Relazione per il corso IN450

Lorenzo Pichetti

Luglio 2020

1 Introduzione

Il breve lavoro quì 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/.
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 dalla lunghezza n selezionata.

4 Aspetti legati alla complessità computazionale

4.1 Ampliamento delle variabili N_α e N_β

Come indicato nel paragrafo 4 dell'articolo, per compensare le perdite dovute alle ripetizioni nella generazione pseudo-casuale, le costanti N_α e N_β vengono moltiplicate per un fattore costante.
In entrambi gli algoritmi N_β viene moltiplicata per la costante 3, mentre N_α 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_α), impiega più tempo rispetto quella del secondo; questo fatto è dovuto alla presenza di costanti moltiplicative in entrambi gli algoritmi più alte su N_α che su N_β. Ovviamente questo squilibrio scomparirà all'aumentare dei byte considerati; va però putroppo 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 molta inferiore di memoria a parità di complessità computazionale in tempo. Vediamo quì 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

Algoritmo 3			
3-Collisione			Memoria utilizzata

Table 1: Tabelle riportanti la memoria utilizzata dai due differenti algoritmi