

SHADB

Matteo Grammatico – Fabio Virgili – Stefano Sammarco

Shadb è un programma che consente di archiviare i valori dei digest associati ai diversi file, in un file apposito, allo scopo di verificare se un dato file è già presente o meno nel proprio archivio.

Struttura e architettura del codice sviluppato

Il codice è suddiviso in moduli, e il tutto si divide ulteriormente in “file .h” che contengono i prototipi delle funzioni e la definizione delle strutture, e “file .c” dove le funzioni vengono implementate. Questo viene fatto per gestire meglio il tutto. Ecco la descrizione di tutti i moduli del programma:

-linked_list.h, linked_list.c: moduli per la gestione di una lista concatenata, in “linked_list.h” è definita la struttura che rappresenta il nodo della lista e inoltre sono presenti diversi prototipi di funzioni per la sua gestione, come la creazione di un nuovo nodo, o la ricerca di un dato all’interno della lista. In “linked_list.c” invece le funzioni sono definite. Sono state utilizzate librerie standard, oltre alle classiche “stdio.h” e “stdlib.h” è presente la libreria “stdbool.h” necessaria per il tipo di ritorno booleano della funzione *isEmpty*.

-sha1.h, sha1.c: moduli dedicati all’algoritmo SHA1, in “sha1.h” è presente la struttura fondamentale e sono presenti i prototipi delle funzioni utili per il calcolo del digest. In “sha1.c” le funzioni sono definite. Oltre alle classiche librerie “stdio.h” e “stdlib.h” sono state utilizzate altre librerie standard come “limits.h” per andare a definire limite massimo di un carattere, e “string.h” per utilizzare funzioni di copia e funzioni di calcolo lunghezza di stringhe.

Stesso discorso vale anche per i file “sha224.h”, “sha224.c”, “sha256.h”, “sha256.c”, “sha384.h”, “sha384.c”, “sha512.h”, “sha512.c”.

-sha1main: modulo principale per l’algoritmo SHA1, qui vengono utilizzate le funzioni e la struttura presente nei file prima descritti. Oltre alle classiche librerie “stdio.h” e “stdlib.h” sono state usate altre librerie standard come “limits.h” per stabilire limite massimo di un carattere con segno.

Stesso discorso vale anche per i file “sha224main.c”, “sha256main.c”, “sha384main.c”, “sha512main.c”.

-shadb.h, shadb.c: modulo principale dell’intero programma, qui le strutture e le funzioni create negli altri moduli vengono utilizzate. Oltre alle classiche librerie “stdio.h” e “stdlib.h” è presente anche “string.h” per utilizzare funzioni di comparazione tra stringhe.

-conf.h, conf.c: file di configurazione del programma. Non sono state usate ulteriori librerie oltre le classiche “stdio.h” e “stdlib.h”.

- [rwfile.h](#), [rwfile.c](#): rispettivamente prototipi e definizioni delle funzioni per gestire la scrittura e la lettura sui file in cui viene salvato il repository. Non sono state utilizzate ulteriori librerie oltre alle classiche “stdio.h” e “stdlib.h”.

Strutture dati implementate e funzionalità sviluppate

- [linked_list.h](#) contiene una struttura dati chiamata “Node”, che rappresenta il nodo della lista concatenata e al suo interno è presente:

- Unsigned int array, per la memorizzazione del digest.
- Due unsigned char, uno per la memorizzazione del path di un file, l'altro per la memorizzazione del tipo di sha utilizzato.
- Un puntatore a nuova struttura, fondamentale per una lista concatenata.

All'interno del file “linked_list.h” sono anche presenti diverse funzioni.

CreateNode() con cui un nodo viene creato. **Append()** serve ad aggiungere un nuovo nodo alla coda della lista concatenata. **Find()** è la funzione con cui si cerca un elemento all'interno della lista.

- [sha1.h](#) contiene una struttura dati chiamata “SHA1Config”, fondamentale per il calcolo del digest con l'algoritmo sha1. All'interno sono presenti:

- Due array di interi senza segno che lo sha usa per creare il digest.
- Cinque variabili unsigned int che immagazzinano i valori in esadecimale standardizzati dello sha e successivamente i valori degli hash.
- Tre variabili unsigned int usate per il calcolo vero e proprio del digest.
- Un intero per salvare la lunghezza attuale del messaggio.
- Un intero per salvare la lunghezza originale del messaggio.
- Un intero per salvare la lunghezza di blocchi del messaggio.

All'interno del file sono inoltre presenti diverse funzioni. **SHA1_read_file()** utile per la lettura del file che si vuole criptare con sha1. **SHA1_set()** con cui si prepara il tutto per iniziare il calcolo del digest. **sha1_exe()** è la funzione in cui avviene il calcolo vero e proprio del digest. **sha1_padding()** è una funzione che si occupa di una fase del calcolo del digest, infatti utilizzata all'interno della precedente funziona descritta.

- [sha224.h](#) contiene le medesime funzioni di “sha1.h”, per la loro descrizione fare riferimento alla descrizione di “sha1.h”.

- [sha256.h](#) contiene le medesime funzioni di “sha1.h”, per la loro descrizione fare riferimento alla descrizione di “sha1.h”.

- [sha384.h](#) contiene le medesime funzioni di “sha1.h”, per la loro descrizione fare riferimento alla descrizione di “sha1.h”.

- [sha512.h](#) contiene le medesime funzioni di “sha1.h”, per la loro descrizione fare riferimento alla descrizione di “sha1.h”.

- [shadb.c](#) sono i moduli principali per l’avvio e l’esecuzione del programma.

- [rwfile.h](#) contiene due funzioni. **ReadFile()** che consente di leggere il file che viene dato in input. **WriteToFile()** è la funzione con cui si scrive su file output.

- [conf.h](#) contiene due funzioni. **writeDigest()** che serve per scrivere il digest sul file output. **WriteListFile()** che serve per scrivere il path del file, il digest e l’algoritmo utilizzato, sul file output.

Implementazione funzionalità richieste

- [linked_list.c](#): vengono implementate le funzioni CreateNode(), Append(), Find(), Lenght(), PrintList().

- CreateNode() che prende come parametro un nodo. La funzione, dopo aver verificato il corretto inserimento dei parametri, assegna il nodo successivo tramite node->next = next;
- Append() che prende come parametro un nodo, la funzione utilizza un ciclo while che parte da un nodo e ne segue il next, per raggiungere l’ultimo nodo della lista. Una volta raggiunto crea un nuovo nodo e lo assegna al next del nodo in fondo.
- PrintList() che prende come parametro solo il nodo alla fine della lista e lo assegna ad una variabile per indicare il nodo corrente, dopo di che tramite un ciclo while scorre tutta la lista concatenata ed ad ogni iterazione stampa a video il path e il digest del nodo corrente.
- Find() che prende come input un tipo SHA1config e un nodo, apre il file di output in modalità lettura. Crea una variabile chiamata “line”, dove verranno immagazzinate le stringhe che verranno lette con il passaggio successivo, e una stringa dove immagazzinare il path del nodo. In seguito con un ciclo while, immagazzina in “line”, tramite fgets ogni riga del file di output e lo confronta con il path del nodo. Se trova un riscontro ritorna true, altrimenti ritorna false.

- [sha1.c](#): vengono implementate le varie funzioni per il calcolo del digest.

- SHA1_set() che inizializza le variabili di sha1 ai valori indicati dall’algoritmo, mettendo i valori di lunghezza, e numero di blocchi del messaggio a 0;

- `SHA1_read_file()` che prende come parametro una stringa per il path del file. Crea una stringa di grandezza massima chiamata `msg` e gli assegna tramite `malloc` un area di memoria. Apre il file del path tramite `fopen()` e ne legge il contenuto tramite un ciclo `while` e la funzione `fscan()`. Infine copia il contenuto del file nella variabile `msg` e la ritorna.

- `Sha1_padding()` che è la funzione che serve per aumentare la lunghezza del messaggio, prende come parametro il tipo `SHA1config` e un `unsigned char`. La funzione va a calcolare quanti bit aggiungere facendo modulo 64 della lunghezza del messaggio, se il risultato è più corto di 56 bytes (448 bits) lo sottrae a 56. Se il risultato è più grande di 56 lo sottrae a 120. In questo modo abbiamo il numero di bit da aggiungere sia per messaggi lunghi che già superano i 448 bit sia quelli corti che non li superano. Una volta ottenuto questo valore la funzione esegue un ciclo `while` dove ad ogni iterazione viene aggiunto un bit a 0 fino al raggiungimento di 448 bit.

Nel passaggio successivo la funzione va ad aggiungere un numero di 0 pari alla lunghezza originale del messaggio. Tramite un ciclo `for` vengono poi impostati 48 bit a 0. La lunghezza originale viene poi moltiplicata per 8 e divisa per il codice esadecimale `0x100` (256 in decimale), moltiplicata di nuovo per 8 ma questa volta al risultato viene effettuato il modulo per `0x100`. Alla fine viene poi messo il terminatore `/0`. Infine divide la lunghezza del messaggio per 64 in modo da avere blocchi di messaggi di 512 bit.

- `sha1_exe` che prende come parametro il tipo `SHA1config` e un `unsigned char` per il messaggio. La funzione prende la lunghezza del messaggio tramite `strlen()` e lo assegna alla lunghezza corrente e originale nel tipo `SHA1config`. Viene poi aggiunto `0x80` alla lunghezza per aggiungere un bit a 1 al messaggio. In seguito esegue la funzione `sha1_padding()` per aumentare la lunghezza del messaggio. Per ogni chunk da 512 bit ricavati dalla funzione `sha1_padding` vengono effettuati prima dei calcoli che porta ogni chunk in parole da 32 bit. Successivamente le 16 parole da 32 bit vengono estese a 80. A questo punto vengono inizializzate le variabili `a`, `b`, `c`, `d`, `e`, contenute le costanti `h0`, `h1`, `h2`, `h3`, `h4`, `h5`.

Seguendo l'algoritmo `sha1`, tramite un ciclo `for` vengono effettuati 80 round e in base al round vengono effettuate operazioni diverse di `and`, di `Or`, `Xor`, e rotazione, sulle variabili `a`, `b`, `c`, `d`, `e`.

Il valore di ogni variabile viene poi assegnata ad un'altra variabile prestabilita e viene riassegnata alle variabili `h`.

Alla fine le variabili `h0`, `h1`, `h2`, `h3`, `h4`, vengono concatenate ed assegnate ad una variabile `unsigned int` per essere ritornate dalla funzione.

-[sha1main.c](#): al suo interno è presente il codice per andare a richiamare le funzioni di "`sha1.c`".

Dopo aver settato il path del file e averlo letto tramite `SHA1_read_file()` va a vedere il suo path assoluto tramite `realpath()` e va a calcolarne il digest tramite `sha1_exe()`;

-[Sha256.c](#): implementa le funzioni per calcolo del digest e contiene al suo interno le variabili usate. E' composto da:

- Una variabile costante `unsigned int` contenente un array di 64 valori esadecimali.

- `Sha256_set()` che prende come parametro un tipo `SHA256` e inizializza le variabili `h[n]` come prestabilito dall'algoritmo e le variabili di lunghezza e numero di blocchi del messaggio a 0.

- `SHA256_read_file()` che prende come parametro una stringa per il path del file. Crea una stringa di grandezza massima chiamata `msg` e gli assegna tramite `malloc` un area di memoria. Apre il file del path tramite `fopen()` e ne legge il contenuto tramite un ciclo `while` e la funzione `fscan()`. Infine copia il contenuto del file nella variabile `msg` e la ritorna.

- `SHA256_pad()` che prende come parametro un tipo `SHA256` e un `unsigned char` per il messaggio. La funzione va a calcolare quanti bit aggiungere facendo modulo 64 della lunghezza del messaggio, se il risultato è più corto di 56 bytes (448 bits) lo sottrae a 56. Se il risultato è più grande di 56 lo sottrae a 120. In questo modo abbiamo il numero di bit da aggiungere sia per messaggi lunghi che già superano i 448 bit sia quelli corti che non li superano. Una volta ottenuto questo valore la funzione esegue un ciclo `while` dove ad ogni iterazione viene aggiunto un bit a 0 fino al raggiungimento di 448 bit.

Nel passaggio successivo la funzione va ad aggiungere un numero di 0 pari alla lunghezza originale del messaggio. Tramite un ciclo `for` vengono poi impostati 48 bit a 0. La lunghezza originale viene poi moltiplicata per 8 e divisa per il codice esadecimale `0x100` (256 in decimale), moltiplicata di nuovo per 8 ma questa volta al risultato viene effettuato il modulo per `0x100`.

Alla fine viene poi messo il terminatore `/0`.

Infine divide la lunghezza del messaggio per 64 in modo da avere blocchi di messaggi di 512 bit.

- `sha256_exe()` prende come parametro un tipo `SHA256` e un `unsigned char` per il messaggio; crea una variabile di tipo `unsigned char` dove salvare il messaggio denominato `mex`, gli alloca tramite `malloc` un'area di memoria e ci copia il messaggio passato come parametro. Aggiorna poi le variabili riguardanti la lunghezza attuale e originale del messaggio. Aggiunge `0x80` al messaggio, andando ad aggiungere un bit a 1 al messaggio, aggiornando lunghezza corrente e impostando il terminatore di stringa.

Aggiunge il padding tramite la funzione `sha256_pad()`.

Per ogni chunk da 512 bit ricavati dalla funzione di padding vengono effettuati prima dei calcoli che porta ogni chunk in parole da 32 bit.

Successivamente le 16 parole da 32 bit vengono estese a 62 bit.

A questo punto vengono inizializzate le variabili `a`, `b`, `c`, `d`, `e`, `f`, `g`, `h`, contenute le costanti `h[0]`, `h[1]`, `h[2]`, `h[3]`, `h[4]`, `h[5]`, `h[6]`, `h[7]`.

Dopodiché seguendo l'algoritmo `sha1`, tramite un ciclo `for`, vengono effettuati 64 round e in base al round vengono compiute operazioni diverse di `and` e di `or` sulle variabili `a`, `b`, `c`, `d`, `e`.

Il valore di ogni variabile viene poi assegnata ad un'altra variabile prestabilita e viene riassegnata alle variabili `h`.

Seguendo l'algoritmo `sha256`, tramite un ciclo `for` vengono effettuati 64 round e in base al round vengono effettuate operazioni diverse di `and`, di `Or`, `Xor`, `shift` e rotazione sulle variabili `a`, `b`, `c`, `d`, `e`, `f`, `g`, `h`, `hs`.

Il valore di ogni variabile viene poi assegnata ad un'altra variabile prestabilita e viene riassegnata alle variabili `h`.

Alla fine le variabili `h0`, `h1`, `h2`, `h3`, `h4`, `h5`, `h6`, `h7` vengono concatenate ed assegnate ad una variabile `unsigned int` per essere ritornate dalla funzione.

-[Sha256main.c](#): ha al suo interno il codice per andare a richiamare le funzioni di `sha256.c`. Dopo aver settato il path del file e averlo letto tramite `SHA256_read_file()` va a vedere il suo path assoluto tramite `realpath()` e va a calcolarne il digest tramite `sha256_exe()`.

-[Sha224.c](#): implementa le funzioni per calcolo del digest e contiene al suo interno le variabili usate. E' composto da:

- Una variabile costante `unsigned int` contenente un array di 64 valori esadecimali.
- `sha224_set()` che prende come parametro un tipo `SHA224`, e inizializza le variabili `h[n]` come prestabilito dall'algoritmo e imposta le variabili di lunghezza, e numero di blocchi del messaggio, a 0.
- `SHA224_read_file()` prende come parametro una stringa per il path del file. Crea una stringa di grandezza massima chiamata `"msg"` e gli assegna tramite `malloc` un'area di

memoria. Apre il file del path tramite `fopen()` e ne legge il contenuto tramite un ciclo `while` e la funzione `fscan()`. Infine copia il contenuto del file nella variabile "msg" e la ritorna.

- `SHA224_pad()` prende come parametro un tipo `SHA224` e un `unsigned char` per il messaggio. La funzione va a calcolare quanti bit aggiungere facendo modulo 64 della lunghezza del messaggio, se il risultato è più corto di 56 bytes (448 bits) lo sottrae a 56. Se il risultato è più grande di 56 lo sottrae a 120. In questo modo abbiamo il numero di bit da aggiungere sia per messaggi lunghi che già superano i 448 bit sia quelli corti che non li superano. Una volta ottenuto questo valore la funzione esegue un ciclo `while`, dove ad ogni iterazione viene aggiunto un bit a 0 fino al raggiungimento di 448 bit. Nel passaggio successivo la funzione va ad aggiungere un numero di 0 pari alla lunghezza originale del messaggio. Tramite un ciclo `for` vengono poi impostati 48 bit a 0. La lunghezza originale viene poi moltiplicata per 8 e divisa per il codice esadecimale `0x100` (256 in decimale), moltiplicata di nuovo per 8 ma questa volta al risultato viene effettuato il modulo per `0x100`. Alla fine viene poi messo il terminatore `/0`.

- `sha224_exe()` prende come parametro un tipo `SHA224` e un `unsigned char` per il messaggio. Crea una variabile di tipo `unsigned char`, per salvare il messaggio, chiamato `mex` e gli alloca tramite `malloc` un'area di memoria e ci copia il messaggio passato come parametro. Aggiorna poi le variabili riguardanti la lunghezza attuale e originale del messaggio. Aggiunge `0x80` al messaggio, andando ad aggiungere un bit a 1 al messaggio, aggiornando lunghezza corrente e impostando il terminatore di stringa. Aggiunge il padding tramite la funzione `sha224_pad()`. Per ogni chunk da 512 bit, ricavati dalla funzione di padding, vengono effettuati prima dei calcoli che porta ogni chunk in parole da 32 bit, successivamente le 16 parole da 32 bit vengono estese a 62 bit. A questo punto vengono inizializzate le variabili `a`, `b`, `c`, `d`, `e`, `f`, `g`, `hs`, contenente le costanti `h[0]`, `h[1]`, `h[2]`, `h[3]`, `h[4]`, `h[5]`, `h[6]`, `h[7]`. Seguendo l'algoritmo `sha1`, tramite un ciclo `for`, vengono effettuati 64 round e in base al round vengono effettuate operazioni diverse di `and` e di `or` sulle variabili `a`, `b`, `c`, `d`, `e`. Il valore di ogni variabile viene poi assegnata ad un'altra variabile prestabilita e viene riassegnata alle variabili `h`. Seguendo l'algoritmo `sha224`, tramite un ciclo `for` vengono effettuati 64 round e in base al round vengono effettuate operazioni diverse di `and`, `Or`, `Xor shift` e rotazione sulle variabili `a`, `b`, `c`, `d`, `e`, `f`, `g`, `hs`. Il valore di ogni variabile viene poi assegnata ad un'altra variabile prestabilita e viene riassegnata alle variabili `h`. Alla fine le variabili `h0`, `h1`, `h2`, `h3`, `h4`, `h5`, `h6`, vengono concatenate ed assegnate ad una variabile `unsigned int` per essere ritornate dalla funzione.

-Sha224main.C: ha al suo interno il codice per andare a richiamare le funzioni di `sha224.c`. Dopo aver settato il path del file e averlo letto tramite `SHA244_read_file()` va a vedere il suo path assoluto tramite `realpath()` e va a calcolarne il digest tramite `sha244_exe()`;

-Sha512.c: implementa le funzioni per calcolo del digest e contiene al suo interno le variabili usate. E' composto da:

- una variabile costante `unsigned int` contenente un array di 80 valori esadecimali.
- `sha512_set()` che prende come parametro un tipo `SHA512` e inizializza le variabili `h[n]` come prestabilito dall'algoritmo e le variabili di lunghezza e numero di blocchi del messaggio a 0.
- `SHA512_read_file()` che prende come parametro una stringa per il path del file. Crea una stringa di grandezza massima chiamata "msg" e gli assegna tramite `malloc` un'area di memoria. Apre il file del path tramite `fopen()` e ne legge il contenuto tramite un ciclo `while` e la funzione `fscan()`. Infine copia il contenuto del file nella variabile "msg" e la ritorna.

- `Sha512_pad()` che prende in input un tipo SHA512 e un unsigned char per il messaggio. La funzione va a calcolare quanti bit aggiungere facendo modulo 128 della lunghezza del messaggio, se il risultato è più corto di 112 bytes (896 bits) lo sottrae a 112. Se il risultato è più grande di 112 lo sottrae a 240. In questo modo abbiamo il numero di bit da aggiungere sia per messaggi lunghi che già superano i 1024 bit sia quelli corti che non li superano. Una volta ottenuto questo valore la funzione esegue un ciclo while dove ad ogni iterazione viene aggiunto un bit a 0 fino al raggiungimento di 1024 bit. Nel passaggio successivo la funzione va ad aggiungere un numero di 0 pari alla lunghezza originale del messaggio. Tramite un ciclo for vengono poi impostati 48 bit a 0. La lunghezza originale viene poi moltiplicata per 8 e divisa per il codice esadecimale 0x100 (256 in decimale), moltiplicata di nuovo per 8 ma questa volta al risultato viene effettuato il modulo per 0x100. Alla fine viene poi messo il terminatore /0. Infine divide la lunghezza del messaggio per 32 bit.

- `Sha512_exe()` prende come parametro un tipo SHA512 e un unsigned char per il messaggio. Crea una variabile di tipo unsigned char dove salvare il messaggio chiamato mex e gli alloca, tramite malloc, un area di memoria e ci copia il messaggio passato come parametro. Aggiorna poi le variabili riguardanti la lunghezza attuale e originale del messaggio. Aggiunge 0x80 al messaggio, andando ad aggiungere un bit a 1 al messaggio, aggiornando lunghezza corrente e impostando il terminatore di stringa. Aggiunge il padding tramite la funzione `sha512_pad()`. Per ogni chunk da 512 bit ricavati dalla funzione di padding vengono effettuati prima dei calcoli, che porta ogni chunk in parole da 64 bit, successivamente le parole da 64 bit vengono estese a 80. A questo punto vengono inizializzate le variabili a, b, c, d, e, f, g, hs, contenute le costanti h[0], h[1], h[2], h[3], h[4], h[5], h[6], h[7]. Seguendo l'algoritmo sha512, tramite un ciclo for vengono effettuati 64 round e in base al round vengono effettuate operazioni diverse di and e di or sulle variabili a, b, c, d, e. Il valore di ogni variabile viene poi assegnata ad un'altra variabile prestabilita e viene riassegnata alle variabili h. Seguendo l'algoritmo sha512, tramite un ciclo for vengono effettuati 80 round e in base al round vengono effettuate operazioni diverse di and, di Or, Xor shift e rotazione, sulle variabili a, b, c, d, e, f, g, hs. Il valore di ogni variabile viene poi assegnata ad un'altra variabile prestabilita e viene riassegnata alle variabili h. Alla fine le variabili h0, h1, h2, h3, h4, h5, h6, h7 vengono concatenate ed assegnate ad una variabile unsigned int per essere ritornate dalla funzione.

- **Sha512main.C**: ha al suo interno il codice per andare a richiamare le funzioni di sha512.c. ha al suo interno funzioneSHA512 che prende in input SHA512config e un char per il path del file.

Dopo aver settato il path del file e averlo letto tramite `SHA512_read_file()` va a vedere il suo path assoluto tramite `realpath()` e va a calcolarne il digest tramite `sha512_exe()`.

- **shadb.c**: che contiene solo la funzione `mainFunction()`

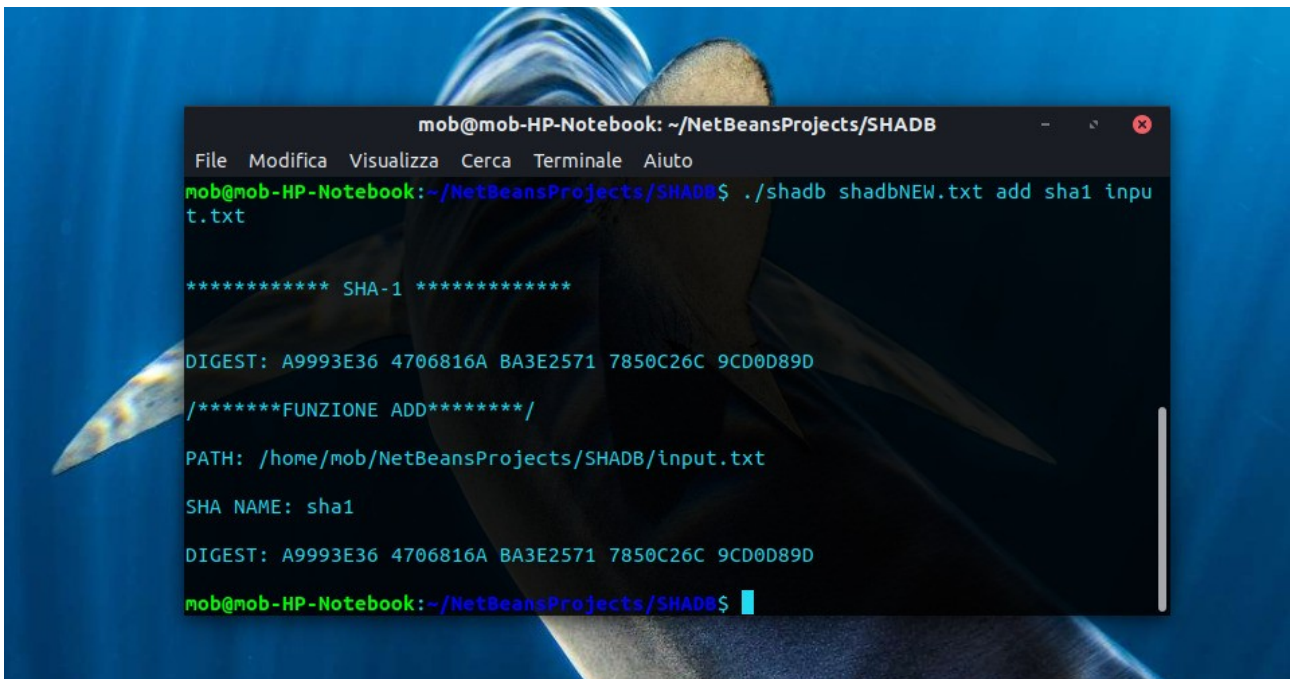
`MainFunction()` che inizializza le variabili per il funzionamento dello sha, oltre alle variabili per il path in input e del file di output. La funzione va a salvare gli argv specifici e calcola il path assoluto del file di input. In base al valore della variabile `s_name` va a richiamare la funzione relativa allo sha passato in `argv[3]`. In seguito crea un nuovo nodo a cui assegna tramite un ciclo for il relativo digest. In seguito gli assegna il tipo di sha usato e il path del file in input.

Successivamente va a vedere quale opzione viene passata in `argv[2]` tramite `strcmp()`, se viene passato “add” la funzione richiama `find()` per vedere se il nodo è già presente nel file di output e in caso negativo lo aggiunge tramite `append()`, in caso positivo viene ritornato un errore e stampato il nodo; se viene passato “find” la funzione richiama la funzione `find()` e presenterà un messaggio positivo o negativo in base al risultato.

Test per verificare il corretto funzionamento del codice

Sono stati effettuati dei test, per andare a verificare il corretto funzionamento del programma. Il primo test effettuato è stato su SHA1, dagli screenshot si può notare come il tutto funzioni in modo corretto.

Nella prima immagine si utilizza la funzione “add”, il file non era già presente all’interno dell’archivio, quindi è stato aggiunto correttamente.

A screenshot of a terminal window titled "mob@mob-HP-Notebook: ~/NetBeansProjects/SHADB". The terminal shows the execution of the command `./shadb shadbNEW.txt add sha1 input.txt`. The output displays the SHA-1 digest `A9993E36 4706816A BA3E2571 7850C26C 9CD0D89D` and confirms the addition of the entry to the database. The terminal also shows the file path `/home/mob/NetBeansProjects/SHADB/input.txt` and the SHA name `sha1`.

```
mob@mob-HP-Notebook: ~/NetBeansProjects/SHADB
File Modifica Visualizza Cerca Terminale Aiuto
mob@mob-HP-Notebook:~/NetBeansProjects/SHADB$ ./shadb shadbNEW.txt add sha1 input.txt

***** SHA-1 *****

DIGEST: A9993E36 4706816A BA3E2571 7850C26C 9CD0D89D

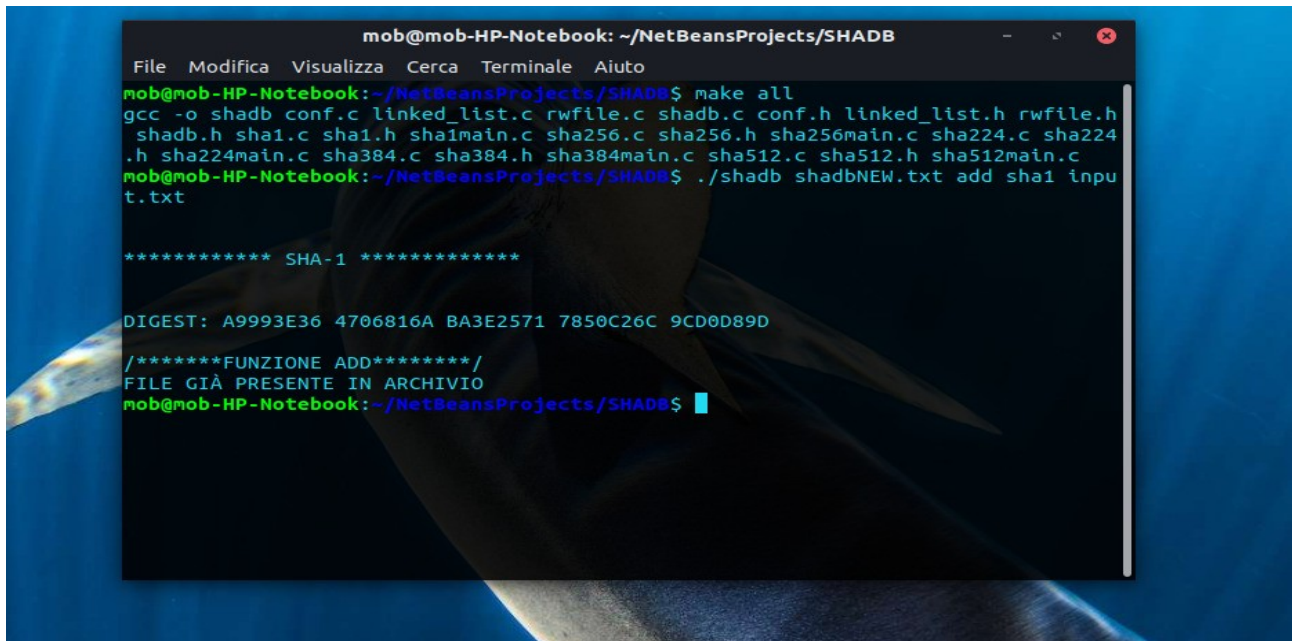
/*****FUNZIONE ADD*****/

PATH: /home/mob/NetBeansProjects/SHADB/input.txt

SHA NAME: sha1

DIGEST: A9993E36 4706816A BA3E2571 7850C26C 9CD0D89D
mob@mob-HP-Notebook:~/NetBeansProjects/SHADB$
```


Nella seconda immagine, invece, si prova ad utilizzare nuovamente la funzione “add”, questa volta però, abbiamo un messaggio di errore che dice “FILE GIÀ PRESENTE IN ARCHIVIO”. Infatti in archivio il file che stavamo cercando di aggiungere, era stato già aggiunto.



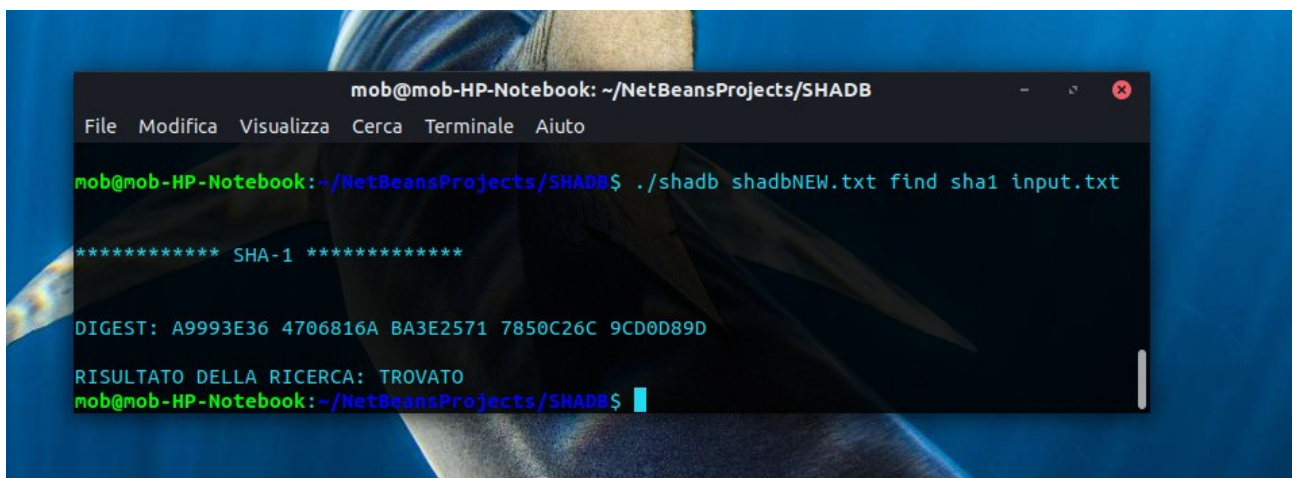
```
mob@mob-HP-Notebook: ~/NetBeansProjects/SHADB
File Modifica Visualizza Cerca Terminale Aiuto
mob@mob-HP-Notebook:~/NetBeansProjects/SHADB$ make all
gcc -o shadb conf.c linked_list.c rwfile.c shadb.c conf.h linked_list.h rwfile.h
shadb.h sha1.c sha1.h sha1main.c sha256.c sha256.h sha256main.c sha224.c sha224
.h sha224main.c sha384.c sha384.h sha384main.c sha512.c sha512.h sha512main.c
mob@mob-HP-Notebook:~/NetBeansProjects/SHADB$ ./shadb shadbNEW.txt add sha1 input
t.txt

***** SHA-1 *****

DIGEST: A9993E36 4706816A BA3E2571 7850C26C 9CD0D89D

/*****FUNZIONE ADD*****/
FILE GIÀ PRESENTE IN ARCHIVIO
mob@mob-HP-Notebook:~/NetBeansProjects/SHADB$
```

Nella terza immagine, abbiamo testato la funzione “find”. Come si può vedere il tutto funziona, infatti, il file era presente in archivio e quindi il messaggio è “TROVATO”.



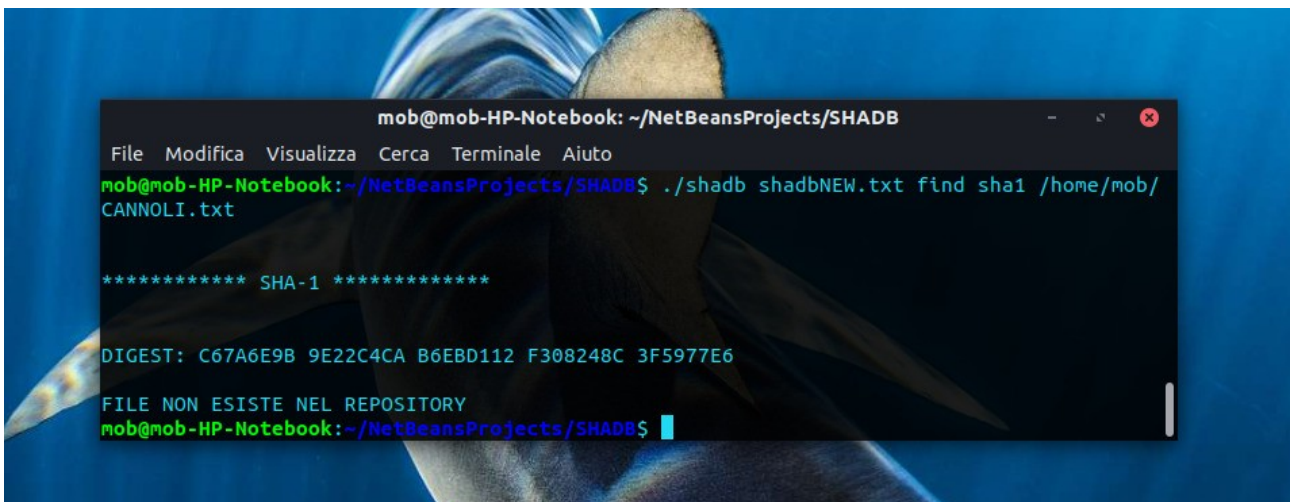
```
mob@mob-HP-Notebook: ~/NetBeansProjects/SHADB
File Modifica Visualizza Cerca Terminale Aiuto
mob@mob-HP-Notebook:~/NetBeansProjects/SHADB$ ./shadb shadbNEW.txt find sha1 input.txt

***** SHA-1 *****

DIGEST: A9993E36 4706816A BA3E2571 7850C26C 9CD0D89D

RISULTATO DELLA RICERCA: TROVATO
mob@mob-HP-Notebook:~/NetBeansProjects/SHADB$
```

Nella quarta immagine, invece, il file non era presente nell'archivio, allora il messaggio di errore ci dice "FILE NON ESISTE NEL REPOSITORY".

A screenshot of a terminal window titled "mob@mob-HP-Notebook: ~/NetBeansProjects/SHADB". The window has a menu bar with "File", "Modifica", "Visualizza", "Cerca", "Terminale", and "Aiuto". The terminal shows a command prompt where the user has entered `./shadb shadbNEW.txt find sha1 /home/mob/CANNOLI.txt`. The output displays the SHA-1 digest: `DIGEST: C67A6E9B 9E22C4CA B6EBD112 F308248C 3F5977E6`. Below this, an error message is shown: `FILE NON ESISTE NEL REPOSITORY`. The prompt then returns to `mob@mob-HP-Notebook:~/NetBeansProjects/SHADB$` with a cursor. The background of the terminal window shows a blurred image of a person's face.

```
mob@mob-HP-Notebook: ~/NetBeansProjects/SHADB
File  Modifica  Visualizza  Cerca  Terminale  Aiuto
mob@mob-HP-Notebook:~/NetBeansProjects/SHADB$ ./shadb shadbNEW.txt find sha1 /home/mob/
CANNOLI.txt

***** SHA-1 *****

DIGEST: C67A6E9B 9E22C4CA B6EBD112 F308248C 3F5977E6

FILE NON ESISTE NEL REPOSITORY
mob@mob-HP-Notebook:~/NetBeansProjects/SHADB$
```

Lo stesso tipo di test sono stati effettuati anche sui restanti algoritmi.