuovo valore. Oltre a questo è noto che questo ciclo verrà eseguito n volte, costringendoci ad usare un ulteriore for, che ci costerà una parola di memoria aggiuntiva. Nella versione in c++ queste variabili sono poste dentro un array, permettendoci di eseguire un ciclo for per aggiornarle (riga 1-5 C++). Nonostante si risparmino linee di codice con questa tecnica dobbiamo salvare in memoria il valore della variabile di iterazione per rendere questo procedimento reversibile. Sempre nell'algoritmo scritto in c++ questo procedimento è eseguito nel for visto in precedenza alla suddivisione in macroblocchi, permettendoci di risparmiare 1 parola in memoria.

Implementazione basso livello

```
1
2  h -= h7
3  push(h)
4
5  g -= h6
6  push(g)
7
8  f -= h5
9  push(f)
10
11 e -= h4
12 push(e)
13
14 d -= h3
15 push(d)
16
17 c -= h2
18 push(c)
19
20 b -= h1
21 push(b)
22
23 a -= h0
24 pop(a)
```

Implementazione in c++	
1	`lp--;`
2	`for (j += mem[lp], mem[lp] -= j; j > 0; j--) {`
3	`wv[j - 1] -= m_h[j - 1];`
4	`pop(&(wv[j - 1]));`
5	`}`

Tabella 2.15: Confronto dell'inversione di codice dell'assegnamento delle varaibili a,...,h

Per rendere reversibile questo procedimento è necessario prendere dallo stack i valori salvati in precedenza ed assegnarli alle variabili, dopo aver decrementato il valore di questi ultimi fino a 0. Stesso procedimento viene fatto nel codice visto in c++, con l'unica differenza che dobbiamo anche prendere dallo stack la variabile di iterazione (riga 2 reversibile C++).

Sapendo che le variabili a,...h vengono riassegnate n volte, ossia il numero di macroblocchi, sappiamo che per implementare questo frammento di codice in maniera reversibile abbiamo bisogno di n*8 celle di memoria aggiuntive, poiché 8 sono le variabili che usiamo. Ragionamento simile avviene nel codice in c++, con la piccola eccezione di avere bisogno di una parola in memoria aggiuntiva per poter salvare la variabile di iterazione. Il costo aggiuntivo sarà quindi di $n * 9$ celle di memoria.

Dopo l'assegnazione delle variabili iniziali dobbiamo compiere dei procedimenti per cambiarle, per poi incrementare le variabili h0,...,h7 dei valori ottenuti.

Implementazione alto livello

```
1
2   for t in range(64):
3       t1 = h + sum1(e) + Ch(e, f, g) + K[i] + macro[i
            ]
4       t2 = (_capsigma0(a) + _maj(a, b, c))
5       h = g
6       g = f
7       f = e
8       e = d + t1
9       d = c
10      c = b
11      b = a
12      a = t1 + t2
```

Implementazione basso livello

```
1  j = 0
2
3  for( j < 64 , j+=1):
4
5      push(t1)
6      t1 += h + sum1(e) + Ch(e, f, g) + K[j] +
           block[i][j]
7
8      push(t2)
9      t2 += sum0(a) + maj(a, b, c))
10
11     push(h)
12     h += g
13
14     push(g)
15     g += f
16
17     push(f)
18     f += e
19
20     push(e)
21     e += (d + t1)
22
23     push(d)
24     d += c
25
26     push(c)
27     c += b
28
29     push(b)
30     b += a
31
32     push(a)
33     a += (t1 + t2)
34
35 endfor
36
37 push(j)
```

Implementazione in c++

```
1  for (j = 0; j < 64; j++) {
2
3      push(&(t1));
4      t1 += wv[7] + SHA256_F2(wv[4]) + SHA2_CH(wv
           [4], wv[5], wv[6])
5           + sha256_k[j] + w[j];
6
7      push(&(t2));
8      t2 += SHA256_F1(wv[0]) + SHA2_MAJ(wv[0], wv
           [1], wv[2]);
9
10     push(&(wv[7]));
11     wv[7] += wv[6];
12
13     wv[6] -= wv[7];
14     wv[6] += wv[5];
15
16     wv[5] -= wv[6];
17     wv[5] += wv[4];
18
19     wv[4] -= wv[5];
20     wv[4] += wv[3] + t1;
21
22     wv[3] -= wv[4] - t1;
23     wv[3] += wv[2];
24
25     wv[2] -= wv[3];
26     wv[2] += wv[1];
27
28     wv[1] -= wv[2];
29     wv[1] += wv[0];
30
31     wv[0] -= wv[1];
32     wv[0] += t1 + t2;
33
34 }
35
36 push(&(j));
```

Tabella 2.16: Confronto di codice del for indentato

Questa serie di procedimenti consistono in un for che itera 64 volte dove alle variabili a,..,h viene assegnato un nuovo valore ogni volta. Oltre a queste 8 variabili in questo procedimento ci avvaliamo anche di due variabili ausiliarie: t1 e t2. Come prima cosa si deve rendere reversibile il for. Come visto anche in precedenza per rendere reversibile un for basta salvarsi in memoria la variabile di iterazione (riga 37 basso livello, riga 36 C++). Dobbiamo tenere a mente che questa operazione verrà effettuata n volte, dove n è il numero di macroblocchi. Sapendo questo si può dedurre che si avrà bisogno di n parole di memoria totali per effettuare un for per ogni macroblocco. Dentro al for ad ogni variabile viene assegnato un nuovo valore. Essendo un assegnamento tutte le informazioni dentro quella variabile verrebbero cancellate. Ciò ci obbliga a salvare ogni singola variabile dentro allo stack per poter poi invertire il procedimento. L'assegnamento del valore alle variabili t1 e t2 (riga 5-9 basso livello,riga 3-8 C++) non ha costi aggiunti in spazio ed energia poiché queste operazioni non devono dissipare bit. Sapendo di dover salvare 10 variabili ad ogni iterazione, che le iterazioni sono 64 e che a loro volta il for viene iterato n volte: si può dedurre di aver bisogno di 640*n celle di memoria aggiuntive per poter rendere reversibile questo frammento di SHA256.

Implementazione basso livello

```
1
2  pop(j)
3
4  for( j > 0 , j -= 1):
5
6      a -= (t1 + t2)
7      pop(a)
8
9      b -= a
10     pop(b)
11
12     c -= b
13     pop(c)
14
15     d -= c
16     pop(d)
17
18     e -= (d + t1)
19     pop(e)
20
21     f -= e
22     pop(f)
23
24     g -= f
25     pop(g)
26
27     h -= g
28     pop(h)
29
30     t2 -= sum0(a) + maj(a, b, c))
31     pop(t2)
32
33     t1 -= h + sum1(e) + Ch(e, f, g) + K[j] +
           block[i][j]
34     pop(t1)
35
36 endfor
```

Implementazione in c++

```
1  lp--;
2  for (j += mem[lp], mem[lp] -= j; j > 0; j--) {
3
4      wv[0] -= t1 + t2;
5      pop(&(wv[0]));
6
7      wv[1] -= wv[0];
8      pop(&(wv[1]));
9
10     wv[2] -= wv[1];
11     pop(&(wv[2]));
12
13     wv[3] -= wv[2];
14     pop(&(wv[3]));
15
16     wv[4] -= wv[3] + t1;
17     pop(&(wv[4]));
18
19     wv[5] -= wv[4];
20     pop(&(wv[5]));
21
22     wv[6] -= wv[5];
23     pop(&(wv[6]));
24
25     wv[7] -= wv[6];
26     pop(&(wv[7]));
27
28     t2 -= SHA256_F1(wv[0]) + SHA2_MAJ(wv[0],
           wv[1], wv[2]);
29   pop(&(t2));
30
31     t1 -= wv[7] + SHA256_F2(wv[4]) + SHA2_CH(
           wv[4], wv[5], wv[6])
32          + sha256_k[j - 1] + w[j - 1];
33   pop(&(t1));
34
35 }
```

Tabella 2.17: Confronto dell'inversione di codice del for indentato

Per invertire il for visto in precedenza basterà riprendere il valore della variabile di iterazione vista in precedenza ed eseguire un for fino a quando la nuova variabile non avrà raggiunto il valore 0 (riga 2-4 reversibile basso livello, riga 1-2 reversibile C++). Dentro al for indentato successivamente dobbiamo riportare le variabili al valore precedente, prima decrementandole del loro valore attuale e successivamente incrementandole del valore salvato precedentemente nello stack.

La parte finale di questo frammento di codice consiste nell'incremento delle variabili h0,...,h7. Questa operazione è molto semplice e per essere resa reversibile basterà decrementare queste variabili della stessa quantità dell'incremento.

Implementazione alto livello	
1	h0 += a
2	h1 += b
3	h2 += c
4	h3 += d
5	h4 += e
6	h5 += f
7	h6 += g
8	h7 += h

Implementazione basso livello	1	h0 += a
	2	h1 += b
	3	h2 += c
	4	h3 += d
	5	h4 += e
	6	h5 += f
	7	h6 += g
	8	h7 += h
Implementazione in c++	1	for (j = 0; j < 8; j++) {
	2	m_h[j] += wv[j];
	3	}
	4	
	5	push(&j);

Tabella 2.18: Confronto di codice dell'aggiornamento delle variabili h0,...,h7

Nonostante questo, se decidessimo di risparmiare sulle righe di codice potremmo optare per la realizzazione di un for. Questo è quello che accede nel codice scritto in c++ (riga 1-5 C++).

Implementazione basso livello	1	h7 −= h
	2	h6 −= g
	3	h5 −= f
	4	h4 −= e
	5	h3 −= d
	6	h2 −= c
	7	h1 −= b
	8	h0 −= a
Implementazione in c++	1	lp−−;
	2	for (j += mem[lp], mem[lp] −= j; j > 0; j−−) {
	3	m_h[j − 1] −= wv[j − 1];
	4	}

Tabella 2.19: Confronto dell'inversione di codice dell'aggiornamento delle variabili

Se per invertire il codice di basso livello è sufficiente un decremento (riga 1-8 basso livello), per invertire il codice scritto in c++ abbiamo bisogno di estrarre il valore della variabile di iterazione salvato in precedenza e decrementarlo fino a che non arrivi a 0.

Sommando lo spazio richiesto per eseguire il primo for, 1 parola, quello per l'assegnamento delle variabili, 8*n parole, quello per l'esecuzione del for indentato, n parole di memoria, quello per l'esecuzione del codice dentro il for indentato, 64*10*n parole di memoria, e quello per l'aggiornamento delle variabili h0,...,h7, 0 parole di memoria, si può concludere che questo spezzone di codice è eseguibile reversibilmente con l'aggiunta di 64*19*n + 1 parole di memoria e con un costo di energia per la dissipazione di bit pari a 0.

2.4.5 Ottimizzazione del for che itera 64 volte

Ci sono delle piccole ottimizzazioni che potrebbero essere apportate a questo spezzone di codice. Guardando l'inizio del codice abbiamo visto che quando le variabili vengono inizializzate non sprecano energia e sono reversibili. Pertanto non sarebbe necessario, nella prima iterazione, salvare il loro valore nello stack. Per fare questo basta mettere un if che controlli che sia la prima iterazione, facendoci risparmiare così 10 blocchi di memoria. Altra accortezza che ridurrebbe di molto il costo in termini di spazio è la riduzione dell'assegnamento delle variabili. Dentro il for innestato possiamo notare come molti assegnamenti sono in realtà delle permutazioni. Ciò vuol dire che il valore di una variabile viene assegnato ad un altra, e non eliminato. Analizzando il codice, per esempio, possiamo vedere come a c venga assegnato il valore di b ed a b quello di a (riga 10-11 alto livello). Invece di salvarci i valori in memoria, quando dovremo tornare indietro, se vogliamo assegnare il vecchio valore di c, sapremo che questo valore si trova attualmente in b. Facendo questo è possibile arrivare a risparmiare fino a 7 parole di memoria per ogni singola iterazione. Facendo dei semplici calcoli possiamo vedere che con questa versione del codice le parole in memoria usate nella versione reversibile di SHA256 sono uguali a $(n-1)*8+64*3*n$. Siamo quindi riusciti a ridurre ad un terzo lo spazio in memoria richiesto.

```
1  a -= (t1 + t2)
2  a += b
3
4  b -= a
5  b += c
6
7  c -= b
8  c += d
9
10 d -= c
11 d += e
12
```

```
13  e -= (d + t1)
14  e += f
15
16  f -= e
17  f += g
18
19  g -= f
20  g += h
21
22  h -= g
23  push(h)
24
25  t2 -= sum0(a) + maj(a, b, c))
26  pop(t2)
27
28  t1 -= h + sum1(e) + Ch(e, f, g) + K[j] + block[i][j]
29  pop(t1)
```

2.5 Assemblaggio del messaggio

Implementazione alto livello	
1	
2	`finalMessage = h0 | h1 | h2 | h3 | h4 | h5 | h6 | h7`
3	
4	`return finalMessage`
5	
6	`unroll`

Implementazione basso livello

```
1
2  finalMessagge += h0 | h1 | h2 | h3 | h4 | h5 |
       h6 | h7
3
4  returnValue += finalMessage #valore ritornato
       alla fine dell'unroll
5
6  #Inizia la parte reverse del codice
7
8  finalMessage -= h0 | h1 | h2 | h3 | h4 | h5 |
       h6 | h7
```

Implementazione in c++

```
1  for (i = 0 ; i < 8; i++) {
2          SHA2_UNPACK32(m_h[i], &digest[i << 2]);
3      }
4  push(&j);
5
6  char buf[2*SHA256::DIGEST_SIZE+1];
7  buf[2*SHA256::DIGEST_SIZE] = 0;
8
9  for (int i = 0; i < SHA256::DIGEST_SIZE; i++)
10     sprintf(buf+i*2, "%02x", digest[i]);
11 //inizio unroll
12
13 lp--
14 for (i += mem[lp], mem[lp] -= i ; i > 0; i--) {
15     REV_SHA2_UNPACK32(m_h[i], &digest[i << 2]);
16 }
```

Tabella 2.20: Confronto di codice del ritorno del messaggio finale

L'ultima parte del codice prevede la concatenazione delle variabili h0,...,h7 per poter creare il messaggio finale che poi verrà restituito in output. Sappiamo che ognuna di queste variabili è composta da 32 bit e che il messaggio finale sarà una concatenazione di questi ultimi, e che quindi sarà composto da 256 bit. Questa parte di codice è reversibile in quanto, se assegnassimo questo valore ad una variabile non inizializzata, basterebbe semplicemente riportare il valore a 0 (riga 8 basso livello), ed in ogni caso dal messaggio finale sappiamo che i blocchi da 32 bit di cui è formato il digest sono le variabili h0,...,h7. Arrivati alla fine dell'algoritmo, ma non avendo ancora invertito l'intero codice, dobbiamo ricordarci di dover salvare il risultato in una cella di memoria apposita (riga 4 basso livello) per poi poterlo ritornare dopo che tutto il codice è stato eseguito all'incontrario L'assemblaggio del messaggio quindi è reversibile con nessun costo aggiuntivo in memoria ed un costo di energia pari a 0.

Nel codice scritto in c++ l'assemblaggio del messaggio viene lasciato eseguire ad una Macro. Questa macro incrementa semplicemente una variabile e per questo non ha costi aggiuntivi. Viene però usato un for per eseguire la macro su ogni variabile h0,..,h7.

Capitolo 3

Risultati

3.1 Costo Ideale

Per ottenere il costo aggiunto in spazio ed energetico dell'algoritmo basterà sommare i costi delle varie parti analizzate. Sapendo che il costo energetico di ogni spezzone di algoritmo è pari a 0 arriviamo alla conclusione che l'algoritmo è realizzabile in maniera reversibile. Dobbiamo comunque tener conto del costo aggiuntivo in memoria. Questo si otterrà in maniera analoga al calcolo del costo in energia, sommando i costi in spazio dei vari spezzoni di codice. Sapendo che la creazione del messaggio costa 1 parola di memoria, che la creazione dei macroblocchi e dei blocchi costa n + 1 parole in memoria, che la creazione dell'output ottimizzata costa $(n-1)*8+64*3*n$ e che l'assemblaggio del digest costa 0 parole aggiuntive in memoria: possiamo affermare che un algoritmo di SHA256 reversibile, con un hardware dedicato, sarebbe realizzabile con uno spazio ausiliario di $201 * n - 6$ parole di memoria e 0 energia consumata per dissipare bit.

3.1.1 Costo Implementazioni

Analizzando le implementazioni sviluppate possiamo ricavare, osservando lo stack apposito, la quantità di memoria necessaria per rendere reversbile l'algoritmo.

Implementazione Python

La versione di SHA256 implementata in Python [11] non è completamente reversibile ma può dare un'idea dell'andamento dello spazio aggiuntivo richiesto dall'algoritmo reversibile. La versione dell'algoritmo in questo linguaggio non presenta ottimizzazioni di sorta ed effettua un simil complete logging. In sostanza al minimo cambiamento le variabili vengono salvate in memoria per poi invertire il processo successivamente. Eseguendo l'algoritmo con vari input è facile constatare che l'andamento della memoria richiesta è lineare. Dopo aver eseguito varie volte l'algoritmo si evince che la versione senza ottimizzazioni ha bisogno di $656 * n + 2$ celle di memoria aggiuntive per essere eseguito, dove n è il numero di macroblocchi generati dal messaggio.

Implementazione c++

La versione di SHA256 implementata in c++ [12] presenta varie versioni: una che esegue un complete logging ed una che effettua delle ottimizzazioni. In entrambi i casi il costo dal punto di vista della memoria è lineare ma nel caso del codice ottimizzato il numero di celle necessarie per la versione reversibile dell'algoritmo diventa un terzo. Questo algoritmo per essere eseguito non presenta un solo stack ma ne presenta 3. Dentro il primo stack vengono immagazzinati gli interi, dentro al secondo byte e dentro al terzo indirizzi di memoria. Dopo aver eseguito varie volte l'algoritmo si evince che la versione senza ottimizzazioni ha bisogno di $720/astn + 2$ celle di memoria aggiuntive per essere eseguito, dove n è il numero di macroblocchi generati dal messaggio. Eseguendo la versione contenente le ottimizzazioni possiamo constatare che questo algoritmo ha bisogno di $273 * n + 2$ parole in memoria aggiuntive per essere reso reversibile.

3.1.2 Constatazioni

Per implementare un algoritmo reversibilmente dobbiamo pensare in una maniera diversa di implementare algoritmi. Tecniche di risparmio di memoria, come abbiamo visto per la suddivisione in blocchi, non sempre possono funzionare in un

programma reversibile. Queste tecniche puntano al riutilizzo di una variabile per salvare informazioni. Nella reversibilità però il riassegnamento di una variabile comporta l'eliminazione di informazione e quindi la necessità di dover salvare in una cella di memoria l'informazione dentro la variabile che si vuole aggiornare.

Conclusioni

In questo elaborato abbiamo analizzato la reversibilità ed il grande potenziale che possiede. Abbiamo visto come sia possibile prendere un algoritmo e renderlo reversibile. Per ora non esiste una tecnica univoca per rendere reversibile il codice e come abbiamo visto anche prima, con delle dovute migliorie, si riesce ad ottenere un codice che risparmia più spazio in memoria rispetto ad un altro. L'analisi del codice ha dato i risultati sperati. Siamo riusciti a determinare con certezza la possibile esistenza di un algoritmo di SHA256 reversibile ed a calcolarne il costo aggiuntivo in memoria rispetto all'algoritmo originale. Per quanto riguarda il codice scritto non credo che potrebbe trovare spazio attualmente nel mondo dei miners. Purtroppo quello del bitcoin è un mondo dove la velocità dei calcoli che si fanno è di cruciale importanza. Dovendo il nostro algoritmo "riavvolgersi" implica un raddoppio del tempo impiegato per terminare un'esecuzione. In un sistema dove anche i millisecondi possono determinare l'arrivo per primi al collegamento di un blocco purtroppo il codice qua sviluppato non riesce a trovare spazio. Nonostante al giorno d'oggi non possa essere utile un codice del genere, per via del principio di Landauer, un giorno, più o meno lontano, codici reversibili saranno la norma e spero che quello che abbiamo fatto in questo studio sarà usato più avanti come ispirazione per altri progetti. La reversibilità è un campo di studio molto vasto ed ancora inesplorato, ma grazie a questo studio spero di avere contribuito in minima parte anche io alla ricerca in questo ambito.

Sviluppi Futuri

Nell'algoritmo visto non abbiamo applicato tutte le ottimizzazioni possibili e, poiché quella mostrata non è l'unica implementazione realizzabile, in un futuro sarebbe interessante ampliare questa ricerca. Sarebbe stato possibile, infatti, salvare informazioni minori di una parola, come per esempio i bit cancellati da uno shift o il numero di iterazioni di un for, in spazi di memoria inferiori o accorpare queste informazioni in uno stesso registro. Sarebbe possibile, inoltre, sviluppare lo stesso algoritmo con diverse premesse, come per esempio hardware non dedicato o con una macchina con un hardware progettato solamente per eseguire questo processo, similmente alle macchine da mining. In futuro si potrebbero anche condurre studi simili a questo per analizzare algortimi diversi da SHA256 per poter esplorare le potenzialità della reversibilità.

Bibliografia

[1] Erik D. Demaine, Jayson Lynch, Geronimo J. Mirano, Nirvan Tyagi (2016) *Energy-Efficient Algorithms*. https://arxiv.org/abs/1605.08448

[2] Scott Aaronson, Daniel Grier, Luke Schaeffer (2015) *The Classification of Reversible Bit Operations*. https://arxiv.org/abs/1504.05155

[3] Michael P. Frank *Introduction to Reversible Computing: Motivation, Progress, and Challenges*, FAMU-FSU College of Engineering 2525 Pottsdamer.

[4] Satoshi Nakamoto *Bitcoin: A Peer-to-Peer Electronic Cash System*. https://bitcoin.org/bitcoin.pdf

[5] R. Landauer *Irreversibility and heat generation in the computing process*, IBM Journal.

[6] Sinan Küfeoğlua, Mahmut Özkuran *Bitcoin mining: A global review of energy and power demand* https://www.sciencedirect.com/science/article/abs/pii/S2214629619305948

[7] Michael Patrick Frank (1999) *Reversibility for efficient computing*, PhD thesis, Massachusetts Institute of Technology,.

[8] Christopher Lutz and Howard Derby (1982) *JANUS: A TIME-REVERSIBLE LANGUAGE* https://www.tetsuo.jp/ref/janus.pdf

[9] Claus Skou Nielsen and Michael Budde*Janus Playground* https://topps.diku.dk/pirc/janus-playground/

[10] *C++ SHA256 function* http://www.zedwood.com/article/cpp-sha256-function

[11] *Reversible SHA256 in Python* https://github.com/Niro-jpg/SHA256-PREV.git

[12] *Reversible SHA256 in C++* https://github.com/Niro-jpg/SHA256-FINAL.git