

# Relazione per il corso IN450

Lorenzo Pichetti

Luglio 2020

## 1 Introduzione

Il breve lavoro qui riportato consiste nell'implementazione in linguaggio C di parte degli algoritmi descritti nell'articolo "Improved Generic Algorithms for 3-Collisions" di Antoine Joux e Stefan Lucks. In particolare è stato da me implementato l'algoritmo standard per la ricerca di 3-collisioni su funzioni di hash generiche ed una delle nuove versioni proposte in cui vengono variati i possibili time-memory tradeoffs.

## 2 Osservazioni Preliminari

Siccome gli algoritmi presentati sono definiti per una generica funzione di Hash che opera sull'insieme  $[0, N]$ , ho scelto di considerare una famiglia di funzioni con byte di output variabili, definita a partire da SHA3-384. Inserendo infatti *2Byte* come lunghezza di output, il programma cercherà 3-collisioni sulla funzione che restituisce come risultato le prime 4 cifre esadecimali del relativo hash calcolato da SHA3-384.

Il calcolo degli hash viene eseguito utilizzando il toolkit OpenSSL che può essere scaricato dal seguente link: [www.openssl.org/source/](http://www.openssl.org/source/).

Va infine notato che entrambi i programmi fanno uso dell'istruzione C *system(...)*, il che potrebbe quindi creare incompatibilità con i sistemi operativi non Unix.

## 3 Descrizione dei Programmi

### 3.1 Cartella "Oracles"

Nella cartella *Oracles* è presente il programma *byteoracle*, che sostituisce il compito che nell'articolo originale era lasciato all'oracolo, ovvero quello di calcolare il valore della funzione di hash. Il programma riceve command line i parametri *n* e *string* dove con *n* viene indicato il numero di byte che si vuole ricevere in output, ed in *string* la stringa di cui si vuole calcolare l'hash.

Nella stessa cartella è anche presente il programma *oracle* che invece di prendere in input *n* prende un parametro *k* tra 1 e 4 al quale è abinato rispettivamente il calcolo di MD5, SHA1, SHA256 e SHA3-384; questo programma non è stato

però utilizzato all'interno degli algoritmi principali.

*Nota:* entrambi i programmi scrivono il loro output in *hash.txt* e, per loro conformazione, hanno bisogno di essere eseguito dalla loro cartella padre e non da quella in cui si trovano.

### 3.2 Implementazione tramite stringhe

In entrambi i programmi (*Algoritmo1* e *Algoritmo3*), tutti gli hash sono calcolati e salvati in memoria come stringhe esadecimali; il confronto tra hash è quindi un confronto tra stringhe, la generazione casuale è una generazione casuale di una stringa (con le dovute limitazioni) ed i vettori `Img[]`, `Pr1[]`, `Pr2[]`, `Start[]` e `End[]` sono vettori di puntatori a stringhe. Questa scelta permette di non avere limitazioni direttamente dovute al tipo (quali ad esempio la massima cifra identificabile dal tipo *int*) e di mantenere lo spazio di memoria contigua necessaria indipendente direttamente dalla lunghezza  $n$  selezionata.

## 4 Aspetti legati alla complessità computazionale

### 4.1 Ampliamento delle variabili $N_\alpha$ e $N_\beta$

Come indicato nel paragrafo 4 dell'articolo, per compensare le perdite dovute alle ripetizioni nella generazione pseudo-casuale, le costanti  $N_\alpha$  e  $N_\beta$  vengono moltiplicate per un fattore costante.

In entrambi gli algoritmi  $N_\beta$  viene moltiplicata per la costante 3, mentre  $N_\alpha$  viene moltiplicata per 8 nell'algoritmo 1 e per 3 nell'algoritmo 3.

### 4.2 Costanti legate alla complessità computazionale

Fedelmente a quanto descritto nell'articolo, entrambi gli algoritmi hanno complessità in tempo  $O(N^\beta)$  ed in spazio  $O(N^\alpha)$ , per le computazioni con numero di byte in uscita basso, però, lo schema riassuntivo finale del programma evidenzia il fatto che la computazione del primo step (quindi relativa a  $N_\alpha$ ), impiega più tempo rispetto quella del secondo; questo fatto è dovuto alla presenza di costanti moltiplicative in entrambi gli algoritmi più alte su  $N_\alpha$  che su  $N_\beta$ . Ovviamente questo squilibrio scomparirà all'aumentare dei byte considerati; va però purtroppo anche notato che, non essendo gli algoritmi parallelizzati, i tempi di computazione su singola CPU iniziano a diventare proibitivi già per  $n > 3$ .

## 5 Conclusioni

L'intuizione che sta alla base di questi nuovi algoritmi è quella di creare, nella prima fase, dei vettori che non contengano solo hash ed una preimmagine, ma già siano composti da 2-collisioni (quindi hash e due preimmagini). Utilizzando i metodi descritti nell'articolo, il secondo algoritmo riesce ad implementare quest intuizione ed utilizzare così un ammontare molto inferiore di memoria a parità di complessità computazionale in tempo. Vediamo qui riportate delle 3-collisioni su funzioni con hash di 24bit con i rispettivi byte di memoria utilizzati per la computazione.

Algoritmo 1			
3-Collisione			Memoria utilizzata
25e422	e04a9d	885bd4	1474560 byte
1e6595	5e67b2	3376e4	1474560 byte
773def	49b4a6	55ad37	1474560 byte
2ab4c1	c21d77	92fc94	1474560 byte
1077c1	773f33	234e54	1474560 byte
45de33	c99e88	b83005	1474560 byte
68957e	460b1a	c8f332	1474560 byte
a5de97	1cb4f7	66c12f	1474560 byte
ed9823	8f7a0b	781bd0	1474560 byte

Algoritmo 3			
3-Collisione			Memoria utilizzata
e6e7ec	5c6606	344dd2	14400 byte
489665	26226f	aefb78	14400 byte
e43340	e80acd	3eebd5	14400 byte
e951de	24b66e	3ced9e	14400 byte
d3753d	25cb4e	42de16	14400 byte
4e1b7b	263ddd	0467e5	14400 byte
0a260b	097aa7	ff0506	14400 byte
49f09e	7fad4a	b8bb1d	14400 byte
45889d	dcac3c	226c9b	14400 byte

Table 1: Tabelle riportanti la memoria utilizzata dai due differenti algoritmi  
*Nota:* per verificare l'effettiva collisione di questi input è sufficiente calcolare gli hash prodotti dalle tre stringhe con SHA3-384 e vedere che le prime 6 cifre di tutti questi coincidono